File format

A **file format** is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium. File formats may be either proprietary or free and may be either unpublished or open.

0:00 / 0:00

wav-file: 2.1 Megabytes.

Some file formats are designed for very particular types of data: PNG files, for example, store bitmapped images using lossless data compression. Other file formats, however, are designed for storage of several different types of data: the Ogg format can act as a container for different types of multimedia including any combination of audio and video, with or without text (such as subtitles), and metadata. A text file can contain any stream of characters, including possible control characters, and is encoded in one of various character encoding schemes. Some file formats, such as HTML, scalable vector graphics, and the source code of computer software are text files with defined syntaxes that allow them to be used for specific purposes.

0:00 / 0:00

ogg-file: 154 kilobytes.

Contents

External links

Specifications

File formats often have a published specification describing the encoding method and enabling testing of program intended functionality. Not all formats have freely available specification documents, partly because some developers view their specification documents as trade secrets, and partly because other developers never author a formal specification document, letting precedent set by other already existing programs that use the format define the format via how these existing programs use it.

If the developer of a format doesn't publish free specifications, another developer looking to utilize that kind of file must either reverse engineer the file to find out how to read it or acquire the specification document from the format's developers for a fee and by signing a non-disclosure agreement. The latter approach is possible only when a formal specification document exists. Both strategies require significant time, money, or both; therefore, file formats with publicly available specifications tend to be supported by more programs.

Patents

Patent law, rather than copyright, is more often used to protect a file format. Although patents for file formats are not directly permitted under US law, some formats encode data using patented algorithms. For example, using compression with the GIF file format requires the use of a patented algorithm, and though the patent owner did not initially enforce their patent, they later began collecting royalty fees. This has resulted in a significant decrease in the use of GIFs, and is partly responsible for the development of the alternative PNG format. However, the GIF patent expired in the US in mid-2003, and worldwide in mid-2004.

Identifying file type

Different operating systems have traditionally taken different approaches to determining a particular file's format, with each approach having its own advantages and disadvantages. Most modern operating systems and individual applications need to use all of the following approaches to read "foreign" file formats, if not work with them completely.

Filename extension

One popular method used by many operating systems, including Windows, macOS, CP/M, DOS, VMS and VM/CMS is to determine the format of a file based on the end of its name, more specifically the letters following the final period. This portion of the filename is known as the filename extension. For example, HTML documents are identified by names that end with `.html` (or `.htm`), and GIF images by `.gif`. In the original FAT file system, file names were limited to an eight-character identifier and a three-character extension, known as an 8.3 filename. There are only so many three-letter extensions, so, often any given extension might be linked to more than one program. Many formats still use three-character extensions even though modern operating systems and application programs no longer have this limitation. Since there is no standard list of extensions, more than one format can use the same extension, which can confuse both the operating system and users.

One artifact of this approach is that the system can easily be tricked into treating a file as a different format simply by renaming it—an HTML file can, for instance, be easily treated as plain text by renaming it from `filename.html` to `filename.txt`. Although this strategy was useful to expert users who could easily

understand and manipulate this information, it was often confusing to less technical users, who could accidentally make a file unusable (or "lose" it) by renaming it incorrectly.

This led more recent operating system shells, such as Windows 95 and Mac OS X, to hide the extension when listing files. This prevents the user from accidentally changing the file type, and allows expert users to turn this feature off and display the extensions.

Hiding the extension, however, can create the appearance of two or more identical filenames in the same folder. For example, a company logo may be needed both in `.eps` format (for publishing) and `.png` format (for web sites). With the extensions visible, these would appear as the unique filenames: "`CompanyLogo.eps`" and "`CompanyLogo.png`". On the other hand, hiding the extensions would make both appear as "`CompanyLogo`", which can lead to confusion.

Hiding extensions can also pose a security risk.[1] For example, a malicious user could create an executable program with an innocent name such as "`Holiday photo.jpg.exe`". The "`.exe`" would be hidden and an unsuspecting user would see "`Holiday photo.jpg`", which would appear to be a JPEG image, usually unable to harm the machine. However, the operating system would still see the "`.exe`" extension and run the program, which would then be able to cause harm to the computer. The same is true with files with only one extension: as it is not shown to the user, no information about the file can be deduced without explicitly investigating the file. To further trick users, it is possible to store an icon inside the program, in which case some operating systems' icon assignment for the executable file (`.exe`) would be overridden with an icon commonly used to represent JPEG images, making the program look like an image. Extensions can also be spoofed: some Microsoft Word macro viruses create a Word file in template format and save it with a `.doc` extension. Since Word generally ignores extensions and looks at the format of the file, these would open as templates, execute, and spread the virus. These issues require users with extensions hidden to be vigilant and never let the operating system choose with what program to open a file not known to be trustworthy (which contradicts the idea of making things easier for the user). This represents a practical problem for Windows systems where extension-hiding is turned on by default.

Internal metadata

A second way to identify a file format is to *use information* regarding the format stored inside the file itself, either information meant for this purpose or binary strings that happen to always be in specific locations in files of some formats. Since the easiest place to locate them is at the beginning, such area is usually called a *file header* when it is greater than a few bytes, or a *magic number* if it is just a few bytes long.

File header

The metadata contained in a file header are usually stored at the start of the file, but might be present in other areas too, often including the end, depending on the file format or the type of data contained. Character-based (text) files usually have character-based headers, whereas binary formats usually have binary headers, although this is not a rule. Text-based file headers usually take up more space, but being human-readable, they can easily be examined by using simple software such as a text editor or a hexadecimal editor.

As well as identifying the file format, file headers may contain metadata about the file and its contents. For example, most image files store information about image format, size, resolution and color space, and optionally authoring information such as who made the image, when and where it was made, what camera model and photographic settings were used (Exif), and so on. Such metadata may be used by software reading or interpreting the file during the loading process and afterwards.

File headers may be used by an operating system to quickly gather information about a file without loading it all into memory, but doing so uses more of a computer's resources than reading directly from the directory information. For instance, when a graphic file manager has to display the contents of a folder, it must read the headers of many files before it can display the appropriate icons, but these will be located in different places on the storage medium thus taking longer to access. A folder containing many files with complex metadata such as thumbnail information may require considerable time before it can be displayed.

If a header is binary hard-coded such that the header itself needs complex interpretation in order to be recognized, especially for metadata content protection's sake, there is a risk that the file format can be misinterpreted. It may even have been badly written at the source. This can result in corrupt metadata which, in extremely bad cases, might even render the file unreadable.

A more complex example of file headers are those used for wrapper (or container) file formats.

Magic number

One way to incorporate file type metadata, often associated with Unix and its derivatives, is just to store a "magic number" inside the file itself. Originally, this term was used for a specific set of 2-byte identifiers at the beginnings of files, but since any binary sequence can be regarded as a number, any feature of a file format which uniquely distinguishes it can be used for identification. GIF images, for instance, always begin with the ASCII representation of either `GIF87a` or `GIF89a`, depending upon the standard to which they adhere. Many file types, especially plain-text files, are harder to spot by this method. HTML files, for example, might begin with the string `<html>` (which is not case sensitive), or an appropriate document type definition that starts with `<!DOCTYPE HTML>`, or, for XHTML, the XML identifier, which begins with `<?xml`. The files can also begin with HTML comments, random text, or several empty lines, but still be usable HTML.

The magic number approach offers better guarantees that the format will be identified correctly, and can often determine more precise information about the file. Since reasonably reliable "magic number" tests can be fairly complex, and each file must effectively be tested against every possibility in the magic database, this approach is relatively inefficient, especially for displaying large lists of files (in contrast, file name and metadata-based methods need to check only one piece of data, and match it against a sorted index). Also, data must be read from the file itself, increasing latency as opposed to metadata stored in the directory. Where file types don't lend themselves to recognition in this way, the system must fall back to metadata. It is, however, the best way for a program to check if the file it has been told to process is of the correct format: while the file's name or metadata may be altered independently of its content, failing a well-designed magic number test is a pretty sure sign that the file is either corrupt or of the wrong type. On the other hand, a valid magic number does not guarantee that the file is not corrupt or is of a correct type.

So-called shebang lines in script files are a special case of magic numbers. Here, the magic number is human-readable text that identifies a specific command interpreter and options to be passed to the command interpreter.

Another operating system using magic numbers is AmigaOS, where magic numbers were called "Magic Cookies" and were adopted as a standard system to recognize executables in Hunk executable file format and also to let single programs, tools and utilities deal automatically with their saved data files, or any other kind of file types when saving and loading data. This system was then enhanced with the Amiga standard Datatype recognition system. Another method was the FourCC method, originating in OSType on Macintosh, later adapted by Interchange File Format (IFF) and derivatives.

External metadata

A final way of storing the format of a file is to explicitly store information about the format in the file system, rather than within the file itself.

This approach keeps the metadata separate from both the main data and the name, but is also less portable than either filename extensions or "magic numbers", since the format has to be converted from filesystem to filesystem. While this is also true to an extent with filename extensions—for instance, for compatibility with MS-DOS's three character limit—most forms of storage have a roughly equivalent definition of a file's data and name, but may have varying or no representation of further metadata.

Note that zip files or archive files solve the problem of handling metadata. A utility program collects multiple files together along with metadata about each file and the folders/directories they came from all within one new file (e.g. a zip file with extension `.zip`). The new file is also compressed and possibly encrypted, but now is transmissible as a single file across operating systems by FTP systems or attached to email. At the destination, it must be unzipped by a compatible utility to be useful, but the problems of transmission are solved this way.

Mac OS type-codes

The Mac OS' Hierarchical File System stores codes for *creator* and *type* as part of the directory entry for each file. These codes are referred to as OSTypes. These codes could be any 4-byte sequence, but were often selected so that the ASCII representation formed a sequence of meaningful characters, such as an abbreviation of the application's name or the developer's initials. For instance a HyperCard "stack" file has a *creator* of `WILD` (from Hypercard's previous name, "WildCard") and a *type* of `STAK`. The BBEdit text editor has a creator code of `R*ch` referring to its original programmer, Rich Siegel. The type code specifies the format of the file, while the creator code specifies the default program to open it with when double-clicked by the user. For example, the user could have several text files all with the type code of `TEXT`, but which each open in a different program, due to having differing creator codes. This feature was intended so that, for example, human-readable plain-text files could be opened in a general purpose text editor, while programming or HTML code files would open in a specialized editor or IDE. However, this feature was often the source of user confusion, as which program would launch when the files were double-clicked was often unpredictable. RISC OS uses a similar system, consisting of a 12-bit number which can be looked up in a table of descriptions—e.g. the hexadecimal number `FF5` is "aliased" to `PoScript`, representing a PostScript file.

Mac OS X uniform type identifiers (UTIs)

A Uniform Type Identifier (UTI) is a method used in macOS for uniquely identifying "typed" classes of entity, such as file formats. It was developed by Apple as a replacement for OSType (type & creator codes).

The UTI is a Core Foundation string, which uses a reverse-DNS string. Some common and standard types use a domain called `public` (e.g. `public.png` for a Portable Network Graphics image), while other domains can be used for third-party types (e.g. `com.adobe.pdf` for Portable Document Format). UTIs can be defined within a hierarchical structure, known as a conformance hierarchy. Thus, `public.png` conforms to a supertype of `public.image`, which itself conforms to a supertype of `public.data`. A UTI can exist in multiple hierarchies, which provides great flexibility.

In addition to file formats, UTIs can also be used for other entities which can exist in macOS, including:

- Pasteboard data
- Folders (directories)
- Translatable types (as handled by the Translation Manager)

- Bundles
- Frameworks
- Streaming data
- Aliases and symlinks

OS/2 extended attributes

The HPFS, FAT12 and FAT16 (but not FAT32) filesystems allow the storage of "extended attributes" with files. These comprise an arbitrary set of triplets with a name, a coded type for the value and a value, where the names are unique and values can be up to 64 KB long. There are standardized meanings for certain types and names (under OS/2). One such is that the ".TYPE" extended attribute is used to determine the file type. Its value comprises a list of one or more file types associated with the file, each of which is a string, such as "Plain Text" or "HTML document". Thus a file may have several types.

The NTFS filesystem also allows storage of OS/2 extended attributes, as one of the file *forks*, but this feature is merely present to support the OS/2 subsystem (not present in XP), so the Win32 subsystem treats this information as an opaque block of data and does not use it. Instead, it relies on other file forks to store meta-information in Win32-specific formats. OS/2 extended attributes can still be read and written by Win32 programs, but the data must be entirely parsed by applications.

POSIX extended attributes

On Unix and Unix-like systems, the ext2, ext3, ReiserFS version 3, XFS, JFS, FFS, and HFS+ filesystems allow the storage of extended attributes with files. These include an arbitrary list of "name=value" strings, where the names are unique and a value can be accessed through its related name.

PRONOM unique identifiers (PUIDs)

The PRONOM Persistent Unique Identifier (PUID) is an extensible scheme of persistent, unique and unambiguous identifiers for file formats, which has been developed by The National Archives of the UK as part of its PRONOM technical registry service. PUIDs can be expressed as Uniform Resource Identifiers using the `info:pronom/` namespace. Although not yet widely used outside of UK government and some digital preservation programmes, the PUID scheme does provide greater granularity than most alternative schemes.

MIME types

MIME types are widely used in many Internet-related applications, and increasingly elsewhere, although their usage for on-disc type information is rare. These consist of a standardised system of identifiers (managed by IANA) consisting of a *type* and a *sub-type*, separated by a slash—for instance, `text/html` or `image/gif`. These were originally intended as a way of identifying what type of file was attached to an e-mail, independent of the source and target operating systems. MIME types identify files on BeOS, AmigaOS 4.0 and MorphOS, as well as store unique application signatures for application launching. In AmigaOS and MorphOS the Mime type system works in parallel with Amiga specific Datatype system.

There are problems with the MIME types though; several organisations and people have created their own MIME types without registering them properly with IANA, which makes the use of this standard awkward in some cases.

File format identifiers (FFIDs)

File format identifiers is another, not widely used way to identify file formats according to their origin and their file category. It was created for the Description Explorer suite of software. It is composed of several digits of the form `NNNNNNNNN-XX-YYYYYYY`. The first part indicates the organisation origin/maintainer (this number represents a value in a company/standards organisation database), the 2 following digits categorize the type of file in hexadecimal. The final part is composed of the usual filename extension of the file or the international standard number of the file, padded left with zeros. For example, the PNG file specification has the FFID of `000000001-31-0015948` where `31` indicates an image file, `0015948` is the standard number and `000000001` indicates the International Organization for Standardization (ISO).

File content based format identification

Another but less popular way to identify the file format is to examine the file contents for distinguishable patterns among file types. The contents of a file are a sequence of bytes and a byte has 256 unique permutations (0–255). Thus, counting the occurrence of byte patterns that is often referred as byte frequency distribution gives distinguishable patterns to identify file types. There are many content-based file type identification schemes that use byte frequency distribution to build the representative models for file type and use any statistical and data mining techniques to identify file types [2]

File structure

There are several types of ways to structure data in a file. The most usual ones are described below.

Unstructured formats (raw memory dumps)

Earlier file formats used raw data formats that consisted of directly dumping the memory images of one or more structures into the file.

This has several drawbacks. Unless the memory images also have reserved spaces for future extensions, extending and improving this type of structured file is very difficult. It also creates files that might be specific to one platform or programming language (for example a structure containing a Pascal string is not recognized as such in C). On the other hand, developing tools for reading and writing these types of files is very simple.

The limitations of the unstructured formats led to the development of other types of file formats that could be easily extended and be backward compatible at the same time.

Chunk-based formats

In this kind of file structure, each piece of data is embedded in a container that somehow identifies the data. The container's scope can be identified by start- and end-markers of some kind, by an explicit length field somewhere, or by fixed requirements of the file format's definition.

Throughout the 1970s, many programs used formats of this general kind. For example, word-processors such as troff, Script, and Scribe, and database export files such as CSV. Electronic Arts and Commodore-Amiga also used this type of file format in 1985, with their IFF (Interchange File Format) file format.

A container is sometimes called a *"chunk"*, although "chunk" may also imply that each piece is small, and/or that chunks do not contain other chunks; many formats do not impose those requirements.

The information that identifies a particular "chunk" may be called many different things, often terms including "field name", "identifier", "label", or "tag". The identifiers are often human-readable, and classify parts of the data: for example, as a "surname", "address", "rectangle", "font name", etc. These are not the same thing as identifiers in the sense of a database key or serial number (although an identifier may well identify its **associated data** as such a key).

With this type of file structure, tools that do not know certain chunk identifiers simply skip those that they do not understand. Depending on the actual meaning of the skipped data, this may or may not be useful (CSS explicitly defines such behavior).

This concept has been used again and again by RIFF (Microsoft-IBM equivalent of IFF), PNG, JPEG storage, DER (Distinguished Encoding Rules) encoded streams and files (which were originally described in CCITT X.409:1984 and therefore predate IFF), and Structured Data Exchange Format (SDXF).

Indeed, any data format must **somehow** identify the significance of its component parts, and embedded boundary-markers are an obvious way to do so:

- MIME headers do this with a colon-separated label at the start of each logical line. MIME headers cannot contain other MIME headers, though the data content of some headers has sub-parts that can be extracted by other conventions.
- CSV and similar files often do this using a header records with field names, and with commas to mark the field boundaries. Like MIME, CSV has no provision for structures with more than one level.
- XML and its kin can be loosely considered a kind of chunk-based format, since data elements are identified by markup that is akin to chunk identifiers. However, it has formal advantages such as schemas and validation, as well as the ability to represent more complex structures such as trees, DAGs, and charts. If XML is considered a "chunk" format, then SGML and its predecessor IBM GML are among the earliest examples of such formats.
- JSON is similar to XML without schemas, cross-references, or a definition for the meaning of repeated field-names, and is often convenient for programmers.
- YAML is similar to JSON, but use indentation to separate data chunks and aim to be more human-readable than JSON or XML.
- Protocol Buffers are in turn similar to JSON, notably replacing boundary-markers in the data with field numbers, which are mapped to/from names by some external mechanism.

Directory-based formats

This is another extensible format, that closely resembles a file system (OLE Documents are actual filesystems), where the file is composed of 'directory entries' that contain the location of the data within the file itself as well as its signatures (and in certain cases its type). Good examples of these types of file structures are disk images, OLE documents TIFF, libraries. ODT and DOCX, being PKZIP-based are chunked and also carry a directory.

See also

- Audio file format
- Chemical file format
- Comparison of executable file formats
- Digital container format
- Document file format
- DROID file format identification utility

- File (command), a file type identification utility
- File conversion
- Future proofing
- Graphics file format summary
- Image file formats
- List of archive formats
- List of file formats
- List of file signatures, or "magic numbers"
- List of filename extensions (alphabetical)
- List of free file formats
- List of motion and gesture file formats
- Magic number (programming)
- Object file
- Video file format
- Windows file types

References

1. PC World (23 December 2003). "Windows Tips: For Security Reasons, It Pays To Know Your File Extensions" (https://web.archive.org/web/20080423022237/http://www.pcworld.com/article/id,113758-page,1/article.html). Archived from the original (http://www.pcworld.com/article/id,113758-page,1/article.html) on 23 April 2008. Retrieved 20 June 2008.
2. "File Format Identification" (https://web.archive.org/web/20090814051001/http://www.forensicswiki.org/wiki/File_Format_Identification). Archived from the original (http://www.forensicswiki.org/wiki/File_Format_Identification) on 2009-08-14. Retrieved 2009-07-21.
    - "Extended Attribute Data Types" (https://web.archive.org/web/20041225112451/http://www.markcrocker.com/rexxtipsntricks/rxtt28.2.0301.html). *REXX Tips & Tricks, Version 2.80*. Archived from the original (http://markcrocker.com/rexxtipsntricks/rxtt28.2.0301.html) on December 25, 2004. Retrieved February 9, 2005.
    - "Extended Attributes used by the WPS" (https://web.archive.org/web/20050321120229/http://markcrocker.com/rexxtipsntricks/rxtt28.2.0300.html). *REXX Tips & Tricks, Version 2.80*. Archived from the original (http://markcrocker.com/rexxtipsntricks/rxtt28.2.0300.html) on March 21, 2005. Retrieved February 9, 2005.
    - "Extended Attributes - what are they and how can you use them ?" (http://www.howzatt.demon.co.uk/articles/06may93.html). *Roger Orr*. Retrieved February 9, 2005.

External links

- File format (https://curlie.org//Computers/Data_Formats/) at Curlie
- *Best Practices for File Formats* (http://library.stanford.edu/research/data-management-services/data-best-practices/best-practices-file-formats), US: Stanford University Libraries, Data Management Services ("The file formats you use have a direct impact on your ability to open those files at a later date and on the ability of other people to access those data")
