Visual Patch®

User's Guide

IndigoRose SOFTWARE DESIGN CORP.

www.indigorose.com

Note: This User's Guide is also available as a professionally printed, perfect-bound manual. To order your copy, please visit www2.ondemandmanuals.com/indigorose.

Welcome!

Introduction

Visual Patch is a fast and efficient system for creating software patches. This unique solution features state-of-the-art binary differencing and compression algorithms, combined with a powerful full-history patching engine. Visual Patch makes an excellent addition to the toolbox of any software developer or IT manager.

We’ve improved the development environment to streamline your workflow and have introduced unprecedented flexibility with a brand-new scripting engine and action library. We’ve also added a customizable screen manager, project themes (skins), action plugins, MD5 security, and many other powerful and timesaving features.

You’ll find that Visual Patch simplifies software version management. It turns the otherwise complicated task of managing point-releases into a fast and automated process. There is no easier way to make professional-quality, full-history binary patches for your software and other electronic content.

What is Visual Patch?

It doesn't matter what kind of electronic content you're distributing. Whether it's application software, databases, spreadsheets, video, audio or any other data, the likelihood is that it will change over time. That's why you need Visual Patch.

Visual Patch makes it easy to create professional, rock-solid software patches. As you release new software (documents, data, etc.), simply drag-and-drop the files into Visual Patch and it will automatically track and package the differences for you.

Using a sophisticated combination of binary differencing, data compression and MD5 fingerprinting, Visual Patch is able to create compact full-history software patches that outperform any comparable tool on a value and feature-by-feature basis.

Visual Patch is an integral part of a complete and effective approach to software lifecycle management. Software developers, network administrators and IT departments all require a professional tool for building software patches. As such, Visual Patch integrates seamlessly with other Indigo Rose products like Setup Factory (software installation) and TrueUpdate (automated updating/patch delivery) to provide a complete solution for all aspects of software delivery.

Key Features of Visual Patch

Vista Compatible

Visual Patch's design workspace and generated patches are compatible with Windows Vista, including a configurable "requested execution level" setting for the patch's manifest.

Custom Resource Stamping

Visual Patch allows you to use your own product icon and provides full control of the version information that you want written into the patch’s resources.

Integrated Code Signing

Protect the integrity of your company and products by code signing your patches with your own certificate during the build process.

Sophisticated Version Management

Visual Patch makes it easy to manage your software releases. You can quickly and easily add, remove and organize product versions, and see all of the important details about your files. The toolbars, columns, sort-order, color-coding and other interface elements are fully customizable, allowing you to view only the information that is important to your workflow.

State-of-the-Art Binary Differencing Engine

Visual Patch features state-of-the-art binary differencing and compression algorithms. This ensures smaller and better performing software patches. The algorithms used by Visual Patch were developed specifically for their impressive speed characteristics. Whereas software patching was once a notoriously slow and cumbersome process, companies choosing Visual Patch have a distinct advantage.

Full-History Patching

With Visual Patch, you have full control over which versions of your software can be patched with a single executable. While some vendors will choose to include only a single version update, others may offer a patch that can update any version that has ever been released. Visual Patch can handle either extreme, or anything in between. Unlike some other products, Visual Patch’s full-history patching ability can update any version of your software to the latest release. By only having one file to download, Visual Patch brand patches are much more user friendly. Your users won’t have to worry about finding multiple patches and applying them in the correct order as they would if you used other, less sophisticated patch builders.

Internet Ready, Single-File Executable

Smaller and faster means a better experience for your customers, and Visual Patch delivers. Compare our tiny ~500 KB runtime overhead to the competition and see for yourself. Patches created with Visual Patch are faster to initialize and install than those created by competitive tools. What’s more, our Publishing Wizard walks you through the build process with a few easy steps. The single-file patch executable (patch.exe) is perfect for distribution by web, email, LAN, CD-ROM or DVD-ROM.

100% Data Integrity

You can rest assured that patches are only applied to files you've specified. Using a combination of 128-bit MD5 hashing, CRC-32 checksums and industry standard data encryption protocols, Visual Patch helps you keep your applications and data safe from unauthorized use, transmission errors and other threats.

Automatic Target Detection

Visual Patch makes quick work out of locating your target application. Using a flexible combination of starting-point techniques (registry keys and file/folder searches) and key-file identification, Visual Patch can quickly locate the application files and folders on the system that require patching. Definitively identifying the target folder greatly simplifies the patching process and is a key requirement for ensuring that your software is updated quickly and correctly.

Customizable User Interface

Featuring a library of over twenty different screen templates, Visual Patch makes it easy to control your project's user interface. There are pre-built layouts to handle just about any task you can imagine, and it's easy to adjust them to fit your needs exactly. You'll find everything from check boxes, radio buttons and edit fields to popular screens like license agreements and folder selection. The Screen Manager allows you to add and remove screens at will and adjust the sequence with a simple drag-and-drop motion. Each screen features a real-time preview so you can see the results of your changes as you work.

Dynamic Interactive or Silent Operation

Visual Patch supports the creation of both fully-interactive "wizard based" patches or completely automatic "silent" patches that operate without displaying user interface dialogs, prompts and other messages. You can choose to dynamically enable the silent mode through a command line switch, or configure the patch to always run in this unattended mode. Either way, patches made with Visual Patch are suitable for use in both consumer/standalone applications and corporate/network patch management solutions.

Powerful Scripting Engine

Visual Patch includes an incredibly powerful free-form scripting engine, giving you unprecedented control over your software patching system. This easy to understand scripting language features everything from “for, repeat and while” loops, to “if/else” conditions, functions, variables and arrays. Paired with the built-in action library, full mathematical evaluation and Boolean expressions, there’s simply nothing you can’t achieve. We’ve also built in an Action Wizard so even complete novices can create powerful projects that handle the most demanding patching tasks.

Comprehensive Action Library

To make the most out of Visual Patch’s scripting engine, we’ve included a library of over 250 high-level actions. With everything from registry editing to file copying to web file downloads, this complete scripting environment contains everything you need to automate complex tasks and handle even the most sophisticated software and system patching requirements. No other tool can match it!

Integrates with TrueUpdate

Visual Patch is an integral component of software lifecycle management. For a complete and fully integrated end-to-end solution, we’d recommend using Visual Patch in conjunction with Indigo Rose’s Setup Factory (software installation) and TrueUpdate (automatic updating/patch delivery); however you’re certainly not locked into doing so. If your company has standardized on other tools for software installation, patch deployment and patch management, Visual Patch will still add significant benefit, being a best-of-breed solution.

Network Patch Management

Visual Patch isn’t just for software vendors. It’s also an extremely valuable tool for use in corporate, government and educational IT departments. Visual Patch helps you get the most out of your network patch management and software management/SMS infrastructure. As a network administrator responsible for keeping hundreds or thousands of computers and servers up to date, being able to create your own bulletproof software patches is invaluable. Simply add them to your patch deployment solution of choice (TrueUpdate, SMS, etc) and manage them just like any other OS or application patch used on your network.

Trusted by Professionals

Thousands of companies trust Indigo Rose software tools. In fact our products, such as Setup Factory, TrueUpdate and AutoPlay Media Studio, are used to distribute and manage software on millions of customer and client systems around the world. Additionally, all of our products are backed up by world-class technical support services.

What's New in Visual Patch?

Vista Compatible

Visual Patch's design workspace and generated patches are compatible with Windows Vista, including a configurable "requested execution level" setting for the patch's manifest.

Custom Resource Stamping

Visual Patch allows you to use your own product icon and provides full control of the version information that you want written into the patch's resources.

Integrated Code Signing

Protect the integrity of your company and products by code signing your patches with your own certificate during the build process.

Improved Workflow

Visual Patch features an unbeatable development environment that puts *you* in control of your files. Each point release gets its own version tab – then you simply drag and drop your files and folders onto the project window and you're ready to build. Visual Patch is smart enough to maintain your folder structure, automatically query version resource information, determine what has been added, changed or deleted and ensures 100% data integrity with reliable MD5 hashing and CRC-32 checksums.

Sophisticated Binary Patching Algorithm

Visual Patch features a cutting-edge binary differencing engine. By way of a sophisticated byte-granular analysis of your file data, Visual Patch is able to extract only the "delta" between one file version and the next. Rather than having to distribute a full copy of changed files, Visual Patch will only package up the parts that have changed. The result is a significant reduction in patch sizes, more security, reduced transmission time and lower bandwidth costs.

Project Wizard Quick-Start

Spend five minutes with Visual Patch's easy to use Project Wizard and come away with a complete and ready to build project. You'll be walked through each option so you can get your project started as quickly as possible. The new wizard helps you to add versions, associate your files, configure the user interface, and much more.

Customizable Wizard Dialogs

Visual Patch is considerably more customizable than before. Included are over twenty different screen templates — easily adjustable to fit your particular needs — that handle just about any task your project requires. There's everything from check boxes, radio buttons and edit fields to popular screens like license agreements, folder selection and many more. The Screen Manager allows you to add and remove screens at will and adjust the sequence with a simple drag-and-drop motion. Each screen features a real-time preview so you can see the result of your changes as you work.

Powerful Scripting Engine

Visual Patch includes the same scripting engine as Setup Factory, TrueUpdate and AutoPlay Media Studio. Based on the popular "Lua" language, this all-new and incredibly powerful free-form scripting engine gives you unprecedented control over your software patching system. This easy to use language features everything from "for, repeat and while" loops, to "if/else" conditions, functions, variables and associative arrays. Paired with the built-in action library, full mathematical evaluation and Boolean expressions, there is simply nothing you can't achieve. We've also built in an "Action Wizard" and "Quick Scripts" feature so you can get right up to speed creating powerful projects to handle even the most demanding tasks.

Extensive Action Library

Visual Patch includes a built-in library of more than 250 powerful, yet easy to use actions. There are high-level actions to handle everything from text file editing to system registry changes. You can execute programs, call DLL functions, query drive information, manipulate strings, copy files, enumerate processes, start and stop services, interact with web scripts, display dialog boxes and much more. Whatever you need your software patch to do, Visual Patch can make it happen!

Easy to Use Action Wizard

You don't have to be a wizard to create powerful patching systems with Visual Patch. We've built the wizard into the software! Simply choose the action you want from a categorized list (complete with on-screen interactive help), fill in the requested information fields and the wizard does the rest. You don’t have to know anything about scripting or programming — just fill in the blanks, and you’re done! Making changes is just as easy. Click on the line you want to change and press the “edit” button to go back to the original form. It’s really that easy!

Professional Script Editor

If you’ve outgrown the Action Wizard interface or simply want to get the most out of the Visual Patch scripting engine, we’ve got you covered. The Visual Patch action editor features all of the professional features you’d expect. There’s color syntax highlighting, code completion, function highlighting, as-you-type action prototypes, Ctrl+Space function listings and even context-sensitive help. If you’re used to programming in Microsoft® Visual Basic, Microsoft® Visual C++ or any other modern development language, you’ll be right at home.

Publishing Wizard

Once you’ve got your project ready to go, the Publishing Wizard will help you package it up into a compact single-file executable. Compare our tiny ~500 KB runtime overhead to the competition and see for yourself — patches created with Visual Patch are smaller and faster than ever before. What’s more, the single-file patch executable is perfect for distribution by web, email, LAN, CD-ROM or DVD-ROM.

Themes, Skins and Backgrounds

Choose from dozens of pre-made themes (skins) for your screens or even make your own. It’s as easy as viewing a live dialog preview and picking your favorite style. You can configure everything from fonts (face, color, size, style) and banner images to body/background graphics, control colors (buttons, check boxes, radio buttons) and more. If you like, you can even customize the desktop background with gradients, images, color washes, headlines and footer text with 3D effects.

Fully Automatic Patches

Visual Patch supports the creation of both fully-interactive “wizard based” patches or completely automatic “silent” patches. Silent patches operate without displaying user interface dialogs, prompts and other messages, making them perfect for use in corporate networks and managed infrastructures. New options let you enable silent operation via a command line switch, or even configure the patch to always run in unattended mode. Visual Patch patches automatically return command line status codes and are ready for use in automatic deployment processes and corporate patch management consoles.

Patch Security

Visual Patch includes a variety of features designed to help you manage access to your software, including the ability to ask for a serial number or password. Visual Patch automatically restricts use of your patch files through the use of binary differencing, key files and sophisticated message digests calculations.

Expandable with Action Plugins

Visual Patch can be easily expanded with Action Plugins. These add on modules can extend the product in infinitely powerful ways, such as adding support for databases, XML, data encryption and FTP file transfers. Tight integration with the design environment, including IntelliSense style code completion and syntax highlighting, makes Plugins just as easy to use as built-in actions. Plugins are available through Indigo Rose as well as third-party developers thanks to Indigo Rose’s freely available plug-in development kit.

International Language Support

Visual Patch gives you everything you need to support your customers and clients around the world. Patches created with Visual Patch can automatically determine the language of the client operating system and adjust the display of screens and messages appropriately. Whether you need to support English, French, German, Spanish, Italian or any other language recognized by Windows, simply provide the text and Visual Patch takes care of the rest!

Built-in Spelling Checker

Now it’s easier than ever to make sure that typos don’t creep into your projects. Just about anywhere you can type, you can perform a spell check to ensure error-free text. Dictionaries are available for over a dozen languages including English, French, German, Italian, Spanish, Dutch, Swedish, Danish, Croatian, Czech, Polish and Slovenian.

Reports and Logs

Keeping track of the essential details of your patching project is now just a couple of clicks away. With improved HTML-based project reports (featuring CSS formatting) and text-based run-time log files, you’ll have an accurate record of everything you need. New options let you control the level of detail being logged, including options for recording errors and script execution details.

Unattended Builds

Visual Patch fits seamlessly into your automated build processes. Software development teams and network managers will appreciate features such as build constants and pre/post build processes. You’ll find that Visual Patch projects integrate right into your daily builds.

Works with Windows 95 and Up

Patches created with Visual Patch work just fine on every Windows operating system from Windows 95 to Vista and beyond. Compare that to competitive tools and you’re sure to be surprised at their requirements. If you need to support legacy systems, your choice is clear!

Frequently Asked Questions

Why use Visual Patch?

Visual Patch simplifies your product management and makes it easy to manage your software releases. There is no easier way to make professional quality, full-history patches for your software and other electronic content. Unlike other products, Visual Patch combines sophisticated binary patching with the flexibility offered by full-history patches. These patches can update any older version to the latest release with a single executable patch file.

Who needs Visual Patch?

Anyone who needs to create compact and secure software patches needs Visual Patch. This includes software developers, network administrators and IT managers, among others. Regardless of the type of data being distributed – executables, documents, databases, videos, etc. – Visual Patch can figure out what files have changed, the exact changes within each file and how to update any previous version to the current version.

How easy is it to learn?

Visual Patch’s point-and-click design takes the difficulty out of building even the most sophisticated full-history patches. With the same ease of use and interface style that has made Setup Factory famous, Visual Patch makes it easier to get from “no patch” to “patch” than ever before.

How does Visual Patch affect my bottom line?

Distributing smaller files is good for both you and your customers. Visual Patch will help you save money on bandwidth, server hardware and network congestion. Your customers and clients will benefit from faster downloads, a reliable and easy to use patching process and increased satisfaction with your product and company.

Does Visual Patch support binary patching?

Yes. Visual Patch features state-of-the-art binary differencing and compression algorithms. Combined with a powerful full-history patching engine, Visual Patch offers a unique approach unequaled by any other product, making it an excellent addition to the toolbox of any software developer or IT manager.

Does Visual Patch support full history patching?

Yes. Visual Patch gives you full control over which versions of your software can be patched with a single executable. Unlike some other products, Visual Patch's full-history patching ability can update any version of your software to the latest release. This makes patches created with Visual Patch easier to apply and friendlier to use than those created with less sophisticated patch builders.

Can I password protect my patches?

Yes. Visual Patch includes a number of features designed to help you manage access to your software. Asking for a password or serial number is no problem. Additionally, Visual Patch automatically restricts use of your patch files through the use of binary differencing, key files and sophisticated message digests calculations.

Will my patch work on all Windows platforms?

Visual Patch builds patches with 100% support for all 32-bit Windows platforms. This includes Windows 95, 98, ME, NT4, 2000, XP, Vista and Server 2003.

What languages can my patch appear in?

Visual Patch gives you everything you need to support your customers and clients around the world. Patches created with Visual Patch can automatically determine the language of the client operating system and adjust the display of screens and messages appropriately. Whether you need to support English, French, German, Spanish, Italian or any other language recognized by Windows, simply provide the text and Visual Patch takes care of the rest!

Why not just use Setup Factory to build a new installer?

When preparing a new release, you will certainly want to use Setup Factory to build an installer for new customers; however, you can save money and ensure the security of your updates by using patch files. Patch files are smaller as they only distribute the files that have changed and – more importantly – they are absolutely useless to anyone who doesn't already have a valid version of your software installed.

What's the difference between an Installer builder and a Patch builder?

Installer builders (such as Indigo Rose's Setup Factory) have a different purpose than patch builders (like Visual Patch). A software installer is used to setup and configure a full software application on a computer system. Once it's installed and working, the installation program's job is done. In a perfect and unchanging world, that would be the end of the story. However, we all know that change is a constant. Software and data need to be updated periodically (bug fixes, features etc), but such changes seldom require the complete overhaul of a program. You need a method that synchronizes the data "in-the-field" with your current release.

Visual Patch greatly simplifies product maintenance by taking care of this for you. Whenever you have a new version ready, Visual Patch figures out what files need to be added, changed or replaced to bring any older version of your software up to date.

Visual Patch saves you time by making the design process as easy as possible. Visual Patch is also incredibly accurate; by automating much of the decision process, it eliminates opportunities for human error compared to developing patches "by hand" with other methods. Also, with the addition of binary differencing, Visual Patch can actually determine the areas of difference within a file and only distribute those particular changes – something that is impossible to do without sophisticated algorithms.

Will my patch be Internet ready?

Yes. Visual Patch generates a compact, single-file, self-executing patch that is easy to distribute, and easy for your users to use. It's perfect for distribution using web, email, LAN, TrueUpdate, CD-ROM or DVD-ROM. It's also Authenticode-ready, so you can digitally sign your patches.

What other tools do I need to use Visual Patch?

None. As a standalone product, Visual Patch can be used by anyone. It doesn't matter what installation product you use or even whether you use one at all. If you need to get newer versions of files out to users, Visual Patch will do it. However, as part of a complete solution for software deployment, you will find that Visual Patch integrates quite nicely with tools like Setup Factory and TrueUpdate.

What kinds of files can Visual Patch update?

Visual Patch will work with any kind of files. You could even use it to update a few slides in a presentation, or individual files in a library of help documents used by your sales team. With a flexible tool like Visual Patch, the possibilities are endless.

Can I customize the runtime interface?

Yes. Visual Patch lets you easily drop in new wizard dialogs using the built-in Screen Manager and Gallery. Altering the display sequence is as simple as clicking on up and down arrows. You can edit text messages, use custom graphics and set conditional display options. A variety of screen types are available, from basic text displays, to check boxes, text input, radio buttons and more.

Can I create patches that target multiple operating systems?

Yes. You can attach conditions to any part of the patch to make them specific to the version of Windows that the user is running. For instance, your patch might install some new files only on Windows 95, or it might check different Registry locations for values if the user is running Windows 98 or Windows XP.

Can I distribute my patch on CD-ROM / DVD-ROM? Email? Web?

Yes. Visual Patch creates standalone, single-file executable patches that you can distribute using virtually any type of media you like.

Can Visual Patch handle advanced patching needs?

Absolutely. With Visual Patch, you aren’t limited to just replacing old files with new ones. We’ve also included many advanced features – you can query the Registry, modify INI files, perform file searches, interact with web scripts, explore folders, delete and rename files, and more.

How does Visual Patch benefit the software developer?

The easier it is for your users to update your software, the more likely it is that your users will be using the latest version. As a result, your technical support team will have fewer legacy issues to deal with. The easier it is for *you* to release updates, the more often you can release them. You won’t have to hold back releases until you have made enough changes to justify the effort required to prepare updates using traditional update methods.

How does Visual Patch impact technical support?

Timely software patches allow your users to benefit quickly from any new features and bug fixes you develop. Ensuring that users benefit from all the bug fixes you've released reduces the incidence of support calls. Keeping users up to date makes it easier to support them when incidents occur.

How will Visual Patch impact our customers and clients?

Today's users are savvy; they demand responsiveness from software companies and they want tools that meet their needs and make them more productive. In order to maintain customer loyalty and maximize the user's experience with your software, you need to make patching your software as easy as possible. Making it easy for users to patch your software shows that you're committed to supporting it.

How does Visual Patch benefit the network administrator?

Keeping a corporate, educational or government network up-to-date with the latest security patches, applications updates and operating system fixes is a time consuming ordeal. Without tools like Visual Patch, the task is virtually impossible. Used in conjunction with Indigo Rose's TrueUpdate, you'll be able to quickly and effectively roll out whatever software patches you need to throughout your organization. The TrueUpdate client software can analyze the computer system, decide what is currently installed and then take action to download and install the patches you've made with Visual Patch to bring that system up-to-date. It's fast, easy and automatic.

I'm not a developer...do I still need Visual Patch?

Absolutely! You don't need to be a software developer to benefit from Visual Patch. You can use Visual Patch to update product catalogs, databases, price lists, help files, quarterly reports, training videos or whatever else you want.

About this Guide

This user's guide is intended to teach you the basic concepts you need to know in order to build a working software patch. You'll learn the ins and outs of the program interface and how to perform many common tasks.

The guide is organized into 10 chapters:

Chapter 1:	Understanding Visual Patch
Chapter 2:	The Project Wizard
Chapter 3:	The Development Environment
Chapter 4:	Versions and Files
Chapter 5:	Creating the User Interface
Chapter 6:	Actions, Scripts and Plugins
Chapter 7:	Session Variables
Chapter 8:	Languages
Chapter 9:	Building and Distributing
Chapter 10:	Scripting Guide

Each chapter begins with a brief overview and a list of the things you will learn in that chapter.

Document Conventions

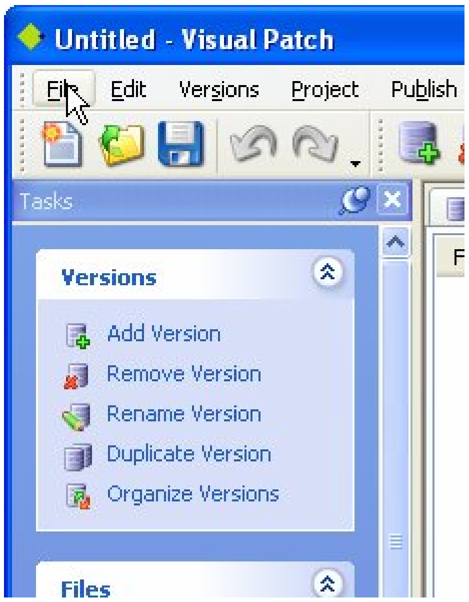
This user's guide follows some simple rules for presenting information such as keyboard shortcuts and menu commands.

Keyboard Shortcuts

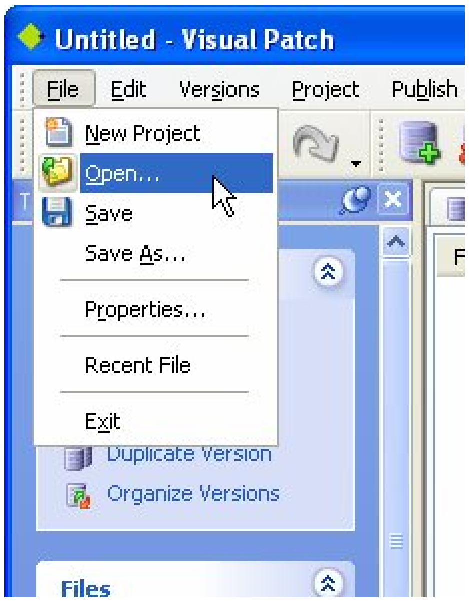
Keyboard shortcuts are described like this: press Ctrl+V. The "+" means to hold the Ctrl key down while you press the V key.

Menu Commands

Menu commands are described like this: choose File > Open. This means to click on the File menu at the top of the Visual Patch program window, and then click on the Open command in the list that appears.



Click on the File menu…



…and click on the Open command

Typed-In Text

When you're meant to type something into a text field, it will be presented in italics, like this: type *"Visual Patch makes software patching easy"* into the Message setting. This means to type in "Visual Patch makes software patching easy", including the quotes.

Chapter 1:

Understanding Visual Patch

Patching software from one version to another is a sophisticated process.

Visual Patch is designed to make the patching process as simple to understand as possible. It takes care of most details for you, such as inspecting your versions to decide which files have changed, and analyzing the individual files in each version in order to extract the differences between them.

However, there are some important concepts that you will want to understand before you begin designing your first patch. Some of these, like the concept of key files, are unique to Visual Patch and are central to how the patching system works.

This chapter will discuss the nature of patching and explain some of the concepts you need to know in order to use Visual Patch effectively.

1

In This Chapter

In this chapter, you’ll learn about:

- Patches
- The benefits of patching vs. distributing a full installer
- Other tasks that Visual Patch can handle
- Binary patching and whole-file patching
- Different patching strategies, such as full-history patching
- Versions and version tabs
- Managing your versions
- What we mean by “installed version” and “target version”
- Version detection (how the installed version is found)
- The application folder, and the three standard detection methods used to find it
- Key files
- MD5 fingerprints
- How the patch handles unrecognized files
- Version numbering

What is a Patch?

A patch is a file that, when run, modifies or replaces specific files on a computer system, usually to bring an already-installed software product up to date.

Patches are used every day to fix problems, add features, solve unforeseen compatibility issues, and fix security holes. They can be used to update all kinds of files: software executables, word documents, satellite images, medical databases, ocean maps, game data files, even parts of the operating system itself.

Because they only contain the data that has changed, patches are also used to transmit changes to very large files as efficiently and securely as possible.

Benefits of Patching

The role of patches in the software deployment cycle is to get already-installed software up to date after it becomes outdated. Patching technology offers numerous benefits over simply redistributing new versions of the original software in whole form.

Smaller file size

Because they only contain the data that has changed from one version to another, patches can be much smaller than a full software installer needs to be. Especially in situations where large data files are involved, the savings are often dramatic—patches that are less than 1% of the original file sizes are possible.

Reduced traffic

Smaller file sizes translate into reduced bandwidth costs, and reducing the amount of traffic leaves more bandwidth for other services.

Faster transmission speeds

Having less data to transmit means that updates can be sent and received faster, which means less time is spent waiting for updates.

Security

The best way to protect information during transmission is to never transmit it in the first place. By only transmitting the data that has changed, patches reduce the risk of third-party interception. Even if some hypothetical future technology made it possible to “crack” the encryption methods used to package the changes, the unchanged data would remain safe.

Integrity

A patch can’t update something that isn’t there. If a user doesn’t already have your software installed, they won’t be able to apply the patch. And if someone is using a modified version of a file, that file won’t be updated—unless you expressly permit it when you design your patch.

What Can a Patch Do?

The basic role of a patch is to modify or replace files so they match the files in a target version of your software. It also might need to back up or remove any *legacy files*, i.e. files from previous versions that no longer exist in the current version.

But Visual Patch isn’t limited to just updating files. With a built-in scripting engine containing over 250 high-level actions, it can also perform many other tasks involving everything from Registry changes to HTTP downloads.

With a little bit of scripting, your patch application can handle any advanced task you need it to: copying files, modifying INI files, starting and stopping services—even calling external DLL functions.

This ability to perform system changes in addition to the basic file updating is an important and valuable feature of Visual Patch.

Tip: If you use Setup Factory to build your software installer, you can use actions to add new files to the uninstall control file so they will be removed along with the original files when the user uninstalls your software via the control panel.

Patching Methods

There are two basic methods that can be used to update a file: binary patching, and whole-file patching.

Binary Patching

Binary patching or “delta compression” involves analyzing two versions of a file in order to extract only the data that has changed. The same changes can then be applied to any file that matches the old version, in order to “transform” it into the new version.

Creating a binary patch involves performing a byte-by-byte comparison between the original file and the new file, and then encoding the differences into a *difference file*. Each difference file contains the actual bytes that are different in the new file, along with a number of instructions that describe which bytes need to change, and which bytes are the same. This information is said to be *encoded* into the difference file.

Tip: The term “difference file” is often shortened to “diff file” or just “diff.”

When the patch is applied, the difference file is *decoded*, and the instructions are used to build the new file by copying the “unchanged” data out of the old file, along with the “changed” data that was encoded into the difference file.

For example, given an old file “A” and a new file “B,” a binary patching engine would compare A to B and then produce a difference file; let’s call it “AB.diff.” Once the difference file is created, you can use it to create the B file from any file that matches the A file. In fact, the binary patching engine could recreate B using A and AB.diff.

Because binary patching only stores the parts that have changed, the difference files can be very small—often less than one percent of the new file’s size. The size of the difference file depends entirely on how much data has changed between the two versions.

Each difference file can update a single, specific version of a file to another single, specific version of that file. The encoded instructions in the difference file are only valid for a file that is a perfect match of the original source file. Note that binary patching cannot be used to update a file if it has been modified in any way.

For patches that need to update multiple files, the patch executable will need to contain a separate difference file for each file that needs to be updated. So, for example, to update a single file from version 1.0 or 1.1 to version 1.2, using a single patch executable, it would need to contain one difference file to go from 1.0 to 1.2, and another to go from 1.1 to 1.2.

In most cases, the difference files are so small that you can fit a lot of versions into a single patch executable and still use less space than you would by just including the whole file, as in whole-file patching (see below).

Note: Visual Patch will automatically switch from binary to whole-file patching on a file-by-file basis whenever the total size of all the difference files surpasses the size of the whole file.

In some cases, encoding the differences between the two files results in a binary patch that is larger than just compressing the new file, for example into a .zip archive. This generally means that there are very little similarities between the two files; in other words, the two files are so different that it is difficult to reuse any of the data in the old file. This can sometimes be the case for files that are already highly compressed, or files that are encrypted. Visual Patch is able to detect these cases and will choose whichever patching method provides the best results.

Whole-File Patching

Whole-file patching operates on a different principle. Instead of only containing the parts that have changed (as binary patches do), whole-file patches just copy the entire file. The “patch” is just a copy of the new version.

Whole-file patches can be faster to apply, because they don't have to search through the original file in order to copy the parts that haven’t changed to the new version. They just overwrite the old file with the new one. The downside, of course, is that whole-file patches tend to be much larger than binary patches.

There are, however, two situations where whole-file patches can actually be smaller: when creating a single patch file that is able to update *many* different versions, and when the files being patched are too dissimilar.

Visual Patch always chooses the patching method that produces the best results. It automatically switches between binary patching and whole-file patching on a file-by-file basis in order to produces the smallest patch possible for your project.

Patching Strategies

Although Visual Patch chooses the right patching method for every situation, it’s up to you to choose an overall patching strategy. The three general strategies that you can choose from relate to the three different “kinds” of patches you can create.

Visual Patch supports three general patching strategies: incremental patching, multi-version patching, and full-history patching.

Incremental Patching

An incremental patch is a patch that is able to update a single, specific version to a single target version. For example, a patch that is able to update version 1.3 to 1.4, and only 1.3 to 1.4, is an incremental patch. Similarly, a patch that is able to update version 1.0 to 1.4, and only 1.0 to 1.4, is an incremental patch.

Incremental patches take full advantage of binary patching. Each patch only needs to contain a single difference file for each file that has changed. This eliminates any unnecessary data in the patch. For example, why bother sending the data needed to update 1.0 to 1.4 if the user has 1.3 installed? Because an incremental patch is targeted at a specific version, it only needs to contain the information needed to update *that* version, and nothing else.

This is especially true for incremental patches that update two *consecutive* versions. Although there may be many changes over the entire history of a software product, the changes between any two consecutive versions are typically very small. For example, if there are files that changed from version 1.2 to 1.3, but these files didn’t change from version 1.3 to 1.4, an incremental patch to go from 1.3 to 1.4 doesn’t need to contain any data for the files that changed from 1.2 to 1.3. This minimizes the amount of data that needs to be included in the patch.

Incremental patching generates the smallest and most secure patches possible.

Multi-Version Patching

A multi-version patch, as the name implies, is a patch that is able to update multiple installed versions to a single target version. For example, a patch that is able to update versions 1.2 ***and*** 1.3 to 1.4 is a multi-version patch.

Multi-version patches are larger than incremental patches. The more versions that a patch supports, the more information it needs to contain. This increases the amount of redundant data within the patch.

When a user runs the patch, they are only interested in updating a single version: the one they currently have installed on their system. All of the other versions that a multi-version patch supports are just excess baggage *for that user*.

The benefit of multi-version patches is that they are simpler to coordinate. A single patch file can be used to update multiple versions. Your users have fewer patches to choose from, and you have fewer patches to distribute. If there are 15 different versions of your software in the field, you would need 15 incremental patches to support them all. Using multi-version patches allows you to support all 15 potentially installed versions with fewer patches.

Full-History Patching

A full-history patch is able to update every previous release of your software up to a single target version. It is essentially a multi-version patch for *every* version of your software.

Full-history patches are the simplest patches to coordinate since the same patch file can update all versions of your software. Your users don't have to know what version they currently have in order to choose the correct patch. You only have to provide a single download that will work for all of your users. It's simple, straightforward, and uncomplicated.

However, full-history patches are the largest patches you can produce. In some cases they can even approach or surpass the size of a full install. For this reason, you will want to weigh the benefits of full-history patching vs. the other patching strategies.

Finding the Right Balance

Each patching strategy has different benefits and limitations. You will need to choose a combination of strategies that provides an appropriate balance between file size and logistical simplicity.

Since a lot depends on how many versions you need to support, and what methods you use to distribute your patches, there is no single "right way" to handle everything. The frequency of your updates is another factor that can determine how "up to date" your users are. For instance, if you release new versions often, and you don't use an automatic updating technology like Indigo Rose's TrueUpdate, the chances are higher that several of your users will be more than one version behind.

You will probably want to use a combination of incremental and multi-version patching in order to get the benefits of both. One strategy that works well is to use two separate patches:

- An incremental patch to go from the previous version to the current version
- A multi-version patch for all of the other versions

The assumption is that if most of your users always stay up to date, you will save a lot of bandwidth by providing the incremental patch. This is especially true if you use technology like TrueUpdate to keep your users up to date automatically.

Even if your patch distribution isn't automated—for example, if the users just click on a download link and run the patch themselves—this approach provides a good balance between minimizing patch size and making things less complicated for the user (by giving them fewer patches to choose from).

If you *are* using a tool like TrueUpdate, you might want to provide even more patch files, and let TrueUpdate decide which one to download and run.

For example, if you're about to release version 1.29 of your software, and you know that most of your users are using 1.28 and 1.27, it would make sense to create two incremental patches: one for the users of 1.28 and one for the users of 1.27.

You might also want to create a few different multi-version patches, for example one for versions 1.20 through 1.26, and another for the really old versions 1.0 to 1.19.

Ultimately, the patching strategy you choose depends on the number and sizes of files you need to update. The key point is to consider which method makes the most sense for you, given the distribution methods available to you, and how comfortable your users are with the patching process.

Versions

A version is the collection of files and folders that makes up a single release of your software.

If your original release is version 1.0, then all of the files in that release—everything that gets installed onto the user's system—constitutes one version.

Each time you modify your software and release it to the public, you create a new version. For example, if your next release is version 1.1, all of the files in that release constitute another version.

Note that a version isn't just the files that have changed from one release to the next. Each version contains *all* of the files in your software from a specific point in time. It even includes all of the files that remain the same.

In essence, a version is a complete copy of your software from a specific point in its life cycle.

Version Tabs

Each version is represented in Visual Patch by a version tab on the project window. The version tabs are listed in increasing order, with the oldest supported version on the left and the newest supported version on the right.

Your project should have one version tab for every version of your software that you want to build patches for. Whenever you build the project, you will be able to select which of these versions you want that particular patch to support.

Each version tab contains a *file list* where you can add the files that belong to that version. Adding a new version to a project involves adding a new version tab, and then adding all of the files from that version into that tab's file list.

You should put all of the files from each version onto a separate version tab. So for example if you have version 1.1 and version 1.2, put all the files from 1.1 onto a "1.1" tab, and put all the files from version 1.2 onto a "1.2" tab.

Note: Make sure you don't have any "empty" tabs, or Visual Patch will report an error when you build your project.

For more information on version tabs, see chapter 4, *Versions and Files*.

Version Management

Since each version tab needs to reference the files that belong to that version, you need to keep a copy of each release of your software.

Each version of your software should be stored in a separate location on your system. A good way to organize your versions is to keep them in separate folders, with each folder named according to the version it contains. Each folder should contain a complete copy of your software, with all internal subfolders intact.

This is a bit different from Visual Patch 1.0, which was able to take “snapshots” of each version. All it ever needed to know was the digital *signatures* of the files in the old versions. Because it only provided whole-file patching, it only needed access to the files from the latest version in order to build.

Visual Patch 3.0 on the other hand uses delta compression to create even smaller patch files. This binary differencing engine needs to analyze the entire contents of each file in each version in order to build a binary patch that contains only the data that has changed from one version to another.

The Installed Version

The version that the patch application detects on the user’s system is known as the *installed version*. It’s the version of the software that was installed by the original software installer.

For example, if the user has version 5.8 of your software installed on their system, and the patch application successfully locates it, “5.8” is the installed version.

In some cases, there may be more than one version of a software program installed on a user’s system. In these cases, the installed version is the one that the patch application identifies as the version to update. Usually this will be based on some kind of information that was recorded on the system by the installer, for example a “current version” entry in the Registry.

The Target Version

The version that a patch application is designed to update an installed version *to* is known as the *target version*.

For example, if you create a patch to update version 5.8 to version 5.9, “5.9” is the target version.

Version Detection

Before a patch can begin updating, it needs to determine whether a compatible version is installed on the user's system. In other words, the first task that the patch must perform is to locate and identify the installed version of your software.

The location where your software is installed is referred to as the *application folder*.

The Application Folder (%AppFolder%)

The application folder is the folder where your software is installed on the user's system. Finding the application folder is very important—without it, the patch has nothing to update.

In Visual Patch, the search for the application folder is implemented using actions in the project's On Startup event. When you use the project wizard to start a new project, it automatically configures the On Startup script to handle this for you.

In screens and actions throughout the project, the application folder is represented by a session variable named %AppFolder%. Storing the application folder path in this session variable is the ultimate goal of any version detection method.

In the default action scripts, this is handled by the VisualPatch.CheckFolderVersion action. This action inspects the folder to determine whether it contains all of the key files for a compatible version. (For more information on key files, see page 41.)

If the folder meets all of the requirements and is recognized as a compatible version, the VisualPatch.CheckFolderVersion action stores the folder path into the %AppFolder% session variable.

Tip: For more information on session variables, see chapter 7.

Detection Methods

In addition to any "custom" methods you might implement, there are three standard detection methods that are used in Visual Patch. Each of these will be implemented for you by the project wizard in the form of an action script in the On Startup event.

By default these detection methods are designed to follow a specific sequence. Assuming all three methods are enabled, the sequence is to check the current folder, check a specific Registry key, and then perform a file search on the user's system.

Current Folder

The current folder method checks the folder that the patch is running in to see if it contains a recognizable version of the software. This is done by checking for specific key files in the folder.

This method is useful when your software is installed in more than one location on the user's system, and the user wants to control which instance of the software is patched. Since the current folder check is performed before the other detection methods, it allows the user to override the other two detection methods by copying the patch file into the folder where the installation they want to patch is installed.

If the current folder doesn't contain a recognizable version, the patch moves on to the next detection method.

Registry Key

The Registry key detection method attempts to retrieve the application folder path from a specific Registry key. This is the recommended detection method, since it is the fastest and most reliable way to locate the application folder.

In order to use this method, your software's installer needs to have written the application folder path into a Registry key so that it can be retrieved by the patch.

If an application path is found in the specified Registry key, the patch will verify that it points to a valid version of your software. As in the current folder method, it does this by confirming the MD5 signatures of specific key files in the folder.

If no path is found, or if the key files don't match, the patch will proceed with the next detection method (assuming it is enabled).

File Search

The file search method searches the user's system for a folder that contains a version of your software, by checking every folder for the existence of key files. The search ends when it finds a folder that contains all of the key files for a compatible version and the MD5 signatures prove that the key files are a perfect match.

Custom Actions

Since it's ultimately all done with actions, it's possible to use a completely different method to determine the folder where your application is installed. In fact, you could even write a script that just set %AppFolder% to a hard-coded path if you were absolutely certain that your software was installed at the exact same place on every system, and was never modified or installed incorrectly.

In the vast majority of cases, though, you will want to use the standard methods described above.

Key Files

Each version of your software usually includes one or more files that are unique to that version. For instance, as new features are added, your software's main executable might change, along with a help file and perhaps a few data files. Visual Patch refers to these "identifiable" files as key files.

Key files are used to locate and identify your software on the user's system.

Designating a file as a key file means that you want Visual Patch to verify its existence and its MD5 signature in order to fully identify the version it belongs to. If the key file doesn't exist, or its MD5 signature doesn't match, Visual Patch will consider that version not found.

Each release of your software must have at least one key file, but you can specify as many as you want. It's important to remember that every key file in a version must be found in order for that version to be identified. In other words, a user must have all of the key files from a given release installed in order for their version to "qualify." If you have four key files for a particular version and only three of them are found, the version on the user's system won't be considered legitimate. The same goes for a user with three key files from one version, and one from another. All the key files must match the original files from a single release absolutely.

It's also important to remember that each key file will be verified by its MD5 signature. Care should be taken to avoid selecting key files that are likely to change for legitimate reasons once they're installed. For example, if your software uses a database file that is constantly updated, that file wouldn't be a good choice for a key file because its MD5 signature will change. The key files on a user's system must all be present, ***and*** their MD5 signatures must all match the original values determined for those files at design time.

If the patch doesn't find a valid release anywhere on the user's system, the user won't be allowed to update their software.

Choosing Appropriate Key Files

Key files are usually files whose contents are unique to a single version. Visual Patch requires at least one unique key file per version in order to uniquely identify that version. The file names and paths don't need to be unique, but their *contents* do. In other words, you must designate at least one key file per version whose contents are different from every other version. Otherwise, the patch won't be able to tell that version apart from the others.

Key files must also not change after being installed or patched because Visual Patch relies on their MD5 signature for validation. Files that are normally modified after they are installed (for example, .ini files) should never be designated as key files. If a key file has been changed in any way from the original file that is referenced in your project, it will prevent the version from being identified.

Each version can contain as many key files as you like. In fact, it's a good idea to designate additional key files to help ensure a positive identification.

Tip: Good candidates for key files are executables, images, help files, PDF docs, readme files...anything that can be used to tell one release from another, but isn't expected to change once it's installed.

The best key files are files that change from one version to the next, such as the main executable for the software. As a matter of fact, having a main .exe file that is different from one version to another is the perfect example of a key file. It's a file that must exist (if the .exe isn't there, the software isn't properly installed). It's also usually different from one version to the next, even if all that changes is the version number or the text on the "Help > About" window.

Mission-Critical Files

Although at least one of the key files in a version needs to be unique to that version, they don't all have to be. You can also use the key file feature to perform validation on mission-critical files. For example, if your software contains a really important file whose existence and integrity must be checked, making it a key file will prevent the patch from proceeding if the file has been removed or modified.

Remember: a version will be identified only if all of its key files are there and they match the originals exactly.

MD5 Fingerprinting

Visual Patch calculates the MD5 fingerprint of each file in order to identify files that are the same and in order to detect whether two versions of a file are different.

The MD5 algorithm is a standard algorithm that is widely used to generate cryptographic signatures. It was developed by Professor Ronald L. Rivest of MIT. To quote RFC 1321, which describes the MD5 standard:

> [The MD5 algorithm] takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest.

An MD5 fingerprint is essentially a large number that is calculated directly from the entire contents of a file. If even a single byte within the file changes, the MD5 signature changes as well. For all practical purposes, no two files can have the same MD5 fingerprint unless their internal data matches exactly.

Unrecognized Files

Once the patch application has detected a valid version, it will update each file in the release on an individual basis. By default, Visual Patch will only update files that it recognizes. If a file that is part of the installed version doesn't match the original source file in your project, it will ***not*** be updated. Visual Patch will skip over the unrecognized file and continue with the rest of the patching process.

This behavior is both a security feature and a requirement:

- If whole-file patching is being used, preventing the file from being installed unless the user has a recognizably valid version is a security feature.
- If binary patching is being used, it is impossible to update the file unless its contents exactly match the original file.

You can override this behavior for individual files by enabling the "Force install" option in your project. For more information on this option, see page 91.

Version Numbering

The whole point of Visual Patch is to make it easier for you and your users to update your software. One part of this process is deciding on the version numbering scheme you will use to identify each release of your software.

Visual Patch allows you complete freedom in naming your versions. (You could call one version “George” and the next version “Henry” if you wanted to.) We recommend using an industry-standard version numbering scheme that will be more readily understood by your users. Whatever you decide on, here are some guidelines you might want to consider.

Give each release a number

Using numbers makes it easy for your users to identify the hierarchy between different versions of your software. If you name one version 1.0.3.2 and the next version 1.0.3.3, it’s readily apparent which version is the newer one.

Make the numbers mean something

Version numbering schemes like “version.revision.sub-revision” are popular because they allow you to make the magnitude of an update readily apparent in the version number itself. Going from 1.5.2 to 1.5.3 normally indicates a small change, like a bug fix. Going from 1.5.3 to 1.6.0 would indicate moderate changes, such as new features being added, or improvements to the program code. Going from 1.6.0 to 2.0.0 would be reserved for sweeping changes, like complete rewrites or a completely new interface design.

Aim for clarity

Try to avoid version numbers that might confuse your users. A prime example is a number like 1.10. Is this version newer or older than 1.9? That depends on the numbering scheme being used. The standard “version.revision” scheme makes 1.10 newer than 1.9, since it marks “the tenth revision of the first version” of the product. But many users mistake the version number for a fraction. Even more savvy users that are aware of the “version.revision” standard might wonder if you were as savvy as they are—enough software has been released using fractional notation to make it a difficult guess. One solution is to use double digits for each part, so that 1.9 becomes 1.09 and the ordering becomes readily apparent.

Don’t rely on file sizes or date stamps

Make it a habit to issue a new version number whenever you release a new version of your software. Don’t expect your users to identify versions based on changes in the file size or date stamp alone. Not all users will be able to determine this information easily, and date stamps may be subject to change as files are downloaded or copied.

Simpler is better (within reason)

Avoid using a numbering scheme that involves long awkward version names like “49823.B345.14231-A.” At the same time, avoid overly simple schemes that might limit your ability to release updates often.

Don’t go overboard

Your external version numbers don’t need to reflect the number of compilations your software has been through since the last version. They also don’t need to reflect how many failed attempts there were before a new feature started working. The version numbers your users see should only reflect the changes that are visible to them. Keep external version numbers “tight” between consecutive releases. (Releasing version 2.0.29 right after version 2.0.21 could have your users searching for nonexistent versions like 2.0.28 if they run into any problems with 2.0.29. If you must track recompilations internally, consider keeping your internal version numbers separate from the version numbers that your users will see.)

Be consistent

If your numbering scheme is “MajorUpdate.MinorUpdate.BugFix,” don’t start incrementing your “major update” number when you’ve only put out a bug fix.

Avoid unnecessary changes

Once you decide on a numbering scheme, stick to it. Switching from one scheme to another could be confusing for your users, especially if the new version numbers look similar to the old ones. If you really must switch, make sure you take steps to explain the changes to your users.

Tell your users what has changed

It’s always good to tell your users what new features or bug fixes a new version brings. A rich and detailed version history makes a good impression on your users, because it shows how much time and effort you’ve spent improving your product.

Chapter 2:

The Project Wizard

Every journey begins with a first step. In Visual Patch, this first step is easily accomplished through the Project Wizard. In this chapter we will walk you through opening Visual Patch for the first time and creating a project using the Project Wizard.

2

In This Chapter

In this chapter, you'll learn about:

- Starting a new project
- Adding versions through the project wizard
- Locating your installed software

Starting a New Project

Everything has to start somewhere. In Visual Patch, the design process starts with the creation of a new project.

A project is simply the collection of files and settings and everything else that goes into building a patch. A typical project will contain all of the files that you want to patch, some screens that inform or gather information from the user, and maybe a few actions to take care of any “extras” (such as storing the patched version number in a registry key for future patches to use).

The project file

Each file that you add to a Visual Patch project has individual settings that control where, when and how the file will be included in the patch at build time. Likewise, each screen that you can display has its own properties that determine everything from the text that appears on the various parts of the screen to the color of the screen itself at run time.

These settings are all stored in a single file called the ***project file***. The project file contains all of the properties and settings of a project and the list of source files that need to be gathered up each time the project is built.

The project file is automatically created for you when you start a new project.

When you start a new project, Visual Patch’s project wizard walks you through the first few steps of project creation. This helps you get your project started quickly without missing any of the basics.

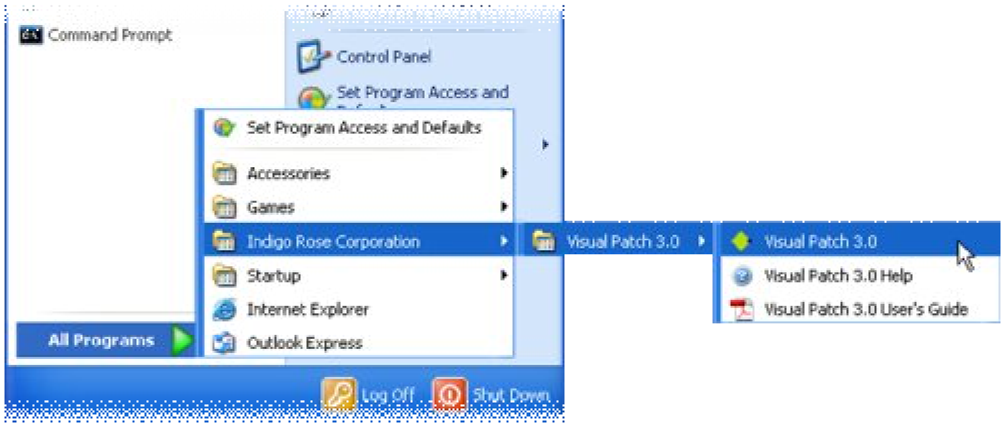
Let’s open the Visual Patch program and start a new project.

1) Open Visual Patch

Use the Start menu to launch the Visual Patch program.

By default, you’ll find Visual Patch under:

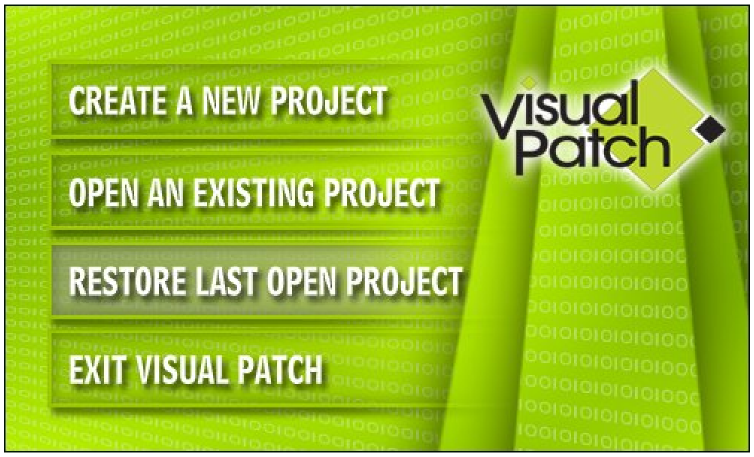
Start > Programs > Indigo Rose Corporation > Visual Patch 3.0



Start > Programs > Indigo Rose Corporation > Visual Patch 3.0

2) Create a New Project

The Welcome dialog appears whenever you run Visual Patch. It not only welcomes you to the program, it also lets you easily create a new project, open an existing one, or restore the last project you worked on. (Restoring the last project automatically opens the project you were working on the last time you ran Visual Patch.)

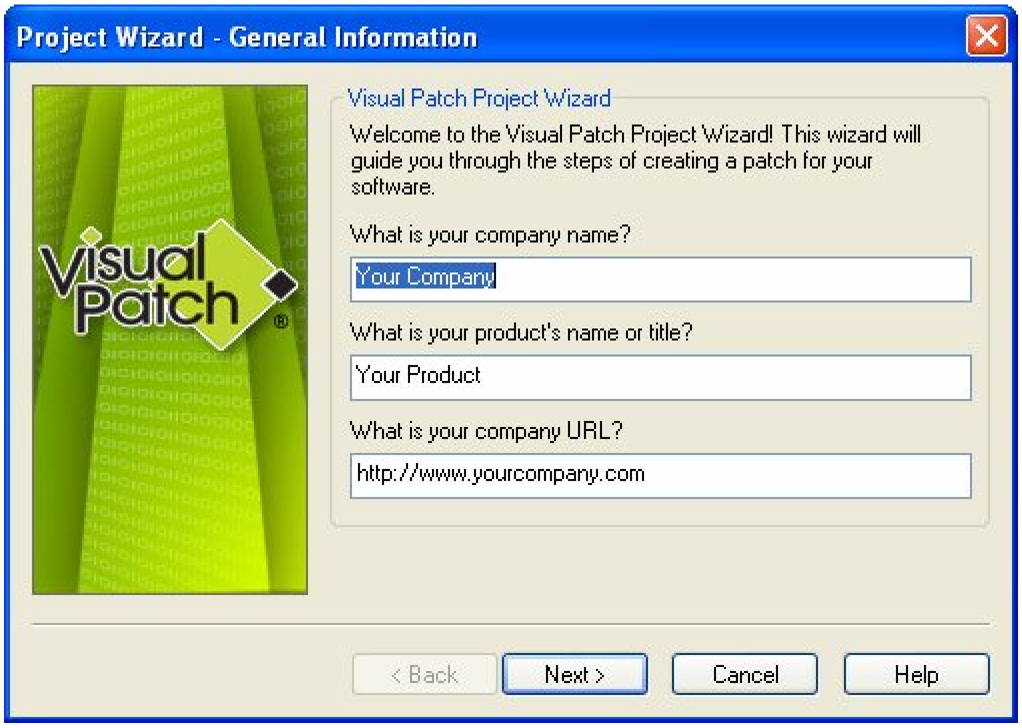


The Welcome dialog

When you click on “Create a new project,” the Welcome dialog closes and the project wizard appears.

3) Enter General Project Information

First, the project wizard asks you for three pieces of information related to your project. Simply enter your company name, product name, and your company URL in the appropriate fields.



Project Wizard - General Information

Visual Patch Project Wizard

Welcome to the Visual Patch Project Wizard! This wizard will guide you through the steps of creating a patch for your software.

What is your company name?

Your Company

What is your product's name or title?

Your Product

What is your company URL?

http://www.yourcompany.com

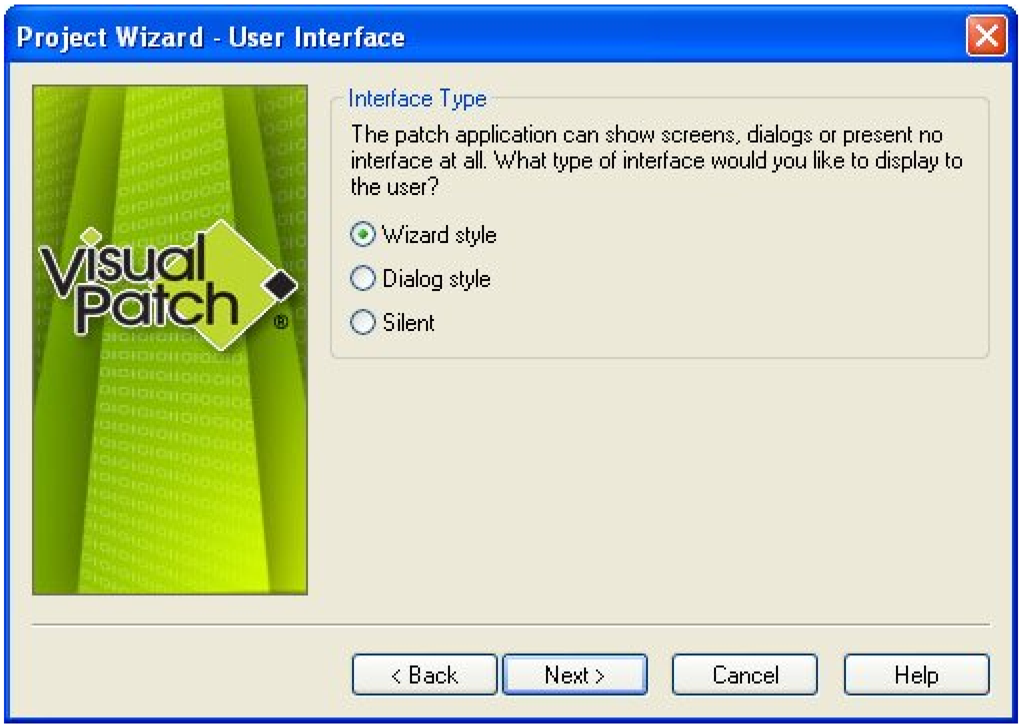
Insert your company name, product name, and company website

When you've entered all your information, click Next to move to the next step in the project wizard.

Tip: At any step in the project wizard, you can click Cancel to go straight to the program window with all of the default project settings untouched (i.e. to start with a "blank" project).

4) Choose an Interface Type

The next step in the project wizard is to specify which type of user interface your patch executable will use.



Interface Type

The patch application can show screens, dialogs or present no interface at all. What type of interface would you like to display to the user?

[✓ Wizard style] Dialog style Silent

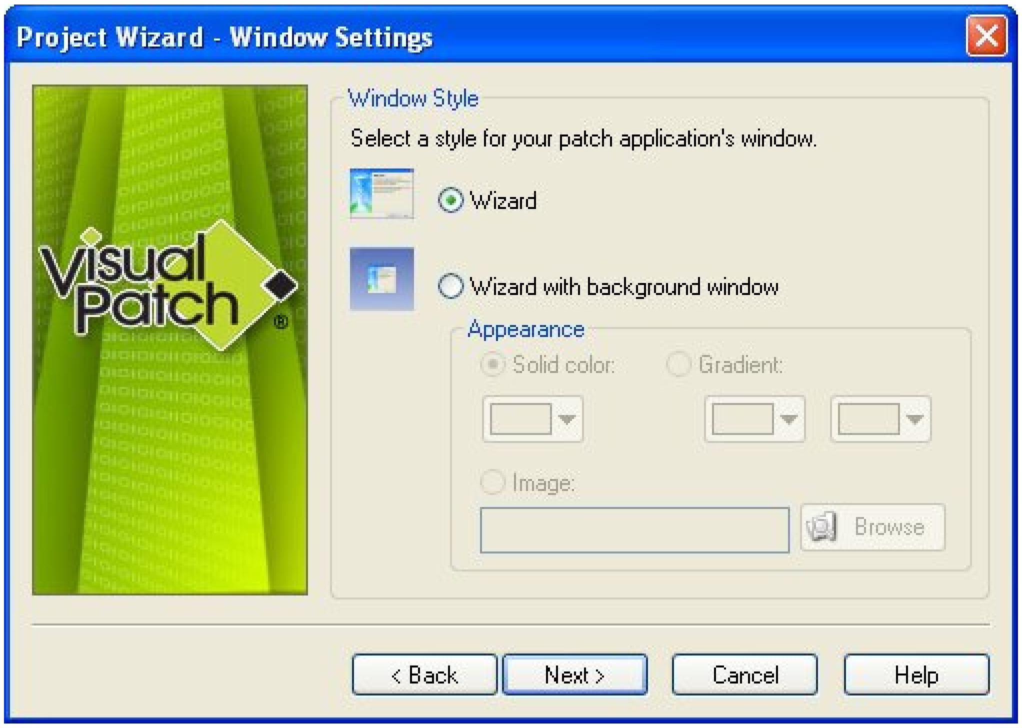
Select an interface type

The most common patch user interface is the default Wizard style. A Wizard style interface presents the user with a series of screens that they can navigate through by clicking Next and Back buttons. Wizard interfaces are considered very user friendly because they present and request information in discrete, guided steps, which makes the overall process easier for the user to understand.

The other two interface styles are Dialog and Silent. A dialog user interface uses popup dialogs or "message boxes" as opposed to screens to guide the user through the patch. A silent patch runs entirely in the background, and has no user interaction whatsoever. It's a great choice for unattended or automatic patching preferred by network administrators.

5) Pick a Window Style

This step determines whether or not your patch will have a background window covering the user's desktop.


Project Wizard - Window Settings

Window Style

Select a style for your patch application's window.

(●) Wizard

( ) Wizard with background window

Appearance

(●) Solid color: ( ) Gradient:

( ) Image:

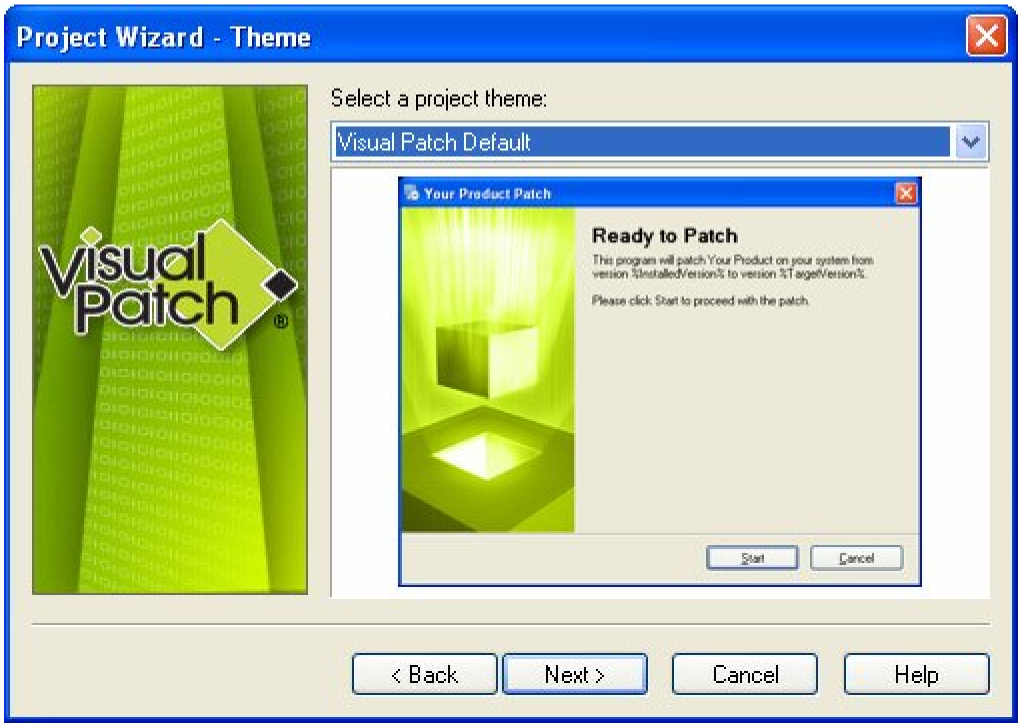
Choose a window style

Note: If you choose to have a background window, you can customize its appearance using the settings in this step.

If you select the background window style, you can specify how you want the background window to look. It can display a solid color throughout, a top-bottom gradient, or an image of your choosing.

6) Select a Project Theme

This step allows you to pick a visual theme for your wizard-style patch. Each theme applies a different overall appearance to the patch. (For more information on themes, see page 118.)



Select a project theme: Visual Patch Default

Ready to Patch

This program will patch Your Product on your system from version %InstalledVersion% to version %TargetVersion%.

Please click Start to proceed with the patch.

Select a theme for your patch executable

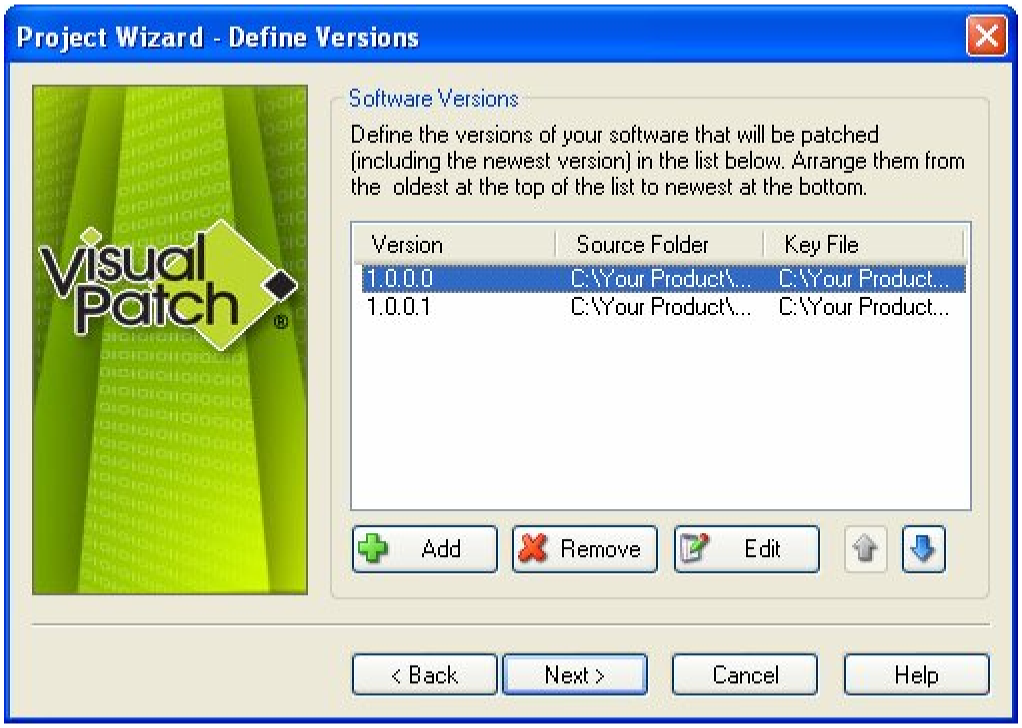
Once you've selected your project theme, click Next to proceed to the next step.

7) Define Versions

Use this step to add versions to your patch. Clicking on the Add button allows you to add a version to your patch. To make adding versions to your patch as easy as possible, store each version of your software in a separate folder on your development system. Then, using this step, add those folders one by one. For each version added, you can opt to add the folder recursively, which means that all files from that point downward in the directory structure will be added.

Once you have added a folder, you must specify a key file. This is the file that Visual Patch will use to determine if your software exists on the user’s system, and what version is already installed. It is important to select a file that is version specific, such as your program’s executable, or a DLL.

Note: You can skip this step and add versions after the wizard is completed. You can also make changes to these versions, or delete them entirely in the design environment. For more information on key files, see Chapter 1.



Define the versions of your software that will be patched (including the newest version) in the list below. Arrange them from the oldest at the top of the list to newest at the bottom.

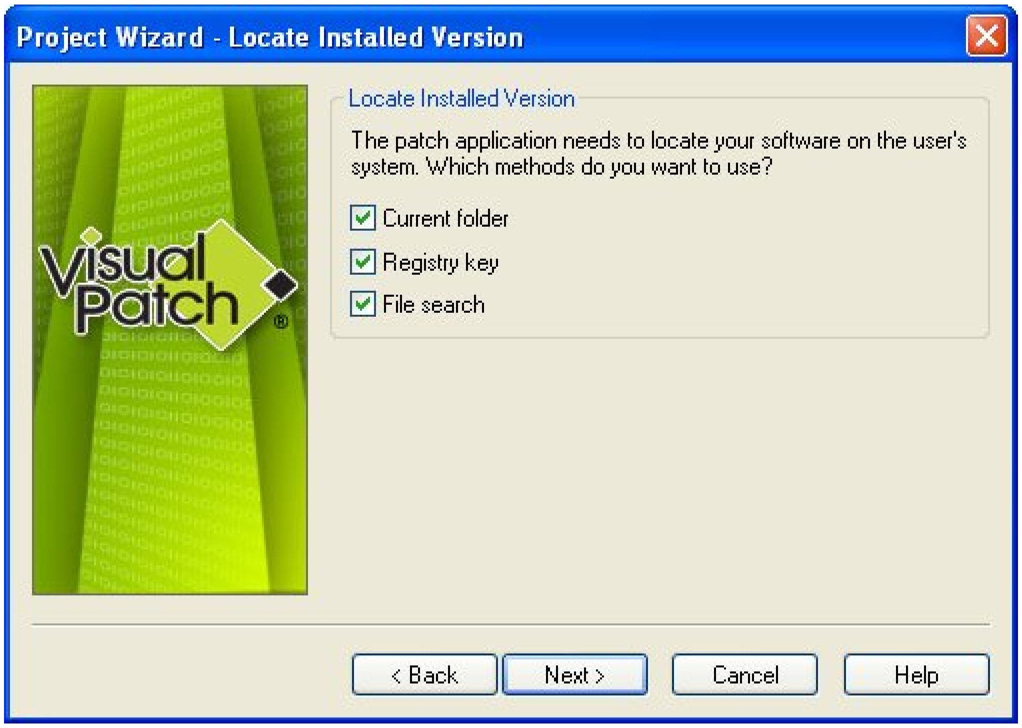
Version	Source Folder	Key File
1.0.0.0	C:\Your Product\...	C:\Your Product...
1.0.0.1	C:\Your Product\...	C:\Your Product...

Add your various product versions to the patch

Once you have added all of your product's versions to the wizard, click Next to continue.

8) Decide How to Locate your Installed Software

Visual Patch must be able to find your existing files on the user's system in order to patch them. There are a number of ways that Visual Patch can do this. Your patch executable can: search for the files in the location from which it was run, retrieve a value from the registry, and search either specific folders or entire drives on the user's system.



Project Wizard - Locate Installed Version

Locate Installed Version

The patch application needs to locate your software on the user's system. Which methods do you want to use?

☑ Current folder

☑ Registry key

☑ File search

Decide how the patch executable will locate your installed software

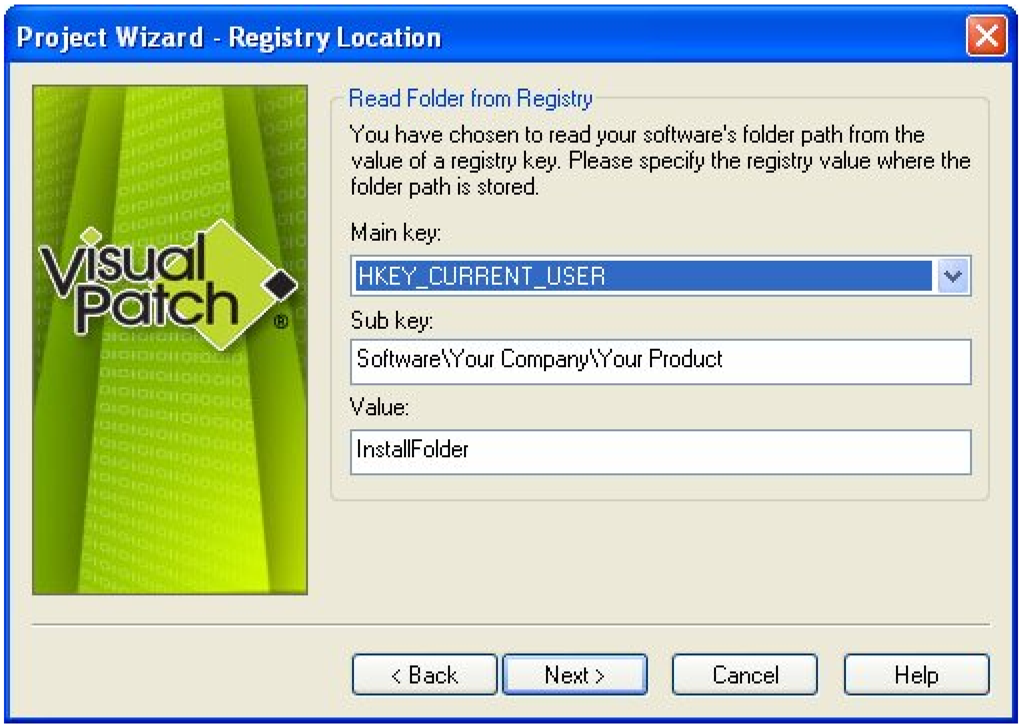
Enabling the Current folder option creates a "failsafe" patching method: if the user has your software installed, but the patch can't find it, you can direct them to copy the patch to the install location and run it. This option by itself, however, is not sufficient as many users have no idea where your software is installed.

Retrieving a value from the system Registry is by far the best locating method, though it does require some forethought on your part. In order to use this method, your software installer must create a registry value during installation. Then, it is a simple matter of configuring Visual Patch to inspect that registry key so the value can be retrieved. This is the most efficient method as accessing a single registry key is quick and not resource intensive.

If neither of the previous methods has located your software, a file search can be performed. This option scans the entire system looking for your software. It is a rather intensive and lengthy operation, but is an excellent choice of last resort.

9) Locate Using Registry Key

Visual Patch can retrieve your software's installation folder from a specific registry key. Use this step to specify the location in the Registry that contains the path where your software was installed.


Project Wizard - Registry Location

Read Folder from Registry

You have chosen to read your software's folder path from the value of a registry key. Please specify the registry value where the folder path is stored.

Main key: HKEY_CURRENT_USER

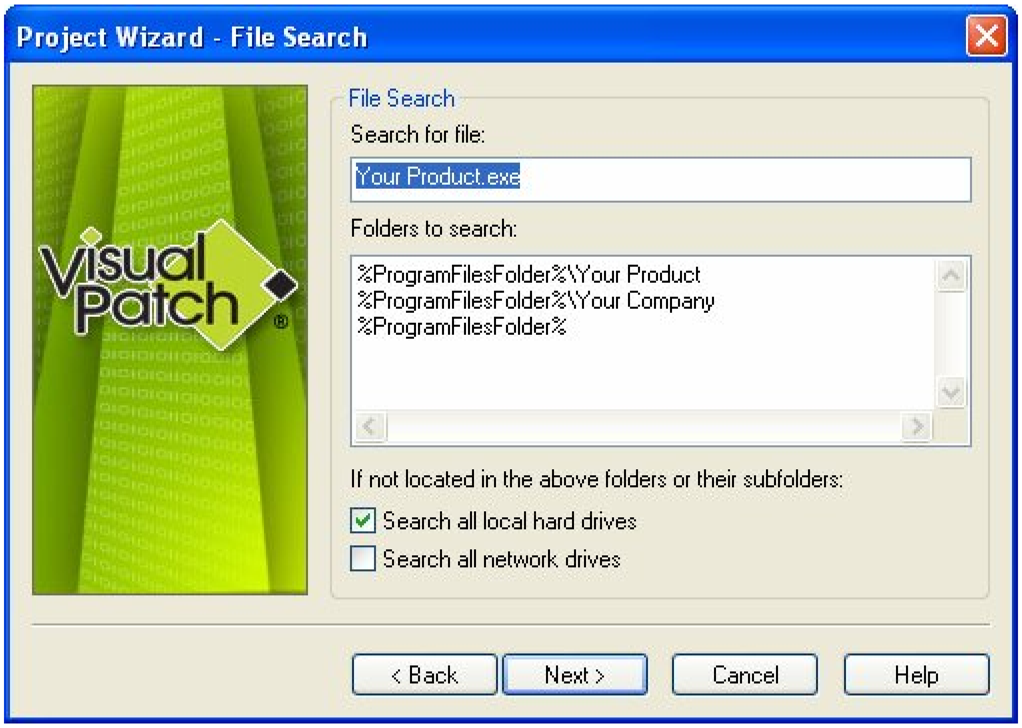
Sub key: Software\\Your Company\\Your Product

Value: InstallFolder

Specify which registry key and value contains the path to your installed software

10) Locate Using File Search

Visual Patch can search all or part of a user's system for key files. Use this step to configure specific locations to search, as well as whether or not entire local and networked drives should be searched if key files are not located in the specified folders.



Search for file: Your Product.exe

Folders to search:

%ProgramFilesFolder%\Your Product
%ProgramFilesFolder%\Your Company
%ProgramFilesFolder%

If not located in the above folders or their subfolders:

[✓] Search all local hard drives

[ ] Search all network drives

Specify a file to search for, and where it could be found

This option searches the user's system for a file known to exist in your product. If found, the software's version number is determined from key files located within the found file's path (a VisualPatch.CheckFolderVersion action is used).

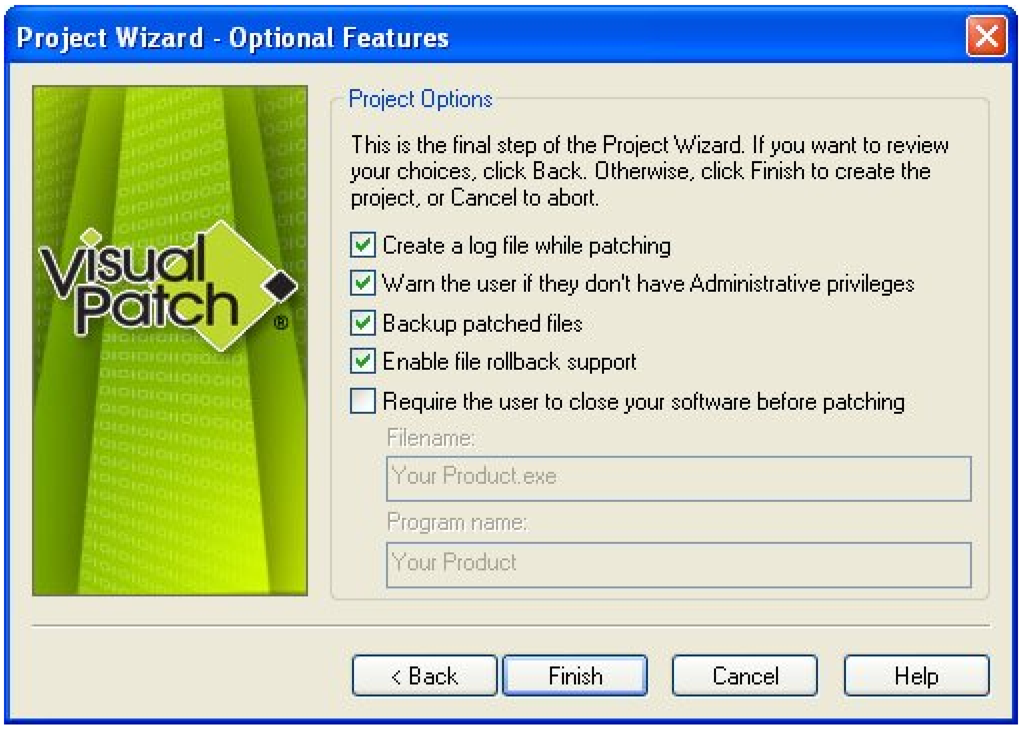
Note: Searching a user's system for a file can be resource intensive, especially if searching entire drives. This should occur only as a last resort. Retrieving a value from a registry key is much more efficient.

11) Select Optional Features

There are several optional features that are useful, though not required. Creating a log file while patching is an excellent tool for diagnosing potential errors that may arise. Additionally, warning the user if they do not have Administrative privileges is an excellent way to prevent patching problems resulting from the user not having the correct permissions.

In addition, requiring that the user closes your application before the patch will continue is an excellent way to minimize ‘locked file’ issues.

Enabling the Backup patched files option creates a backup copy of any file modified during the patching process. Enabling rollback support allows files modified to be ‘rolled back’ to their unmodified versions should an error occur while patching or if the user aborts the patching process.


Project Wizard - Optional Features

Project Options

This is the final step of the Project Wizard. If you want to review your choices, click Back. Otherwise, click Finish to create the project, or Cancel to abort.

- [✓] Create a log file while patching
- [✓] Warn the user if they don't have Administrative privileges
- [✓] Backup patched files
- [✓] Enable file rollback support
- [ ] Require the user to close your software before patching

Filename: Your Product.exe

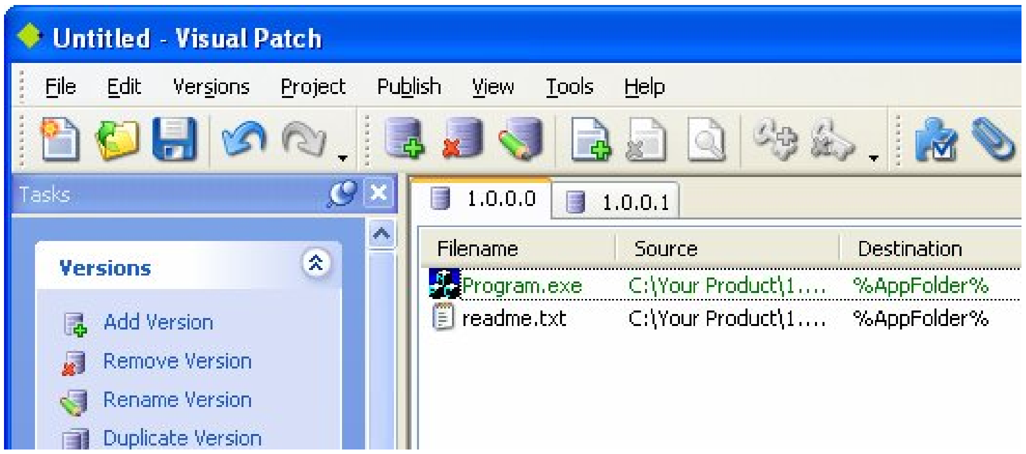
Program name: Your Product

Turn on or off various optional features

12) Click Finish to Create your Patch

After you click Finish, the project wizard will close and the design environment will appear, complete with scripts, screens, and folder structures configured with the settings you chose in the project wizard.

At this point, you could build the project and generate a basic patch for those files. Of course, you'd probably want to customize the screens before distributing it. To learn how to customize the screens in your installer, see Chapter 5.



Filename	Source	Destination
Program.exe	C:\Your Product\1....	%AppFolder%
readme.txt	C:\Your Product\1....	%AppFolder%

The Visual Patch design environment once the wizard is complete

Tip: Once you're in the design environment, you can start a new project by choosing File > New.

Chapter 3:

The Development Environment

This chapter will take you on a tour of Visual Patch's development environment. You'll learn how to use the features of the interface that allow you to create a comfortable and productive work environment, customized for the way *you* want to use the program. You'll also learn how to take advantage of Visual Patch's self-help resources, which are designed to answer any questions you might have while working with Visual Patch as quickly and efficiently as possible.

3

In This Chapter

In this chapter, you’ll learn about:

- The Visual Patch Program Window
- Toolbars
- Using the Task Pane
- Version Tabs
- Setting Preferences
- Updating Visual Patch
- Getting Help

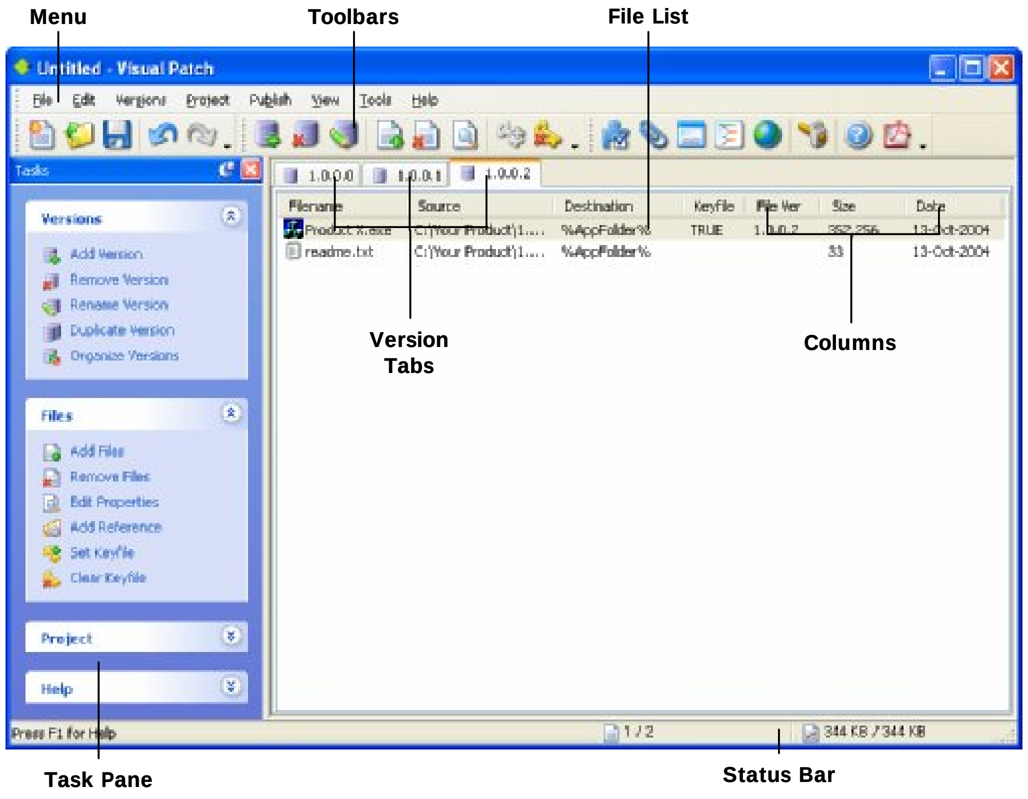
The Visual Patch Program Window

Now that you have started Visual Patch and created a new project, either through the Project Wizard or from scratch, it's time to get comfortable with the program interface itself.

The Visual Patch program window is divided into a number of different parts.

At the top of the window, just under the title bar, is the program menu. You can click on this program menu to access various commands, settings and tools.

Below the program menu are a number of toolbars. The buttons on these toolbars give you easy access to many of the commands that are available in the program menu.



Filename	Source	Destination	Keyfile	File Ver	Size	Date
Product X.exe	C:\Your Product\1....	%AppFolder%	TRUE	1.0.0.2	352,256	13-Oct-2004
readme.txt	C:\Your Product\1....	%AppFolder%			33	13-Oct-2004

Visual Patch's Design Environment

Most of the program window is taken up by the file list, which is where all the files in your project are listed. There is one tabbed list for each version of your software.

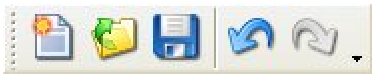
At the very bottom of the window, a status bar reflects your interaction with the program and offers a number of informative readouts.

The rest of the program window is made up of individual sub-windows known as *panes*. Each pane can be docked, tabbed, pinned, resized, dragged, and even made to float on top of the design environment. As well, panes remember their positions even after you unpin them. If you unpin a pane, and then pin it again, it will return to the position it had when it was pinned.

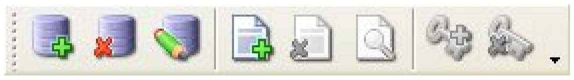
Tip: When you're dragging panes, it's the position of the mouse cursor that determines where the outline snaps into place. For example, to dock a pane below another one, drag the pane so the cursor is near the bottom edge of that pane. To "tab" one pane with another, drag the pane so the cursor is on top of the other pane's title bar.

Toolbars

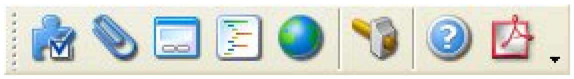
Toolbars in Visual Patch provide quick access to many of the more commonly used features. There are four toolbars available – Standard, Versions, Common, Filters – with the first three displayed by default.



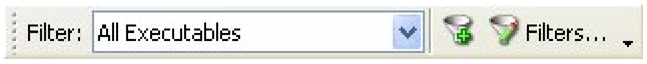
The standard toolbar



The versions toolbar



The common toolbar



The filters toolbar

Showing & Hiding

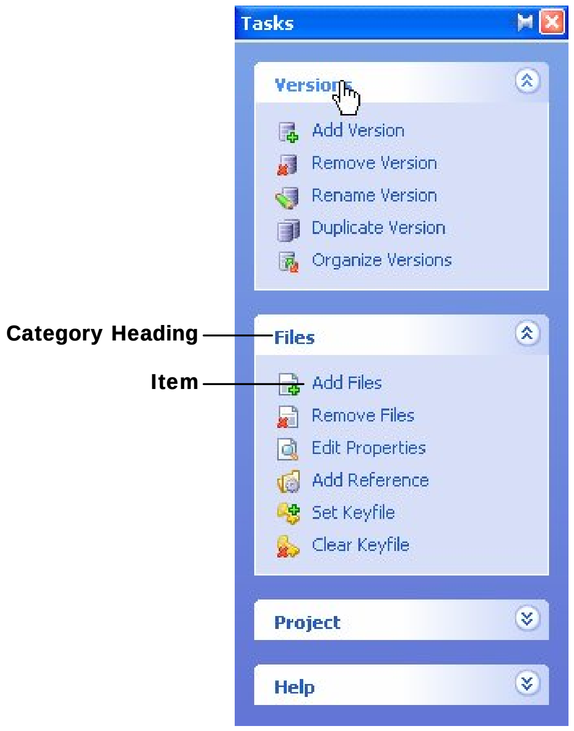
All four toolbars can be shown or hidden in the design environment. To show or hide a toolbar, simply right click on any toolbar, and click on the one you wish to change.

Customizing

If the default toolbar configurations are not fulfilling your needs, you can customize them. To customize the toolbars, right click on any toolbar and choose ‘customize’. Doing this opens the Customize dialog, where you can show or hide existing toolbars, create new toolbars, add or remove individual toolbar buttons, create keyboard shortcuts, change the toolbar themes, and adjust various other options.

Task Pane

The tall pane on the left is the task pane. The task pane provides easy access to the parts of Visual Patch you will use the most. It provides an alternative to the program menu for accessing the various parts of the Visual Patch design environment.



The Task pane

Version Tabs

In Visual Patch, your application's various versions are represented by version tabs on the project window. In a nutshell, every version of your software loaded into your Visual Patch project will have its own version tab.

Note: For more information about versions, files, and other related concepts see Chapter 1 and Chapter 4.

File Lists

Each version tab is linked to a separate file list. File lists contain all of the files and folder references associated with that particular version of your software, and provide you with a central location to manage its contents.

From this file list, you can change the properties of one or many files or folder references; you can change the source folders, destination folders, whether or not certain files are key files, and under what condition a file will be patched.

Columns

The file list is divided into many columns, allowing you to quickly locate relevant information. You can decide which columns are displayed by selecting View > Columns. From the same dialog, you can change their display order.

You can use the displayed columns to sort your file list; by clicking on a column header, you can sort your entire file list by the information contained in that column. For example, by clicking on the filename column, all files and folder references will be sorted by their filename.

Filters Toolbar

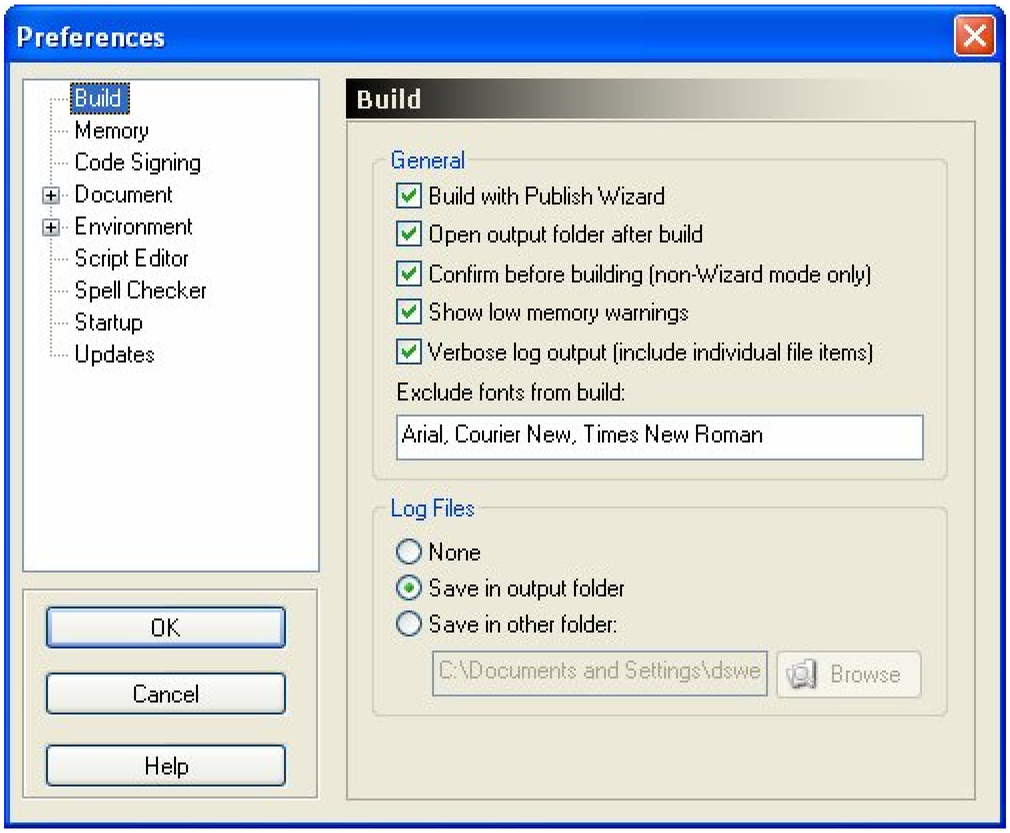
The filters toolbar allows you to apply both default and custom filters to the file list. Filters display files in the file list matching certain criteria, and hide those that don't. By default Visual Patch is configured with three filters: All executables, all key files, and all missing files. You can create custom filters based on any available column in Visual Patch. Note that the column used in the filter does not need to be visible in the design environment, and that only one filter can be applied at a time.

Right-Click Context Menus

A quick way to change many settings within your Visual Patch project is to use the various right-click context menus available. For example, right clicking on a version tab will allow you to add a version, remove the selected version, rename the selected version, or organize all available versions. Right clicking on the file list brings up various options for working with existing files in the list, and allows you to add new files and folder references to your project.

Setting Preferences

There are a number of preferences that you can configure to adjust the Visual Patch design environment. To access the Preferences dialog, choose Edit > Preferences from the file menu.



[✓] Build with Publish Wizard
[✓] Open output folder after build
[✓] Confirm before building (non-Wizard mode only)
[✓] Show low memory warnings
[✓] Verbose log output (include individual file items)

Exclude fonts from build: Arial, Courier New, Times New Roman

Log Files: None [✓ Save in output folder] Save in other folder: C:\Documents and Settings\dswe

The Preferences dialog

The preferences are arranged into categories listed on the left side of the dialog. When you click on a category, the corresponding preferences appear on the right side of the dialog.

The first category contains the build preferences. Here you can specify whether or not the publish wizard should be used, as well as how Visual Patch behaves before and after the build. You can also specify where the log files should be saved to, or even disable the log files.

The Document preferences allow you to change settings that affect the project file. For example, you can configure the auto-save feature that automatically saves your project file as you're working on it to avoid any accidental loss of data. You can also configure the number of undo/redo levels, and choose whether to use the project wizard to create new projects or to simply start with a new, blank project.

Tip: It can be helpful to set the number of undo levels to a larger value, like 25 or 50. That way you can undo even more "steps" back if you change your mind while you're working on a project.

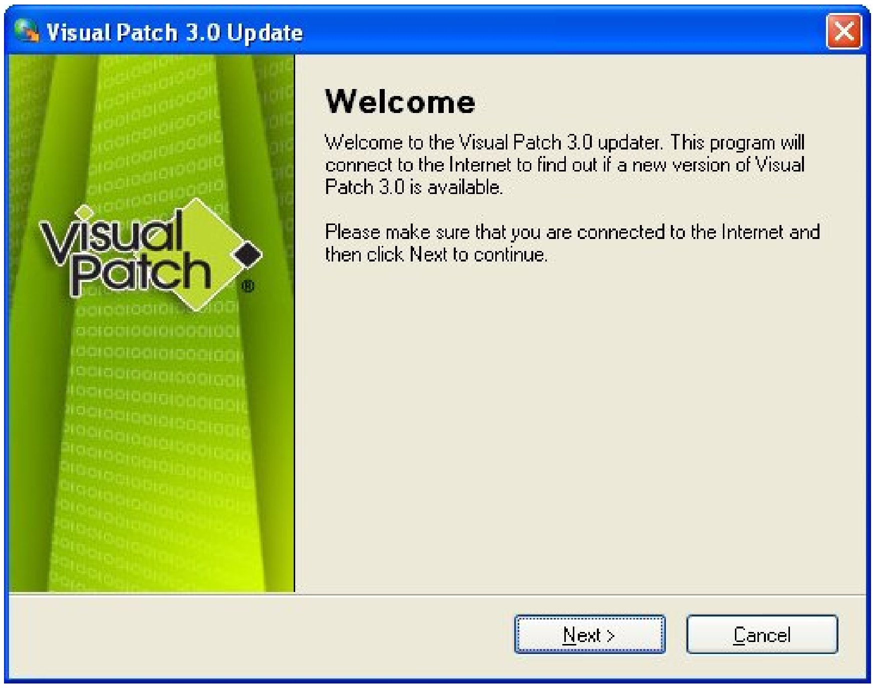
The Environment category allows you to customize the design environment, and the Environment > Folders category allows you to specify the locations of various folders that are used by the project.

There are many other preferences that you can set, such as what to do when the design environment is started (in the Startup category) and what happens when you add files to the project (in the Document > Adding Files category). Take some time to look through the categories and familiarize yourself with the different options that are available.

Updating Visual Patch

As with any application, Visual Patch will from time to time be updated to include bug fixes and possibly new features. In order to ensure you have the most current version of Visual Patch installed, you can check for updates by choosing Help > Check for Update from the program menu.

Note: Visual Patch will automatically prompt you to check for updates every 30 days by default. You can modify this value at any time through the Preferences dialog (Help > Preferences).



Welcome

Welcome to the Visual Patch 3.0 updater. This program will connect to the Internet to find out if a new version of Visual Patch 3.0 is available.

Please make sure that you are connected to the Internet and then click Next to continue.

Check for the most current version of Visual Patch

Tip: To include this type of update functionality in your software application, check out TrueUpdate (www.trueupdate.com) today!

Getting Help

If you still have questions after reading the user's guide, there are many self-help resources at your disposal.

F1 Help

The online help is only a key press away! Visual Patch comes with an extensive online program reference with information on every action and feature in the program.

In fact, whenever possible, pressing F1 will actually bring you directly to the appropriate topic in the online help. This context-sensitive help is an excellent way to answer any questions you may have about a specific dialog or object.

Note: You can also access the online help system by choosing Help > Visual Patch Help.

There are three ways to navigate the online help system: you can find the appropriate topic using the table of contents, with the help of the keyword index, or by searching through the entire help system for a specific word or phrase.

User Forums

Visual Patch is used by developers all over the world. Many users enjoy sharing ideas and tips with other users. The online forums can be an excellent resource when you need help with a project or run into a problem that other users may have encountered.

Choosing Help > User Forums opens your default web browser directly to the online user forums at the Indigo Rose website.

Technical Support

Choosing Help > Technical Support allows you to either check out the various support options available to you or to contact Indigo Rose's technical support department.

Choosing Support Options takes you to the Visual Patch web site, where a variety of online technical support resources are available to you, including a knowledge base with answers to common questions. This is also where you can find information about ordering one of our premium support packages and submitting a support request.

Choosing Contact Support takes you directly to the support ticket submission web page within Visual Patch's support website.

Chapter 4:

Versions and Files

A version is the collection of files and folders that makes up a single release of your software. Having accurate information about your versions and the files they contain is a crucial part of performing a successful patch.

Visual Patch uses this information to analyze the differences between your versions, to determine what data needs to be included in the patch, to identify which version (if any) is actually installed on the user’s system, and to update the files in the installed version so they match the files in your target version.

Since the entire patching process depends on the information you provide, it’s important to understand how to configure all of the versions and files in your project. This chapter will explain everything you need to know about versions and files in order to create a successful patch for your software.

Note: Patching files is the ultimate purpose of any software patching tool. As such, you are expected to be completely familiar with files and folder structures in order to use Visual Patch. If you need more information on the basics of files and hierarchical folder structures, please consult the “Windows Basics” section of the Visual Patch help file.

4

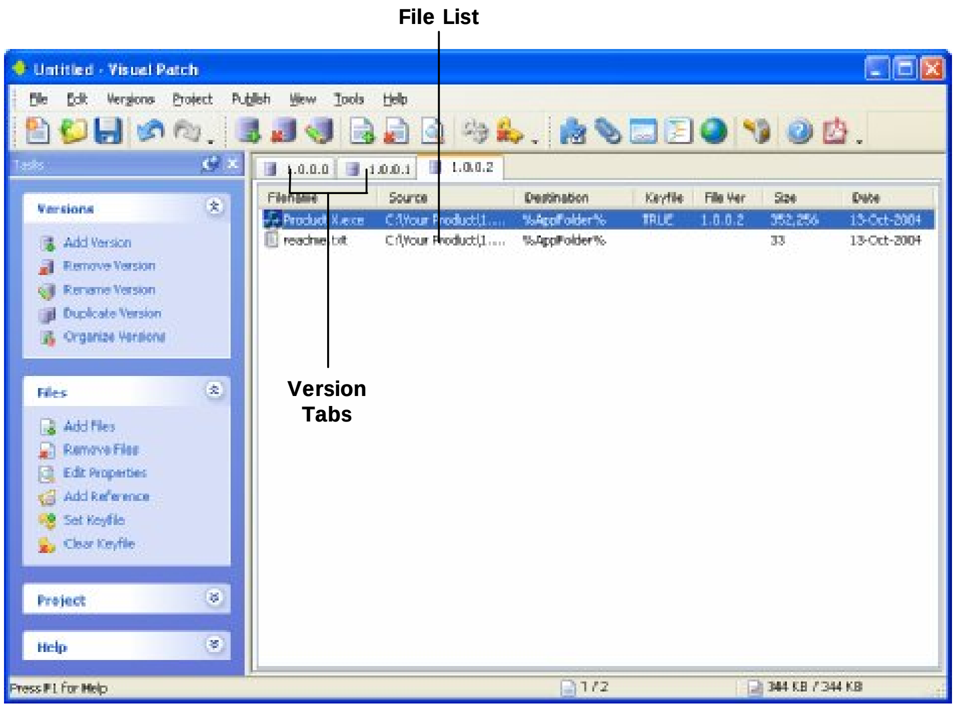
In This Chapter

In this chapter, you'll learn about:

- The project window
- Version tabs
- Adding and removing versions
- File lists
- Filtering file lists
- Folder references
- Adding files and folder references
- Removing files and folder references
- File and folder reference properties
- What %AppFolder% is
- Working with multiple files
- Missing files
- Primer files

The Project Window

The project window contains a series of version tabs, each containing the lists of files that are part of each version. From these lists you can highlight specific files and view or edit their properties.



Filename	Source	Destination	Keyfile	File Ver	Size	Date
ProductX.exe	C:\Your Product\1....	%AppFolder%	TRUE	1.0.0.2	352,256	13-Oct-2004
readme.txt	C:\Your Product\1....	%AppFolder%			33	13-Oct-2004

The Visual Patch Project Window

Version Tabs

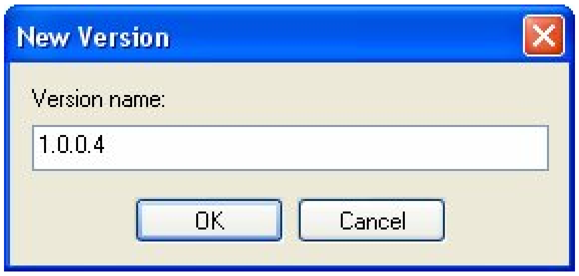
The version tabs on the project window represent each version of your software. New versions may be required to fix bugs or to introduce new features. Every version of your software that your patch will update must have a separate file list that contains all of the files that are part of that particular version. Version tabs are organized from left to right where the leftmost version tab is the oldest version and the rightmost tab is the newest version. These tabs allow you to switch between the versions in your project to view or edit their file lists.

Adding Versions

Each time a new release of your software is created, you will need to add a new version tab to your project so you can add all of the new version's files. The following steps can be taken to add a new version to your project:

1. Select Versions > Add from the program menu.

Selecting Versions > Add from the program menu opens the New Version dialog where you can specify the name you want to give the new version.



Specifying the version name

2. Specify the name of the new version.

Version numbers are normally used as the version names, but you are not limited to version numbers. You could also use a word or phrase.

Tip: If you use version numbers, whenever you add a new tab Visual Patch will automatically try to guess the next version number based on the previous versions in the project.

3. Click the OK button to create the new version.

When you click the OK button, a new version tab with the specified name will be added to the right of all existing version tabs.

Removing Versions

Occasionally mistakes are made that require the removal of a version from your project. To remove a version, select the version you want to remove and then choose Versions > Remove from the program menu. A confirmation dialog will be shown before the version and its referenced files are removed from your project.

Renaming and Duplicating Versions

Versions can be renamed in your project by choosing Versions > Rename from the program menu. This will open the Rename Version dialog where you can specify its new name.

It is also possible to duplicate a version and all of its file references in your project. To do this, simply select the version you want to make a copy of, and choose Versions > Duplicate from the program menu. This will add a new version tab with an automatically generated name containing the same file list as the copied version. After the version duplication, you may want to rename the new version to suit your project.

Organizing Versions

The order of the version tabs in your project is very important. Positioning your versions in the wrong order would result in a patch that is unable to update your software. In order to build a patch that can successfully deal with all your software versions, you need to ensure that Visual Patch “sees” the versions in the correct order, from the oldest on the left, to the newest on the right.

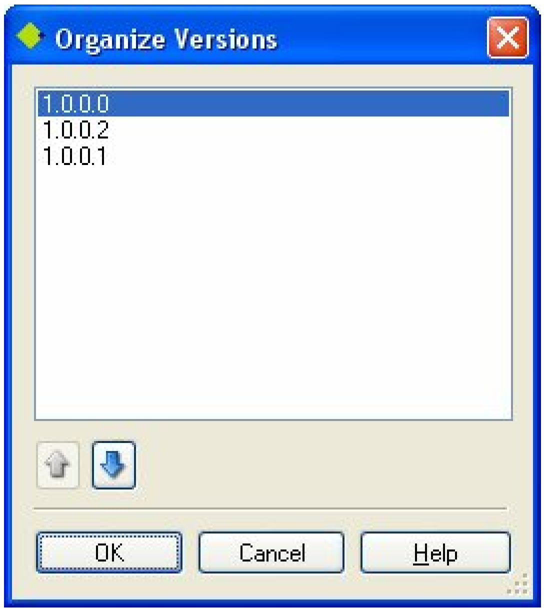
When you build a patch, Visual Patch analyzes the files in the newest version of your software and compares them to the files in all the other versions that are defined in the project. It uses this information to determine what changes are required to update each version to the newest.

If the order of your versions is incorrect, the following steps can be taken to reorganize them to match your software’s version history:

1. Select Versions > Organize from the program menu.

The organization of the version tabs in your project can be changed on the Organize Versions dialog. You can access this dialog by choosing Versions > Organize from the program menu.

The Organize Versions dialog shows a list of all the versions that are currently in your project and orders them top down, from oldest to newest.



The Organize Versions dialog

2. Select the name of the version whose order needs to change, and click the Move Down button.

On the Organize Versions dialog shown in step 1, you will notice that version 1.0.0.2 is in the second position while version 1.0.0.1 is in the last position. This order indicates that 1.0.0.1 is the newest version. Lets assume that this is incorrect and that 1.0.0.2 is the newest version of the software. This means that version 1.0.0.2 needs to be moved into the last position in the list.

To move version 1.0.0.2, you would first select it in the list, and then click the Move Down button (the one with the arrow pointing down).

3. Confirm the positions of all versions, and click OK.

Repeat step 2 until all the versions are in the correct order. Once you’re satisfied with the order of the versions, click OK to make the changes to your project. The version tabs in your project should now follow the order you specified on the Organize Versions dialog. For example, if you moved version 1.0.0.2 to the bottom of the list, version 1.0.0.2 would now be the rightmost version tab.

File Lists

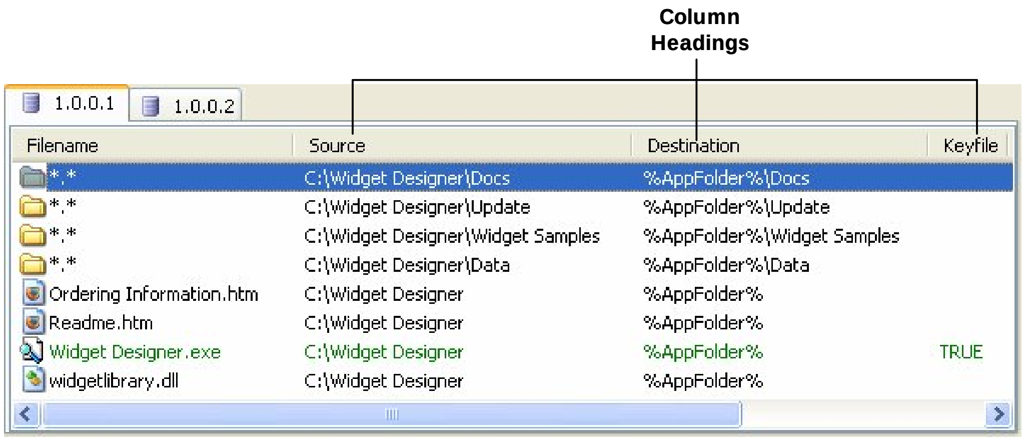
Version tabs in Visual Patch were designed to contain the lists of files that are part of each version of your software. Since a minimum of two versions is required to create a patch, your project will normally contain at least two lists of files. These lists contain references to each file on your local system or LAN where Visual Patch can access them during the build process. These lists should only contain the files that are distributed to the user.

The best strategy for managing your source files is to keep each version of your software in a separate folder. It is also best to preserve the internal folder structure for each version, i.e. to store each version complete with any subfolders relative to the main application's folder. This greatly simplifies the process of adding the versions' files to your project and setting their destination paths.

File lists can also be sorted and filtered so you can easily find or focus on the files you need to customize. Each file within a list has properties you can view or set such as its source and destination paths, key file property, and patch conditions.

Column Headings

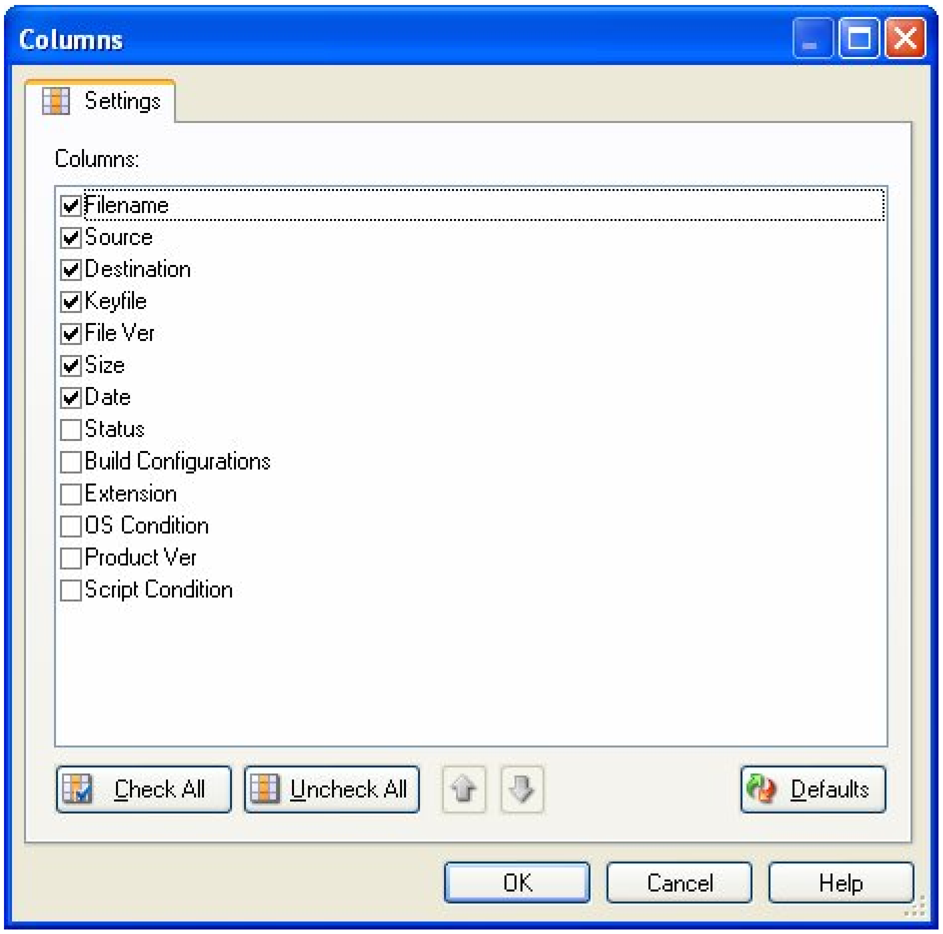
All file lists have columns that display different information about each file in your project. You can sort the information along any of the columns by clicking on the heading for that column. If you click on the same heading again, the files will be sorted by that category in reverse order.



Filename	Source	Destination	Keyfile
.	C:\Widget Designer\Docs	%AppFolder%\Docs	
.	C:\Widget Designer\Update	%AppFolder%\Update	
.	C:\Widget Designer\Widget Samples	%AppFolder%\Widget Samples	
.	C:\Widget Designer\Data	%AppFolder%\Data	
Ordering Information.htm	C:\Widget Designer	%AppFolder%	
Readme.htm	C:\Widget Designer	%AppFolder%	
Widget Designer.exe	C:\Widget Designer	%AppFolder%	TRUE
widgetlibrary.dll	C:\Widget Designer	%AppFolder%	

Column headings

You can customize the columns by choosing View > Columns. This opens the Columns dialog, where you can choose which columns to display in the file list, and change the order (left to right) that the columns are listed in.



- [✓] Filename
- [✓] Source
- [✓] Destination
- [✓] Keyfile
- [✓] File Ver
- [✓] Size
- [✓] Date
- [ ] Status
- [ ] Build Configurations
- [ ] Extension
- [ ] OS Condition
- [ ] Product Ver
- [ ] Script Condition

The Columns dialog

Tip: Moving a column up in the columns list moves it to the left in the file list; moving a column down in the columns list moves it to the right.

File List Items

The items that appear in the Visual Patch file lists refer to files on your system. When you add a file to your project, the file is not copied into the project. It is only *referenced* by the project.

In other words, only the path to the file (and a few other properties, such as the size of the file) is stored in your project. If the original file is renamed, moved or deleted, Visual Patch won't be able to build it into the patch executable. (In fact, it will show up in the file list as a missing file.)

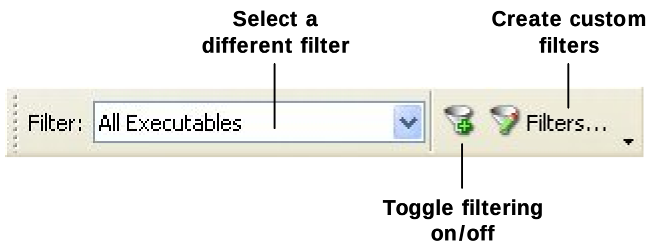
The file itself, however, is not part of the project. If you copy your project from one system to another, you will not be able to build it unless you also copy the files, and place them in the exact same folder structure—in other words, you must place the files where the project expects them to be located.

Note: The files that your project references are commonly referred to as *source files*.

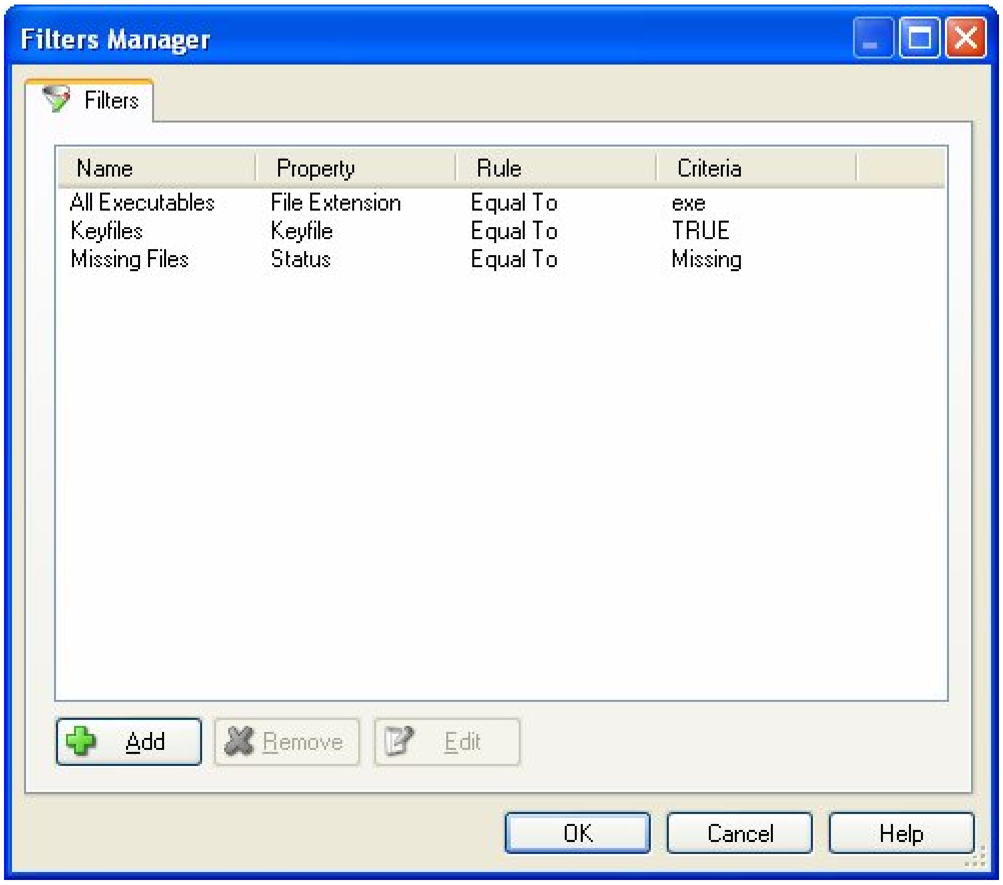
Filtering the File List

If your software contains many files, you might find it easier to temporarily "hide" some of them from the file list in order to focus on the files that you're interested in at the moment. You can do this by applying a filter to the file list so it only shows files that meet specific criteria. For example, you could filter the list to only show files that are larger than 5 MB, or files that are currently missing from your system.

This is all done using the Filters toolbar.



Clicking the Filters... button on the toolbar opens the filters manager, where you can configure the list of filters that appear in the drop-down selector on the toolbar.



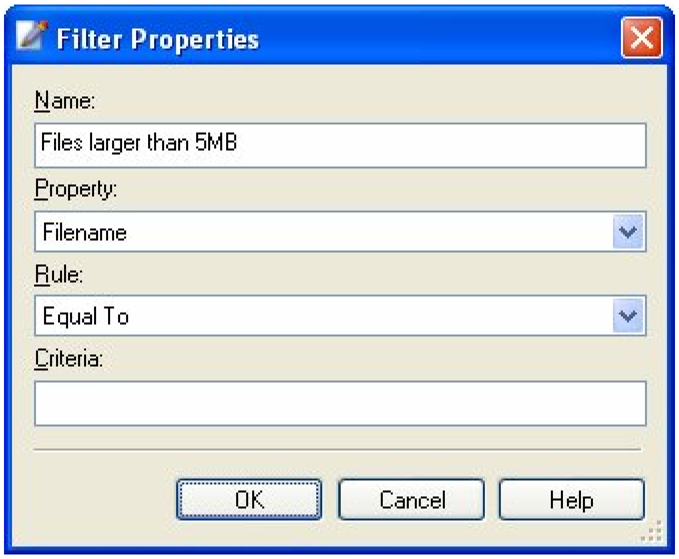
Name	Property	Rule	Criteria
All Executables	File Extension	Equal To	exe
Keyfiles	Keyfile	Equal To	TRUE
Missing Files	Status	Equal To	Missing

The Filters Manager dialog

The filters manager allows you to define custom filters using a wide range of criteria.

Creating a Filter

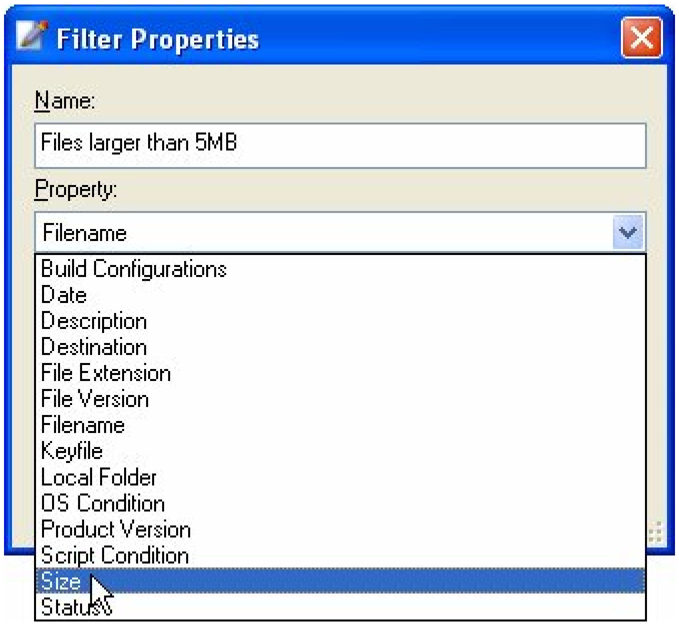
Setting up a new filter is incredibly easy. After clicking on Add in the filters manager, you simply give the filter a name:



Name: Files larger than 5MB
Property: Filename
Rule: Equal To
Criteria:

Naming the filter

…then select the property you want to filter on:



Name:

Files larger than 5MB

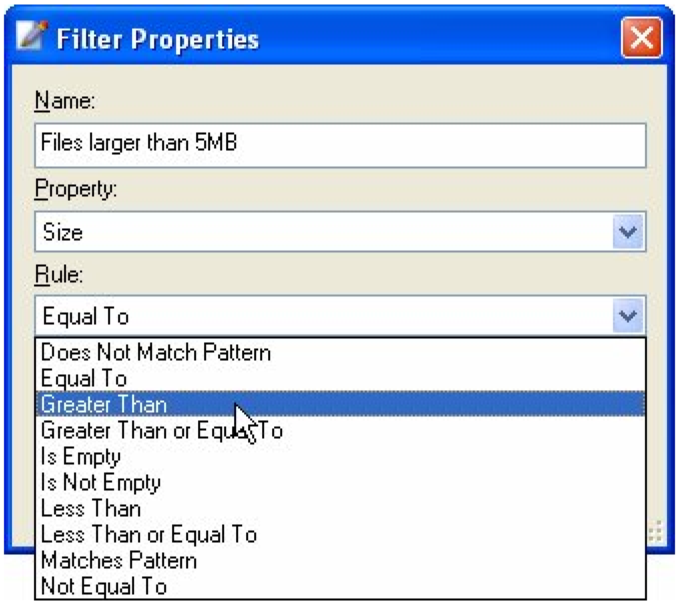
Property:

Filename

- Build Configurations
- Date
- Description
- Destination
- File Extension
- File Version
- Filename
- Keyfile
- Local Folder
- OS Condition
- Product Version
- Script Condition
- Size
- Status

Selecting the filter property

…select the rule that you want to use:

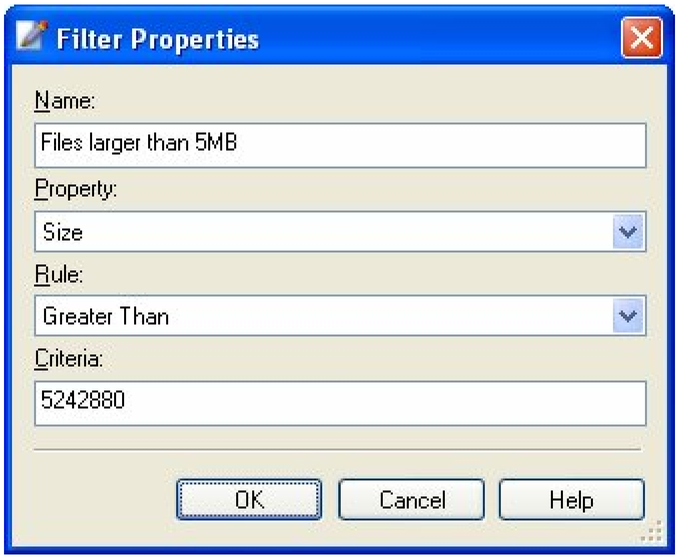


Name:
Files larger than 5MB
Property:
Size
Rule:
Equal To

Does Not Match Pattern
Equal To
Greater Than
Greater Than or Equal To
Is Empty
Is Not Empty
Less Than
Less Than or Equal To
Matches Pattern
Not Equal To

Selecting the filter rule

…and finally provide the criteria:



Name: Files larger than 5MB
Property: Size
Rule: Greater Than
Criteria: 5242880

Providing the filter criteria

The filters are saved on a global basis, so once a filter is defined it can be used in all of your projects. You can switch between filters by using the drop-down selector on the filters toolbar.

Folder References

Folder references are similar to files, but they reference a folder on your system instead of a single file. They're useful for including folders full of files without having to reference each file individually.

For example, if you have a folder full of documents that you need to patch, you can use a folder reference to add the entire folder to your project. This also lets you configure the settings for all of those files from a single item in the file list.

You can even choose whether or not to reference folders recursively. In other words, you can choose whether or not to include the files in any subfolders (and subfolders of subfolders, and so on) that may be in the selected folder. (This is in fact the default behavior of folder references; however, you can choose to only include the files in the immediate folder if you want by simply turning off the “Recurse subfolders” option.)

Each folder reference also has a File Mask setting that you can use to include only files that match a specific pattern. For instance, if you wanted to use different settings for the .doc files and all other files in your product’s folder, you could use two folder references—one for each type. So, for example, you could set up one folder reference to add everything beneath C:\Program Files\ProductX that matches “*.doc” and another one to add everything in that same path that are “Files that do not match” the “*.doc” file mask. You could then configure their settings differently—for example, you could set one up with a condition so it only patches the .doc files on specific operating systems.

Tip: Folder references are one of the most powerful features in Visual Patch! Be sure to use them to your advantage.

Overriding Individual Files

There may be cases where you want to include a folder full of files, but you want to use different settings for some of the files in the folder. Visual Patch makes this incredibly easy to accomplish because you can override a folder reference with individual file items.

The rule is simple: the settings for individual file items always take precedence over folder references. So if you have a folder reference that happens to include files A, B, and C, and you also add file B to your project on its own, the settings for the standalone file item will be used for file B, and not the settings for the folder reference.

Note: It doesn’t matter what order the files are in — individual file items always override folder references.

Adding Files

One method for keeping source files organized is to keep each version’s source files in a separate folder. While this strategy may take up additional hard drive space, it greatly simplifies the management of your patches.

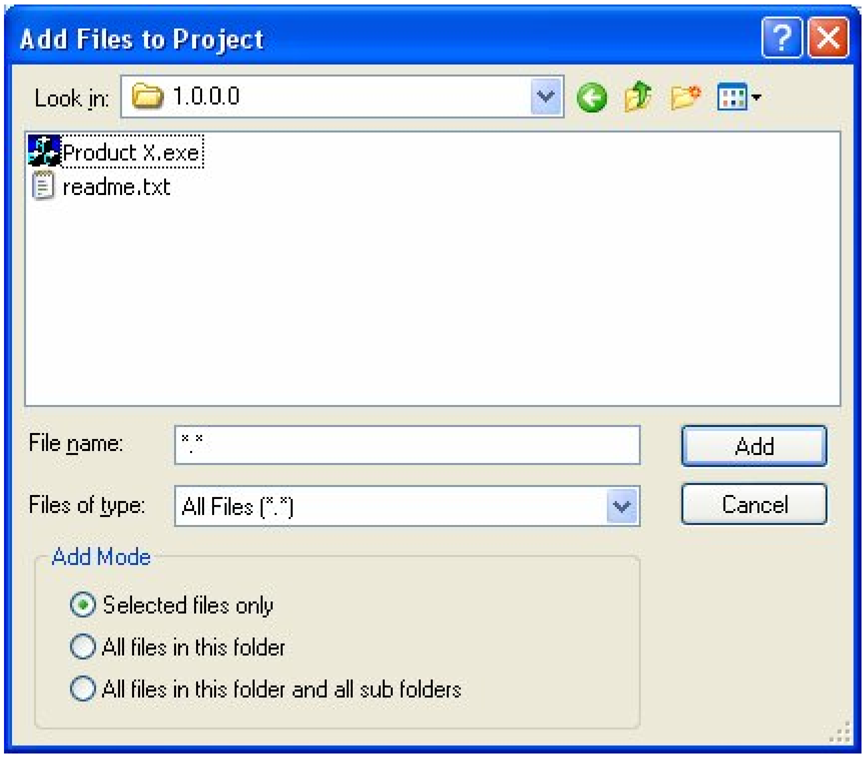
Once you have added a new version tab to your project, the next step is to add all of its source files.

1. Click on one of the version tabs.

Select the version tab whose source files you want to add to the project.

2. Choose Versions > Add Files from the program menu.

This will open the Add Files to Project dialog.



Adding files to the project

Tip: You can also click the Add Files button on the toolbar, press the Insert key, or right-click on the project window and select Add Files from the right-click menu.

3. Select the add mode that you want to use.

At the bottom of the Add Files to Project dialog is a section labeled “Add Mode.” This setting controls which files in the folder structure will be added to the version’s file list. There are three add modes to choose from:

- The “Selected files only” option will only add the files that you select
- The “All files in this folder” option will add all of the files in the folder that the Add Files to Project dialog is currently displaying
- The “All files in this folder and all subfolders” option will add all of the files in the current folder, and all of the files in any subfolders as well

4. Select the files that you want to add, or navigate to the folder where all the files (and possibly subfolders) that you want to add are located.

Depending on the add mode you chose, you either need to select one or more individual files in the browse dialog, or navigate into the folder that you want to add files from.

5. Click the Add button to add the files to your project.

Once you click the Add button, the files will appear in the file list.

Tip: You can also add files to your project by dragging and dropping them onto the selected version tab in the project window.

Adding Folder References

If your software contains folders of files whose settings do not need to be individually customized, folder references are a convenient way to add those files to your project.

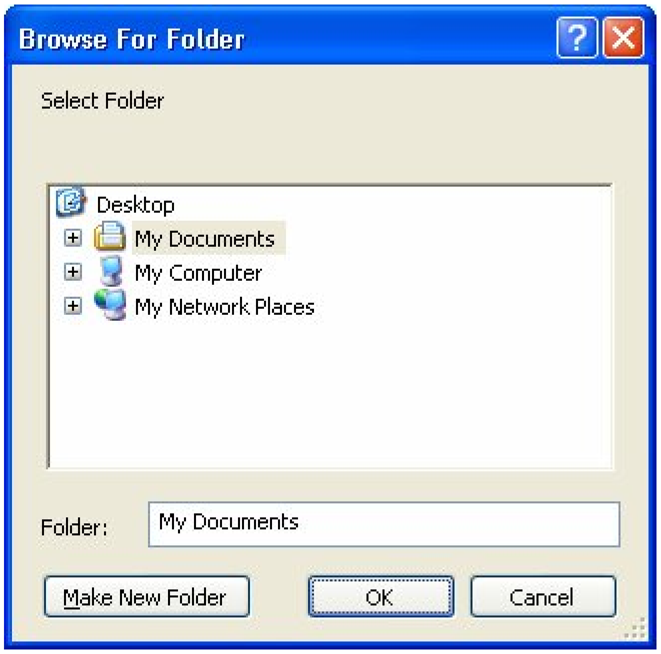
Assuming you have a version tab ready for files to be added, the following steps can be taken to add a folder reference to your project:

1. Click on one of the version tabs.

Select the version tab whose source files you want to add to the project as a folder reference.

2. Choose Versions > Add Reference from the program menu.

This will open the Browse For Folder dialog.



Browsing for a folder to reference

Tip: You can also click the Add Folder Reference button on the toolbar, press Ctrl+Insert, or right-click on the project window and select Add Folder Reference from the right-click menu.

3. Select the folder that you want to add.

Simply browse for the folder that you want to add. When you select a folder, its name will appear in the Folder field near the bottom of the dialog.

4. Click OK to add the folder to your project.

When you click the OK button, the folder reference will appear in the file list.

Tip: You can also drag and drop a folder onto the selected version tab to add a folder reference to your project. This will open the Drag and Drop Assistant dialog to confirm its addition.

Removing Files

To remove files from your Visual Patch project:

1. Select the files that you want to remove on the project window.

Simply click on the file items that you want to remove.

2. Choose Edit > Delete.

Choosing Edit > Delete from the program menu will remove the selected file items.

Tip: You can also click the Remove Files button, press the Delete key, or right-click on the project window and select Remove from the context menu.

Note: Removing files from your project removes the file items from the file list, but does not delete the original files. The original files remain completely unaffected.

Removing Folder References

Removing folder references is exactly like removing files. Simply select the folder references that you want to remove and choose Edit > Delete (or press the Delete key, etc.).

File Properties

You can use the File Properties dialog to view and edit the settings for any file in your project.

To access the File Properties dialog:

1. Select a file on one of your version tabs.

Simply select the file item that you want to edit.

2. Choose Versions > File Properties.

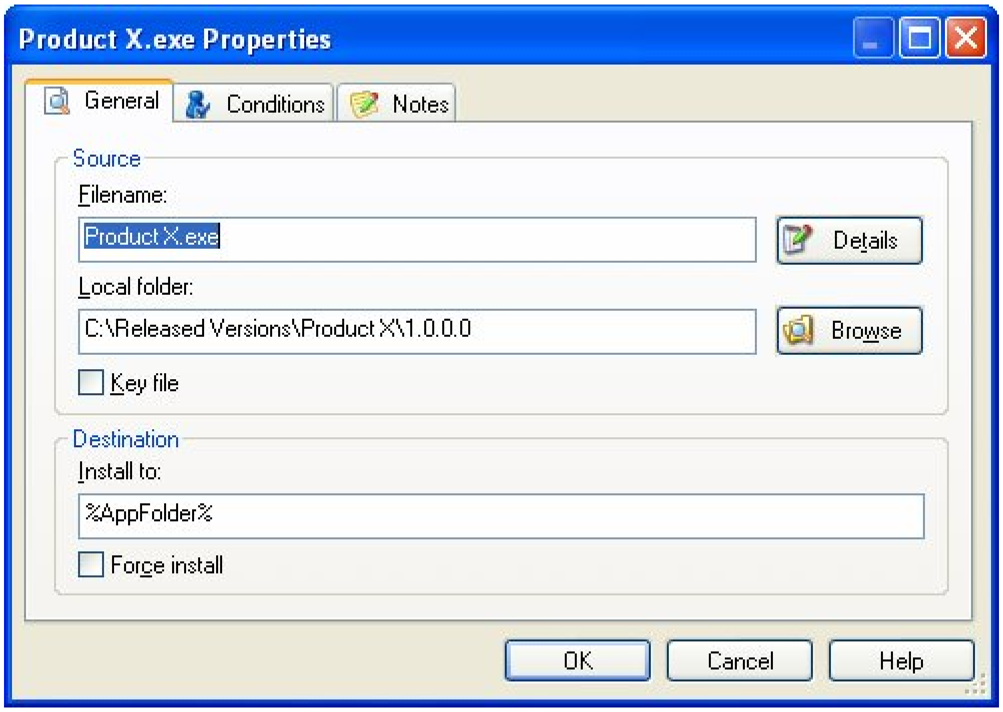
This will open the File Properties dialog, where you can view and edit the properties of the file you selected.

Tip: There are several other ways to access the file properties as well. You can click the File Properties button, press Ctrl+Enter, right-click on the file item and select File Properties from the right-click menu, or simply double-click on a file in the list.

There are three tabs on the File Properties dialog: General, Conditions, and Notes.

General

The General tab is where you will find information about the file itself, such as its name and location on your system, whether it's a key file, and where it is expected to be on the user's system (the destination path).



Product X.exe Properties

Source
Filename: Product X.exe
Local folder: C:\Released Versions\Product X\1.0.0.0
Key file: not checked

Destination
Install to: %AppFolder%
Force install: not checked

The General tab of the File Properties dialog

Filename

The Filename field specifies the name and extension of the file being referenced.

Local folder

The Local folder specifies the *current location* of the file on your system. This must be a folder that is within local access of your system or LAN.

Visual Patch uses the local folder in combination with the filename to retrieve the file's information (e.g. when you open a project) and to analyze the file when it builds the patch executable (i.e. when you build the project).

Key file

Key files are a very important part of Visual Patch. Key files are used to identify what version of the software is on the user's system and whether or not it is a valid version. Designating a file as a key file means that you want Visual Patch to verify its existence and its MD5 signature in order to fully identify the version it belongs to. If the key file doesn't exist, or its MD5 signature doesn't match, Visual Patch will consider that version not found.

Key files are usually files whose contents are unique to a single version. The best key files are files that change from one version to the next, such as the main executable for the software. The file names and paths don't need to be unique, but the *contents* do.

Visual Patch requires at least one unique key file per version in order to uniquely identify that version. In other words, you must designate at least one key file per version whose contents are different in every other version.

Key files must also not change after being installed or patched since Visual Patch relies on their MD5 signature for validation. Files that are normally changed after they are installed (for example, .ini files) should not be designated as key files. If a key file has been changed in any way from the original file that is referenced in your project, it will prevent the version from being identified.

A version can contain as many key files as you like; however, only one key file *must* be unique between versions. If your software contains mission critical files whose existence and integrity must be checked, you can set them as key files as well, even though they haven't changed between versions. Again, only one *unique* key file is required to identify a version.

Install to

The Install to (or "destination") property specifies where the file is expected to exist on the user's system. Though this seems simple enough, it is complicated by the fact that you can't predict the layout of the folder structure on the user's system.

For instance, if you want to patch a file located in the user's "My Documents" folder, how do you know what the full path to that folder is? It could be anything from "C:\Documents and Settings\JoeUser\My Documents" to "F:\Joe's Docs."

Visual Patch gets around this uncertainty by providing you with a number of built-in session variables for common system folders. These are essentially placeholders (like %MyDocumentsFolder%) that you can use in your file paths, and that will be replaced by the corresponding path on the user's system at run time. You simply use the appropriate placeholder, and Visual Patch will take care of investigating what the actual path is when your user runs the patch file.

For more information on session variables, see Chapter 7.

%AppFolder%

In most cases you know the directory structure of your software, however the user is often given the choice of where they would like to install. For this reason, the exact folder path is not known until run time. Visual Patch uses a special session variable to represent the main folder that your software was installed to. The name of this session variable is %AppFolder%, which is short for *Application Folder*, i.e. the folder where your software application is installed. At run time, this variable is populated with the folder path from a Registry value, file search, or some other method implemented using action script. (By default, the setting of %AppFolder% is handled in the On Startup event by the VisualPatch.CheckFolderVersion action.)

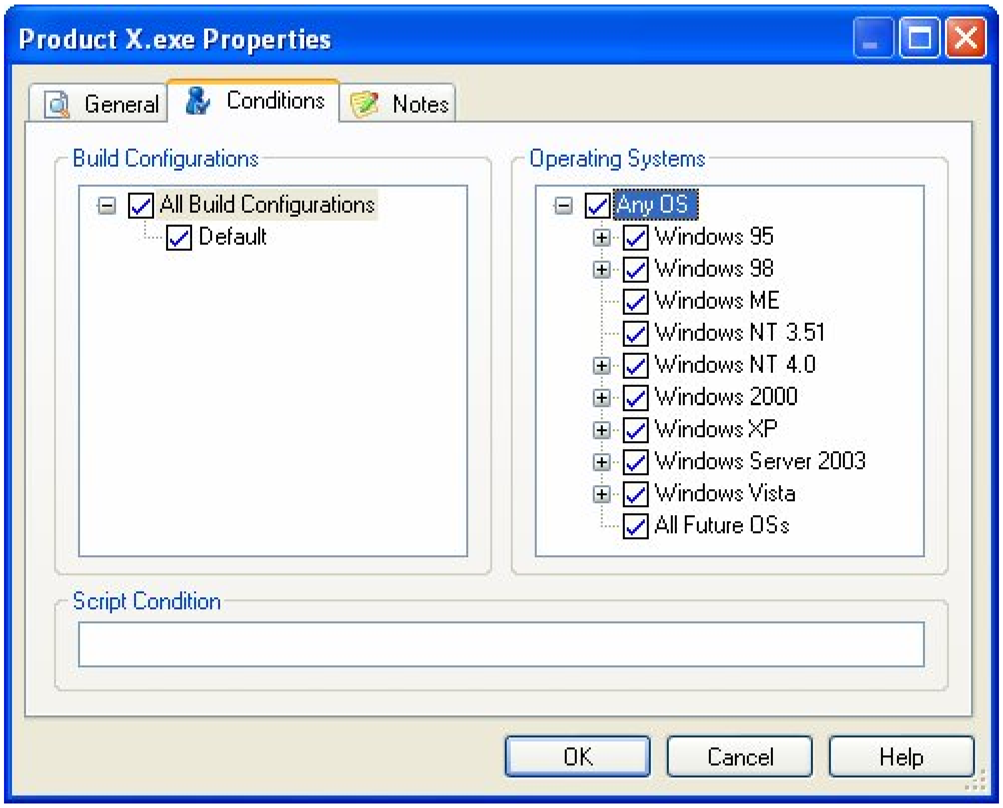
Force Install

The Force install check box is for files that you *always* want installed as part of the patching process for a given version. It instructs Visual Patch to include the whole file in the patch and to overwrite any existing version of the file that it finds on the user's system.

You should only enable this option when you're sure that it's okay for a file to have been modified after it was installed, *and* you want to overwrite the modified version with the new file. (Generally you will want to leave this option unchecked.)

Conditions

The Conditions tab is where you can edit settings that affect whether the file will be included in the patch executable, and whether it will actually be used when the patch is run. In other words, it is used to conditionally include or exclude a file from the patch at build time or run time.


Product X.exe Properties

Build Configurations

- [✓] All Build Configurations
  - [✓] Default

Operating Systems

- [✓] Any OS
  - [✓] Windows 95
  - [✓] Windows 98
  - [✓] Windows ME
  - [✓] Windows NT 3.51
  - [✓] Windows NT 4.0
  - [✓] Windows 2000
  - [✓] Windows XP
  - [✓] Windows Server 2003
  - [✓] Windows Vista
  - [✓] All Future OSs

Script Condition

The Conditions tab of the File Properties dialog

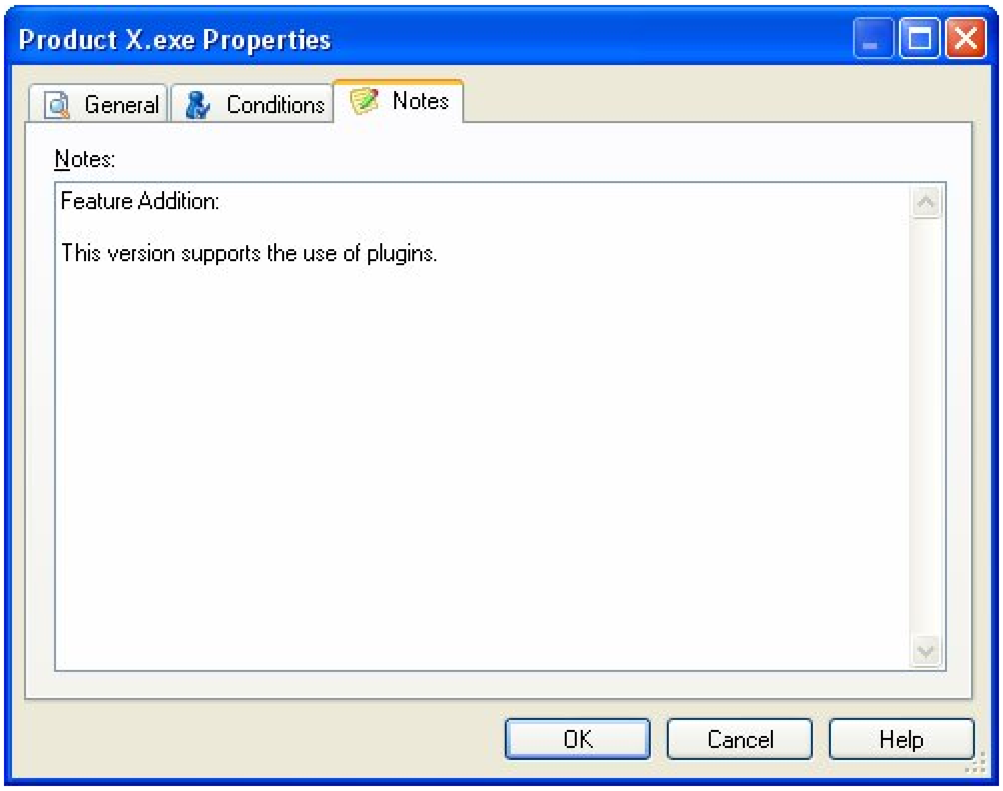
The list of build configurations allows you to select the build configurations that the file will be included in. The file will only be included in the patch if it is included in the build configuration that is selected when you build the project.

The list of operating systems allows you to specify which operating systems the file will be patched on. The file will only be patched on an operating system if it is marked with a check in this list. So, for example, if you uncheck Windows 95 and Windows 98 for a file, it will not be patched on a system running Windows 95 or Windows 98.

The script condition is a short expression written in Lua script. It must evaluate to a Boolean value of either true or false. At run time this result will determine whether or not the file will be patched. For example, if you have a variable used in your project called MyNum and you want to check to see if its value is 5, the script condition would be MyNum == 5. If the contents of the variable is 5, the condition evaluates to true and the file will be patched. If the variable contains some other value, the condition evaluates to false, and the file will not be patched.

Notes

The Notes tab is where you can type any notes that you want to keep about the file. These are not used by the patch, and are not included in the patch executable. The Notes tab is simply a convenient place for you to keep notes about the file for your own purposes. For example, you could list any special instructions involved in preparing the file for the build process, or you could keep track of who last edited the file's properties.



Notes:

Feature Addition:

This version supports the use of plugins.

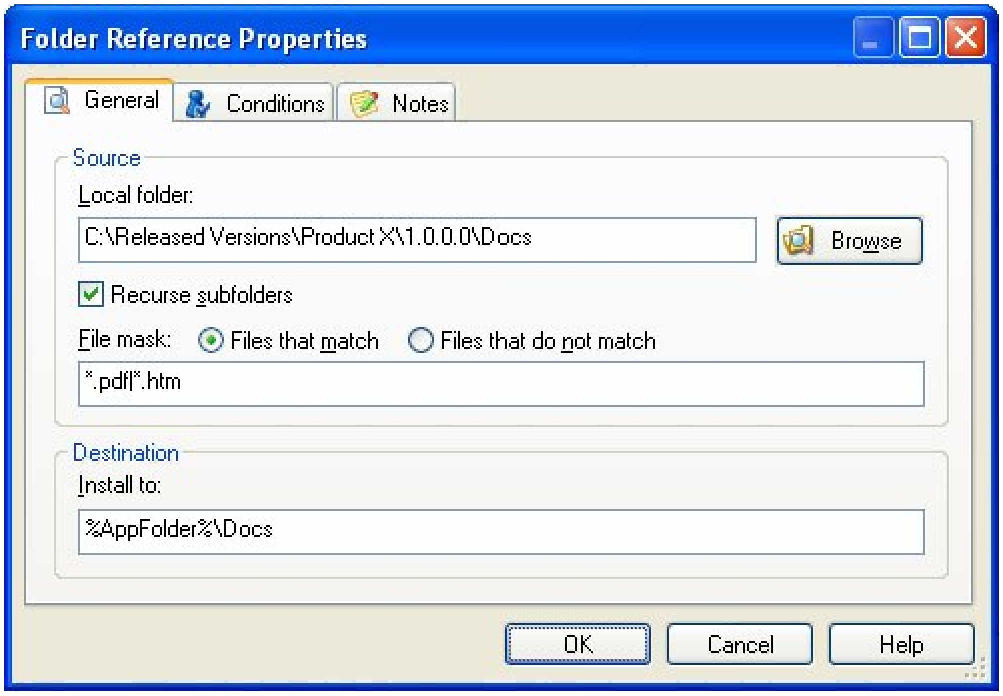
The Notes tab of the File Properties dialog

Folder Reference Properties

For the most part, Folder References have the same properties as files do. In fact, the Conditions and Notes tabs are exactly the same. However, there are a couple differences on the General tab worth mentioning:

Folder references have a “Recurse subfolders” option that controls whether files in subfolders (and their subfolders) should be included as well. This allows you to include full folder trees if you wish.

Folder references also have a File mask setting that allows you to include or exclude a range of files that match a filename pattern, e.g. “*.exe”. This allows you to use custom settings for different file types.



Local folder: C:\Released Versions\Product X\1.0.0.0\Docs

[✓] Recurse subfolders

File mask: [● Files that match] ○ Files that do not match

.pdf|.htm

Install to: %AppFolder%\Docs

The General tab of the Folder Reference Properties dialog

Working with Multiple Files

You can use the Multiple File Properties dialog to edit the properties of more than one file (or folder reference) in your project at a time.

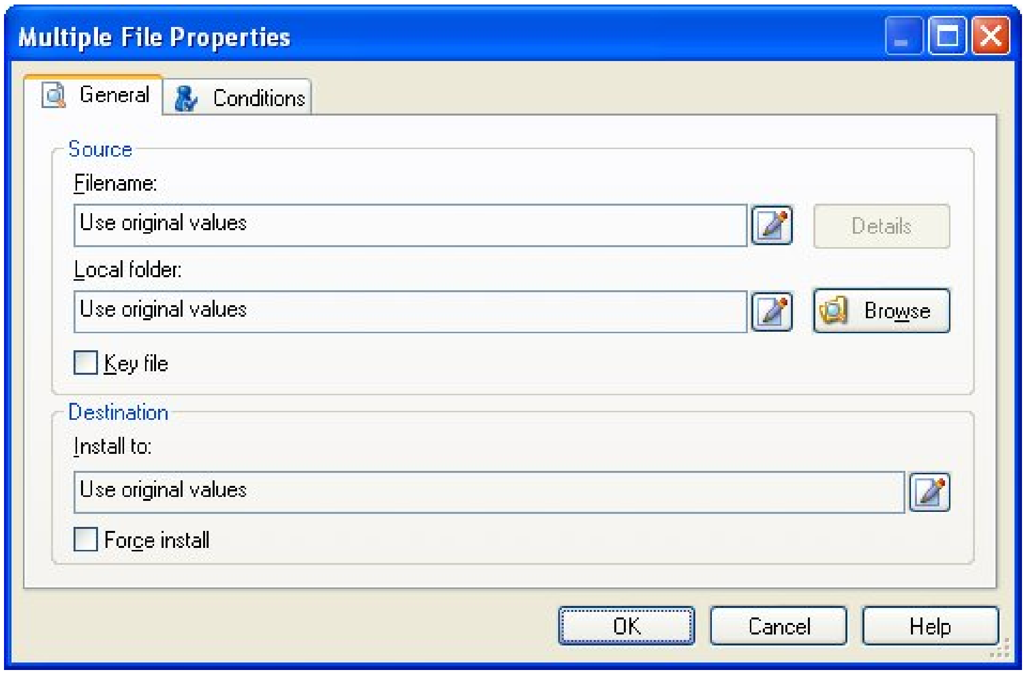
To access the Multiple File Properties dialog:

1. Select more than one file on a version tab. (You can select multiple files by pressing and holding the Ctrl key or the Shift key while you click on the files.)

2. Choose Versions > File Properties from the menu.

   (You can also click the File Properties button, press the Ctrl+Enter key, or right-click on the files and select File Properties from the right-click menu.)

   This will open the Multiple File Properties dialog, where you can view and edit the properties of the files you selected.



Filename: Use original values

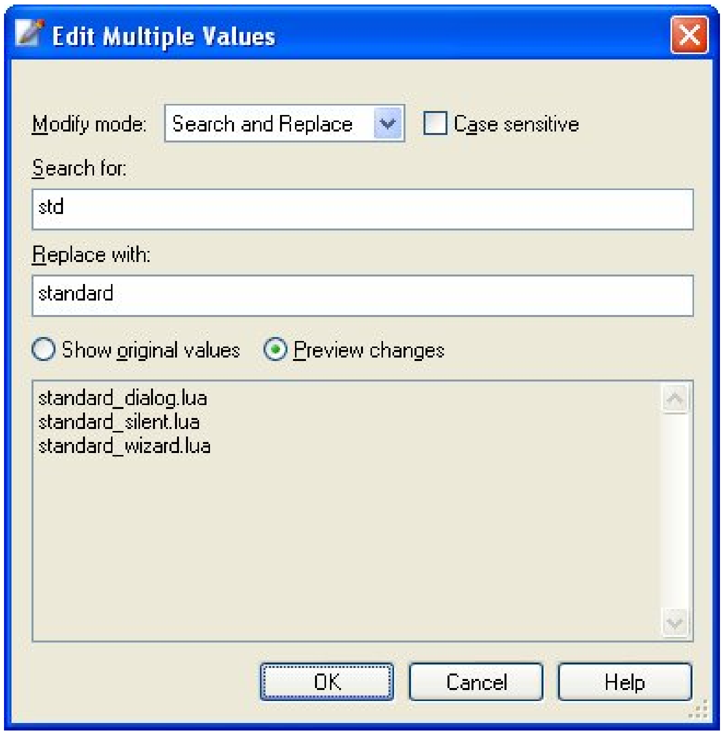
Local folder: Use original values

Install to: Use original values

The General tab of the Multiple File Properties dialog

The Multiple File Properties dialog is similar to the File Properties dialog, but there are some important differences:

- There is no Notes tab.
- You cannot enter text in a field directly; instead, you must click the Edit button to the right of the field, and use the Edit Multiple Values dialog to change the text. Note that the Edit Multiple Values dialog allows you to completely replace the text for all selected files, append (or prepend) some text to the value for each selected file, or search and replace among the values for all selected files.



Modify mode: Search and Replace

[ ] Case sensitive

Search for: std

Replace with: standard

Show original values [✓ Preview changes]

```
standard_dialog.lua
standard_silent.lua
standard_wizard.lua
```

Changing three filenames at once using the Edit Multiple Values dialog

- Check boxes on the Multiple File Properties dialog have three states: enabled (☑), disabled (☐), and mixed (☑). The mixed state preserves the settings for that check box in all of the selected files.

For more information on using the Multiple File Properties dialog, please consult the Visual Patch help file.

Missing Files

Since Visual Patch only records an informational link to each file, and doesn't actually maintain a copy of the file itself, it's possible for files to go "missing" from Visual Patch's point of view. For example, if a file in your project has moved to another folder, Visual Patch will no longer be able to find it at its original location. The same thing happens when a file is renamed or deleted. Visual Patch only knows to look for a file at the place where it was when you "showed" the file to Visual Patch by adding it to your project.

Although Visual Patch might not know where a missing file has ended up, it definitely knows when a file is missing. Whenever a file in your project can't be found at its original location, Visual Patch displays the file's information in red instead of black. The red color makes it easy for you to see which files in your project are missing.

The file's status will also show as "Missing" instead of "OK" in the Status column on the list view, if that column is enabled. You can enable the Status column from the column settings dialog found by choosing View > Columns from the program menu.

Tip: A quick way to determine which files are missing in your project is to click on the Status column heading to sort the files in the list view according to their status.

If you find that your files are suddenly playing hide-and-seek with Visual Patch, try to remember if you've made any changes to your files recently. If you moved the files, you can try moving them back. If you renamed the files, you can restore the original names. If you deleted the files, you'll have to replace them.

If Visual Patch still shows the files in red after you've corrected the situation, you probably just need to refresh the display. To do so, simply choose View > Refresh from the program menu, or press the F5 key. Refreshing the display causes Visual Patch to re-examine the location of every file in your project.

Note: Visual Patch automatically refreshes the display for you when you open a project or initiate the build process.

If you moved, renamed, or deleted the files on purpose, and you want Visual Patch to use the files at their new locations, remember the files by their new names, or just forget about the past and move on, you'll need to change the local source path on the File Properties dialog for each file. This can easily be done by selecting all the files and then modifying their properties using the Multiple File Properties dialog.

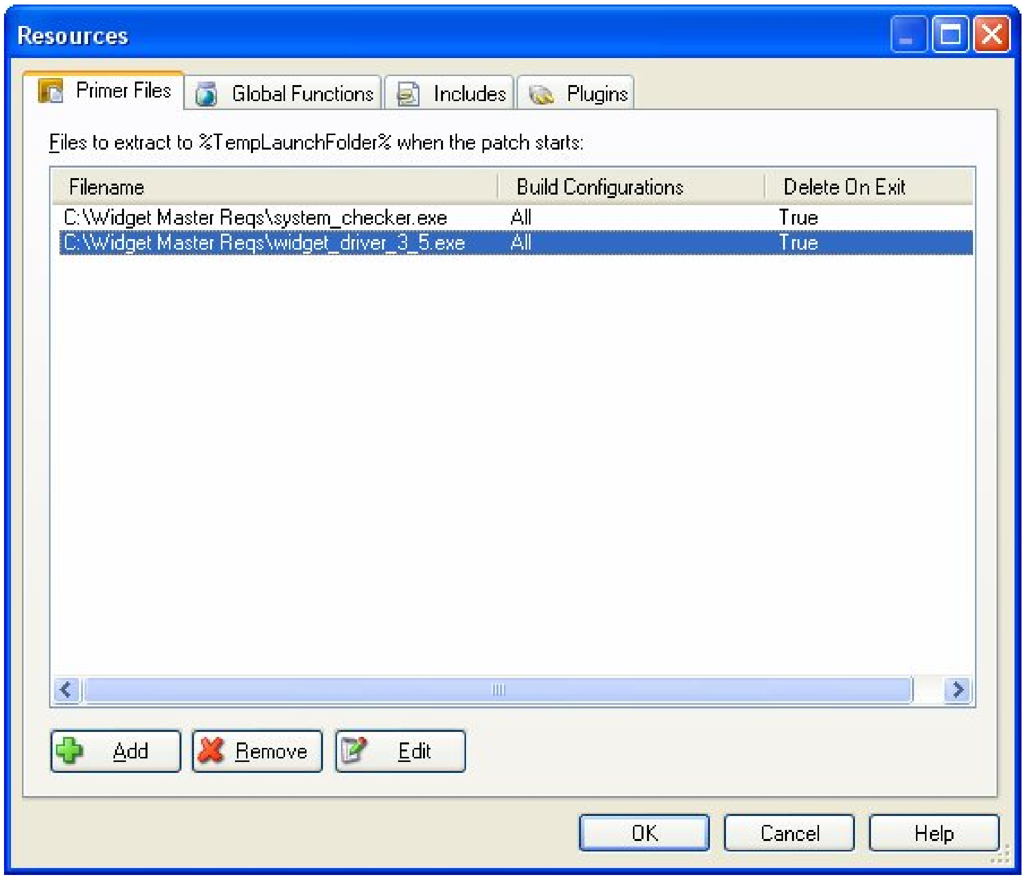
Primer Files

Primer files get extracted from the patch executable *before* the patch process begins. This means you can use primer files at the very start of the patch process, right after the user runs the patch executable. Primer files are useful in situations where changes need to be made before the normal patching process begins.

One use of primer files is to run a program on the user’s system before the rest of your files are patched. For example, you might need to execute a custom program or DLL function before your software is patched—perhaps to perform some product-specific, low-level pre-patch tests on the user’s hardware and write the results to the Registry. By adding your custom program or .dll to the list of primer files, you could distribute it “inside” your patch executable and still be able to run it before any of the “normal” files in your project are patched.

Primer files are automatically extracted to a temporary folder at run time. The path to this folder is stored in a session variable named %TempLaunchFolder%. You should use this session variable in actions when you need to access your primer files.

Any type of file can be added as a primer file simply by adding it to the Primer Files tab on the Resources dialog. All files on this tab will be included in the patch executable when you build your patch, but are not part of any version of your software.



Files to extract to %TempLaunchFolder% when the patch starts:

Filename	Build Configurations	Delete On Exit
C:\Widget Master Reqs\system_checker.exe	All	True
C:\Widget Master Reqs\widget_driver_3_5.exe	All	True

The Primer Files dialog

Tip: Another way to access files at the start of the patch is to distribute them "alongside" your patch. For instance, you could store the files "externally" on the same CD-ROM, and access them directly; or, you could download the files from your web site using an HTTP.Download action in the On Startup project event.

Chapter 5:

Creating the User Interface

Creating the user interface is an integral part of every project since the user interface is the first thing your end users will see when they run your patch. It also serves as a bridge between the information that the user has and that the patch wants. Having an easy to use yet fully functional user interface is something that all developers working with Visual Patch should be concerned about.

This chapter will introduce you to the user interface and get you well on your way to creating a sharp and consistent look and feel for all of your projects.

5

In This Chapter

In this chapter, you'll learn about:

- The user interface
- Screens
- Themes
- The background window
- Taskbar settings
- Dialog and Status Dialog actions

The User Interface

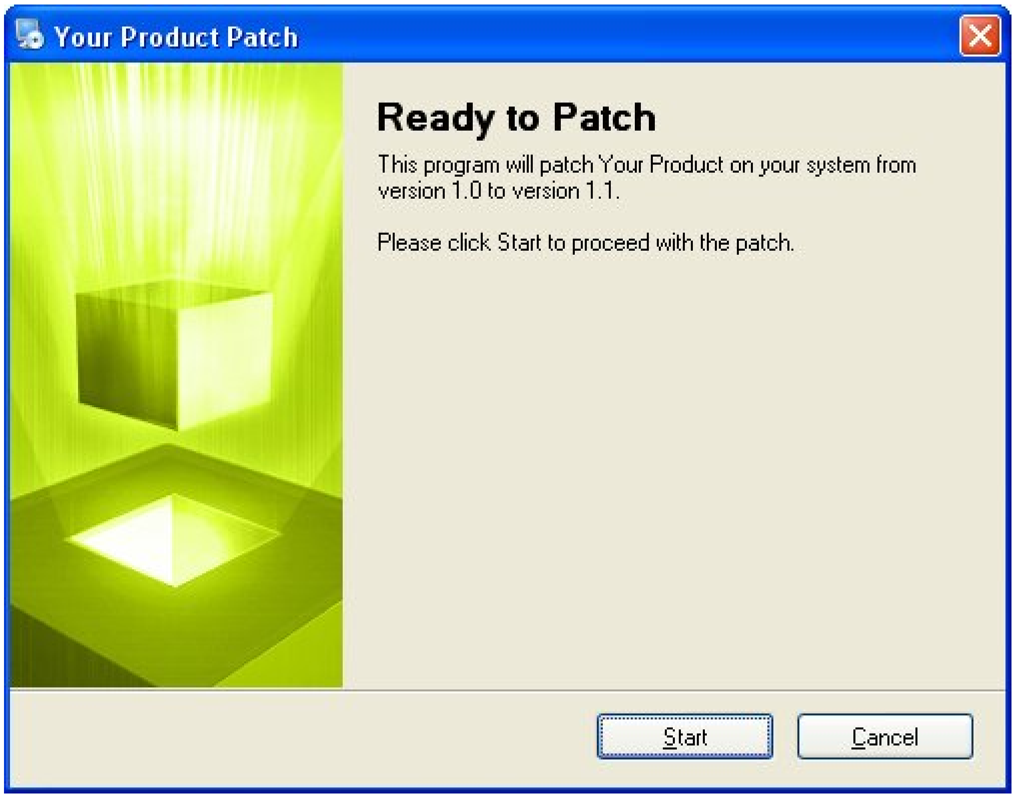
You can think of the user interface as any part of the patch process that the end user will see. When a screen is displayed, the end user is seeing part of the user interface. When the end user clicks the Next button, the end user is interacting with the user interface.

The basic elements of the user interface are:

- Screens
- Themes
- The Background Window
- Taskbar visibility
- Any actions that generate a user interface element (e.g. Dialog.Message, StatusDlg.Show)

Screens

The most important aspects of your user interface are the screens that you choose to display, since this is where your end user will actually interact with the patch. Screens allow you to provide important information, such as whether or not a valid software version was located.



Ready to Patch

This program will patch Your Product on your system from version 1.0 to version 1.1.

Please click Start to proceed with the patch.

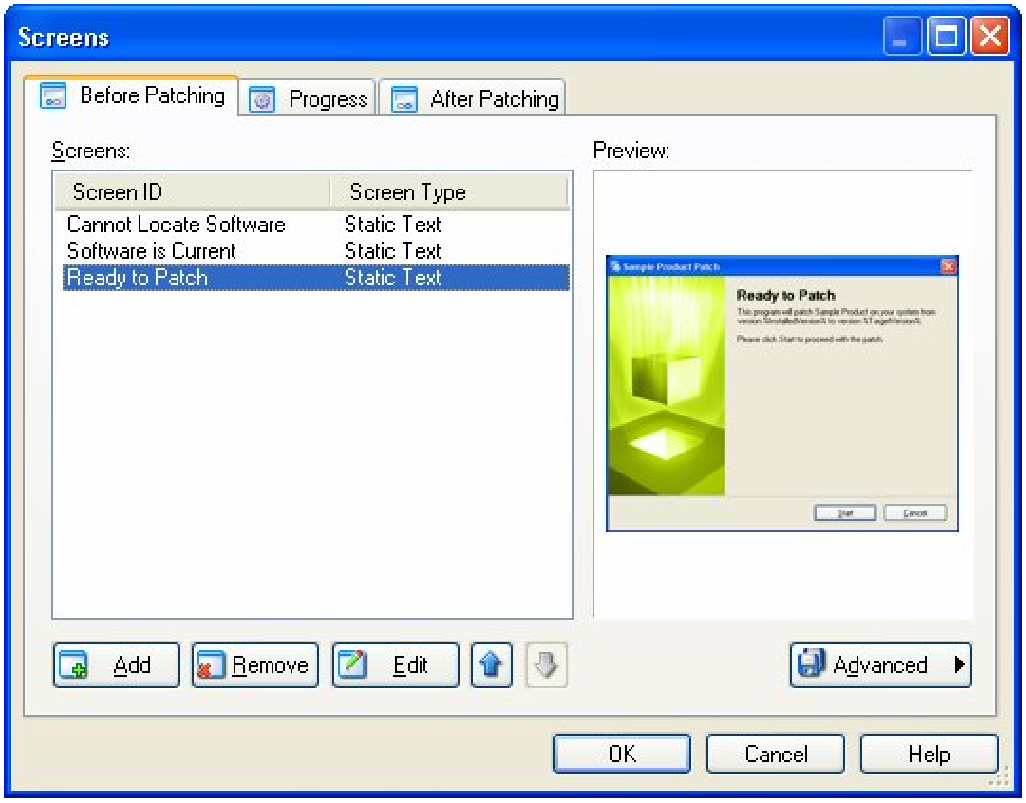
A typical 'Ready to Patch' screen

Screens are the individual windows that make up your patch. When you navigate through a patch by clicking the Next and Back buttons, you are navigating from screen to screen, exactly as you would navigate through any wizard.

Tip: You can think of the screens in your project as steps in a wizard, walking your user through the process of patching your software.

The Screen Manager

The screen manager is where you will configure all of your screens. In Visual Patch, the screens are divided into three stages.



Screen ID	Screen Type
Cannot Locate Software	Static Text
Software is Current	Static Text
Ready to Patch	Static Text

Ready to Patch

This program will patch Sample Product on your system from version %InstalledVersion% to version %TargetVersion%.

Please click Start to proceed with the patch.

Before Patching tab in the Screen Manager

The three patching stages at which screens can be displayed are:

- Before Patching
- Progress
- After Patching

Each screen stage is represented by a tab on the screen manager.

Before Patching

The Before Patching screens are displayed before the actual patching of any files occurs. In general, if the end user cancels the patch during this stage, no files will be patched on their computer.

This stage is used to perform such tasks as letting the user know what version his or her software is being patched to, displaying a license agreement, and collecting user information.

Progress

The Progress screen stage is displayed while the files on the end user's computer are being patched. This stage differs from both the Before Patching and After Patching stages in that only one screen can be displayed (as opposed to a sequence of screens), and it must be a progress screen.

This stage is used to display the file patching progress as it occurs. In other words, while the user's files are being patched, the screen you chose for the Progress stage will display information about the progress of the patch, such as what file is being patched, what percentage of all files have been patched thus far, and what percentage of the current file has been patched.

After Patching

The After Patching stage is the final screen stage and occurs after all of the files have been patched on the end user's computer.

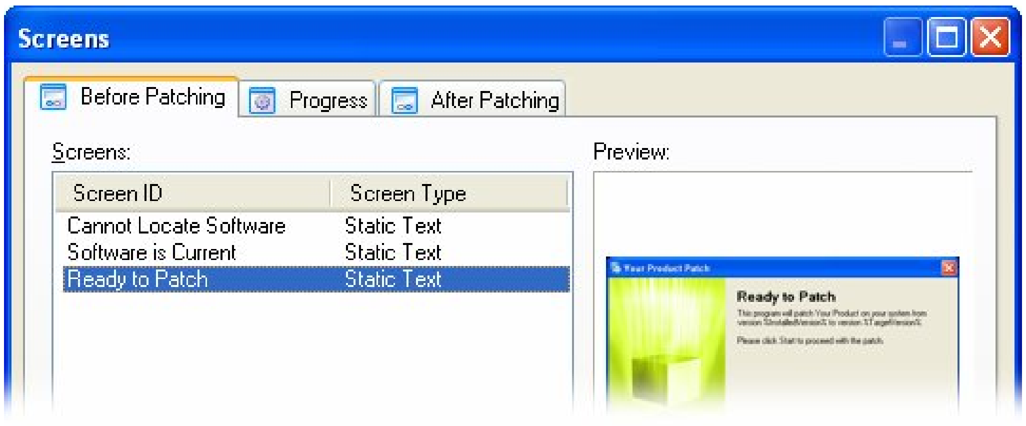
This stage is typically used to show a reboot advisement (if needed), provide any post-patch instructions, and inform the user that the patch has finished successfully.

Adding Screens

Adding a screen to your patch is easy. Select the stage where you want to add the screen, and click the Add button at the bottom of the screen manager.

Note: The Progress screen stage is different since a maximum of one screen is allowed. For this stage, the button is labeled “Change” instead of “Add.”

Clicking the Add button brings up a screen gallery where you can select from a variety of screen types. Select the type of screen you want and click the OK button to add it to the list of screens for the stage that you’re currently working on.



Screen ID	Screen Type
Cannot Locate Software	Static Text
Software is Current	Static Text
Ready to Patch	Static Text

Ready to Patch

This program will patch Your Product on your system from version %InstalledVersion% to version %TargetVersion%.

Please click Start to proceed with the patch.

The Screen Gallery

Removing Screens

To remove a screen from your patch, simply select it in the screen list and click the Remove button.

Tip: If you remove a screen from your patch by accident, you can undo the deletion by pressing Ctrl + Z, or by pressing the Cancel button.

Organizing Screens

In general, the order in which screens appear in the screen manager will be the order in which they appear during that screen stage. The screen that is at the top of the list will appear first and the screen that is at the bottom of the list will be the final screen of that stage.

To change the order of your screens, select a screen in the list and click the Up or Down button until it's in the desired location. Or, if you prefer, you can simply drag the screen from one position to another.

Tip: You can also use the cursor keys to move a screen up or down in the list. Simply select the screen you want to move, and press Alt+Up or Alt+Down.

Editing Screens

To edit a screen's properties, select it in the list and click the Edit button.

Clicking the Edit button opens the Screen Properties dialog where you can edit and customize all of the settings for that screen.

Tip: You can also edit a screen by double-clicking on it in the list.

Screen Properties

The Screen Properties dialog is where you can edit the properties of a specific screen. All Screen Properties dialogs have the same four tabs (although the specific content on these tabs may differ): Settings, Attributes, Style and Actions.

Settings

The Settings tab allows you to edit properties that are specific to the selected screen. Each screen type has different settings that are specific to that type of screen. For example, a Check Boxes screen will have settings that apply to check boxes on its Settings tab.

For more information on the specific screen settings, please see the Visual Patch help file.

Attributes

The Attributes tab contains settings that are common to all screens. The only difference that you will find between Attributes tabs is that progress screens lack options for the Next, Back, and Help buttons. (Those buttons don't exist on progress screens.)

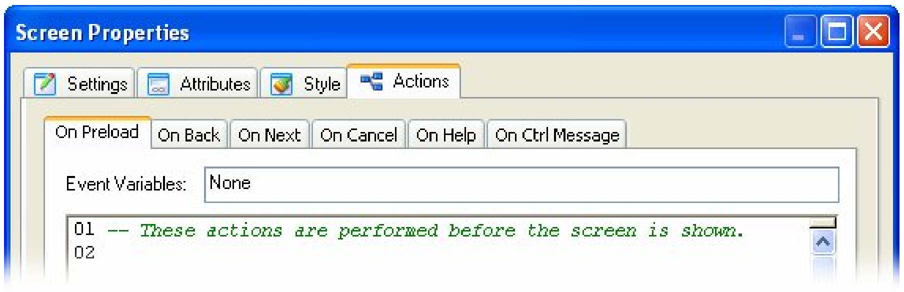
The Attributes tab is where you can configure which banner style to use, the name of the screen, and the navigation button settings.

Style

The Style tab is where you can override the project theme on a per-screen basis. By default the project theme is applied to all screens throughout your project; however, there might be some instances where you feel a certain screen needs something a bit different in order to stand out. You can use the Style tab to override any of the theme settings on a specific screen. (The changes will only be applied to that screen.)

Actions

The Actions tab is where you can edit the actions associated with the screen's events. For more information on actions please see Chapter 6.



Event Variables: None

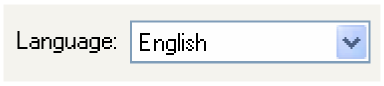
```
-- These actions are performed before the screen is shown.
```

Actions that will be performed before the screen is visible to the user

The Language Selector

The Settings and Attributes tabs both have a language selector in the bottom right corner. The language selector is a drop-down list containing all of the languages that are currently enabled in the project. It is used for creating multilingual patches.

Selecting a language in the list allows you to edit the text that will be used on the screen when that language is detected.



The language selector

Session Variables

Session variables play a large part in the way that screens work and how they display their text. Anytime you see something in Visual Patch that looks like %ProductName%, you are looking at a session variable.

Note: A session variable is essentially just a name (with no spaces) that begins and ends with %.

Session variables are very similar to normal variables in that they serve as "containers" for values that may change. We say that values are "assigned to" or "stored in" session variables. When you use a session variable, its name (e.g. %ProductName%) is replaced at run time by its value (e.g. "Visual Patch"). Session variables are basically *placeholders* for text that gets inserted later.

Session variables are often used in the default text for screens. They are automatically expanded the first time the screen is displayed, so instead of seeing %ProductName% on the screen, the end user will actually see the product name that you entered in the session variable editor (Project > Session variables).

Session variables are also used to store return values when screens or controls need them. For example, an edit field screen will use session variables to store information that the user has entered.

Tip: Session variables can be created and changed at run time using actions like SessionVar.Expand, SessionVar.Get, SessionVar.Remove, and SessionVar.Set.

For more information please see Chapter 7, which discusses session variables in more detail.

Screen Navigation

Screen navigation can be thought of as the path that the end user takes through the visible part of the patch process. The end user navigates forward through the screens by clicking the Next button, and backward through the screens by clicking the Back button.

The default screen navigation is a linear path from the top of your screen list in the screen manager to the bottom. Generally, the order of your screen list in the screen manager is exactly the order in which the navigation will proceed.

Although there are other ways to control the path through the screens (e.g. using actions to create a “branching” path), in most cases the default behavior is all that is needed.

How Screen Navigation Works

In its simplest form, screen navigation is when the end user moves forward or backward through your patch by clicking the Next and Back buttons. By default, this moves the end user down or up through the list of screens on the screen manager.

This is actually accomplished using actions. Each screen has Screen.Next and Screen.Back actions on its On Next and On Back events which are performed when the Next and Back buttons are clicked. If you ever need to, you can modify or override the default behavior of any screen by editing or replacing the default actions. Most of the time, however, you will not even need to know that the actions are there.

Navigation Buttons

Navigation buttons are the Back, Next, and Cancel buttons that are usually visible along the bottom (or “footer”) of each screen. The Next button moves the end user down the screen list from the top to the bottom, the Back button moves the end user up through the screen list, and the Cancel button stops the user’s navigation by canceling the patch.

The settings for these buttons can be found on the Attributes tab of the screen properties dialog. There you can change the text, enabled state and visible state of these buttons.

The two options for the visibility state are self-explanatory; they make the button either visible or invisible. The two options for the enabled state make the button enabled or disabled. If a button is in the enabled state, it looks and functions like a normal button; it will depress when the user clicks on it, and the text is displayed in its normal color (usually black). When a button is in the disabled state, however, it will not respond to the user's mouse, and is typically drawn in less prominent gray colors (also known as being "ghosted" or "grayed out").

Each navigation button has an event that will be fired when the button is clicked. These events can be found on the Actions tab of the screen properties dialog.

Note: A Help button is also available on the footer of each screen but is generally not considered a navigation button.

Navigation Events

An event is something that can happen during the patching process. When an event is triggered (or "fired"), any actions that are associated with that event are performed. Note that an event *must* be triggered in order for its actions to be performed.

Each event represents something that can happen while your patch executable is running. For example, all screens have an On Preload event, which is triggered just before the screen is displayed. To make something happen before a screen is displayed, you simply add an action to its On Preload event.

All of the three navigation buttons have an event that will be fired when they are clicked. The events are "On Back" for the Back button, "On Next" for the Next button and "On Cancel" for the Cancel button.

In the case of the three navigation buttons, navigation actions are executed when their respective events are fired. This allows the end user to navigate through the patch from the beginning to the end.

There are other events that are associated with screens but aren't necessarily related to screen navigation:

- On Preload – just before the screen is displayed.
- On Help – when the help button is selected.
- On Ctrl Message – triggered by a control on the screen.

Navigation Actions

There are six navigation actions available to you in Visual Patch: Screen.Back, Screen.End, Screen.Jump, Screen.Next, Screen.Previous, and Application.Exit. Of those, the most commonly used are Screen.Next and Screen.Back.

When the Next button is clicked, the user is attempting to navigate from the current screen to the next screen or phase of the patch. The easiest way to implement this behavior is to insert the Screen.Next action on the On Next event. This is done by default for all screens.

The same holds true for the Back button; when the Back button is clicked, the user is attempting to move backwards in the patch to the previous screen. To implement this behavior, a Screen.Back action needs to be executed when the On Back event is fired.

Tip: The Screen.Back action moves backward in the patch's history in the same way that a Back button does in a web browser. To move up (back) one screen in the screen list, use the Screen.Previous action.

In certain situations, simply moving down the screen list is not the appropriate behavior; instead, jumping to a specific screen in the screen stage is necessary. You can accomplish this by using a Screen.Jump action. If the goal is to jump to the next phase in the patch—i.e., to end the current screen stage—a Screen.End action can be used to jump past all of the screens in the current screen stage.

To interrupt screen navigation—which usually occurs when the Cancel button is clicked—you can use an Application.Exit action. The Application.Exit action causes your patch to stop as soon as it is performed.

Note: For more information on the specifics of screen actions, please see the help file.

Screen Layout

In Visual Patch choosing a layout for your screens and the controls on them is simple. You can switch between all three banner styles (top, side, and none) on any screen that you like. The controls on your screens will dynamically position themselves ensuring that all of your information is visible.

Note: A control can be thought of as the visible elements on a screen, from edit fields, to radio buttons, to static text controls. However, when the term "control" is used, it does not generally refer to the navigation buttons or banner text.

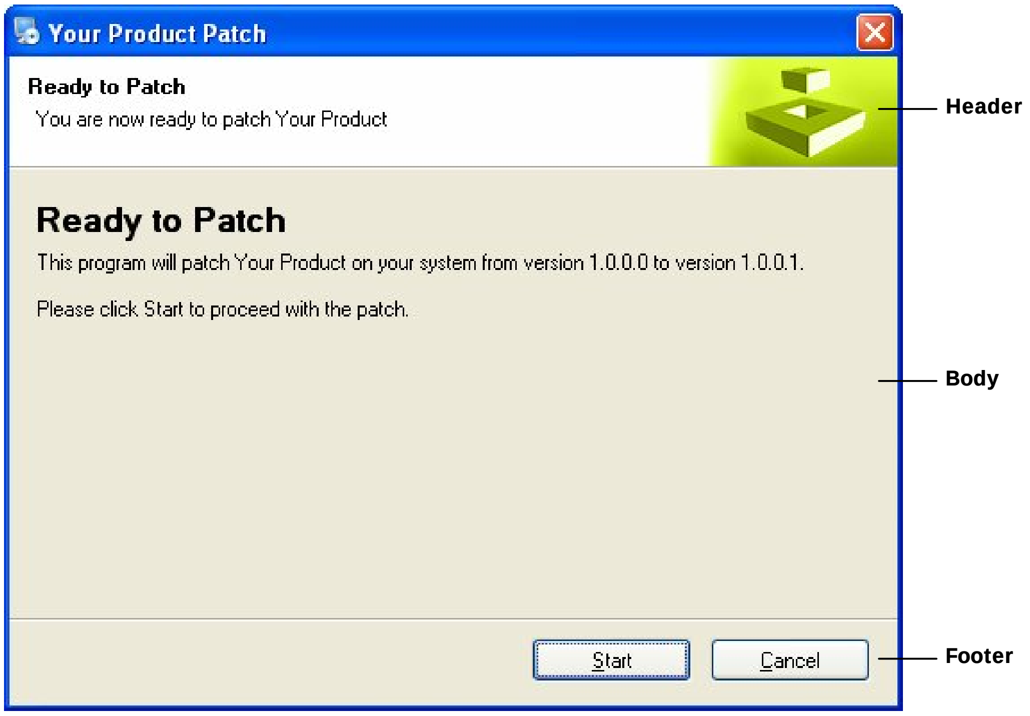
Header, Body, Footer

Screens in Visual Patch are divided into three basic parts: the header, the body and the footer.

The header runs across the top of each screen and is the area that the top banner fills.

The footer is similar to the header area except that it runs along the bottom of each screen. This is the area of the screen where the navigation buttons are placed.

The body is any part of the screen that is not taken up by the header or footer— it takes up the majority of each screen and contains most of the screen's information.



Ready to Patch

You are now ready to patch Your Product

Ready to Patch

This program will patch Your Product on your system from version 1.0.0.0 to version 1.0.0.1.

Please click Start to proceed with the patch.

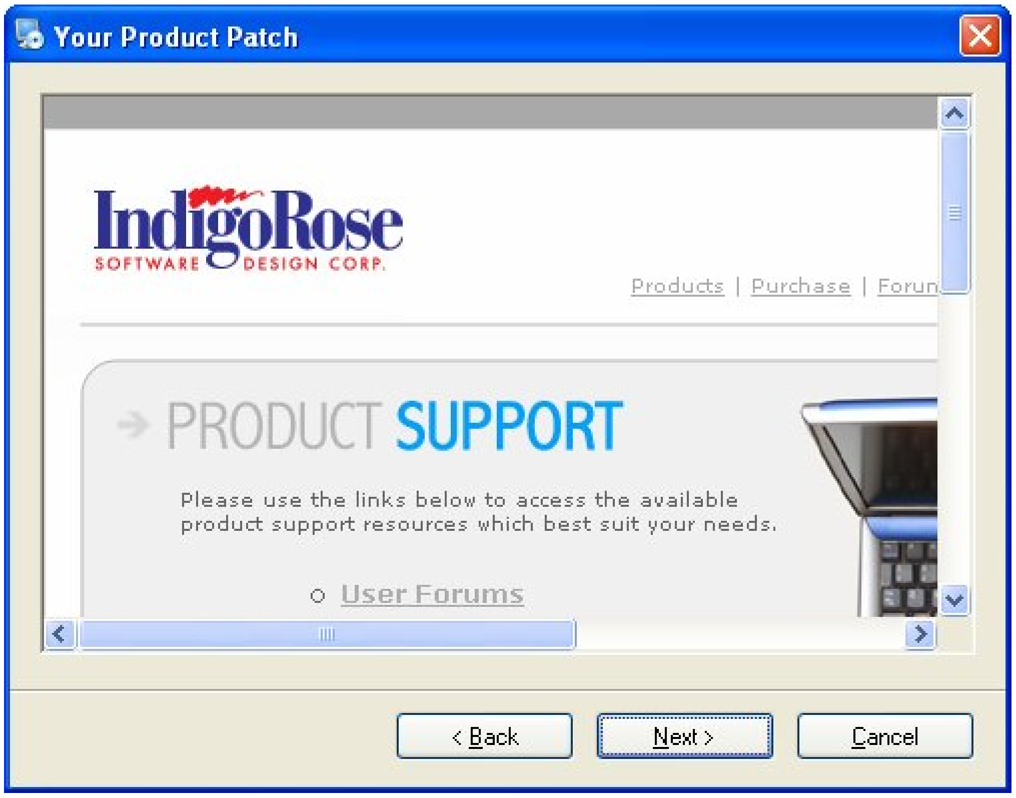
Basic parts of a screen

Banner Style

In Visual Patch the term banner refers to an area of the screen that is special and somewhat separate from the rest of the screen. You can use the banner area to display some descriptive text, an image, or both.

There are three different types of banner styles available in Visual Patch: none, top, and side.

Setting the banner style to none is the easiest of all three styles to understand since it means that there will be no banner displayed on the screen.

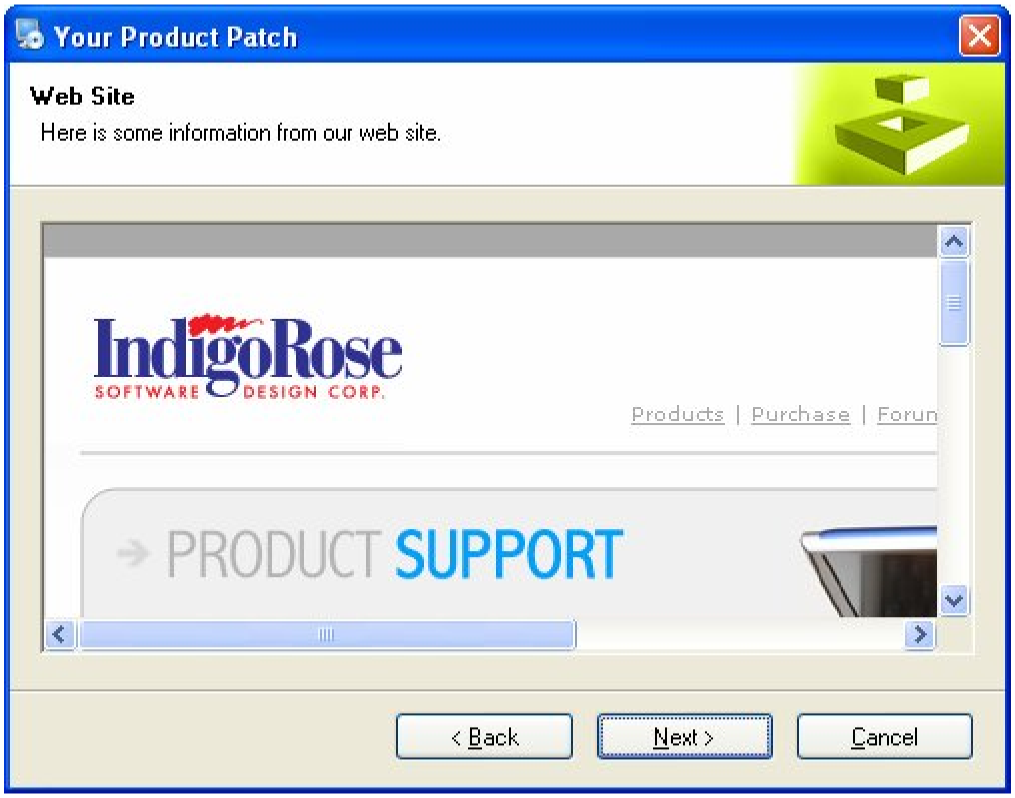

Your Product Patch
IndigoRose
SOFTWARE DESIGN CORP.
Products | Purchase | Forum
PRODUCT SUPPORT
Please use the links below to access the available product support resources which best suit your needs.
User Forums
< Back
Next >
Cancel

The 'none' banner style

The top banner style will give you a long thin banner across the top of your screen (the header). This is the style that you will probably apply to the majority of the screens in your project. The top banner style supports two lines of text referred to as the heading text and the subheading text. This text is usually used to describe the current screen and/or provide some information about what is required of the end user.

The top banner style also supports an image that will be placed on the right hand side of the banner. Visual Patch will resize this image proportionally so that its height matches the height of the top banner. The width of the image can be as wide as you want and can take up the entire banner if necessary.

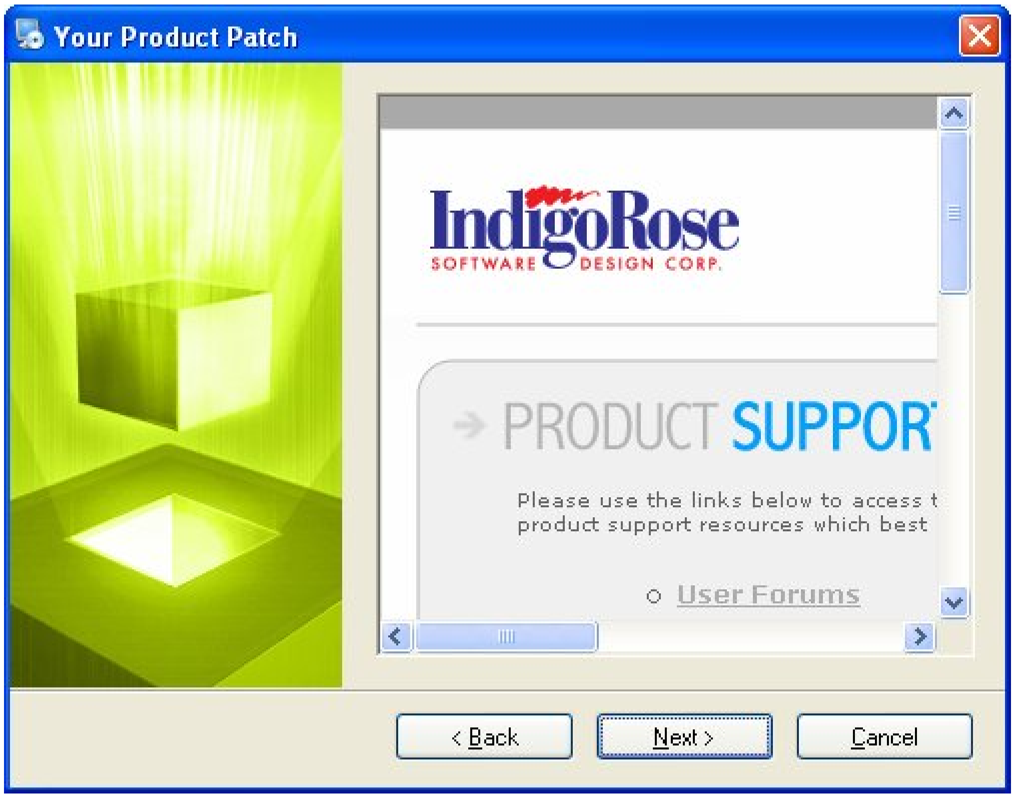
Any area of the top banner that is not covered by an image will be painted with a color according to the project theme.



The ‘top’ banner style

The side banner style will give you a thick banner that runs down the left side of your screen. The side banner will start at the top of the screen and then stop just above the screen’s footer. This is the side banner area.

The side banner style supports an image that will automatically be stretched or resized by Visual Patch to fill the entire side banner area.



The ‘side’ banner style

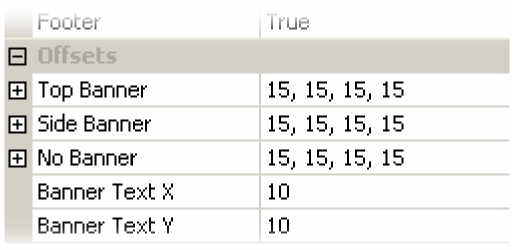
Dynamic Control Layout

One of the best features of the Visual Patch system is the dynamic control layout ability of screens. Visual Patch will dynamically reposition the controls on your screen so that the maximum amount of information stays visible.

Dynamic control layout means that controls will resize and layouts will adjust automatically as you add more text. The layout will be automatically chosen based upon your control settings.

For example, you do not have to worry about fitting your descriptive text within two or three lines. If you want a fourth (or fifth) line just type it in, and all of the controls on the screen will adjust to fit the new lines of text.

The dynamic repositioning of controls takes place within an area called the control area. The control area of a screen occupies a sub-section of the screen's body area. Its size is controlled by the theme settings.



Footer	True
⊟ Offsets	
⊞ Top Banner	15, 15, 15, 15
⊞ Side Banner	15, 15, 15, 15
⊞ No Banner	15, 15, 15, 15
Banner Text X	10
Banner Text Y	10

Control area offsets (Style tab, Screen Properties dialog)

The best part of the dynamic control layout feature is that it works without any effort on your part. Simply fill your screens with all of the information and controls that you desire, and Visual Patch will re-position all the controls so that everything is visible.

Of course, you still have full control over how controls will be displayed on your screen. Many screens (Edit Fields, Checkboxes, etc.) allow you to add as many as 32 controls to your screen, which Visual Patch will dynamically position. You have the ability to choose how many columns you want the controls displayed in, whether they are distributed horizontally or vertically and, in the case of the Edit Fields screen, how many columns each control spans!

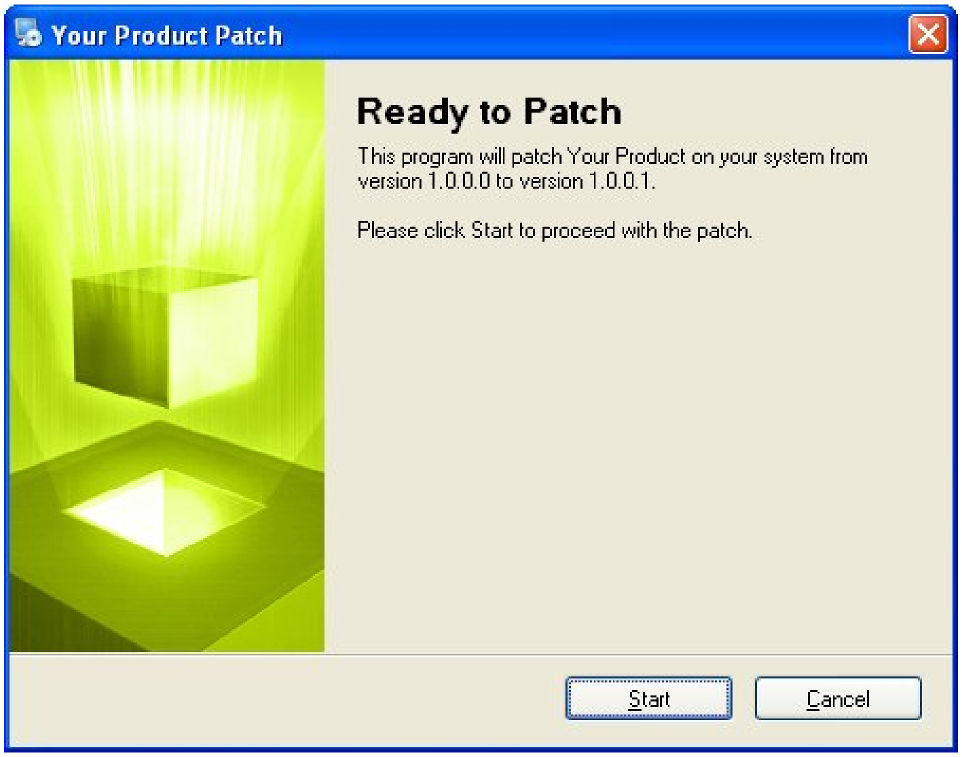
The best way to understand the dynamic control layout feature is to use it. Try playing around with the settings of a Check Boxes screen or Select Folder screen and observe how Visual Patch positions your controls to achieve the best look possible.

Themes

A theme is a group of settings and images that control the way your patch looks. You’ve probably encountered themes before when using other applications or even Windows XP. Themes do not change *what* is displayed; instead, they change *how* it is displayed.

Themes in Visual Patch control the general appearance of your screens and the controls they contain. Rather then controlling the position of screen controls or the banner style used, themes control the color and font of screens and controls. Themes are project-wide and affect all screens in the patch unless intentionally overridden on the style tab of the screen’s properties.

Themes provide an easy way to change the look and feel of your screens and controls without having to go to each screen and update it every time you wish to apply a new visual style.

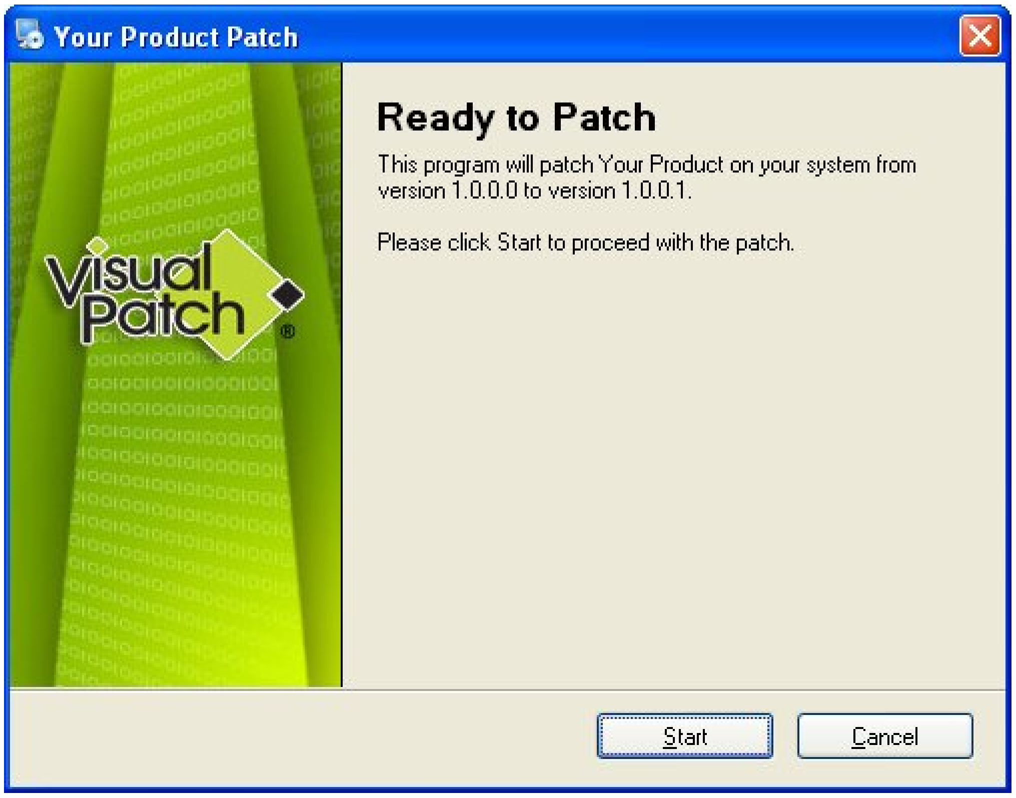


Ready to Patch

This program will patch Your Product on your system from version 1.0.0.0 to version 1.0.0.1.

Please click Start to proceed with the patch.

A Welcome screen with the "Visual Patch Default" theme applied



Ready to Patch

This program will patch Your Product on your system from version 1.0.0.0 to version 1.0.0.1.

Please click Start to proceed with the patch.

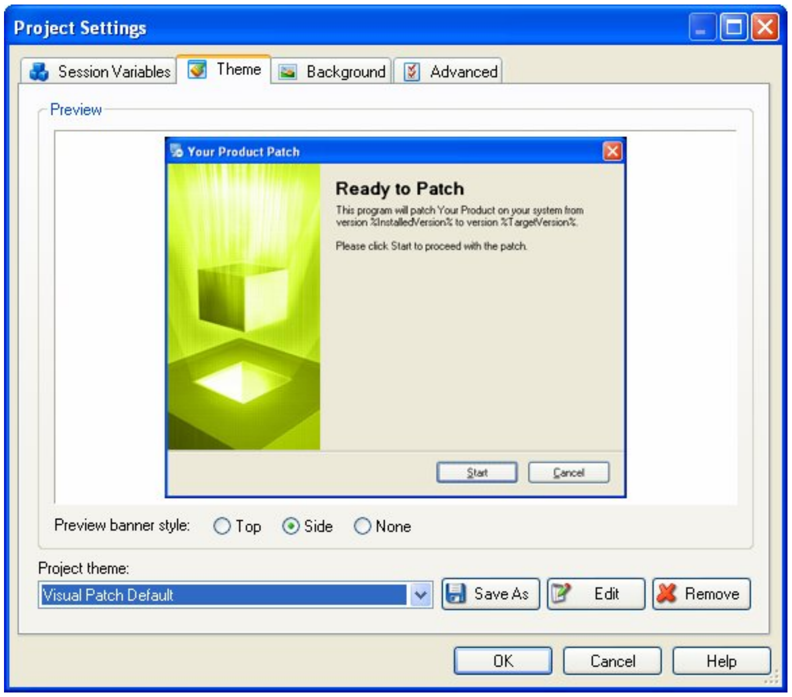
The same screen with the "Visual Patch Theme" theme applied

Choosing a Theme

You can choose a theme for your project on the theme tab of the Project Settings dialog, which you can access by choosing Project > Settings, and clicking on the Theme tab.

The drop-down list on the Theme tab contains a list of all the themes that are available in the project. Selecting a theme in this drop-down list will apply the theme to all of the screens in your project. For your convenience, a preview of the currently selected theme is displayed on the Theme tab as soon as you make a selection.

As shown above, themes affect the appearance of screens and their controls. For example, choosing a theme that colors static text controls purple will result in *all* static text controls being purple, and choosing a theme that colors static text controls black will result in *all* static text controls being colored black.



Ready to Patch

This program will patch Your Product on your system from version %InstalledVersion% to version %TargetVersion%.

Please click Start to proceed with the patch.

Preview banner style: Top [✓ Side] None

Project theme: Visual Patch Default

The Project theme selector (accessible by choosing Project > Settings)

Creating a Custom Theme

Visual Patch allows you to create your own custom themes. This provides you with an easy way to share the same custom look and feel between multiple projects.

Here is a brief step-by-step guide to help you in the creation of a custom theme.

1) Start a new project and save a copy of a pre-existing theme.

Start a new project by choosing File > New Project from the menu, then open up the theme settings by choosing Project > Settings, making sure that the theme tab is selected.

The first step in creating a custom theme is to select an existing theme to base your new theme upon. If you cannot find a suitable theme, simply choose the default theme.

Once you have selected a theme to start from, use the Save As button to save a copy of it under a new name. Choose a name that describes the theme you plan to make; this theme is what you will be modifying in order to create your new theme.

2) Edit your new theme in the Theme Properties dialog and then press Ok to save your changes.

Make sure that your new theme is selected and press the edit button to bring up the theme properties dialog. Here you will be able to edit all of the properties of your theme. Once you have made all of your changes, simply click the Ok button and the changes to your theme will automatically be saved.

Now you have a working theme that will be available to you in all your Visual Patch projects.

Note: If you are not happy with the changes made while editing your theme, simply click the Cancel button and your changes will not be saved.

Overriding Themes

As stated earlier, project themes affect every screen in your project. While in the vast majority of situations this is the desired effect, there may be a few instances where this is not exactly what you want. Fortunately, Visual Patch allows you to override any or all of the theme settings on any of your screens.

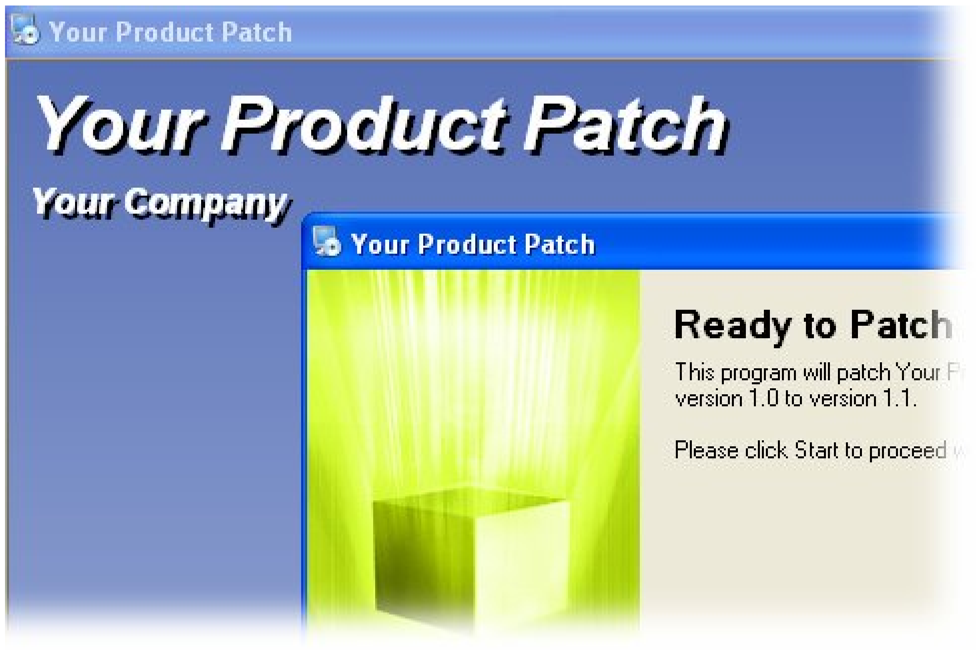
As mentioned in the Screen Properties section, each screen has a style tab associated with it. If you look at the Style tab you will notice that it looks identical to the Theme tab on the Screen Properties dialog except that it has a checkbox in the top left corner labeled “Override project theme.”

Choosing the override project theme option will enable the theme settings and allow you to make changes to the theme settings strictly for the current screen. The changes you make on the Style tab will not affect any of the other screens in your project.

Note: If you decide that you want to go back to the project theme on a screen where you have overridden it, simply go to the Style tab and uncheck the Override project theme checkbox. There is no need to re-create the screen.

The Background Window

The background window is an optional maximized window that can appear behind all screens in your patch. Its main use is to focus the end user's attention on your patch by blocking out the rest of the desktop.



This program will patch Your P
version 1.0 to version 1.1.

Please click Start to proceed

The background window behind a 'Ready To Patch' screen

The background window can be enabled in one of three styles:

- Standard – all the features of a normal application window (i.e., a title bar with a close button and a standard window border).
- Bordered – a standard background window without a title bar or close button.
- Kiosk – a standard background window without a title bar, close button, or border.

Note: The text on the window title bar is controlled by the %WindowTitle% session variable.

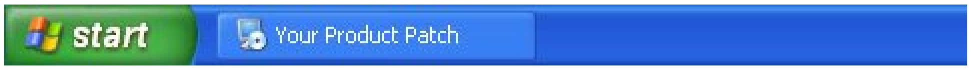
No matter which of the above styles you choose, you can configure the appearance of the background window through the Background Window tab on the Project Settings dialog (Project > Settings). You can specify what the content of the background window should be (solid color, gradient, or an image), what text will appear on the window, and how the text will look (font, size, and style). You have the option of having a heading, subheading, or footer in any combination, or none at all.

Other Interface Options

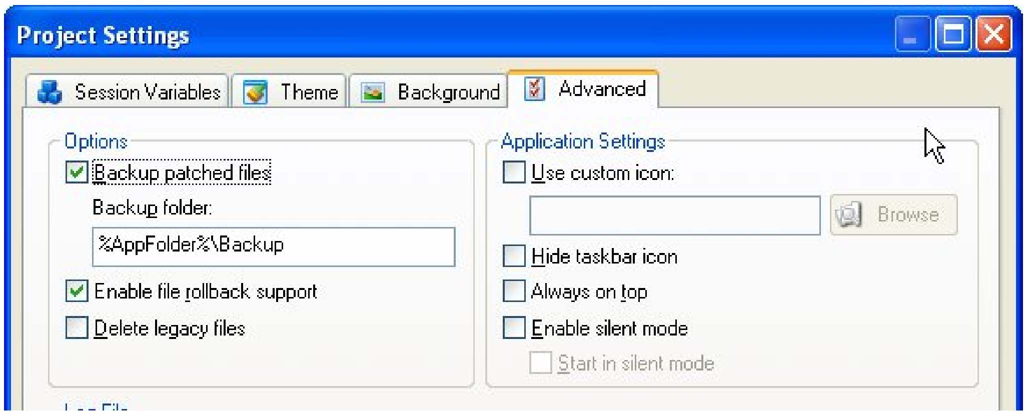
There are a few other options in Visual Patch that relate to the user interface of your patch. These options may not be as important as screens or themes, but they just might provide the elements necessary to perfect your project's look and feel.

Taskbar Settings

The taskbar is the bar that runs across the bottom of all modern Windows operating systems beginning with the START button on the left. When a program is running, its icon and name will generally appear in a button in the taskbar.



Visual Patch allows you to choose whether or not to show an icon in the task bar and to choose what that icon will be. Both of these settings can be found on the Advanced tab of the project settings: Project > Settings.



Options
[✓] Backup patched files
Backup folder: %AppFolder%\Backup
[✓] Enable file rollback support
[ ] Delete legacy files

Application Settings
[ ] Use custom icon:
[ ] Hide taskbar icon
[ ] Always on top
[ ] Enable silent mode
[ ] Start in silent mode

Taskbar Settings available in the Project Settings dialog

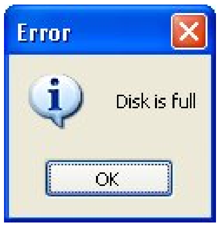
To hide the taskbar icon simply check the hide taskbar icon checkbox on the Advanced tab. To choose a custom icon simply check the use custom icon checkbox to turn on the custom icon setting and then click the browse button to locate the icon of your choice.

Note: If you configure your patch to be a silent patch in the advanced settings then no taskbar entry will appear.

Actions

Some of the actions available to you in Visual Patch are capable of showing user interface elements. These actions can be divided up into two main categories: Dialog actions and Status Dialog actions.

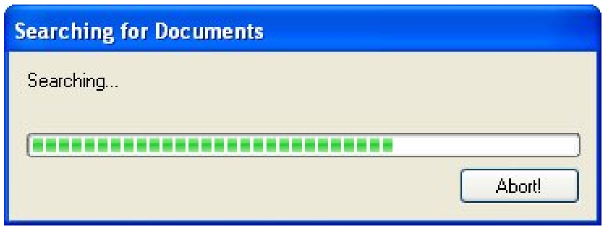
Dialog actions are used to show pop up dialogs to the end user. Examples include the Dialog.Message action that lets you display a message in a dialog, and the Dialog.TimedMessage action that lets you show a dialog with a message for specific amount of time.



Disk is full

A typical message dialog

Status dialogs are the other main user interface elements that are available to you through scripting. Status dialogs are mainly used to show progress or status during a lengthy event like an HTTP.Download action or a File.Find action.



A status dialog

Status dialogs are shown and configured using actions like StatusDlg.SetMessage, StatusDlg.ShowProgressMeter, and StatusDlg.Show.

Note: It is generally recommended that progress be shown in a more integrated manner using a progress screen, however there are situations where a status dialog may be more appropriate.

Note: For more information on the Dialog and StatusDlg actions, please see the Visual Patch help file.

Chapter 6:

Actions, Scripts and Plugins

Visual Patch comes standard with a plethora of actions allowing you to create more powerful patches than ever before. They are what you use to accomplish the specific tasks that make your patch unique. Each action is a specific command that tells your patch to do something, such as retrieving a value from the Registry.

Actions also allow your patch to react to different situations in different ways. Does the user only have the evaluation version of your software installed? Is an Internet connection available? You can use actions to answer these types of questions and have your patch respond accordingly.

Scripts are essentially sequences of actions that work together to perform a specific task. Plugins allow you to extend the built-in actions with additional libraries of commands.

Together, actions, scripts and plugins allow you to extend the default functionality of Visual Patch with virtually limitless possibilities.

6

In This Chapter

In this chapter, you'll learn about:

- Actions—what they are and what they're good for
- The action editor (including features such as syntax highlighting, intellisense, quickhelp, and context-sensitive help)
- Project and screen events
- Adding and removing actions
- Scripting basics, such as using a variable, adding an if statement, testing a numeric value, using a for loop, and creating functions
- Global functions
- Plugins
- External script files

What are Actions?

Actions are specialized commands that your patch can perform at runtime. Each action is a short text instruction that tells the patch to do something—whether it's to open a document, download a file from the web, create a shortcut, or modify a registry key.

Actions are grouped into categories like "File" and "Registry." The category and the name of the command are joined by a decimal point, or "dot," like so: File.Run, Registry.GetValue. The text "File.Run" essentially tells Visual Patch that you want to perform a "Run" command from the "File" category…a.k.a. the "File.Run" action.

Tip: The period in an action name is either pronounced "dot," as in "File-dot-Open," or it isn't pronounced at all, as in "File Open."

By default Visual Patch handles the most common patch needs without you ever having to add additional actions. As a result you will only need to use actions when you want to supplement the basic behavior and handle advanced patch tasks. Actions let you customize what Visual Patch does for you automatically so that you can meet your patch needs exactly.

You can use actions to call functions from DLLs, submit data to a website, start and stop programs, get the amount of free space on a drive, and much more.

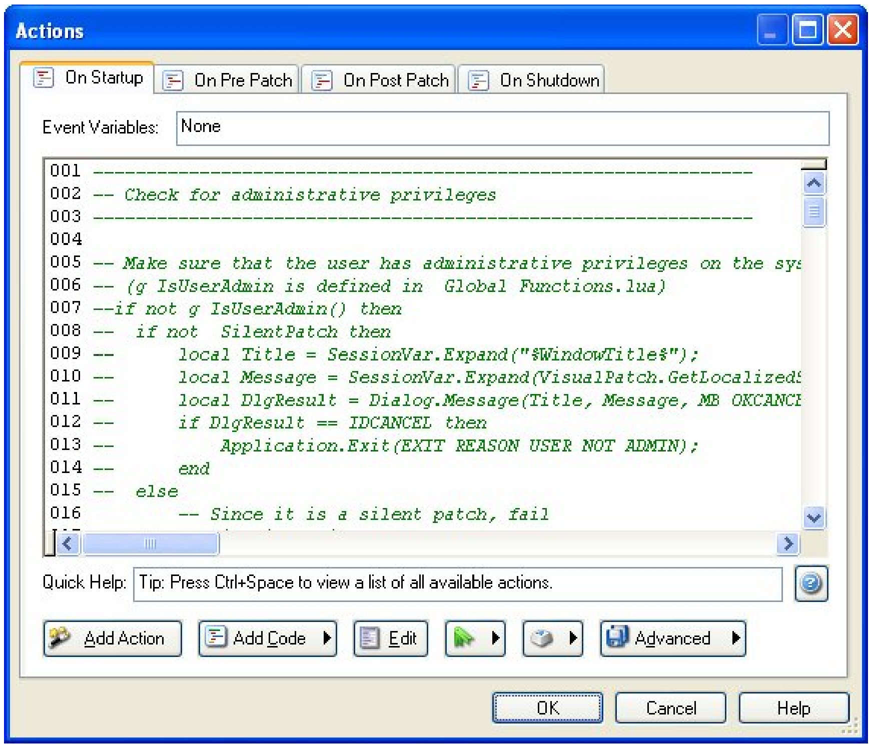
Note: You can find a complete list of actions in the Visual Patch help file (Help > Visual Patch Help).

Note: In order to try out the example scripts in this chapter, you will need a basic project that you can build so you can run the patch application and see the scripts in action. Please see Chapter 10 for more information.

The Action Editor

The action editor is where you create and edit your actions in the Visual Patch design environment. In general you can expect to find an action editor for every event in Visual Patch.

Essentially the action editor functions like a text editor by allowing you to type the actions that you want to use.



Event Variables: None

```
--------------------------------------------------------------
-- Check for administrative privileges
--------------------------------------------------------------

-- Make sure that the user has administrative privileges on the sys
-- (g IsUserAdmin is defined in  Global Functions.lua)
--if not g IsUserAdmin() then
--  if not  SilentPatch then
--      local Title = SessionVar.Expand("$WindowTitle$");
--      local Message = SessionVar.Expand(VisualPatch.GetLocalizedS
--      local DlgResult = Dialog.Message(Title, Message, MB OKCANCE
--      if DlgResult == IDCANCEL then
--          Application.Exit(EXIT REASON USER NOT ADMIN);
--      end
--  else
        -- Since it is a silent patch, fail
```

Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

Visual Patch's action editor

However, while it may function like a text editor, the action editor has powerful features that make it closer to a full-fledged programming environment. Some of these features include syntax highlighting, intellisense, quickhelp and context-sensitive help.

One of the most important features of the action editor is the *action wizard*. The action wizard provides an easy dialog-based way to select, create and edit actions without ever having to type a line of script.

The action editor gives you the best of both worlds: pure scripting capabilities for advanced users and programmers, and the easy-to-use action wizard interface for those who’d rather not use free form scripting.

Programming Features

As mentioned briefly above, the action editor provides some powerful features that make it a useful and accessible tool for programmers and non-programmers alike. Along with the action wizard (covered later under *Adding Actions*), the four most important features of the action editor are: syntax highlighting, intellisense, quickhelp and context-sensitive help.

Syntax Highlighting

Syntax highlighting colors text differently depending upon syntax. This allows you to identify script in the action editor as an operator, keyword, or comment with a quick glance.

Note: You can customize the colors used for syntax highlighting via the action editor settings. The action editor settings are accessed via the advanced button: Advanced > Editor Settings.

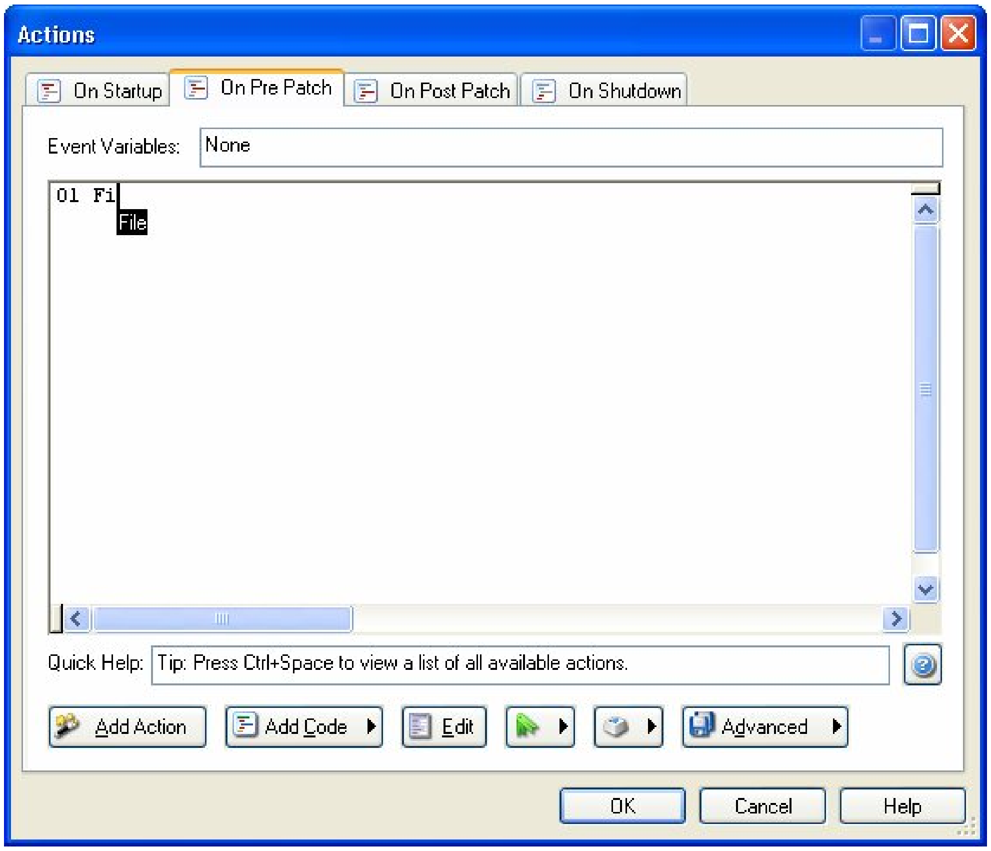
Intellisense

Intellisense is a feature of advanced programming environments that refers to the surrounding script and the position of the cursor at a given moment to provide intelligent, contextual help to the programmer.

Intellisense is a term that has been used differently by different programs. In Visual Patch, intellisense manifests itself as two features: autocomplete, and the autocomplete dropdown.

Autocomplete is the editor’s ability to automatically complete keywords for you when you press Tab. As you type the first few letters of a keyword into the action editor, a black tooltip will appear nearby displaying the whole keyword. This is the intellisense feature at work.

Whenever you type something that the intellisense recognizes as a keyword, it will display its best guess at what you are typing in one of those little black tooltips. Whenever one of these tooltips is visible, you can press the Tab key to automatically type the rest of the word.



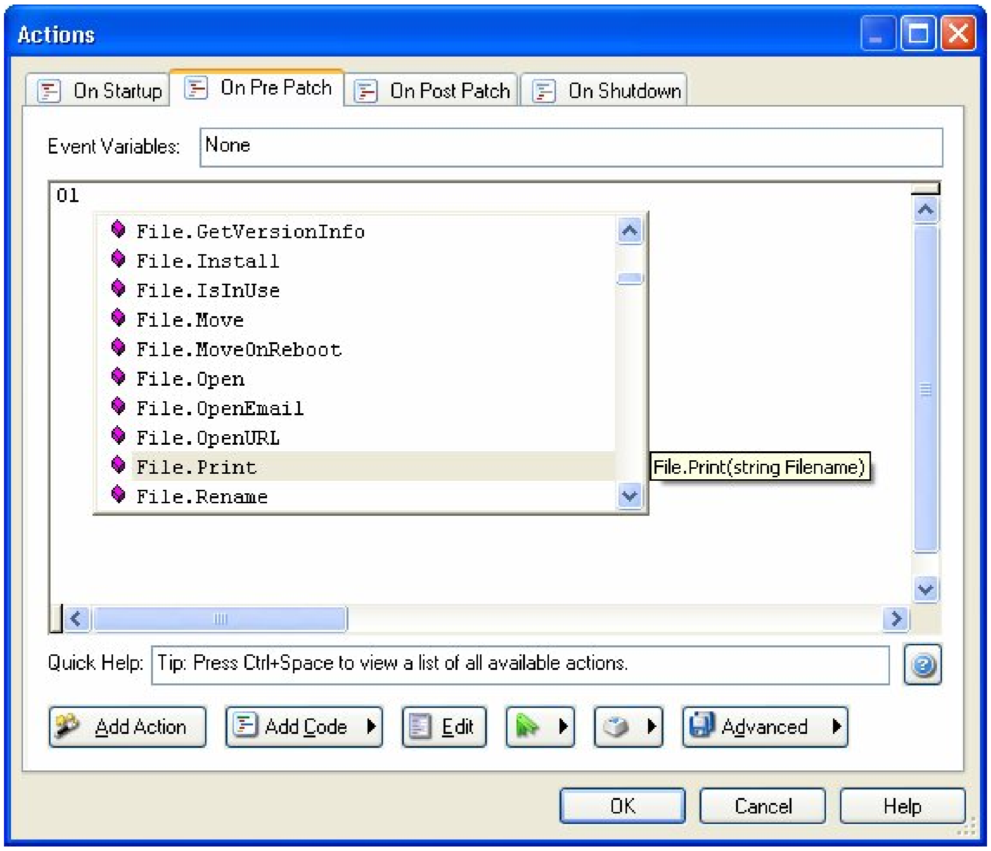
Event Variables: None

```
01 Fi
   File
```

Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

Intellisense predicting what is being typed

Another feature of intellisense is the autocomplete dropdown. By pressing Ctrl+Space while your cursor is in the code window, a drop-down list will appear containing the names of all available actions, constants and global variables. You can choose one of the listed items and then press Tab or Enter to have it automatically typed.



Event Variables: None

```
01
File.GetVersionInfo
File.Install
File.IsInUse
File.Move
File.MoveOnReboot
File.Open
File.OpenEmail
File.OpenURL
File.Print
File.Rename
```

File.Print(string Filename)

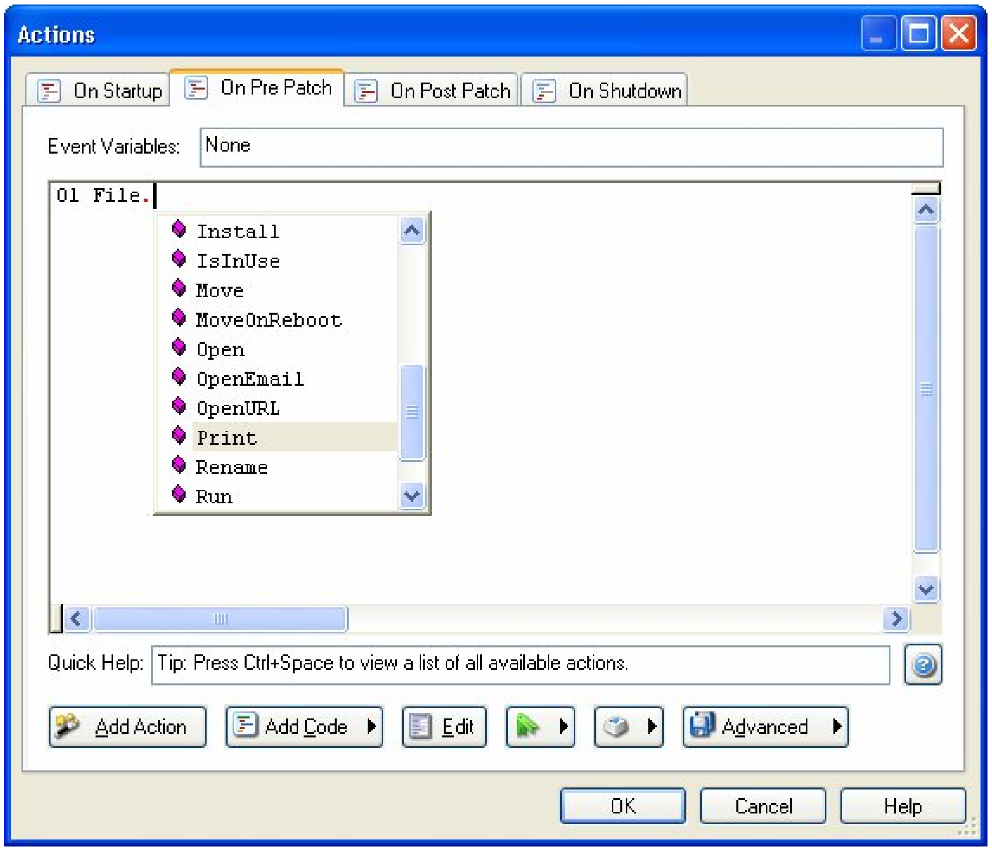
Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

The autocomplete dropdown list accessed by Ctrl + Space

Note: This dropdown cannot be accessed if your cursor is inside a set of quotes (a string).

The autocomplete dropdown is also available for completing action names after the category has been typed.

For example, when you type a period after the word File, the intellisense recognizes what you’ve typed as the beginning of an action name and presents you with a drop-down list of all the actions that begin with “File.”



Event Variables: None

```
01 File.
```

- Install
- IsInUse
- Move
- MoveOnReboot
- Open
- OpenEmail
- OpenURL
- Print
- Rename
- Run

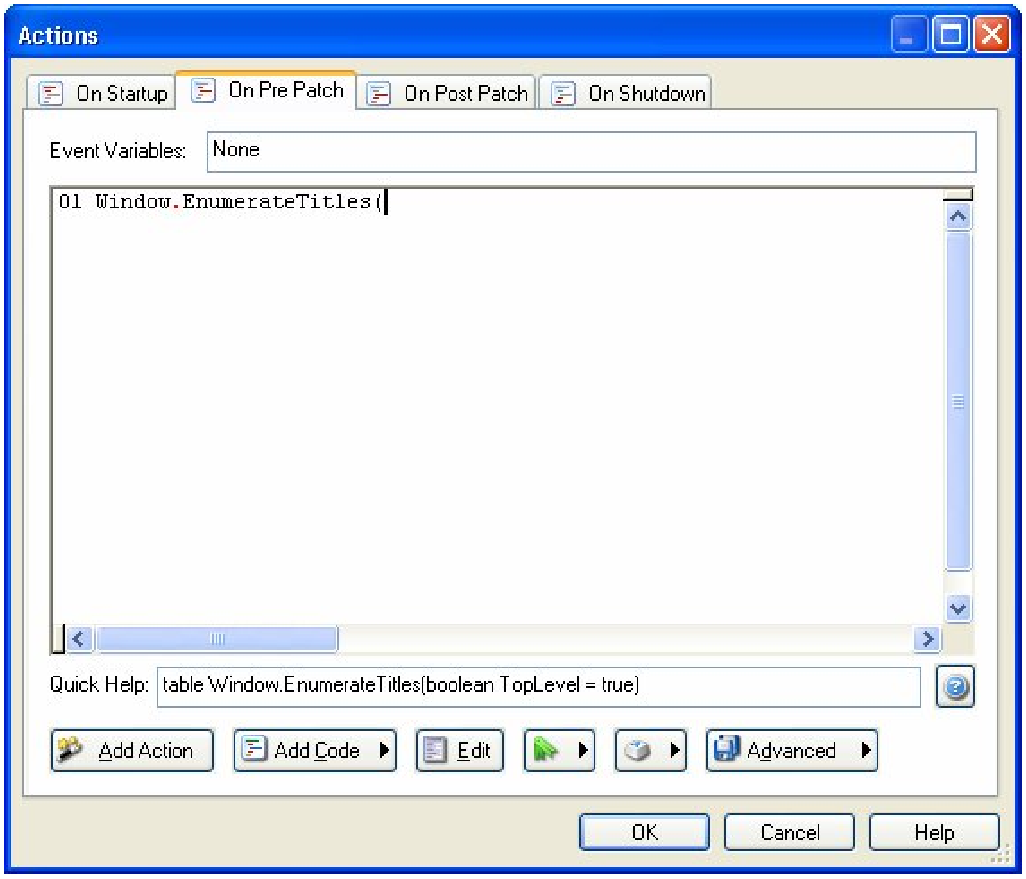
Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

The autocomplete dropdown list automatically appearing as an action is typed

The word will automatically be typed for you if you choose it and then press Tab or Enter. However, you don't have to make use of the dropdown list; if you prefer, you can continue typing the rest of the action manually.

Quickhelp

Once you’ve typed something that the action editor recognizes as the name of an action, the quickhelp feature appears near the bottom of the window.



Event Variables: None

```
Window.EnumerateTitles(
```

Quick Help: table Window.EnumerateTitles(boolean TopLevel = true)

Quickhelp for the action Window.EnumerateTitles

Quickhelp is essentially a “blueprint” for the action that lists the names of the action’s parameters and indicates what type of value is expected for each one. In the case of our Window.EnumerateTitles action, the quickhelp looks like this:

```
table Window.EnumerateTitles(boolean TopLevel = true)
```

The quickhelp above indicates that the action Window.EnumerateTitles takes a single parameter called TopLevel, and that this parameter needs to be a boolean value. It also indicates that Window.EnumerateTitles returns a table value.

From this brief example, you can see how useful the quickhelp feature will be when you are working on your scripts.

Context-sensitive Help

Context-sensitive help, as its name suggests, provides help for you based upon what you are currently doing. In the action editor, the context sensitive help lets you jump directly to the current action's help topic.

For instance, if you are typing an action into the action editor and the quickhelp feature isn't giving you enough information (perhaps you would like to see an example), press the F1 key and the help file will open directly to that action's help topic.

Note: The context-sensitive help feature is only available when the action editor recognizes the action that the cursor is on. It is easy to know when this is the case since the action will appear in the quickhelp.

Tip: You can also click the Help button to trigger the context-sensitive help feature.

Events

Events are simply things that can happen when your patch is running. For example, all screens have an On Preload event, which is triggered just before the screen is displayed. To make something happen just before the screen is displayed, you simply add an action to its On Preload event.

When an event is triggered, the actions (or "script") associated with that event will be executed. By adding actions to different events, you can control when those actions occur during the patch.

In Visual Patch there is an action editor for every event. When you add an action in an event's action editor, you are adding that action to the event. At run time, when that event is triggered, the action that you added will be performed.

Every patch has four main project events: On Startup, On Pre Patch, On Post Patch, and On Shutdown. These events can be thought of as bookends to the main phases of the patch. The remaining events in Visual Patch are triggered by the screens.

Project Events

There are four project events. They are the main events of your patch and will be triggered before or after important stages of the patch. The project events can be found under Project > Actions.

The four project events are:

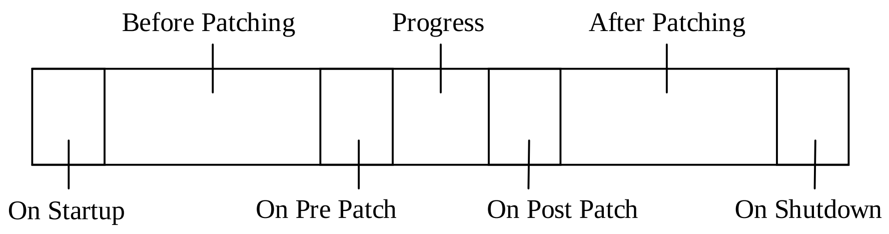
- On Startup
- On Pre Patch
- On Post Patch
- On Shutdown

The On Startup event is the first event triggered in Visual Patch and thus occurs before any screens are displayed. This is a good place to perform pre-patch tasks, such as making sure the end user has a previous version of your product installed, or that they haven't already run the patch.

The On Pre Patch event is triggered just before the patch enters the patching phase and begins patching files on the user's computer. This event is fired right after the last screen in the Before Patching screen stage is displayed.

The On Post Patch event is triggered after the patching phase of the patch has been completed, right before the first screen of the After Patching screen stage is displayed.

The On Shutdown event is the last event that will be triggered in your patch. It occurs after all screens in your patch have been displayed. It is your last chance to accomplish any action in your patch.



Visual Patch's screen stages (top) and project events (bottom)

Screen Events

The settings for each screen can be configured in the screen properties dialog. The screen events can be found on the Actions tab of the screen properties dialog.

Screen events are events that are triggered either by the controls on the screen or by the screen itself. Screen events are used for screen navigation (e.g. moving to the next screen), to change what is displayed on the screen (e.g. updating progress text), and to perform the task required of the screen (e.g. set the patch folder).

There are two main types of screens in Visual Patch: progress screens and non-progress screens. The majority of screens are non-progress screens, which basically means that these screens are not used to show progress.

Progress screens are used to show progress in your patch. They can be further broken down into two types of progress screens: those used in the Before Patching and After Patching screen stages, and those used in the Progress stage. In general progress screens used in the Before and After Patching screen stages show progress that arises from actions. Progress screens that are used during the While Patching screen stage are used to show the progress of the main patch phase.

There are six events associated with non-progress screens:

- On Preload – triggered just before the screen is going to be displayed.
- On Back – when the back button is clicked.
- On Next – when the next button is clicked.
- On Cancel – when the cancel button is clicked.
- On Help – when the help button is clicked.
- On Ctrl Message – Triggered when a control on the screen fires a control message. Every time a control message event is fired, there will be event variables passed to the event to describe the message and which control fired it.

The On Ctrl Message event is where you will interact with the controls on the screens via script. For example, if you choose to add a Button screen to your patch, the On Ctrl Message event is where you will specify what each button does.

On the Buttons screen, each button will fire a control message when it has been clicked, basically saying “Hey I’ve been clicked!” You will then have a script in the

On Ctrl Message event that will check the event variables to see which button has been clicked and the corresponding actions will be performed.

There are four events associated with progress screens used in the Before and After Patching screen stages:

- On Preload – triggered just before the screen is going to be displayed.
- On Start – fired as soon as the screen is displayed. This is where you will put all of the actions that are to occur while the progress screen is visible. The actions placed here are what will cause the progress that the progress screen is going to display.
- On Finish – triggered as soon as all of the On Start's event actions have finished executing. This will usually be used to move to the next screen.
- On Cancel – triggered when the cancel button is pressed.

There are three events associated with progress screens used in the While Patching screen stage:

- On Preload – triggered just before the screen is going to be displayed.
- On Progress – fired as progress is made during the patch phase. Event variables are passed to this event to describe which stage is triggering the progress event and other information.
- On Cancel – triggered when the cancel button is pressed.

Adding Actions

In order to have an action performed when an event is triggered, you must add it to that event using the action editor.

Actions can either be typed directly, or you can use the Add Action button to start the action wizard. The action wizard is a dialog-based way for you to add actions in the action editor. It will guide you through the process of selecting your action and configuring its parameters.

Note: As you type an action, you can use the intellisense and code completion features to simplify this process.

Here is a brief example that shows how easy it is to add an action. It will explain how to add a Dialog.Message action to the On Startup event of your patch. The Dialog.Message action pops up a dialog with a message on it.

1) Start a new project and open the Actions dialog to the On Startup tab.

Start a new project and bring up the Actions dialog by choosing Project > Actions. On the actions dialog, select the On Startup tab so that you can edit the On Startup event.

2) Click the Add Action button. When the action wizard appears, switch to the Dialog category and then click on the action called Dialog.Message.

The action wizard will walk you through the process of adding an action to the action editor. The first step is to choose a category using the drop-down list.

When you choose the “Dialog” category from the drop-down list, all of the actions in that category will appear in the list below.

To select an action from the list, just click on it. When you select an action in the list, a short description appears in the area below the list. In this description, the name of the action will appear in blue. You can click on this blue text to get more information about the action from the online help.

3) Click the Next button and configure the parameters for the Dialog.Message action.

Parameters are values that get “passed” to an action. They tell the action what to do. For instance, in the case of our Dialog.Message action, the action needs to know *what* the title of the dialog and the message should be. The first parameter lets you specify the title of the dialog and the second parameter lets you specify the text that will be displayed on the dialog. For now the other parameters are not important but you should take some time to look at them and their options. By default Visual Patch will fill most parameters with defaults.

For now change the title to:

```
"Visual Patch"
```

and the text to:

```
"Message from Chapter 6!"
```

Note: Be sure to include the quotes on either side of both parameters. These are string parameters and the quotes are needed for Visual Patch to properly interpret them.

Once you've set the action's parameters, click the Finish button to close the action wizard. The Dialog.Message action will appear in the action editor. It will look like this:

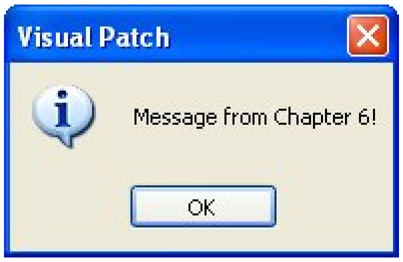
```
Dialog.Message("Visual Patch", "Message from Chapter 6!");
```

4) Build and run your project. When the project starts you should see the dialog created by the Dialog.Message action.

Start the publish wizard by selecting Publish > Build from the menu and click the Next button. Build your patch to whichever folder you want and using whichever name you like. Once the build has completed successfully, make sure that you check the open output folder checkbox and then click the Finish button.

Note: You must have at least 2 version tabs in your project (containing at least one file each) in order to successfully build your project.

Once the folder where you built your patch appears, double-click on your patch executable to launch it. You should see the following dialog message appear:



Message from Chapter 6!

Editing Actions

Often you will want to change a few of your actions' settings after you add them; to do this you need to edit the action. You can edit the action by typing directly into the action editor or by using the actions properties dialog.

The easiest way to bring up the actions properties dialog is by double-clicking on the action. If you prefer you can bring it up by placing the cursor in the action editor and then pressing the edit action button. (You can tell that the cursor is in an action when the actions function prototype appears in the quickhelp.)

Here is a quick example illustrating how to edit the Dialog.Message action that we created in the previous section.

1) Open the Actions dialog to the On Startup tab and bring up the Action Properties dialog.

Open the Actions dialog by choosing Project > Actions. Make sure you are on the On Startup tab and that you see the Dialog.Message action created in the previous topic.

To edit the action, just double-click on it. Double-clicking on the action opens the Action Properties dialog, where you can modify the action's current parameters.

Tip: You can also edit the action's text directly in the action editor, just like you would edit text in a word processor. For example, you can select the text that you want to replace, and then simply type some new text in.

2) Change the Type parameter to be MB_OKCANCEL and the Icon parameter to MB_ICONNONE.

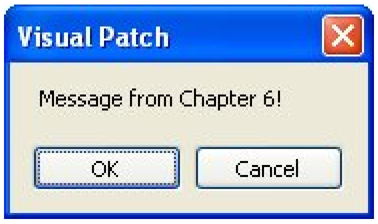
Click on the Type parameter, click the select button, and choose MB_OKCANCEL from the drop-down list. Then click on the Icon parameter, click the select button, and choose MB_ICONNONE from the drop-down list. This will add a cancel button to the dialog (MB_OKCANCEL) and get rid of the icon (MB_ICONNONE). Finally, click OK to finish editing the action. Notice that the changes that you made now appear in the action editor.

Constants

SW_SHOWNORMAL is a *constant*. A constant is a name that represents a value, essentially becoming an "alias" for that value. Constants are often used to represent numeric values in parameters. It's easier to remember what effect SW_MAXIMIZE has than it is to remember what happens when you pass the number 3 to the action.

3) Build and run your project. When the project starts, you should see the dialog created by the Dialog.Message action.

When you run your patch, a dialog will pop up. The dialog is created by the Dialog.Message action on the On Startup event of your patch. Notice that there is no icon and that a cancel button has been added the dialog.


Visual Patch

Message from Chapter 6!

Getting Help on Actions

You can get help on actions in a variety of different ways in Visual Patch. As previously mentioned, when using the action wizard some text will be displayed in the bottom of the dialog describing the current action or parameter selected. You can also click the blue hyperlink text in the action wizard to receive context-sensitive help.

The same holds true for the action editor itself. The quickhelp will display the current action's function prototype. The help button to the right of the quickhelp will open the Help file directly to that action's help topic if there is a current action (as with the F1 key). The current action is simply the action that the cursor is in. If the cursor is not in an action, then there is no current action.

Note: For more information on the quickhelp, please see the Action Editor Features section above.

The Help button on the action editor can be clicked at any time to open the Visual Patch help file. Here you can view more information on the event that the actions are being added to.

The help file contains detailed information for each action available to you in Visual Patch. It is divided into two topics for each action: Overview and Examples. The overview section provides detailed information about the action while the examples section provides one or more working examples of that action.

In the overview topic, the help file will provide you with the function prototype, which serves as a definition of the action, showing what (if anything) the action returns, and the parameters and their types.

A function prototype is the definition of the action. It defines the types of all of the parameters, the type of the return value (if any), and whether or not any of the parameters have defaults.

```
number File.Run ( string  Filename,
                  string  Args = "",
                  string  WorkingFolder = "",
                  number  WindowMode = SW_SHOWNORMAL,
                  boolean WaitForReturn = false )
```

The help file also describes each parameter in detail and what its purpose is. For example, if the action returns a value, the help file will describe the return value and what it might be.

Scripting Basics

A script can be thought of as a sequence of actions. Scripting, therefore, is basically the creation of a sequence of actions. A script can contain one action or as many actions as you can type.

Although you can accomplish a lot in Visual Patch without any scripting knowledge, even a little bit of scripting practice can make a big difference. You can accomplish far more in a project with a little bit of scripting than you ever could without it. Scripting opens the door to all sorts of advanced techniques, from actions that are only performed when specific conditions are met, to functions that you can define, name and then call from somewhere else.

Using a Variable

One of the most powerful features of scripting is the ability to make use of variables. Variables are essentially just “nicknames” or “placeholders” for values that may need to be modified or re-used in the future. Each variable is given a name that you can use to access its current value in your script.

We say that values are “assigned to” or “stored in” variables. If you picture a variable as a container that can hold a value, assigning a value to a variable is like “placing” that value into a container.

You place a value into a variable by assigning the value to it with an equals sign. For example, the following script assigns the value 10 to a variable called “amount.”

```
amount = 10;
```

Note: The semi-colon at the end of the line tells Visual Patch where the end of the statement is. It acts as a *terminator*. (No relation to Arnold, though.) Although technically it’s optional—Visual Patch can usually figure out where the end of the statement is on its own—it’s a good idea to get in the habit of including it, to avoid any potential confusion.

You can change the value in a variable at any time by assigning a different value to it. (The new value simply replaces the old one.)

For example, the following script assigns 45 to the amount variable, replacing the number 10:

```
amount = 45;
```

...and the following script assigns the string "Woohoo!" to the variable, replacing the number 45:

```
amount = "Woohoo!";
```

Note that you can easily replace a numeric value with a string in a variable. Having a number or a string in a variable doesn't “lock” it into only accepting that type of value—variables don’t care what kind of data they hold.

This ability to hold changeable information is what makes variables so useful.

Here is an example that demonstrates how variables work:

1) Start a new project and follow the steps from the Adding an action example on page 139.

Start a new Visual Patch project by selecting File > New Project from the menu. If you have any unsaved changes in your current project, Visual Patch will prompt you to save them. If the project wizard dialog comes up, simply hit cancel to get rid of it for now.

Once you have a brand new project, follow along with the three steps in the Adding an Action example.

2) In the action editor, replace the “Message from Chapter 6!” string with a variable named strMsg.

Just edit the script on the On Startup event so that it looks like this instead:

```
Dialog.Message("Visual Patch", strMsg);
```

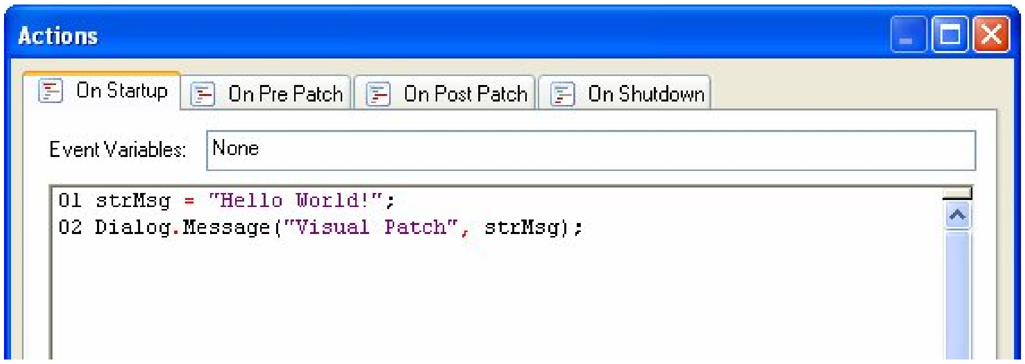
This will make the Dialog.Message action display the current value of the strMsg variable when it is performed. Before we try it out, though, we need to assign a value to that variable somewhere.

Note: A common practice among programmers is to give their variables names with prefixes that help them remember what the variables are supposed to contain. This practice is often referred to as Hungarian notation. One such prefix is “str,” which is used to indicate a variable that contains a string. Other common prefixes are “n” for a numeric value (e.g. nCount, nTotal) and “b” for a Boolean true/false value (e.g. bLoaded, bReadyToStart).

3) Insert the following text on the first line, before the Dialog.Message action:

```
strMsg = "Hello World!";
```

Remember to press Enter at the end of the line to move the Dialog.Message action to the next line. The script should look like this when you're done:



Event Variables: None

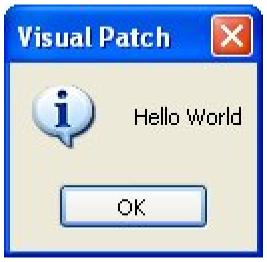
```
strMsg = "Hello World!";
Dialog.Message("Visual Patch", strMsg);
```

An On Startup script that will display a message dialog when the patch is run

This will assign the string "Hello World!" to the variable named strMsg before the Dialog.Message action is executed.

The On Startup event is triggered as soon as the patch begins, just before the first screen is displayed. This makes it a good place to put any initialization script, such as setting up default values or preparing variables that will be used in the patch. Another good location is the Global Functions section that will be described later.

4) Build and run your project. When the project starts, you should see the dialog created by the Dialog.Message action.



Hello World

Note that the variable's name, strMsg, is nowhere to be found. Instead, the value that is currently in the variable is displayed. In this case, it's the value that was assigned to the variable in the project’s On Startup event.

Note: You must have at least 2 version tabs in your project (containing at least one file each) in order to successfully build your project.

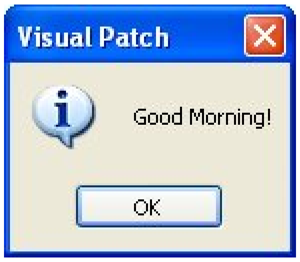
5) Exit the patch. In the action editor, change the value that is assigned to the strMsg variable to *"Good Morning!"*.

Edit the On Startup script so it looks like this instead:

```
strMsg = "Good Morning!";
Dialog.Message("Visual Patch", strMsg);
```

6) Build and run your project. When the project starts, you should see the dialog created by the Dialog.Message action.

This time, the message looks like this:



As you can see, you've just changed the message without even touching the Dialog.Message action.

Now imagine if you had the same or similar Dialog.Message actions on fifty different events throughout your project, and you decided you wanted to change the text. With variables and a little planning, such changes are a piece of cake.

Adding an If Statement

The if statement gives your scripts the ability to make decisions, and do one thing or another in different circumstances.

Each if statement consists of a *condition* (or “test”), followed by a *block* of script that will only be performed if the condition is found to be true.

The basic syntax is:

if *condition* then
        do something here
end

For example:

```
if age < 18 then
    Dialog.Message("Sorry!", "You must be 18 to access this CD.");
    Application.Exit();
end
```

The above script checks to see if the value in a variable named “age” is less than 18. If it is, it puts up a message saying "You must be 18 to access this CD," and then immediately exits from the application.

For example, if we set age to 17 first:

```
age = 17;
if age < 18 then
    Dialog.Message("Sorry!", "You must be 18 to access this CD.");
    Application.Exit();
end
```

...the block of script between the “then” and “end” keywords will be performed, because 17 is less than 18. In this case, we say that the if statement’s condition “passed.”

However, if we set age to 20:

```
age = 20;
if age < 18 then
    Dialog.Message("Sorry!", "You must be 18 to access this CD.");
    Application.Exit();
end
```

...the block of script between the “then” and the “end” isn’t performed, because 20 isn’t less than 18. This time, we say that the if statement’s condition “failed.”

Note: An if statement's condition can be any expression that evaluates to true or false.

Let's modify the script on our project’s On Startup event to only display the message if it is “Hello world!”

1) Open the On Startup event and edit the script to enclose the Dialog.Message action in an if statement, like this:

```
strMsg = "Good Morning!";
if strMsg == "Hello World!" then
    Dialog.Message("Visual Patch", strMsg);
end
```

The double equals compares for absolute equality, so this if statement's condition will only be true if strMsg contains "Hello World!" with the exact same capitalization and spelling.

Tip: An easy way to add an if statement on the action editor is to highlight the line of text that you want to put "inside" the if, click the Add Code button, and then choose "if statement" from the menu. An if statement "template" will be added around the line that you highlighted. You can then edit the template to fit your needs.

2) Press F7 to build your project. When the build is complete, launch the patch to trigger the script.

This time nothing happens because strMsg is still set to "Good Morning!" in the first line of the On Startup event. "Good Morning!" doesn't equal "Hello World!" so the if condition fails, and the block of code between the "then" and "end" keywords is skipped entirely.

3) Exit from the patch. Edit the On Startup script to change the == to ~=.

The script should look like this when it's done:

```
strMsg = "Good Morning!";
if strMsg ~= "Hello World!" then
    Dialog.Message("Visual Patch", strMsg);
end
```

The tilde equals (~=) compares for inequality (does not equal), so this if statement's condition will only be true if strMsg contains anything *but* "Hello World!" with the exact same capitalization and spelling.

4) Build and execute the project.

This time, because strMsg contains "Good Morning!", which is definitely not equal to "Hello World!", the message will appear.

The == and ~= operators are fine when you want to check strings for an exact match. But what if you’re not sure of the capitalization? What if the variable contains a string that the user typed in, and you don’t care if they capitalized everything correctly?

One solution is to use an old programmer’s trick: just convert the contents of the unknown string to all lowercase (or all uppercase), and do the same to the string you want to match.

Let's modify our script to ask the user for a message, and then display what they typed, but only if they typed the words "hello world" followed by an exclamation mark.

5) Exit from the patch. Edit the project’s On Startup script to look like this:

```
strMsg = Dialog.Input("", "Enter your message:");
if String.Upper(strMsg) == "HELLO WORLD!" then
    Dialog.Message("Visual Patch", strMsg);
else
    Dialog.Message("Um...", "You didn't type Hello World!");
end
```

The first line uses a Dialog.Input action to pop up a message dialog with an input field that the user can type into. Whatever the user types is then assigned to the strMsg variable.

In the if statement’s condition, we used a String.Upper action to convert the contents of strMsg to all uppercase characters, and then compare that to "HELLO WORLD!".

We’ve added an “else” keyword after the first Dialog.Message action. It basically divides the if statement into two parts: the “then” part, that only happens if the condition is true; and the “else” part, that only happens if the condition is false. In this case, the else part uses a Dialog.Message action to tell the user that they didn’t type the right message.

6) Build and run the patch. Try typing different messages into the input field by closing and re-running the patch.

Depending on what you type, you’ll either see the "Hello World!" message, or "You didn’t type Hello World!".

Note that if you type some variation of "Hello World!", such as "hello world!", or "hEllO WorlD!", the capitalization that you type will be preserved in the message that appears. Even though we used a String.Upper action to convert the message string to all uppercase letters in the if statement's condition, the actual contents of the variable remain unchanged. When the String.Upper action converts a value to all uppercase letters, it doesn't change the value in the variable...it just retrieves it, converts it, and passes it along.

Now you know how to compare two strings without being picky about capitalization.

Tip: You can also use a String.CompareNoCase action to perform a case-insensitive comparison.

Testing a Numeric Value

Another common use of the if statement is to test a numeric value to see if it has reached a certain amount. To demonstrate this, let's create another script that will stop your patch from running after it has been run on the same system more then 5 times.

1) Add the following script to a new project's On Startup event:

```
nRuns = Application.LoadValue("VPUsersGuide", "Chapter6");
if nRuns == "" then
    nRuns = 0;
end

nRuns = nRuns + 1;

Application.SaveValue("VPUsersGuide", "Chapter6", nRuns);

if nRuns>5 then
   Dialog.Message("Sorry!", "Patch has been run: ".. nRuns
                            .. " times and will exit");
   Application.Exit();
end
```

The first line tries to read an application value using the Application.LoadValue action. It stores the result in a variable named nRuns.

The second line tests the return value of the Application.LoadValue action. If a blank string has been returned ("") then this is the first time that the patch has been executed on this computer. As a result the third line then assigns a value of 0 to nRuns. The fourth line simply completes the if statement.

The fifth line adds 1 to the current value contained in the variable nRuns, and the sixth line saves that number using the Application.SaveValue action.

The rest of the script is just an if statement that tests whether the value of nRuns is greater than 5, displays a message when that's true, and then exits the patch.

We used the concatenation operator to join the current value of nRuns to the end of the string "Patch has been run: " and then used them again to add the string " times and will exit" to the end. Note that this string has a space at the end of it, so there will be a space between the colon and the value of nRuns. (The resulting string just looks better that way.)

The concatenation operator consists of two periods, or dots (..). In fact, it's often referred to as the "dot-dot" operator. It is used to join two strings together to form a new, combined string. It's kind of like string glue.

Note: Technically, the value inside nRuns isn't a string—it's a number. That doesn't really matter, though. When you're doing concatenation, Visual Patch will automatically convert a number into the equivalent string, as required.

2) Build the project and then run it six times.

Each time you run the project the value of nRuns will be loaded, incremented by one, and then saved again. After you have run the patch six times (and all times after that) the message will appear and the patch will stop.

There you go. You've just created a pretty sophisticated little program.

Using a For Loop

Sometimes it's helpful to be able to repeat a bunch of actions several times, without having to type them over and over and over again. One way to accomplish this is by using a *for* loop.

The basic syntax of the for loop is:

```
for variable = start,end,step do
        do something here
end
```

For example:

```
for n = 10,100,10 do
    Dialog.Message("", n);
end
```

The above script simply counts from 10 to 100, going up by 10 at a time, displaying the current value of the variable n in a dialog message box. This accomplishes the same thing as typing:

```
Dialog.Message("", 10);
Dialog.Message("", 20);
Dialog.Message("", 30);
Dialog.Message("", 40);
Dialog.Message("", 50);
Dialog.Message("", 60);
Dialog.Message("", 70);
Dialog.Message("", 80);
Dialog.Message("", 90);
Dialog.Message("", 100);
```

Obviously, the for loop makes repeating similar actions much easier.

Note: If the step is missing from the for loop, it will default to 1.

Let's use a simple for loop to add all of the digits between 1 and 100, and display the result.

1) Start a new project and add the following script to the project's On Startup event:

```
n = 0;
for i = 1, 100 do
    n = n + i;
end

Dialog.Message("", "The sum of all the digits is: " .. n);
```

The first line creates a variable called n and sets it to 0. The next line tells the for loop to count from 1 to 100, storing the current "count" in a variable named i.

During each "pass" through the loop, the script between the "do" and the "end" will be performed. In this case, this consists of a single line that adds the current values of n and i together, and then stores the result back into n. In other words, it adds i to the current value of n.

This for loop is the same as typing out:

```
n = n + 1;
n = n + 2;
n = n + 3;
n = n + 4;
```

...all the way up to 100.

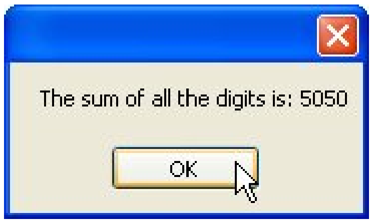
The last line of our script displays only the result of the calculation to the user in a dialog message box.

Tip: You can use the Add Code button to insert an example for loop, complete with comments explaining the syntax. You can then edit the example to fit your own needs.

2) Build and run the patch.

When you run the patch, the for loop will blaze through the calculation 100 times, and then the Dialog.Message action will display the result.

It all happens *very* fast.



The sum of all the digits is: 5050

3) Exit the patch.

You probably won't find yourself using for loops as often as you'll use if statements, but it's definitely worth knowing how to use them. When you need to repeat steps, they can save you a lot of effort.

Of course, when it comes to saving you effort, the real champions are functions.

Creating Functions

A function is just a portion of script that you can define, name and then call from somewhere else.

You can use functions to avoid repeating the same script in different places, essentially creating your own custom “actions”—complete with parameters and return values if you want.

In general, functions are defined like this:

```
function function_name (arguments)
        function script here
        return return_value;
end
```

The “function” keyword tells Visual Patch that what follows is a function definition. The *function_name* is simply a unique name for the function. The *arguments* part is the list of parameters that can be passed to the function when it is called. It’s essentially a list of variable names that will receive the values that are passed. (The resulting variables are local to the function, and only have meaning within it.) A function can have as many arguments as you want, even none at all.

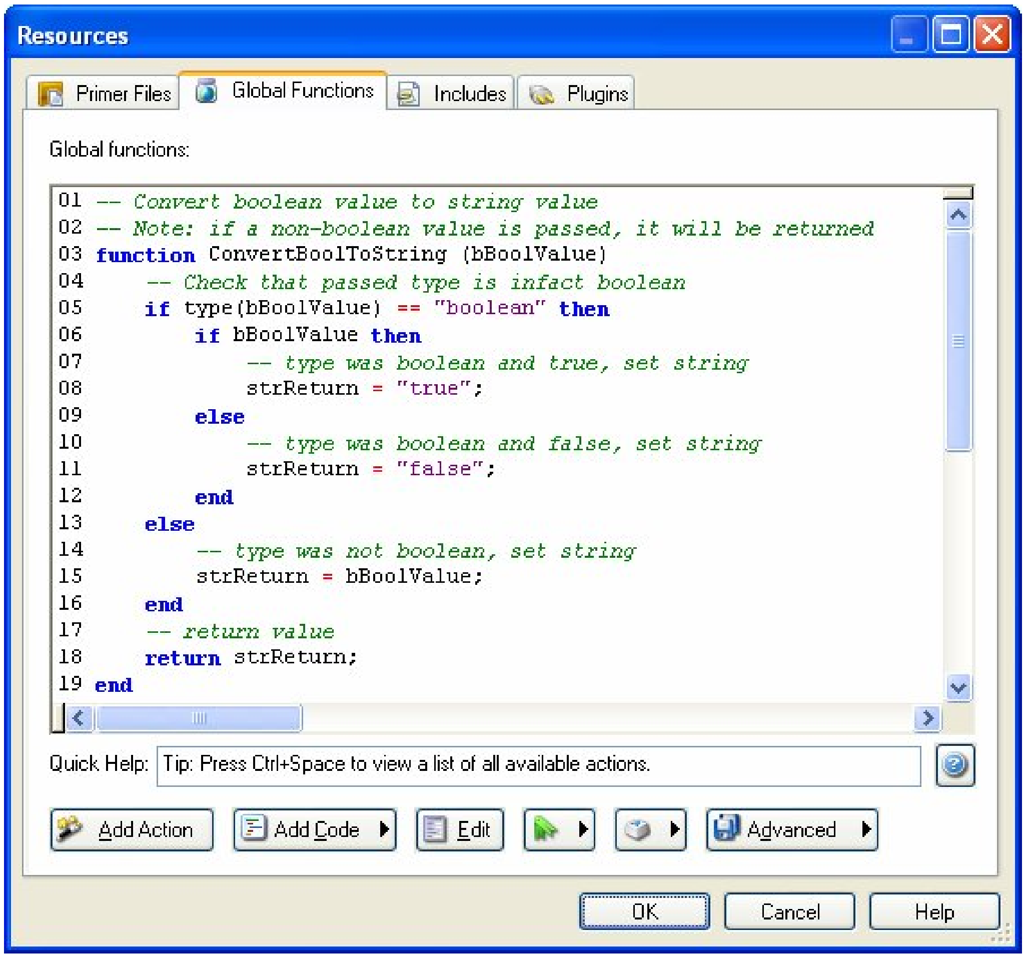
The “return” keyword tells the function to return one or more values back to the script that called it.

The easiest way to learn about functions is to try some examples, so let’s dive right in.

1) Start a new project and then choose Project > Resources.

This opens the resources dialog. Click on the Global Functions tab.

The global functions section is a convenient place to put any functions or variables that you want to make available throughout your project. Any script that you add to this section will be performed when your application is launched, right before the project's On Startup event is triggered.



```
-- Convert boolean value to string value
-- Note: if a non-boolean value is passed, it will be returned
function ConvertBoolToString (bBoolValue)
    -- Check that passed type is infact boolean
    if type(bBoolValue) == "boolean" then
        if bBoolValue then
            -- type was boolean and true, set string
            strReturn = "true";
        else
            -- type was boolean and false, set string
            strReturn = "false";
        end
    else
        -- type was not boolean, set string
        strReturn = bBoolValue;
    end
    -- return value
    return strReturn;
end
```

Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

Making the function ConvertBoolToString available to the entire project

2) Type the following script into the Global Functions tab, then click OK to close the dialog:

```
function SayHello(name)
    Dialog.Message("", "Hello " .. name);
end
```

This script defines a function named SayHello that takes a single argument (which we've named "name") and displays a simple message.

Note that this only *defines* the function. When this script is performed, it will "load" the function into memory, but it won't actually display the message until the function is called.

Once you've entered the function definition, click OK to close the Global Functions dialog.

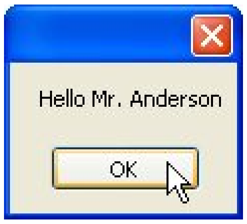
3) On the project's On Startup event add the following script:

```
SayHello("Mr. Anderson");
```

This script *calls* the SayHello function that we defined on the global functions section, passing the string "Mr. Anderson" as the value for the function's "name" parameter.

4) Build and run the patch.

When you run the patch, the script on the On Startup event calls the SayHello function, which then displays its message.



Hello Mr. Anderson

Note that the SayHello function was able to use the string that we passed to it as the message it displayed.

5) Exit the patch. Choose Resources > Global Functions, and add the following script below the SayHello function:

```
function GetName()
    local name = Dialog.Input("", "What is your name:");
    return name;
end
```

When you're done, click OK to close the dialog.

This script defines a function called GetName that does not take any parameters. The first line inside the function uses a Dialog.Input action to display a message dialog with an input field on it asking the user to type in their name. The value returned from this action (i.e., the text that the user entered) is then stored in a local variable called name.

The “local” keyword makes the variable only exist inside this function. It’s essentially like saying, “for the rest of this function, whenever I use ‘name’ I’m referring to a temporary local variable, and not any global one.” Using local variables inside functions is a good idea—it prevents you from changing the value of a global variable without meaning to. Of course, there are times when you *want* to change the value of a global variable, in which case you just won’t use the “local” keyword the first time you assign anything to the variable.

The second line inside the function returns the current value of the local “name” variable to the script that called the function.

Tip: We could actually make this function’s script fit on a single line, by getting rid of the variable completely. Instead of storing the return value from the Dialog.Input action in a variable, and then returning the contents of that variable, we could just put those two statements together, like so:

```
function GetName()
    return Dialog.Input("", "What is your name:");
end
```

This would make the GetName function return the value that was returned from the Dialog.Input action, without storing it in a variable first.

6) Edit the script in the project’s On Startup event so it looks like this instead:

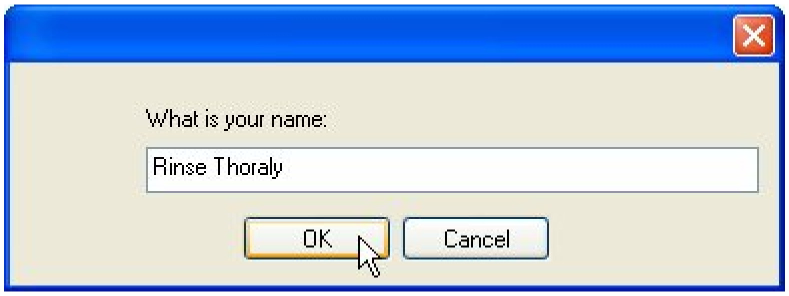
```
strName = GetName();
SayHello(strName);
```

The first line calls our GetName function to ask the user for their name, and then stores the value returned from GetName in a variable called strName.

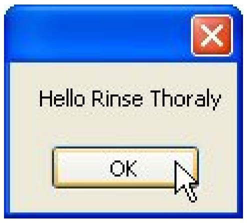
The second line passes the value in strName to our SayHello function.

7) Build and launch the patch.

When the patch begins the On Startup script will automatically be executed and the input dialog will appear, asking you to enter your name.



After you type in your name and click OK (or press Enter), a second dialog box will appear, greeting you by the name you entered.



Pretty neat, huh?

8) Exit the patch. Edit the script in the project's On Startup event so it looks like this:

```
SayHello(GetName());
```

This version of the script does away with the strName variable altogether. Instead, it uses the return value from our GetName function as the argument for the SayHello function.

In other words, it passes the GetName function's return value directly to the SayHello function.

Whenever a function returns a value, you can use a call to the function in the same way you would use the value, or a variable containing the value. This allows you to use the return value from a function without having to come up with a unique name for a temporary variable.

9) Build and launch the patch again to try out the script again. When you’re done, exit the patch.

The script should work exactly the same as before: you’ll be asked for your name, and then greeted with it.

This is just a simple example, but it should give you an idea of what an incredibly powerful feature functions are. With them, you can condense large pieces of script into simple function calls that are much easier to type and give you a single, central location to make changes to that script. They also let you create flexible “subroutines” that accept different parameters and return results, just like the built-in Visual Patch actions.

And despite all that power, they are really quite simple to use.

Action Resources

There are three additional resources at your disposal that can be useful when you are working with actions in Visual Patch: Global Functions, Plugins, and Script Files.

Global Functions

The Global Functions resource in Visual Patch can be found in the design environment by choosing Project > Resources and clicking the Global Functions tab. The Global Functions resource is an action editor that will let you enter script.

The script in the global functions section will be loaded into the patch before the On Startup event is executed. This makes it a great place to create any functions that you want to have available throughout your project and a great place to initialize any global variables.

Note that any script in the Global Functions resource will be executed, so it is generally best to only use it for variable initialization and function declarations. It is a good idea to place all other scripts on events within your patch.

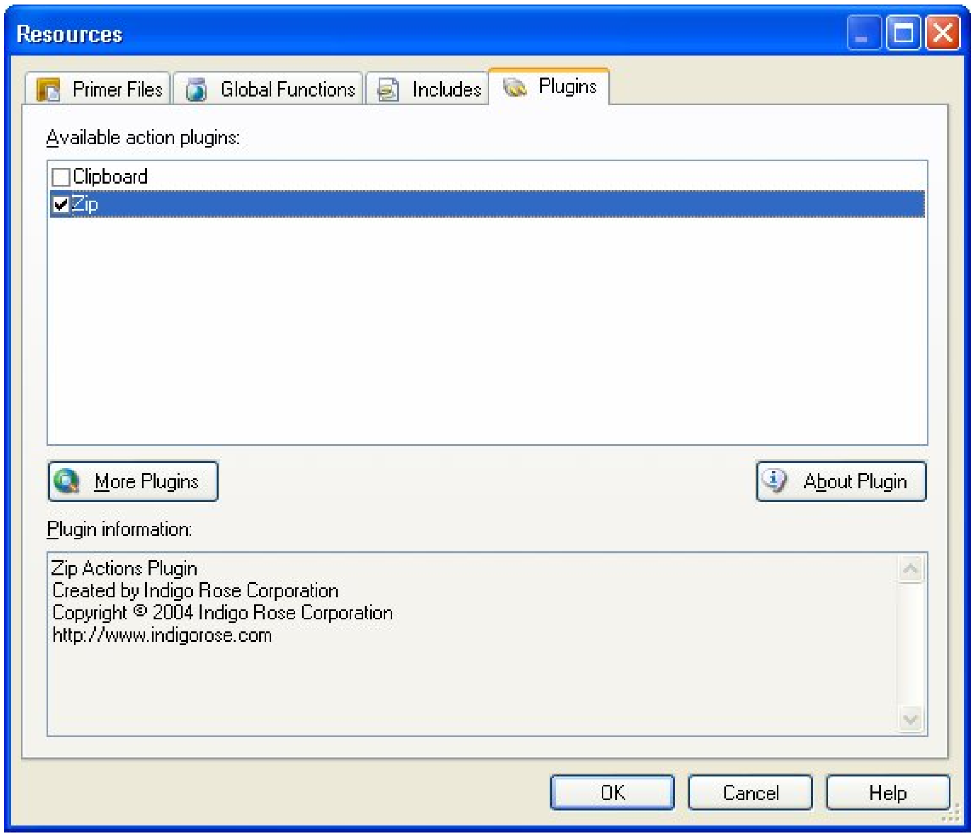
Note: Since the global function section is normally used to create global functions and initialize global variables, the script that is contained there is sometimes referred to as global script.

Plugins

Plugins are actions that are external to the Visual Patch program. They are independently developed and distributed and can be integrated into your projects to extend their functionality. Some plugins may be developed by Indigo Rose, while others may be developed by third parties.

You can refer to the plugin's documentation for information on what features it adds and how to use them (click the About Plugin button to bring up the About Plugin dialog and then click the Plugin Help button).

Plugins can be added or removed from your project on the Plugins tab, which can be accessed by choosing Project > Resources and clicking on the Plugins tab in the Visual Patch design environment.



☐ Clipboard
☑ Zip

Zip Actions Plugin
Created by Indigo Rose Corporation
Copyright © 2004 Indigo Rose Corporation
http://www.indigorose.com

Project resources: enabling the zip plugin

Note: Any action plugins that are located in the Includes\Plugins folder within the Visual Patch program directory will be available on the Plugins tab.

The More Plugins button is an easy way to see what plugins are currently available on the Indigo Rose website. Pressing this button will bring you to the Visual Patch plugins area of the Indigo Rose website.

If a plugin has a check in the checkbox beside its name on the plugins tab, it will be included in your patch. Since there is some overhead in terms of file size, it is recommended that you only include plugins that are needed by your patch. If you do not check the checkbox beside a plugin, it will not be included in the patch and will not take up any extra space.

Once you have included the plugin in your project all of its actions will be available to you in the action editor and action wizard.

Note: Help for plugin actions can be accessed on the action editor in the same way that you would access help for built-in actions.

Script Files

Script files are external text files that contain script usually with the .lua file extension. They can be added to your project on the Includes tab of the Resources dialog, which you can access by choosing Project > Resources from the program menu.

Note: All action editors in Visual Patch have the ability to export their script via the Advanced button. Select Advanced > Save to save the script as an external file.

Script files are very similar to the global functions section of Visual Patch except that instead of the script being kept in the Global Functions resource, it is stored externally to your project in a text file.

Script files are very useful if you share important and complex code between a variety of different projects.

Here is a quick comparison of the differences between sharing script between projects using the Global Functions resource versus script files.

If you have some script that you want to share between many projects using the Global Functions resource, you have to copy and paste the script into each patch that needs it. Now each project has an exact copy of the script.

The method works fine until you discover a bug or want to change some of the code. Since each project contains a copy of the script, you will have to edit the script in each project in order to be sure that it is using the correct code.

If you were to use an external script file, you need to develop a working piece of script and then get that script into a text file. Then, include the script file in each project that needs it. Using this script file method, each project does not contain a *copy* of the script; rather, each project *references* the *same* piece of script.

If you find an error or want to change any of the script, you do not have to edit the script in each project; you simply have to edit the script in the text file. Since each project references the same script file, you know that the next time you build a project, it will be using the correct script.

Essentially, with the Global Functions resource, you have to maintain the script for each project that uses it. On the other hand, the script file method allows you to maintain your script in a single location, the script file. For this reason, it’s a good idea to use script files when you want to share global script between several projects.

Chapter 7:

Session Variables

When designing a patch, it’s often desirable to make parts of it dynamic. For example, the user might input a value on one screen that you’d like to display on the next. Or you might want to display a path on a screen (as the default value in an edit field, perhaps), but the path includes a folder like “My Documents” that is likely to have a different location on each user’s system.

Although you could use regular script variables along with actions to manipulate the screen text at run time, session variables allow you to accomplish the same result in a more direct way: by simply including “placeholders” in your screen text that will automatically be replaced by specific values before the screen is shown.

In this chapter you will learn everything there is to know about session variables in Visual Patch.

7

In This Chapter

In this chapter, you'll learn about:

- Built-in and custom session variables
- Setting session variables
- Removing session variables
- Using session variables on screens
- Expanding session variables in scripts

What Are Session Variables?

Session variables are designed to handle dynamic data during the patching process. Essentially "placeholders" for changeable text, session variables give you an easy way to insert dynamic values into the text that appears on your screens.

They also give you an easy way to compose paths to locations that cannot be known in advance, such as the path to the user's My Documents folder. For example, you can use a session variable like %MyDocumentsFolder% in a path to be replaced by the appropriate full path at run time.

Like regular variables, session variables allow you to "store" information in them, acting like named "containers" that you can assign values to. The main difference between session variables and the "regular" variables you use in scripts is simply that session variables in a screen's text are automatically expanded before the screen is shown. This makes them especially useful for displaying dynamic text on screens.

Even though all session variables are functionally identical, there are two distinct categories of session variables that can be used in Visual Patch: built-in session variables, and custom session variables.

Built-in Session Variables

For convenience, Visual Patch contains a variety of built-in session variables for values that are commonly used in projects. These variables are automatically assigned appropriate values when the patch application is started.

Most of the built-in session variables hold information that has been gathered from the user's system. For example, since the path to the Windows folder can differ between systems, a session variable named %WindowsFolder% is provided which automatically contains the correct path.

Note: Many of these values are also available in the form of global variables that you can use directly in your scripts, e.g. _WindowsFolder and _ProgramFilesFolder. There are also actions like Shell.GetFolder that you can use to get additional system paths. The built-in session variables are provided primarily for use in paths and default values that are displayed on screens.

Here is the list of built-in session variables, in alphabetical order:

%AppFolder%

The main directory where your software has been found on the user’s system. The value of this session variable is set through actions. (A default action script to determine this folder path is automatically generated for you by the project wizard.)

%ApplicationDataFolder%

The path to the Application Data folder on the user’s system. This folder serves as a common repository for application-specific data. Typically, this path is something like “C:\Documents and Settings\YourName\Application Data.”

%ApplicationDataFolderCommon%

The path to the all-user Application Data folder on the user’s system. This folder serves as a common repository for application-specific data. Typically, this path is something like “C:\Documents and Settings\All Users\Application Data.”

%CommonFilesFolder%

The user’s Common Files folder. Typically, this is something like “C:\Program Files\Common Files.”

%CompanyName%

Your company’s name. The value of this variable is set on the Session Variables tab of the Project Settings dialog.

%Copyright%

The copyright message for your product. The value of this variable is set on the Session Variables tab of the Project Settings dialog.

%DesktopFolder%

The path to the user’s Desktop folder. On Windows NT/2000/XP/Vista, this is the path from the per-user profile.

%DesktopFolderCommon%

The path to the user’s Desktop folder. On Windows NT/2000/XP/Vista, this is the path from the All Users profile. On a non-Windows NT system, this is the same as %DesktopFolder%.

%FontsFolder%

The path to the user’s font directory (e.g. “C:\Windows\Fonts”).

%MyDocumentsFolder%

The user’s personal (“My Documents”) folder on their system. Usually this is something like “C:\Documents and Settings\YourName\My Documents” on Windows 2000/XP and “C:\My Documents” on Windows 98/ME and "C:\Users\YourName\Documents" on Windows Vista.

%ProductName%

The name of the product that you are patching. The value of this variable is set on the Session Variables tab of the Project Settings dialog.

%ProgramFilesFolder%

The user’s Program Files folder (e.g. “C:\Program Files”).

%RegOwner%

The name of the registered user of the system.

%RegOrganization%

The organization of the registered user of the system.

%SourceDrive%

The drive that the patch executable was run from (e.g. “C:” or “D:”).

%SourceFolder%

The full path to the folder that the patch executable was run from (e.g. “C:\Downloads” or “D:\”).

%SourceFilename%

The full path, including the filename, for the current patch executable. For example, if the user was running “patch.exe” from “C:\Downloads,” %SourceFilename% would be expanded to “C:\Downloads\patch.exe.”

%StartFolder%

The path to the user’s Start menu folder. On Windows NT/2000/XP/Vista, this is the path from the per-user profile.

%StartFolderCommon%

The path to the user’s Start menu folder. On Windows NT/2000/XP/Vista, this is the path from the All Users profile. On a non-Windows NT system, this is the same as %StartFolder%.

%StartProgramsFolder%

The path to the Programs folder in the user’s Start menu. On Windows NT/2000/XP/Vista, this is the path from the per-user profile.

%StartProgramsFolderCommon%

The path to the Programs folder in the user’s Start menu. On Windows NT/2000/XP/Vista, this is the path from the All Users profile. On a non-Windows NT system, this is the same as %StartProgramsFolder%.

%StartupFolder%

The path to the user’s Startup folder. On Windows NT/2000/XP/Vista, this is the path from the per-user profile.

%StartupFolderCommon%

The path to the user’s Startup folder. On Windows NT/2000/XP/Vista, this is the path from the All Users profile. On a non-Windows NT system, this is the same as %StartupFolder%.

%SystemFolder%

The path to the user’s Windows System folder (e.g. “C:\Windows\System”).

%SystemDrive%

The drive that the user’s Windows System directory is located on (usually “C:”).

%TempFolder%

The path to the user’s Temp folder.

%TempLaunchFolder%

The path to the temporary directory where Visual Patch extracts the files it will need for the patch. (For example, this is the directory where primer files are extracted.)

Usually this will be the user’s temporary directory, unless overridden with the /T command line option.

%WindowsFolder%

The path to the user’s Windows folder (e.g. “C:\Windows”).

%WindowTitle%

The text that will appear on the windows task bar while the patch is running. The value of this variable can be changed on the Session Variables tab of the Project Settings dialog.

Custom Session Variables

You can define your own session variables to supplement the built-in session variables that are automatically provided in Visual Patch. The session variables that you define are known as “custom” session variables.

Custom session variables can be used everywhere that built-in session variables can be used; in fact, they are functionally identical. The only difference is that the built-in session variables are automatically created for you in each project, whereas custom session variables don’t exist until you assign a value to them.

Setting Session Variables

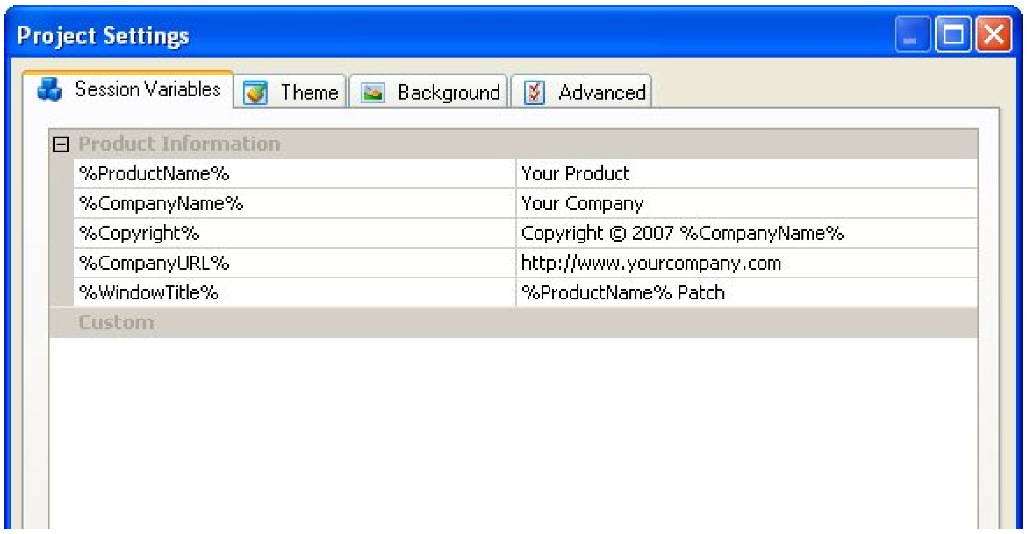
Each session variable consists of a name, e.g. “%ProductName%,” and a value that you assign to it, e.g. “Widget Master 2.0.” When a session variable is expanded at run time, its name is replaced by the value that is currently assigned to it. (For example, “Thank you for using %ProductName%” would become “Thank you for using Widget Master 2.0.”)

There are two ways you can assign a value to a session variable: you can set its initial value on the Session Variables tab, or you can use an action to set its value anywhere in your project.

Using the Session Variables Tab

The Session Variables tab provides a convenient location for setting the initial value of session variables at startup. It is primarily used for values that need to be displayed on screens and that don’t need to be determined dynamically at run time using actions. In other words, it is where you can specify values that you know in advance and that you want to display on the earliest screens in your project.

The Session Variables tab is located on the Project Settings dialog, which you can access by choosing Project > Settings from the program menu.



Product Information	
%ProductName%	Your Product
%CompanyName%	Your Company
%Copyright%	Copyright © 2007 %CompanyName%
%CompanyURL%	http://www.yourcompany.com
%WindowTitle%	%ProductName% Patch
Custom	

The Session Variables tab

The Session Variables tab contains two categories: Product Information and Custom.

The Product Information category contains built-in session variables for values that are commonly displayed on screens, such as the product name and the name of the company that produced it. To set the value of one of these variables, simply edit the appropriate field in the right-hand column.

The Custom category is where you can add, remove and edit your own session variables to supplement the ones in the Product Information category. To add a custom session variable, click on the Add button at the bottom of the dialog.

Tip: As the session variable list grows, it may help to hide portions of the list. Each category can be expanded or collapsed by clicking the “+” icon on the left hand side of the category text.

Using Actions

An action is also available to set the value of a session variable. This action is called SessionVar.Set. It allows you to set the value of an existing session variable that was defined on the Session Variables tab, or to create a brand new one.

The SessionVar.Set action can be used with any event (i.e. in any script) throughout the project. The function prototype for this action is:

```
SessionVar.Set(string VariableName, string Value)
```

For example, if you want to assign the value “My Value” to a session variable named %MyVar%, the action would look like this:

```
SessionVar.Set("%MyVar%", "My Value");
```

After the above action is performed, all occurrences of the text %MyVar% on future screens will be replaced with the text “My Value.”

Note: The SessionVar.Set action works with *all* session variables, including built-in session variables like %MyDocumentsFolder%. It is possible to overwrite a built-in session variable’s value using the SessionVar.Set action, so be very careful when setting session variables with actions. Under normal circumstances, there should be no reason to modify the values of built-in variables.

Removing Session Variables

When you remove a session variable from your project, Visual Patch will no longer recognize the variable's name as special placeholder text. For example, removing the session variable %ProductName% causes the name to revert back to its actual characters. In other words, the text "%ProductName%" ceases to be anything other than the letters %, P, r, o, d, u, c…and so on. After the session variable is "removed," there is no longer a value associated with the name, and no expansion occurs.

There are two methods for removing session variables from your project: using the Session Variables tab, or using actions.

Using the Session Variables Tab

Similar to adding session variables, removal of session variables can also be accomplished from the Session Variables tab. However, only those within the Custom category can be removed from your project. To remove a custom session variable, click on the desired session variable name in the list to highlight it, and then click on the Remove button. The session variable will be removed from the list.

Using Actions

Session variables can also be removed at any point during the patching process with an action. The action used to remove a session variable is called SessionVar.Remove. The function prototype for this action can be seen below:

```
SessionVar.Remove(string VariableName)
```

For example, if you wanted to remove a session variable called %MyVar%, the action script would look like the following:

```
SessionVar.Remove("%MyVar%");
```

Note: Since both custom and built-in session variables behave the same, it is possible to remove a built-in session variable using the SessionVar.Remove action. For this reason, extra care should be taken when removing session variables with actions.

Using Session Variables on Screens

The main use of session variables is for the dynamic expansion of text strings on screens. One example of a valuable use of session variables is when you need to use a value on multiple screens, such as a product version number. While you can certainly enter the text directly for each screen, if that string changes in the future, it would require finding every location where it is used in order to change its value. Using a session variable in place of that text would only require the modification of the session variable's value in one location.

Another valuable use of session variables is for gathering data on one screen that needs to be displayed on another screen. In this case, the values are not known until some point during the patching process, and therefore could not be directly entered at design time.

Tip: Session variables can also be useful in multilingual patches for custom messages that you wish to display depending on the language detected or chosen.

When Are Session Variables Expanded?

Session variables are automatically expanded before each screen is shown—specifically, before each screen's On Preload event. Any session variable that is used on a screen will automatically display the value it contained *before* that screen was shown.

This means that if you change the value of a session variable in a screen's On Preload event, in most cases the old value will still appear. There are a few exceptions, such as some static text controls which will automatically be "refreshed" after the On Preload event and will therefore display their new values. As a general rule, however, the On Preload event is already "too late" for any changes to a session variable to be made if you want the new value to automatically appear on the screen.

One way to get the current value from a session variable is to expand it "manually" using the SessionVar.Expand action. SessionVar.Expand allows you to retrieve the current value of a session variable at any point in your project. In fact, you can use SessionVar.Expand on a screen's On Preload event to retrieve a session variable's value, and then use actions to replace the screen text with new text that includes the current value.

Expanding Session Variables in Scripts

Session variables are often used on screens that gather information from the user. For example, the Edit Fields screen stores the user's input in separate session variables—one session variable for each edit field on the screen. This is fine if you simply want to display the user's input on another screen; in that case, all you need to do is include the appropriate session variables in that other screen's text. If you want to use the inputted values in your scripts, however, you need a way to expand the session variables in your script. This can easily be accomplished using the SessionVar.Expand action.

The function prototype for this action is:

```
string SessionVar.Expand(string Text);
```

Basically, the SessionVar.Expand action takes a string of text and gives you back the same text, but with all of the session variables in the string expanded. In other words, it returns a copy of the text in which all of the session variables have been replaced by their current values.

For example, if a session variable called %MyName% contains the string "They call me nobody," you can access the string using the following action script:

```
strContents = SessionVar.Expand("%MyName%");
```

In the above line of script, the variable strContents would receive an expanded version of "%MyName%." The end result is that the value stored in %MyName% ("They call me nobody") would be assigned to strContents.

However, SessionVar.Expand isn't limited to retrieving the contents of a single session variable. It will happily expand a string containing several session variables—or even one containing no session variables at all. (In the latter case, the string it returns will be an exact copy of the original string.)

For example, using SessionVar.Expand on the string "When asked for his name, all he said was: %MyName%" would return the entire string with the expanded contents of the session variable %MyName%:

When asked for his name, all he said was: They call me Nobody

In addition, when the SessionVar.Expand action expands a string, it performs a *recursive* expansion. Session variables within session variables are expanded as well. This means that if the value in a session variable is a string that has a session variable in it, the “internal” session variable will also be expanded. You can think of the expansion as being a “loop” that continues until there are no more session variables left to expand.

For example, consider the following two session variables:

%verb% - whose value is “flying”
%message% - whose value is “Look at me, I’m %verb%!”

After the following action script is executed:

```
strContents = SessionVar.Expand("%message%");
```

…the contents of the variable strContents would be:

Look at me, I’m flying!

As you can see, %message% was replaced by “Look at me, I’m %verb%!” and then %verb% was replaced by “flying.”

Expanding Without Recursion

Visual Patch also contains an action that will prevent the recursive expansion of session variables. This action is called SessionVar.Get, and its function prototype is:

```
string SessionVar.Get(string Text);
```

Using the previous example, let’s say we only wanted to expand the contents of %message%, without expanding %verb%. In that case, the following action script could be used:

```
strContents = SessionVar.Get("%message%");
```

…and the contents of the variable strContents would be:

Look at me, I’m %verb%!

As you can see, the SessionVar.Get action would expand the %message% session variable, but wouldn’t go any further; the %verb% variable would remain unexpanded.

Expanding after On Preload

Session variables are automatically expanded before each screen is shown. This automatic expansion happens before any of the screen's events are triggered—even before the earliest screen event, On Preload. This means that if you use the SessionVar.Set action to change the value of a session variable from On Preload, the new value will not appear on the screen, because at that point the session variables have already been expanded. (There are exceptions to this rule, such as the static text controls on some screens, but it is safer to assume that it is true in all cases.)

In order to change the text on a screen from within that screen's events, you must use a different method. Luckily, there are two ways in which you can expand session variables on the current screen.

The first and most straightforward method is to formulate a new text string with the session variable in it, expand that string using SessionVar.Expand, and then assign the expanded string to the desired screen control.

For example, the following script would expand a string that contains two session variables and then replace the text on a scrolling text screen with the new text:

```
NewText = SessionVar.Expand("First: %FirstName%\r\nLast: %LastName%");
DlgScrollingText.SetProperties(CTRL_SCROLLTEXT_BODY, {Text=NewText});
```

Although this method works well, it requires you to write and edit the text within your script, instead of composing the text directly on the Settings tab for that screen. This isn't too difficult if your project only supports one language. However, if you're creating a multilingual project, you'll need to use control structures to assign different text to the screen that is appropriate for the user's system language. This would make your scripts much longer and more difficult to maintain.

An alternative method is to use an action to retrieve the original text for the screen, which still contains the session variables in their unexpanded form. This allows your script to essentially "re-expand" the text that you entered on the Settings tab.

Here is a version of the previous script that uses the Screen.GetLocalizedString action to retrieve the original unexpanded text for the scrolling text control:

```
OriginalText = Screen.GetLocalizedString(IDS_CTRL_SCROLLTEXT_BODY);
NewText = SessionVar.Expand(OriginalText);
DlgScrollingText.SetProperties(CTRL_SCROLLTEXT_BODY, {Text=NewText});
```

The key to this second approach is the Screen.GetLocalizedString action, which retrieves the text for a specific screen control as it was entered on the Settings tab, automatically choosing the appropriate text for the language on the user's system.

When your project supports multiple languages, it is much easier to edit the screen text directly on the Settings tab, where you can use the language selector to switch between all of the languages that your project supports.

Tip: If you would like to see an advanced example of session variables in use, examine some of Visual Patch’s built-in screens. For example, the Select Drive screen uses actions to set, update, and display the session variable %SpaceAvailable%.

Chapter 8:

Languages

The Internet has opened many new markets to software developers whose past products would usually have supported only their own local language. In the international marketplace, it is important not only to offer software in a variety of languages, but also to keep this software current.

As you’ll see in this chapter, you can use Visual Patch to create a patch that will automatically display messages and prompts in your user’s native language. With integrated language selection built into all screen dialogs, Visual Patch also makes it very easy to modify your existing translations at any time.

8

In This Chapter

In this chapter, you'll learn about:

- Internationalizing your patch
- How run-time language detection works
- The language manager
- Language files
- Setting the default language
- Adding and removing languages
- The language selector
- Localizing screens and actions
- Customizing error messages and prompts
- Advanced techniques, such as using actions to determine the current language and "changing" the current language for testing purposes

Internationalizing Your Patch

Visual Patch has the ability to automatically detect the user’s system’s language and to display messages and screens in that language. As the developer, you have full control over which languages are supported in your project and over the content that you would like presented to the user.

Language text is mainly used for messages generated throughout the patching process and on screens, both of which can be easily translated for multilingual patches.

Visual Patch allows you to localize your patch application in two areas:

- Common error messages, status messages and prompts
- Application-specific screens

The following sections of this chapter will look more closely at how to achieve this localization.

Run-time Language Detection

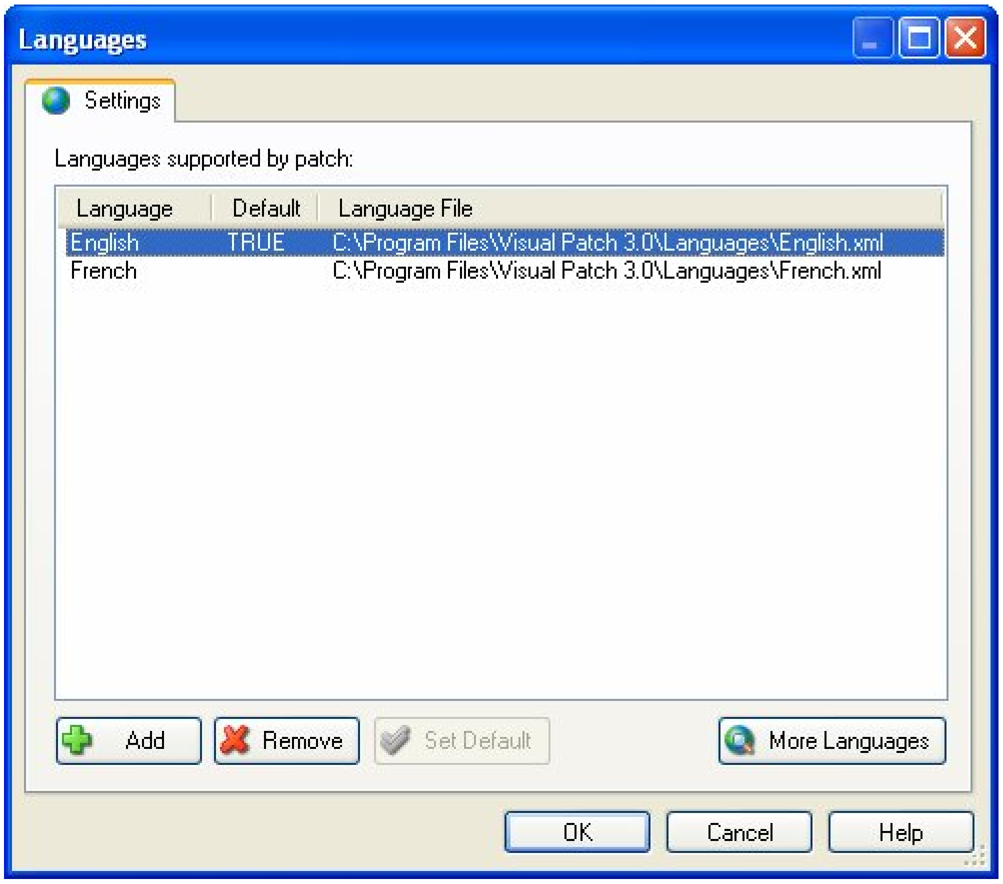
The language that the patch application detects is based on the user’s regional and language settings. These settings allow Windows users to configure which language is displayed, which input locale is used and which keyboard layout is supported in the Windows operating system environment. The settings are configured when Windows is installed and can usually be changed from the Windows Control Panel. For example, in Windows XP, a user can select Start > Settings > Control Panel and then launch the Regional and Language Settings control panel application.

Each language in Windows has a constant language identifier. A language identifier is a standard international numeric abbreviation for the language that is used in a country or geographical region. Each language identifier is made up of a primary and secondary language ID. (There is a complete list of primary and secondary language IDs in the Visual Patch help file.)

Visual Patch maps all known languages and sub-languages according to the language identifiers used by Windows. These mappings exist in a file called language_map.xml located in Visual Patch’s Language folder (usually “C:\Program Files\Visual Patch 3.0\Languages”). You can look at this file to see which primary and secondary language IDs are mapped to which languages. (It is *not* recommended that you modify this file unless you have a very specific reason to do so.)

The Language Manager

The languages that are supported by your patch are all configured from the Language Manager. You can access the language manager by selecting Project > Languages from the menu.



Languages supported by patch:

Language	Default	Language File
English	TRUE	C:\Program Files\Visual Patch 3.0\Languages\English.xml
French		C:\Program Files\Visual Patch 3.0\Languages\French.xml

The Language Manager

Although you can localize messages in several areas of the design environment, the Language Manager is the only place where you can control the project's default language as well as which languages are supported by your patch. For example, if you are editing one of your screens and decide that you would like to add a German translation, you will have to use the Language Manager to do so. Once you add German support here, it will be available in all other areas of your project.

When you add a language to your project, you are indicating that you want the patch application to recognize that particular language identifier on a system and to use specific messages for that language when identified. Conversely, if a system's language is not represented in the languages list, it does not mean that the patch will not run on that system; rather, it means that the patch will use the default language.

Default Language

Every project must have a default language. The default language is the one that will be used when the patch application encounters a system language that is not represented in the languages list.

For example, let's suppose that you have English, French and German support in your project with English as the default language. If your user runs the patch on a Greek system, the user will see the English messages since you did not specifically include support for the Greek language.

Note that the default language must be one that has a corresponding language file (see the next section).

Language Files

A language file is an XML file that contains all of the "internal" error messages, status messages and prompts that are used by the patch application. Language files do not contain project-specific messages, such as the text that you enter on screens.

The language files are located in Visual Patch's Languages folder (usually "C:\Program Files\Visual Patch 3.0\Languages"). They are named according to the English name for the language they represent. Each file contains a language map that identifies which language the file is responsible for and all of the built-in messages that will be used for that language.

Not all languages have a pre-configured language file. If you add a language to your project that does not have a language file, that language will use the same messages as the default language.

Getting Additional Language Files

If you need a language file that is not shipped with Visual Patch, please visit the Indigo Rose website (www.indigorose.com) and user forums (www.indigorose.com/forums/) where new language files are made available from time to time.

Making Your Own Language File

If after consulting the Indigo Rose web site you still can't find the language file you need, you can always make one yourself. To make a new language file, simply make a copy of the existing language file that you want to translate from, rename it to the new language name, change the language map in the file accordingly and then translate the messages.

To clarify this process, here is an example of how you would create a French language file like the one that ships with Visual Patch:

1. Open Windows Explorer to Visual Patch's Languages folder (usually "C:\Program Files\Visual Patch 3.0\Languages".)

2. Make a copy of English.xml and name it French.xml.

3. Open French.xml in a text editor such as Notepad.

4. Open language_map.xml from the Languages folder in a text editor as well.

5. Locate the section that maps French in the language_map.xml file. It should look like this:

```
<Language>
  <Name>French</Name>
  <Primary>12</Primary>
  <Secondary>
        <ID>1</ID>
        <ID>2</ID>
        <ID>3</ID>
        <ID>4</ID>
        <ID>5</ID>
        <ID>6</ID>
  </Secondary>
</Language>
```

6. Copy the above section from language_map.xml and paste it in place of the <Language></Language> section of the French.xml file. This will allow Visual Patch to recognize this file as a language file for the French language.

7. Translate all messages in the <Messages></Messages> section to French. Do not change any actual XML tags. For example:

```
<MSG_SUCCESS>Success</MSG_SUCCESS>
```

becomes:

```
<MSG_SUCCESS>Succès</MSG_SUCCESS>
```

8. Save the file and re-open Visual Patch. The new language will now be available.

9. Share the file with others! Go to http://support.indigorose.com and open a support ticket saying that you have made a language that you would like to share with other Visual Patch users. We will take it from there.

Adding Languages

To add a new language to your patch, click the Add button in the Language Manager. This will open the Add New Language dialog. Simply select the language that you want to add and click OK. You will then see the language appear in the languages list.

What Happens When You Add a New Language

When you add a new language to your project, the following happens automatically:

- Visual Patch searches the Languages folder for an appropriate language file for the language. If one is not found, the new language is set to use the default language's language file.

- The newly added language is added to all screens. That is, all screens have messages added to them for the new language. If a translated language file for that particular screen already exists, it will be used. Otherwise, the messages from the default language will be replicated for the new language.

- The language is added to the list of languages that you can select from in the language selectors throughout the design environment.

Of course, you will still need to go into the Screens dialogs to verify and/or translate the text for the new language.

Removing Languages

To remove a language from the project, select it in the Language Manager's list and click the Remove button.

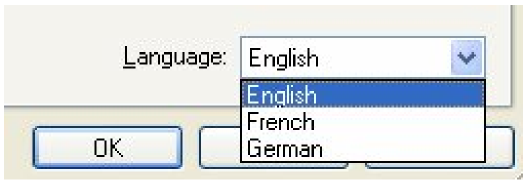
Note that the default language cannot be removed. In order to remove a language that is currently being used as the default language for a project, you will need to make another language the default language first.

When you remove a language, all of the translations for that language are removed from the screens in the project. Therefore, use caution when removing languages.

The Language Selector

Once a language is added to your project, it is available for translation. When you're editing the text on a screen, you can select the language that you want to edit by using the *language selector*. The language selector is simply a drop-down list that lets you choose the current language (for editing purposes) of the screen.

For example, if your project supports English, French and German, the language selector on every screen's properties dialog will let you choose whether to edit the English, French, or German text.



The language selector

Selecting a language in the language selector replaces the editable text on the dialog with the text for that language. Any changes that you make to the screen text will be restricted to that language.

Note: If you select a newly added language for which there is no language file, the editable text will initially be the same as the text for the default language.

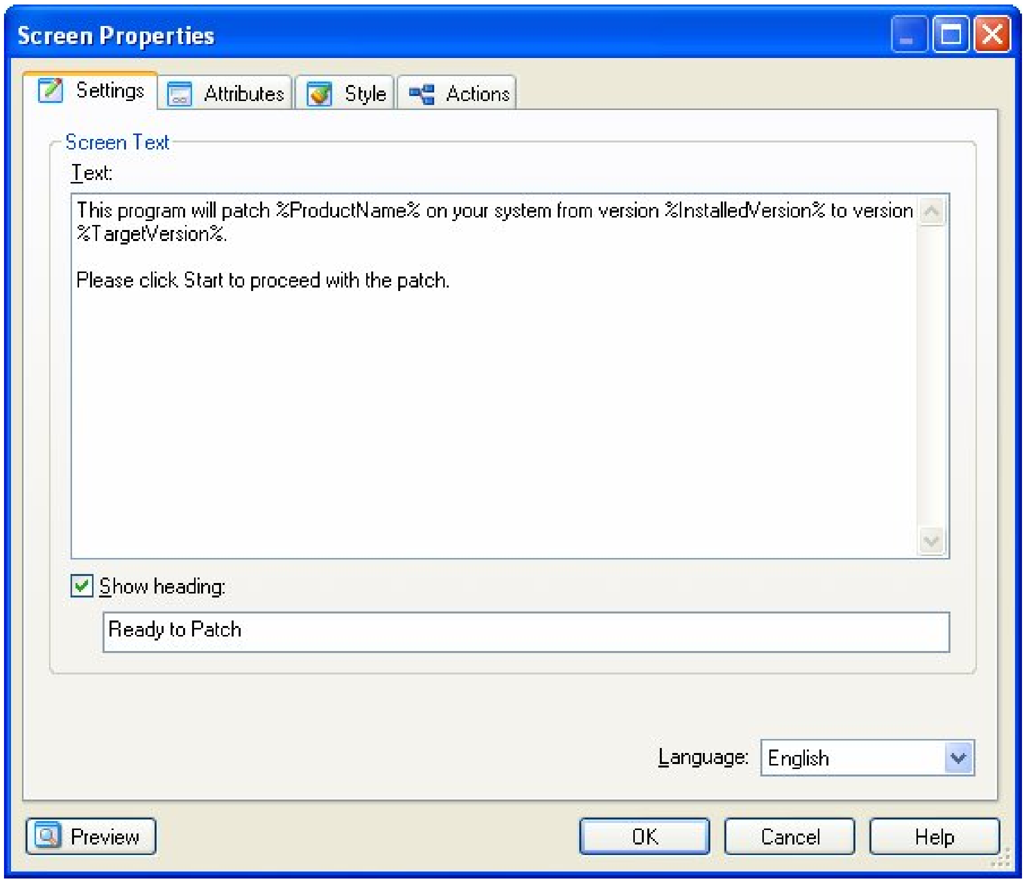
Only the languages that have been added to the project will appear in the language selector. If you want to work on a language that is not in the drop-down list, you will first have to add it to the project using the Language Manager.

Localizing Screens

To localize a screen, open the Screens dialog (Project > Screens) and double-click on a screen to open the screen's properties. Next, select the language that you want to edit in the language selector. Then, simply type the text that you want for the various fields in that language.

Note: The Screen ID field on the Attributes tab cannot be translated. The Screen ID is a unique identifier for the screen and is never displayed to the user.

Here is an example of a screen with English text:



Text:

This program will patch %ProductName% on your system from version %InstalledVersion% to version %TargetVersion%.

Please click Start to proceed with the patch.

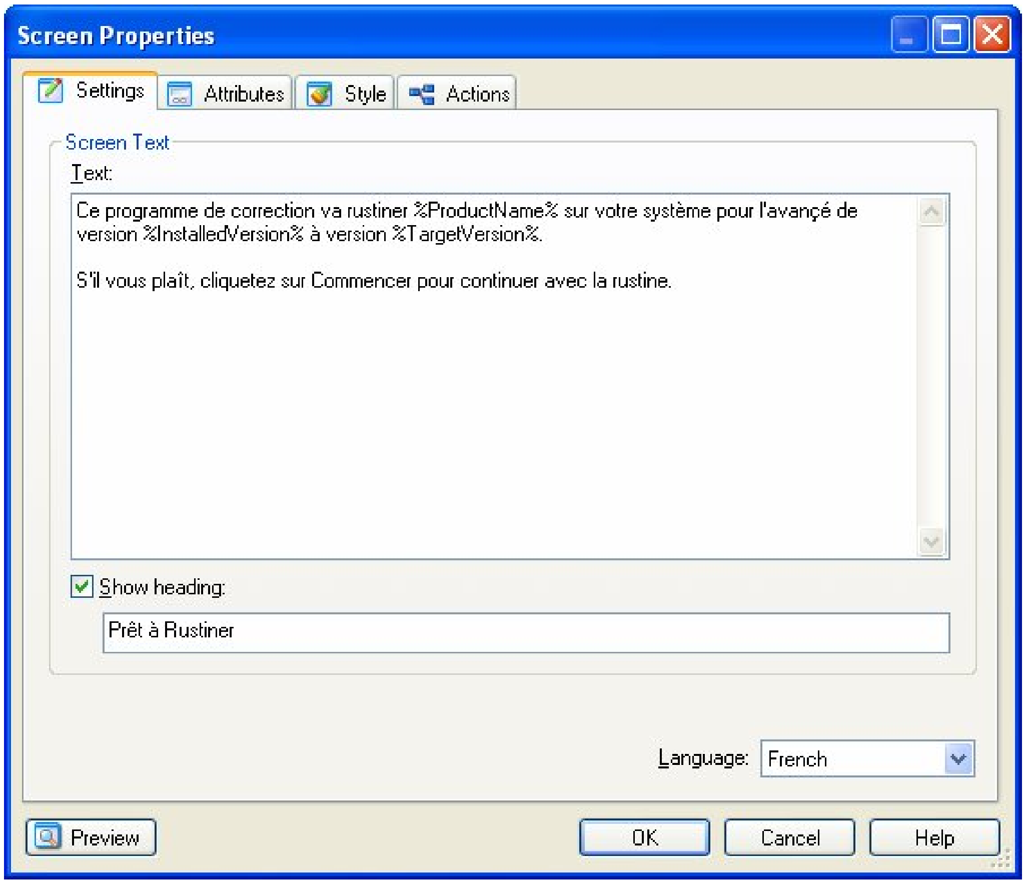
✓ Show heading:

Ready to Patch

Language: English

A "Ready to Patch" screen with English text

And here is the same screen with French text:



Screen Text

Text:

Ce programme de correction va rustiner %ProductName% sur votre système pour l'avançé de version %InstalledVersion% à version %TargetVersion%.

S'il vous plaît, cliquetez sur Commencer pour continuer avec la rustine.

[✓] Show heading:

Prêt à Rustiner

Language: French

A "Ready to Patch" screen with French text

Notice that in the second screenshot, "French" has been selected in the language selector near the bottom of the dialog. The text that you enter always corresponds to the language that is selected in the language selector.

Tip: If the language that you want to translate to doesn't appear in the language selector, you need to add support for that language to your project. This is done by adding the language in the language manager. For more information, see *The Language Manager* beginning on page 183 and *Adding Languages* on page 186.

Importing and Exporting Screen Translations

There may be times when you want to have your screens translated by a third party translator. If the translator owns a copy of Visual Patch, you can simply send them your project file, have them translate the screens using the method explained above, and then have them send the project file back to you.

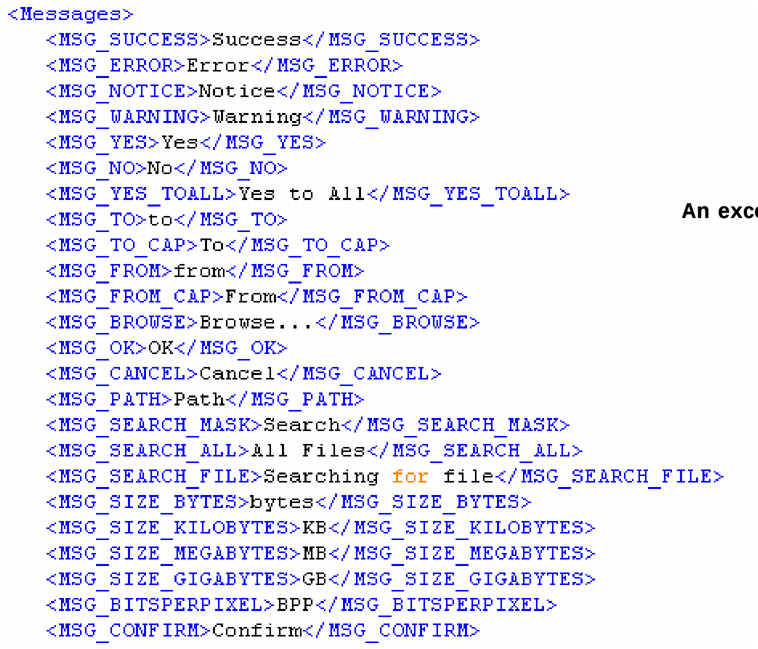
However, it may be that the translator does not own Visual Patch or that you need to work with the project in other ways while the translation is taking place. Visual Patch has a solution for this situation. You can:

1. Open the Screens dialog (Project > Screens).
2. Select the screen that you want to have translated in the Screens list.
3. Click the Advanced button (this will open a popup menu).
4. In the popup menu, select Export Language followed by the language that you want to export.
5. Choose a location to save the file to.
6. Send the exported file to your translator.
7. When you receive the file back, select the screen in the Screens list.
8. Click the Advanced button and then choose Import Language from the popup menu.
9. Locate the translated file and click Open.
10. You will now have the new translated strings in your screen.

Customizing Error Messages and Prompts

If you want to change the default error messages, prompts or status messages, you can edit the appropriate XML file in the Languages folder using a text editor such as Notepad. However, this is not generally recommended; the messages that you change will be propagated to all projects that are built after the changes are made.

Note also that if you choose to change default messages, your modified language file may be overwritten by a future update to Visual Patch. To avoid this, you may want to rename any language files that you modify, for example you might rename english.xml to my_english.xml. (In cases where there is more than one language file for a given language, Visual Patch will use the last one that it finds in the Languages folder.)



```
<Messages>
   <MSG_SUCCESS>Success</MSG_SUCCESS>
   <MSG_ERROR>Error</MSG_ERROR>
   <MSG_NOTICE>Notice</MSG_NOTICE>
   <MSG_WARNING>Warning</MSG_WARNING>
   <MSG_YES>Yes</MSG_YES>
   <MSG_NO>No</MSG_NO>
   <MSG_YES_TOALL>Yes to All</MSG_YES_TOALL>
   <MSG_TO>to</MSG_TO>
   <MSG_TO_CAP>To</MSG_TO_CAP>
   <MSG_FROM>from</MSG_FROM>
   <MSG_FROM_CAP>From</MSG_FROM_CAP>
   <MSG_BROWSE>Browse...</MSG_BROWSE>
   <MSG_OK>OK</MSG_OK>
   <MSG_CANCEL>Cancel</MSG_CANCEL>
   <MSG_PATH>Path</MSG_PATH>
   <MSG_SEARCH_MASK>Search</MSG_SEARCH_MASK>
   <MSG_SEARCH_ALL>All Files</MSG_SEARCH_ALL>
   <MSG_SEARCH_FILE>Searching for file</MSG_SEARCH_FILE>
   <MSG_SIZE_BYTES>bytes</MSG_SIZE_BYTES>
   <MSG_SIZE_KILOBYTES>KB</MSG_SIZE_KILOBYTES>
   <MSG_SIZE_MEGABYTES>MB</MSG_SIZE_MEGABYTES>
   <MSG_SIZE_GIGABYTES>GB</MSG_SIZE_GIGABYTES>
   <MSG_BITSPERPIXEL>BPP</MSG_BITSPERPIXEL>
   <MSG_CONFIRM>Confirm</MSG_CONFIRM>
```

An excerpt from English.xml

Advanced Techniques

There are a number of advanced techniques that you can use to manipulate the language and the translated language strings in your patch at run time. Most of these methods are accomplished using actions. This section covers a few of these advanced techniques.

Determining the Current Language

There are two actions that can be used to retrieve information about the user's language ID: System.GetDefaultLangID and Application.GetLanguage. Although both actions return a table of information containing language IDs, it is important to know the difference between the two.

System.GetDefaultLangID

System.GetDefaultLangID is used to get the primary and secondary language ID that the user employs in Windows. This is absolute and cannot be changed with any other actions. For example, if this action returns 10 as the primary ID and 13 as the secondary ID, you will know that the user's Windows system is configured for Spanish (Chile). (There is a complete list of primary and secondary language IDs in the Visual Patch help file.)

The information returned by this action can be used in cases where you want to make specific choices about what to do based on the user's absolute system language. For example, if you have a Web site that is translated into several different languages, you might want to make a series of if…then statements to open the appropriate site:

```
-- Determine the absolute system language
local tblSysLang = System.GetDefaultLangID();

-- Set a default
local strURL = "http://www.yourcompany.com/english");

-- See if we should go to a different site
if (tblSysLang.Primary == 7) then
    strURL = "http://www.yourcompany.com/german");
elseif (tblSysLang.Primary == 10) then
    strURL = "http://www.yourcompany.com/spanish");
elseif(tblSysLang.Primary == 16) then
    strURL = "http://www.yourcompany.com/italian");
end
```

```
-- Go to the Web site
File.OpenURL(strURL);
```

Application.GetLanguage

Application.GetLanguage is used to retrieve the primary and secondary language ID that is actually being used by the patch application. Note that Application.GetLanguage may return a different result than System.GetDefaultLangID if the language being used by the patch differs from the language your user employs in Windows.

For example, let’s say that you have three languages in your project: English (which is the default language), French, and German. Suppose a user from Chile runs the patch on their system. Even though their system uses primary ID 10 and secondary ID 13 (which would be returned by System.GetDefaultLangID), Application.GetLanguage would return a primary ID of 9 and a secondary ID of 1, which is the project’s default language (English). The patch application would be using the default language because Spanish was not added to the project.

The value returned by Application.GetLanguage will always correspond to a language that you added to your project using the Language Manager. It will never identify languages that were not explicitly included in the project.

Changing the Current Language

Normally, if the patch application detects a language on the user’s system that is not supported in the Language Manager, the default language will be used. However, you may prefer to have your update use a different language in such situations. You can accomplish this by using the Application.SetLanguage action.

The Application.SetLanguage action allows you to directly set the primary and secondary language IDs that will be used for the patch. Calling this action changes all subsequent error and status messages as well as the text shown on the screens. It effectively “forces” the patch application to act like it detected that language in the first place.

For example, let’s say that your project supports two languages: English (which is the default language) and Simplified Chinese. Since English is the default language, it will be used whenever the patch is run on anything other than Simplified Chinese.

However, you might prefer that the patch application use Simplified Chinese if the user runs it on a system configured to use Traditional Chinese.

In other words, you want to override the default language rule and force your patch application to use Simplified Chinese whenever Traditional Chinese is detected. You can do so easily by placing the following short script at the beginning of the On Startup event, before any dialogs are shown:

```
-- Determine the absolute system language
local tblSysLang = System.GetDefaultLangID();
if (tblSysLang.Primary == 4) and (tblSysLang.Secondary == 1) then
    -- Traditional Chinese on user's system,
    -- so use Simplified instead
    Application.SetLanguage(4, 2);
end
```

Testing Different Languages

You may also want to use the Application.SetLanguage action for testing purposes. For example, if you are running an English version of Windows, you might want to see how your patch will look on an Italian system. Because your system is running in English, the patch application will always choose English as the language to display when you run it on your system. However, you could force the patch to use Italian by putting the following script at the beginning of your On Startup event:

```
Application.SetLanguage(16, 1);
```

You could even modify your On Startup script to check for a custom command line option, so you could force your patch application to use a different language at any time. This could be useful for testing purposes, or to handle any language detection issues that are discovered after the patch has been distributed.

Tip: The global variable _CommandLineArgs can be used to determine what arguments were passed to the patch executable.

Localizing Actions

You may have noticed that there is no language selector when editing scripts in Visual Patch. Any text you enter directly into a script will remain the same, regardless of what language is detected on the user’s system.

However, it is possible to use actions to detect the current language, and to use multiple “if” statements to specify different text for different languages.

For example, let’s say that your project supports English and French and you want to show a dialog box using actions that will be localized according to those languages.

The following script first determines the language that the patch application is using, and then displays one of two possible greetings:

```
local tblSysLang = Application.GetLanguage();

if (tblSysLang.Primary == 9) then
    Dialog.Message("Welcome",
                   "Welcome to the patch");
end

if (tblSysLang.Primary == 12) then
    Dialog.Message("Bienvenue",
                   "Bienvenue au programme de correction");
end
```

The Application.GetLanguage action returns a table containing the primary and secondary language ID that is being used by the patch application.

Note that this may or may not be the exact language that was detected on the user's system; rather, it is the appropriate language that the patch application has chosen from the list of supported languages in the project.

In other words, it is the user's system language if that language is supported by the patch; otherwise, it is the default language.

Tip: You can get the user's *actual* system language by using the System.GetDefaultLangID action. However, it is usually preferable to use the Application.GetLanguage action so the scripted language behavior will match the "automatic" language behavior of the screens and error messages in the project.

Working with Existing Translated Messages

Visual Patch allows you to get and set translated messages from the language files and screens at run time. This is done using the following actions.

VisualPatch.GetLocalizedString

This action allows you to retrieve the localized text for a general message from the language files at run time. The message will be returned in the language currently being used by the patch application.

For example, the default language files provide confirmation messages that can be used if the user wants to abort the patching process.

Here is a way to show a dialog that asks the user for confirmation in the current language before exiting:

```
local strTitle = VisualPatch.GetLocalizedString("MSG_CONFIRM");
local strPrompt = VisualPatch.GetLocalizedString("MSG_CONFIRM_ABORT");
local nResult = Dialog.Message( strTitle
                              , strPrompt
                              , MB_YESNO
                              , MB_ICONQUESTION
                              , MB_DEFBUTTON2 );

if(nResult == IDYES)then
    Application.Exit();
end
```

VisualPatch.SetLocalizedString

This action allows you to change the value of a localized string. This can be useful if you want to override the default value of an error message from script, so you don't have to permanently change your language file.

For example, let's say that you want to change the message that is displayed if the user tries to cancel the patch. By default it is "The patch is not finished! Do you really want to abort?", but you want to change it to: "Stopping now is not a good idea. Are you sure?"

```
VisualPatch.SetLocalizedString("MSG_CONFIRM_ABORT",
    "Stopping now is not a good idea. Are you sure?");
```

Screen.GetLocalizedString and Screen.SetLocalizedString

These actions are used to get and set the value of a localized string from the current screen's message file. Every editable text item on a screen has a corresponding string ID that is used internally to retrieve the appropriate text for the current language when the screen is displayed.

These actions can be used to create new, temporary localized strings that are only valid on the current screen. In other words, they permit you to define your own localized strings (using custom string IDs) for use on the current screen.

For example, you could create custom localized error messages for use in a screen's event scripts, without having to implement "if…then" branching to choose the appropriate translated text wherever an error message can be displayed.

Note: The Screen.GetLocalizedString and Screen.SetLocalizedString actions work the same way that VisualPatch.GetLocalizedString and VisualPatch.SetLocalizedString do, except that they will only access strings used on the current screen.

Chapter 9:

Building and Distributing.

Once you have created your Visual Patch project and configured all relevant options, the only two steps left are to build your patch and distribute it.

Building your patch is made easy using the built-in Publish Wizard. For more advanced cases, you can customize the build process through the Build Settings dialog, which gives you access to many advanced features such as build configurations and pre/post build steps. Whatever route you choose, Visual Patch makes building your patch a seamless part of the development process.

Once your patch is built, you must distribute it to your users. This chapter will introduce you to both the standard and advanced aspects of the build process, and will get you well on your way to distributing your patch

9

In This Chapter

In this chapter, you'll learn about:

- The build process
- The publish wizard
- Build settings
- Build configurations
- Constants
- Pre- and post-build steps
- Build optimizations
- Build-related preferences
- Testing and distributing your patch

The Build Process

Visual Patch’s ultimate goal is to produce the smallest patch possible. When you choose to build your patch, all of the files in your included versions are first verified and then analyzed. Visual Patch begins analyzing at the latest version in your patch, and compares all previous versions’ files to the ones in the latest version.

Depending on what the previous versions contain, files listed in the most current version are either not included in the final patch, are partially included in the form of diff files, or are included in their entirety. This ensures that the patch is of the smallest size possible while maintaining compatibility with all desired previous versions.

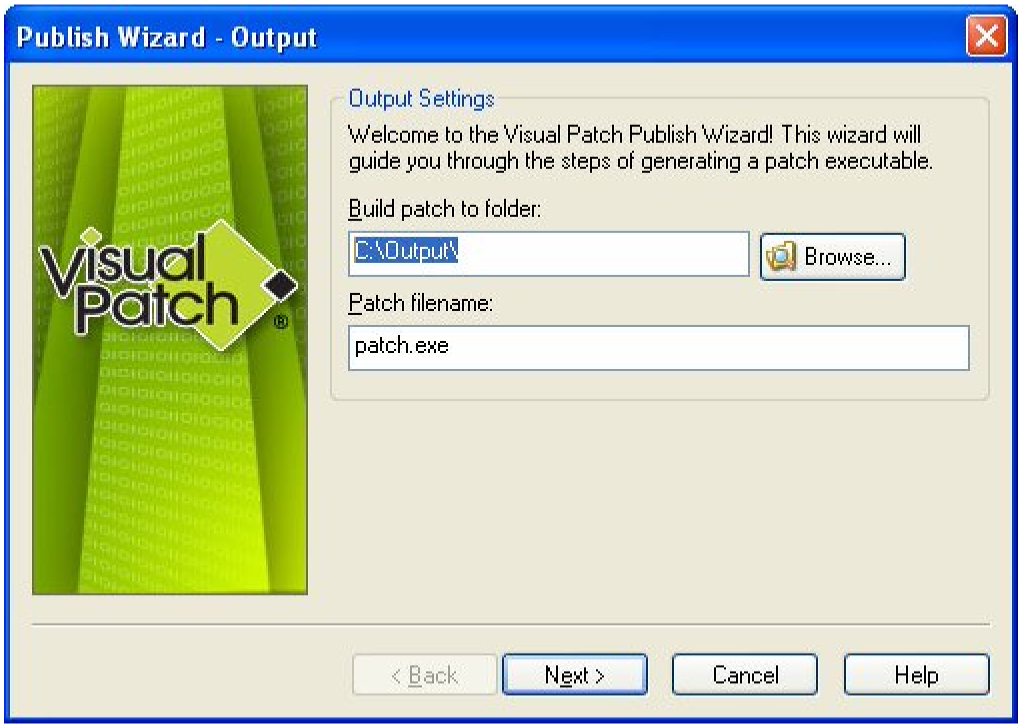
After analyzing all of the included files, diff files are created for any file existing in all versions but which is not the same across all versions. These diff files are created using Indigo Rose’s proprietary binary differencing compression engine. If a file exists in the latest version that does not exist in one or more previous versions, the complete file is included in the patch. Lastly, if a file exists in all versions, and is unchanged across all versions, it is not included in the patch.

During the build process, the files to be included – both diff files and complete files – are compressed into the final single patch executable. It is important to note that the sum of the diff file sizes will never be greater than the individual file’s calculated compressed file size. In other words, if Visual Patch can create a smaller patch by simply compressing an entire file rather than including diff files, it will include the entire file.

The Publish Wizard

With the goal of making the build process as seamless as possible, Visual Patch includes a Publish Wizard to assist you.

The first screen of the publish wizard prompts you for the location that you would like to publish to, as well as the name of the file that you would like to create.



Output Settings

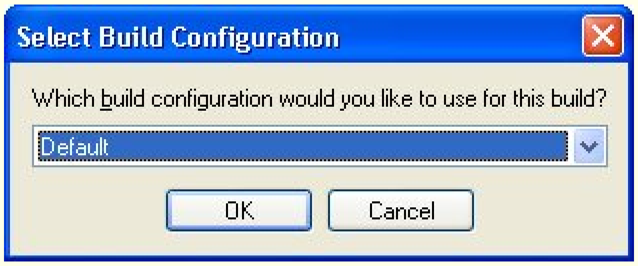
Welcome to the Visual Patch Publish Wizard! This wizard will guide you through the steps of generating a patch executable.

Build patch to folder: C:\Output\

Patch filename: patch.exe

Indicate where the patch executable should be built to

Note: If your project contains more than one build configuration, the wizard will prompt you to choose which build configuration you want to use before it starts:



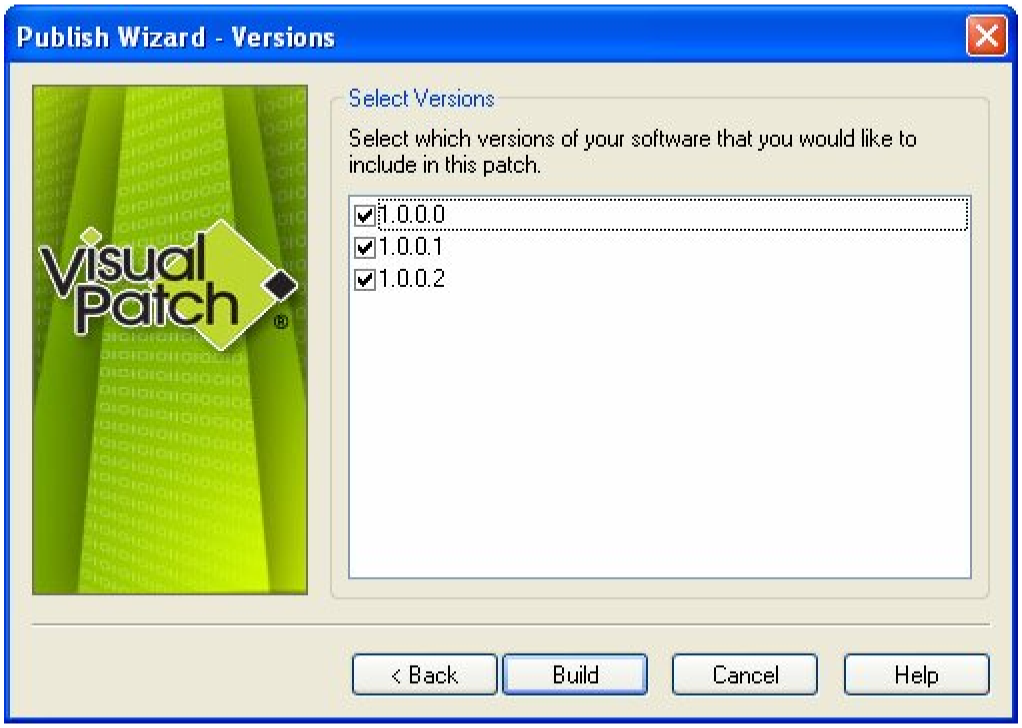
Which build configuration would you like to use for this build?

Default

Selecting a build configuration when the build wizard starts

Build configurations are discussed on page 207.

The second screen of the publish wizard allows you to select which versions will be built into your patch.



Select which versions of your software that you would like to include in this patch.

- [✓] 1.0.0.0
- [✓] 1.0.0.1
- [✓] 1.0.0.2

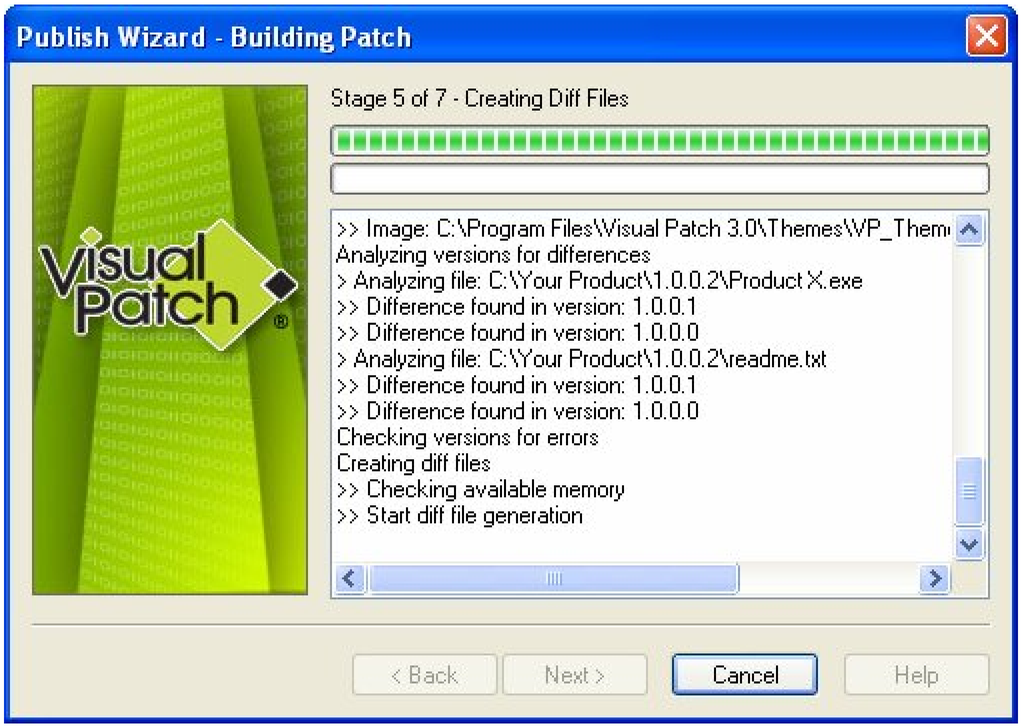
Specify which versions should be included in the patch executable

This is a very important step in the build process. It allows you to individually include or exclude each of the versions in your project from the built patch executable.

Being able to specify which versions are included in your patch is extremely useful. For example, imagine you have a product that has been updated fifteen times. Including all of these versions in your patch could potentially result in a rather large patch. On the other hand, creating a patch that applies only to the last three previous versions would be a substantially smaller file.

The wonderful thing about the Visual Patch build process is you don't have to choose between building a complete full history patch and building a patch applicable to only a few of your past versions. Build one, launch the build wizard again, and easily build the other! Then, let the user decide which patch they should download, or better yet, incorporate TrueUpdate (www.trueupdate.com) into your software and have that decision made automatically, without any user interaction required.

When the Next button is clicked, Visual Patch begins building the patch.

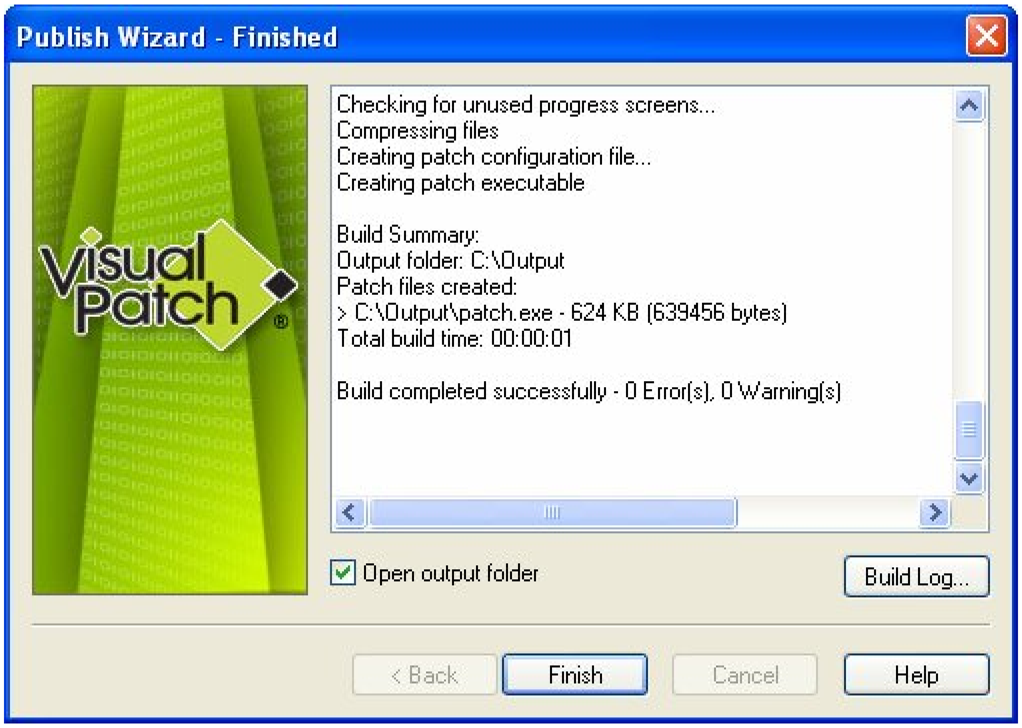


Stage 5 of 7 - Creating Diff Files

```
>> Image: C:\Program Files\Visual Patch 3.0\Themes\VP_Them
Analyzing versions for differences
> Analyzing file: C:\Your Product\1.0.0.2\Product X.exe
>> Difference found in version: 1.0.0.1
>> Difference found in version: 1.0.0.0
> Analyzing file: C:\Your Product\1.0.0.2\readme.txt
>> Difference found in version: 1.0.0.1
>> Difference found in version: 1.0.0.0
Checking versions for errors
Creating diff files
>> Checking available memory
>> Start diff file generation
```

The patch being built

Once the build process is complete, the following summary screen is displayed. This screen presents a summary of the build process and displays any errors.



```
Checking for unused progress screens...
Compressing files
Creating patch configuration file...
Creating patch executable

Build Summary:
Output folder: C:\Output
Patch files created:
> C:\Output\patch.exe - 624 KB (639456 bytes)
Total build time: 00:00:01

Build completed successfully - 0 Error(s), 0 Warning(s)
```

A summary of the build process

Once you are finished viewing this screen, you can click Finish to dismiss the build wizard. If the Open output folder option is checked, the output folder will be opened automatically, giving you instant access to the generated files.

Tip: If you need more information or want to trace an error, you can view the entire build log file by clicking the Build Log button.

Build Settings

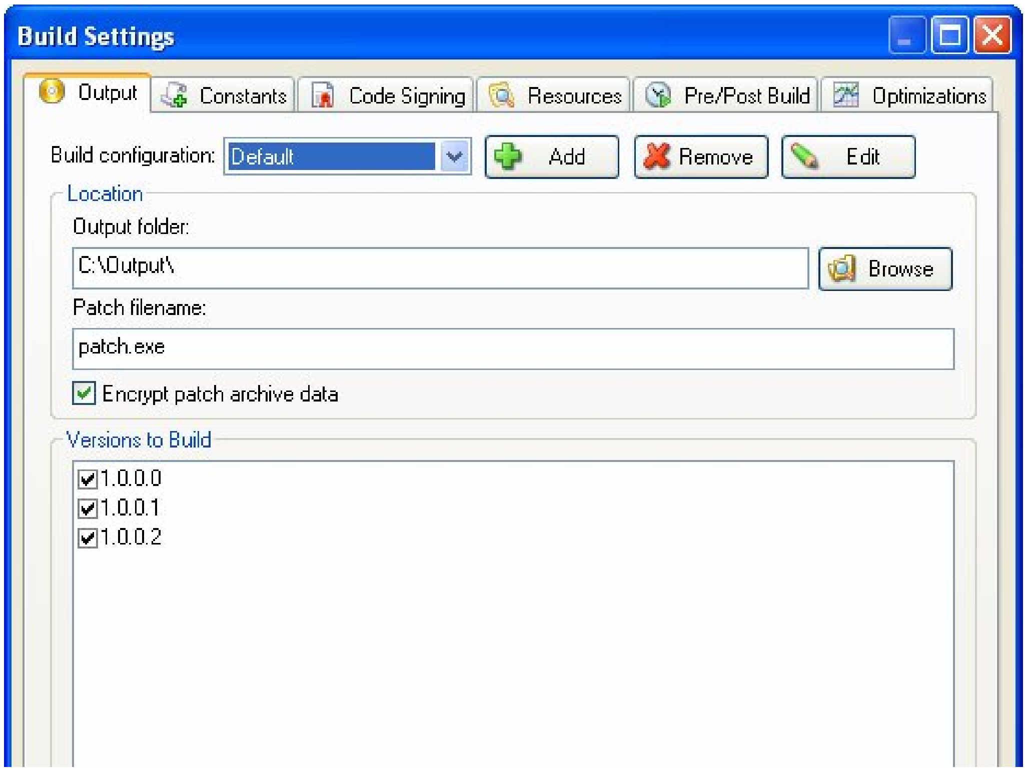
The Build Settings dialog, accessible by choosing Settings from the Publish menu, allows you to set the defaults you would like to use for the build process. Basically, any setting affecting the final built patch file is found here. If you opted to publish without using the publish wizard, this is where you must specify your build settings. These settings are organized into tabbed sections in the Build Settings dialog: Output, Constants, Pre/Post Build, and Optimizations.

Output

All of your patch output settings are located on this tab. You can specify where your final patch executable will be built by adjusting the Location settings.

Enabling the “Encrypt patch archive data” option encrypts your patch data with a random key during the build process, helping to hide and further protect your information.

The Versions to Build list allows you to specify which versions to include in the final patch executable. The executable will only be able to patch the versions you select.



Build configuration: Default

Output folder: C:\Output\

Patch filename: patch.exe

☑ Encrypt patch archive data

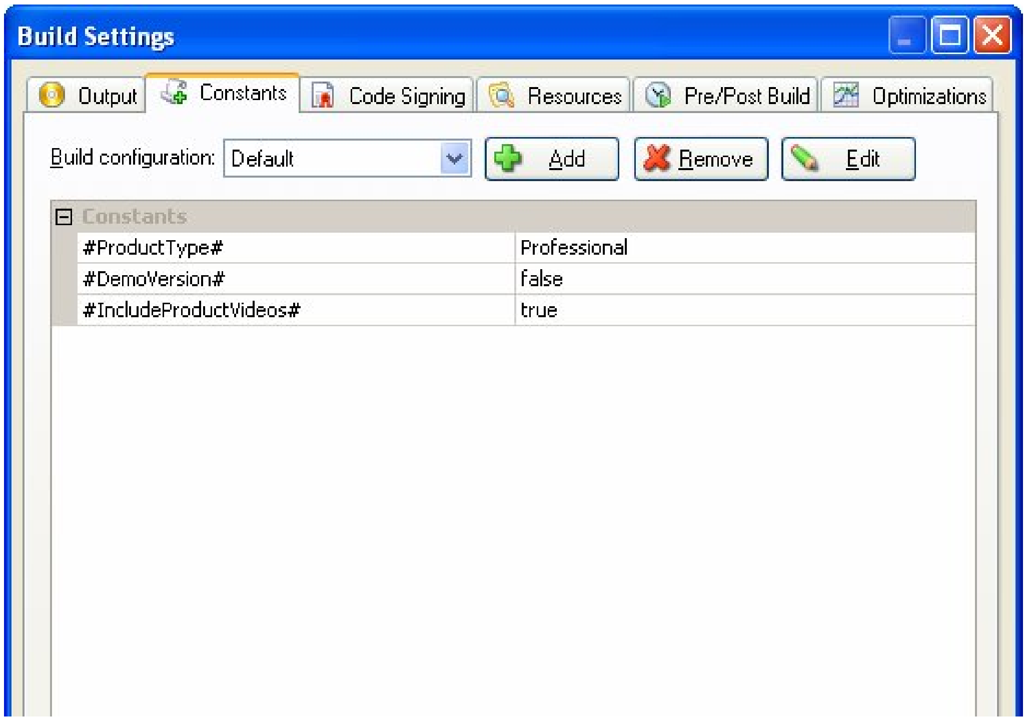
☑ 1.0.0.0
☑ 1.0.0.1
☑ 1.0.0.2

Patch output settings

Constants

Constants are essentially design time variables. They are similar to session variables, but instead of being expanded as the user runs the patch, they are expanded when you build the project. In other words, when you build your project, all of the constants are automatically replaced by the values assigned to them.

You can define constants on the Constants tab of the Build Settings dialog.

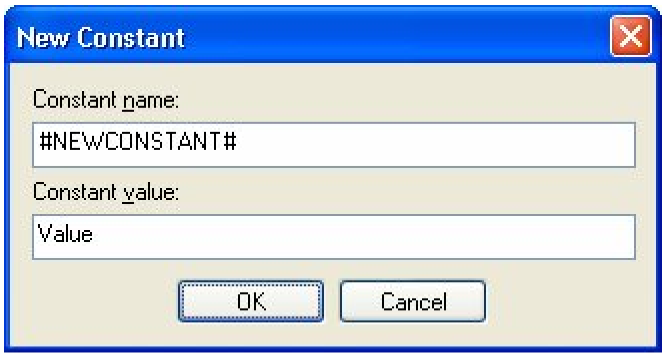


Build configuration: Default

Constants	
#ProductType#	Professional
#DemoVersion#	false
#IncludeProductVideos#	true

Custom constants

Clicking the Add button displays a dialog where you can name the constant and give it a value.



Constant name: #NEWCONSTANT#

Constant value: Value

Add a new constant

Each constant has a name that begins and ends with a # sign, and an associated value. The name will be replaced by this value throughout the project when it is built. It's exactly like a big search-and-replace operation that happens whenever you build the project.

Since each constant is essentially just a name that gets replaced with different text, you can use them just about anywhere. You can use them on screens, in file paths, in actions…pretty much anywhere that you can enter text.

Build Configurations

One of the many timesaving features of Visual Patch is the ability to define build configurations. Build configurations allow you to define different build settings for your project and switch between them easily.

For example, it's often useful to have a build configuration for each type of patch that you wish to build. One build configuration could create a full-history patch, capable of updating all previous versions (e.g. 1.0, 1.1, 1.2, 1.3, 1.4) up to the current version. Another could be an incremental patch to bring the previous version up to the current one (e.g. from 1.4 to 1.5). A third could be a multi-version patch to bring two common previous versions up to the latest version (e.g. from 1.2 or 1.3 to 1.5). You could then build each type of patch by switching between the build configurations.

Note: Files and folders can also be associated with specific build configurations on their Conditions tab. This makes it easy to conditionally enable or disable the patching of specific files within a version.

You can add a build configuration to the current project by clicking the Add button (Add) near the top of the Constants tab of the Build Settings dialog.

Constants and Build Configurations

The values that you assign to constants are specific to each build configuration. In fact, this is what makes constants so useful. By assigning different values to the constants in each build configuration, you can make sweeping changes to your project by simply selecting a different build configuration when you build it.

Note: The build configuration determines which values are used when the project is built.

Here are some things you can do with constants and build configurations:

- use constants in the paths for your source files, and use different versions of the files for different build configurations (professional version vs. standard version patch)
- use actions that test the value of a constant named #DEBUG#, and set it to true in a “Debug” build configuration and false in all other configurations. Then you can leave helpful debugging code in your scripts, and only have it performed when you build the Debug configuration. For example:

```
function MyFunc()
    nCount = nCount + 1;
    if #DEBUG# then
       Debug.ShowWindow();
       Debug.Print("Inside MyFunc() - nCount = " .. nCount);
    end
end
```

Constants and Unattended Builds

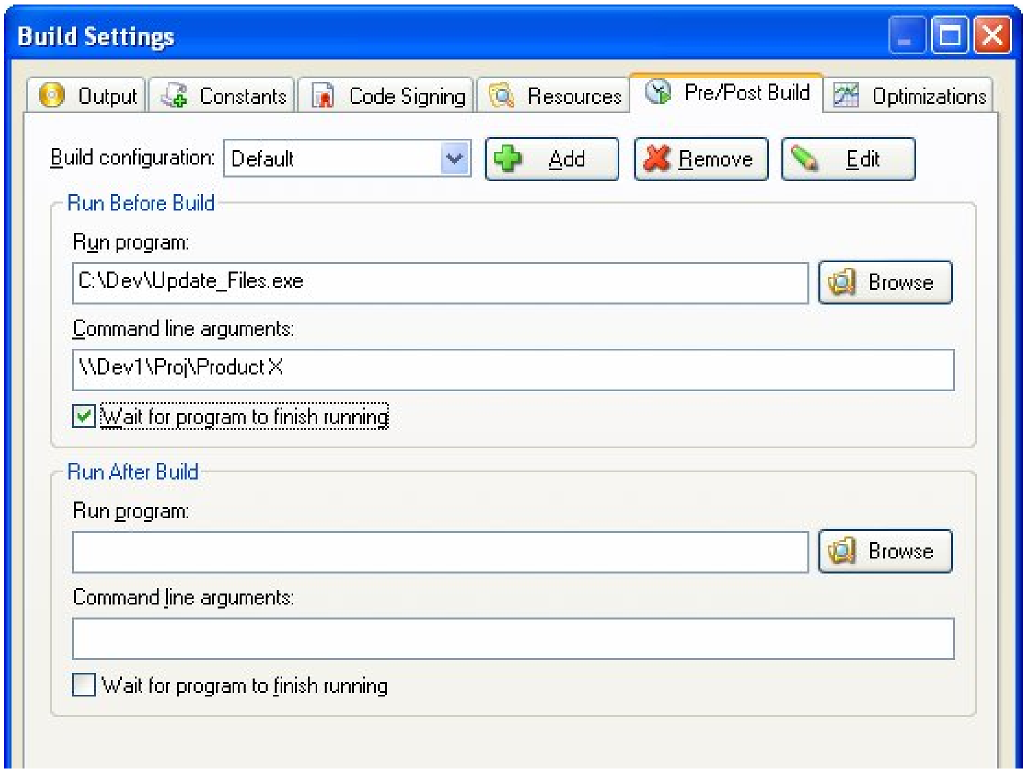
Constants are extremely useful for performing unattended builds because they can be changed using a command line argument. You can even use constants to specify the output location, and which build configuration to use. The possibilities are endless.

For example, you could create a batch file that would generate a unique patch for every one of your customers, with the customer’s name showing up on one or all of the screens during the patch.

For more information on performing unattended builds of your project, search for “Unattended Build Options” in the Visual Patch help file.

Pre/Post Build.

There may be instances where you want to run a program either before or after you build your patch. Visual Patch makes this process easy through Pre and Post Build Steps. On the Pre/Post Build tab of the Build Settings dialog, you can set a program to Run Before Build, and a program to Run After Build.



Build configuration: Default

Run Before Build

Run program: C:\Dev\Update_Files.exe

Command line arguments: \\Dev1\Proj\Product X

[✓] Wait for program to finish running

Run After Build

Run program:

Command line arguments:

[ ] Wait for program to finish running

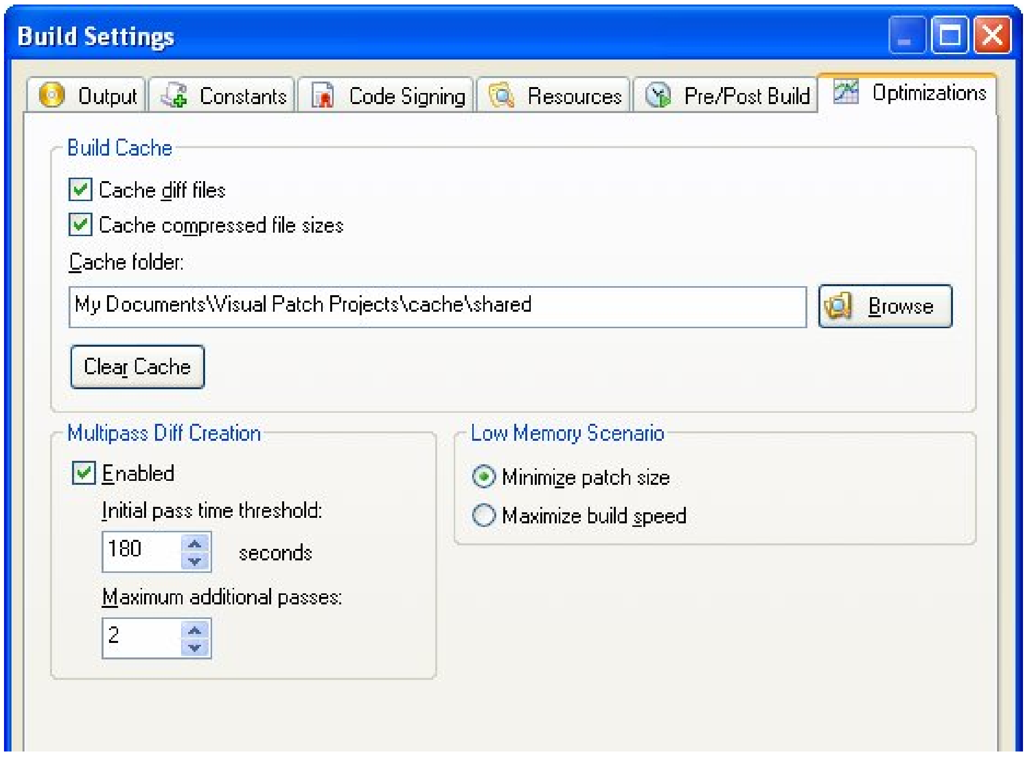
Run programs before and/or after the build process

The uses of this feature are limited only by your imagination. You could, for example, launch a program that would compare all files from your company's development network to the files about to be patched to ensure the most up-to-date files are included – the program could alert you if the local files do not match those on the network. You would want this program run at the last possible second to ensure last-minute bug fixes are not left out. Alternatively, you could have Visual Patch call an FTP client program with command line arguments to automatically upload your latest patch to your website. The possibilities are limitless.

You can specify the path to the desired program in either of the Run Program fields. Any command line arguments that are needed should be specified in the Command Line arguments fields. If you want Visual Patch to 'sleep' while the other program is open, check the Wait for program to finish running checkbox.

Optimizations

There are a number of size and speed optimizations on the build settings dialog that allow some control over the build process. For example, enabling the "Cache diff files" and "Cache compressed file sizes" options allow Visual Patch to store any binary difference files that it creates so it only needs to perform the delta compression once for each file, instead of every time the patch is built.



Cache diff files [✓]
Cache compressed file sizes [✓]
Cache folder: My Documents\Visual Patch Projects\cache\shared
Enabled [✓]
Initial pass time threshold: 180 seconds
Maximum additional passes: 2
[✓ Minimize patch size] Maximize build speed

Build process optimizations

Enabling the "Cache diff files" option stores the generated diff files in a folder on your system so they can be reused the next time you build, instead of having to be generated again. Subsequent builds of the same project are much faster when this option is enabled.

Likewise, Visual Patch calculates the compressed file size of each file and compares this value to the total size of all the diff files that are generated for that file. Enabling the “Cache compressed file sizes” option stores the compressed size information so it doesn’t need to be recalculated the next time you build the project.

The Multipass Diff Creation options determine whether Visual Patch will perform multiple passes on each file in order to achieve the smallest possible diff size. Each successive pass will look for longer patterns in the file, which, depending on the internal structure of the file, may result in better compression.

Since each pass adds to the creation time, you can limit the multipass optimization so it is only performed when the initial pass takes less than a specific amount of time. This is known as the “Initial pass time threshold.”

The “Maximum additional passes” option controls the maximum number of additional passes that will be attempted in order to search for successively longer patterns.

Tip: Enable the “Cache diff files” option and set the “Initial pass time threshold” to 0 so the multipass optimization will be performed for every file, but only the first time you build the project. This will allow Visual Patch to generate the smallest diff files, and will allow subsequent builds to be performed as quickly as possible.

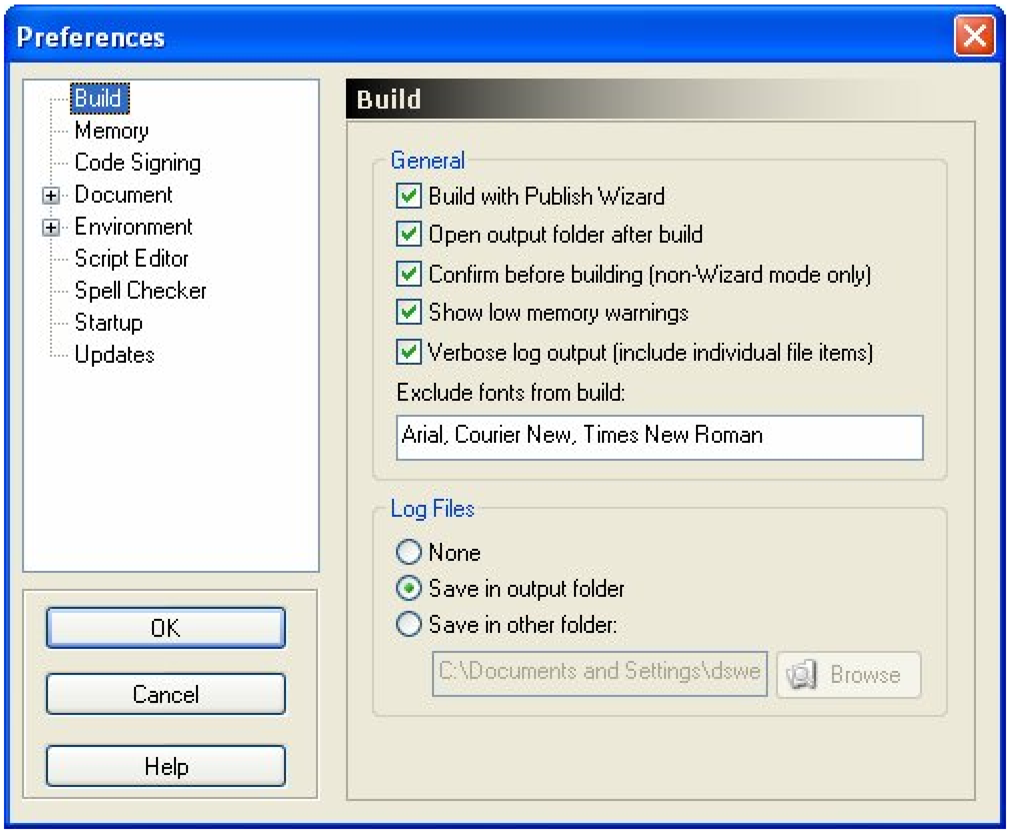
The Low Memory Scenario options allow you decide how you want Visual Patch to behave in low memory situations. You can choose to favor small patch sizes or shorter build times. In general, it is best to keep this set to “Minimize patch size,” however you may wish to change the setting on systems with limited memory resources if you find that your builds are taking a very long time to complete.

Tip: For large files, the size of the generated diff files depends on your computer’s available memory. Because of this, it is recommended that you clear out the diff cache (by clicking the Clear Cache button) whenever your computer’s RAM is upgraded.

Build Preferences

To control how Visual Patch handles the build process, you can modify the build preferences by choosing Edit > Preferences from the menu.

You can use the build preferences to control whether Visual Patch opens the output folder by default, disable the publish wizard and, when not using the wizard, control whether a confirmation dialog is displayed before the build process begins.



- [✓] Build with Publish Wizard
- [✓] Open output folder after build
- [✓] Confirm before building (non-Wizard mode only)
- [✓] Show low memory warnings
- [✓] Verbose log output (include individual file items)

Exclude fonts from build: Arial, Courier New, Times New Roman

Log Files: None [✓ Save in output folder] Save in other folder: C:\Documents and Settings\dswe

Preferences dialog (Edit > Preferences)

Testing Your Patch

One of the most important and often overlooked steps when creating a patch is testing it after it has been built. You should test your patch on as many computers and operating systems as possible. Try it on every operating system that your software product supports. All versions of windows have differences between them, and should be thoroughly tested. As well, test your patch on systems with a lot of hard-drive space and on those with a very limited amount of hard drive space. And lastly, test your patch on a system where the defaults for your program's installation were not used. For example, install your software to C:\SomeDirectory\ and then patch it. Does your patch fail, or does it work as expected?

If you have made use of Visual Patch's multilingual support, be sure to test out each language in your patch.

Many problems with patches have simple causes, such as forgetting to include a file, or moving a file to an incorrect location. Often a runtime problem is simply due to the patch being unable to locate the files to patch. Care must be taken when designing the directory structure of your patch, ensuring that it is compatible with the installed software being patched.

If you do run into a problem with your patch, test it on as many systems as possible to narrow down the problem. You might discover a common factor between the systems that is causing the patch to fail.

Tip: For information related to tracking down script-related issues, see *Debugging Your Scripts* in Chapter 10.

Distributing Your Patch

Once you have thoroughly tested your patch, it is time to distribute it. There are several ways to distribute your software patch, but the four main media are floppy disks, CD-ROMs, DVDs, and the Internet.

Preparing your patch for distribution is as easy as publishing to a hard drive folder. Once the build operation has completed, simply copy the patch executable to your desired media, and you are ready to go! Or, if you have opted to distribute via the Internet, simply take the generated patch file and upload it onto your website for your customers to download.

For an automated patch delivery system, consider TrueUpdate. TrueUpdate incorporates easily into any windows-based application, and allows the user to check for updates through your software application. Or, your application could use TrueUpdate to check for updates periodically. TrueUpdate supports either a wizard, dialog, or silent interface.

TrueUpdate isn't just an easy way to let your customers know software updates are available; it can automatically select the best patch to apply. For example, consider a product that has experienced fifteen version changes. You may choose to publish one full history patch, and one patch that updates the last three versions to the most current version. TrueUpdate can check the user's version and download the appropriate patch, potentially saving you bandwidth costs and your users time.

For more information on TrueUpdate, please contact Indigo Rose Software or visit the TrueUpdate website at www.trueupdate.com.

Chapter 10:

Scripting Guide

One of the powerful features of Visual Patch is its scripting engine. This chapter will introduce you to the new scripting environment and language.

Visual Patch scripting is very simple, with only a handful of concepts to learn. Here is what it looks like:

```
a = 5;
if a < 10 then
      Dialog.Message("Guess what?", "a is less than 10");
end
```

(Note: this script is only a demonstration. Don’t worry if you don’t understand it yet.)

The example above assigns a value to a variable, tests the contents of that variable, and if the value turns out to be less than 10, uses a Visual Patch action called “Dialog.Message” to display a message to the user.

New programmers and experienced coders alike will find that Visual Patch is a powerful, flexible yet simple scripting environment to work in.

10

In This Chapter

In this chapter, you'll learn about:

- Important scripting concepts
- Variables
- Variable scope and variable naming
- Types and values
- Expressions and operators
- Control structures (if, while, repeat, and for)
- Tables (arrays)
- Functions
- String manipulation
- Debugging your scripts
- Syntax errors and functional errors
- Other scripting resources

Before You Begin

In order to try out the example scripts in this chapter, you will need a basic project that you can build so you can run the patch application and see the scripts in action.

If you already have a Visual Patch project established, you can use a copy of it for this purpose. (Definitely use a *copy* of the project, though, so you can modify its scripts freely.)

Alternatively, you can create a simple project to use just for trying out your scripts.

All you need is a project that you can successfully build. Even if you have no intention of ever publishing the project, it will need a few items in order to get past the various checks performed during the build process.

The minimum, therefore, is a project with:

- at least two version tabs
- at least one file on each tab
- at least one key file on each tab

By default, every new project in Visual Patch has some standard scripts that provide default functionality and serve as a starting point for the project. For the purposes of this chapter, however, you will want to remove the default actions.

You can remove a default script just like you would remove any text in a text editor: simply select the text (e.g. press Ctrl+A to select it all) and press the Delete key.

The examples in this chapter assume that you have removed the default script from the On Startup event (Project > Actions) and that the project can build successfully.

Tip: If you want to test actions in the On Startup event without proceeding with the rest of the patching process, add an Application.Exit() action to the end of your script for that event. This action will immediately exit from the patch application as soon as it is encountered in the script.

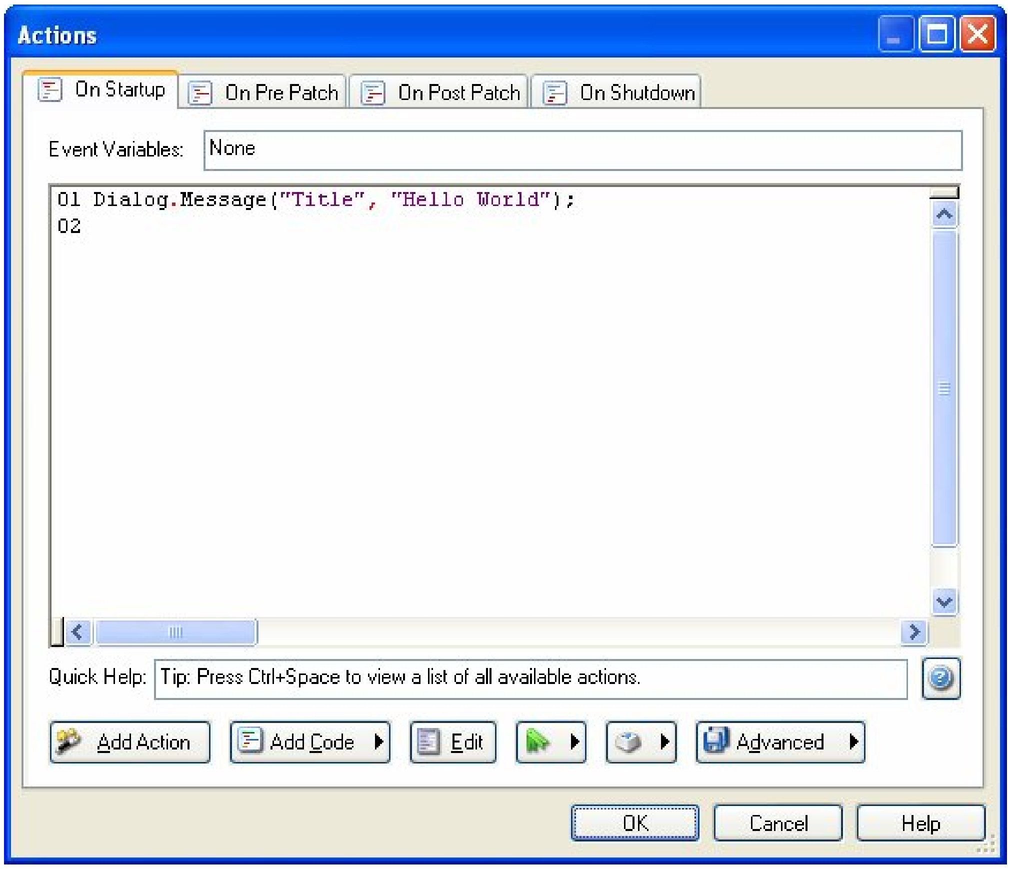
A Quick Example of Scripting in Visual Patch

Here is a short tutorial showing you how to enter a script into Visual Patch and preview the results.

1. Set up a working project as described in the previous section, *Before you Begin*.
2. In the On Startup event (Project > Actions), add the following line:

```
Dialog.Message("Title", "Hello World");
```

It should look like this when you're done:



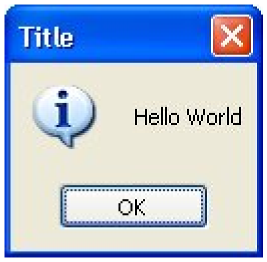
Event Variables: None

```
Dialog.Message("Title", "Hello World");
```

Quick Help: Tip: Press Ctrl+Space to view a list of all available actions.

3. Click OK to close the action editor.
4. Choose Publish > Build from the menu, and go through the publish wizard.
5. Once you have built the patch, run it so the script you entered is performed.

   You should see the following dialog appear:



Congratulations! You have just made your first script. Though this is a simple example, it shows you just how easy it is to make something happen in your project. You can use the above method to try out any script you want in Visual Patch.

Note: If you are working with actions that interact with screens, you will need to perform the actions from a screen event.

Important Scripting Concepts

There are a few important things that you should know about the Visual Patch scripting language in general before we go on.

Script is Global

The scripting engine is global to the runtime environment. That means that all of your events will “know” about other variables and functions declared elsewhere in the product. For example, if you assign “myvar = 10;” in the project’s On Startup event, myvar will still equal 10 when the next event is triggered. There are ways around this global nature (see *Variable Scope* on page 224), but it is generally true of the scripting engine.

Script is Case-Sensitive

The scripting engine is case-sensitive. This means that upper and lower case characters are important for things like keywords, variable names and function names.

For example:

```
ABC = 10;
aBC = 7;
```

In the above script, ABC and aBC refer to two different variables, and can hold different values. The lowercase "a" in "aBC" makes it completely different from "ABC" as far as Visual Patch is concerned.

The same principle applies to function names as well. For example:

```
Dialog.Message("Hi", "Hello World");
```

...refers to a built-in Visual Patch function. However,

```
DIALOG.Message("Hi", "Hello World");
```

...will not be recognized as the built-in function, because DIALOG and Dialog are seen as two completely different names.

Note: It's entirely possible to have two functions with the same spelling but different capitalization—for example, GreetUser and gREeTUSeR would be seen as two totally different functions. Although it's definitely possible for such functions to coexist, it's generally better to give functions completely different names to avoid any confusion.

Comments

You can insert non-executable comments into your scripts to explain and document your code. In a script, any text after two dashes (--) on a line will be ignored. For example:

```
-- Assign 10 to variable abc
abc = 10;
```

...or:

```
abc = 10; -- Assign 10 to abc
```

Both of those examples do the exact same thing—the comments do not affect the script in any way.

You can also create multi-line comments by using --[[and]]-- on either side of the comment:

```
--[[ This is
a multi-line
comment ]]--
a = 10;
```

You should use comments to explain your scripts as much as possible in order to make them easier to understand by yourself and others.

Delimiting Statements

Each unique statement can either be on its own line and/or separated by a semi-colon (;). For example, all of the following scripts are valid:

Script 1:

```
a = 10
MyVar = a
```

Script 2:

```
a = 10; MyVar = a;
```

Script 3:

```
a = 10;
MyVar = a;
```

However, we recommend that you end all statements with a semi-colon (as in scripts 2 and 3 above).

Variables

What are Variables?

Variables are very important to scripting in Visual Patch. Variables are simply "nicknames" or "placeholders" for values that might need to be modified or re-used in the future. For example, the following script assigns the value 10 to a variable called "amount."

```
amount = 10;
```

Note: We say that values are "assigned to" or "stored in" variables. If you picture a variable as a container that can hold a value, assigning a value to a variable is like "placing" that value into a container. You can change this value at any time by assigning a different value to the variable; the new value simply replaces the old one. This ability to hold changeable information is what makes variables so useful.

Here are a couple of examples demonstrating how you can operate on the "amount" variable:

```
amount = 10;
amount = amount + 20;
Dialog.Message("Value", amount);
```

This stores 10 in the variable named amount, then adds 20 to that value, and then finally makes a message box appear with the current value (which is now the number 30) in it.

You can also assign one variable to another:

```
a = 10;
b = a;
Dialog.Message("Value", b);
```

This will make a message box appear with the number 10 in it. The line "b = a;" assigns the value of "a" (which is 10) to "b."

Variable Scope

As mentioned earlier in this document, all variables in Visual Patch are *global* by default. This just means that they exist project-wide, and hold their values from one script to the next. In other words, if a value is assigned to a variable in one script, the variable will still hold that value when the next script is executed.

For example, if you enter the script:

```
foo = 10;
```

...into the patch's On Startup event, and then enter:

```
Dialog.Message("The value is:", foo);
```

...into a screen's On Preload event, the second script will use the value that was assigned to "foo" in the first script. As a result, when the screen loads, a message box will appear with the number 10 in it.

Note that the order of execution is important...in order for one script to be able to use the value that was assigned to the variable in another script, that other script has to be executed first. In the above example, the On Startup event is triggered *before* the screen's On Preload event, so the value 10 is already assigned to foo when the On Startup event's script is executed.

Local Variables

The global nature of the scripting engine means that a variable will retain its value throughout your entire project. You can, however, make variables that are non-global, by using the special keyword "local." Putting the word "local" in front of a variable assignment creates a variable that is local to the script, function, or block of code.

For example, let's say you have the following three scripts in the same project:

Script 1:

```
-- assign 10 to x
x = 10;
```

Script 2:

```
local x = 500;
Dialog.Message("Local value of x is:", x);
x = 250; -- this changes the local x, not the global one
Dialog.Message("Local value of x is:", x);
```

Script 3:

```
-- display the global value of x
Dialog.Message("Global value of x is:", x);
```

Let’s assume these three scripts are performed one after the other. The first script gives x the value 10. Since all variables are global by default, x will have this value inside all other scripts, too. The second script makes a *local* assignment to x, giving it the value of 500—but only inside that script. If anything else inside that script wants to access the value of x, it will see the local value instead of the global one. It’s like the “x” variable has been temporarily replaced by another variable that looks just like it, but has a different value.

(This reminds me of those caper movies, where the bank robbers put a picture in front of the security cameras so the guards won’t see that the vault is being emptied. Only in this case, it’s like the bank robbers create a whole new working vault, just like the original, and then dismantle it when they leave.)

When told to display the contents of x, the first Dialog.Message action inside script #2 will display 500, since that is the local value of x when the action is performed. The next line assigns 250 to the local value of x—note that once you make a local variable, it completely replaces the global variable for the rest of the script.

Finally, the third script displays the global value of x, which is still 10.

Variable Naming

Variable names can be made up of any combination of letters, digits and underscores as long as they do not begin with a number and do not conflict with reserved keywords.

Examples of **valid** variables names:

a
strName
_My_Variable
data1
data_1_23
index
bReset
nCount

Examples of **invalid** variable names:

1
1data
%MyValue%
$strData
for
local
FirstName+LastName
User Name

Reserved Keywords

The following words are reserved and cannot be used for variable or function names:

and	break	do	else	elseif
end	false	for	function	if
in	local	nil	not	or
repeat	return	table	then	true
until	while			

Types and Values

Visual Patch's scripting language is dynamically typed. There are no type definitions—instead, each value carries its own type.

What this means is that you don't have to declare a variable to be of a certain type before using it. For example, in C++, if you want to use a number, you have to first declare the variable's type and then assign a value to it:

```
int j;
j = 10;
```

The above C++ example declares j as an integer, and then assigns 10 to it.

As we have seen, in Visual Patch you can just assign a value to a variable without declaring its type. Variables don’t really have types; instead, it’s the values inside them that are considered to be one type or another. For example:

```
j = 10;
```

...this automatically creates the variable named “j” and assigns the value 10 to it. Although this value has a type (it’s a ***number***), the variable itself is still typeless. This means that you can turn around and assign a different type of value to j, like so:

```
j = "Hello";
```

This replaces the number 10 that is stored in j with the string “Hello.” The fact that a string is a different type of value doesn’t matter; the variable j doesn’t care what kind of value it holds, it just stores whatever you put in it.

There are six basic data types in Visual Patch: number, string, nil, Boolean, function, and table. The sections below will explain each data type in more detail.

Number

A number is exactly that: a numeric value. The number type represents real numbers—specifically, double-precision floating-point values. There is no distinction between integers and floating-point numbers (also known as “fractions”)...all of them are just “numbers.” Here are some examples of valid numbers:

4   4.   .4   0.4   4.57e-3   0.3e12

String

A string is simply a sequence of characters. For example, “Joe2” is a string of four characters, starting with a capital “J” and ending with the number “2.” Strings can vary widely in length; a string can contain a single letter, or a single word, or the contents of an entire book.

Strings may contain spaces and even more exotic characters, such as carriage returns and line feeds. In fact, strings may contain any combination of valid 8-bit ASCII characters, including null characters (“\0”). Visual Patch automatically manages string memory, so you never have to worry about allocating or de-allocating memory for strings.

Strings can be used quite intuitively and naturally. They should be delimited by matching single quotes or double quotes. Here are some examples that use strings:

```
Name = "Joe Blow";
Dialog.Message("Title", "Hello, how are you?");
LastName = 'Blow';
```

Normally double quotes are used for strings, but single quotes can be useful if you have a string that contains double quotes. Whichever type of quotes you use, you can include the other kind inside the string without escaping it. For example:

```
doubles = "How's that again?";
singles = 'She said "Talk to the hand," and I was all like "Dude!"';
```

If we used double quotes for the second line, it would look like this:

```
escaped = "She said \"Talk to the hand,\" and I was all like \"Dude!\"";
```

Normally, the scripting engine sees double quotes as marking the beginning or end of a string. In order to include double quotes inside a double-quoted string, you need to *escape* them with backslashes. This tells the scripting engine that you want to include an actual quote character *in* the string.

The backslash and quote (\") is known as an *escape sequence*. An escape sequence is a special sequence of characters that gets converted or “translated” into something else by the script engine. Escape sequences allow you to include things that can’t be typed directly into a string.

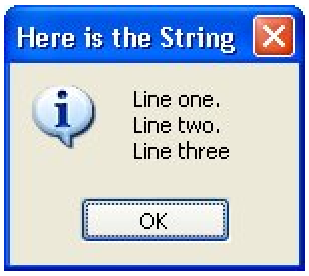
The escape sequences that you can use include:

\a	-	bell
\b	-	backspace
\f	-	form feed
\n	-	newline
\r	-	carriage return
\t	-	horizontal tab
\v	-	vertical tab
\\	-	backslash
\"	-	quotation mark
\'	-	apostrophe
\[	-	left square bracket
\]	-	right square bracket

So, for example, if you want to represent three lines of text in a single string, you would use the following:

```
Lines = "Line one.\nLine two.\nLine three";
Dialog.Message("Here is the String", Lines);
```

This assigns a string to a variable named Lines, and uses the newline escape sequence to start a new line after each sentence. The Dialog.Message function displays the contents of the Lines variable in a message box, like this:



Here is the String

```
Line one.
Line two.
Line three
```

Another common example is when you want to represent a path to a file such as C:\My Folder\My Data.txt. You just need to remember to escape the backslashes:

```
MyPath = "C:\\My Folder\\My Data.txt";
```

Each double-backslash represents a single backslash when used inside a string.

If you know your ASCII table, you can use a backslash character followed by a number with up to three digits to represent any character by its ASCII value. For example, the ASCII value for a newline character is 10, so the following two lines do the exact same thing:

```
Lines = "Line one.\nLine two.\nLine three";
Lines = "Line one.\10Line two.\10Line three";
```

However, you will not need to use this format very often, if ever.

You can also define strings on multiple lines by using double square brackets ([[and]]). A string between double square brackets does not need any escape characters. The double square brackets let you type special characters like backslashes, quotes and newlines right into the string.

For example:

```
Lines = [[Line one.
Line two.
Line three.]];
```

is equivalent to:

```
Lines = "Line one.\nLine two.\nLine three";
```

This can be useful if you have preformatted text that you want to use as a string, and you don't want to have to convert all of the special characters into escape sequences.

The last important thing to know about strings is that the script engine provides automatic conversion between numbers and strings at run time. Whenever a numeric operation is applied to a string, the engine tries to convert the string to a number for the operation. Of course, this will only be successful if the string contains something that can be interpreted as a number.

For example, the following lines are both valid:

```
a = "10" + 1; -- Result is 11
b = "33" * 2; -- Result is 66
```

However, the following lines would not give you the same conversion result:

```
a = "10+1"; -- Result is the string "10+1"
b = "hello" + 1; -- ERROR, can't convert "hello" to a number
```

For more information on working with strings, see page 255.

nil

nil is a special value type. It basically represents the absence of any other kind of value.

You can assign nil to a variable, just like any other value. Note that this isn't the same as assigning the letters "nil" to a variable, as in a string. Like other keywords, nil must be left unquoted in order to be recognized. It should also be entered in all lowercase letters.

nil will always evaluate to false when used in a condition:

```
a = nil;
if a then
    -- Any lines in here
    -- will not be executed
end
```

It can also be used to “delete” a variable:

```
y = "Joe Blow";
y = nil;
```

In the example above, “y” will no longer contain a value after the second line.

Boolean

Boolean variable types can have one of two values: true, or false. They can be used in conditions and to perform Boolean logic operations. For example:

```
boolybooly = true;
if boolybooly then
    -- Any script in here will be executed
end
```

This sets a variable named boolybooly to true, and then uses it in an if statement. Similarly:

```
a = true;
b = false;
if (a and b) then
    -- Any script here will not be executed because
    -- true and false is false.
end
```

This time, the if statement needs both “a” and “b” to be true in order for the lines inside it to be executed. In this case, that won’t happen because “b” has been set to false.

Function

The script engine allows you to define your own functions (or “sub-routines”), which are essentially small pieces of script that can be executed on demand. Each function has a name which is used to identify the function. You can actually use that function name as a special kind of value, in order to store a “reference” to that function in a variable, or to pass it to another function. This kind of reference is of the *function* type.

For more information on functions, see page 250.

Table

Tables are a very powerful way to store lists of indexed values under one name. Tables are actually associative arrays—that is, they are arrays which can be indexed not only with numbers, but with any kind of value (including strings).

Here are a few quick examples (we cover tables in more detail on page 242):

Example 1:

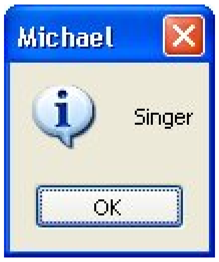
```
guys = {"Adam", "Brett", "Darryl"};
Dialog.Message("Second Name in the List", guys[2]);
```

This will display a message box with the word “Brett” in it.

Example 2:

```
t = {};
t.FirstName = "Michael";
t.LastName = "Jackson";
t.Occupation = "Singer";
Dialog.Message(t.FirstName, t.Occupation);
```

This will display the following message box:



You can assign tables to other variables as well. For example:

```
table_one = {};
table_one.FirstName = "Bruce";
table_one.LastName = "Springsteen";
table_one.Occupation = "Singer";
table_two = table_one;
occupation = table_two.Occupation;
Dialog.Message(b.FirstName, occupation);
```

Tables can be indexed using array notation (my_table[1]), or by dot notation if not indexed by numbers (my_table.LastName).

Note that when you assign one table to another, as in the following line:

```
table_two = table_one;
```

...this doesn't actually copy table_two into table_one. Instead, table_two and table_one both refer to the *same* table.

This is because the name of a table actually refers to an address in memory where the data within the table is stored. So when you assign the contents of the variable table_one to the variable table_two, you're copying the *address*, and not the actual data. You're essentially making the two variables "point" to the same table of data.

In order to copy the contents of a table, you need to create a new table and then copy all of the data over one item at a time.

For more information on copying tables, see page 247.

Variable Assignment

Variables can have new values assigned to them by using the assignment operator (=). This includes copying the value of one variable into another. For example:

```
a = 10;
b = "I am happy";
c = b;
```

It is interesting to note that the script engine supports multiple assignment:

```
a, b = 1, 2;
```

After the script above, the variable "a" contains the number 1 and the variable "b" contains the number 2.

Tables and functions are a bit of a special case: when you use the assignment operator on a table or function, you create an alias that points to the same table or function as the variable being “copied.” Programmers call this copying *by reference* as opposed to copying *by value*.

Expressions and Operators

An expression is anything that evaluates to a value. This can include a single value such as “6” or a compound value built with operators such as “1 + 3”. You can use parentheses to “group” expressions and control the order in which they are evaluated. For example, the following lines will all evaluate to the same value:

```
a = 10;
a = (5 * 1) * 2;
a = 100 / 10;
a = 100 / (2 * 5);
```

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numbers. The following mathematical operators are supported:

+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)
unary -	(negation)

Here are some examples:

```
a = 5 + 2;
b = a * 100;
twentythreepercent = 23 / 100;
neg = -29;
pos = -neg;
```

Relational Operators

Relational operators allow you to compare how one value relates to another. The following relational operators are supported:

>	(greater-than)
<	(less-than)
<=	(less-than or equal to)
>=	(greater than or equal to)
~=	(not equal to)
==	(equal)

All of the relational operators can be applied to any two numbers or any two strings. All other values can only use the == operator to see if they are equal.

Relational operators return Boolean values (true or false). For example:

```
10 > 20; -- resolves to false

a = 10;
a > 300; -- false

(3 * 200) > 500; -- true
"Brett" ~= "Lorne" -- true
```

One important point to mention is that the == and ~= operators test for *complete equality*, which means that any string comparisons done with those operators are case sensitive. For example:

```
"Jojoba" == "Jojoba"; -- true
"Wildcat" == "wildcat"; -- false
"I like it a lot" == "I like it a LOT"; -- false

"happy" ~= "HaPPy"; -- true
```

Logical Operators

Logical operators are used to perform Boolean operations on Boolean values. The following logical operators are supported:

and	(only true if both values are true)
or	(true if either value is true)
not	(returns the opposite of the value)

For example:

```
a = true;
b = false;
c = a and b; -- false
d = a and nil; -- false
e = not b; -- true
```

Note that only nil and false are considered to be false, and all other values are true.

For example:

```
iaminvisible = nil;
if iaminvisible then
    -- any lines in here won't happen
    -- because iaminvisible is considered false
    Dialog.Message("You can't see me!", "I am invisible!!!!");
end

if "Brett" then
    -- any lines in here WILL happen, because only nil and false
    -- are considered false...anything else, including strings,
    -- is considered true
    Dialog.Message("What about strings?", "Strings are true.");
end
```

Concatenation

In Visual Patch scripting, the concatenation operator is two periods (..). It is used to combine two or more strings together. You don’t have to put spaces before and after the periods, but you can if you want to.

For example:

```
name = "Joe".." Blow"; -- assigns "Joe Blow" to name
b = name .. " is number " .. 1; -- assigns "Joe Blow is number 1" to b
```

Operator Precedence

Operators are said to have *precedence*, which is a way of describing the rules that determine which operations in a series of expressions get performed first. A simple example would be the expression 1 + 2 * 3. The multiply (*) operator has higher precedence than the add (+) operator, so this expression is equivalent to 1 + (2 * 3). In other words, the expression 2 * 3 is performed first, and then 1 + 6 is performed, resulting in the final value 7.

You can override the natural order of precedence by using parentheses. For instance, the expression (1 + 2) * 3 resolves to 9. The parentheses make the whole sub-expression “1 + 2” the left value of the multiply (*) operator. Essentially, the sub-expression 1 + 2 is evaluated first, and the result is then used in the expression 3 * 3.

Operator precedence follows the following order, from lowest to highest priority:

and	or				
`<`	`>`	`<=`	`>=`	`~=`	`==`
..					
+	-				
*	/				
not	- (unary)				
^					

Operators are also said to have *associativity*, which is a way of describing which expressions are performed first when the operators have equal precedence. In the script engine, all binary operators are left associative, which means that whenever two operators have the same precedence, the operation on the left is performed first. The exception is the exponentiation operator (^), which is right-associative.

When in doubt, you can always use explicit parentheses to control precedence. For example:

```
a + 1 < b/2 + 1
```

...is the same as:

```
(a + 1) < ((b/2) + 1)
```

...and you can use parentheses to change the order of the calculations, too:

```
a + 1 < b/(2 + 1)
```

In this last example, instead of 1 being added to half of b, b is divided by 3.

Control Structures

The scripting engine supports the following control structures: if, while, repeat and for.

If

An if statement evaluates its condition and then only executes the “then” part if the condition is true. An if statement is terminated by the “end” keyword. The basic syntax is:

```
if condition then
    do something here
end
```

For example:

```
x = 50;
if x > 10 then
    Dialog.Message("result", "x is greater than 10");
end

y = 3;
if ((35 * y) < 100) then
    Dialog.Message("", "y times 35 is less than 100");
end
```

In the above script, only the first dialog message would be shown, because the second if condition isn't true...35 times 3 is 105, and 105 is not less than 100.

You can also use else and elseif to add more “branches” to the if statement:

```
x = 5;
if x > 10 then
    Dialog.Message("", "x is greater than 10");
else
    Dialog.Message("", "x is less than or equal to 10");
end
```

In the preceding example, the second dialog message would be shown, because 5 is not greater than 10.

```
x = 5;
if x == 10 then
    Dialog.Message("", "x is exactly 10");
elseif x == 11 then
    Dialog.Message("", "x is exactly 11");
elseif x == 12 then
    Dialog.Message("", "x is exactly 12");
else
    Dialog.Message("", "x is not 10, 11 or 12");
end
```

In that example, the last dialog message would be shown, because x is not equal to 10, or 11, or 12.

While

The while statement is used to execute the same “block” of script over and over until a condition is met. Like if statements, while statements are terminated with the “end” keyword. The basic syntax is:

```
while condition do
    do something here
end
```

The condition must be true in order for the actions inside the while statement (the “do something here” part above) to be performed. The while statement will continue to loop as long as this condition is true. Here's how it works:

If the condition is true, all of the actions between the “while” and the corresponding “end” will be performed. When the “end” is reached, the condition will be reevaluated, and if it's still true, the actions between the “while” and the “end” will be performed again. The actions will continue to loop like this until the condition evaluates to false.

For example:

```
a = 1;
while a < 10 do
    a = a + 1;
end
```

In the preceding example, the “a = a + 1;” line would be performed 9 times.

You can break out of a while loop at any time using the “break” keyword. For example:

```
count = 1;
while count < 100 do
    count = count + 1;
    if count == 50 then
        break;
    end
end
```

Although the while statement is willing to count from 1 to 99, the if statement would cause this loop to terminate as soon as count reached 50.

Repeat

The repeat statement is similar to the while statement, except that the condition is checked at the *end* of the structure instead of at the beginning. The basic syntax is:

```
repeat
    do something here
until condition
```

For example:

```
i = 1;
repeat
    i = i + 1;
until i > 10
```

This is similar to one of the while loops above, but this time, the loop is performed 10 times. The “i = i + 1;” part gets executed before the condition determines that i is now larger than 10.

You can break out of a repeat loop at any time using the “break” keyword. For example:

```
count = 1;
repeat
    count = count + 1;
    if count == 50 then
        break;
    end
until count > 100
```

Once again, this would exit from the loop as soon as count was equal to 50.

For

The for statement is used to repeat a block of script a specific number of times. The basic syntax is:

```
for variable = start, end, step do
    do something here
end
```

The *variable* can be named anything you want. It is used to “count” the number of trips through the for loop. It begins at the *start* value you specify, and then changes by the amount in *step* after each trip through the loop. In other words, the *step* gets added to the value in the *variable* after the lines between the for and end are performed. If the result is smaller than or equal to the *end* value, the loop continues from the beginning.

For example:

```
-- This loop counts from 1 to 10:
for x = 1, 10 do
    Dialog.Message("Number", x);
end
```

This displays 10 dialog messages in a row, counting from 1 to 10.

Note that the step is optional; if you don’t provide a value for the step, it defaults to 1.

Here’s an example that uses a step of -1 to make the for loop count backwards:

```
-- This loop counts from 10 down to 1:
for x = 10, 1, -1 do
    Dialog.Message("Number", x);
end
```

That example would display 10 dialog messages in a row, counting back from 10 and going all the way down to 1.

You can break out of a for loop at any time using the “break” keyword. For example:

```
for i = 1, 100 do
    if count == 50 then
        break;
    end
end
```

Once again, this would exit from the loop as soon as count was equal to 50.

There is also a variation on the for loop that operates on tables. For more information on that, see *Using For to Enumerate Tables* on page 245.

Tables (Arrays)

Tables are very useful. They can be used to store any type of value, including functions or even other tables.

Creating Tables

There are generally two ways to create a table from scratch. The first way uses curly braces to specify a list of values:

```
my_table = {"apple","orange","peach"};
```

```
associative_table = {fruit="apple", vegetable="carrot"}
```

The second way is to create a blank table and then add the values one at a time:

```
my_table = {};
my_table[1] = "apple";
my_table[2] = "orange";
my_table[3] = "peach";

associative_table = {};
associative_table.fruit = "apple";
associative_table.vegetable = "carrot";
```

Accessing Table Elements

Each “record” of information stored in a table is known as an *element*. Each element consists of a key, which serves as the index into the table, and a value that is associated with that key.

There are generally two ways to access an element: you can use array notation, or dot notation. Array notation is typically used with numeric arrays, which are simply tables where all of the keys are numbers. Dot notation is typically used with associative arrays, which are tables where the keys are strings.

Here is an example of array notation:

```
t = {"one", "two", "three"};
Dialog.Message("Element one contains:", t[1]);
```

Here is an example of dot notation:

```
t = {first="one", second="two", third="three"};
Dialog.Message("Element 'first' contains:", t.first);
```

Numeric Arrays

One of the most common uses of tables is as arrays. An array is a collection of values that are indexed by numeric keys. In the scripting engine, numeric arrays are one-based. That is, they start at index 1.

Here are some examples using numeric arrays:

Example 1:

```
myArray = {255,0,255};
Dialog.Message("First Number", myArray[1]);
```

This first example would display a dialog message containing the number “255.”

Example 2:

```
alphabet = {"a","b","c","d","e","f","g","h","i","j","k",
"l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"};
Dialog.Message("Seventh Letter", alphabet[7]);
```

This would display a dialog message containing the letter “g.”

Example 3:

```
myArray = {};
myArray[1] = "Option One";
myArray[2] = "Option Two";
myArray[3] = "Option Three";
```

This is exactly the same as the following:

```
myArray = {"Option One", "Option Two", "Option Three"};
```

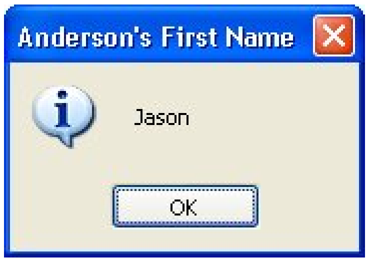
Associative Arrays

Associative arrays are the same as numeric arrays except that the indexes can be numbers, strings or even functions.

Here is an example of an associative array that uses a last name as an index and a first name as the value:

```
arrNames = {Anderson="Jason",
            Clemens="Roger",
            Contreras="Jose",
            Hammond="Chris",
            Hitchcock="Alfred"};

Dialog.Message("Anderson's First Name", arrNames.Anderson);
```

The resulting dialog message would look like this:



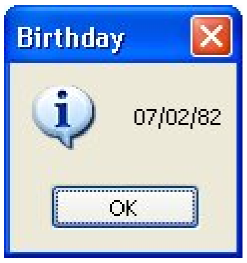
Anderson's First Name

Jason

Here is an example of a simple employee database that keeps track of employee names and birth dates indexed by employee numbers:

```
Employees = {}; -- Construct an empty table for the employee numbers

-- store each employee's information in its own table
Employee1 = {Name="Jason Anderson", Birthday="07/02/82"};
Employee2 = {Name="Roger Clemens", Birthday="12/25/79"};

-- store each employee's information table
-- at the appropriate number in the Employees table
Employees[100099] = Employee1;
Employees[137637] = Employee2;

-- now typing "Employees[100099]" is the same as typing "Employee1"
Dialog.Message("Birthday",Employees[100099].Birthday);
```

The resulting dialog message would look like this:



07/02/82

Using For to Enumerate Tables

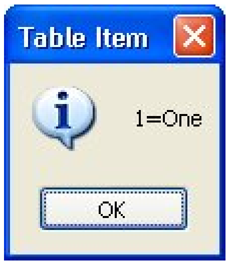
There is a special version of the for statement that allows you to quickly and easily enumerate the contents of an array. The syntax is:

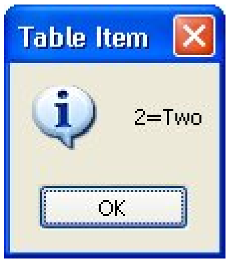
```
for index,value in table do
    operate on index and value
end
```

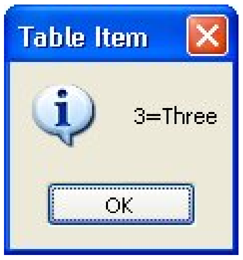
For example:

```
mytable = {"One","Two","Three"};

-- display a message for every table item
for j,k in mytable do
    Dialog.Message("Table Item", j .. "=" .. k);
end
```

The result would be three dialog messages in a row, one for each of the elements in mytable, like so:







Remember the above for statement, because it is a quick and easy way to inspect the values in a table. If you just want the indexes of a table, you can leave out the *value* part of the for statement:

```
a = {One=1, Two=2, Three=3};

for k in a do
    Dialog.Message("Table Index", k);
end
```

The above script will display three message boxes in a row, with the text “One,” “Three,” and then “Two.”

Whoa there—why aren’t the table elements in order? The reason for this is that internally the scripting engine doesn’t store tables as arrays, but in a super-efficient structure known as a hash table. The important thing to know is that when you define table elements, they are not necessarily stored in the order that you define or add them, unless you use a numeric array (i.e. a table indexed with numbers from 1 to whatever).

Copying Tables

Copying tables is a bit different from copying other types of values. Unlike variables, you can't just use the assignment operator to copy the contents of one table into another. This is because the name of a table actually refers to an address in memory where the data within the table is stored. If you try to copy one table to another using the assignment operator, you end up copying the address, and not the actual data.

For example, if you wanted to copy a table, and then modify the copy, you might try something like this:

```
table_one = { mood="Happy", temperature="Warm" };

-- create a copy
table_two = table_one;

-- modify the copy
table_two.temperature = "Cold";

Dialog.Message("Table one temperature is:", table_one.temperature);
Dialog.Message("Table two temperature is:", table_two.temperature);
```

If you ran this script, you would see the following two dialogs:

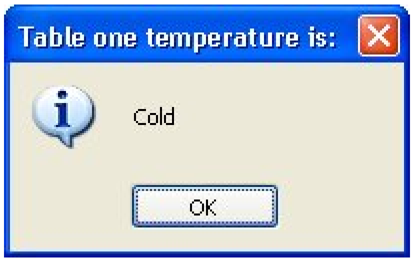


Table one temperature is:

Cold

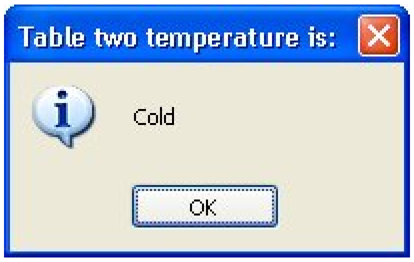


Table two temperature is:

Cold

Wait a minute...changing the “temperature” element in table_two also changed it in table_one. Why would they both change?

The answer is simply because the two are in fact the same table.

Internally, the name of a table just refers to a memory location. When table_one is created, a portion of memory is set aside to hold its contents. The location (or “address”) of this memory is what gets assigned to the variable named table_one.

Assigning table_one to table_two just copies that memory address—not the actual memory itself.

It’s like writing down the address of a library on a piece of paper, and then handing that paper to your friend. You aren’t handing the entire library over, shelves of books and all...only the location where it can be found.

If you wanted to actually copy the library, you would have to create a new building, photocopy each book individually, and then store the photocopies in the new location.

That’s pretty much how it is with tables, too. In order to create a full copy of a table, contents and all, you need to create a new table and then copy over all of the elements, one element at a time.

Luckily, the for statement makes this really easy to do. For example, here’s a modified version of our earlier example, that creates a “true” copy of table_one.

```
table_one = { mood="Happy", temperature="Warm" };

-- create a copy
table_two = {};
for index, value in table_one do
    table_two[index] = value;
end

-- modify the copy
table_two.temperature = "Cold";

Dialog.Message("Table one temperature is:", table_one.temperature);
Dialog.Message("Table two temperature is:", table_two.temperature);
```

This time, the dialogs show that modifying table_two doesn't affect table_one at all:

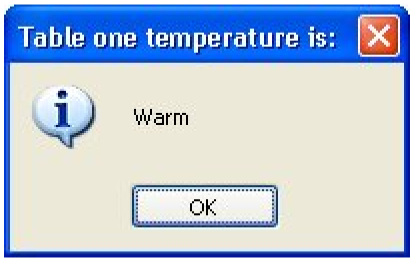


Table one temperature is:

Warm

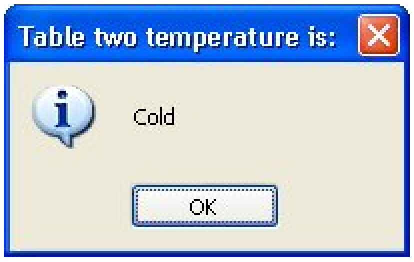


Table two temperature is:

Cold

Table Functions

There are a number of built-in table functions at your disposal, which you can use to do such things as inserting elements into a table, removing elements from a table, and counting the number of elements in a table. For more information on these table functions, please see *Program Reference / Actions / Table* in the online help.

Functions

By far the coolest and most powerful feature of the scripting engine is functions. You have already seen a lot of functions used throughout this document, such as "Dialog.Message." Functions are simply portions of script that you can define, name and then call from anywhere else.

Although there are a lot of built-in Visual Patch functions, you can also make your own custom functions to suit your specific needs. In general, functions are defined as follows:

```
function function_name (arguments)
    function script here
    return return_value;
end
```

The first part is the keyword "function." This tells the scripting engine that what follows is a function definition. The *function_name* is simply a unique name for your function. The ***arguments*** are parameters (or values) that will be passed to the function every time it is called. A function can receive any number of arguments from 0 to infinity (well, not infinity, but don't get technical on me). The "return" keyword tells the function to return one or more values back to the script that called it.

The easiest way to learn about functions is to look at some examples. In this first example, we will make a simple function that shows a message box. It does not take any arguments and does not return anything.

```
function HelloWorld()
    Dialog.Message("Welcome","Hello World");
end
```

Notice that if you put the above script into an event and build your install, nothing script related happens. Well, that is true and not true. It is true that nothing visible happens but the magic is in what you don't see. When the event is fired and the function script is executed, the function called "HelloWorld" becomes part of the scripting engine. That means it is now available to the rest of the install in any other script.

This brings up an important point about scripting in Visual Patch. When making a function, the function does not get "into" the engine until the script is executed. That means that if you define HelloWorld() in a screen's On Preload event, but that event never gets triggered (because the screen is never displayed), the HelloWorld()

function will never exist. That is, you will not be able to call it from anywhere else. That is why, in general, it is best to define your global functions in the global script of the project. (To access the global script, choose Resources > Global Functions from the menu.)

Now back to the good stuff. Let’s add a line to actually call the function:

```
function HelloWorld()
    Dialog.Message("Welcome","Hello World");
end

HelloWorld();
```

The “HelloWorld();” line tells the scripting engine to “go perform the function named HelloWorld.” When that line gets executed, you would see a welcome message with the text “Hello World” in it.

Function Arguments

Let’s take this a bit further and tell the message box which text to display by adding an argument to the function.

```
function HelloWorld(Message)
    Dialog.Message("Welcome", Message);
end

HelloWorld("This is an argument");
```

Now the message box shows the text that was “passed” to the function.

In the function definition, “Message” is a variable that will automatically receive whatever argument is passed to the function. In the function call, we pass the string “This is an argument” as the first (and only) argument for the HelloWorld function.

Here is an example of using multiple arguments.

```
function HelloWorld(Title, Message)
    Dialog.Message(Title, Message);
end

HelloWorld("This is argument one", "This is argument two");
HelloWorld("Welcome", "Hi there");
```

This time, the function definition uses two variables, one for each of its two arguments...and each function call passes two strings to the HelloWorld function.

Note that by changing the content of those strings, you can send different arguments to the function, and achieve different results.

Returning Values

The next step is to make the function return values back to the calling script. Here is a function that accepts a number as its single argument, and then returns a string containing all of the numbers from one to that number.

```
function Count(n)

    -- start out with a blank return string
    ReturnString = "";

    for num = 1,n do
        -- add the current number (num) to the end of the return string
        ReturnString = ReturnString..num;

        -- if this isn't the last number, then add a comma and a space
        -- to separate the numbers a bit in the return string
        if (num ~= n) then
            ReturnString = ReturnString..", ";
        end
    end

    -- return the string that we built
    return ReturnString;
end

CountString = Count(10);
Dialog.Message("Count", CountString);
```

The last two lines of the above script uses the Count function to build a string counting from 1 to 10, stores it in a variable named CountString, and then displays the contents of that variable in a dialog message box.

Returning Multiple Values

You can return multiple values from functions as well:

```
function SortNumbers(Number1, Number2)
    if Number1 <= Number2 then
        return Number1, Number2
    else
        return Number2, Number1
    end
end

firstNum, secondNum = SortNumbers(102, 100);
Dialog.Message("Sorted", firstNum .. ", " .. secondNum);
```

The above script creates a function called SortNumbers that takes two arguments and then returns two values. The first value returned is the smaller number, and the second value returned is the larger one. Note that we specified two variables to receive the return values from the function call on the second last line. The last line of the script displays the two numbers in the order they were sorted into by the function.

Redefining Functions

Another interesting thing about functions is that you can override a previous function definition simply by re-defining it.

```
function HelloWorld()
    Dialog.Message("Message","Hello World");
end

function HelloWorld()
    Dialog.Message("Message","Hello Earth");
end

HelloWorld();
```

The script above shows a message box that says “Hello Earth,” and not “Hello World.” That is because the second version of the HelloWorld() function overrides the first one.

Putting Functions in Tables

One really powerful thing about tables is that they can be used to hold functions as well as other values. This is significant because it allows you to make sure that your functions have unique names and are logically grouped. (This is how all of the Visual Patch functions are implemented.) Here is an example:

```
-- Make the functions:
function HelloEarth()
    Dialog.Message("Message", "Hello Earth");
end

function HelloMoon()
    Dialog.Message("Message", "Hello Moon");
end

-- Define an empty table:
Hello = {};

-- Assign the functions to the table:
Hello.Earth = HelloEarth;
Hello.Moon = HelloMoon;

-- Now call the functions:
Hello.Earth();
Hello.Moon();
```

It is also interesting to note that you can define functions right in your table definition:

```
Hello = {
Earth = function () Dialog.Message("Message", "Hello Earth") end,
 Moon = function () Dialog.Message("Message", "Hello Moon") end
};

-- Now call the functions:
Hello.Earth();
Hello.Moon();
```

String Manipulation

In this section we will briefly cover some of the most common string manipulation techniques, such as string concatenation and comparisons.

(For more information on the string functions available to you in Visual Patch, see *Program Reference / Actions / String* in the online help.)

Concatenating Strings

We have already covered string concatenation, but it is well worth repeating. The string concatenation operator is two periods in a row (..). For example:

```
FullName = "Bo".." Derek"; -- FullName is now "Bo Derek"

-- You can also concatenate numbers into strings
DaysInYear = 365;
YearString = "There are "..DaysInYear.." days in a year.";
```

Note that you can put spaces on either side of the dots, or on one side, or not put any spaces at all. For example, the following four lines will accomplish the same thing:

```
foo = "Hello " .. user_name;
foo = "Hello ".. user_name;
foo = "Hello " ..user_name;
foo = "Hello "..user_name;
```

Comparing Strings

Next to concatenation, one of the most common things you will want to do with strings is compare one string to another. Depending on what constitutes a “match,” this can either be very simple, or just a bit tricky.

If you want to perform a case-sensitive comparison, then all you have to do is use the equals operator (==).

For example:

```
strOne = "Strongbad";
strTwo = "Strongbad";

if strOne == strTwo then
    Dialog.Message("Guess what?", "The two strings are equal!");
else
    Dialog.Message("Hmmm", "The two strings are different.");
end
```

Since the == operator performs a case-sensitive comparison when applied to strings, the above script will display a message box proclaiming that the two strings are equal.

If you want to perform a case-*insensitive* comparison, then you need to take advantage of either the String.Upper or String.Lower function, to ensure that both strings have the same case before you compare them. The String.Upper function returns an all-uppercase version of the string it is given, and the String.Lower function returns an all-lowercase version. Note that it doesn't matter which function you use in your comparison, so long as you use the same function on both sides of the == operator in your if statement.

For example:

```
strOne = "Mooohahahaha";
strTwo = "MOOohaHAHAha";

if String.Upper(strOne) == String.Upper(strTwo) then
    Dialog.Message("Guess what?", "The two strings are equal!");
else
    Dialog.Message("Hmmm", "The two strings are different.");
end
```

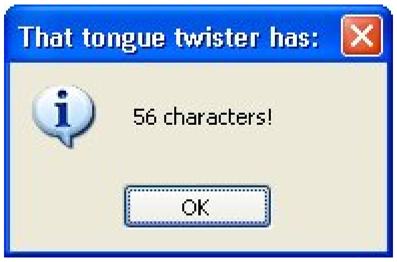
In the example above, the String.Upper function converts strOne to "MOOOHAHAHAHA" and strTwo to "MOOOHAHAHAHA" and then the if statement compares the results. (Note: the two original strings remain unchanged.) That way, it doesn't matter what case the original strings had; all that matters is whether the letters are the same.

Counting Characters

If you ever want to know how long a string is, you can easily count the number of characters it contains. Just use the String.Length function, like so:

```
twister = "If a wood chuck could chuck wood, how much would...um...";
num_chars = String.Length(twister);
Dialog.Message("That tongue twister has:", num_chars .. " characters!");
```

...which would produce the following dialog message:


That tongue twister has:

56 characters!

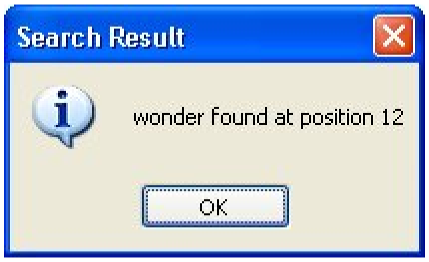
Finding Strings:

Another common thing you'll want to do with strings is to search for one string within another. This is very simple to do using the String.Find action.

For example:

```
strSearchIn = "Isn't it a wonderful day outside?";
strSearchFor = "wonder";

-- search for strSearchIn inside strSearchFor
nFoundPos = String.Find(strSearchIn, strSearchFor);

if nFoundPos ~= nil then
    -- found it!
    Dialog.Message("Search Result", strSearchFor ..
                   " found at position " .. nFoundPos);
else
    -- no luck
    Dialog.Message("Search Result", strSearchFor.." not found!");
end
```

...would cause the following message to be displayed:


Search Result

wonder found at position 12

Tip: Try experimenting with different values for strSearchFor and strSearchIn.

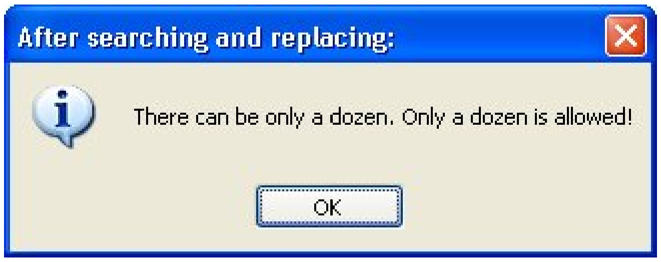
Replacing Strings:

One of the most powerful things you can do with strings is to perform a search and replace operation on them.

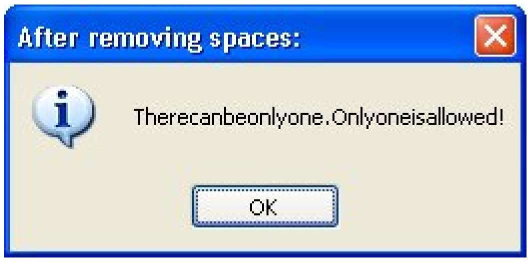
The following example shows how you can use the String.Replace action to replace every occurrence of a string with another inside a target string:

```
strTarget      = "There can be only one. Only one is allowed!";
strSearchFor   = "one";
strReplaceWith = "a dozen";
strNewString   = String.Replace( strTarget
                               , strSearchFor
                               , strReplaceWith );

Dialog.Message("After searching and replacing:", strNewString);

-- create a copy of the target string with no spaces in it
strNoSpaces = String.Replace(strTarget, " ", "");

Dialog.Message("After removing spaces:", strNoSpaces);
```

The above example would display the following two messages:


After searching and replacing:

There can be only a dozen. Only a dozen is allowed!


After removing spaces:

Therecanbeonlyone.Onlyoneisallowed!

Extracting Strings

There are three string functions that allow you to “extract” a portion of a string, rather than copying the entire string itself. These functions are String.Left, String.Right, and String.Mid.

String.Left copies a number of characters from the beginning of the string. String.Right does the same, but counting from the right end of the string instead. String.Mid allows you to copy a number of characters starting from any position in the string.

You can use these functions to perform all kinds of advanced operations on strings.

Here’s a basic example showing how they work:

```
strOriginal = "It really is good to see you again.";

-- copy the first 13 characters into strLeft
strLeft = String.Left(strOriginal, 13);

-- copy the last 18 characters into strRight
strRight = String.Right(strOriginal, 18);
```

```
-- create a new string with the two pieces
strNeo = String.Left .. "awesome" .. strRight .. " Whoa.";

-- copy the word "good" into strMiddle
strMiddle = String.Mid(strOriginal, 13, 4);
```

Converting Numeric Strings into Numbers

There may be times when you have a numeric string, and you need to convert it to a number.

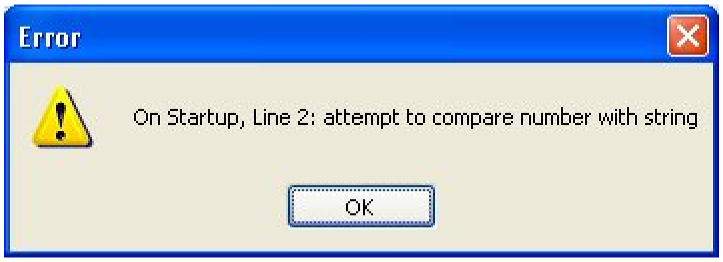
For example, if you have an input field where the user can enter their age, and you read in the text that they typed, you might get a value like “31”. Because they typed it in, though, this value is actually a string consisting of the characters “3” and “1”.

If you tried to compare this value to a number, you would get a syntax error saying that you attempted to compare a number with a string.

For example, the following script (when placed in the On Startup event):

```
age = "31";
if age > 18 then
    Dialog.Message("", "You're older than 18.");
end
```

...would produce the following error message:



```
On Startup, Line 2: attempt to compare number with string
```

The problem in this case is the line that compares the contents of the variable “age” with the number 18:

```
if age > 18 then
```

This generates an error because age contains a string, and not a number. The script engine doesn’t allow you to compare numbers with strings in this way. It has no way of knowing whether you wanted to treat age as a number, or treat 18 as a string.

The solution is simply to convert the value of age to a number before comparing it. There are two ways to do this. One way is to use the String.ToNumber function.

The String.ToNumber function translates a numeric string into the equivalent number, so it can be used in a numeric comparison.

```
age = "31";
if String.ToNumber(age) > 18 then
    Dialog.Message("", "You're older than 18.");
end
```

The other way takes advantage of the scripting engine’s ability to convert numbers into strings when it knows what your intentions are. For example, if you’re performing an arithmetic operation (such as adding two numbers), the engine will automatically convert any numeric strings to numbers for you:

```
age = "26" + 5; -- result is a numeric value
```

The above example would not generate any errors, because the scripting engine understands that the only way the statement makes sense is if you meant to use the numeric string as a number. As a result, the engine automatically converts the numeric string to a number so it can perform the calculation.

Knowing this, we can convert a numeric string to a number without changing its value by simply adding 0 to it, like so:

```
age = "31";
if (age + 0) > 18 then
    Dialog.Message("", "You're older than 18.");
end
```

In the preceding example, adding zero to the variable gets the engine to convert the value to a number, and the result is then compared with 18. No more error.

Other Built-in Functions

Script Functions

There are three other built-in functions that may prove useful to you: dofile, require, and type.

dofile

Loads and executes a script file. The contents of the file will be executed as though it was typed directly into the script. The syntax is:

dofile(*file_path*);

For example, say we typed the following script into a file called MyScript.lua (just a text file containing this script, created with notepad or some other text editor):

```
Dialog.Message("Hello", "World");
```

Now we include the file as a primer file in our Visual Patch installer. Wherever the following line of script is added:

```
dofile(SessionVar.Expand("%TempLaunchFolder%\\MyScript.lua"));
```

...that script file will be read in and executed immediately. In this case, you would see a message box with the friendly “hello world” message.

Tip: Use the dofile function to save yourself from having to re-type or re-paste a script into your projects over and over again.

require

Loads and runs a script file into the scripting engine. It is similar to dofile except that it will only load a given file once per session, whereas dofile will re-load and re-run the file each time it is used. The syntax is:

require(*file_path*);

So, for example, even if you do two requires in a row:

```
require("%AppFolder%\\foo.lua");
require("%AppFolder%\\foo.lua"); -- this line won't do anything
```

...only the first one will ever get executed. After that, the scripting engine knows that the file has been loaded and run, and future calls to require that file will have no effect.

Since require will only load a given script file once per session, it is best suited for loading scripts that contain only variables and functions. Since variables and functions are global by default, you only need to "load" them once; repeatedly loading the same function definition would just be a waste of time.

This makes the require function a great way to load external script libraries. Every script that needs a function from an external file can safely require() it, and the file will only actually be loaded the first time it's needed.

type

This function will tell you the type of value contained in a variable. It returns the string name of the variable type. Valid return values are "nil," "number," "string," "boolean," "table," or "function." For example:

```
a = 989;
strType = type(a); -- sets strType to "number"

a = "Hi there";
strType = type(a); -- sets strType to "string"
```

The type function is especially useful when writing your own functions that need certain data types in order to operate. For example, the following function uses type() to make sure that both of its arguments are numbers:

```
-- find the maximum of two numbers
function Max(Number1, Number2)
    -- make sure both arguments are numeric
    if (type(Number1) ~= "number") or (type(Number2) ~= "number") then
        Dialog.Message("Error", "Please enter numbers");
        return nil; -- we're using nil to indicate an error condition
    else
        if Number1 >= Number2 then
            return Number1;
        else
            return Number2;
        end
    end
end
```

Actions

Visual Patch comes with a large number of built-in functions. In the program interface, these built-in functions are commonly referred to as *actions*. For scripting purposes, actions and functions are essentially the same; however, the term “actions” is generally reserved for those functions that are built into the program and are included in the alphabetical list of actions in the online help. When referring to functions that have been created by other users or yourself, the term “functions” is preferred.

Debugging Your Scripts

Scripting (or any kind of programming) is relatively easy once you get used to it. However, even the best programmers make mistakes, and need to iron the occasional wrinkle out of their code. Being good at debugging scripts will reduce the time to market for your projects and increase the amount of sleep you get at night. Please read this section for tips on using Visual Patch as smartly and effectively as possible!

This section will explain Visual Patch’s error handling methods as well as cover a number of debugging techniques.

Error Handling

All of the built-in Visual Patch actions use the same basic error handling techniques. However, this is not necessarily true of any third-party functions, modules, or scripts—even scripts developed by Indigo Rose Corporation that are not built into the product. Although these externally developed scripts can certainly make use of Visual Patch's error handling system, they may not necessarily do so. Therefore, you should always consult a script or module's author or documentation in order to find out how error handling is, well, handled.

There are two kinds of errors that you can have in your scripts when calling Visual Patch actions: syntax errors, and functional errors.

Syntax Errors

Syntax errors occur when the syntax (or “grammar”) of a script is incorrect, or a function receives arguments that are not appropriate. Some syntax errors are caught by Visual Patch when you build.

For example, consider the following script:

```
foo =
```

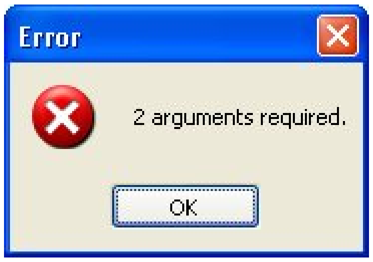
This is incorrect because we have not assigned anything to the variable foo—the script is incomplete. This is a pretty obvious syntax error, and would be caught by the scripting engine at build time (when you build your project).

Another type of syntax error is when you do not pass the correct type or number of arguments to a function. For example, if you try and run this script:

```
Dialog.Message("Hi There");
```

...the project will build fine, because there are no ***obvious*** syntax errors in the script. As far as the scripting engine can tell, the function call is well formed. The name is valid, the open and closed parentheses match, the quotes are in the right places, and there's even a terminating semi-colon at the end. Looks good!

However, at run time you would see something like the following:



2 arguments required.

Looks like it wasn't so good after all. Note that the message says two arguments are required for the Dialog.Message action. Ah. Our script only provided one argument.

According to the function prototype for Dialog.Message, it looks like the action can actually accept up to *five* arguments:

```
number Dialog.Message ( string Title,
                        string Text,
                        number Type = MB_OK,
                        number Icon = MB_ICONNONE,
                        number DefaultButton = MB_DEFBUTTON1 )
```

Looking closely at the function prototype, we see that the last three arguments have default values that will be used if those arguments are omitted from the function call. The first two arguments—Title and Text—don't have default values, so they cannot be omitted without generating an error. To make a long story short, it's okay to call the Dialog.Message action with anywhere from 2 to 5 arguments...but 1 argument isn't enough.

Fortunately, syntax errors like these are usually caught at build time or when you test your installer. The error messages are usually quite clear, making it easy for you to locate and identify the problem.

Functional Errors

Functional errors are those that occur because the functionality of the action itself fails. They occur when an action is given incorrect information, such as the path to a file that doesn't exist. For example, the following code will produce a functional error:

```
filecontents = TextFile.ReadToString("this_file_don't exist.txt");
```

If you put that script into an event right now and try it, you will see that nothing appears to happen. This is because Visual Patch's functional errors are not automatically displayed the way syntax errors are. We leave it up to you to handle (or to not handle) such functional errors yourself.

The reason for this is that there may be times when you don't care if a function fails. In fact, you may expect it to. For example, the following code tries to remove a folder called C:\My Temp Folder:

```
Folder.Delete("C:\\My Temp Folder");
```

However, in this case you don't care if it really gets deleted, or if the folder didn't exist in the first place. You just want to make sure that if that particular folder exists, it will be removed. If the folder isn't there, the Folder.Delete action causes a functional error, because it can't find the folder you told it to delete...but since the end result is exactly what you wanted, you don't need to do anything about it. And you certainly don't want the user to see any error messages.

Conversely, there may be times when it is very important for you to know if an action fails. Say for instance that you want to copy a very important file:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");
```

In this case, you really want to know if it fails and may even want to exit the program or inform the user. This is where the Debug actions come in handy. Read on.

Debug Actions

Visual Patch comes with some very useful functions for debugging your patches. This section will look at a number of them.

Application.GetLastError

This is the most important action to use when trying to find out if a problem has occurred. At run time there is always an internal value that stores the status of the last action that was executed. At the start of an action, this value is set to 0 (the number zero). This means that everything is OK. If a functional error occurs inside the action, the value is changed to some non-zero value instead.

This last error value can be accessed at any time by using the Application.GetLastError action.

The syntax is:

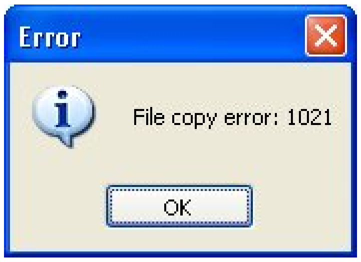
last_error_code = Application.GetLastError();

Here is an example that uses this action:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

error_code = Application.GetLastError();
if (error_code ~= 0) then
    -- some kind of error has occurred!
    Dialog.Message("Error", "File copy error: "..error_code);
    Application.Exit();
end
```

The above script will inform the user that an error occurred and then exit the patch. This is not necessarily how all errors should be handled, but it illustrates the point. You can do anything you want when an error occurs, like calling a different function or anything else you can dream up.

The above script has one possible problem. Imagine the user seeing a message like this:


Error

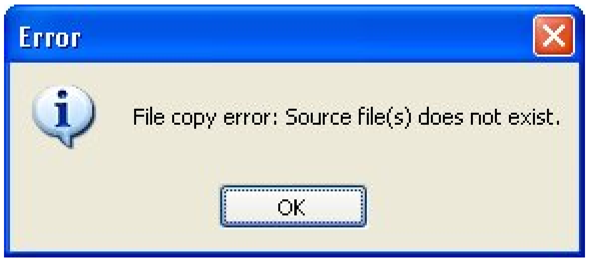
File copy error: 1021

It would be much nicer to actually tell them some information about the exact problem. Well, you are in luck! At run time there is a table called _tblErrorMessages that contains all of the possible error messages, indexed by the error codes. You can easily use the last error number to get an actual error message that will make more sense to the user than a number like “1021.”

For example, here is a modified script to show the actual error string:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

error_code = Application.GetLastError();

if (error_code ~= 0) then

    -- some kind of error has occurred!
    Dialog.Message( "Error", "File copy error: "
                    .. _tblErrorMessages[error_code] );

    Application.Exit();

end
```

Now the script will produce the following error message:


Error

File copy error: Source file(s) does not exist.

Much better information!

Just remember that the value of the last error gets reset every time an action is executed. For example, the following script would not produce an error message:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

-- At this point Application.GetLastError() could be non-zero, but...

Dialog.Message("Hi There", "Hello World");

-- Oops, now the last error number will be for the Dialog.Message action,
-- and not the File.Copy action. The Dialog.Message action will succeed,
-- resetting the last error number to 0, and the following lines will not
-- catch any error that happened in the File.Copy action.

error_code = Application.GetLastError();

if (error_code ~= 0) then

    -- some kind of error has occurred!
    Dialog.Message( "Error", "File copy error: "
                    .. _tblErrorMessages[error_code] );

    Application.Exit();

end
```

Debug.ShowWindow

The Visual Patch runtime has the ability to show a debug window that can be used to display debug messages. This window exists throughout the execution of your patch, but is only visible when you tell it to be.

The syntax is:

Debug.ShowWindow(*show_window*);

...where *show_window* is a Boolean value. If true, the debug window is displayed, if false, the window is hidden. For example:

```
-- show the debug window
Debug.ShowWindow(true);
```

If you call this script, the debug window will appear on top of your patch, but nothing else will really happen. That’s where the following Debug actions come in.

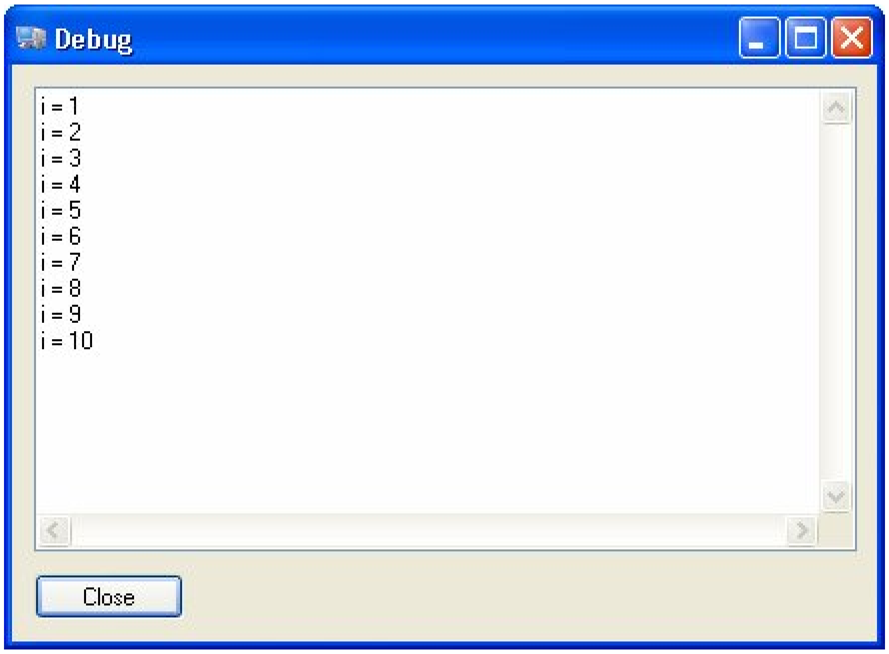
Debug.Print

This action prints the text of your choosing in the debug window. For example, try the following script:

```
Debug.ShowWindow(true);

for i = 1, 10 do
    Debug.Print("i = " .. i .. "\r\n");
end
```

The "\r\n" part is actually two escape sequences that are being used to start a new line. (This is technically called a "carriage return/linefeed" pair.) You can use \r\n in the debug window whenever you want to insert a new line.

The above script will produce the following output in the debug window:



```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

You can use this method to print all kinds of information to the debug window. Some typical uses are to print the contents of a variable so you can see what it contains at run time, or to print your own debug messages like "inside outer for loop" or "foo() function started." Such messages form a trail like bread crumbs that you can trace in order to understand what’s happening behind the scenes in your project. They can be invaluable when trying to debug your scripts or test your latest algorithm.

Debug.SetTraceMode

Visual Patch can run in a special “trace” mode at run time that will print information about every line of script that gets executed to the debug window, including the value of Application.GetLastError() if the line involves calling a built-in action. You can turn this trace mode on or off by using the Debug.SetTraceMode action:

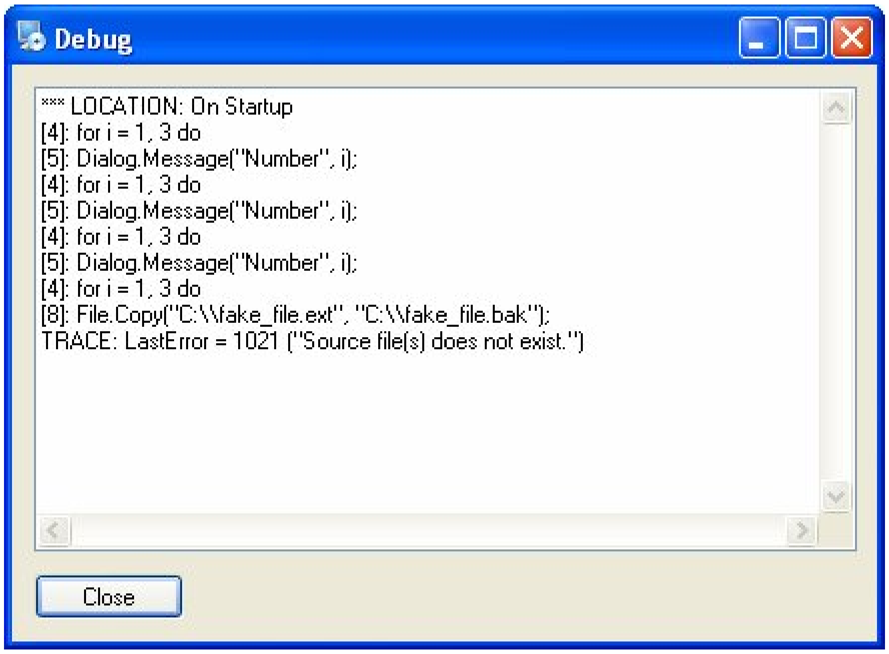
Debug.SetTraceMode(*turn_on*);

...where *turn_on* is a Boolean value that tells the program whether to turn the trace mode on or off.

Here is an example:

```
Debug.ShowWindow(true);
Debug.SetTraceMode(true);

for i = 1, 3 do
    Dialog.Message("Number", i);
end

File.Copy("C:\\fake_file.ext", "C:\\fake_file.bak");
```

Running that script will produce the following output in the debug window:



```
*** LOCATION: On Startup
[4]: for i = 1, 3 do
[5]: Dialog.Message("Number", i);
[4]: for i = 1, 3 do
[5]: Dialog.Message("Number", i);
[4]: for i = 1, 3 do
[5]: Dialog.Message("Number", i);
[4]: for i = 1, 3 do
[8]: File.Copy("C:\\fake_file.ext", "C:\\fake_file.bak");
TRACE: LastError = 1021 ("Source file(s) does not exist.")
```

Notice that every line produced by the trace mode starts with "TRACE:" This is so you can tell them apart from any lines you send to the debug window with Debug.Print. The number after the "TRACE:" part is the line number that is currently being executed in the script.

Turning trace mode on is something that you will not likely want to do in your final, distributable patch, but it can really help find problems during development.

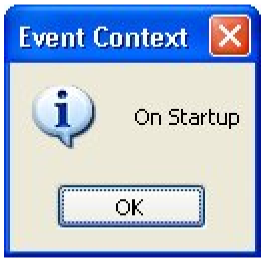
Debug.GetEventContext

The Debug.GetEventContext action is used to get a descriptive string about the event that is currently being executed. This can be useful if you define a function in one place but call it somewhere else, and you want to be able to tell where the function is being called from at any given time.

For example, if you execute this script from the On Startup event:

```
Dialog.Message("Event Context", Debug.GetEventContext());
```

...you will see something like this:



Dialog.Message

This brings us to good ole' Dialog.Message. You have seen this action used throughout this document, and for good reason. This is a great action to use throughout your code when you are trying to track down a problem.

For example, you can use it to display the current contents of a variable that you're working with:

```
Dialog.Message("The current value of nCats is: " .. nCats);
```

You can also use it to put up messages at specific points in a script, to break it into arbitrary stages. This can be helpful when you're not sure where in a script an error is occurring:

```
function foobar(arg1, arg2)

    Dialog.Message("Temporary Debug Msg", "In foobar()");

    -- bunch of script

    Dialog.Message("Temporary Debug Msg", "1");

    -- bunch of script

    Dialog.Message("Temporary Debug Msg", "2");

    -- bunch of script

    Dialog.Message("Temporary Debug Msg", "Leaving foobar()");

end
```

Final Thoughts

Hopefully this chapter has helped you to understand scripting in Visual Patch. Once you get the hang of it, it is a really fun, powerful way to get things done.

Other Resources

Here is a list of other places that you can go for help with scripting in Visual Patch.

Help File

The Visual Patch help file is packed with good reference material for all of the actions and events supported by Visual Patch, and for the design environment itself. You can access the help file at any time by choosing Help > Visual Patch Help from the menu.

Tip: If you are in the action editor and you want to learn more about an action, simply click on the action and press the F1 key on your keyboard.

Visual Patch Web Site

The Visual Patch web site is located at http://www.visualpatch.com. Be sure to check out the user forums where you can read questions and answers by fellow users and Indigo Rose staff as well as ask questions of your own.

Tip: A quick way to access the online forums is to choose Help > User Forums from the menu.

Indigo Rose Technical Support

If you need help with any scripting concepts or have a mental block to push through, feel free to open a support ticket at http://support.indigorose.com. Although we can't write scripts for you or debug your specific scripts, we will be happy to answer any general scripting questions that you have.

The Lua Web Site

Visual Patch's scripting engine is based on a popular scripting language called *Lua*. Lua is designed and implemented by a team at Tecgraf, the Computer Graphics Technology Group of PUC-Rio (the Pontifical Catholic University of Rio de Janeiro in Brazil). You can learn more about Lua and its history at the official Lua web site:

http://www.lua.org

The Lua website is also where you can find the latest documentation on the Lua language, along with tutorials and a really friendly community of Lua developers.

Note that there may be other built-in functions that exist in Lua and in Visual Patch that are not officially supported in Visual Patch. These functions, if any, are documented in the Lua 5.0 Reference Manual.

Only the functions listed in the online help are supported by Indigo Rose Software. Any other “undocumented” functions that you may find in the Lua documentation are not supported. Although these functions may work, you must use them entirely on your own.

INDEX

Notes