Chapter 24

Installation and Setup

"If a builder erect a house for a man and do not make its construction firm, and the house which he built collapse and cause the death of the owner of the house, that builder shall be put to death."

Code of Hammurabi, c. 2250 B.C.

24.1 Overview

Installing the DirectX runtime components required for an application is a non-trivial task. Further, installation of new runtime components may require a driver update for best functionality and performance. Beyond the requirements of the DirectX runtime, an application will also have its own executable and data files for installation to the user's machine. Beyond the simple act of copying files, the application may require the registration of COM objects, installation of fonts, installation of system services, and so-on.

An setup application must first determine the version of the currently installed DirectX runtime in order to decide if an upgrade should be suggested. The SDK provides a sample that determines the version of the DirectX runtime. Properly checking the installed version can eliminate potential installation problems for your users.

If you want to install the runtime with your setup program, you can use the DirectSetup API to perform the installation safely. Your setup program can then proceeed to install your application. A callback procedure can be used with DirectSetup to customize the installation of the runtime.

Windows Installer is a system component that can aid in the creation of robust application installation software. An overview of Windows Installer is presented. An example is given that shows how Windows Installer can be integrated with DirectSetup for DirectX applications.

Result	Meaning
`0x0000`	No DirectX installed
`0x0100`	DirectX version 1 installed
`0x0200`	DirectX 2 installed
`0x0300`	DirectX 3 installed
`0x0500`	At least DirectX 5 installed.
`0x0600`	At least DirectX 6 installed.
`0x0601`	At least DirectX 6.1 installed.
`0x0700`	At least DirectX 7 installed.
`0x0800`	At least DirectX 8 installed.
`0x0801`	At least DirectX 8.1 installed.
`0x0802`[1]	At least DirectX 8.1b installed.

Table 24.1: `GetDXVersion` return values. [1]This return value is only available with the implementation of `GetDXVersion` provided with this book.

24.2 Determining the DirectX Version

The SDK includes a sample called GetDXVer that obtains the version of Direct-X currently installed on the machine. It operates by looking for DLL files that are part of DirectX and examining them to determine the level of API support. You can use this in your setup program to determine if the appropriate DirectX version support is available for your application.

In the GetDXVer sample, the file `getdxver.cpp` contains the function `GetDXVersion` that returns a `DWORD` containing the version of the installed DirectX runtime components. Unfortunately, the version of this code in the 9.0c SDK leaks a DLL handle in the check for version 9.0c. A corrected version of this code can be found in the `launcher` sample. The return values for `GetDXVersion` are summarized in table 24.1.

```
DWORD GetDXVersion();
```

It is important that applications allow for the possibility that a newer version of DirectX than that required is installed. This means that the result of `GetDXVersion` should always be tested with an inequality rather than an equality. The following example checks to see if at least DirectX 8.1 is installed before continuing and displays a message box if the required version is not present.

```
if (::GetDXVersion() < 0x801)
{
    ::MessageBox(NULL, _T("DirectX 8.1 is required."),
        _T("Fatal Error:"), MB_ICONEXCLAMATION | MB_OK);
    return;
}
```

Many applications have incorrectly tested against the version number with an equality during their installation, only to fail when the user has a newer version of DirectX than that required for the application. This results in a frustrating user installation experience and makes a bad first impression on the user before they've even had a chance to run the application.

24.3 DirectSetup

The DirectX SDK provides a way for applications to install the DirectX runtime components through DirectSetup. DirectSetup consists of an API and a set of redistributable files that are used to install the appropriate DirectX components on a user's sytem. The header file `<dsetup.h>` defines the API for using DirectSetup.

24.3.1 Determining the DirectX Version

When using DirectSetup, an application can call `DirectXSetupGetVersion` before installation to get the version of DirectX currently installed. If this function is called after DirectSetup completes, then the version number returned will be the version of DirectX installed by DirectSetup if DirectSetup updated the system.

```
int ::DirectXSetupGetVersion(DWORD *version, DWORD *revision)
```

The `version` parameter returns the major and minor version of DirectX in the `HIWORD` and `LOWORD`, respectively. The `revision` parameter returns the release number and the build number in the `HIWORD` and `LOWORD`, respectively. The version numbers for different DirectX releases are summarized in table 24.2. If either of these values is not required by the application, `NULL` can be passed to the function.

24.3.2 Installing the DirectX Runtime

If your install program needs to install the DirectX runtime components to the user's machine, you can do this easily with DirectSetup. Attempting to install the runtime components yourself can corrupt the configuration of the user's machine. DirectSetup provides a single function, `DirectXSetup`, that installs the necessary runtime components to the user's machine.

```
int ::DirectXSetup(HWND window, LPSTR path, DWORD flags)
```

The `window` argument is used as the parent window for any setup dialog boxes required by DirectSetup. The `path` argument is the location of the DirectX 9.0c redistributable files, or `NULL` if the redistributable files are in the same directory as the application calling `DirectXSetup`. The `flags` argument is a combination of one or more of the following flags.

Major	Minor	Version	Build	DirectX	Operating System
4	5	0	155	5.0	Windows 98
4	5	1	1600	5.2	
4	6	0	318	6.0	
4	6	2	407	6.1	
4	6	2	436	6.1	
4	7	0	700	7.0	Windows 2000
4	8	0	400	8.0	
4	8	1	810	8.1	Windows XP
4	8	1	881	8.1a	
4	8	1	901	8.1b	
4	9	0	900	9.0	
4	9	0	904	9.0c	

Table 24.2: `DirectXSetupGetVersion` return values. Build numbers are changed every time DirectX is compiled at Microsoft, so you may encounter other values for the build number. For DirectX 9.0c, the build number will be at least the value shown.

```
#define DSETUP_DXCORE        0x00010000
#define DSETUP_DIRECTX       0x00010018
#define DSETUP_TESTINSTALL   0x00020000
#define DSETUP_USEROLDERFLAG 0x02000000
```

At a minimum you should use `DSETUP_DXCORE`, which installs the core runtime components, but does not attempt to update the user's drivers. The value `DSETUP_DIRECTX` installs the core runtime components and may install display and audio driver updates as well. The value `DSETUP_TESTINSTALL` allows you to test your installation program by going through all the motions of an installation without making any modifications to the system. You should use this flag when you are testing your install program's ability to update the runtime components on a target machine. The `DSETUP_USEROLDERFLAG` value allows the caller to detect when the user has a version of DirectX that is newer than the version in the redistributable files used by DirectSetup.

The return value can be one of the following values. Note that these return values are not `HRESULT`s, they are simple integer error codes.

```
#define DSETUPERR_SUCCESS_RESTART        1
#define DSETUPERR_SUCCESS                0
#define DSETUPERR_BADWINDOWSVERSION     -1
#define DSETUPERR_SOURCEFILENOTFOUND    -2
#define DSETUPERR_BADSOURCESIZE         -3
#define DSETUPERR_BADSOURCETIME         -4
#define DSETUPERR_NOCOPY                -5
#define DSETUPERR_OUTOFDISKSPACE        -6
```

```
#define DSETUPERR_CANTFINDINF          -7
#define DSETUPERR_CANTFINDDIR          -8
#define DSETUPERR_INTERNAL             -9
#define DSETUPERR_UNKNOWNOS            -11
#define DSETUPERR_USERHITCANCEL        -12
#define DSETUPERR_NOTPREINSTALLEDONNT  -13
#define DSETUPERR_NEWERVERSION         -14
#define DSETUPERR_NOTADMIN             -15
#define DSETUPERR_UNSUPPORTEDPROCESSOR -16
```

A succesful install will return DSETUPERR_SUCCESS, while DSETUPERR_SUCCESS_-RESTART indicates a successful install that will take effect once the machine has been restarted.

Before attempting to install DirectX, the setup program should check that approximately 20 MB of free space are available on the system drive for Windows 95, 98 and Windows Millenium Edition (ME). Windows 2000 and Windows XP will require approximately 25 MB of free space on the sytem drive. Installing DirectX is a system update and requires administrator privileges on operating systems with security, such as Windows 2000 and Windows XP.

24.3.3 Using a Callback Procedure

An installation program can fine tune the behavior of DirectSetup through the use of a callback procedure. To use a callback procedure, call DirectXSetup-SetCallback with a pointer to your callback procedure before calling Direct-XSetup.

```
int ::DirectXSetupSetCallback(DSETUP_CALLBACK Callback)

typedef
DWORD (PASCAL *DSETUP_CALLBACK)(DWORD reason,
    DWORD type,
    LPSTR message,
    LPSTR name,
    void *info);
```

The callback procedure will be invoked by DirectSetup during the process of the installation of the DirectX runtime components and device drivers. The reason for the call is given in the reason parameter and will be one of the following values.

```
#define DSETUP_CB_MSG_NOMESSAGE                0
#define DSETUP_CB_MSG_CANTINSTALL_UNKNOWNOS    1
#define DSETUP_CB_MSG_CANTINSTALL_NT           2
#define DSETUP_CB_MSG_CANTINSTALL_BETA         3
#define DSETUP_CB_MSG_CANTINSTALL_NOTWIN32     4
#define DSETUP_CB_MSG_CANTINSTALL_WRONGLANGUAGE 5
```

```
#define DSETUP_CB_MSG_CANTINSTALL_WRONGPLATFORM 6
#define DSETUP_CB_MSG_PREINSTALL_NT             7
#define DSETUP_CB_MSG_NOTPREINSTALLEDONNT       8
#define DSETUP_CB_MSG_SETUP_INIT_FAILED         9
#define DSETUP_CB_MSG_INTERNAL_ERROR            10
#define DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE      11
#define DSETUP_CB_MSG_OUTOFDISKSPACE            12
#define DSETUP_CB_MSG_BEGIN_INSTALL             13
#define DSETUP_CB_MSG_BEGIN_INSTALL_RUNTIME     14
#define DSETUP_CB_MSG_BEGIN_INSTALL_DRIVERS     15
#define DSETUP_CB_MSG_BEGIN_RESTORE_DRIVERS     16
#define DSETUP_CB_MSG_FILECOPYERROR             17
```

The `type` parameter is zero to display status messages given in the `message` parameter, or a combination of flags as would be passed for the `uType` parameter to the `::MessageBox` function. The `message` parameter will contain a localized string containing error or status messages that can be displayed with `::MessageBox`. The `name` parameter will be `NULL` unless the callback reason is `DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE`.

The callback procedure indicates the action to be taken by the return the value zero or the same values that would be returned by the `::MessageBox` function. If the value zero is returned, then DirectSetup takes the default action for upgrading the component or device driver. Otherwise, DirectSetup acts as if a message box had been displayed and the user had clicked the button corresponding to the return value. If your setup application wishes to automatically proceed or cancel a particular action, you can do this by controlling the return value from the callback function without displaying a message box. When the `type` parameter is zero, the callback procedure should return `IDOK` and no user input should be requested.

When the callback reason is `DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE`, the `info` parameter points to a `DSETUP_CB_UPGRADEINFO` structure and the `name` parameter gives the name of the driver being upgraded.

```
typedef struct _DSETUP_CB_UPGRADEINFO
{
    DWORD UpgradeFlags;
} DSETUP_CB_UPGRADEINFO;
```

The `UpgradeFlags` member contains a flag describing the upgrade type. The upgrade type indicates how Windows will perform the upgrade to the system. The keep flag indicates that Windows may fail if the device driver is updated, therefore DirectSetup will keep the existing driver. The safe flag indicates that the device driver may be safely upgraded and the upgrade is recommended. A "safe" upgrade is only safe with respect to Windows itself, other application programs may be adversely affected by a driver upgrade. The force flag indicates Windows may not function properly if the upgrade is not performed, therefore

DirectSetup will force the upgrade. The unknown flag indicates that DirectSetup does not recognize the existing driver for the device, which may happen quite often. Upgrading the driver in this situation may cause the device to stop functioning properly and an upgrade is not recommended.

```
#define DSETUP_CB_UPGRADE_TYPE_MASK             0x000F
#define DSETUP_CB_UPGRADE_KEEP                  0x0001
#define DSETUP_CB_UPGRADE_SAFE                  0x0002
#define DSETUP_CB_UPGRADE_FORCE                 0x0004
#define DSETUP_CB_UPGRADE_UNKNOWN               0x0008
```

The `UpgradeFlags` will also contain one or more of the following flags. The can't back up flag indicates that old system components cannot be backed up. If the upgrade is performed, then the components and drivers upgraded cannot be restored after the upgrade. The device active flag indicates that the device to be updated is currently in use. The display and media flags indicate that the device being updated is a display device or a media device, respectively. The has warnings flags indicates that DirectSetup can upgrade the driver for the device, but doing so may impact one or more installed applications, therefore the upgrade is not recommended. The names of the affected applications are given in the `message` parameter.

```
#define DSETUP_CB_UPGRADE_CANTBACKUP            0x0200
#define DSETUP_CB_UPGRADE_DEVICE_ACTIVE         0x0800
#define DSETUP_CB_UPGRADE_DEVICE_DISPLAY        0x1000
#define DSETUP_CB_UPGRADE_DEVICE_MEDIA          0x2000
#define DSETUP_CB_UPGRADE_HASWARNINGS           0x0100
```

The following code snippet shows how a callback procedure might respond to a safe upgrade of a device driver by automatically selecting the driver to be upgraded.

```
const DSETUP_CB_UPGRADEINFO *upgrade =
    static_cast<DSETUP_CB_UPGRADEINFO *>(info);
if (DSETUP_CB_UPGRADE_SAFE ==
    (upgrade->UpgradeFlags & DSETUP_CB_UPGRADE_TYPE_MASK))
{
    switch (type & 0xF)
    {
    case MB_YESNO:
    case MB_YESNOCANCEL:
        return IDYES;
    default:
        return IDOK;
    }
}
```

When the callback reason is `DSETUP_CB_MSG_FILECOPYERROR`, the `info` parameter points to a `DSETUP_CB_FILECOPYERROR` structure. The structure contains the error code encountered during the file copy operation.

```
typedef struct _DSETUP_CB_FILECOPYERROR
{
    DWORD dwError;
} DSETUP_CB_FILECOPYERROR;
```

24.3.4 Registering With DirectPlay Lobby

If your application uses DirectPlay, you can register your application with the DirectPlay Lobby with DirectSetup. Your setup program can call `DirectX-RegisterApplication` to register your application with the DirectPlay Lobby.

```
int ::DirectXRegisterApplication(HWND window, void *info);
```

The `info` argument is a pointer to a `DIRECTXREGISTERAPP` structure containing information that describes how to launch the application.

```
typedef struct _DIRECTXREGISTERAPP
{
    DWORD  dwSize;
    DWORD  dwFlags;
    LPTSTR lpszApplicationName;
    LPGUID lpGUID;
    LPTSTR lpszFilename;
    LPTSTR lpszCommandLine;
    LPTSTR lpszPath;
    LPTSTR lpszCurrentDirectory;
} DIRECTXREGISTERAPP;
```

The size member must be set to the size of the structure. The flags member is reserved and should be zero. The rest of the structure describes the application and the method of invoking the application.

To unregister your application with the DirectPlay Lobby, call `DirectXUn-RegisterApplication` with the same GUID used to register the application. If your application is registered with DirectPlay during installation you will need to unregister it before it is uninstalled to ensure proper functioning of the target system.

```
int ::DirectXUnRegisterApplication(HWND window,
        GUID *application);
```

For more information about DirectPlay, see the DirectPlay documentation in the SDK.

24.3.5 Preparing the Redistributables

The DirectX SDK comes with a complete set of files that allow you to install DirectX with DirectSetup. The set of files that you can provide with your application's setup program are called the redistributable files for dx. The `DirectX81` subdirectory of the `Redist` directory in the SDK contains the redistributable files for DirectX 9.0c. These files must be copied in their entirety to your source media in order to install the runtime as part of your setup application.

Thelocation of these file can be passe directly as the `path` argument to `DirectXSetup`. If `NULL` is passed as the path, these files reside in the same directory as your setup program. For example, the following initiates a test installation of the DirectX runtime with the redistributable files located in the `Redist`
`DirectX81` directory relative to the setup program.

```
const int result = ::DirectXSetup(window, "Redist\\DirectX81",
    DSETUP_DXCORE | DSETUP_TESTINSTALL);
if ((DSETUPERR_SUCCESSRESTART != result) &&
    (D3DSETUPERR_SUCCESS != result)
{
    return;
}
```

24.4 Windows Installer

While DirectSetup ensures that the correct version of the runtime is present for your application, you will still need to copy program and data files from the source distribution onto the target machine. For a sophisticated application you may need additional setup tasks such as manipulating the system Registry, providing COM objects, .Net assemblies, and system services. With a traditional setup program this may require extensive coding, testing and debugging of custom setup program code or installation scripts for traditional setup authoring tools.

Windows Installer provides a base facility that reduces most installation authoring to specifying information about the installation tasks into a database. The database is stored in an `MSI` file and consists of tables containing rows. Each row supplies values to the columns of the table. The actions necessary to install the application – such as copying files, writing registry entries, and so-on – are stored in a ordered sequence in the database. Most actions are standard actions and result in typical installation program operations, such as copying files. A setup database can contain custom installation actions supplied as either a DLL shared library file or an executable file.

Windows Installer does not provide a standard action for ensuring a particular version of the runtime, so we must provide a way to ensure the proper runtime. One approach would be to ensure that the installation of the application cannot proceed if the proper runtime is not present. The user would be required to install the proper runtime before the application can be installed. Because the installation of the DirectX runtime is an irreversible process and often results in a required restart of the system, it may be preferred to require the user to undertake this update before installing the application. The display driver is a critical part of the system and some administrators may prefer that any changes to the display subsystem be tightly controlled.

The remainder of this section assumes some familiarity with Windows Installer and its corresponding database tables. For more information about Windows Installer, see the Platform SDK documentation.

24.4.1 Using a Launch Sequencer

You can use a program to ensure that appropriate system updates are applied to the system before your application installation is performed. Some systems may not have Windows Installer, or may require a newer version of Windows Installer for your application. Your program may require other system updates, such as the DirectX runtime, DCOM or other system components. These system updates are usually available for redistribution as self-contained setup programs that must be launched directly and cannot be incorporated as merge modules in a Windows Installer based installation. System updated, such as the DirectX runtime, also often require that the target system be rebooted at the end of the installation for the update to take effect. Rebooting in the middle of your application's installation is not recommended and your application may require the reboot to have occurred before your installation is run.

Using a launch sequencing program simplifies the needs of your Windows Installer database and performs such system component updates in a robust manner. A launch sequencer is a simple program that probes the target system for the necessary system components and launches your application's installation if the necessary items are present. When the system components must be updated to provide the minimum supported level for your application, the launch sequencer can invoke the redistributable setup programs for the system components. It is recommended that the launch sequencer provide some sort of simple user interface before invoking the setup programs so that the user is aware of the updates and has a chance to cancel them. Some system updates, such as the DirectX runtime, are irreversible once installed and users do not appreciate having irreversible updates performed on their computer without their consent.

24.4.2 Requiring the Proper Runtime Version

The LaunchCondition table contains rows specifying conditions that must be true before the installation can proceed. The installation for a DirectX application can use a custom action to set a property to indicate if the required version of the DirectX runtime is installed. This property can then be used in the LaunchCondition table to allow the installation to proceed only when the proper version of the runtime is present.

An example of this approach can be found in the `MSI` file for this book. Requiring the proper runtime to be installed before running your application's install allows your application to be launched to perform any necessary performance tests and one-time configuration.

24.4.3 Using a Custom Action to Invoke DirectSetup

Another approach to ensuring the correct runtime version is to invoke DirectSetup from within your `MSI` file with a custom action. The disadvantage here is that your application cannot perform any necessary performance or configuration operations until the runtime is completely installed. To completely install the runtime will require a restart of the system. Restarting the system in the middle of your Windows Installer based installation can be difficult to get working properly and is not recommended. However, if your application does not require the use of the runtime until after the system has been restarted, this approach is fine.

You still need a rudimentary launch sequencing program, however, as one of the system updates that the launch sequencer checks for is the presence of Windows Installer itself. Windows Installer is not guaranteed to be present on Windows 95, Windows 98 or Windows NT4. It is shipped as part of the operating system with Windows 2000 and Windows XP. Even in those cases where it is guaranteed to be present, it may need to be upgraded to a newer version used by your installation. Since the launch sequencer program is required in all cases anyway, the preferred method to installing the runtime is to add those checks to the launch sequencer and invoke the DirectX installation program directly.

An example of a custom action invoking DirectSetup directly is given in the sample code accompanying this book.

24.5 Logo Requirements

Microsoft runs a program that allows your application to display the "Built for Microsoft Windows" logo on the materials associated with your product. Microsoft requires that applications adhere to a set of standards before the logo can be displayed in conjunction with your program. One of the requirements for the logo program is that applications use Windows Installer for installation to ensure proper, robust and resilient installation of applications. There are additional requirements beyond the use of Windows Installer and these are detailed on the Microsoft web site. Adhering to the requirements of the logo program and displaying the logo on your software allows consumers to know that your program adheres to the requirements published by Microsoft, establishing an easily visible confidence in the quality of your product.

24.6 AutoRun Enabled Installations

If your application is distributed on removable media, you can include files to enable the automatic launching of your setup application as soon as the media is recognized. This scenario is called AutoRun in the Platform SDK documentation. AutoRun is enabled when the root directory of the distribution media contains a file called `autorun.inf` which specifies the program to launch.

For more information about AutoRun enabled applications, see the documentation in the Platform SDK.

24.7 Testing Installation Programs

Testing installation programs can be a difficult process, since the installation can perform an irreversible update to the system under test. Further, bugs in the installation process may have left the target system in an unknown state with only a portion of the application installed properly. Using the system to remove these partial installations may prove difficult if their presence was due to a faulty setup application, even one using Windows Installer.

The easiest way to always ensure that systems are in a consistent state before testing an installation program is to prepare a “virgin” system without any of the system updates necessary for your application. Then perform a complete backup of this “virgin” system. A very easy way to accomplish this is to use a tool that makes a snapshot of a hard-drive onto a CD-R, such as the Norton Ghost utility. Then the installation program can be tested on the virgin system and the hard drive image used to effect a complete restoration of the original system before invoking the test again.

24.8 Sample Program

The DXInstall miscellaneous sample in the SDK gives a complete example of calling DirectSetup with a callback procedure and a custom user interface.

TODO: revise CD-ROM discussion

The entire installation for this book is in the samples. The CD-ROM is AutoRun enabled and the `autorun.inf` file invokes `launcher.exe`. The launcher program is a simple C++ program that determines the version of Windows Installer and DirectX currently installed and passes them to `setup.hta`, which acts as the GUI front end for the installation. It allows the user to launch the installation of any necessary system updates, including the installation of the DirectX SDK, before installing the software samples and extras that come with the book. The file `setup.hta` is a simple HTML Application that functions as the launch sequencing application. An HTML application was chosen simply because it allows such a GUI front end to be easily constructed. It does require that the user have Internet Explorer 5 or later installed, however. When launching `setup.hta` is not successful, `launcher.exe` displays a `ReadMe.txt` file giving manual installation instructions to the user.

TODO: revise CD-ROM

The book samples and extras themselves are installed with a Windows Installer MSI database. This database is present on the CD-ROM in the form of an XML file, created with the open source utility `msi2xml`. The XML file is located in the `MSI` directory on the CD-ROM.