

4

This partial chapter is an excerpt from the *Learning InstallScript Projects using InstallShield 2009* training manual. For information about purchasing this training manual and to learn about other Acreoso software training materials, please visit www.acresso.com.

Working with Scripts

In an InstallScript project, the user interface and behavior of your installation program are controlled by your project's InstallScript code. This chapter explains how to modify your project's script to perform additional tasks during installation or uninstallation. In particular, this chapter provides examples of modifying your script to:

- Include additional dialog boxes in the installer's user interface.
- Search for files on the user's system.
- Read data from the user's registry and `.ini` files.
- Create shortcuts and program folders.
- Hide and select program features.
- Check system requirements.
- Launch your application when the installation is complete.

For information about InstallScript syntax and the families of built-in InstallScript functions, see [Appendix A, "The InstallScript Language"](#) and the online InstallScript Language Reference in the InstallShield Help Library.

The InstallScript View

You edit your project's script in the **InstallScript** view, in the Behavior and Logic view group. The InstallScript view contains the Script Editor, with which you modify the code contained in any script files contained in your project, and a Functions tree that shows all the functions defined in your project's scripts.

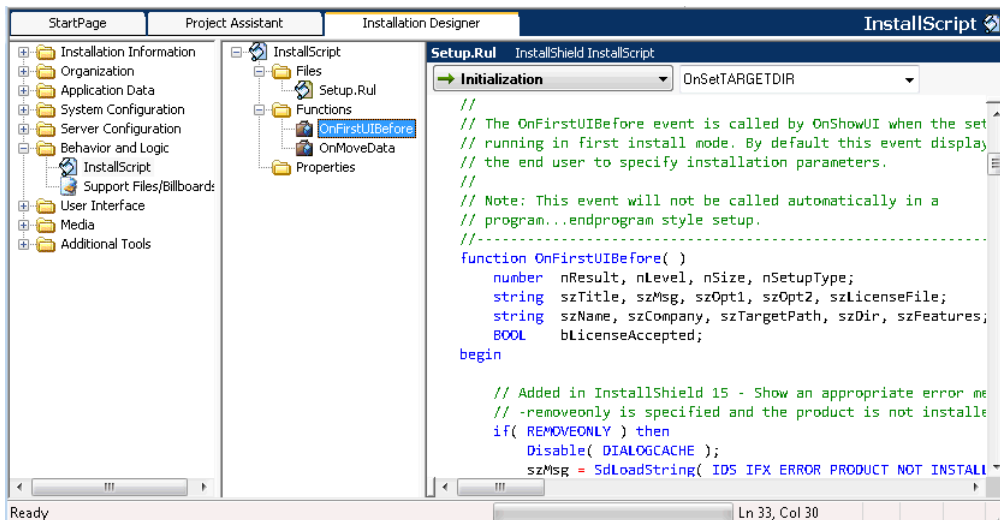


Figure 4-1: The InstallScript view

The Properties part of the script tree is used primarily in Object projects, which are described in [Chapter 10, “InstallShield Objects”](#).

Syntax Coloring

The Script Editor displays parts of a script in different colors according to the role the part plays in the InstallScript language, a feature called syntax coloring. Text not recognized by the script editor is displayed in black (or the system default) font, which gives you a quick way to catch any typing errors when using built-in functions and constants.



Task To change the default colors used in the Script Editor:

1. Right-click in the Script Editor and choose **Properties** to display the Window Properties dialog box.
2. Click the **Color/Font** tab and change the color settings for the various script items.

IntelliScript

The Script Editor also provides the IntelliScript feature. With IntelliScript, you can type the first few letters of a function name and press **Ctrl+Spacebar**, which displays a list of possible completions for the function name. Moreover, when you type a function name and the opening parenthesis for the function’s arguments, a tool tip appears, displaying the number and types of arguments for the function, using Hungarian notation (as described in [Appendix A, “The InstallScript Language”](#)).

Compiling Your Script

You can ensure that your sample program contains no errors by compiling it, which converts the script into code that the InstallScript run-time engine understands.



Task To compile a script, do one of the following:

1. Click the **Compile Setup** button in the toolbar.
2. From the **Build** menu, choose **Compile**.
3. Press **Ctrl+F7**.

If the script compiles correctly, the **Compile** tab of the Output window displays the following message:

```
Setup.inx - 0 error(s), 0 warning(s).
```

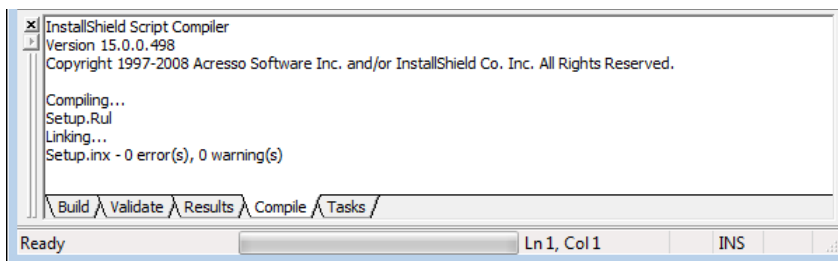


Figure 4-2: The Compile tab of the Output window

Error Messages

If compiling the script generates any error or warning messages, double-clicking an error message in the Output window causes the script editor to highlight the line containing the error. Common errors include failing to capitalize a function name properly, or omitting a semicolon after a function call.

It is good practice to deal with error messages from top to bottom, as punctuation errors early in a script may cause misleading error messages later in the script. Outstanding errors and warnings from the compiler are also listed in the **Tasks** tab of the Output window, which also contains hyperlinks to Knowledge Base articles related to the errors and warnings.

Running Your Script

After you have compiled your script, you can execute the script by clicking the **Run** button in the toolbar, selecting **Run ReleaseName** from the **Build** menu, or pressing **Ctrl+F5**.

If you run a script without compiling it first, InstallShield automatically compiles the script.

Event Handlers

Each of the functions defined in a Project Assistant script is an event handler, called in response to an event in the script. All of a script's event handlers are listed in the InstallScript view, and you can highlight and edit an existing event handler function in the script editor by selecting its name in the Functions tree in the project workspace.

To insert an event handler, select the class of event from the left list in the script toolbar, and then select the specific event handler you want to create from the right list. If a specific event handler is already defined in the current script, its name is displayed in bold; otherwise, it is displayed in plain roman type.



Figure 4-3: The event category and event lists in the Script Editor

The event categories include:

- Initialization events
- Before Move Data events
- Move Data events
- After Move Data events
- Miscellaneous events
- Advanced events

Initialization Event Handlers

Initialization event handlers are called directly by the installation engine; they are described in the following table.

Table 4-1: Initialization event handlers

Event Handler	Description
OnSetUpdateMode	The first function called when the user launches the installation program, even during a maintenance installation. (During a maintenance installation, InstallScript sets the system variable <i>MAINTENANCE</i> to TRUE, and you can test the value of <i>MAINTENANCE</i> in an <i>if</i> statement to specify a routine to run only on a first-time setup.) OnSetUpdateMode determines whether the installation is an update to an existing installation and sets the value of the system variable <i>UPDATEMODE</i> accordingly.
OnSetTARGETDIR	Sets the value of <i>TARGETDIR</i> , the main destination folder for your application. As described in Chapter 2, "Creating an Installation Project" , you can set the default value for <i>TARGETDIR</i> in the Product Properties view.
OnCheckMediaPassword	Prompts the user for the media password if you specified one if your release settings, as described in Chapter 3, "Creating Distribution Media" .

Before Move Data Event Handlers

The Before Move Data event handlers are triggered before files are transferred to the target system. The following table describes the Before Move Data event handlers (and the order in which they are called).

Table 4-2: Before Move Data event handlers

Event Handler	Description
OnBegin	Called when the user launches the installation program, even during a maintenance installation. (During a maintenance installation, InstallScript sets the system variable <i>MAINTENANCE</i> to TRUE, and you can test the value of <i>MAINTENANCE</i> in an <code>if</code> statement to specify a routine to run only on a first-time installation.) OnBegin is useful for testing system requirements, aborting the installation if the target system does not meet the application requirements.
OnCCPSearch	(Compliance Checking Program Search) Called only for a first-time installation. OnCCPSearch is used for searching for an earlier version of an application.
OnAppSearch	(Application Search) Called only for a first-time installation. OnAppSearch is used for searching for an application related to the current installation, for example using FindAllFiles or by searching for the application's registry key.
OnFilterComponents	Called for all installations. OnFilterComponents handles component filtering—that is, the including and excluding of features' components in the data transfer, based on their Language and Operating System settings. Filtering based on component operating system and language is described in Chapter 5, "Special Considerations" and in Chapter 8, "Localizing Your Installation" .
OnFirstUIBefore	Called only for first-time installations. By default, this event displays dialog boxes that enable end users to specify installation parameters.
OnMaintUIBefore	Defines the user interface for maintenance installations.
OnUpdateUIBefore	Defines the user interface for update installations.

Move Data Event Handlers

The Move Data event handlers are triggered immediately before, during, or immediately after the installation or uninstallation of all features on the target system. The following table describes the Move Data event handlers (not including feature event handlers, which are described in the following section).

Table 4-3: Move Data event handlers

Event Handler	Description
OnMoveData	Called by the OnShowUI advanced handler's default code to handle the data transfer. The default code for OnMoveData calls FeatureTransferData to transfer the files and other data; FeatureTransferData, in turn, triggers the rest of the Move Data events.
OnCustomizeUninstInfo	Called to customize the display name of your product in the user's Add or Remove Programs panel in the case of a multi-instance installation project. Multi-instance installations are described in Chapter 7, "Additional Deployment Topics" .
OnMoving	Called just before data transfer begins. The default implementation of OnMoving creates the application-path values in the registry.
OnInstallingFile	Called for each file being installed (file name stored in an argument to the function).
OnUninstallingFile	Called for each file being removed (file name stored in an argument to the function).
OnMoved	Called when file transfer is complete.

After Move Data Event Handlers

The After Move Data event handlers are triggered after files and other data are transferred to the target system. The following table describes the After Move Data event handlers.

Table 4-4: After Move Data event handlers

Event Handler	Description
OnFirstUIAfter	Specifies the user interface to display after file transfer is complete for a first-time installation. The default implementation of OnFirstUIAfter calls SdFinishReboot or SdFinish to inform the user that the installation program is complete.
OnMaintUIAfter	Specifies the user interface to display after file transfer is complete for a maintenance installation. The default implementation of OnMaintUIAfter calls SdFinish or SdFinishReboot .
OnUpdateUIAfter	Specifies the user interface to display after data transfer is complete for an update installation. The default implementation calls SdFinish or SdFinishReboot .

Table 4-4: After Move Data event handlers

Event Handler	Description
OnEnd	The last function called by an installation program, for first-time and maintenance and update installations.

Advanced Event Handler

The Advanced event handler, `OnShowUI`, is called by the installation engine. `OnShowUI` initiates the user interface and data transfer. By default, `OnShowUI` optionally enables the installation background window, and then calls the appropriate `OnXXXXXUIBefore` handler, the `OnMoveData` handler, and the appropriate `OnXXXXXUIAfter` handler.

Feature Event Handlers

You can attach up to four event handlers to any feature:

- `OnInstalling`
- `OnInstalled`
- `OnUnInstalling`
- `OnUnInstalled`

`OnInstalling` and `OnUnInstalling` are called just before the feature is installed or removed, and `OnInstalled` and `OnUnInstalled` are called just after the feature is installed or removed.

To create a feature event handler, select the name of the feature in the category list in the script toolbar, and then select the desired event (`Installing`, `Installed`, `UnInstalling`, `UnInstalled`). InstallShield creates a function block for the desired feature event handler.

Each feature event handler has the initial name `FeatureName_Event` (for example, `ProgramFiles_Installing`), and you can specify a different existing event handler function in the Features view. InstallShield places the feature event handler functions that it creates in a source file called `FeatureEvents.rul`; the end of `Setup.rul` imports the file with the `#include` directive.

Miscellaneous Events

InstallScript defines several additional miscellaneous events. You can add a miscellaneous event handler by selecting **Miscellaneous** from the category (left) list of the script toolbar, and selecting the desired event from the event (right) list. InstallShield calls the following event handlers when specific types of files are installed or removed.

Table 4-5: Miscellaneous event handlers

Event Handler	Description
OnFileReadOnly	Called when a file that is to be overwritten or removed has the read-only attribute set. The default implementation of OnFileReadOnly prompts the user whether to remove or skip the operation being performed on the read-only file; and you can modify the event handler to display a different message to the user, or not to display any feedback.
OnFileLocked	Called when an executable file (.exe, .dll, .ocx, and so on) to be overwritten or removed is in use. Note that OnFileLocked is called only for files in components not marked as Potentially Locked. Locked files are discussed in Chapter 5, “Special Considerations” .
OnNextDisk	Prompts the user to insert the next disk (using the EnterDisk dialog box) in a multi-disk installation.
OnRemovingSharedFile	Called when an uninstallation is ready to remove a formerly shared file. The default implementation of OnRemovingSharedFile is to prompt the user whether to remove the file. Shared files are discussed in Chapter 5, “Special Considerations” .
OnAbort	Called when an abort event is generated by the InstallScript abort statement.
OnMD5Error	Called when an installation encounters a checksum (CRC) error, generally when a file is corrupted. OnMD5Error should return one of the constants <code>ERR_ABORT</code> , <code>ERR_RETRY</code> , or <code>ERR_IGNORE</code> . As described in Chapter 3, “Creating Distribution Media” you can specify in the Release Wizard whether the installation should use the OnMD5Error event handler.
OnSelfRegistrationError	Called when a file fails to register. Chapter 5, “Special Considerations” discusses self-registering files.
OnRebooted	Called after a system reboot.
OnFeatureError	Called when an installation experiences a run-time error during data transfer.
OnFileError	Called when an installation experiences a run-time error with a particular file. OnFileError should return one of <code>ERR_ABORT</code> , <code>ERR_RETRY</code> , or <code>ERR_IGNORE</code> .
OnInternetError	Called when a One-Click Install encounters an Internet-related error.

Table 4-5: Miscellaneous event handlers

Event Handler	Description
OnCheckMediaPassword	Called in some circumstances to prompt the end user for a password, and then validate it using the FeatureValidate function. For information on when this event handler is called, see Chapter 3, “Creating Distribution Media” .

The following miscellaneous events are called in response to user actions.

Table 4-6: Miscellaneous event handlers

Event Handler	Description
OnCanceling	Called when the end user cancels the installation by clicking the Cancel button on a dialog box, or presses the Escape key.
OnHelp	Called when the end user presses the F1 key.

Common Script-Based Tasks

This section explains how you can perform many common tasks inside your installation script.



Note: The sample scripts in this chapter use hard-coded strings in display messages and other function parameters. In a multi-language installation program, display strings used by your `InstallScript` code should be stored in string-table entries, to assist with displaying localized text to the end user. String-table entries are described in [Chapter 8, “Localizing Your Installation”](#).

Displaying Dialog Boxes

Most of your installation program’s user interface is defined in the following event handlers:

- `OnFirstUIBefore`
- `OnFirstUIAfter`
- `OnMaintUIBefore`
- `OnMaintUIAfter`
- `OnUpdateUIBefore`
- `OnUpdateUIAfter`

In a project you create with the Project Assistant, the dialogs you selected in the Installation Interview page are contained in the `OnFirstUIBefore` event handler, which defines the user interface for a first-time installation.

The beginning of a typical **OnFirstUIBefore** function appears as shown in the following script example. Script comments containing the text `IS_SCRIPT_TAG` may be inserted by the Project Assistant to assist with enabling and disabling dialog boxes in the Installation Interview page.

```
Dlg_Start:  
    // beginning of dialogs label  
Dlg_SdWelcome:  
    szTitle = "";  
    szMsg   = "";  
    nResult = SdWelcome( szTitle, szMsg );  
    if (nResult = BACK) goto Dlg_Start;  
Dlg_SdRegisterUser:  
    szMsg   = "";  
    szTitle = "";  
    nResult = SdRegisterUser( szTitle, szMsg, szName, szCompany );  
    if (nResult = BACK) goto Dlg_SdWelcome;  
  
// ... etc. ...
```

Every dialog box function returns a constant indicating the button the user clicked to exit the dialog box. To handle the user clicking the **Back** button on a dialog box, directly after the dialog box is an `if` statement that compares the dialog's return value to the constant `BACK`, using a `goto` statement to jump to a label just before the previous dialog box. In the previous code example, if the user clicks **Back** on the **SdRegisterUser** dialog box, the `goto` statement jumps to a label just before the previous **SdWelcome** dialog box. Therefore, if (for example) you insert a dialog box function between **SdWelcome** and **SdRegisterUser**, you need to adjust the `if` statements, labels, and `goto` statements.

Changing Dialog Box Text

Most dialog box functions accept a string argument called `szTitle` that defines the text appearing in the title area of the dialog box. For example, the call to **SdRegisterUser** appears as follows:

```
SdRegisterUser( szTitle, szMsg, szName, szCompany );
```

By default, the parameter `szTitle` is defined as a null string (""), which indicates that the dialog box should use the default title text. For **SdRegisterUser**, the default title appears as shown in the following figure.

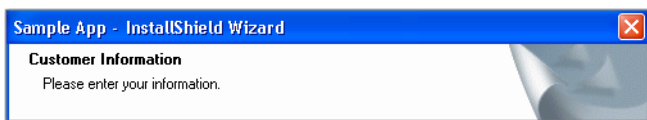


Figure 4-4: The default title text for the `SdRegisterUser` dialog box

To modify the title, you can enter specific strings to display for the `szTitle` parameter. The title is divided into two sections—the bold text at the top of the title area, and the roman text under the main title.

To specify the two sections in the value of `szTitle`, place a newline character `\n` between the two sections, as in the following.

```
SdRegisterUser ("New Title\nThis is a new subtitle.",  
szMsg, szName, szCompany);
```

After recompiling the script and running the installation, the changed title appears as shown in the following figure.

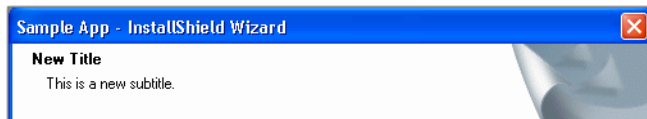


Figure 4-5: Changed title text for the `SdRegisterUser` dialog box

To change other text displayed on a dialog box, there are usually one or more parameters (such as `szMsg`) that contain the text to be displayed. As with dialog box titles, a null string in a message parameter indicates that the dialog box should use the default message text provided by InstallShield.

Changing the Bitmap on Dialog Boxes

You can also change the bitmap used in **Sd** dialog boxes, using the **DialogSetInfo** function with the `DLG_INFO_ALTIMAGE` argument. All **Sd** dialog boxes that appear after the call to **DialogSetInfo** display the new bitmap. You can include the bitmap in the Support Files/Billboards view (in, for example, the Language Independent section), and refer to the directory containing the bitmap as `SUPPORTDIR`.



Note: If a custom bitmap is larger than the recommended size, it will be cropped, rather than resized.



Task To change the bitmap:

1. Copy a bitmap (for example, one named `Alt.bmp`) into the Support Files/Billboards view. The bitmap should be roughly 60 pixels wide and 60 pixels high.
2. Insert the following function call into the script. You can type the function call or use the Function Wizard, which is described in [Appendix A, “The InstallScript Language”](#).

```
DialogSetInfo(DLG_INFO_ALTIMAGE, SUPPORTDIR ^ "Alt.bmp", TRUE);
```

3. Rebuild the media set and run the installation.

At run time, InstallShield displays the custom bitmap image in the upper-right corner of all **Sd** dialog boxes.

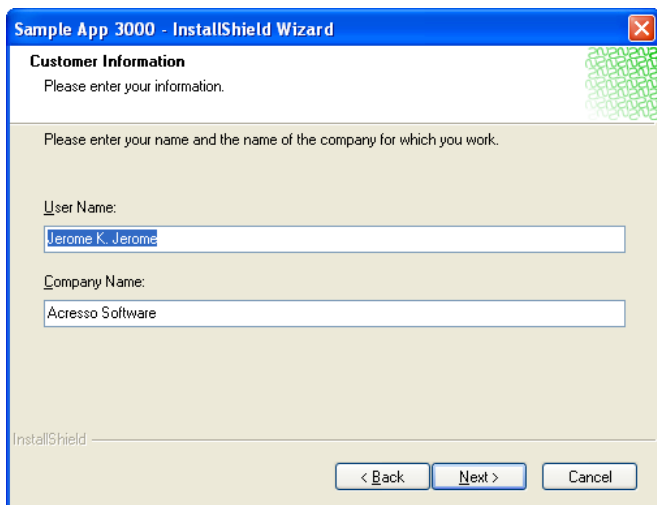


Figure 4-6: The Customer Information dialog box with a custom bitmap image



Tip: Chapter 3, “Creating Distribution Media” describes how to specify a skin to use with the dialog boxes displayed by your installation program.

Copying Files

As explained in Chapter 2, “Creating an Installation Project”, you define the files you want your installation to transfer in the Components view. To transfer additional files, you can use the InstallScript function **XCOPYFILE** to copy files to a target system. Files copied with **XCOPYFILE** are logged for uninstallation.

As with setting component destinations, when working with script functions you will commonly use InstallScript system variables to refer to special directories on the target system. When your installation program initializes, it populates these system variables by querying the target system’s registry or Windows API functions.

Some of the available (read-only) directory variables are described in the following table. As described in Chapter 5, “Special Considerations”, many of these variables refer either to the per-machine directory or to the per-user directory based on the value of the `ALLUSERS` variable.

Table 4-7: Directory variables

Variable	Description
<code>PROGRAMFILES</code>	The target system’s Program Files folder.
<code>COMMONFILES</code>	The target system’s Common Files folder.
<code>FOLDER_DESKTOP</code>	The All Users or per-user Desktop folder.
<code>FOLDER_PROGRAMS</code>	The All Users or per-user Programs menu folder.
<code>FOLDER_STARTMENU</code>	The All Users or per-user Start menu folder.
<code>FOLDER_PERSONAL</code>	The user’s My Documents folder.

Table 4-7: Directory variables

Variable	Description
<code>FOLDER_TEMP</code>	The Temp folder.
<code>FOLDER_APPDATA</code>	The Application Data folder.

Two InstallScript system variables commonly used with **XCOPYFILE** are `SRCDIR`, which refers to the installation source directory, and `TARGETDIR`, which refers to the main program installation directory. For example, to copy an uncompressed file called `Readme.txt` from the installation source directory to `TARGETDIR`, you can use the following code:

```
XCOPYFILE(SRCDIR ^ "Readme.txt", TARGETDIR, COMP_NORMAL);
```

The last argument for **XCOPYFILE** is a combination of predefined constants that indicate how to overwrite existing files on the target system, and special installation instructions. The possible values for the `nOp` parameter are described in the following table. To specify more than one argument, use the bitwise-or operator `|`.

Table 4-8: `nOp` arguments

Constant	Description
COMP_NORMAL	Transfer the file regardless of its version or modification-date relationship with an existing file.
COMP_UPDATE_SAME	Overwrite a file even if the file in the installation program has the same version or date as the existing file. You must specify <code>COMP_UPDATE_VERSION</code> or <code>COMP_UPDATE_DATE</code> with this constant; otherwise, it is ignored.
COMP_UPDATE_VERSION	Overwrite a file only if the file in the installation program has a newer version than the existing file.
COMP_UPDATE_DATE	Overwrite a file only if the file in the installation program has a later modification date and time than the existing file.
SELFREGISTER	Carries out the self-registration process immediately when using the “non-batch method” of installing self-registering files. (Installation of self-registering files is described in Chapter 5, “Special Considerations” .)
SHAREDFILE	Causes XCOPYFILE to treat all files as shared files and increment the registry reference counter by one when the file exists in the target directory and it has a reference count greater than 0. If the shared file does not exist in the target directory and it has no reference counter, InstallShield creates the counter and sets it to 1. If the shared file already exists in the target directory but has no reference counter, InstallShield creates the counter and initializes it to 2 as a precaution against accidental removal during uninstallation.

Table 4-8: nOp arguments

Constant	Description
LOCKEDFILE	Causes XCOPYFile to record locked .dll and .exe files for update when Windows or the system is rebooted. A locked file is a file that is in use by an application or the system when InstallShield attempts to access or update the file. The LOCKEDFILE option works like SHAREDFILE except that LOCKEDFILE does not create registry entries or modify the registry reference counter. You cannot use the LOCKEDFILE option when using the SHAREDFILE option. There are some unshared files (such as shell extensions) for which the script writer does not want a registry entry and reference counter. These files should never be uninstalled, except by the application itself. LOCKEDFILE enables XCOPYFile to handle locked files that are not shared.
EXCLUDE_SUBDIR	Specifies not to include subdirectories contained in the source path.
INCLUDE_SUBDIR	Specifies that subdirectories below the source path must also be copied. Note that empty subdirectories are not copied.

You can use wildcard specifications in the source argument to **XCOPYFile** to copy a collection of files; the constants **INCLUDE_SUBDIR** and **EXCLUDE_SUBDIR** let you specify whether to copy subdirectories of the source directory to the target system.

You can also use the functions **DeleteFile** and **DeleteDir** to remove files and folders from the target system.

Many file-transfer and file-information functions are Internet-enabled, meaning the source argument can be a URL. The collection of Internet-enabled functions includes **XCOPYFile**, **CopyFile**, **Is**, **ExistsDir**, **OpenFile**, **GetLine**, **ReadBytes**, and **SeekBytes**.

Searching for Files

Sometimes an installation program needs to verify that certain files already exist on the target system.

Files with Known Locations

If you know the full path to an existing file, you can use the **Is** function with the **FILE_EXISTS** constant, which returns **TRUE** if the file exists. For example, if your installation program requires **Notepad.exe** to be located in the user's Windows or WINNT folder, you can use code similar to the following in your **OnBegin** event handler. (The **abort** statement exits the installation, and silently uninstalls any changes made to the target system.)

```
// abort installation if Notepad.exe is not present
// in the user's Windows folder
function OnBegin( )
begin
if (!Is(FILE_EXISTS, WINDIR ^ "Notepad.exe")) then
    MessageBox("This application requires Notepad.exe.\n\n" +
        "Setup will now exit.", SEVERE);
    abort;
```

```
endif;
end;
```

Files with Unknown Locations

If you do not know the location of a file on the target system, you can use the **FindAllFiles** function in your script to search for one or more copies of a file with a given name in a specified directory or its subdirectories. You can include wildcard specifications in the name of the file you want to search for. The `OnAppSearch` event handler, which is called only for a first-time installation, is provided for you to search for existing files on the target system.

For example, the following `OnAppSearch` event handler displays the full path to the file `ISDev.exe` if it is present in the user's Program Files folder, or displays an error message and aborts the installation if the file is not found. Because the search may take several seconds, you can display the **SdShowMsg** dialog box, shown in the following figure, while **FindAllFiles** is operating.

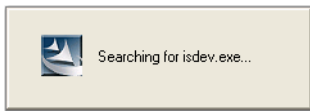


Figure 4-7: The `SdShowMsg` dialog box

You can also call `Enable(HOURGLASS)` and `Disable(HOURGLASS)` to change the mouse pointer while an operation is taking place.

```
// abort installation if isdev.exe is not present in the user's
// Program Files folder
function OnAppSearch( )
    // local variables used by FindAllFiles
    STRING svFilepath; // returned full path to file
    NUMBER nReturn;    // status returned by FindAllFiles
begin
    // show status dialog box
    SdShowMsg("Searching for isdev.exe...", TRUE);
    // use RESET for first search; use CONTINUE to search for more
    // copies of the file
    nReturn =
        FindAllFiles(PROGRAMFILES, "isdev.exe", svFilepath, RESET);
    // remove status dialog box
    SdShowMsg("", FALSE);
    if (nReturn = 0) then // search was successful
        MessageBox("isdev.exe is located at: " + svFilepath, INFORMATION);
    else
        MessageBox("Setup is unable to locate isdev.exe.\n\n" +
            "Setup will now exit.", SEVERE);
        abort;
```

```
endif;  
// release resources for further searches with FindAllFiles  
FindAllFiles(PROGRAMFILES, "isdev.exe", svFilepath, CANCEL);  
end;
```

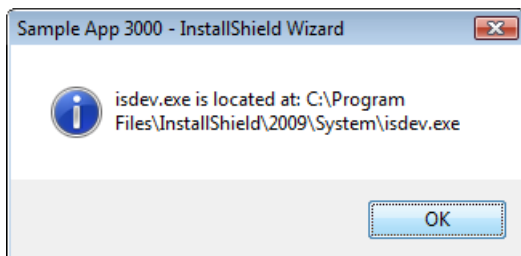


Figure 4-8: Message box showing the results of FindAllFiles

If you want to generate a list of every location of a file with a particular name, see the **FindAllFiles** example in the InstallScript Language Reference. Furthermore, if you want to search every available hard drive for a file with a specific name, you can construct a list of drive letters with **GetValidDrivesList**, and then loop over the drive letters with calls to **FindAllFiles**.

If you want to search for a particular folder name, you can use the InstallScript function **FindAllDirs**. This function generates a list of fully qualified folder names where the search folder name has been found.

Working with the Registry

You can sometimes determine if a program is present on the target system by detecting registry keys used by the program. For example, many programs store data in the registry key `HKLM\Software\CompanyName\ProductName\ProductVersion`. To detect if a registry key exists, you can call **RegDBKeyExist**, as follows:

```
// always set root key before calling other RegDB functions  
RegDBSetDefaultRoot(HKEY_LOCAL_MACHINE);  
if (RegDBKeyExist("Software\\ThisCo\\ThisApp") = 1) then  
    MessageBox("ThisApp is on the system...", INFORMATION);  
endif;
```

Another available root key is `HKEY_USER_SELECTABLE`. If you pass `HKEY_USER_SELECTABLE` as an argument to **RegDBSetDefaultRoot**, then subsequent registry function calls will use `HKEY_LOCAL_MACHINE` as the root key if the `ALLUSERS` system variable is nonzero, or `HKEY_CURRENT_USER` as the root key if `ALLUSERS` is `FALSE` (zero).

To read an existing value from the registry, you can use **RegDBGetKeyValueEx**, in which you specify the key and value to read, and which returns the value type, value data, and value size. For example, to read the "RegisteredOrganization" value from the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`, you might use the following code.

```
function OnBegin( )  
    // local variables used by RegDBGetKeyValueEx  
    STRING svOrganization;
```

```

        NUMBER nvType, nvSize;
begin
// set the default root key
RegDBSetDefaultRoot (HKEY_LOCAL_MACHINE);
// store the value data in svOrganization
RegDBGetKeyValueEx (
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion",
    "RegisteredOrganization",
    nvType, svOrganization, nvSize);
MessageBox("RegisteredOrganization = " + svOrganization, INFORMATION);
end;

```

Similarly, you can create registry keys using **RegDBCreateKeyEx**, and create (or overwrite) registry values using **RegDBSetKeyValueEx**. Keys and values that you create with these functions are logged for uninstallation, even if the key or value already existed on the target system. To disable uninstallation logging, you can wrap the functions in `Disable (LOGGING)` and `Enable (LOGGING)`.

You can also enumerate the values and subkeys under a given key using **RegDBQueryKeyEx**. You can use **RegDBCoppyKeys** to copy registry key and values from a source registry key to a target registry key and use **RegDBCoppyValues** to copy values from a source registry to a target registry key.

If you want to modify the registry on a remote system, use **RegDBConnectRegistry** to connect to the remote registry and then use the registry functions like **RegDBGetKeyValueEx**, **RegDBSetKeyValueEx**, **RegDBCreateKeyEx**, and so forth, to read, write, and create registry keys and values.

The function **RegDBGetUninstCmdLine** gets the command line for the uninstallation specified in the registry. This function gets the command line that gets executed when an end user launches the uninstaller from the Add/Remove Programs tool.

The **RegDBDeleteItem** function deletes values under the per application paths key or the application uninstallation key, depending on the value of `nItem`.

INI Files

To read string data from an existing `.ini` file, you can use the function **GetProfString**. For example, suppose your installation program transfers an `.ini` file called `License.ini` to `TARGETDIR` during the installation, where `License.ini` has the following contents:

```

[LicenseKey]
Serial=ABC-XXX

```

To read the `Serial` value from the `.ini` file (in an event handler such as `OnMoved`), you can use code similar to the following:

```

GetProfString (
    TARGETDIR ^ "License.ini", // file name
    "LicenseKey",             // section name
    "Serial",                  // key name
    svSerial);                // string variable to store value

```

To write data to an .ini file from within a script (creating the file if it does not already exist), you can use **WriteProfString** and **WriteProfInt**. To get the number of keys in a section in an .ini file, you can use the function **GetProfSectionKeyCount**.

Shortcuts and Program Folders

Chapter 2, “Creating an Installation Project” explained how to create program shortcuts in the Shortcuts view of the Installation Designer. As an alternative, you can create shortcuts and program folders using InstallScript functions.

For example, if you want the user to be able to select the folder in which your program’s shortcuts will be installed, you can call the **SdSelectFolder** dialog box near the end of the `OnFirstUIBefore` event handler.

InstallScript provides a system variable `ALLUSERS` that is used to set the profile to All users or current user, based on whether the user has administrative privileges. This system variable supersedes the **ProgDefGroupType** function used in older versions of InstallScript. (The system string variable `SHELL_OBJECT_FOLDER` is provided for storing the user’s folder selection.)

```
if (SYSINFO.WINNT.bAdmin_Logged_On) then
// user has administrative privileges
    ALLUSERS = TRUE;
else
    ALLUSERS = FALSE;
endif;

nResult = SdSelectFolder("", "", SHELL_OBJECT_FOLDER);
```

At run time, the dialog box appears as shown in the following figure.

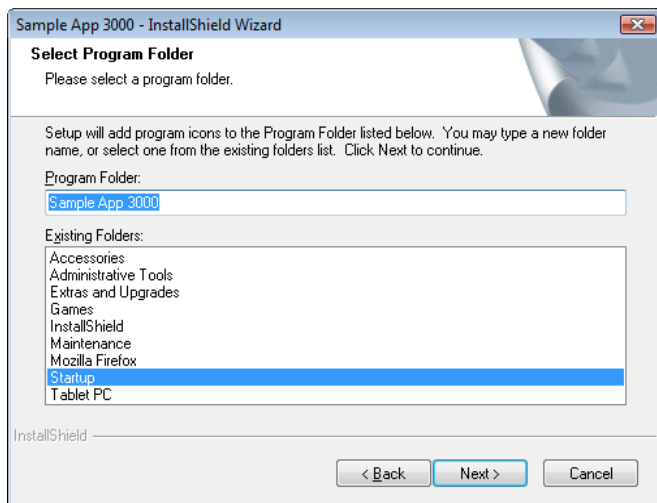


Figure 4-9: The Select Program Folder dialog box displayed as a result of a `SdSelectFolder` call

Next, call **CreateProgramFolder** and **AddFolderIcon** to create the program folder the user selected, and create the shortcut inside the program folder:

```
// create the folder only if it doesn't already exist
if (ExistsDir(FOLDER_PROGRAMS ^ SHELL_OBJECT_FOLDER) = NOTEXISTS) then
```

```

        CreateProgramFolder(SHELL_OBJECT_FOLDER);
    endif;
    szCommandLine = TARGETDIR ^ "SampleApp.exe";
    LongPathToQuote(szCommandLine, TRUE);
    AddFolderIcon(
        SHELL_OBJECT_FOLDER,          // folder
        "Sample App",                 // shortcut display name
        szCommandLine,                // quoted command line
        TARGETDIR,                    // working dir
        "", 0,                         // alt. icon, index
        "", REPLACE);                 // shortcut key, behavior

```

The target application path should be surrounded in quotation marks if it has any spaces; **LongPathToQuote** places quotation marks around a path that contains spaces.

When the user uninstalls your product, the program folder and shortcut will be removed.

Features

The InstallScript functions **FeatureGetData** and **FeatureSetData** enable you to read and modify a feature's properties at run time. For example, you can use **FeatureSetData** to hide or deselect a feature based on characteristics of the target system. You can also use **FeatureSelectItem** to select or deselect a feature.

For example, you might want to hide and deselect a feature containing administrative tools if the user running your installation program does not have administrative privileges. Assuming the feature is called AdminTools, the following code uses **FeatureSetData** to hide and deselect the feature if the user does not have administrative privileges. The name you pass to **FeatureSetData** is the name appearing in the Features view, and not the feature's localizable display name.

```

if (!SYSINFO.WINNT.bAdmin_Logged_On) then
    // hide the feature
    FeatureSetData(MEDIA, "AdminTools",
        FEATURE_FIELD_VISIBLE, FALSE, "");
    // deselect the feature
    FeatureSetData(MEDIA, "AdminTools",
        FEATURE_FIELD_SELECTED, FALSE, "");
endif;

```

You can override a feature's default icon that gets displayed in the feature selection dialog boxes by using the function **FeatureSetData** and passing `FEATURE_FIELD_IMAGE` as one of the arguments.



Note: To refer to a subfeature in an InstallScript function, you should specify the fully qualified path to the feature, as in "FeatureName\SubfeatureName".

Similarly, you can determine if a feature has been selected for installation using the **FeatureItemSelected** function.

As described above, you can specify code that you want to run only if a feature is selected by defining a feature event handler function.

FeatureFileEnum can be used to get a list of files in a component associated with a specific feature, and **FeatureFileInfo** can be used to retrieve information of a file in the media set. For example, assume you have a feature called “Main App” with an associated component “App Executables”; the following code uses **FeatureFileEnum** to get a list of all the files in the component.

```
FeatureFileEnum(  
    MEDIA, "Main App", "App Executables\\*.\"", listFiles, NO_SUBDIR);
```

The following example code retrieves the version number of the file `SampleApp.exe` present in the media set.

```
FeatureFileInfo(  
    MEDIA, "Main App", "App Executables\\SampleApp.exe",  
    FEATURE_INFO_VERSIONSTR, nvResult, svResult);
```

`FeatureSetTarget(MEDIA, szUserVar, szLocation)` enables you to specify a user-defined variable (called a *script-defined folder*) to be used as a component’s Destination setting. For example, suppose you have a component named “Data Files” that you want to be installed to the directory `C:\Data Files`. In the Installation Designer, open the Components view, and for the component’s Destination setting, select the item **Browse, create, or modify a directory entry**. In the Browse for Directory panel, right-click the Script-defined Folders icon, select **New**, and enter the name `<DATAFILESDIR>`.

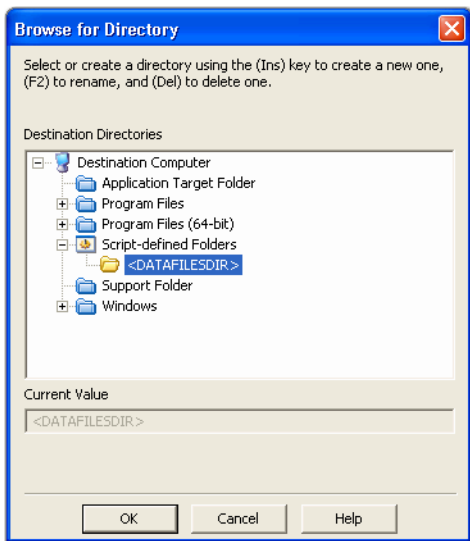


Figure 4-10: The Browse for Directory dialog box

Next, associate the script-defined folder `DATAFILESDIR` with a destination on the target machine. In the script, before performing file transfer, include the following function call:

```
FeatureSetTarget(MEDIA, "<DATAFILESDIR>", "C:\\Data Files");
```

Note that you can use **FeatureSetTarget** to associate user-defined variables with values represented in the Registry view and the Shortcuts view, as well.

As described in the InstallScript Language Reference, InstallScript also provides functions for working with setup types.

Checking System Requirements

InstallScript contains several functions and structures that enable you to detect characteristics of the target system, and therefore test that the target system meets your product's system requirements. Because it is called before any system changes are performed, the `OnBegin` event handler is generally the best event handler for checking system requirements.

Recall that the `OnBegin` event handler is called for both first-time installations and maintenance installations. If you want to perform system-requirement checking only for a first-time installation, you can place your code inside an `if` statement that uses the system variable `MAINTENANCE`, as shown in the following code sample.

```
if (!MAINTENANCE) then
    // ...this is a first-time installation...
endif;
```

For example, if your application requires service pack 3 or later on Windows 2000 systems, you need to end the installation if the target system does not meet the requirement. To abort the installation if this system requirement is not met, you can add the following code to your `OnBegin` event handler.

```
function OnBegin( )
begin
    // abort on first-time installation if user has Windows 2000 with SP < 3
    if (!MAINTENANCE) then
        if (SYSINFO.WINNT.bWinNT && SYSINFO.WINNT.nServicePack < 3) then
            MessageBox(
                "This program requires service pack 3 or later on " +
                "Windows 2000.\n\n" +
                "Setup will now exit.",
                SEVERE);
            abort;
        endif;
    endif;
end;
```

Users and Groups

InstallScript has a collection of dialog boxes that let the user select a user, group, or server existing on the target system, or create a user account on the target system. These functions all begin with `SdLogonUser`, as in `SdLogonUserInformation` and `SdLogonUserBrowse`.

Before using any user-and-group dialog boxes, you must add an extra InstallScript library to the compiler path. To add the library, from the **Build** menu, select **Settings**, select the (default) **Compile/Link** tab, and type the following in the **Libraries** field:

```
<ISProductFolder>\Script\ISRT\lib\NetApiRT.obl
```

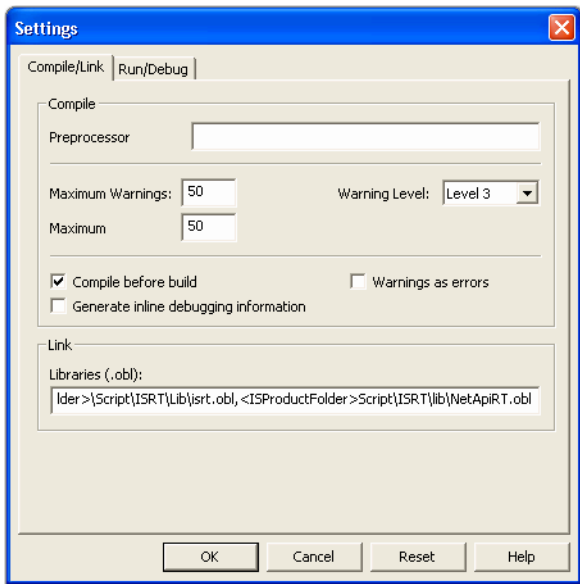


Figure 4-11: The Build Settings dialog box

At run time, for example, the **SdLogonUserInformation** dialog box appears as shown in the following figure.

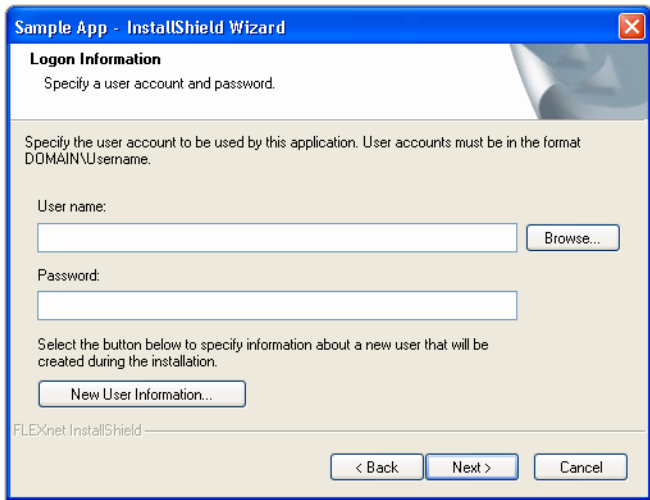


Figure 4-12: The Logon Information dialog box displayed as a result of a call to SdLogonUserInformation

In this dialog box, the end user can type or browse for a domain and user name, type a password, and create a new user account.

Uninstallation

If you make any of the following system changes with InstallScript, they will be logged for uninstallation:

- File transfer with **XCOPYFile** or **CopyFile**
- Shortcuts and program folders created with **AddFolderIcon** and **CreateProgramFolder**
- Registry keys and values created with **RegDB** functions
- Most `.ini` file changes

To specify that certain changes should not be removed when your product is removed, call `Disable (LOGGING)` before making the changes, and call `Enable (LOGGING)` after making the changes. For example, to disable uninstallation logging for a value you create with **RegDBSetKeyValueEx**, you can use code similar to the following:

```
RegDBSetDefaultRoot (HKEY_LOCAL_MACHINE);
Disable (LOGGING);
RegDBSetKeyValueEx (
    "SOFTWARE\MyCompany\MyProduct",
    "MyValue",
    REGDB_STRING, "My Data", -1);
Enable (LOGGING);
```



Note: `Disable (LOGGING)` has no effect on system changes you attach to a component in the InstallShield interface, such as installing files, registry data, and `.ini` file changes. To specify that these types of system changes not be removed when your product is removed, set the **Uninstall** setting to **No** for the component containing the permanent data.

Your installation program caches your compiled script file `Setup.inx` and your InstallScript uninstallation log file (with the `.ilg` extension) in a hidden folder called “InstallShield Installation Information” inside the end user’s Program Files folder. Inside your script, you can refer to this directory using the system variable `DISK1TARGET`.

You can use the function **UninstallApplication** to launch the uninstaller of any installed application that has a subkey under the registry key `<root-key>\Software\Microsoft\Windows\CurrentVersion\Uninstall`. This function first checks for the subkey under `HKEY_CURRENT_USER` and, if it does not find the key there, it checks under `HKEY_LOCAL_MACHINE`.

For installations created with InstallShield Professional 5.53 and earlier, this key is the name of the application. For installations created with InstallShield Professional 6.0 and later (including InstallShield X and later), this key is the product GUID surrounded by braces `{}`. This function must not be used to uninstall the product being installed.

For more information about uninstallation, see [Chapter 5, “Special Considerations”](#).

