UNIX Code Migration Guide

UNIX Application Migration Guide

patterns & practices
proven practices for predictable results

Chapter 7: Creating the Development Environment

Larry Twork, Larry Mead, Bill Howison, JD Hicks, Lew Brodnax, Jim McMicking, Raju Sakthivel, David Holder, Jon Collins, Bill Loeffler
Microsoft Corporation

October 2002

Applies to:
Microsoft® Windows®
UNIX applications
Microsoft® Visual Studio®

Summary: Chapter 7: Creating the Development Environment discusses the various tools and development environments available for your migration strategy. The installation and configuration of Microsoft Visual Studio and Services for UNIX are covered as the core development environments as well as additional add-on capabilities. (62 printed pages)

Contents

Introduction

This chapter describes how to create a development environment for application migration. First, it describes some of the tools and facilities that are available. Then, in the three main sections, it describes how to:

- Configure the development environment.
- Populate with source code and build structures.
- Use the development environment.

The decisions made while setting migration strategy will help to define the development environment. This environment is either native Microsoft® Win32® application programming interface or Microsoft Interix.

Solutions in this chapter use the Microsoft Visual Studio® development system for Win32 source editing, for builds, and for makefile-based batch builds. Solutions use Microsoft Visual SourceSafe® version control software for source control. Interix solutions use the Microsoft Visual Studio for source editing, and the included Interix SDK tools for compiling, linking, and debugging (**gcc** and **gdb**).

Development Environment Overview

The development environment comprises all of the hardware and software that enables migration from UNIX to the Microsoft Windows® operating system, including:

- Software development kits (SDKs), which include compilers, editors, and debuggers, provide the basis for creating, building, and debugging the application.
- Integrated development environments (IDEs), such as Visual Studio, provide uniform and coherent access to tools otherwise provided by the SDK.
- 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.

These facilities are appropriate to either the developers or the testers of the system, as shown in Figure 1.

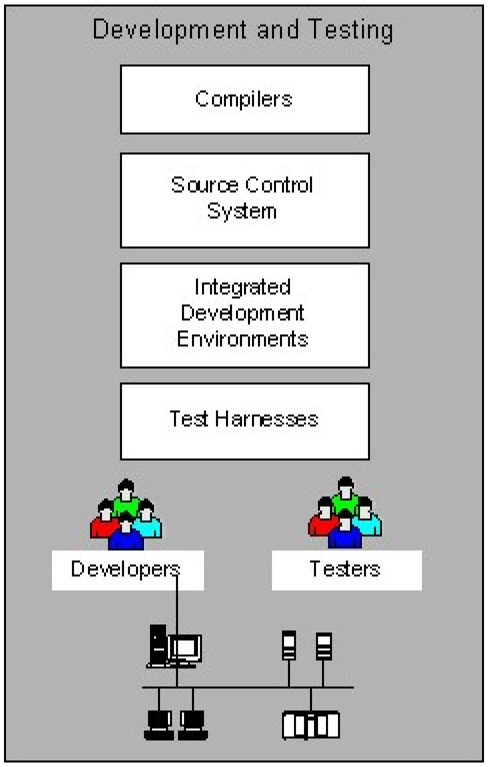

Figure 1. The development environment

Visual Studio is a comprehensive and powerful tool for developers. It includes features such as interactive IntelliSense for editing source, edit and continue debugging, integration with Visual SourceSafe for source control, and an extensive set of user customizations and help. These features can make Visual Studio seem overpowering, especially to UNIX developers who are comfortable with command-line utilities and with editors such as **vi**. However, Visual Studio also offers a set of short cuts and automation macros to streamline editing sessions.

The best way to overcome apprehension about the size and feature set of Visual Studio is to start using it. Developers will soon see how easy Visual Studio is to use, and how features that once seemed overwhelming can lead to greater productivity. Experienced developers will soon have the Visual Studio IDE customized to meet their needs and to eliminate any unneeded features.

Windows Platform SDK

An SDK contains tools to help build, debug, test, and deliver applications. It also contains all the definitions (include files) and libraries needed to compile programs. In general SDK tools are run from the command line, just like 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 dialogs.

The Microsoft Platform SDK is a set of tools and API definitions to help create applications for the Windows platform. It contains include files and documentation for all subsystems and APIs in Windows. It also contains definitions and documentation for the Microsoft Windows Server 2003 family, including Microsoft BizTalk® Server, Microsoft Commerce Server and Microsoft SQL Server. Installation of the Windows Server 2003 components is optional.

The Platform SDK is available on CD or as a free download on the Web. It is also available with the Microsoft Visual C++® development system and Visual Studio, or with an MSDN Professional or Universal subscription. To order or download the SDK, go to the Platform SDK Update site.

Application build tools

The Windows Platform SDK contains tools in several categories to help build applications, including:

- Cryptography, to create and manage certificates
- Microsoft DirectX® application programming interface to manage 3-D models, graphs, and DirectX devices
- Messaging API (MAPI), for message, address book, and other viewers
- The Multimedia for AVI editor
- OLE DB, for OLE object and file viewers and the MIDL compiler
- Resource files for image editor and resource compiler
- Telephony API (TAPI), for TAPI browsers
- Text Files tools, for identifying file differences
- Windows Sockets, for service provider ordering

Most migration projects rely heavily on the Text Files tools. *WinDiff*, for example, is a graphical tool for comparing files and directories, recursively if desired. Files with differences can be viewed with the different lines highlighted in colors for easy examination. 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 JScript® and Microsoft 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, 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 Windows Script Host to access operating system resources, such as files.

Debugging tools

The symbolic debuggers in the Platform SDK 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 are:

- NTSD for debugging user mode programs
- KD for kernel debugging
- WinDBG for debugging Windows-based programs, services, and kernel-mode drivers

The Platform SDK also contains Spy and Viewer programs that debug Windows objects:

- The Spy program (Spy.exe) helps to monitor messages sent to one or more windows and to examine the values of message parameters.
- The Object Viewer (Winobj.exe) examines all of the objects in the running system, such as kernel objects and file systems, as well as user-viewable properties of objects.

Performance monitoring and tuning tools

Some tools that help improve the performance of applications are included with both the Platform SDK and Visual Studio. This section discusses only these Platform SDK command-line tools:

- Call Attributed Profiler (CAP)
- Performance meter (Perfmtr)
- Pstat
- VADump
- Working Set Tuner (WST)

(For information on Windows-based tools included with both the SDK and Visual Studio, see "Performance Tuning" under "Integrated Development Environments.")

Call Attributed Profiler (CAP), a normal function call profiler, runs from the command line. CAP is similar to the UNIX prof profiler. CAP can show how much time is spent in each function and in other functions it calls.

To control the granularity of profiling, specify which DLLs and functions to include. CAP produces a call tree of all the functions called in the module being profiled, together with the time spent in each.

CAP supports the following measurement techniques:

- Measuring calls from an executable file (.exe)
- Measuring calls from a DLL
- Measuring calls from an .exe file to all of its DLLs
- Measuring calls from a DLL to all of its DLLs
- Any combination of the above

The **Perfmtr** (performance meter) command-line utility dynamically displays system performance information. The information is updated every 2-3 seconds. **Perfmtr** starts by displaying CPU usage data. However, single-letter keyboard commands prompt it to display performance information such as those shown in Table 1.

Table 1. Perfmtr keyboard commands

Key	Displays
C	CPU usage
V	Virtual memory usage
F	File cache usage
R	Cache Manager read and write operations
P	Pool usage
I	I/O usage
X	X86 VDM statistics
S	Server statistics
H	Repeats the current information type
Q	Quit the utility

Pstat displays process statistics as text. Command-line options instruct **Pstat** to display all statistics for all processes, for user-mode processes, or for kernel-mode processes. (If **Pstat** gets statistics for all processes, output will be at least several hundred lines.)

VADump examines the virtual address of a running process. Depending on the options specified, the output of VADump can include:

- Each address, along with its size, state, protection, and type
- Total committed memory for the executable
- Total committed memory for each DLL
- Total mapped committed memory
- Total private committed memory
- Total reserved memory
- Information about working set
- Information about paged and non-paged pool usage

> **Note** Because VADump can generate a very large amount of output, it is advisable to redirect the output to a file.

The Working Set Tuner (WST) reduces the memory a program uses. WST can help a program coexist efficiently with other programs and can help reduce the time it takes to load an application.

The working set of a program consists of those pages from the virtual address space that are mapped into active physical memory for the process. It includes both private data and shared data (program code including executable files, DLLs, and system DLLs.)

Integrated Development Environments

Unlike the SDK command-line tools, integrated development environments offer the user an integrated set of tools such as editors, compilers, and debuggers. Microsoft Visual Studio, the IDE discussed here, comes with a choice of languages such as C/C++ and Visual Basic, compilers and debuggers, and a large set of integrated tools that do everything from create resources to package applications for installation.

This chapter uses Visual Studio 6.0 as the basis for features and examples. The newest version of Visual Studio is Visual Studio .NET. This version includes the new features of the Microsoft .NET Framework and the common language runtime. Visual Studio .NET is covered in greater detail in Chapter 15.

This section discusses the Visual Studio features that provide compiling and linking, debugging, and performance tuning. It also includes a section that discusses third-party extensions to Visual Studio.

Compiling and linking

Visual Studio provides a state-of-the-art C/C++ optimizing compiler and preprocessor. Users can specify options to the compiler by using the GUI dialog boxes, or from the command line when using an external **make** procedure. As in UNIX, link options to the compiler are passed on to the linker.

Options to the compiler can include:

- Optimization options turn optimizations on or off.
- Code generation options identify capabilities, such as structured exception handling (SEH) model

and processor.

- Output Files options identify names of output files, such as executable files and output files for debugging information.
- Debugging options set preferences for debugging information output.
- Preprocessor options set, for example, location of files and defined constants.
- Language options enable or disable language extensions.
- Linking options include DLL output, threading model, and others.
- Precompiled header options set compilation speed.
- Miscellaneous options include Help, compile without linking, and others.

Debugging

Each built-in Visual Studio debugger for C/C++ and Visual Basic has its own window in the Visual Studio IDE. Many of these windows can be docked, that is, attached to a side of the main IDE window, or they may float over the IDE window.

Performance tuning

Visual Studio includes tools for analyzing the performance of applications, including:

- Pview
- Profiler
- Spy++

The PView process viewer uses dialog boxes to view and modify running processes and their threads.

PView can monitor:

- Memory usage of a process, of threads, and of individual DLLs
- CPU time used by processes and threads
- How an application or the system runs with different system priorities

  Note PView features provide powerful tools for a developer to monitor an application's processes and threads at different priorities. Changing priorities may also affect other applications running on the same system. Developers need to take this into account when using PView.

Profiler helps to analyze program execution either at the function level or at the level of individual lines of source code. Profiler can help determine which parts of code take a long time to execute and which parts are executed many times.

Profiler can be used:

- To count how many times a function is called. If the test data shows that a function is called too many or too few times, the application code could be in error.
- To determine the amount of time used by a function. If a function takes too long to execute, inefficient algorithms could be to blame.
- To determine coverage of testing procedures by verifying that all functions are called or that they are called the correct number of times.

Spy++ (Spyxx.exe) shows a graphical view of the system's processes, threads, windows, and window messages. Spy++ has a toolbar, hyperlinks for fast navigation, a Window Finder tool for easily selecting a window, and a font dialog box for customizing view windows. Spy++ also saves and restores user preferences.

Third-party extensions to Visual Studio

Third-party tools that integrate with Microsoft Visual Studio to enable detection of run-time errors include BoundsChecker from Compuware and PurifyPlus from Rational 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 in addition to 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 6.0 to support the UNIX **lex** and **yacc** utilities. The **lex** utility is a scanning utility that analyzes input text files for the occurrence of pre-defined text patterns. When the text pattern is found, a set of C source code is run. The **lex** utility reads its input from a specified input file, or from standard in. The input is in the form of pairs, with the text expression to scan for first, followed by the C code to execute when the text expression is found. The **lex** utility produces a C source file that is compiled to create the scanning program.

The **yacc** (yet another compiler compiler) utility is a parser that takes a setup input token and applies a set of grammar rules to the token. The **yacc** utility generates a C program based on the input of grammar rules from an input file or from standard in. The C program parses an input based on the grammar rules supplied when **yacc** was run.

The combination of **lex** and **yacc** provide developers with a powerful method for generating compilers, front-end processors for compilers, text processors and language translators. The **lex** utility performs the function of creating token based on regular expressions; yacc perform grammar rules on the tokens.

The MKS **lex** and **yacc** utilities work as add-ins to Visual Studio 6.0 that allow developers to create scanners and parsers within the Visual Studio 6.0 IDE. These help automate the time-consuming and monotonous tasks of lexical analysis and grammar matching from within Visual Studio 6.0.

Source Code Management Tools

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

Visual SourceSafe, the source management tool included with the Visual Studio IDE, creates projects in a tree view structure. Visual SourceSafe provides full source control for check in, check out, and difference tracking. Visual SourceSafe uses COM Automation for customization.

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

Visual SourceSafe, like other source code control systems, maintains a library of projects from which users check out files for exclusive or nonexclusive access. After modification and testing, users check in the files. If nonexclusive checkout is allowed, the files are merged at check-in time, interactively if necessary. Each incremental checked-in version can be recalled if functionality is inadvertently deleted. VSS can also create releases by labeling the files.

Visual SourceSafe handles text as well as binary files.

Packaging Tools

Once an application is created, the developer needs a way to effectively distribute the application to the target users. This needs to be done in a reliable and repeatable manner. The Microsoft Installer (MSI) technology provides this capability.

MSI allows the developer to generate an installation package that organizes an application's setup into a set of features that can be installed through the command line, or through the Control Panel, by using the Add or Remove Programs options. In addition to installation, MSI provides uninstall functionality, automatic application repair, and installation with elevated privileges.

MSI creates installation packages through a set of programming APIs and a package installation database. A developer can work with MSI by using the API directory. However, a tool that generates MSI scripts can help automate the process. The Microsoft Platform SDK provides such a tool—Orca.exe. Orca allows the developer to directly edit the content of the MSI database. Detailed information about Orca can be found on the Platform SDK Online Help or the Microsoft MSDN Web site Platform SDK: Windows Installer pages.

Third parties such as Wise Solutions and InstallShield provide products that further automate the creation of MSI installation packages. These tools have options that allow a developer to create a "before" snapshot of a target system. Then the developer can manually install the package. Now the developer can take an "after" snapshot of the system. The difference becomes the MSI package.

For more detailed information on MSI please visit the Microsoft TechNet Web site. The current page with MSI content as of creation of this guide is Using Windows Installer.

Cross-Platform Packages and Libraries

Every migration of an application from UNIX to Windows requires some interoperability between the UNIX development environment and the Windows development environment.

A rewrite to Win32 requires the least interoperation between the two environments. In this case, the code simply moves to a native Windows environment, with no further need for the UNIX development environment. At the other end of the scale, when a ported application requires ongoing cross-platform development, UNIX and Windows development environments must be well integrated.

Third-party cross-platform environments or libraries are popular solutions for maintaining code on multiple versions of UNIX and Windows operating systems. Because a layer of abstraction isolates the differences in the platforms in this case, users can develop applications without as much attention to operating system-specific code.

A given third-party library or toolset may not provide complete coverage for all aspects of program design. For example, one library could be designed to create an abstraction for operating system functionality such as processes and threads. Another library could address graphical user interface functionality.

For this reason, the functionality applications need is a major factor in choosing a vendor. The level of the interoperability required between the UNIX and Windows environments influences vendor choice. For more information on interoperability, see Chapter 6, UNIX and Windows Interoperability.

Interix

Interix is a multiuser UNIX environment that operates on computers running Windows. The Interix subsystem and its accompanying utilities provide an environment that resembles any other UNIX system. It includes case-sensitive file names, job control, compilation tools, and the use of more than 300 UNIX commands, utilities, and shell scripts. Because the Interix subsystem is layered on top of the Windows kernel, it offers true UNIX functionality without any emulation. (For more information about Interix, see Chapter 4, "Assessment and Analysis.")

This section discusses Interix APIs, Interix utilities, and Interix networking and security issues.

Interix APIs

Interix includes a Software Development Kit (the Interix SDK) that provides a full set of APIs (more than 1900) plus header and library files for creating and migrating UNIX applications.

The Interix SDK includes documentation and header files for the following categories of APIs and services:

- POSIX.1 APIs
- Cryptography
- User interface services, including curses and terminal routines and X-Windows
- Database (dbm)
- Remote procedure calls (RPCs)
- Sockets
- Memory-mapped files
- System V interprocess communication (IPC) mechanisms
- Berkeley Software Distribution (BSD) string and memory functions
- Pseudo terminals
- Controlling terminals
- Security
- Setup and system administration
- Tools and scripting

Interix utilities

The Interix environment includes both the Korn and C shells and provides more than 300 UNIX utilities. In addition, Interix includes the usual UNIX developments tools, such as text editors, **make** utilities, compilers, linkers, debuggers, and shell scripts.

Table 2 shows some widely used development tools included with Interix.

Table 2. UNIX-like development tools included with Interix

Category	Tools
Text editors and processors	Text editors: **ex**, **vi**, **view** File-pattern searchers: **grep**, **egrep**, **fgrep** Stream editor: **sed**
Archives, compression, and source management tools	To create and maintain library archives: **ar** To display file or directory differences: **diff**, **sdiff** To compress or expand files: **gzip**, **gunzip**, **uncompress**, **zcat** To create new revision control system (RCS) files or to change attributes of existing files: **rcs** To create archive files: **tar**
Configuration, management and build process tools	To translate Awk to Perl: **a2p** Pattern-directed scanning and processing language: **awk/gawk** C preprocessor interface to the make utility: **imake** Fast lexical analyzer generator: **lex** To maintain program dependencies (Berkeley make): **make** To create dependencies in makefiles: **makedepend** Practical Extraction and Report Language (Perl): **perl** For pipe fitting; to duplicate standard input and help when recording and watching the build process at the same time: **tee** To construct argument list(s) and execute: **xargs** To call imake with appropriate configuration file: **xmkmf** LALR parser generator: **yacc**
Compilers, linkers	Portable GNU (not UNIX) assembler: **as** Compiler interface: **c89**, **cc** (The c89 utility is an interface to the Microsoft Visual C/C++ compiler.) GNU project C and C++ Compiler: **gcc**, **g++** GNU project Fortran Compiler (v0.5.21): **g77** GNU assembler macro preprocessor: **gasp** GNU linker: **ld**
Debuggers and debugging assistance	GNU debugger: **gdb** ASCII, decimal, hexadecimal, octal dump: **hexdump** To display information from object files: **objdump** To list symbolic names stored in an object file: **nm** To trace system calls and signals: **truss** To display or set process limits: **ulimit** To print contents of X-Windows events: **xev**
Miscellaneous tools	To run a Win32 command (such as during migration of scripts): **runwin32**

Terminal emulator for X-Windows (requires third-party X-Windows server): **xterm**
UNIX to Windows text formatting converter: **flip**

In addition, the Interix /usr/contrib/win32/bin directory includes shell script wrappers for common Windows command-line utilities, such as **net**, **calc**, **cacls**, and **netstat**. Because this directory is automatically placed in the PATH variable when an Interix shell starts, these Windows utilities can be executed by typing the command name at the shell prompt. For example, to run the Windows calculator in the Korn shell, type **calc** at the dollar sign (**$**) prompt. To look for Windows commands in other directories, set the PATH_WINDOWS variable in the startup files.

Interix networking and security

This section covers some of the networking and security differences between UNIX and Interix. In particular it covers:

- Mapping network drives
- Network resource security
- User name differences
- Daemons and services

The /net virtual directory is similar to the Windows Universal Naming Convention (UNC). For example, the file referred to as \\Saturn\marketing\report.xls using Windows UNC can be referred to as /net/Saturn/marketing/report.xls using Interix.

To make resource ownership information consistent across computers, Interix uses User Name Mapping (when available) to associate Windows users with user identifiers (UIDs) and group identifiers (GIDs). If there is no User Name Mapping server available, the Interix subsystem computes a UID or GID as it encounters each Windows user or group. This computed value is normally unique, and all hosts in the same domain compute the same UID or GID for the user or group. If Interix cannot compute a unique UID or GID for the user or group, it makes a log entry in the Windows-based application's event log. If Interix can no longer communicate with a User Name Mapping server, the Interix subsystem reverts from mapped UIDs and GIDs to computed UIDs and GIDs.

The Interix subsystem principal domain is designed to reduce the length of the displayed user and group names. The principal domain is separate from the system's primary domain. The *primary domain* for a system is the domain to which the system is joined; that is, the domain displayed in System Properties. The Interix *principal domain* is set by default to the system primary domain, but that can be altered. For example, when the primary domain for the system is WESTCORP and users are also in the WESTCORP domain, the Interix principal domain is WESTCORP by default. A **who** command displays a user named Carl in the WESTCORP domain as "Carl." A user from another domain, such as Susannah in EASTCORP, is displayed as EASTCORP+Susannah.

UNIX system services are handled by server programs called *daemons*. Windows server-level programs are called *services*. Unlike a daemon, a service logs on to the computer by using a user account. This gives the Windows administrator greater control over the privileges granted to the service and, when the service logs on with a domain account, even allows the service to access network resources.

Interix can take advantage of both Windows and UNIX mechanisms to provide services such as **inetd** (the UNIX super server for Internet services). By default, Interix runs such services as traditional UNIX daemons, using the **init** utility to start and stop these daemons. Similar to a traditional UNIX system, Interix runs **init** as part of its startup procedure. When it starts, **init** executes scripts referenced by symbolic links located in /etc/rc2.d. Then, **init** continues to run until it receives a termination signal, at which point it runs shutdown scripts for the daemons it started.

Rogue Wave SourcePro

Rogue Wave SourcePro consists of a set of libraries that enable cross-platform compatibility of applications between UNIX and Windows. Its C++ components handle many of the intricacies of the C++ language and provide high-level object-oriented interfaces to complex underlying APIs. With minimal code changes, applications built by using SourcePro Core run on multiple operating systems, including Windows, and Linux and other popular UNIX operating systems.

Rogue Wave SourcePro consists of libraries for core functionality and for database and networking support.

SourcePro core

The Rogue Wave SourcePro core library consists of five modules:

- The Standard C++ Library Module is a complete implementation of the ISO/ANSI Standard for the C++ Programming Language. Sun, Hewlett-Packard, and Compaq ship Rogue Wave's implementation of the ANSI Standard C++ Library with their C++ compilers.
- The Essential Tools Module offers an internationalized set of fundamental C++ components that are useful in almost any type of C++ application. They provide user interfaces to the underlying ANSI Standard C++ Library and features beyond those in the ANSI standard, including classes for handling dates and times. Developers can use this module to write a single C++ application that can ship to any country.
- The Advanced Tools Module offers a mechanism for complex stream transformations and object serializations, which can stream C++ classes with minimal code changes.
- The XML Streams Module builds on the Advanced Tools Module, enabling C++ data to be written to or read from an XML stream. Because instances of C++ classes can be turned into XML without writing XML streaming code, developers can integrate existing C++ code into systems that use XML to communicate.
- The Threads Module provides a higher level, object-oriented API that hides many of the complexities of multithreading in C++; that is, developers are insulated from dealing with the native C threading library implementations. The Threads Module classes handle fundamental tasks such as creating, manipulating, synchronizing, and deleting threads. Advanced threading functionality includes server pools and producer/consumer queues.

SourcePro networking, web, and security

Developers can create secure or nonsecure networked and Internet-enabled applications by using Rogue Wave SourcePro Net to handle the details of socket programming and Internet protocols. In SourcePro Net, developers can implement standard Simple Object Access Protocol (SOAP) concepts using their existing C++ knowledge and XML. Because SourcePro Net has a layered architecture, developers can choose the level of abstraction for an application, such as the abstract layer for ease of use, or the protocol layer for finer control of details.

SourcePro Net includes four modules:

- The Essential Networking Module includes a high-level API for networking applications (it encapsulates the details of socket programming).
- The Internet Protocols Module provides an API for developing Internet-enabled client-side applications (it encapsulates the details of key Internet protocols).
- The Secure Communication Module provides an API for developing secure networked and Internet-enabled applications (it encapsulates the details of the low-level HTTPS protocol). This module also offers a high-level C++ API to Secure Socket Layer/Transport Layer Security [SSL/TLS] implementations, including RSA's commercial product BSAFE SSL-C and the open source alternative OpenSSL.
- The Web Services Module encapsulates the current SOAP specifications, including SOAP 1. 2, SOAP with attachments, and XML Schema 2001. By using the Web Services Module, developers can convert SOAP objects to and from strings, enabling the objects to work with any networking or deployment technology.

SourcePro database support

Rogue Wave SourcePro DB provides relational database access in C++. The SourcePro DB object-oriented interface abstracts the complexity of writing database applications, but still allows access to the native database client libraries when needed. SourcePro DB encapsulates the ANSI SQL 92 standard and supplies a consistent, high-level C++ interface to relational databases. Its layered architecture comprises a database-independent interface module and a variety of database-specific access modules, including Oracle, Oracle 8, Sybase, DB2, Informix and Microsoft SQL Server, as well as general access by using ODBC. The object-oriented encapsulation of relational database concepts eliminates the need to generate SQL programmatically.

The OpenSQL API helps to improve performance on a specific database by providing lower-level access that uses a statement-based architecture. This allows the creation of SQL, binds the variables to the statement, and executes the statement. By using SourcePro DB, developers can also get direct access to native structures and functions and make database-specific implementation choices.

Cross-platform GUI tools

By using cross-platform graphical user interface (GUI) tools, developers can write applications with an API consistent with both UNIX and Windows. The cross-platform tools usually create an abstraction through which the application can use the X-Windows system on UNIX and the Windows API on the Microsoft platform.

Some cross-platform GUI tools come from an independent software vendor (ISV) and others are open source. Even some open source versions come from a vendor with a support option. Examples of these cross-platform tools are the Carnac 2D graphics toolkit from INT, Inc., and the Qt toolkit from Trolltech.

INT Carnac

Carnac, a cross-platform interactive display system, is a C++ library available on several UNIX systems as well as on the Windows NT® and Windows 2000 operating systems.

Carnac is designed to handle applications that have high-end graphics requirements. For optimum performance, Carnac drives the native graphics pipeline of the target system.

Carnac supports the following features in prototypes or applications:

- Applications written in C++
- Thread-safe library
- Hard copy produced by PostScript
- Computer Graphics Metafile (CGM) (optional)
- Web browser support (optional)
- Visual properties that can be set on shapes depending on target device
- View layering (data can be categorized)
- Advanced scaling, rotation, selection, and shape modification
- Ability to persist pictures as binary or ASCII

For further information, see the INT Web site.

Trolltech Qt

The Trolltech Qt toolkit, a C++ cross-platform library available on several Windows-based and UNIX systems, provides platform-independent toolsets. Because of this, developers can use it to write applications to a single API with a single source code base. Qt is maintained by Trolltech, which delivers both a free, open-source version and full-featured versions suitable for creating commercial software.

Qt application development functionality, implemented as C++ classes, covers GUI, database, networking, and file handling. Classes and libraries used for database and file handling (and others) are separate from and are not required for using the GUI classes, where the Qt toolkit focuses its functionality.

Qt does not implement GUI by wrapping the native windowing capabilities. Instead, it creates its own abstraction layer, which means the toolkit can change look-and-feel characteristics between Windows, Motif, and other customizable schemes, depending on whether the application runs on a Windows or Motif system.

Qt also supports advanced graphics requirements that use the OpenGL extensions. Applications developed using Qt widgets can use OpenGL functionality.

For more information, see the Trolltech Web site.

UNIX emulation systems

Another approach to creating system architecture that runs UNIX programs on Windows operating systems is to create UNIX emulation by using Win32. Such a program runs as a Win32-based application or service that accepts commands from the user through a command window. Because these systems are emulations and not native Windows subsystems, performance of migrated UNIX applications could suffer and some UNIX functionality may not work in exactly the same way as it does on UNIX. In contrast, because Interix is implemented as an environment subsystem, it can deliver the exact behavior of a UNIX system.

UNIX emulation systems for Windows include:

- Cygwin, which was developed by Cygnus Solutions and later Red Hat, features a port of the GNU development tools and utilities. Cygwin offers a UNIX-like development environment running on Windows. Cygwin does not implement a single, complete UNIX environment. Instead, it combines the features of several UNIX versions and Linux.

- NuTCRACKER, which is part of the MKS Toolkit, features both the C and Korn shells. This makes many UNIX developers feel at home. MKS delivers most of the UNIX 98 APIs plus features from specific UNIX versions and POSIX, such as threads. There is also a complete X-Windows system including a server.
- U/WIN (UNIX for Windows), which was developed by David Korn, can be downloaded free from the AT&T Research Web site for educational and research purposes. Commercial users must have a license. U/WIN uses the Korn shell. U/WIN provides support for most UNIX features including signals, UCB-style sockets, and devices.

Configuring the Development Environment

Before source code begins migrating, a decision about the target build environment needs to be made. This decision should have been a part of the business case for the migration project because each choice has cost and benefit implications. This section considers Interix and Microsoft Visual Studio as target build environments.

Installation of each set of tools requires a suitably sized and configured Windows-based infrastructure. This includes hardware and software for the development servers and for the developer's desktop machines. For more information about implementing a Windows network or about procuring and installing tool software, see your Microsoft Windows documentation or third-party tools documentation.

Configuring Interix

After Interix is installed, it needs to be configured. Administrators probably want to add directories to the PATH variable in startup files. An administrator may want PATH to point to, say, programs and shell scripts in a directory called 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 login scripts in the home directory. The Korn shell uses the login script .profile. Interix provides the /etc/profile.usr file as a template for creating .profile. To use the Interix template, copy /etc/profile.usr to the home directory and rename it .profile. Then 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. The C shell runs all of these files when a user logs on.

Interix maps the root directory (/) to the Windows Services for UNIX installation directory, which is C:\SFU by default. Under the Interix root directory are the 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.

By default, Interix stores system binaries in one of three directories: /bin, /usr/contrib, or /usr/contrib/bin. It stores the X11 binaries in /usr/X11R5/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 to load the source tree (and of course 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.

For more information about the installation and configuration of Interix with Visual Studio, see the following sections.

Configuring Microsoft Visual Studio

Because of the way the Interix references the Visual C++ compiler, the first issue to consider if both are to be installed is which to install first. The following factors need to be considered:

- If Visual C++ is installed before Interix, it configures the Interix SDK to work with the Visual C++ compiler. That is, the **c89** compiler interface utility is set up automatically as the interface to the Microsoft Visual C/C++ compiler. (The **c89** utility can also be invoked as **cc**.)
- If Visual C++ is installed after Interix, the administrator specifies the location of the Visual C++ compiler and linker to the **cc** and **c89** utilities. The information supplied by using the Windows System Properties dialog box creates a Windows system variable named INTERIX_COMPILERDIR and sets its value, in POSIX format, to the path of the directory where Visual C++ is installed. For example, if Visual C++ is installed in directory C:\MSDEV, the value of INTERIX_COMPILERDIR is /dev/fs/C/MSDEV. If the path contains spaces, the 8.3-format version of the path must be used.

The default installation location for Microsoft Visual C++ is the Program Files directory, whose name includes a space. Many utilities, such as **makedepend**, have trouble reading directory names that contain spaces. If the installation directory name includes spaces, the PATH value has spaces. To avoid these issues, add double quotation marks (") in the PATH variable around paths that contain spaces.

For example, change

```
/dev/fs/C/Program Files/Microsoft Visual Studio/VC98/Bin
```

to

```
"/dev/fs/C/Program Files/Microsoft Visual Studio/VC98/Bin"
```

If **cc** or **c89** still cannot find the compiler, try using the 8.3-format version of the directory name. For example, the 8.3-format name for Program Files is PROGRA~1. Make the same change to the value of the environment variable INTERIX_COMPILERDIR, whose value is the directory in which Visual C++ was installed.

Creating a project in Visual Studio

The creation and management of projects in Visual Studio needs to take into account the sources to be migrated from UNIX to Windows. Applications come in all sizes. To migrate the sources for a small to medium-sized application to the Visual Studio development, simply create the projects manually. Larger projects and their associated batch builds could require additional attention.

In Visual Studio, projects can be created by using the project wizards, by manually creating blank projects, or by using the Visual Studio Automation interface. Which method to use depends on how the source will migrate from UNIX.

Creation of a Visual Studio project by any of these methods must follow the following three steps:

1. Decide the project type. For example, is it a Win32 project or a Custom C project?
2. Determine which third-party libraries are necessary.
3. Define the project settings for the application.

The decision matrix shown in Table 3 shows how to use criteria to create an example Visual Studio project. This example shows considerations for two applications: one is an X-Windows OpenGL package; the other is a stand-alone console application.

Table 3. Application decision matrix

Application type	Visual Studio project type	Third-party libraries	Project settings
OpenGL Windows application	Win32 project	OpenGL libraries and cross-platform Windows libraries	Use precompiled headers, compile with **edit** and **continue**
Console C application	Custom C project	None	Use standard C library, not Windows defaults

Populating the Development Environment

One problem to solve early is how to transfer files from one system to the other. A UNIX-to-Windows migration requires a plan for moving UNIX code to a Windows-based computer. To keep parallel versions of the software running on both platforms, it could also be necessary to move modified files back to the UNIX system from the Windows-based system.

After getting 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 the "test" of whether or not the code compiles and links is part of the port or rewrite itself.

This section discusses moving code between systems in general, then it looks at populating the Interix and the Visual Studio environment with the source. Finally, it describes how to construct a build environment in Visual Studio.

General Considerations

When moving source code from one platform to another, look first 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 root, /usr.

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 are probably 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 NFS, then 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 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. Because most installation programs allow the user to customize the name of the top-level directory, these can vary. However, the directory structure under the top-level installation directory is probably 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 Windows, the project files belong in the area allocated for the developer's documents, say, "My Documents." By default, this is C:\Documents and Settings\<user name>\My Documents. Therefore, look for the myapp sources under C:\Documents and Settings\<user name>\My Documents\mysystem\myapp.

Note When migrating files, remember that the UNIX text file format is different from Windows. Depending on the target environment, Windows files may need to be convert to UNIX or vice versa. Interix provides the **flip** utility to do this.

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 utility also allows the use of wildcards. The programs **ksh**, **make**, and **awk** (with the **-f** option) can read script files in either the Interix or Windows format.

Exporting Files from the UNIX Environment

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

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

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

- Use FTP (file transfer protocol) 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, network file system (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 machine issuing the copy commands. When using NFS or CIFS, the remote share needs to be mounted.

Populating an Interix Environment

It should be as straightforward to import an application's configuration, build, and source files into the Interix development environment as it is to move them between two UNIX systems. Applications typically arrive as a compressed or uncompressed **tar** archive file. (For more information on **tar**, see the discussion in "Exporting Files from the UNIX Environment.")

Because Interix integrates with the Windows environment, Interix provides a program group that can start either a ksh or csh shell. To start one of these shells, click **Start** and then point to one of:

- Programs, then Windows Services for UNIX, and then korn Shell

  —or—

- Programs, then Windows Services for UNIX, and then C Shell

From the C or korn shell, the import can proceed.

To import the application configuration, build and source files

1. Change the working directory to /usr/examples with the **cd** command, and then copy the compressed archive from the CD-ROM drive by typing:

   ```
   % cd /usr/examples
   % cp /dev/fs/<CD-ROM Drive Letter>/c_count.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 it into the environment by typing:

   ```
   % cp /dev/fs/<drive letter>/<source directory>/c_count.tar.gz .
   ```

3. Enter the **ls** list directory command:

   ```
   % ls
   ```

   Output from this command looks similar to this:

   ```
   c_count.tar.gz        win32       gawk        admin      rpc
   ```

4. Uncompress and extract the files from the archive by typing:

   ```
   % gunzip < c_count.tar.gz | tar xf -
   ```

   Output from this command looks similar to:

   ```
   % ls
   COPYING    c_count.tar  config.status  descrip.mms  makefile.in
   readme    Makefile  changes    config_h.in  getopt.c
   manifest    system.h  c_count.1  config.cache  configure
   getopt.h  mkdirs.sh  tags    c_count.c  config.h
   configure.bak  install.sh  patchlev.h  testing    c_count.lsm
   config.log  configure.in  makefile  portal
   ```

After the files have been extracted from the archive, then populating the Interix environment is complete.

Populating a Windows Environment

Using the Platform SDK or Visual Studio, populating the Windows environment is almost the same as that described in "Populating an Interix Environment." 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 being imported.

Once the tar file or compressed tar file is in the Windows environment, it needs to be unpacked into the local Windows project area. Use a Windows-based version of tar to do this. Systems that include tar include Interix, MKS NutCracker, and Cygwin. A gz archive can be opened on Windows by using the WinZip archive utility.

After the source files are unpacked, the text for the source file may need a minor conversion. In UNIX, the line-feed character (hex 0A) is used to indicate a new line. In Windows, the combination is carriage return (hex 02) and line feed. To determine whether this conversion is needed, open a source file in Notepad. If the source lacks the proper line feeds, then a conversion is needed. A number of utilities can be used for this. One of them, **flip**, is delivered with Interix. (For more information on flip, see "General Considerations" in this chapter.)

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 simply checking out the source to the appropriate project area.

If the source control system does not allow both clients, however, follow these 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 walks through these steps by using tools available on both UNIX and Windows.

First, the source is checked out of the UNIX source control system into a local user's working directory on UNIX. Most code management tools provide facilities to export an entire code hierarchy and create a tar file directly. If this is not the case, the hierarchy must be checked out and then packaged. The steps to package, move, and unpack the source take place in much the same way as already described in this section.

Once 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
```

Migrating a Build Environment to the Windows SDK

The Platform SDK can be used to compile and link C and C++ programs without Visual Studio projects. The **nmake** tool contains the following functions for building a project:

- **Description blocks**

The **nmake** tool can group dependencies for a specified block of command operations by using a description block. A description block can list dependent source files that need compilation if a source is out of date, for example.

- **Command blocks**

  These group commands that perform operations on the dependencies listed in the description block. Execution of the C compiler (cl) is an example of a command in a command block.

- **Macros**

  Macros can be used to represent strings of characters, such as to represent specific compiler options in an nmake file. For example, CC could represent one set of compiler options.

- **Inference rules**

  These rules infer dependents for targets, such as to update description block targets, even when there are no specified commands in the description block.

- **Derivatives**

  These can be used to preprocess commands in an nmake file. For example, a derivation can specify a macro to process decisions in the nmake file, such as to indicate options for a debug build versus options for a release build.

- **Dot derivatives**

  These can be used to set specific redefined options in an nmake file. For example the .SUFFIXES dot derivative can set the suffixes for inference rule referencing.

Migrating a Build Environment to Visual Studio

The task of maintaining UNIX style code on the Windows platform must take into account the nature of UNIX applications and build procedures. In UNIX, programs are compiled into object modules. The object modules can be linked into executable files or used to create libraries. Libraries can either be static archives (created with **ar**) used to add code to executable files, or be shared objects. UNIX normally links by using the C/C++ compilers.

Windows programs are compiled by using the compiler. They are linked as executable files, static libraries, or DLLs.

Table 4 compares the UNIX and Windows methods for creating executable files, static libraries, and shared libraries.

Table 4. UNIX and Windows file creation comparison

File type	Utility to create	Command options
UNIX executable file	**Linker (ld)**	
UNIX static library	**Ar**	
UNIX shared library	**Linker (ld)**	**-shared**

Windows executable file	**Linker**
Windows static library	**Linker**
Windows dynamic library	**Linker**

There are methods to migrate the build environment:

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

Because each method starts with the UNIX makefile, the first step is to look into the project makefiles to see what information needs to be migrated.

Look at setting and configurations first. To do this, 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 for the functions to be performed, options for each function, and directory locations. Common examples of macros in a makefile are shown in the Table 5.

Table 5. Makefile macro examples

Macro	Used for
CC	An alias for the C/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 CFLAGS, LDFLAGS, LIB, and INCLUDE variables yield the main information needed 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 utilities such as **grep** or **awk**. In the example shown below, **grep** finds the text pattern CFLAGS in the file mymake:

```
% grep CFLAGS mymake
```

Dependency rules can also be expressed in the makefile, as shown in Table 6:

Table 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. By 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

For small to medium-sized applications, often the easiest way to migrate the project is to create the Visual Studio project by hand. 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 could be to standardize on the Visual Studio tools. When all applications, including the largest ones, are considered, manually creating and maintaining the Visual Studio projects becomes a daunting task.

Converting an application build from a makefile-based to a Visual Studio-based project

Rogue Wave SourcePro 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 flags passed to the compiler are the same, the necessary **#defines** are the same, and that the required header file and library file paths are properly set in the Visual Studio environment. One way to obtain this information is to run **nmake** from a command line, and redirect the output to a file. For example, if you use the following command from the command line:

```
nmake > exampleBuild.txt
```

nmake will place the output of the compile and link into the exampleBuild.txt file, provided that there is a makefile (named makefile) in the current directory.

Once 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 Rogue Wave application into an MSVC project

1. Start Visual Studio.
2. On the **File** menu, click **New**.
3. Select **Win32 Console Application** (all Rogue Wave examples are console applications), give it a name, and click **OK**.
4. Select an empty project and click **Finish**.
5. Copy all the source code files from the example application folder to the project folder.
6. Under **FileView** (on the left of the screen), right-click **Source Files**, click **Add files to folder**, and select all of the .cpp files copied to the project folder.
7. Repeat step 6 for **Header Files** and add all the header files in the project folder.
8. On the **Project** menu, click **Settings**, and then select the **C++** tab. At the bottom of the dialog box, you will find a text field for **Project Options**. You will need to find all of the compile options that were used in the Rogue Wave application example and copy them here. To do this:
    - Convert **-I**<*anything*> to **/I** "<*anything*>". If relative path is used in the application, you need to convert it to absolute path.
    - Convert **-D**<*anything*> to **/D** "<*anything*>".

- Convert other options using a similar format; for example, **-GX** becomes **/GX**, and so on.

9. On the **Project** menu, click **Settings**, and then select the **Link** tab. Select **Input** from the drop-down list at the top of the dialog box.
    a. Find an **-L**<*anything*> from the Rogue Wave example application, and copy it to **Additional library path** by entering <*anything*>.
    b. Find the list of all libraries used in the example application, and copy the library names under **Object/library modules**. Remove any pre-existing library names.
10. Compile the project. If you receive error messages about conflicting compiler options, then the options cited in the error message were not found in the Rogue Wave example application. Remove any unfound options.

Using batch builds with Visual Studio

The task of creating a Windows nmake file is similar to creating a Visual Studio project. However, the process of creating a Windows makefile is different. When the Visual Studio IDE creates a project, it automatically generates a Windows makefile suitable for use with a batch build. You can also use a Windows text editor, such as Notepad, to create a project manually.

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.

Creating a custom tool

To avoid the drawbacks of creating a project manually or using a batch build, a possible solution is to create a tool. A custom tool can drive the Visual Studio COM Automation model to automatically create Visual Studio projects. Equipped with such a tool, a developer can create a project that builds the required sources in debug while the rest of the project remains in release. The tool can accomplish this without manually changing an existing Visual Studio project or creating a new one.

Such a tool requires the following:

- A database of source files and dependencies. The UNIX makefile for a typical batch build is the logical starting place for this information. A simple database can be created by parsing the makefile and generating XML to store the resulting information.
- A batch script that uses the source dependency database to drive the Visual Studio COM Automation interface. Windows Script Host provides all the capabilities necessary to access data in XML and to use the automation model.
- A developer familiar with the Visual Studio COM Automation model must be available to create the batch script.

Although using this strategy can slow build times, it should increase developer productivity. The gain will be proportional to the size of the application source base; the larger the base, the greater the gain.

This strategy is also recommended when UNIX is to remain the primary development environment. The tool can automatically convert UNIX makefiles to be used for Windows development. Thus, developers can build the application on Windows and debug cross-platform problems.

In the section "Automation Script for Visual Studio," later in this chapter, a Windows Script Host script uses VBScript and the Visual Studio 6.0 COM Automation model to create a project based on an XML definition. By using scripts like these, an administrator can leverage the interactive debug features of Visual Studio while maintaining a batch build migrated from UNIX.

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, considerations include the following:

- 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 runtime libraries and OpenGL cross-platform libraries.
- Migrate source to multiple platforms with each release, rather than synchronizing source across multiple platforms.
- 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 utility, 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.

Using the Development Environment

This section presents specific information on building, debugging, and automation in the Platform SDK, Visual Studio, and Interix development environments. Each environment has specific tools and advantages.

Using the Platform SDK

The Platform SDK includes a comprehensive set of debugging facilities, as described below.

Microsoft debugging tools first generate the proper work files by using compile-time options. Developers must decide what level of symbols to generate. The debug environment also plays a role.

Debug information is typically generated by using one of the options shown Table 7.

Table 7. Options for generating debug information

Option	Use
/Zi	Creates a program debug database (.pdb) file. The debugger uses this file to step through the source during program execution.
/ZI	Similar to **/Zi**. However, the .pdb file also supports edit

	and continue.
/RTCc	Report runtime truncation error
/RTCs	Enables runtime stack checking, including overrun and under-run of local variables, and stack register verification.
/RTCu	Enables reporting of variables used without initialization.
/RTC1	Combination of **RTCs** and **RTCu**
/Zd	Generates line number information.
/Yd	Places debug information in the .obj files.

Any programs written in languages that support the creation of the .pdb files can be used in the Windows debug environment. Because of this, a developer can debug an application source file where calls are made between C, C++, and Fortran.

A developer can use the Platform SDK WinDBG tool to open a source file, then run the corresponding debug version of an executable file. WinDBG searches for debug symbol information for the modules that the executable file uses in the symbol image path.

To use WinDBG to debug an executable

1. From the GUI, open the source file corresponding to the executable file to debug.
2. Set the path for the symbols needed to debug the executable file.
3. Set desired breakpoints in the source file.
4. Open the executable file.

Other options for using WinDBG include:

- Attach to a running process.
- Open a crash dump file.
- Use a remote debug session to connect to another system.

The command debugger (CMD) can also be used to debug programs. CMD is especially useful to debug a running program or to debug remotely. CMD also can analyze crash dumps by using symbols. For example, this code uses CDB to analyze a crash dump file:

```
cdb -y <Symbol Path>  -i <Image Path> -z <Dumpfile>
```

Using Visual Studio

When migrating a development environment from UNIX to Windows, one goal is usually to leverage the Visual Studio development tools. However, it can appear to be a large manual task to manage the creation and administration of Visual Studio projects. This is especially true for large projects with hundreds or thousands of sources.

To support developer productivity, Visual Studio provides debugging and automation facilities, which are particularly useful for repetitious tasks.

Debugging with Visual Studio

To debug a program using Visual Studio, the program must be compiled with the appropriate options set. (For more information on the options, see the list in "Using the Platform SDK" previously in this chapter.) To use the Visual Studio debugger, in the configuration manager, set the active configuration to debug. When the project is compiled, the debug session can begin.

You can run a debug session within a project opened in the Visual Studio 6.0 or by opening a remote connection from the IDE. You can choose either option from **Build** menu in the Visual Studio IDE.

This section covers local debug sessions. You can start a local debug session by selecting **Start debug** on the **Build** menu, and then either starting the program to be debugged or attaching debug to a running process.

A developer can control a debug session through menu options or toolbars in the Visual Studio 6.0 IDE, or through the use of shortcut keys. A top-level debug is created once a debug session starts. The **Debug** menu includes the symbol for the toolbar button that accomplishes the same command. Table 8 describes the commonly used debug commands.

Table 8. Debug commands

Command	Description
Go	Menu item–Build->Start Debugging->Go Shortcut–F5
Step Into	Menu item–Debug->Step Into Shortcut–F11
Step Over	Menu item–Debug->Step over Shortcut–F10
Step Out	Menu Item–Debug->Step Out Shortcut–Shift +F11
Run to Cursor	Menu item–Debug->Run to Cursor Shortcut–Ctrl+F10
Stop Debugging	Menu item–Debug->Stop Debugging Shortcut–Shift+F5
Restart	Menu Item–Debug->Restart Shortcut–Ctrl+Shift+F5.
Set Break Point	First move the cursor to the line in the source file where you want a break point. Menu item–Right click line in source, then select "Set Break Point". Shortcut–Ctrl+B

To start a debug session

1. Set a breakpoint in the source where the debug execution will stop: right-click the appropriate line in the source, then select **Insert breakpoint** or press **CTRL+B**.
2. On the **IDE Debug** menu, select **Start**, or press **F5**.

   The program executes until the breakpoint is reached. At the breakpoint, from the **Debug** menu you can view local variables, the call stack, registers, memory, processes, and threads from windows.

3. To continue, set another breakpoint. Press **F5** to continue, press **F11** to step into the next line of

code, press **F10** to step over the next code line, or position the cursor to the next line to stop and select **Run to cursor**.

This process continues until you terminate the debug session, the program abnormally terminates, or the program terminates normally.

Developers can use the Visual Studio edit and continue feature during a debug session to change the source and then recompile. If the developer changes the source while debug execution is paused (such as at a breakpoint), the debugger can compile the change before proceeding with the debug session. To enable this feature, compile the source with the **-ZI** option.

Automation with macros

Macros are built by using Visual Basic for Applications (VBA). The VBA IDE can be invoked from within Visual Studio.

Macros can automate project generation and maintenance. In a conversion, macros can synchronize with a batch build environment, integrate tools needed to support the Microsoft development environment, or automate any repetitive process used during development. For example, macros can create add-ins or wizards for the Visual Studio IDE.

A wizard can, for example, automate the task of creating project that supports OpenGL. The wizard offers options such as support for Win32 GUI in the open OpenGL application, or making the OpenGL application console based.

Based on input from the wizard, the appropriate Visual Studio project settings and libraries are included. For example, if Win32 GUI support is needed, the wizard can include the OpenGL graphics library utility toolkit (GLUT) libraries in the project.

Using Interix

The Interix subsystem can be used in conjunction with the Windows Platform SDK or Visual Studio. The SDK, which provides a front end for Microsoft Visual C++, offers the benefits of the native compiler for Windows while retaining a UNIX development environment. But the Interix environment also offers the benefits of the GNU gcc and g++ compilers.

Differences between Interix and UNIX

The Interix environment is a UNIX environment, but there are some differences:

- The Interix SDK can be used to create UNIX-style .so libraries. Although similar to Windows dynamic-link libraries (DLLs), .so libraries are not the same in implementation or semantics.
- Interix does not have or use the /etc/passwd file (or NIS passwd map file), which typically contains a field with the name of each user's login shell. Therefore, when a user logs on to a system by using Telnet, the login shell is not determined by a shell entry in /etc/password, but by how the Telnet server is configured on the remote system. The Interix telnetd server, which can be configured to be started by the inetd daemon (by entering a line in the inetd.conf file), uses the Korn shell as the default login shell.

  In Interix, user and group information is stored either in a Windows SAM database or in an Active

Directory database. The default login shell for Telnet is stored under the User Comment field of a user account. An administrator can change the User Comment field from the command line by using the Windows **net** command as follows:

net user *logname* **/usercomment:"** *default shell***"** [/*domain*]

The *logname* argument is the name with which the user logs on. The /*domain* switch is optional if the user logs on from the current domain. For example, the following command sets the default logon shell to the C shell for a user named "carlos" in the current domain:

```
net user carlos /usercomment:"/bin/csh"
```

- On most UNIX systems, when a user logs on to the Korn shell, the shell runs at least two startup files (login scripts), /etc/profile, and $HOME/.profile for setting up the environment. The C shell runs similar scripts, /etc/csh.cshrc and /etc/csh.login, and multiple files in the home directory.

  Interix shells work in a similar way, but include additional files that have the .lcl extension. The Korn shell reads /etc/profile, which calls /etc/profile.lcl. The C shell reads /etc/csh.login, which calls /etc/csh.lcl. Make any local system changes in the .lcl files. These settings affect all users who log on to a shell.

Building applications with Interix

Applications are built by using Interix in the same way as they are built in the UNIX environment. When building applications in the Interix environment, keep the following considerations in mind:

- The Interix SDK supports Visual C++ for compiling C programs. It does not support Visual C++ for compiling C++ programs. For C++ programs, the GNU g++ compiler is provided.
- The version of **make** provided with the Interix Software Development Kit (SDK) is based on the Berkeley Software Distribution (BSD) 4.4 **make** and supports all BSD features. It also conforms to POSIX.2. The GNU make (**gmake**) utility can help when porting the application's makefiles.
- To convert existing makefiles to run under the Interix subsystem, change macro definitions. Edit the value of $(CFLAGS) to use only flags supported by **cc** or **c89**.
- The Interix SDK supports shared or dynamically linked libraries. Dynamic linking is supported by using standard calls, such as dlopen(). Dynamically linked applications and shared libraries can be created only by using **gcc** and the other GNU compiler tools.
- A typical problem when porting a UNIX application to Interix is that on builds, gcc in the Interix SDK defaults to static libraries. To link to the shared libraries, use the compiler option **--dynamic**. To force the compiler to link to the static libraries, use the compiler option **--static**.
- Shared libraries should linked to the libc.so library visible to them. It is usually incorrect to link shared libraries to libc.a. Shared libraries should also link to libpsxdll.a, except when a shared library makes no direct calls to libpsxdll.a entry points (the system calls). Because these system calls are located in libpsxdll.a, a dynamic-link library, missing system call errors usually indicate that libpsxdll.a was not included in the linking.
- The Interix SDK includes the **liblock** command, which locks a library to prevent the linker from using it. If any process attempts to use a locked library to link, the ld command reports a fatal error. There is no tool provided to unlock a library that has been locked by using **liblock**. However, locking is vulnerable and is not a high-security option. The syntax is:

```
% liblock lib.so
```

Although the **gcc** command understands these rules of linking, use extreme caution if you are invoking **ld** directly.

For an example of building an application with Interix, see "Building and Debugging with Interix" in this chapter.

Running applications with Interix

On a Windows platform, applications run on specific subsystems and in specific environments. On the Interix subsystem, applications run in a UNIX environment; on the Windows subsystem, applications run in a Windows environment. The following examples illustrate this difference:

- When running an application that uses the Korn shell or C shell that come with Interix, or any application that was compiled to use the Interix subsystem, the application runs in a UNIX environment. This environment features case-sensitive file names, paths that use the format /usr/examples, and other paradigms and tools a UNIX developer expects in any UNIX environment.
- When running an application by using the command processor cmd.exe , the application runs in a Windows environment. In this environment, file names are not case sensitive and paths use the C:\SFU\usr\examples convention.

If the UNIX application to be migrated is a daemon, create symbolic links in /etc/rc2.d to scripts that start and stop it, such as the start and stop scripts from the UNIX environment. Interix programs designed to run as daemons can be controlled by using the usual UNIX mechanisms, such as by using the **kill** utility to send signals to the program.

The Interix Korn shell follows traditional Korn shell behavior, which is almost identical to POSIX behavior. But for strict POSIX conformance from the Korn shell, run the shell in POSIX mode.

To run the Korn shell in POSIX mode

1. Invoke the Korn shell with the **-o posix** option.
2. Run this command from within the shell: **set -o posix**.
3. Set the POSIXLY_CORRECT shell parameter.

The C shell does not support POSIX conformance. Unlike the Korn shell, it has no POSIX-mode command options.

Debugging with Interix

The example in "Building and Debugging with Interix" shows how the gdb debugger can be used to debug applications in the Interix environment. Table 9 describes the commands available for gdb:

Table 9. gdb commands

Command	Description
run [*args*]	Runs the program with the arguments specified by *args*. If no arguments are specified, the last arguments given are used.
break [*address*]	Sets a breakpoint to stop execution at the specified

	address. The address can be a function name, a line in the current file, a file name and a line number (as in *filename*: *number*), or an offset from the current stop. If no address is given, the breakpoint is set at the next instruction.
continue [*count*]	Continues execution. If count is specified, this breakpoint is ignored for the next *count* passes.
step [*count*]	Executes until the next line is reached, and repeats *count* times.
next [*count*]	Executes next line, including function calls, and repeats *count* times.
list [*address*]	Shows the next 10 lines of source or shows lines around address. The address can be [*filename*:] *number*, or an offset (indicated by + or -), or a range indicated by *first*, *last*.
backtrace [*n*]	Prints all frames in the stack. If *n* is positive, prints *n* innermost frames. If *n* is negative, prints *n* outermost frames.
print *expression*	Prints the value of *expression*.
Quit	Exits gdb.

The gdb debugger also provides a **help** command for the various operations. To see a list of help issues, type **help** at the gdb prompt. In addition to gdb, the Interix SDK includes the **pstat** and **truss** utilities to help in debugging programs.

The **pstat** utility displays detailed information about a specified process.

> **Note** If you have Administrator privileges (that is, if you are signed on as Administrator and a member of local Administrator group), you can view any process ID. If you do not have Administrator privileges, you can only view process IDs owned by your shell (that is, launched from your shell).

For example, this is the output for process identifier (PID) 10187 (a C shell process):

```
% pstat 10187
pid=10187 ppid=1 pgid=10187 sid=10187 state=3 Active
flags=40000022 execed pgrp asleep
signal trampoline = 0x77EA1F66,0x77EA1F8F  nullapi = 0x77EA7A07,0x77EA7A4F
current syscall=sigsuspend()
IP=77f8224d  SP=0088da1c  BP=0088daf8  FL=00000246  AX=00000000  BX=00000001
CX=00000000  DX=00000000  DI=0088eddc  SI=0088da78
CS=0000001b  DS=00000023  ES=00000023  FS=00000038  GS=00000000  SS=00000023
```

Although it does not have the same functionality, the **truss** utility was inspired by the System V utility of the same name. The **truss** utility outputs information about system calls and signals.

For example, this output represents the system calls made on behalf of the **cat** concatenate and print files utility:

```
% truss cat config.log
tracing pid 12491
```

```
null() null returned 0
open("config.log", 0x1) open returned 3
fstat(1, 0x1210650, 0x1200650) fstat ret: 0 dev: 0x0 ino: 0x00000000
read(3, 0x8228E0, 512) read returned 127 0x7F
write(1, 0x8228E0, 127) This file contains any messages produced by compilers
while running configure, to aid debugging if configure makes a mistake.
write returned 127 0x7F
read(3, 0x8228E0, 512) read returned 0
close(3) close returned 0
close(1) close returned 0
exit(0) process exited with status 0
```

Building and Debugging with Interix

This section presents a C source code line-counting application written by T. E. Dickey. The source can be downloaded from the c_count Web page

This source code provides an example of how to use Interix to build and run an application.

The process involves three stages:

1. Run a configuration script.
2. Build the application.
3. Debug the application.

By creating the application's makefile, a configuration script adapts the source code to the prgoram's execution environment, that is, the hardware and operating system. Configuration scripts convey configuration information to the application package by defining preprocessor variables, indicating the presence or absence of specific features. Typically, the **make** variable CFLAGS provides this information. However, the information can also be put into a file named config.h. With the config.h solution, all the configuration information is in one location. The config.h method is provided by the c_count program.

The config.h file contains information similar to this listing for the c_count program:

```
/*
 * $Id: config_h.in,v 7.1 1994/06/12 23:48:18 tom Exp $
 * config_h.in is a template file used by configure to produce config.h.
 * config_h is then transformed (by using config.status) into the header file
 * config.h - Kevin Buettner.
 */
#define const
#define HAVE_STDLIB_H   1
#define HAVE_GETOPT_H   1
#define HAVE_STRING_H   1
#define HAVE_MALLOC_H   1
#define DECLARED_GETOPT 1
```

The config.h listing for c_count shows that the configuration information defines some compile time macros to be true (1). These are used in the c_count.c program (by including system.h) as arguments to preprocessor #ifdef commands. Thus, they include some specific library header files, such as stdlib.h.

To verify that Interix provides these header files, an administrator can perform specific searches, for example:

```
% find /usr -name stdlib.h -print
/usr/include/stdlib.h
```

Or the administrator can list the contents of the main header directory. This is good check before execution of the configuration script. For example:

```
ls /usr/include
alloca.h   float.h   mpool.h    rpc       typeinfo
ar.h       fnmatch.h ndbm.h     search.h  tzfile.h
arpa       form.h    netdb.h    setjmp.h  ucontext.h
assert.h   fts.h     netinet    signal.h  ulimit.h
cpio.h     ftw.h     new        stdarg.h  unctrl.h
ctype.h    g++       new.h      stddef.h  unistd.h
curses.h   glob.h    nl_types.h stdio.h   utime.h
db.h       gnu       nl_types_private.h stdlib.h utmpx.h
dirent.h   grp.h     opennt     string.h  va_list.h
dlfcn.h    interix   panel.h    strings.h varargs.h
err.h      libgen.h  paths.h    stropts.h vis.h
errno.h    limits.h  poll.h     sys       wait.h
eti.h      locale.h  protocols  syslog.h  wchar.h
exception  malloc.h  pty.h      tar.h     xti.h
excpt.h    math.h    pwcache.h  term.h
fcntl.h    memory.h  pwd.h      termios.h
features.h menu.h    regex.h    time.h
```

Because the following configuration script is not a GNU configure script, it needs to be executed as shown. The script first checks for system-supported features. It then creates the config.status file (used to make the config.h file) and the makefile script to compile and link the application.

```
/configure -prefix=/usr/local -host=intel-pclocal-interix
loading cache ./config.cache
checking for gcc... (cached) cc
checking whether we are using GNU C... (cached) no
checking for a BSD compatible install... (cached) ./install.sh -c
checking how to run the C preprocessor... (cached) cc -E
checking whether make sets ${MAKE}... (cached) no
checking for working const... (cached) no
checking whether cross-compiling... (cached) yes
checking for ANSI C header files... (cached) no
checking for stdlib.h... (cached) yes
checking for getopt.h... (cached) yes
checking for string.h... (cached) yes
checking for malloc.h... (cached) yes
checking for strchr... (cached) no
checking for getopt... (cached) no
checking if compiler supports prototypes... (cached) no
checking if getopt is declared... (cached) yes
creating ./config.status
creating makefile
creating config_h
creating config.h
removing config_h
```

Because configuration scripts are used for a number of application packages, it is helpful to create the following script to run configure scripts as part of the build environment setup. This script sets the compilation flags for C, C++, and CPP. It also includes the /usr/local directory for both include and library files.

```
$ cat /usr/local/bin/runconfig
set -x
CPPFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
CXXFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
CFLAGS="-D_ALL_SOURCE -I/usr/local/include" \
LDFLAGS="-L/usr/local/lib" \
./configure --prefix=/usr/local \
-host=intel-pclocal-interix $*
```

Next, try the **make** command:

```
% make
cc -c -I. -I. -DHAVE_CONFIG_H -g c_count.c
mv c_count oc_count
mv: rename c_count to oc_count: No such file or directory
*** Error code 1 (ignored)
cc  -o c_count c_count.o
```

The compile-time macro definition specifies that there is a config.h file. An error is generated, but only because the makefile script attempts to save an old c_count program file to oc_count by using an **mv** command. This is the first build; therefore, an old c_count program does not exist. The error will not occur on subsequent builds.

This initial test uses the program to perform a self-analysis, which yields the following results:

```
% ./c_count c_count.c
  1015   347    |c_count.c
--------
  1015   347    total lines/statements
   147  lines had comments     14.5 %
    36  comments are inline    -3.5 %
    78  lines were blank       7.7 %
    41  lines for preprocessor 4.0 %
   785  lines containing code  77.3 %
  1015  total lines   100.0 %
  4335  comment-chars 18.5 %
   922  nontext-comment-chars 3.9 %
  5014  whitespace-chars      21.3 %
   643  preprocessor-chars    2.7 %
 12577  statement-chars       53.5 %
 23491  total characters      100.0 %
  1770  tokens, average length 4.85
  0.33  ratio of comment:code
```

The word count command performs a simple check:

```
% wc c_count.c
    1015    3309   23491 c_count.c
```

Notice the agreement with the total number of lines and characters in c_count.c.

This program also includes a self-test script, which yields the following output:

```
% cd testing
% ./run_test.sh
**
**      Case 1: Count lines in test-files (which have both unbalanced quotes and
**              "illegal" characters:
**      (ok)
**
**      Case 2: Suppressing unbalanced-quote:
**      (ok)
**
**      Case 3: Counting by names given in standard-input:
warning: this program uses gets(), which is unsafe.
**      (ok)
**
**      Case 4: Counting bulk text piped in standard-input:
**      (ok)
**
**      Case 5: Counting history-comments
**      (ok)
**
**      Case 6: Display as a spreadsheet
**      (ok)
**
**      Case 7: Display as a spreadsheet (per-file)
**      (ok)
```

In case the application detects a problem, the gdb debugger is available in the Interix SDK.However, gdb has certain limitations in Interix. It provides useful information for object files compiled with gcc and g++, but not for files compiled with cc or c89. It also does not implement watch points.

The Interix implementation of the gdb debugger works with shared libraries. It also includes some additional commands. For more information, see gdb help for the **shared** and **info shared** commands.

gdb can be used on a program or an object file only if the program was created with debugging information, that is, compiled with the **-g** option. The **g** option stores symbol information and line numbers. The debugger refers to the source code only if the source code is available on the system. By default, gdb looks for the source files in the directory from which gdb was executed. Also, gdb can be used to examine the core files created when an application faults.

The basic steps for debugging a program are:

1. Include debugging information when compiling and linking the program.
2. Start gdb.
3. Debug the program.

The following shows the output from the gdb command when executing the c_count application:

```
% gdb c_count
GNU gdb 4.16.1
Copyright 1997 Free Software Foundation, Inc.
gdb is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for gdb. Type "show warranty" for details.
This gdb was configured as "i386-pc-interix"...
(gdb) file c_count
```

```
Reading symbols from c_count …
Done.
(gdb) run c_count.c  (This statement says run the program with argument
"c_count.c")
Starting program: /usr/examples/c_count/c_count c_count.c
  1015   347   |c_count.c
--------
  1015   347    total lines/statements
   147  lines had comments        14.5 %
    36  comments are inline       -3.5 %
    78  lines were blank           7.7 %
    41  lines for preprocessor     4.0 %
   785  lines containing code     77.3 %
  1015  total lines              100.0 %
  4335  comment-chars             18.5 %
   922  nontext-comment-chars      3.9 %
  5014  whitespace-chars          21.3 %
   643  preprocessor-chars         2.7 %
 12577  statement-chars           53.5 %
 23491  total characters         100.0 %
  1770  tokens, average length 4.85
  0.33  ratio of comment:code
Program exited normally. (Notice the program's output is identical.)
(gdb) list
917     int     main (
918             _ARG(int,       argc),
919             _ARG(char **,   argv)
920                     )
921             _DCL(int,       argc)
922             _DCL(char **,   argv)
923     {
924             register int j;
925             auto    char    name[BUFSIZ];
926             auto    int     opt_all = -1;
The gdb list command lists the next 10 lines of source code. (Note:
  gdb was able to find source file in the current directory.)
(gdb) list 929 (A line number can be specified, to list 10 lines
  starting 5 lines before the specified line)
924             register int j;
925             auto    char    name[BUFSIZ];
926             auto    int     opt_all = -1;
927
928             quotvec = typeCalloc(char *, (size_t)argc);
929             while ((j = getopt(argc,argv,"cdijlo:pq:stv")) !=
  EOF) switch(j) {
930             case 'l':       opt_all = FALSE; opt_line = TRUE;
break;
931             case 'c':       opt_all = FALSE; opt_char = TRUE;
break;
932             case 'i':       opt_all = FALSE; opt_name = TRUE;
break;
933             case 's':       opt_all = FALSE; opt_stat = TRUE;
break;
(gdb) break 929  (Set a breakpoint at source line 929)
Breakpoint 1 at 0x402cd4: file c_count.c, line 929.
(gdb) run
Starting program: /usr/examples/c_count/c_count c_count.c
Breakpoint 1, main (argc=2, argv=0x106ac) at c_count.c:929
929             while ((j = getopt(argc,argv,"cdijlo:pq:stv")) !=
  EOF) switch(j) {
```

```
(gdb) print argc    (Print out argc, the number of arguments)
$1 = 2
(gdb) print argv[1]  (Print out the value of the second argument)
$4 = 0x107e2 "c_count.c"
(gdb) print argv[0]   (Print out the value of the first argument)
$5 = 0x107c4 "/usr/examples/c_count/c_count"
(gdb) continue  (Now continue)
Continuing.
  1015   347   |c_count.c
--------
  1015   347    total lines/statements
   147  lines had comments        14.5 %
    36  comments are inline       -3.5 %
    78  lines were blank           7.7 %
    41  lines for preprocessor     4.0 %
   785  lines containing code     77.3 %
  1015  total lines              100.0 %
  4335  comment-chars             18.5 %
   922  nontext-comment-chars      3.9 %
  5014  whitespace-chars          21.3 %
   643  preprocessor-chars         2.7 %
 12577  statement-chars           53.5 %
 23491  total characters         100.0 %
  1770  tokens, average length 4.85
  0.33  ratio of comment:code
Program exited normally.
```

For more information about building and debugging with Interix, see Help for Services for UNIX and the gdb online help.

Building and Debugging with Visual Studio

This section examines creating a project for the T.E. Dickey Count program for Win32 using Visual Studio 6.0. (You should already have Visual Studio 6.0 installed on your system prior to running this example.)

First, you must create a Visual Studio project.

To create a Visual Studio project

1. Start Visual Studio 6.0.
2. Create a directory for the count project. For this example, create a Samples directory in your My Documents folder. If you are using the C drive, the full path is C:\Documents and Settings\<*CurrentUser*>\My Documents\Samples\.
3. Start Visual C++ 6.0.
4. Create a new project: on the **File** menu, click **New**.
5. Select the **Projects** tab. In the **Location** box, enter the path for the new project directory. In this example, the path is C:\Documents and Settings\<*Current User*>\My Documents\Samples.
6. In the **Project Name** box, enter the name of the project. For this example, use vscount.
7. Click the **Create New Workspace** button.
8. In the **Platforms** list, select the **Win32** check box.
9. Select **Win32 Console Applications** from the list box, and click **OK**.
10. A new dialog box will appear with the title **Win32 Console Application–Step 1 of 1**. Select the **An Empty Project** button, and click **Finish**.

A summary dialog box will appear with the following information:

Win32 Console Application will create a new skeleton with the following specifications:

+Empty console application

+No files will be created or added to the project

Project Directory:

C:\Documents and Settings\<Current User>\Samples\vscount"

You now have a Visual C++ 6.0 empty console application project and are ready to populate the project with the count source files.

To populate the project

1. Download the c_count source.
2. This site contains a gzip tar file. Extract the gzip tar file to a temporary directory. (WinZip will perform the extract.) Copy the following extracted files to the project directory you created.
   - Source files—c_count.c (extracted in the c_count-7.7 directory), getopt.c (extracted in the c_count-7.7\porting directory)
   - Header files—patchlev.h and system.h (extracted in the c_count-7.7 directory), getopt.h (extracted in the c_count-7.7\porting directory)
3. Now you are ready to add the count source files to the empty project, as follows:
   a. Select the **File View** tab on the Visual C++ 6.0 Project Explorer window. This is usually on the left side of the project workspace. Expand the tree view for vscount by clicking the "+" character. If the "-" character is shown, the project folders are already in view.
   b. Highlight the **Source Files** folder.
   c. On the **Project** menu, select **Add to Project**, and then select **Files**. The **Insert Files into Project** window appears.
   d. Navigate to the directory that contains c_count.c and getopt.c. For the example, this is the C:\Documents and Settings\<Current User>\Samples\vscount folder.
   e. Select the files to insert into the project.

The program is now ready to compile. Visual C++ automatically selects Win32 Release and Win32 Debug as available configurations to compile and sets Win32 Debug as the active configuration. You can add other configurations by selecting **Configurations** on the **Build** menu. A window will appear that lists the two available configurations, and allows an option to add configurations. However, for this the example, Win32 Release and Win32 and Win32 Debug are the only configurations needed.

Although the project automatically sets Win32 Debug as the active configuration, you can easily change this by selecting **Set Active Configuration** on the **Build** menu. The **Set Active Project Configuration** window will appear. To change the active configuration, highlight the desired configuration in the list, and click **OK**. However, for this the example, no change is needed.

To compile the project, select **Build vscount** on the **Build** menu or press the **F7** shortcut key. If you don't change any of the Visual C++ default compile and link settings, the build will report the following in the output window. This particular set of source generates one warning, but otherwise builds cleanly.

```
Deleting intermediate files and output files for project 'vscount - Win32 Debug'.
  --------Configuration: vscount - Win32 Debug----------
Compiling...
getopt.c
c_count.c
c:\Documents and Settings\<Current User>\Samples\vscount\c_count-
7.7\c_count.c(1164) : warning C4013: '_setargv' undefined;
assuming extern returning int
Linking...
vscount.exe - 0 error(s), 1 warning(s)
```

There are various settings you could change by selecting **Setting** on the **Project** menu or by pressing the **Alt+F7** keys. A window will appear with tabs that categorize the particular options. These options are described in Table 10.

Table 10. Compile settings

Dialog tab	Description
General	General options include the project directories, project arguments, executable name
Debug	Options for debugging
C/C++	The various compiler options
Link	The various linker options
Resources	Language, resource file and pre-processor definitions
Browse info	Options for builder the browse info files
Custom build	Additional command options to add to the build
Pre-link step	List of commands to run prior to the link
Post-build step	List of commands to run after the build

The default project setting for the Win32 Debug configuration creates the files you need so that you can debug the project. However, you need to set a few additional options to actually debug the program.

- You must set the debug working directory. To do this, click the Debug tab on the Project Settings window, and enter the path in the Working Directory box. For this example, use C:\Documents and Settings\<Current User>\Samples\vscount. (You should have already specified this directory, so you can skip this step.)
- You must set the program arguments. To do this, click the Debug tab on the Project Settings window, and enter the arguments in the Program Arguments text box. For this example, enter the name of the source file to be counted: in this case, c_count.c.

Now you are ready to run the program in a debug session. First run the program without setting break points. To do this, on the **Build** menu, select the **! Execute vscount.exe** menu option or press the **Ctrl+F5** shortcut keys or the **!** symbol on the toolbar. If you run the program without break points, you receive the following output in a command line window.

```
1270   453   |c_count.c
-------
  1270   453    total lines/statements

   185  lines had comments        14.6 %
    33  comments are inline       -2.6 %
    92  lines were blank           7.2 %
    49  lines for preprocessor     3.9 %
   977  lines containing code     76.9 %
  1270  total lines              100.0 %

  5127  comment-chars             16.6 %
  1037  nontext-comment-chars      3.3 %
  7725  whitespace-chars          24.9 %
   851  preprocessor-chars         2.7 %
 16231  statement-chars           52.4 %
 30971  total characters         100.0 %
  2243  tokens, average length 5.00
  0.30  ratio of comment:code
    46  top-level blocks/statements
     7  maximum blocklevel
2.80  ratio of blocklevel:code
```

To begin an actual debug session, you must determine how to start the program and where to stop the program. For this example, set a break point at line 1164 in the c_count.c program.

To set the break point and start the debug session

1. Select the Project Explorer **FileView** tab.
2. Open the c_count.c source in the main project window by double-clicking the appropriate file name under the **Source Files** folder of the **vscount** tree view.
3. Move the cursor to line 1164 in the source file, and set a break point by right-clicking and selecting **Insert/Remove Breakpoint** on the menu. You can also set a break point by pressing **Ctrl+B** and specifying line 1164 on the **Break at** list.
4. Start the debug session by clicking the **Build** menu, selecting **Start Debug**, and clicking **Go**. You can also start the debug session by pressing **F5**.

The debug session will start and the following output will appear when the program stops at line 1164.

```
Loaded 'ntdll.dll', no matching symbolic information found.
 Loaded 'C:\WINNT\system32\kernel32.dll', no matching symbolic information found.
```

At this point, you can view debug information about the project. Table 11 lists the debug information windows available and the values for the variables, call stack windows, and registers.

Table 11. Debug output

Window	Description
Output	Output window for debug information
Watch	Window to define variables to watch during the debug session
Variables	Values of local variable within functions in the call stack
Registers	Values of system registers
Call Stack	List of the current call stack
Memory	Values at memory address

Variables Window

```
+     &argc  0x0012ff88
```

```
      argc   0x00000002
+     &argv  0x0012ff8c "•D"
+     argv   0x00440e80
      j      0xcccccccc
+     name   0x0012fd7c ""
```

Call Stack Window

```
main(int 0x00000002, char * * 0x00440e80) line 1164
mainCRTStartup() line 206 + 25 bytes
KERNEL32! 77e7eb69()
```

Registers Window

```
EAX = CCCCCCCC EBX = 7FFFF000
 ECX = 00000000 EDX = 00440D80
 ESI = 00000000 EDI = 0012FF80
 EIP = 0040135E ESP = 0012FD24
 EBP = 0012FF80 EFL = 00000206 CS = 001B
 DS = 0023 ES = 0023 SS = 0023 FS = 0038
 GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=0
 PE=1 CY=0
 ST0 = +0.00000000000000000e+0000
 ST1 = +0.00000000000000000e+0000
 ST2 = +0.00000000000000000e+0000
 ST3 = +0.00000000000000000e+0000
 ST4 = +0.00000000000000000e+0000
 ST5 = +0.00000000000000000e+0000
 ST6 = +0.00000000000000000e+0000
 ST7 = +0.00000000000000000e+0000
 CTRL = 027F STAT = 0000 TAGS = FFFF
 EIP = 00000000 CS = 0000 DS = 0000
 EDO = 00000000
```

You can now try the additional debug functions such as:

- Stepping line by line through the program
- Setting another break point
- Stepping in or out of functions
- Continuing with the program execution

For now, continue program execution by selecting the **Go** on the **Debug** menu option or by pressing **F5**. The debug menu appears when a debug session starts. A complete list of debug navigation options is available in the "Debugging with Visual Studio" section of this chapter.

If you continue program execution, the program will terminate normally, adding the following entries to the output window.

```
The thread 0x1DC has exited with code 0 (0x0).
The program 'C:\Documents and Settings\<Current
User>\SAMPLES\vscountv\Debug\vscount.exe' has exited with code 0 (0x0).
```

> **Note** The thread number will vary by depending on the load of the system used for testing this example.

Automation Script for Visual Studio

This section presents a Windows Script Host script that uses VBScript and the Visual Studio 6.0 COM Automation model to create a project based on an XML definition. The script automates the process of creating Visual Studio projects for testing and debug purposes. It assumes the application uses a batch build environment.

In this example, "control script" creates and populates a Visual Studio 6.0 project by using a project definition contained in an XML file. XML was chosen as a portable, self-describing data format that can be used in the command-line interface of any integrated development environment. Because of this, the XML definition need not change even if the Automation interfaces for Visual Studio change.

Project Automation Script

```
<?xml version="1.0"?>
<!--
File Name: VSAutomation.wsf
Purpose  : This file contains jobs to process the scripts.
           It needs VSAutoLib.vbs and VSProjectData.xml files.
Author   : Microsoft Consulting Services
Date     : March 18, 2002
-->
<package>
  <job id="CREATE">
    <script language="VBScript" src="VSAutoLib.vbs" />
    <script language="VBScript">
      <![CDATA[
        createVSProjects "E:\\VSProjectData.xml"
      ]]>
    </script>
  </job>
  <job id="BUILD">
    <script language="VBScript" src="VSAutoLib.vbs" />
    <script language="VBScript">
      <![CDATA[
        buildVSProjects "E:\\VSProjectData.xml"
      ]]>
    </script>
  </job>
</package>
XML Project Data Definition
<?xml version="1.0"?>
<!--
  File Name: VSProjectData.xml
  Purpose  : Visual Studio Project Data
  Author   : Microsoft Consulting Services
  Date     : March 18, 2002
-->
<Workspace>
  <Projects>
    <Project>
      <Name>myProject</Name>
      <Type>Console Application</Type>
      <Path>E:\MigrationSource\myProjects\myProject\</Path>
      <Files>
        <File>
```

```
          <Name>mySrcFile1.cpp</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
        <File>
          <Name>mySrcFile2.cpp</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
        <File>
          <Name>myHdrFile1.h</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
      </Files>
      <Configurations>
        <Configuration>
          <Name>Win32 Debug</Name>
          <BuildFlag>No</BuildFlag>
        </Configuration>
        <Configuration>
          <Name>Win32 Release</Name>
          <BuildFlag>yes</BuildFlag>
        </Configuration>
      </Configurations>
    </Project>
    <Project>
      <Name>myProject2</Name>
      <Type>Console Application</Type>
      <Path>E:\MigrationSource\myProjects\myProject2\</Path>
      <Files>
        <File>
          <Name>mySrcFile1.cpp</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
        <File>
          <Name>mySrcFile2.cpp</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
        <File>
          <Name>myHdrFile1.h</Name>
          <Path>E:\MigrationSource\source\</Path>
        </File>
      </Files>
      <Configurations>
        <Configuration>
          <Name>Win32 Debug</Name>
          <BuildFlag>yes</BuildFlag>
        </Configuration>
        <Configuration>
          <Name>Win32 Release</Name>
          <BuildFlag>no</BuildFlag>
        </Configuration>
      </Configurations>
    </Project>
  </Projects>
</Workspace>
<!-- End of Data -->
```

VBScript Using Visual Studio COM Automation

```
Option Explicit
```

```
'*********************************************************************
' File Name: VSAutoLib.vbs
' Purpose  : This file contains library functions used by
  VSAutomation.wsf
' Author   : Microsoft Consulting Services
' Date     : March 18, 2002
' History  :
' ==========
'
'
'*********************************************************************
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Procedure Name: createVSProjects
' Parameters    : 1. XML Data File Name
' Logic         : o Load the XML Data File
'                 o Traverse through the Project Information and
  Create it
'                   if did not exist
'                 o Traverse through the File Information and Add
  them to
'                   the Project
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Public Sub createVSProjects(strXMLDataFileName)
  Dim xmlDoc               'XML Document Object
  Dim xmlProjectNodeList   'Project Node List
  Dim xmlProjectChildNode  'Project Child Node
  Dim xmlFileNodeList      'File Node List
  Dim xmlFileChildNode     'File Child Node
  Dim intProjectIndex      'Project Index
  Dim intFileIndex         'File Index
  Dim strPName             'Project Name
  Dim strPType             'Project Type
  Dim strPPath             'Project Path
  Dim strPFullName         'Project Full Name
  Dim strFName             'File Name
  Dim strFPath             'File Path
  Dim strFFullName         'File Full Name
  Dim objApplication       'Visual Studio Application Object
  Dim colDocuments         'Documents Collection Object
  Dim objDocument          'Document object
  Dim colProjects          'Projects Collection Object
  Dim objProject           'Project Object
  Dim objFileSystem        'File System Object
  Dim blnProjectExists     'Project Flag
  Dim blnFileExists        'File Flag

  '***
  ' Initialize the object handles
  '***
  Set xmlDoc = Nothing
  Set colDocuments = Nothing
  Set xmlProjectNodeList = Nothing
  Set xmlProjectChildNode = Nothing
  Set xmlFileNodeList = Nothing
  Set xmlFileChildNode = Nothing
  Set objProject = Nothing
  Set colProjects = Nothing

  '***
  ' Load the XML Project Data file
  '***
  Set objFileSystem = CreateObject("Scripting.FileSystemObject")
  blnFileExists = objFileSystem.FileExists(strXMLDataFileName)
  Set objFileSystem = Nothing

  If blnFileExists Then
    WScript.echo "XML Data File Name: " & strXMLDataFileName
    Set xmlDoc = CreateObject("Microsoft.XMLDOM")
    xmlDoc.Async = False
    xmlDoc.Load(strXMLDataFileName)

    '***
    ' Create the Application Object
    '***
    Set objApplication = CreateObject("MSDEV.Application")
    objApplication.Visible = False
    '***
    ' Get the Project Node List
    '***
    Set xmlProjectNodeList =
xmlDoc.documentElement.selectNodes("Projects/Project")

    If xmlProjectNodeList.Length > 0 Then
      WScript.Echo "Total Number of Projects: " &
        xmlProjectNodeList.Length
      WScript.Echo

      '***
      ' Traverse the Project Node List
      '***
      For intProjectIndex = 0 To (xmlProjectNodeList.Length - 1)
        Set xmlProjectChildNode = xmlProjectNodeList.nextNode

        '***
        ' Get the Project Information
        '***
        strPName = xmlProjectChildNode.selectSingleNode("Name").text
        strPType = xmlProjectChildNode.selectSingleNode("Type").text
        strPPath = xmlProjectChildNode.selectSingleNode("Path").text
        strPFullName = strPPath & strPName & ".dsp"

        WScript.Echo "Project Number: " & intProjectIndex + 1
        WScript.Echo "        Name  : " & strPFullName
        WScript.Echo "        Type  : " & strPType
        WScript.Echo
        '***
        ' Check the Project Existence and Create the Project
        '***
        Set objFileSystem = CreateObject("Scripting.FileSystemObject")
        blnProjectExists = objFileSystem.FileExists(strPFullName)
        Set objFileSystem = Nothing
        Set colDocuments = objApplication.Documents
        colDocuments.CloseAll

        If blnProjectExists Then
          WScript.Echo "Project " & strPFullName & " already
exists!!!"
          WScript.Echo
          '***
          ' Open the Project
          '***
          colDocuments.Open strPFullName, "Auto"
        Else
          WScript.Echo "Project: " & strPFullName & " does not
exist!!!"
          objApplication.AddProject strPName, strPPath, strPType,
True
          WScript.Echo "Error: " & Err.Description
          WScript.Echo "Project: " & strPFullName & " has been
created!!!"
          WScript.Echo
        End If

        '***
        ' Traverse through the VS Project collection
        '***
        Set colProjects = objApplication.Projects

        WScript.Echo "Total Projects in Collection: " &
          colProjects.Count

        For Each objProject In colProjects
          If objProject.FullName = strPFullName Then
            '***
            ' Get the File Node List
            '***
            Set xmlFileNodeList =
xmlProjectChildNode.selectNodes("Files/File")

            If xmlFileNodeList.Length > 0 Then
              WScript.Echo Space(5) & "Total Number of Files: " &
                xmlFileNodeList.Length
              WScript.Echo
              '***
              ' Traverse the File Node List
              '***
              For intFileIndex = 0 To (xmlFileNodeList.Length - 1)
                Set xmlFileChildNode = xmlFileNodeList.nextNode

                '***
                ' Get the File Information
                '***
                strFName =
                  xmlFileChildNode.selectSingleNode("Name").text
                strFPath =
                  xmlFileChildNode.selectSingleNode("Path").text
                strFFullName = strFPath & strFName

                WScript.Echo Space(5) & "File Number: " &
                  intFileIndex + 1
                WScript.Echo Space(5) & "     Name  : " & strFName
                WScript.Echo Space(5) & "     Path  : " & strFPath
                WScript.Echo

                '***
                'Add File to the Project
                '***
                If objProject.Type = "Build" Then
                    On Error Resume Next
                    objProject.AddFile strFFullName
```

```
                      WScript.Echo Space(5) & "File " & strFFullName
                      & " added to the Project"
                End If
              Next
              WScript.Echo
            Else
              WScript.Echo "XML Data does not have any Files to be
                added to the Current Project!!!"
            End If
            '***
            ' Done with the current project - exit the loop
            '***
            Exit For
          End If
        Next
      Next
    Else
      WScript.Echo "XML Data does not have any Project
Information!!!"
    End If
  Else
    WScript.Echo "XML Data File " & strXMLDataFileName & " does not
      exist!!!. Please check the Path."
  End If
  '***
  ' Quit the Application
  '***
  objApplication.Quit
  '***
  ' Cleanup
  '***
  If Not (objProject Is Nothing) Then
    Set objProject = Nothing
  End If

  If Not (colProjects Is Nothing) Then
      Set colProjects = Nothing
  End If
  If Not (xmlFileChildNode Is Nothing) Then
    Set xmlFileChildNode = Nothing
  End If

  If Not (xmlFileNodeList Is Nothing) Then
    Set xmlFileNodeList = Nothing
  End If

  If Not (xmlProjectChildNode Is Nothing) Then
    Set xmlProjectChildNode = Nothing
  End If

  If Not (xmlProjectNodeList Is Nothing) Then
    Set xmlProjectNodeList = Nothing
  End If

  If Not (xmlDoc Is Nothing) Then
    Set xmlDoc = Nothing
  End If
  If Not (colDocuments Is Nothing) Then
    Set colDocuments = Nothing
  End If
  If Not (objApplication Is Nothing) Then
    Set objApplication = Nothing
  End If

  WScript.Echo
  WScript.Echo "Done!!!"
End Sub
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Procedure Name: buildVSProjects
' Parameters    : 1. XML Data File Name
' Logic         : o Load the XML Data File
'                 o Traverse through the Project Information and Get
the
'                   Configurations
'                 o Build the requested configurations
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Public Sub buildVSProjects(strXMLDataFileName)
  Dim xmlDoc              'XML Document Object
  Dim xmlProjectNodeList  'Project Node List
  Dim xmlProjectChildNode 'Project Child Node
  Dim intProjectIndex     'Project Index
  Dim xmlConfigNodeList   'Configuration Node List
  Dim xmlConfigChildNode  'Configuration Child Node
  Dim intConfigIndex      'Configuration Index
  Dim strPName            'Project Name
  Dim strPType            'Project Type
  Dim strPPath            'Project Path
  Dim strPFullName        'Project Full Name
  Dim objApplication      'Visual Studio Application Object
  Dim colDocuments        'Documents Collection Object
  Dim colProjects         'Projects Collection Object
  Dim objProject          'Project Object
  Dim objFileSystem       'File System Object
  Dim blnProjectExists    'Project Flag
  Dim objConfiguration    'Configuraion Object
  Dim colConfigurations   'Configuraion Collection Object
  Dim strConfigName       'Configuration Name
  Dim strBuildFlag        'Configuration Build Flag
  Dim blnFileExists       'File Flag

  '***
  ' Initialize the object handles
  '***
  Set xmlDoc = Nothing
  Set xmlProjectNodeList = Nothing
  Set xmlProjectChildNode = Nothing
  Set xmlConfigNodeList = Nothing
  Set xmlConfigChildNode = Nothing
  Set objProject = Nothing
  Set colProjects = Nothing
  Set objConfiguration = Nothing
  Set colConfigurations = Nothing
  '***
  ' Load the XML Project Data file
  '***
  Set objFileSystem = CreateObject("Scripting.FileSystemObject")
  blnFileExists = objFileSystem.FileExists(strXMLDataFileName)
  Set objFileSystem = Nothing
  If blnFileExists Then
    WScript.echo "XML Data File Name: " & strXMLDataFileName
    Set xmlDoc = CreateObject("Microsoft.XMLDOM")
    xmlDoc.Async = False
    xmlDoc.Load(strXMLDataFileName)

    '***
    ' Create the Application Object
    '***
    Set objApplication = CreateObject("MSDEV.Application")
    objApplication.Visible = False

    '***
    ' Get the Project Node List
    '***
    Set xmlProjectNodeList =
      xmlDoc.documentElement.selectNodes("Projects/Project")

    If xmlProjectNodeList.Length > 0 Then
      WScript.Echo "Total Number of Projects: " &
        xmlProjectNodeList.Length
      WScript.Echo

      '***
      ' Traverse the Project Node List
      '***
      For intProjectIndex = 0 To (xmlProjectNodeList.Length - 1)
        Set xmlProjectChildNode = xmlProjectNodeList.nextNode

        '***
        ' Get the Project Information
        '***
        strPName = xmlProjectChildNode.selectSingleNode("Name").text
        strPPath = xmlProjectChildNode.selectSingleNode("Path").text
        strPFullName = strPPath & strPName & ".dsp"

        WScript.Echo "XML Project Name: " & strPFullName
        WScript.Echo

        Set colDocuments = objApplication.Documents
        colDocuments.CloseAll

        On Error Resume Next
        colDocuments.Open strPFullName, "Auto"

        Set colProjects = objApplication.Projects
        For Each objProject In colProjects
          If objProject.FullName = strPFullName Then
            '***
            ' Get the Configuration Node List
            '***
            Set xmlConfigNodeList =
xmlProjectChildNode.selectNodes("Configurations/Configuration")

            If xmlConfigNodeList.Length > 0 Then
              WScript.Echo "Total Number of Configurations: " &
                xmlConfigNodeList.Length
              WScript.Echo
              For intConfigIndex = 0 To (xmlConfigNodeList.Length -
1)
```

```
                Set xmlConfigChildNode = xmlConfigNodeList.nextNode

                '***
                ' Get the Configuration Information
                '***
                strConfigName =
                  xmlConfigChildNode.selectSingleNode("Name").text
                strBuildFlag  =
xmlConfigChildNode.selectSingleNode("BuildFlag").text

                WScript.Echo "Configuration Number: " &
                  intConfigIndex + 1
                WScript.Echo "              Name  : " & strConfigName
                WScript.Echo "              Flg   : " & strBuildFlag
                WScript.Echo
                Set colConfigurations = objProject.Configurations
                For Each objConfiguration In colConfigurations
                  If InStr(UCase(objConfiguration.Name),
                    UCase(strConfigName)) > 0 Then
                    If UCase(strBuildFlag) = "YES" Then
                      WScript.Echo "Building " & strConfigName &
"..."
                      WScript.Echo
                      objApplication.Build objConfiguration
                    End If
                    Exit For
                  End If
                Next
              Next
            Else
              WScript.Echo "XML Data does not have any Configurations
                for the Current Project!!!"
            End If
            Exit For
          End If
        Next
      Next
    Else
      WScript.Echo "XML Data does not have any Project
Information!!!"
    End If
  Else
    WScript.Echo "XML Data File " & strXMLDataFileName & " does not
      exist!!!. Please check the Path."
  End If

  '***
  ' Quit the Application
  '***
  objApplication.Quit
  '***
  ' Cleanup
  '***
  If Not (objProject Is Nothing) Then
    Set objProject = Nothing
  End If

  If Not (colProjects Is Nothing) Then
      Set colProjects = Nothing
  End If
  If Not (xmlConfigChildNode Is Nothing) Then
    Set xmlConfigChildNode = Nothing
  End If

  If Not (xmlConfigNodeList Is Nothing) Then
    Set xmlConfigNodeList = Nothing
  End If

  If Not (xmlProjectChildNode Is Nothing) Then
    Set xmlProjectChildNode = Nothing
  End If

  If Not (xmlProjectNodeList Is Nothing) Then
    Set xmlProjectNodeList = Nothing
  End If

  If Not (xmlDoc Is Nothing) Then
    Set xmlDoc = Nothing
  End If
  If Not (objConfiguration Is Nothing) Then
    Set objConfiguration = Nothing
  End If
  If Not (colConfigurations Is Nothing) Then
    Set colConfigurations = Nothing
  End If

  If Not (objApplication Is Nothing) Then
    Set objApplication = Nothing
  End If

  WScript.Echo
  WScript.Echo "Done!!!"
End Sub
```

patterns & practices
proven practices for predictable results

Send feedback to Microsoft