Unreal Packages

Tim Sweeney
Epic MegaGames, Inc.
http://www.epicgames.com/

Audience: Level Designers, UnrealScript Programmers, C++ Programmers.
Last Updated: 07/21/99

About this document

This document has been written with our licensees in mind (Microprose, Ion Storm, Legend, and others). However, much of the info is useful to the Unreal community, so I am releasing it here unedited. The sections that refer to C++ code are not relevent to the community because we aren't releasing the C++ source. But, there is still a lot of useful information here.

Overview

An Unreal "package" is a collection of related objects such as textures, sound effects, and scripts which are grouped together for convenient distribution and use.

For example, in Unreal, textures are stored in texture package files, which use the ".utx" extension. A .utx file contains many (tens or hundreds) of textures which are part of a similar theme, such as the "SkyCity.utx" texture package.

Unreal's file format is based on packages because packages enable the game engine and content to be developed modularly. Artists maintain their textures in .utx files; the programmers working on the engine and game maintain separate packages of UnrealScript classes; people in the Unreal community can create and distribute their own packages online; and so on.

One package may refer to objects which are contained in another package. For example, an .unr level package may refer to textures which reside in various external .utx files; sounds in other .uax files; and music in a .umx file. We say that the level file is "dependent" on the other packages. To load that .unr level file successfully, you need to have all of the other .utx, .uax, and .umx files which that level refers to. If one of those packages is missing, you will not be able to load the .unr file.

Package files

All Unreal package files are stored in the same binary file format, but they use different extensions to identify their type:

- .utx files are packages of textures.
- .uax files are packages of audio (sound effects).
- .umx files are packages of music.
- .unr files contain one game level.
- .u files are packages containing UnrealScript classes and objects (like textures, sounds, and meshes) they contain.

When the Unreal engine needs to load a package, it looks in the subdirectories listed in the [Core.System] section, in the Path#= list. The relevent lines of the clean (unmodified) Unreal.ini file look like this:

```
[Core.System]
CachePath=..\Cache
CacheExt=.uxx
Paths[0]=..\System\*.u
Paths[1]=..\Maps\*.unr
Paths[2]=..\Textures\*.utx
Paths[3]=..\Sounds\*.uax
Paths[4]=..\Music\*.umx
```

When you try to load a package by its raw name (such as "SkyCity"), the engine first searches in the CachePath for a matching package downloaded from the Internet; then it searches in each of the numbered directories in order (0, 1, 2...) for matching files. For example, it looks for \Unreal\Cache\[some funky name].uxx, then \Unreal\System\SkyCity.u, then \Unreal\Maps\SkyCity.unr, then \Unreal\Textures\SkyCity.utx -- and the file \Unreal\Textures\SkyCity.utx does exist, so it uses that file.

Because of this storage and search scheme, you must not create two packages with the same base name, which only differ by extension. For example, you must not create a texture file named SkyCity.utx and a music file named SkyCity.umx. If you do that, the engine will always fail to load one of the packages and give an error.

All Unreal package files (.utx, .uax, .umx, .unr, and .u) use the exact same file format, and are processed in exactly the same way by the engine. The only reason we define different extensions is for convenience, and to interoperate with Windows cleanly. For example, .unr files are known to contain levels, so we associate the Unreal icon with them, and set them up so right-clicking on an .unr file in Windows launches Unreal. And, UnrealEd knows that .utx files contain textures. The meanings of these file extensions are purely superficial. They all use the same file format.

Internet package file downloading and caching

Whenever you save a package file to disk, the file is internally stamped with a GUID ("Globally Unique Identifier"), which is a 128-bit number which is guaranteed to be unique. In single-player play, this GUID is never used and is irrelevent. However, when you connect to an Internet-based server, the server sends a list of packages that are required for gameplay there (for example, the level it uses, any texture packages it uses, any sound packages, etc). These packages are identified by name and GUID. If you don't have one of the packages required by the server, the package file is automatically downloaded to your \Unreal\Cache directory, and given a filename based on its GUID.

A typical \Unreal\Cache directory looks something like this:

```
Directory of F:\Unreal\Cache

06/02/98 03:37a <DIR> .
06/02/98 03:37a <DIR> ..
06/02/98 03:37a 8,932 2306087A11D1E2FD4F00DBA5CAA00349.uxx
06/02/98 03:38a 8,932 2401042A11D1E2FD4F00DBA5CAA00381.uxx
3 File(s) 8,932 bytes
698,601,472 bytes free
```

Packages stored in the \Unreal\Cache directory are identified by their GUID rather than the package name given to them by their creator, to avoid conflicting names. For example, there are bound to be a ton of levels on the net called Test.unr, a ton of textures named Walls.utx, etc. The GUID scheme avoids conflicts.

In UnrealEd, each time a package file is saved, it is assigned a new GUID. This way, each version of a package is guaranteed to be tracked and cached separate from other versions. This is good, because it prevents people from trying to play on a server using different versions of a level. This is bad, because if you simply load a texture file, resave it, and copy it to your UnrealServer, all of the players who come to your server will have to re-download the package. Therefore, we recommend modifying your package files only when necessary, and releasing them only after enough testing and use that you know they're not going to change and force redownloading a day later -- especially with large package files.

Why Unreal is based on packages

- Packages enable an open plug-in architecture for easy community and licensee extensibility.
- Packages make the Unreal engine expandable in content, UnrealScript code, and C++ code.
- Packages help to improve the modularity of the code and reduce interdependencies.
- Packages make it much easier for licensees to separarate the Unreal engine from the Unreal game.
- Packages simplify Web-style downloading of content in Internet play.

Standard class packages that ship with Unreal

Core	The high-level framework Unreal is built on, similar to the MFC core or the Java Virtual Machine. The Core package contains the definitions of object, class, subsystem, script, property, name, and the object manager. The core is indepenent of any notion of a game or 3d engine.
Engine	The Unreal 3d engine, including support for vector math, actors, physics, collision, interaction, cameras, and networking.
Editor	The UnrealEd 3d editing environment with geometry editing, realtime lighting, and actor editing support.
Render	The Unreal rendering engine implementation.
Window	The platform-specific code supporting Unreal.
Galaxy	The implementation of Unreal's sound system (UAudioSubsystem) running on top of the Galaxy sound system for Windows. It is possible (for example, when porting Unreal to another platform) to completely replace the Galaxy package with an implementation of another sound system.
GlideDrv	Unreal 3d hardware support code for 3dfx's Glide library.
UnrealI	The Unreal I game implementation.
WinDrv	Windows viewport and client driver.
SoftDrv	Software rendering driver.
Fire	Fractal fire and water effects.
IpDrv	TCP/IP networking driver.

Licensees: Unreal package usage

Licensees may use all our .u packages except UnrealI verbatim or you may modify them to fit your needs. Bear in mind that it is attractive to use them in an unmodified form, as that makes merging with future code updates much easier and makes it easy to isolate problems in each others' code.

Licensees may use the UnrealI package as a reference and for internal development use, but the content in UnrealI is specific to our particular game, and it mustn't be used in your final product.

Licensees should create one more more new packages, for example "WheelOfTime" for their game-specific content. Keep in mind that your package and UnrealI can exist side-by-side during your development cycle until you've fully migrated away from the UnrealI specific content.

Licensee programmers: Sample directory layout for the "Engine" package

Here is an example of the directory structure for all the files that are related to the "Engine" package. The other packages follow a similar structure.

Distributable files:

\Unreal\System\Engine.dll	The package's C++ DLL file.
\Unreal\System\Engine.u	The package's UnrealScript class file.

Developer files:

\Unreal\Engine\Inc\Engine.h	Main public header file for the engine package.
\Unreal\Engine\Inc\*.*	Various public header files for the engine package, included by Engine.h.
\Unreal\Engine\Src\EnginePrivate.h	Private implementation header file.

\Unreal\Engine\Src\EngineClasses.h	Scripted classes exported to C++.
\Unreal\Engine\Src\Engine.cpp	Generates the package's precompiled header file.
\Unreal\Engine\Src\*.*	Various C++ files comprising the engine's code.
\Unreal\Engine\Lib\Engine.lib	The export library for linking with the engine.
\Unreal\Engine\Classes\*.uc	The UnrealScript classes used in building Engine.u.
\Unreal\Engine\Textures\*.*	Graphics files used in building Engine.u.
\Unreal\Engine\Models\*.*	Model files used in building Engine.u.
\Unreal\Engine\Sounds\*.*	Sound files used in building Engine.u.

Licensee programmers: Associating packages and DLL's

When you declare an UnrealScript class as "intrinsic", the engine looks in the .DLL file corresponding to the class's package for the class's corresponding C++ code. For example, here is the declaration of the "PlayerPawn" class in the "Engine" package:

```
//=============================================
// PlayerPawn.
//=============================================
class PlayerPawn expands Pawn
        intrinsic;
```

(You can find the above code in the file \Unreal\Engine\Classes\PlayerPawn.uc.)

When you declare a class as "intrinsic" in UnrealScript, you are telling the engine that you have created some corresponding C++ code for the class. In this way, you can mix and match UnrealScript and C++ code when you write your game code.

When you declare "intrinsic" classes, Unreal looks for the .DLL file in the \Unreal\System directory whose name matches your package's name. For example, to find the C++ implementation of the PlayerPawn class, Unreal looks in the "\Unreal\System\Engine.dll" file.

In C++, you must use the IMPLEMENT_INTRINSIC macro (defined in UnObjBas.h) to tell the engine that you have implemented a new class in C++. Internally, this macro generates some __declspec(DLL_EXPORT) declarations that Unreal looks for in your DLL. For example, the PlayerPawn class is exposed in the source file "\Unreal\Engine\UnPawn.cpp" as follows:

```
IMPLEMENT_CLASS(APlayerPawn);
```

In addition, once per package you need to use Unreal's package declaration macro to expose the package:

```
IMPLEMENT_Package(Engine);
```

So, to summarize:

- Each Unreal package corresponds to a .u file, for example, "\Unreal\System\MyPackage.u".
- If one or more classes in the package are "intrinsic" classes, then Unreal expects to find a C++ .DLL file corresponding to the package, for example, "\Unreal\System\MyPackage.dll".
- Unreal dynamically loads and unloads packages and their optional, corresponding .DLL files, on the fly -- they are loaded and unloaded based on whether they are needed in the current game level.

Licensee programmers: Running the game and editor without the UnrealI.u package

You can start working with the "baseline" version of the Unreal engine (that is, the engine with all game-specific content removed), with:

```
        unreal.exe level.unr?class=engine.playerpawn?class=engine.gameinfo
```

While you're playing with the baseline configuration, you won't be able to see the player model and you won't be able to use any weapons because, of course, all of those things are game-specific and stored in the UnrealI package. But, the point is, you can edit and play levels with only the baseline engine, without any of our game-specific content.

See below for a description of what the Unreal.ini [EditPackages], [DefaultGame], and [DefaultPlayer] settings mean.

Licensee programmers: Replacing UnrealI.u with your own game-specific package

The first question is, "How do I create my own game package like your UnrealI.u?"

The easiest way to create a new package is by creating your first new class within UnrealEd. When creating a new game using the Unreal technology, you will need to create a custom "GameInfo" class to define high-level gameplay rules. Here's how:

1. Run UnrealEd
2. Bring up the "Classes" browser on the right-hand side of the screen.
3. Expand the "Info" class.
4. Right-click the "GameInfo" class.
5. Pick the "Create new class below GameInfo..." menu option.
6. Give the new class a name, for example "KlingonsGameInfo" and a package, for example "Klingons".
7. Press F7 (or use the Script/Compile menu option) to recompile the new script.
8. Save your package's .u binary file by hitting the "Save" button in the class browser and selecting your package.
9. At the bottom of the class browser, hit the "Export" button to export all classes which you have modified. This generates *.uc files in a directory corresponding

to your package. For example, if you just created a class named "KlingonsGameInfo" in a package called "Klingons", UnrealEd will generate the file "\Unreal\Klingons\KlingonGameInfo.uc". The .uc files are pure text versions of your scripts, which can later be cleanly rebuilt.

The next question is, "How do I have my new package used in the editor and during gameplay?" The Unreal.ini file defines many of these project-specific settings you will want to modify for your title:

- The [EditPackages] section lists the class packages (.u files) to load for editing when UnrealEd is run. You will generally want to remove UnrealI from the list, and add your package(s) to the list, so that your level designers can see all of your content from within UnrealEd.
- The [DefaultGame] section says which GameInfo-derived class is responsible for moderating gameplay. The GameInfo class (see \Unreal\Engine\GameInfo.uc) makes many high-level gameplay decisions in single-player and network play, i.e. it implements the game rules. For the baseline game, use Engine.GameInfo. To customize the game rules, you must create a subclass of GameInfo.
- The [DefaultPlayer] section says which PlayerPawn-derived class should be spawned to represent the player in the game. For the baseline game, use Engine.PlayerPawn. To customize the player controls, mesh, etc., create a subclass of PlayerPawn.

Licensee programmers: Working with classes and packages in UnrealEd

The most important thing you need to realize while working with classes is that each class may be stored redundently in two separate places:

- A binary copy of the class is stored in its package's .u file in the \Unreal\System directory. This binary copy of the class includes all objects referenced by the class. For example, the Skaarj class in the UnrealI package is stored in the file \Unreal\System\UnrealI.u. This binary version includes the Skaarj's mesh and all of the monster's sound effects. The binary copy of the class must be saved by using the "Save" button in the class browser. However, the binary file will break whenever the .u file format changes, which will happen several times prior to the release of Unreal.
- For licensee programmers who have the entire Unreal C++ and UnrealScript codebase in standalone form, a text copy of the class is stored in one .uc file. For example, the text version of the Skaarj class is stored in \Unreal\UnrealI\Classes\Skaarj.uc. You can save these text files with the "Export" button in the class browser. The .uc file is a text file, so it doesn't contain any binary data. Rather, it contains UnrealScript "#exec" commands to import all ancillary objects like meshes, sound effects, and textures, when you perform a clean rebuild.

There are two ways you can create and edit UnrealScript classes:

- You create new classes in UnrealEd by right-clicking on an existing class and subclassing it (by choosing the "Create new class below..." menu option), and then use the built-in script editor and compiler. To edit an existing class, just double-click on it. Use Script/Compile (or F7) to recompile your script.
- You can create new classes from outside UnrealEd by creating a new .uc file in the package's directory. For example, to add a new class named "Timster" to the "UnrealI" package, create a new file called "\Unreal\UnrealI\Classes\Timster.uc". This way, you can use your favorite stand-alone editing tool, such as the text editor built into Visual C++, to edit your script. However, when you manually work with .uc files, you must cleanly rebuild your package in order to regenerate the binary .u file which UnrealEd requires.

To summarize:

- You can create new classes in UnrealEd using the built-in editor, or you can work directly with the .uc files using your favorite stand-alone editor.
- There is one text .uc for each class. Think of this as your source code.
- There is one binary .u file for each package. Think of this as your object code.

Programmers: Cleanly rebuilding your packages from the command line

You can rebuild individual .u files, or all of the .u files, from the command line as follows:

1. Delete the .u file(s) which you want to rebuild. For example, type "del \Unreal\System\*.u".
2. Type "Unreal -make" to run Unreal and rebuild the .u files.
3. Now, Unreal will go through all of the packages listed in the Unreal.ini [EditPackages] section, and rebuild any packages whose .u files are missing.

When Unreal rebuilds a package, it switches into the \Unreal\packagename\Classes directory (for example, if you're rebuilding the Engine package, it switches into \Unreal\Engine\Classes), and it imports all of the *.uc files it finds there.

The .uc files are text files containing UnrealScript code. There is one .uc file for each class. The .uc filename must correspond to the class name. For example, the PlayerPawn class must reside in the file "PlayerPawn.uc".

The .uc files may contain special UnrealEd commands which begin with the keyword "#exec". These "#exec" commands import various objects as the script is compiled. For example, #exec commands exist to import textures, sounds, meshes, and music. The "#exec" commands are not currently documented; you will need to see the example .uc files in the UnrealI package for examples of usage.

Rebuilding certain packages, such as the Engine package and the UnrealI package, requires that you have the source artwork, sounds, and meshes, which are available on the UnEdit ftp site as a separate download. The source data is not included in the regular source distribution because of its huge size. The source artwork can be extracted with WinZip into the various subdirectories of the Unreal\Engine\ and \Unreal\UnrealI\ directories (for example, the engine sounds go into \Unreal\Engine\Sounds).

When cleanly rebuilding, any UnrealScript compiler errors cause the rebuild to halt, and an appropriate error message is displayed on the screen.

To summarize:

- Using "Unreal -make" cleanly rebuilds any missing .u files.
- All .uc files (UnrealScript class definitions) are imported and recompiled.
- The .uc files may contain "#exec" commands to import ancillary content.
- The end result is that a new .u file is generated for each missing package.

Package file attributes: .upkg Files

When Unreal rebuilds your package using "Unreal -make", it tries to load a text PackageName.upkg definition file from the same \Unreal\PackageName\Classes directory. If not found, the default package options are used. A sample package definition file looks like this:

```
[Flags]
AllowDownload=False
ClientOptional=False
ServerSideOnly=True
```

The options that appear in a package definition file are:

- AllowDownload: Whether you want players to be able to download this file when they enter a server. This should be enabled for user-created modifications, and disabled for game publisher distribution (so that people aren't able to download retail files freely).
- ClientOptional: Whether the package is optional on the client side. Not currently supported; may be supported in the future for things like player skins.
- ServerSideOnly: Says the package should only be loaded on the server side, and is not necessary for clients. For example, the IpDrv.u package is tagged as ServerSideOnly, because it contains classes like the server-uplink which only need to run on the server, and will be frequently updated.

Licensee programmers: Generating C++ header files which mirror your UnrealScript classes

If you create any intrinsic UnrealScript classes, or if you need to access any UnrealScript classes from C++ code, then you can have Unreal generate a C++ header file which mirrors the classes you've defined in UnrealScript.

For example, the Actor class any many of its child classes are defined by scripts in the Engine package (such as \Unreal\Engine\Classes\Actor.uc). The C++ code in the engine often needs to access objects in the Actor class. Early in development, we manually created C++ definitions which mirrored all of the UnrealScript classes. This process became tedious and error-prone as the number of classes increased. So, we added an option to Unreal to automatically generate a C++ header file, based on your UnrealScript definitions.

For an example of an automatically-generated C++ header file, check out "\Unreal\Engine\Inc\EngineClasses.h". This file was not written by a human. Unreal generated it.

To automatically generate a C++ header for a package, do the following:

1. Delete the package's .u file. For example, type "del \Unreal\System\Engine.u".
2. Run Unreal with the following parameters: "Unreal -make -h".
3. Unreal will create a new file for you, called "\Unreal\Engine\Inc\EngineClasses.h".
4. Now recompile the C++ source code.

Because there are many ways which UnrealScript and C++ can share information back and forth, the automatically-generated header file (for example EngineClasses.h) contains a variety of definitions:

- The enumerations defined in the package.
- C++ class definitions for all the intrinsic classes defined in the package. Note that non-intrinsic classes are not exported to the header file, because you can only access intrinsic classes from C++. (Note: this does not in any way limit what you can do in UnrealScript vs. C++).
- For each class, C++ class member functions for all intrinsic functions defined in the class.
- For each class, a C++ "#include" statement if a class-specific header file exists (for example, the Unreal generated C++ definition of the Actor class contains the #include "AActor.h" statement because the file \Unreal\Engine\Inc\AActor.h exists.

For a more concrete example, carefully examine the UnrealScript file "\Unreal\Engine\Classes\TcpLink.uc" and the following definitions in the file \Unreal\Engine\Inc\EngineClasses.h": enum ETcpLinkState, enum ETcpMode, and class ATcpLink. This provides an example of a fairly isolated UnrealScript class which defines several enumerations and intrinsic functions.

Finally, to avoid a chicken-and-egg scenario, you should create new classes in the following order: first create a class in UnrealScript, then generate the C++ header, then recompile the engine code (adding in the IMPLEMENT_CLASS macro for any new intrinsic classes you've added).

To summarize the important points here:

- You created a new class in UnrealScript.
- You needed to access that new class from C++.
- You used the "Unreal -make -h" command to automatically generate a C++ header file defining the package's classes.

Programmers: Package related problems to avoid

- Don't remove objects from packages which other people are relying on. For example, say level designers are using a texture package like SkyCity.utx which contains a texture named "SkyWall2" which they're using in many of their levels. If you go through and remove that texture from the .utx file, then the level designers won't be able to load their levels successfully. Rather they will get an error message along the lines of "Texture SkyWall2 was not found in package SkyCity.utx".
- Avoid package filename collisions. Packages are identified by their base (extension-less) names, so you should node create a texture package named "Fred.utx" and a level named "Fred.unr". The base filename of each package must be unique.
- Avoid circular package dependencies in .u files. For example, if you create two packages, "MyPackage" and "YourPackage" which both contain UnrealScript classes that refer to the other package, it will be impossible to cleanly rebuild both packages. During the clean rebuild process, packages are imported in the order they are specified in the Unreal.ini [EditPackages] section, and each package may only be dependent on the packages listed before it.
- If you delete an UnrealScript, you must manually remove the .uc file corresponding to it. When you do a clean rebuild, Unreal imports all of the *.uc files in each package's directory.

Licensee programmers: Putting .uc script files under source control with SourceSafe

For programmers who like to keep their work archived under Microsoft Visual SourceSafe, here are some tips. We use SourceSafe internally for our development, though it is purely optional -- nothing in the Unreal source code assumes that you have SourceSafe:

- Put all of your .uc files under source control (.uc files are text files containing the text of the UnrealScript code).
- If someone removes a .uc file from source control, be sure to delete the corresponding .uc file on disk.
- In the UnrealEd class browser, the "Export" button exports all scripts which you've modified. If you try to export a script which you haven't checked out of SourceSafe, you will get a (safe, non-fatal) warning message that the .uc file is write-protected. If that happens, just check the .uc file out of SourceSafe and hit the "Export" button again.

Programmers: Adding/Removing variables in UnrealScript classes can be dangerous

When you add or remove variables in an UnrealScript class, that changes the binary layout of objects belonging to that class. For example, if you have a script that contains some variable declarations like this:

```
class TestClass expands Actor;

var int i, j;
var texture t;
```

And you remove one of the variables like this:

```
class TestClass expands Actor;
var int i;
var texture t;
```

Then any objects in memory belonging to TestClass will become corrupted as soon as you recompile the script. The solution is to only add or remove variables from a script in an empty level. However, empty levels contain an actor belonging to the LevelInfo class, as well as one or more Camera actors, so it's never safe to add or remove variables from any of the following classes in UnrealEd: Actor, Info, LevelInfo, Camera, PlayerPawn, Pawn. To work with these files, edit their .uc files on-disk, and do a clean rebuild.

Also, bear in mind that Unreal makes heavy use of the Actor, Pawn, PlayerPawn, LevelInfo, and other intrinsic classes in C++ code. So, adding and removing variables to any intrinsic classes will cause the C++ code and UnrealScript code to disagree about the binary layout of classes. Whenever you change an intrinsic class, the safe thing to do is a clean rebuild with the "Unreal -make -h" option to generate new C++ header files for your classes, then recompile the Unreal C++ source.

Programmers: The package file format

Abandon all hope ye who try to parse this file format.

The Unreal package file format is only useful to people who want to write external utilities that read and write Unreal files. Level designers and UnrealScript programmers don't need to know this info. But, the Unreal package file format is quite complex compared to what Quake, Quake 2, and Jedi Knight utility writers are accustomed to. It's so complex, in fact, that I doubt many utility writers are going to want to deal with it directly. It is complex for several reasons:

- Unreal files can contain a wide range of objects (there are hundreds of unique classes defined in Unreal), versus perhaps 20-30 for Quake.
- Unreal objects can contain complex interdependencies: for example, a level package can refer to a texture package, which can refer to a class package, which can contain a class that refers to a sound in a sound package. So, each package file needs to keep track of not only the objects it contains, but also the external objects that it refers to.
- Unreal objects are expandable by users, so we needed to create a generalized system for serializing (a.k.a. saving and loading) these complex user-defined objects.
- Unreal objects are scoped within hierarchical packages (much like DOS and Windows files are scoped within a system of hierarchical directories). Therefore, the entire tree structure of the object scope hierarchy must be described within the file.

The coming release of the enhanced version of UnrealEd in 3+ months will include a very modular plug-in interface for creating built-in UnrealEd tools, file importing tools, and file exporting tools, which will operate directly on the engine's C++ data. That release will greatly simplify the task of programmers who want to write Unreal utilities, because it eliminates the need to read and write package files directly. Rather, with the UnrealEd plug-in interface, you will be able to use C++ to create all of your tools and compile them into DLL's. You'll make calls to the Unreal engine to allocate objects, save package files, load package files, etc. with the hairy file-format issues taken care of by the engine.

Data types stored in package files.

- The "int" type is signed, 4 bytes long, and is stored in Intel byte order.
- The "float" type is signed, IEEE standard, 4 bytes long, and stored in Intel byte order.
- The "string" type is stored as a bunch of characters (one nonzero byte each) terminated by a zero byte.
- The "FCompactIndex type" corresponds to a 32-bit signed value but are stored in a compacted format. See below for a description.
- The "FGuid" type corresponds to a set of four consecutive int's defining a globally unique identifier.

Object and Name flags.

Each object and name stored in a package file can contain any combination of the following bitflags:

- RF_LoadForClient (0x00010000): Must be loaded for game client.
- RF_LoadForServer (0x00020000): Must be loaded for game server.
- RF_LoadForEdit (0x00040000): Must be loaded for editor.
- RF_Public (0x00000004): Object may be imported by other package files.
- RF_Standalone (0x00080000): Keep object around (don't garbage collect) for editor even if unreferenced.
- RF_Intrinsic (0x04000000): Class or name is defined in C++ code and must be bound at load-time.
- RF_SourceModified (0x00000020): The external data source corresponding to this object has been modified.

- RF_Transactional (0x00000001): Object must be tracked in the editor by the Undo/Redo tracking system.
- RF_HasStack (0x02000000): This object has an execution stack allocated and is ready to execute UnrealScript code.

Any other bitflags stored in package files are irrelevent, should be set to zero when saving a package file, and should be ignored when loading a package file.

Compact Indices.

Compact indices exist so that small numbers can be stored efficiently. An index named "Index" is stored as a series of 1-5 consecutive bytes with the following C++ code. Basically, the "Ar << B0" type code serializes the byte stored in the variable B0. Serialize can mean read or write, depending on the internal implementation of the archive object Ar.

```
//
// FCompactIndex serializer.
//
FArchive& operator<<( FArchive& Ar, FCompactIndex& I )
{
        INT Original = I.Value;
        DWORD V = Abs(I.Value);
        BYTE B0 = ((I.Value>=0) ? 0 : 0x80) + ((V < 0x40) ? V : ((V & 0x3f)+0x40));
        I.Value = 0;
        Ar << B0;
        if( B0 & 0x40 )
        {
            V >>= 6;
            BYTE B1 = (V < 0x80) ? V : ((V & 0x7f)+0x80);
            Ar << B1;
            if( B1 & 0x80 )
            {
                V >>= 7;
                BYTE B2 = (V < 0x80) ? V : ((V & 0x7f)+0x80);
                Ar << B2;
                if( B2 & 0x80 )
                {
                    V >>= 7;
                    BYTE B3 = (V < 0x80) ? V : ((V & 0x7f)+0x80);
                    Ar << B3;
                    if( B3 & 0x80 )
                    {
                        V >>= 7;
                        BYTE B4 = V;
                        Ar << B4;
                        I.Value = B4;
                    }
                    I.Value = (I.Value << 7) + (B3 & 0x7f);
                }
                I.Value = (I.Value << 7) + (B2 & 0x7f);
            }
            I.Value = (I.Value << 7) + (B1 & 0x7f);
        }
        I.Value = (I.Value << 6) + (B0 & 0x3f);
        if( B0 & 0x80 )
            I.Value = -I.Value;
        if( Ar.IsSaving() && I.Value!=Original )
            appErrorf("Mismatch: %08X %08X",I.Value,Original);
    }
    return Ar;
}
```

Names

Names are stored a compact indices with values >= 0 which index into the file's name table.

Objects References.

Objects are stored as compact indices with values defined as follows:

- If Index==0: The object is NULL (known as NULL in C++, None in UnrealScript).
- If Index<0: Refers to the (-Index-1)th object in this file's import table.
- If Index>0: Refers to the (Index-1)th object in this file's export table.

The Package file header.

- int Tag: Always 0x9E2A83C1 (see PACKAGE_FILE_TAG in UnObjVer.h).
- int FileVersion: Version of the engine which saved the file. Currently 61. Utilities should only operate on files with this exact version number, because only the file header is guaranteed not to change in future versions...anything else could change (See PACKAGE_FILE_VERSION in UnObjVer.h).
- int PackageFlags: Bitflags describing the package:
  - PKG_AllowDownload (0x0001): The package is allowed to be downloaded to clients freely.
  - PKG_ClientOptional (0x0002): All objects in the package are optional (i.e. skins, textures) and it's up to the client whether he wants to download them or not. Not yet implemented; currently ignored.
  - PKG_ServerSideOnly (0x0004): This package is only needed on the server side, and the client shouldn't be informed of its presence. This is used with packages like IpDrv so that it can be updated frequently on the server side without requiring downloading stuff to the client.
- int NameCount: Number of names stored in the name table. Always >= 0.
- int NameOffset: Offset into the file of the name table, in bytes. 0 designates the first byte of the file, of course.

- int ExportCount: Number of exported objects in the export table. Always >= 0.
- int ExportOffset: Offset into the file of the export table.
- int ImportCount: Number of imported objects in the import table. Always >= 0.
- int ImportOffset: Offset into the file of the import table.
- int HeritageCount: Number of GUID's stored in the package's heritage table. Always >= 1.
- int HeritageOffset: Offset into the file of the heritage GUID table.

The following tables reside in the file at the offset specified by the header. If a table contains zero entries (as specified in the header), the offset value is meaningless and should be ignored.

Name table.

Contains a list of human-readable Unreal names (which correspond to the UnrealScript "name" datatype and the C++ "FName" data type). Each name is stored consecutively in the following format:

- string NameString: String representation of the name, up to NAME_SIZE characters (currently 64, may increase with future versions).
- int NameFlags: Internal flags describing the name. See "Object and Name Flags" above.

Export table.

Contains a list of objects contained in (a.k.a. "exported by") this file. Similar to a Windows DLL's export table.

- object reference ClassIndex: Points to the class object describing the class of this object.
- int PackageIndex: Points to the package object describing the package this object resides in.
- object reference SuperIndex: If this is a field (a struct, class, property, or another field subclass): Points to the superfield object of the field.
- FName ObjectName: This object's name.
- int ObjectFlags: Flags.
- int SerialSize: Size (in bytes) of the object's serialized data stored in this file.
- if SerialSize>=0, int SerialOffset: Offset into this file of the start of the object's serialized data.

Import table.

Contains a list of objects in other packages which this packages refers to. Similar to a Windows DLL's import table.

- name ClassPackage: The name of the package which this object's class object resides in.
- name ClassName: The name of this object's class.
- int PackageIndex: The index of the package this object resides in.
- FName ObjectName: The name of this object.

Heritage GUID table.

Contains a list of GUID's which this package is compatible with. In the current version of Unreal, this table always contains exactly one GUID, because the backwards-compatible package management code has not yet been written. In the future, it may contain more than one GUID. Each entry contains:

- A GUID (4 consecutive int's).

Serialize object format.

Extremely complex. Different for each class. Hard to read. Not yet documented.