Take Control of Setup

Click & Retrieve Source CODE!

BY MICHIEL DE BRUIJN

VB4's SetupWizard, combined with tips and tricks described here, can help you cut distribution size and create problem-free setup disks.

It's well known that the SetupWizard for VB3 was downright difficult to use at best, and at worst, too buggy to even consider. The good news is that Microsoft took note of this problem and improved the SetupWizard for VB4. The bad news is that setup is more complex for VB4 because the new release requires more files to get your users up and running for even a simple application. Setup requires additional work, such as registering OLE components.

Fortunately, the new SetupWizard is stable enough serve your basic setup needs. However, just running it fresh out of the box could result in problems for your users and require unnecessarily large distribution disks.

Use the tips and insights I'll present here to cut memory requirements and ensure hassle-free setup, while taking a look "under the hood" to learn how the new SetupWizard works.

The new SetupWizard comes in 16- and 32-bit editions. Like most parts of VB4, the 32-bit version contains the most features, such as the ability to remove installed apps later. But the 16-bit edition of SetupWizard comes with some unique goodies of its own.

For example, you'll never send the wrong system files to your users again because the 16-bit SetupWizard comes with a copy of the most recently shared system files, and it writes those files to your setup disks instead of writing to versions in your Windows system directory (most likely 32-bit). Before going into detailed tips and tricks for creating customized setup systems, I'll walk you through the new setup process step by step.

Like the Wizards you're used to in Microsoft products, SetupWizard guides you through a complicated process (in this case, the process of building distribution disks), in a number of easy steps. Here's the basic sequence.

First, SetupWizard asks about the location of your project (MAK or VBP) file. If you check the "Rebuild the Project's EXE file" box, and the EXE is missing or of the wrong type, the SetupWizard will shell out to VB and automatically compile the EXE for you. It will also scan all files in the project for dependencies on external files, such as OCXs, DLLs, or OLE servers.

Next, if the SetupWizard detects that

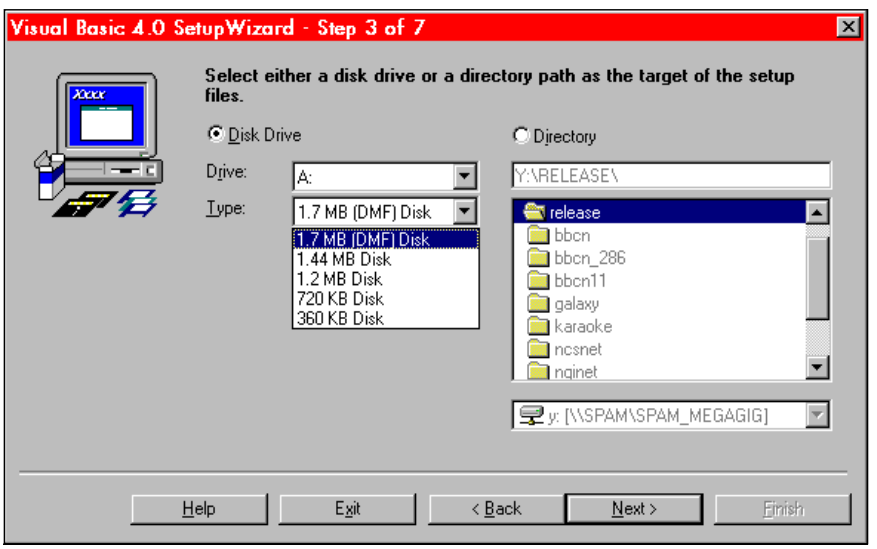

FIGURE 1 ***Adding Disk Options.*** *You can add alternative disk sizes to the choices offered by the SetupWizard, such as the DMF format shown here, by modifying the [SetupWiz] or [SetupWiz-32] section of the SWDEPEND.INI file. As a bonus, you can also prevent the 32-bit SetupWizard from coming up with 1.2 MB disks as the default by re-ordering some entries.*

Michiel lives in Rotterdam, The Netherlands, and is co-owner of VBD Automatiseringsdiensten, where he is the designer and lead programmer of the Network Communications System, a Windows-based messaging system written almost entirely in VB. He is a section leader in the VBPJ *CompuServe Forum, and is active on the Microsoft Basic Forum. For his online efforts, he has received a Microsoft Most Valuable Professional award. Contact Michiel on CompuServe at mdbruijn or on the Internet at mdb@vbd.nl.*

you're using Data Access features, it lets you specify external ISAM drivers. Otherwise it skips to step three, where you can specify the target of your distribution files. This can be a floppy disk or a regular path: the latter is handy for network or CD-ROM distributions.

During step four, the SetupWizard allows you to specify OLE servers in addition to the ones it detected while scanning your project files. This is necessary because it's possible to create indirect references to OLE servers in your code. In step five, you can specify other components, such as DLLs and OCXs.

It's also possible to remove files during this phase if you spot a component that you don't want to send out or for which you don't have the redistribution rights.

In step six, the SetupWizard wants to know a few details about your application: whether it's installed in its own directory (as will be the case with most programs) or should be handled as an OLE component, in which case it'll end up in a shared-system directory. Also, Enterprise Edition users need to indicate here if the distribution contains any parts to be installed as a remote OLE server.

When these steps are completed the SetupWizard will process all data it gathered and present you with a final list of all files you need to send out to your users. You can view or modify the default file placement by selecting a file from the list and clicking on File details, but in most cases the SetupWizard will be correct about such details.

You can now build the disks by selecting Finish, though it might be a good idea to use Save Template first if you plan on using this set of files more than once.

Now you have complete setup disks that are guaranteed to install all the required files on your user's systems, assuming you didn't forget application-specific data files. The only exception is when you need to include 16-bit ODBC components, because these have a separate, self-contained setup disk (found on your VBCD in the \VB\ODBC directory).

Though the VB manual states that the 16-bit Remote Procedure Call (RPC) for OLE Remote Automation components also has a separate setup program, these files will in fact be added by the SetupWizard like regular components and no additional steps are needed.

ON TARGET

When you build your distribution disks for the first time, you'll probably think that something has gone wrong. After all, two disks with more than 20 files, just to distribute your 16-bit "Hello World" test program must be a mistake, right?

16-Bit VB4 Runtime Files		
Actual Application	YOURAPP.EXE	Main application executable
	THREED16.OCX	Custom control
Setup Application	CTL13DV2.DLL	3- D library for SETUP.EXE/OCX runtime
	SETUP.EXE	First-stage setup program
	SETUP.LST	Setup configuration file
	SETUP1.EXE	User setup program
	STKIT416.DLL	Setup kit support routines
VB Runtime	VB40016.DLL	Main VB runtime ("VBRUN400")
	VAEN21.OLB	VBA 2.1 object library
OLE Runtime	COMPOBJ.DLL	Component Object Model support
	OC25.DLL	Runtime for OLE custom controls (OCXs)
	OLE2.DLL	Main OLE libraries
	OLE2.REG	Registry information for OLE libraries
	OLE2CONV.DLL	Conversion for graphical OLE output
	OLE2DISP.DLL	Main OLE Automation support
	OLE2NLS.DLL	International OLE support
	OLE2PROX.DLL	OLE proxy support
	SCP.DLL	Code page translation library
	STDOLE.TLB	Standard OLE type library
	STORAGE.DLL	Support for structured storage
	TYPLIB.DLL	OLE Automation type library support
	VSHARE.386	Sharing support (required for OLE file I/O)
32-Bit VB4 Runtime Files		
Actual application	YOURAPP.EXE	Main application executable
	THREED32.OCX	Custom control
Setup application	CTL3D32.DLL	3D library for SETUP.EXE
	SETUP.EXE	First-stage setup program
	SETUP.LST	Setup configuration file
	SETUP132.EXE	User setup program
	ST4UNST.EXE	Uninstall support program
	STKIT432.DLL	Setup kit support routines
VB Runtime	VB40032.DLL	Main VB runtime
	VEN2232.OLB	VBA 2.2 object library
Various	MFC40.DLL	Microsoft Foundation Classes runtime
	MSVCRT40.DLL	Microsoft Visual C++ 4.0 runtime
	MSVCRT.DLL	Microsoft Visual C++ 2.x runtime
	OLEPRO32.DLL	Extensions to OLE libraries

TABLE 1 ***OLE Means Lots of Runtime Files.*** *The size of the VB4 runtime files is significantly larger than in previous versions. The biggest increase in size on 16-bit platforms (top) is caused by the inclusion of the OLE libraries. But even in 32-bit distributions (bottom), where OLE is guaranteed to be a part of the target system already, the MFC and C++ runtimes (required by VB and most OCXs) take up a lot of space.*

Unfortunately, the SetupWizard is right on target with these numbers. For the benefits of component-based OLE programming we pay a price in memory because Visual Basic 4.0 itself is entirely OLE based, which means that you need the OLE library DLLs for even the most trivial VB program, whether or not it has OLE features.

If you're developing for Windows 3.1 using Visual Basic 4.0, you pay a price because VB4 needs more recent versions of the OLE libraries than those that Microsoft ships with Windows 3.1. Keep in mind that you'll always need at least two disks for Windows 3.1 applications.

And even though simple 32-bit applications designed to run on Windows 95 or Windows NT 3.51 (both systems already contain up-to-date OLE code) initially will fit on one disk, the size of the runtime is so large that for most real-world applications you'll need two disks for these platforms.

Even a bare-bones Visual Basic 4.0 application that uses one or more custom controls (either VBXs or OCXs) requires a lot of files (see Table 1).

What can you do to cope with expanding memory requirements, except to switch to CD-ROM distribution? The simplest action you can take, and one that works regardless of the setup program used, is to select References from VB's Tools menu while your project is loaded, and check for any unneeded type-library

references. For example, if you include a reference to the Data Access Object (DAO) library, VB will expect that functionality to be present at run time and you'll have to distribute the extra DLLs. If you don't need it, omitting the DAO reference will cut disk requirements.

You can further slash disk requirements by using data compression that's superior to Microsoft's. Though the algorithm employed by the COMPRESS.EXE program (as used by the SetupWizard) is quite effective, other utilities may yield much better results.

The solution Microsoft uses for their own products, the so-called Distribution Media Format (DMF), combines improved data compression and an alternative disk formatting process that allows as much as 1.7 MB of uncompressed data on a 3.5-inch high-density disk. Also, DMF provides protection against software piracy because DMF disks are harder to copy.

Although Microsoft doesn't make DMF available to third parties, utilities for formatting your own DMF-style disks are starting to become available (a program that's worth checking out is WinImage 2.20, available as WIMA9522.ZIP from various places.

Because both Windows 95 and Windows NT support the DMF disk format, it should work fine for distributing 32-bit applications.

For 16-bit platforms, you need special code or a TSR/device driver that loads before Windows, in order to read the disks, which makes DMF a lot less attractive there. Still, if you discover that DMF significantly cuts your disk expense, it's worth implementing.

YOU CAN SLASH DISK REQUIREMENTS BY USING DATA COMPRESSION THAT IS SUPERIOR TO MICROSOFT'S.

Changing the data compression algorithm means saying goodbye to SetupWizard as your one-stop setup solution. If you choose this approach, you'll have to build your setup disks manually because the SetupWizard is hardwired to use the same LZW compression code.

However, you could still use SetupWizard to generate the initial file listing. If you switch to DMF disks, you can continue to use the SetupWizard, provided you make a small change to its configuration file. If you've executed both the 16- and 32-bit editions of the SetupWizard at least once, you'll find a file called SWDEPEND.INI in your Windows directory, that contains [SetupWiz] and [SetupWiz-32] sections (these don't necessarily have to be near each other).

In these sections you'll find a number of Drive*n* keys, where *n* is the sequence number in the SetupWizard drive-selection combo box. By adding one or more entries here (renumbering the other lines, if required), you can easily add new disk formats as they become available.

For example, to add support for the 1.7 MB DMF-style format as created by the WinImage utility, add:

```
Drive1=1.7 MB (DMF) Disk,1716224,1024
```

The first part of this string is the textual description of the new format, followed by the number of free bytes on a freshly formatted disk.

The last part of the string is the number of bytes per sector: SetupWizard needs this information in order to do accurate free-space calculations. Even if you don't need to add custom disk formats, customizing SWDEPEND.INI lets you swap the values for the Drive1 and Drive2 lines in the [SetupWiz-

Uninstalling Applications

Uninstalling your application from a user's system is one of the requirements to qualify for the "Designed for Windows 95" logo.

This feature is in huge demand by users because this process is feasible only on Windows 95 (which was designed with a solution to the uninstall problem in mind) and Windows NT, only 32-bit SetupWizard-generated setup programs support it. Hence, for 32-bit apps you can create an uninstall program rather than buy one.

To enable uninstallation, the setup program adds a line to a file called ST4UNST.LOG (initially stored in the Windows directory and moved to the application directory after a successful installation) for each action it performs, such as copying a new file onto the system, replacing an existing file, adding or changing a Registry or INI key, adding or replacing a shell link or Program Manager item, and more.

The file is human-readable, and can also be used to diagnose setup problems (if the installation is aborted, SETUP1 will ask the user whether to keep the file or not).

The actual uninstallation is performed by a Microsoft program called ST4UNST.EXE that resides in the Windows directory. This program essentially reverses the actions logged in ST4UNST.EXE, which sounds simple, but can get quite complicated in practice.

The problem is with shared files: if your application installed a copy of a popular file (say, THREED32.OCX), it can't just delete it when the user decides to uninstall your application, because a new program, installed in the meantime, may also rely on THREED32.OCX. Deleting the file would render that application useless.

To prevent such problems and still allow effective uninstallation, Microsoft built support for this process into the operating system. Any Windows 95-compatible setup program is supposed to create or increase a usage counter for the shared files it installs in a named value under the Registry key \HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs. The uninstall program will decrease the usage counter again for each file it wants to remove, and delete it only if the counter reaches zero.

This scheme works if all programs follow this guideline, but in practice it can lead to critical files being deleted due to non-compliant setup apps and files required by 16-bit applications that don't know how to access Registry keys in the first place.

Still, ST4UNST.EXE is an easy way to offer uninstall features to your customers: like the rest of the VB4 Setup code and the SetupWizard itself, it's a very usable solution. Even if you're content with your current third-party or home-grown setup solution, you might want to take a good look at it. —M.dB.

32] section. Hence, you can make 1.44 MB floppies the default for the 32-bit SetupWizard instead of the pesky 1.2 MB disks (anyone remember those?) it insists on each time by default (see Figure 1).

GATHERING THE FILES

OK, back to the actual disk-building process. One question you might have is, "What does the SetupWizard know that I don't?" After all, it usually includes all the files that an application requires to run.

The key to the SetupWizard's power is the SWDEPEND.INI file. SWDEPEND.INI is not really a VB-specific configuration file: it's used by various tools to look up component-dependency information. A typical entry in this file describes which files a single software component (for example, a certain DLL or VBX or a more complex component, like the Jet database engine) relies on in order to work (see Listing 1).

By resolving these references for every file in your project (as taken from the VBP/MAK file and any external function declarations or OLE references in your module, class and form files), the SetupWizard can build an accurate and complete file list.

However, if a certain component is not listed in SWDEPEND.INI, and the component depends on additional files (for example, an OCX that needs certain DLLs to be present), things will go wrong.

While SetupWizard includes the base component like it should, the additional files on which it depends will not be distributed to your users, and they'll probably get "File not found" errors at runtime. So adding such components to SWDEPEND.INI is a good idea if you use them on a regular basis.

The entries in an SWDEPEND.INI section tell the SetupWizard things about how the files belonging to a component should be installed (see Listing 1).

If you take a look at Listing 1, you'll see the actual SWDEPEND.INI entries for files that make up the basic VB4 32-bit runtime. Each section contains a number of Uses*n* keys, where *n* is a sequence number.

Files listed this way are called the dependencies of a certain component. According to SWDEPEND.INI (and as you've seen by examining actual files generated by the SetupWizard), the VB4/32 runtime consists of six files that have no further dependencies themselves.

The latter is not always the case: for example, it's quite common for OCX files to rely on other files, such as Microsoft Foundation Classes (MFC) DLLs. In such a case, determining dependencies for a single component becomes a recursive process, which can yield a surprising number of files.

REGISTER OR ELSE

In addition to the Uses*n* entries, the Registry key can be found in a SWDEPEND.INI section. Though its value will be rather trivial for most files—either empty or $(DLLSelfRegister)—the Registry key is related to one of the most important features a VB4 setup program should have.

VB4 and its component files (OCXs) are based entirely on OLE. Before an OLE server of any kind can function, it needs to have one or more entries in the Registry so the operating system and other components can find and activate it.

Of course, such entries can be made manually using REGEDIT, but this obviously isn't something you would want to ask an end user to do. To overcome this problem, older OLE-based applications used an REG file that was merged into the Registry by shelling out to REGEDIT. Although this works well, the solution used by VB4 (called autoregistration) is much more elegant because it doesn't rely on

external programs or files. Autoregistration minimizes the risk of an OLE component becoming unusable due to the lack of a REG file—for example, if the REG file is not copied when the component file is moved to another system.

An autoregistration-compatible OLE server includes the magic word OLESelfRegister in the version information that's part of the executable file, and implements two special functions: DllRegisterServer, and DllUnregisterServer. The DllRegisterServer function adds the information required to activate the control to the Registry: DllUnregisterServer removes it again. Any program that looks for the OLESelfRegister string and subsequently calls the registration functions, such as your setup program or the standalone REGSVR.EXE utility, can easily manage registration information without the need for external utilities.

Executable OLE servers created by VB also support autoregistration, but they use a slightly different mechanism: during installation (or at any other time), executable OLE servers can be invoked with the /REGSERVER command-line switch to update the Registry with their OLE binding information. To remove these entries again, run the server with the /UNREGSERVER switch. Components that support this kind of (un)registration are marked as $(EXESelfRegister) in SWDEPEND.INI.

Note that registration is necessary on both 16- and 32-bit platforms. Though the Registry isn't as powerful in Windows 3.1 as it is in Windows 95 and Windows NT, it's certainly there (with limitations, such as a 64K size limit) and VB4 really does uses it.

And VB4 Enterprise Edition users should keep in mind that a special kind of registration is performed for Remote OLE components created using Enterprise Edition. The actual OLE object your application calls is probably not included in the distribution because it runs on a remote server, but the Registry needs to be updated to use it.

To update the Registry regarding remote OLE objects you can include so-called VBR files in the list of OLE servers offered by the SetupWizard. A VBR file is created whenever you compile an OLE server and check the Remote Server Support Files box in the Options screen of the Make EXE option in the File menu.

The VBR file should contain the same registration information as the server itself. When the final setup program polls a VBR file, it will query the user for missing information, such as the network protocol used to talk to the remote server. The program also updates the Registry, using either the CLIREG16 or the CLIREG32 utility program.

WHERE DID MY FILES GO?

The SetupWizard suggests a default location on the target system for distribution files in addition to determining which files need to be included in the distribution.

Because the actual path for each location varies across systems and operating systems, they're represented by macro names in all files the SetupWizard uses and creates (see Table 2). You might be surprised to see that Windows 3.1 and Windows NT use the same directories (the only difference is that Windows NT has a SYSTEM32 directory for 32-bit shared system files, which is missing from 3.1 for obvious reasons).

The similarity occurs because Windows NT predates the new directory scheme designed for Windows 95, which is supposed to make it easier to maintain files and support operations such as

uninstallations. The next version of Windows NT will probably support the Windows 95 conventions.

LET SETUPWIZARD DECIDE

If you use the SetupWizard and follow standard installation procedures, you should have no problem getting the right file into the right place because the SetupWizard makes the right decision for you, courtesy of the SWDEPEND.INI file.

However, if you're writing your own installation program or using components that don't have entries in SWDEPEND.INI, you could encounter confusion handling all the choices.

Here are the most important rules for deciding where a file should go, according to the new layout ruleds introduced with Windows 95:

- Files required to start your application go into a subdirectory under the $(ProgramFiles) root, such as $(ProgramFiles)\WinWidgets. Files not directly related to program execution, such as optional modules, data files, and example files, should go in their own subdirectories, such as $(ProgramFiles)\WinWidgets\System, $(ProgramFiles)\WinWidgets\Data, and $(ProgramFiles)\WinWidgets\Samples.
- Files shared by multiple applications, like a specific VBX or OCX that is used by all applications your company produces, go into a subdirectory of the $(CommonFiles) root, such as $(CommonFiles)\AcmeCo. Conventionally, OLE server components are installed in $(CommonFiles)\OLESVR.
- Only files that need to be shared on a system-wide level, such as DLLs, VBXs, and OCXs belonging to a well-known product, go into the $(WinSysPath) directory. Never install such components in another directory—including ($WinPath)—unless you actually like causing version problems. Installing shared components in private directories may seem like a good idea at first, but private directories will cause trouble sooner or later, especially on 16-bit systems.

So far I've discussed preparation for setup, not the actual installation process. The actions performed by the SetupWizard prepare your application for installation on a target system. The task of installing files is carried out by two separate programs, SETUP.EXE and SETUP1.EXE.

As you may know, writing setup applications entirely in VB is impossible because you can't run a program that needs a runtime module to execute if the program's task is to install the runtime module in the first place. The SETUP.EXE is written in C and of course it doesn't need a runtime module to execute.

When SETUP.EXE is started, it reads the [Bootstrap] section of the SETUP.LST file (which, in spite of its LST extension is formatted like a standard INI file) to determine which files it should install to get the VB runtime and the actual setup program running. Each File*n* key in this section, where *n* is a sequence number, contains this information:

- The number of the disk the file is stored on. As I explained, the 16-bit VB4 runtime is so large it spans multiple disks; consequently, the user must change disks before the main setup program even starts.
- A flag indicating if the file is split across multiple disks: either blank or SPLIT. The SetupWizard and the setup programs provided with VB split and concatenate files, respectively. Because the SetupWizard source code is not provided, and "How do I split a file across disks?" is a frequently asked question on the various online VB forums, I've included source code for both operations (see Listing 2).
- The compressed and uncompressed names of the file.
- The target location of the file in the SWDEPEND.INI macro format you should be familiar with by now.
- The registration method for the file: either blank (no registration required), $(DLLSelfRegister), or $(EXESelfRegister).

VB4

```
[VB Runtime 0409-32]
Dest=$(WinSysPath)
Uses1=VB40032.DLL
Uses2=ven2232.olb
Uses3=olepro32.dll
Uses4=msvcrt20.dll
Uses5=msvcrt40.dll
Uses6=ctl3d32.dll
Register=
Uses7=
  [VEN2232.OLB]
Register=
Dest=$(WinSysPathSysFile)
Uses1=
  [OLEPRO32.DLL]
Dest=$(WinSysPath)
Register=$(DLLSelfRegister)
Uses1=
  [MSVCRT20.DLL]
Register=
Dest=$(WinSysPathSysFile)
Uses1=
  [MSVCRT40.DLL]
Register=
Dest=$(WinSysPath)
Uses1=
  [CTL3D32.DLL]
Register=
Dest=$(WinSysPathSysFile)
Uses1=
```

LISTING 1 ***Describing Dependencies.*** *The SWDEPEND.INI file lists all VB runtime files as well as many others—it's a system-wide resource used by many Microsoft and non-Microsoft tools. It allows SetupWizard to determine exactly which files are needed to make your program run.*

Macro String	Description
$(AppPath)	Application path as specified by the user. Constructs like $(AppPath)\SYSTEM also work.
$(ProgramFiles)	Root path for applications. Expands to *d*:\Program Files\ (by default) for Windows 95, *d*:\ for other versions. Never install files to this path. Always use $(ProgramFiles)\MyApp.
$(CommonFiles)	Root path for files that are shared by multiple applications. Expands to *d*:\Program Files\Common Files\ (by default) for Windows 95, *d*:\WINDOWS for other versions. Never install files to this path. Always use $(CommonFiles)\MyApplet.
$(CommonFilesSys)	Root path for system files shared by multiple applications. Expands to $(CommonFiles)\System.
$(WinPath)	The user's *d*:\WINDOWS directory. Avoid installing files here (use $(CommonFiles) or $(WinSysPath) instead).
$(WinSysPath)	The user's *d*:\WINDOWS\SYSTEM directory.
$(WinSysPathSysFile)	Used for 32-bit system files. Expands to *d*:\WINDOWS\SYSTEM32 on NT, $(WinSysPath) on other versions.
$(MSAppPath)	Root path for the common Microsoft applet directory. Expands to $(CommonFiles)\Microsoft on Windows 95, *d*:\WINDOWS\MSAPPS for other versions.
$(MSDAOPath)	Expands to $(MSAppPath)\DAO.

TABLE 2 ***Now Where Did That File Go?*** *The SetupWizard and the configuration files use macros to indicate file locations, because file locations differ across systems. This table shows what these sometimes cryptic strings expand to on the various platforms supported by VB4.*

• A flag indicating if the file is shared: either blank or $(Shared). For 32-bit setup programs, special operations are performed to allow uninstallation of shared files (see the accompanying sidebar, "Uninstalling Applications").
• The file date, size, and version: used to check if existing files should be overwritten or not. While this information could also be extracted at run time, having it in the SETUP.LST file speeds up the installation process quite significantly.

The VB3 SetupWizard moved at a glacial pace because setup apps it generated had to read version information from compressed files on the distribution floppies. Once SETUP.EXE has installed and registered the VB runtime files, SETUP1.EXE can take over.

SETUP1.EXE is the actual setup program, for which you can find the source in the \VB\SetupKit\Setup1 directory. It uses the two remaining sections from SETUP.LST to determine what to do.

The [Setup] section contains keys that control how the setup application presents itself to the user (the application title and default installation directory), in addition to controlling which program item will be created in Program Manager or Explorer at the end of the installation.

Unlike programs created by the VB3 SetupWizard, VB4's SETUP1.EXE creates only one icon per application by default, in keeping with the new Windows 95 guideline for preventing the Start menu from

VB4

```
Static Sub SplitUpFile(Fil As String)
Dim x As Integer, Root As String, Ext As String
Dim hSource As Integer, hDest As Integer
Dim SourceLen As Long, DestCount As Integer
Dim Dest As Integer, Length As Long
Dim Res As Integer
'//SplitUpFile: split up the file passed to this
'//function into fragments named
'//<basefile>.000, <basefile>.001, etc.
Const MaxSize = 1450000
'//free space on typical 1.4MB floppy
Const FirstChunk = 120000
'//remaining free space on first disk
'//(which contains setup files)
'//Open the file that is to be split
x = InStr(Right(Fil, 4), ".")
Ext = Right(Fil, 4 - x)
Root = Left(Fil, Len(Fil) - Len(Ext))
hSource = FreeFile
Open Root & Ext For Binary As #hSource
  SourceLen = LOF(hSource)
  '//Sanity check
  If SourceLen <= FirstChunk Then
    MsgBox "No need to split file!", 16
    Exit Sub
  End If
  '//Determine # of output files
  DestCount = 2 + (SourceLen - FirstChunk) \ MaxSize
  '//Create output files
  For Dest = 1 To DestCount
    hDest = FreeFile
    Open Root & Right("000" & _
      Trim(Str(Dest)), 3) For Binary As #hDest
      If Dest = 1 Then
        Length = FirstChunk
      ElseIf Dest = DestCount Then
        Length = (SourceLen - FirstChunk) _
          Mod MaxSize
      Else
        Length = MaxSize
      End If
      Res = CopyChunk(hSource, hDest, Length)
      If Res <> 0 Then
        MsgBox "Error during copy: " _
          & Error(Err), 16
        Exit Sub
      End If
    Close #hDest
  Next Dest
Close #hSource
End Sub
Sub AssembleFile(Root, As String, Ext As String)
Dim SourceCount As Integer, Fil As String
Dim hDest As Integer, Source As Integer
Dim hSource As Integer, Res As Integer
'//AssembleFile: re-assemble <Root>.### fragments
'//into <Root>.<Ext>
If Right(Root, 1) <> "." Then Root = Root & "."
'//Count file fragments
SourceCount = 0
Fil = Dir(Root & "0??")
While Len(Fil)
  SourceCount = SourceCount + 1
  Fil = Dir
Wend
'//Sanity check
If SourceCount = 0 Then
  MsgBox "No pieces of " & Root & " _
    found to assemble!", 16
  Exit Sub
End If
'//Create output file
hDest = FreeFile
Open Root & Ext For Binary As #hDest
  '//Merge fragment files into output file
  For Source = 1 To SourceCount
    hSource = FreeFile
    Open Root & Right("000" & Trim_
      (Str(Source)), 3) For Binary As #hSource
      Res = CopyChunk(hSource, _
        hDest, LOF(hSource))
      If Res <> 0 Then
        MsgBox "Error during copy: _
          " & Error(Err), 16
        Exit Sub
      End If
    Close #hSource
  Next Source
Close #hDest
End Sub
Static Function CopyChunk(hSource, _
  As Integer, hDest, As Integer,Length As Long)
Dim BlockSize As Integer, Buffer As String
Dim Blocks As Integer, LastBlockSize As Integer
Dim Block As Integer
'//CopyChunk: generic helper routine
'//that copies 8K blocks from one file
'//to another
On Local Error Resume Next
BlockSize = 8192
Buffer = Space(BlockSize)
Blocks = (Length \ BlockSize) + 1
LastBlockSize = Length Mod BlockSize
Err = 0
For Block = 1 To Blocks
  If Block = Blocks Then _
    Buffer = Space(LastBlockSize)
  Get #hSource, , Buffer
  Put #hDest, , Buffer
Next Block
CopyChunk = Err
End Function
Sub main()
'//Test driver for SplitUp & Assemble File
'//winword.exe -> winword.00? -> winword.ex2
SplitUpFile "c:\msoffice\winword\winword.exe"
AssembleFile "c:\msoffice\winword\winword.", "ex2"
End Sub
```

LISTING 2 ***Splitting Files Across Disks.*** *Though SetupWizard will split files across disks for you, no source code for doing so is provided. Study this code to learn how to do it yourself and take control of splitting files in both Visual Basic 3 and Visual Basic 4. The code in this listing also enables you to reassemble the resulting file fragments.*

VB4

User Tip

DELETE A FILE TO THE WIN95 RECYCLING BIN IN VB4/32

VB4/32 apps can call the Win32 API's SHFileOperation function to delete a file to the Windows 95 recycling bin. Here's how (this code also demonstrates VB4's new ParamArray statement):

```
Option Explicit
Type SHFILEOPSTRUCT
   hWnd        As Long
   wFunc       As Long
   pFrom       As String
   pTo         As String
   fFlags      As Integer
   fAborted    As Boolean
   hNameMaps   As Long
   sProgress   As String
End Type
Public Const FO_DELETE = &H3
Public Const FOF_ALLOWUNDO = &H40
Declare Function SHFileOperation Lib "shell32.dll" _
   Alias "SHFileOperationA" (lpFileOp As SHFILEOPSTRUCT) As Long

Public Function ShellDelete(ParamArray vntFileName() As Variant)

   Dim I As Integer
   Dim sFileNames As String
   Dim SHFileOp As SHFILEOPSTRUCT
   For I = LBound(vntFileName) To UBound(vntFileName)
   sFileNames = sFileNames & vntFileName(I) & vbNullChar
   Next
   sFileNames = sFileNames & vbNullChar
   With SHFileOp
      .wFunc = FO_DELETE
      .pFrom = sFileNames
      .fFlags = FOF_ALLOWUNDO
   End With
   ShellDelete = SHFileOperation(SHFileOp)
End Function
```

The SHFileOperation function's ParamArray argument allows you to call it in several ways:

```
' Delete a single file
lResult = ShellDelete("DELETE.ME")
' Pass file names in an array
sFileName(1) = "DELETE.ME"
sFileName(2) = "LOVE_LTR.DOC"
sFileName(3) = "COVERUP.TXT"
lResult = ShellDelete(sFileName())
' Pass file names as parameters
lResult = ShellDelete("DELETE.ME", "LOVE_LTR.DOC", "COVERUP.TXT")
```

—Phil Weber, *VBPJ* Technical Review Board

SEND YOUR TIP

If it's cool and we publish it, we'll pay you $25. If it includes code, limit code length to 10 lines if possible. Be sure to include a clear explanation of what it does and why it is useful. Send to 74774.305@compuserve.com or Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, CA, USA, 94301-2500.

becoming overpopulated. The [Files] section of SETUP1.EXE describes all files the setup apps needs to copy to the user's disk and register if necessary. The [Files] format is exactly the same as the format in the [Bootstrap] section, and the actions carried out are also identical, except that SETUP1 handles the $(AppPath), $(CommonFiles), and $(ProgramFiles) macros. Using these inside the [Bootstrap] section is fatal.

MORE INTERESTING STUFF

Because SETUP1.EXE comes with source code, you may want to modify it to display an alternate background, offer optional installation sections as described in the manual, add a form that asks for user registration information, and write it to your EXE, or install multiple program icons on installation, for example.

The code to do the latter is also interesting if you're wondering how to create shell links (items that appear directly on the Windows 95 Start menu instead of in a group under the Programs item). Also, functions inside the STKIT432.DLL might be interesting for your own applications.

Studying the source code of SETUP1.EXE is a great way to learn more about how the setup process works. The source may look a bit intimidating at first because none of the forms contain text.

The text is loaded at runtime from a resource file, and the actual code is full of conditional compilation statements and seemingly complicated function calls. Relax, though, because you can find a lot of useful information once you are familiar with the programming style and see the difference between the low-level logging functionality required to make uninstallation work, and the code sections where the real action is.

Of course, you can write your own setup program from scratch. I've updated my program (called VBPJstub) to work in 16- and 32-bit environments and to handle the new OLE registration requirements.

As described in my article, "A Good Install Is A Good Start," in the August 1995 issue of *VBPJ*, VBPJstub replaces Microsoft's SETUP.EXE. It includes C source code for your perusal, and the code in this article. Download VBPJstub from *VBPJ's* online sites (see "How to Reach Us" in the Letters to the Editor section).

The guidelines in my previous article still apply to Windows 95 and VB4. When in doubt, use the source code for SETUP1.EXE as a reference: this version contains no known logic errors that can cause you problems. With the VB4 SetupWizard and the tips I've explained you can take control of the entire setup process. ×