Uniform Type Identifiers Overview

Data Management: Data Types & Collections

2009-11-17

Contents

Figures and Tables

Introduction to Uniform Type Identifiers Overview

One of the challenges facing application developers is the proliferation of methods to identify types of data. For example, some text files may be assigned a `'TEXT'` file type (as originally designed for Mac OS 9 and earlier), while others may simply have a `.txt` filename extension. Some may have the `.text` extension instead. In addition, some file types might be subsets of other types; an application that opens all `.txt` files should probably also be able to open those with a `.html` extension. Determining all the possible files an application could read could become impossible. The user experience then suffers, with users not understanding why an application can open one text file but not another.

To solve this problem, Apple has defined a syntax for special data identifiers called **uniform type identifiers**. Each UTI provides a unique identifier for a particular file type, data type, directory or bundle type, and so on. In addition, other type identifier namespaces for a particular type can be grouped under one UTI, with utility functions available to translate from one format to another.

Who Should Read This Document?

This document is for Mac OS X and iOS application developers that need to create or otherwise manipulate data that may be exchanged with other applications or services. For example, applications often need to be aware of the type of data they handle when:

- Displaying, or manipulating, files, bundles, or folders
- Accessing streaming data
- Copying and pasting between documents or applications
- Dragging and dropping between applications

Support for uniform type identifiers is available in Mac OS X v10.3 and later and iOS 3.0 and later.

Organization of This Document

This document is organized into the following chapters:

- "Uniform Type Identifier Concepts" (page 9) describes the syntax and usage of UTIs.
- "Adopting Uniform Type Identifiers" (page 15) describes how to adopt UTIs in your applications.
- "Declaring New Uniform Type Identifiers" (page 19) describes how to declare new UTIs in your applications.

See Also

UTType Reference describes the functions used to manipulate UTIs.

Uniform Type Identifiers Reference provides a description of known UTIs.

Uniform Type Identifier Concepts

Uniform type identifiers (UTIs) provide a unified way to identify data handled within the system, such as documents, pasteboard data, and bundles. This chapter describes the concepts behind UTIs and shows how to specify them in your application bundles.

What Is a Uniform Type Identifier?

A uniform type identifier is a string that uniquely identifies a class of entities considered to have a "type." For example, for a file or other stream of bytes, "type" refers to the format of the data. For entities such as packages and bundles, "type" refers to the internal structure of the directory hierarchy. Most commonly, a UTI provides a consistent identifier for data that all applications and services can recognize and rely upon, eliminating the need to keep track of all the existing methods of tagging data. Currently, for example, a JPEG file might be identified by any of the following methods:

- A four-character file type code (an `OSType`) of `'JPEG'`
- A filename extension of `.jpg`
- A filename extension of `.jpeg`
- A MIME type of `image/jpeg`

A UTI replaces all these incompatible tagging methods with the string `public.jpeg`. This string identifier is fully compatible with any of the older tagging methods, and you can call utility functions to translate from one to the other. That is, for a given UTI, you can generate the equivalent `OSType`, MIME type, or filename extension, and vice versa.

Because UTIs can identify any class of entity, they are much more flexible than the older tagging methods; you can also use them to identify any of the following entities:

- Pasteboard data
- Folders (directories)
- Bundles
- Frameworks
- Streaming data
- Aliases and symbolic links

In addition, you can define your own UTIs for application-specific use. For example, if your application uses a special document format, you can declare a UTI for it. Third-party applications or plug-ins that want to support your format can then use that UTI to identify your files.

The UTI Character Set

A uniform type identifier is a Unicode string that usually contains characters in the ASCII character set. However, only a subset of the ASCII characters are permitted. You may use the Roman alphabet in upper and lower case (A–Z, a–z), the digits 0 through 9, the dot ("."), and the hyphen ("-"). This restriction is based on DNS name restrictions, set forth in RFC 1035.

Uniform type identifiers may also contain any of the Unicode characters greater than U+007F.

Important: Any illegal character appearing in a UTI string—for example, underscore ("_"), colon (":"), or space (" ")—will cause the string to be rejected as an invalid UTI. At the API layer, no error is generated for invalid UTIs.

The UTI Syntax

Uniform type identifiers use the reverse-DNS format initially used to describe elements of the Java class hierarchy and now also used in Mac OS X and iOS for bundle identification. Some examples:

```
com.apple.quicktime-movie
com.mycompany.myapp.myspecialfiletype
public.html
com.apple.pict
public.jpeg
```

The UTI syntax ensures that a given identifier is unique without requiring a central authority to register or otherwise keep track of them. Note that the domain (`com`, `public`, and so on) is used only to identify the UTIs position in the domain hierarchy; it does not imply any grouping of similar types.

- The `public` domain is reserved for common or standard types that are of general use to most applications:

  ```
  public.text
  public.plain-text
  public.jpeg
  public.html
  ```

  UTIs with the `public` domain are called public identifiers. Currently only Apple can declare public identifiers.

- The `dyn` domain is reserved for special dynamic identifiers. See "Dynamic Type Identifiers" (page 13) for more information.

- All other domains are available for use by third parties. Typically, identifiers declared by companies will begin with the `com` domain.

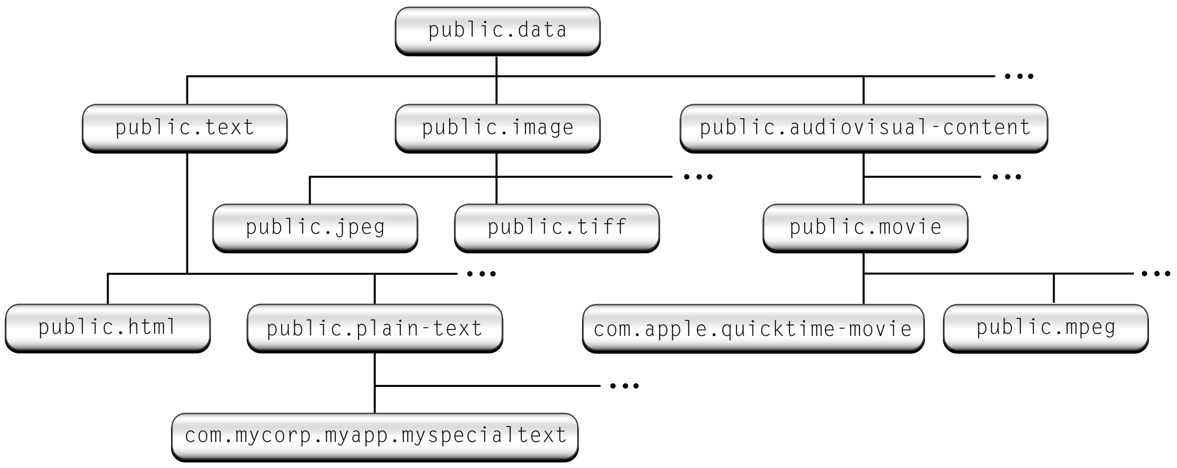
  ```
  com.apple.quicktime-movie
  com.yoyodyne.buckybits
  ```

Conformance

A key advantage of uniform type identifiers over other type identification methods is that they are declared in a conformance hierarchy. A conformance hierarchy is analogous to a class hierarchy in object-oriented programming. Instead of "type A conforms to type B," you can also think of this as "all instances of type A are also instances of type B."

Figure 1-1 (page 11) shows a conformance hierarchy for some uniform type identifiers.

Figure 1-1 A conformance hierarchy



For example, the UTI `public.html`, which defines HTML text, conforms to the base text identifier, `public.text`. In this case, conformance lets applications that can open general text files identify HTML files as ones it can open as well.

You need to declare conformance only with your type's immediate "superclass," because the conformance hierarchy allows for inheritance between identifiers. That is, if you declare your identifier as conforming to the `public.tiff` identifier, it automatically conforms to identifiers higher up in the hierarchy, such as `public.image` and `public.data`.

The conformance hierarchy supports multiple inheritance. For example, the UTI for an application bundle (`com.apple.application-package`) conforms to both the generic bundle type (`com.apple.bundle`) and the packaged directory type (`com.apple.package`).

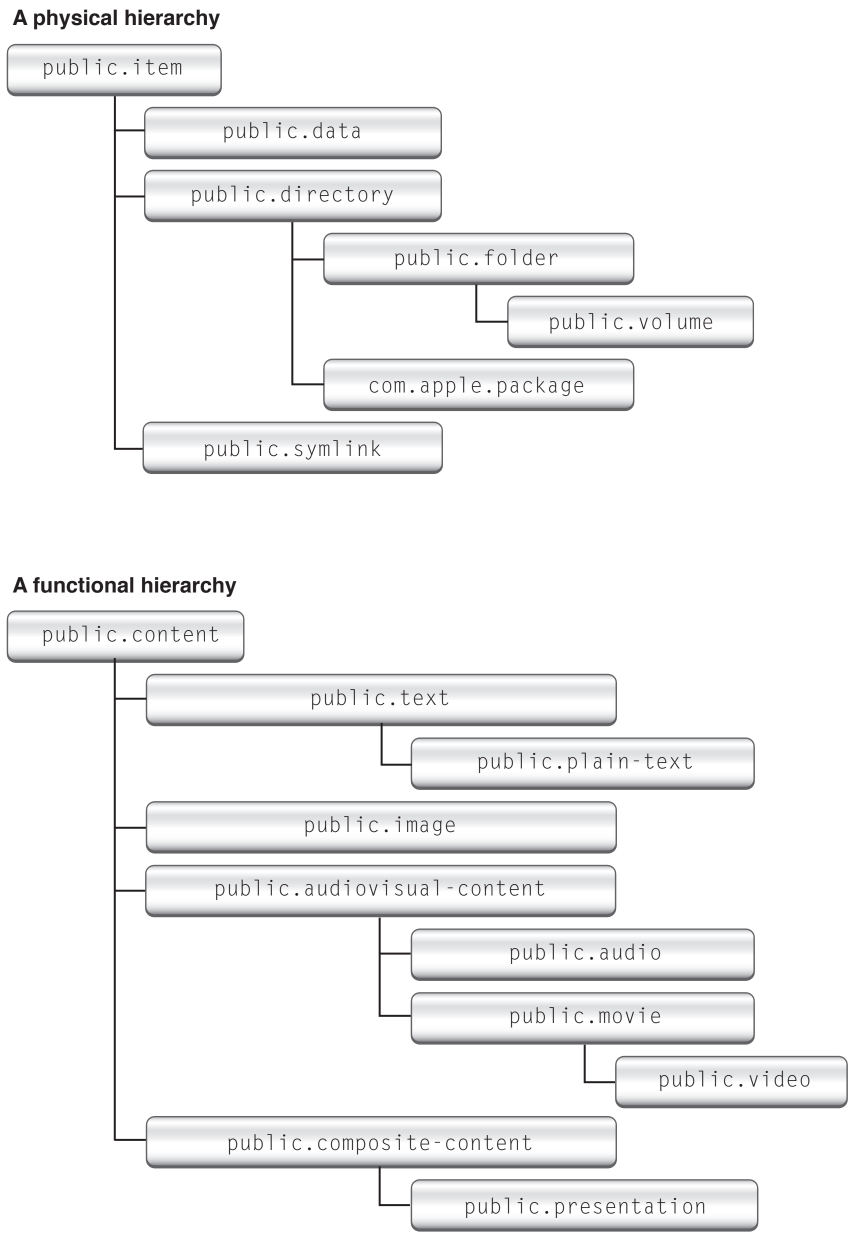
When specifying conformance for your UTI, your items should ideally conform to both a physical and functional hierarchy. That is, the conformance should specify both its physical nature (a directory, a file, and so on) as well as its usage (an image, a movie, and so on).

- A UTI in the physical hierarchy should conform through the inheritance hierarchy to `public.item`.
- A UTI in a functional hierarchy should conform through inheritance to a base UTI that is not `public.item`. For example, `public.content`, `public.executable` and `public.archive` are all examples of functional base UTIs.

While conforming to the functional hierarchy is not mandatory, doing so allows for better integration with system features. For example, Spotlight associates named attributes (title, authors, version, comments, and so on) with functional UTIs.

Figure 1-2 (page 12) shows examples of physical and functional hierarchies:

Figure 1-2 Physical and functional hierarchies



In some cases, you need to declare conformance to only one UTI to cover both hierarchies. For example, `public.text`, `public.image` and `public.audiovisual-content` conform to both `public.data` (physical) and `public.content` (functional), so conforming (directly or indirectly) to one of these items covers both hierarchies.

Conformance gives your application much more flexibility in determining what types it is compatible with; not only do you avoid writing lots of conditional code, your application can be compatible with types that you had never anticipated.

Dynamic Type Identifiers

Sometimes you may run across a data type that does not have a UTI declared for it. UTIs handle this case transparently by creating a dynamic identifier for that type. For example, say your application finds a NSPasteboard type that it does not recognize. Using the utility functions, it can still convert the type to a UTI that it can then pass around.

Dynamic identifiers have the domain `dyn`, with the rest of the string that follows being opaque. You handle dynamic identifiers just as any other UTI, and you can extract the original identifier tag using utility functions. You can think of a dynamic identifier as a UTI-compatible wrapper around an otherwise unknown filename extension, MIME type, `OSType`, and so on.

Identifier Tags

Each UTI can have one or more tags associated with it. These tags indicate alternate methods of type identification, such as filename extension, MIME type, or NSPasteboard type. You use these tags to assign specific extensions, MIME types, and so on, as being equivalent types in a UTI declaration.

For example, the `public.jpeg` identifier declaration includes one OSType tag (`'JPEG'`) and two filename extension tags (`.jpg` and `.jpeg`). These tags are then considered alternate identifiers for the `public.jpeg` type.

Essentially, you use the tags to group all the possible methods of identifying a type under one UTI. That is, a file with extension `.jpg` or `.jpeg`, or an OSType of `'JPEG'` are all considered to be of type `public.jpeg`.

Declaring New UTIs

Mac OS X applications can declare new UTIs for their own proprietary formats. You declare new UTIs inside a bundle's information property list. See "Declaring New Uniform Type Identifiers" (page 19) for more information.

Adopting Uniform Type Identifiers

This chapter gives some guidelines for adopting uniform type identifiers in your application, and gives an overview of the utility functions used to manipulate UTIs.

Guidelines for UTI Usage

Adopting UTIs in your application is a two-part process. You should use UTIs whenever you need to identify or exchange data, and you should declare specific UTIs for any proprietary types your application uses.

Adding UTI Support to Mac OS X Applications

Apple uses UTIs throughout Mac OS X. For example: building in UTI support for most data-interchange needs. For example, the following technologies all support UTIs:

- The Pasteboard Manager and Translation Services use UTIs to identify data types.
- Navigation Services allows you to specify UTIs for file filtering.
- Launch Services supports UTI-based document claims.
- NSView and NSWindow support UTI-based drag-and-drop promises.
- NSDocument, NSOpenPanel, NSSavePanel, and NSWorkspace all support UTIs.
- NSSound, NSImage and NSImageRep use UTIs to return supported data formats.

Further, Apple has deprecated most older mechanisms for identifying data in favor of UTIs.

If you have specific needs that are not addressed by the above, you can match types to UTIs in your own code. Typically this requires you to find a type with an alternate identifier (such as an `OSType`), create a UTI from that identifier, then check for conformance with UTIs defining the types your application can handle. For an example of how to do this, see Navigation Services Tasks in *Navigation Services Programming Guide*.

Important: When using system-defined UTIs in your code, you should use the constants defined in `UTCoreTypes.h` in the Launch Services framework when available, rather than the actual UTI strings. For example, pass `kUTTypeApplication` rather than “`com.apple.application`”. “System-Declared Uniform Type Identifiers ” (page $@) lists these constants in addition to the UTI strings.

Adding UTI Support to iOS Applications

iOS applications use UTIs to represent pasteboard types. For more information, see *UIPasteboard Class Reference*.

Important: When using system-defined UTIs in your code, you should use the constants defined in `UTCoreTypes.h` in the MobileCoreServices framework when available, rather than the actual UTI strings. For example, pass `kUTTypeApplication` rather than "`com.apple.application`". "System-Declared Uniform Type Identifiers" (page $@) lists these constants in addition to the UTI strings.

An Overview of UTI Functions

You can find the functions used to manipulate UTIs in `UTType.h` in the Launch Services framework on Mac OS X and the MobileCoreServices framework on iOS.

Testing for Equality and Conformance

When testing to see if two UTIs are identical, you should always use the `UTTypeEqual` function rather than direct string comparison:

```
Boolean UTTypeEqual (
        CFStringRef inUTI1,
        CFStringRef inUTI2
    );
```

The two UTIs are equal if

- the UTI strings are identical
- a dynamic identifier's tag specification is a subset of the other UTI's tag specification.

However, in many cases you want to determine whether one UTI is compatible with another, in which case you should check for conformance rather than equality:

```
Boolean UTTypeConformsTo (
        CFStringRef inUTI1,
        CFStringRef inUTI2
    );
```

The `UTTypeConformsTo` function returns `true` if `inUTI1` conforms to `inUTI2`. Conformance relationships are transitive: if A conforms to B, and B conforms to C, then A conforms to C.

Manipulating Tags

Often to use UTIs effectively, you must be able to convert various other type identifiers (OSType, MIME, and so on) to UTIs and vice versa.

To convert an identifier to a UTI, you can use the `UTTypeCreatePreferredIdentifierForTag` function:

```
CFStringRef UTTypeCreatePreferredIdentifierForTag(
        CFStringRef inTagClass,
        CFStringRef inTag,
        CFStringRef inConformingToUTI
    );
```

For the tag class, you pass one of the following tag class constants that define the alternate identifiers:

```
const CFStringRef kUTTagClassFilenameExtension;
const CFStringRef kUTTagClassMIMEType;
const CFStringRef kUTTagClassNSPboardType;
const CFStringRef kUTTagClassOSType;
```

You can pass a UTI in the `inConformingToUTI` parameter as a hint, in case the given tag appears in more than one UTI declaration. For example, if you know that the filename extension tag is associated with a file, not a directory, you can pass `public.data` here, which causes the function to ignore any types with the same extension that conform to `public.directory`. Pass `NULL` for this parameter if you have no hints.

In the rare case that two or more types exist that have the same identifier, this function prefers public UTIs over others. If no declared UTI exists for the identifier, `UTTypeCreatePreferredIdentifierForTag` creates and returns a dynamic identifier.

If you want to obtain all the UTIs that correspond to a given identifier, you can call `UTTypeCreateAllIdentifiersForTag`:

```
CFArrayRef UTTypeCreateAllIdentifiersForTag(
    CFStringRef inTagClass,
    CFStringRef inTag,
    CFStringRef inConformingToUTI );
```

This function returns an array of UTIs that you can examine to determine which one to use.

If you want to create an alternate identifier from a UTI, you call the `UTTypeCopyPreferredTagWithClass` function:

```
CFStringRef UTTypeCopyPreferredTagWithClass(
    CFStringRef inUTI,
    CFStringRef inTagClass );
```

The preferred tag is the first one listed in the tag specification array for a given tag class.

Converting OSType Identifiers

The UTI utility functions assume that all alternate identifier tags can be represented as Core Foundation strings. However, because type `OSType` is integer-based rather than string-based, it may not be immediately obvious how to correctly translate between type `CFStringRef` and type `OSType`. To ensure error-free encoding and decoding of OSType identifiers, use the following conversion functions:

```
CFStringRef UTCreateStringForOSType( OSType inOSType );

OSType UTGetOSTypeFromString( CFStringRef inTag );
```

Note: For OSType values containing only printable 7-bit ASCII characters, you can still use the `CFSTR` macro with a four-character string literal (for example, `CFSTR("TEXT")` to create a valid OSType tag.

Accessing UTI Information

To obtain a copy of a UTI's declaration, use the `UTTCopyDeclaration` function:

```
CFDictionaryRef UTTypeCopyDeclaration(
    CFStringRef inUTI );
```

To obtain a URL to the bundle that contains the declaration for a given UTI, use the `UTTypeCopyDeclaringBundleURL` function:

```
CFURLRef UTTypeCopyDeclaringBundleURL(
    CFStringRef inUTI );
```

To obtain the localized description of a given UTI, call the `UTTypeCopyDescription` function:

```
CFStringRef UTTypeCopyDescription(
    CFStringRef inUTI );
```

Declaring New Uniform Type Identifiers

Mac OS X applications can add new uniform type identifiers for proprietary data formats. You declare new UTIs in the information property list (`info.plist`) file of a bundle. You can declare new UTIs in any of the following:

- Application bundles
- Spotlight Importer bundles
- Automator action bundles

Declaring UTIs

In addition to declaring the UTI string, the declaration can contain any of the following properties:

- The type's tag specification, specifying all the alternate identifier tags that match this type
- A list of UTIs to which this identifier conforms
- The icon to use when displaying items of this type
- A user-readable string describing this identifier, which the containing bundle may localize

Your UTI declarations must be either imported or exported:

- An exported UTI declaration means that the type is available for use by all other parties. For example, an application that uses a proprietary document format should declare it as an exported UTI.
- An imported UTI declaration is used to declare a type that the bundle does not own, but would like to see available on the system. For example, say a video-editing program creates files using a proprietary format whose UTI is declared in its application bundle. If you are writing an application or plugin that can read such files, you must make sure that the system knows about the proprietary UTI, even if the actual video-editing application is not available. To do so, your application should redeclare the UTI in its own bundle but mark it as an imported declaration.

If both imported and exported declarations for a UTI exist, the exported declaration takes precedence over imported one.

Here is a sample declaration for the `public.jpeg` UTI, defined as an exported type, as you would find in a property list:

```
<key>UTExportedTypeDeclarations</key>
       <array>
           <dict>
               <key>UTTypeIdentifier</key>
               <string>public.jpeg</string>
               <key>UTTypeReferenceURL</key>
```

```
                <string>http://www.w3.org/Graphics/JPEG/</string>
                <key>UTTypeDescription</key>
                <string>JPEG image</string>
                <key>UTTypeIconFile</key>
                <string>public.jpeg.icns</string>
                <key>UTTypeConformsTo</key>
                <array>
                    <string>public.image</string>
                    <string>public.data</string>
                </array>
                <key>UTTypeTagSpecification</key>
                <dict>
                    <key>com.apple.ostype</key>
                    <string>JPEG</string>
                    <key>public.filename-extension</key>
                    <array>
                        <string>jpeg</string>
                        <string>jpg</string>
                    </array>
                    <key>public.mime-type</key>
                    <string>image/jpeg</string>
                </dict>
            </dict>
        </array>
```

Table 3-1 (page 20) shows a list of the available property key lists that you use in UTI declarations.

Table 3-1 Property list keys for uniform type identifiers

Key	Value type	Description
`UTExportedType-Declarations`	array of dictionaries	An array of exported UTI declarations (that is, identifiers owned by the bundle's publisher).
`UTImportedType-Declarations`	array of dictionaries	An array of imported UTI declarations (that is, identifiers owned by another company or organization).
`UTTypeIdentifier`	string	The UTI for the declared type. This key is required for UTI declarations.
`UTTypeTagSpecification`	dictionary	A dictionary defining one or more equivalent type identifiers.
`UTTypeConformsTo`	array of strings	The UTIs to which this identifier conforms.
`UTTypeIconFile`	string	The name of the bundle icon resource to associate with this UTI.
`UTTypeDescription`	string	A user-visible description of this type (may be localized).
`UTTypeReferenceURL`	string	The URL of a reference document describing this type.

Recommendations for Declaring new Uniform Type Identifiers

If your application uses proprietary data formats, you should declare them in the `Info.plist` file of your application bundle. Some guidelines:

- Your UTI string must be unique. Following the reverse-DNS format beginning with `com.`*companyName* is a simple way to ensure uniqueness. While the system can support different UTI strings with the same specification, the reverse is not true.
- If your code relies on third-party UTI types that may not be present on the system, you should declare those UTIs as imported types in your bundle.
- Be sure to add conformance information if your proprietary type is a subtype of one or more existing types. In most cases you should not specify conformance to a nonpublic type, unless you are also declaring that type in your bundle. For a list of public and Apple-defined UTIs, see "System-Declared Uniform Type Identifiers " (page $@).

Document Revision History

This table describes the changes to *Uniform Type Identifiers Overview*.

Date	Notes
2009-11-17	Extracted table of system-defined UTIs.
2009-09-09	Added information about how UTIs are utilized in Mac OS v10.5, Mac OS v10.6, and iOS.
2008-04-08	Added information about the UTI character set.
2007-10-31	Made minor technical and editorial corrections.
2005-11-09	Fixed typographical errors. Clarified usage of UTTypeConformsTo. Added public.archive to "conforms to" list for com.pkware.zip-archive.
2005-06-04	Updated list of system and imported UTIs. Made corrections and updates for Mac OS X v10.4, including additional conformance information.
2005-04-29	Changed title from "Understanding Uniform Type Identifiers." First public version.
2004-06-28	First seed release.