# Microsoft Cabinet File Format



***Abstract***

*This document defines the internal format of a Microsoft Cabinet File.  This information may be used to build cabinet file creation or extraction tools, or other cabinet-enabled applications.  The actual data compression formats, if used, are described in other documents.*

# 1 Introduction

This specification defines the Microsoft cabinet file format. Cabinet files are compressed packages containing a number of related files. The format of a cabinet file is optimized for maximum compression. Cabinet files support a number of compression formats, including MSZIP, LZX, or uncompressed. This document does not define these internal compression formats. For data compression formats, refer to the documents titled *Microsoft MSZIP Data Compression Format* and *Microsoft LZX Data Compression Format*.

# 2 Specification

## 2.1 Conventions

The types `u1`, `u2`, and `u4` are used to represent unsigned 8-, 16-, and 32-bit integer values, respectively. All multi-byte quantities are stored in little-endian order, where the least significant byte comes first.

The cabinet file format is described here using a C-like structure notation, where successive fields appear in the structure sequentially without padding or alignment. Header fields followed by (optional) may or may not be present, depending on the values in the `CFHEADER` flags byte.

## 2.2 Overview

Each file stored in a cabinet is stored completely within a single folder. A cabinet file may contain one or more folders, or portions of a folder. A folder can span across multiple cabinets. Such a series of cabinet files form a set. Each cabinet file contains name information for the logically adjacent cabinet files. Each folder contains one or more files.

Throughout this discussion, cabinets are said to contain “files”. This is for semantic purposes only. Cabinet files actually store streams of bytes, each with a name and some other common attributes. Whether these byte streams are actually files or some other kind of data is application-defined.

A cabinet file contains a cabinet header (`CFHEADER`), followed by one or more cabinet folder (`CFFOLDER`) entries, a series of one or more cabinet file (`CFFILE`) entries, and the actual compressed file data in `CFDATA` entries. The compressed file data in the `CFDATA` entry is stored in one of several compression formats, as indicated in the corresponding `CFFOLDER` structure. The compression encoding formats used are detailed in separate documents.

## 2.3

## *Detailed structure specification*

### 2.3.1 CFHEADER

The CFHEADER structure provides information about this cabinet file.

```
struct CFHEADER
{
    u1  signature[4];       /* cabinet file signature */
    u4  reserved1;          /* reserved */
    u4  cbCabinet;          /* size of this cabinet file in bytes */
    u4  reserved2;          /* reserved */
    u4  coffFiles;          /* offset of the first CFFILE entry */
    u4  reserved3;          /* reserved */
    u1  versionMinor;       /* cabinet file format version, minor */
    u1  versionMajor;       /* cabinet file format version, major */
    u2  cFolders;           /* number of CFFOLDER entries in this cabinet */
    u2  cFiles;             /* number of CFFILE entries in this cabinet */
    u2  flags;              /* cabinet file option indicators */
    u2  setID;              /* must be the same for all cabinets in a set */
    u2  iCabinet;           /* number of this cabinet file in a set */
    u2  cbCFHeader;         /* (optional) size of per-cabinet reserved area */
    u1  cbCFFolder;         /* (optional) size of per-folder reserved area */
    u1  cbCFData;           /* (optional) size of per-datablock reserved area */
    u1  abReserve[];        /* (optional) per-cabinet reserved area */
    u1  szCabinetPrev[];    /* (optional) name of previous cabinet file */
    u1  szDiskPrev[];       /* (optional) name of previous disk */
    u1  szCabinetNext[];    /* (optional) name of next cabinet file */
    u1  szDiskNext[];       /* (optional) name of next disk */
};
```

**u1 signature[4]**

Contains the characters 'M','S','C','F' (bytes 0x4D, 0x53, 0x43, 0x46). This field is used to assure that the file is a cabinet file.

**u4 reserved1**

Reserved field, set to zero.

**u4 cbCabinet**

Total size of this cabinet file in bytes.

**u4 reserved2**

Reserved field, set to zero.

**u4 coffFiles**

Absolute file offset of first CFFILE entry.

**u4 reserved3**

Reserved field, set to zero.

**u1 versionMinor**
**u1 versionMajor**

Cabinet file format version. Currently, versionMajor = 1 and versionMinor = 3.

**u2 cFolders**

The number of CFFOLDER entries in this cabinet file.

**u2 cFiles**

The number of CFFILE entries in this cabinet file.

**u2 flags**

Bit-mapped values which indicate the presence of optional data:

```
#define cfhdrPREV_CABINET       0x0001
#define cfhdrNEXT_CABINET       0x0002
#define cfhdrRESERVE_PRESENT    0x0004
```

flags.cfhdrPREV_CABINET is set if this cabinet file is not the first in a set of cabinet files. When this bit is set, the szCabinetPrev and szDiskPrev fields are present in this CFHEADER.

flags.cfhdrNEXT_CABINET is set if this cabinet file is not the last in a set of cabinet files. When this bit is set, the szCabinetNext and szDiskNext fields are present in this CFHEADER.

flags.cfhdrRESERVE_PRESENT is set if this cabinet file contains any reserved fields. When this bit is set, the cbCFHeader, cbCFFolder, and cbCFData fields are present in this CFHEADER.

Other bit positions in the flags field are reserved.

**u2 setID**

An arbitrarily-derived (random) value which binds a collection of linked cabinet files together. All cabinet files in a set will contain the same setID. This field is used by cabinet file extractors to assure that cabinet files are not inadvertently mixed. This value has no meaning in a cabinet file that is not in a set.

**u2 iCabinet**

Sequential number of this cabinet in a multi-cabinet set. The first cabinet has `iCabinet`=0. This field, along with `setID`, is used by cabinet file extractors to assure that this cabinet is the correct continuation cabinet when spanning cabinet files.

**u2 cbCFHeader** (optional)

If `flags.cfhdrRESERVE_PRESENT` is not set, this field is not present, and the value of `cbCFHeader` defaults to zero. Indicates the size in bytes of the `abReserve` field in this `CFHEADER`. Values for `cbCFHeader` range from 0 to 60,000.

**u1 cbCFFolder** (optional)

If `flags.cfhdrRESERVE_PRESENT` is not set, then this field is not present, and the value of `cbCFFolder` defaults to zero. Indicates the size in bytes of the `abReserve` field in each `CFFOLDER` entry. Values for `cbCFFolder` range from 0 to 255.

**u1 cbCFData** (optional)

If `flags.cfhdrRESERVE_PRESENT` is set, then this field is not present, and the value for `cbCFData` defaults to zero. Indicates the size in bytes of the `abReserve` field in each `CFDATA` entry. Values for `cbCFData` range from 0 to 255.

**u1 abReserve[cbCFHeader]** (optional)

If `flags.cfhdrRESERVE_PRESENT` is set and `cbCFHeader` is non-zero, then this field contains per-cabinet-file application information. This field is defined by the application and used for application-defined purposes.

**u1 szCabinetPrev[]** (optional)

If `flags.cfhdrPREV_CABINET` is not set, then this field is not present. NUL-terminated ASCII string containing the file name of the logically-previous cabinet file. May contain up to 255 bytes plus the NUL byte. Note that this gives the name of the most-recently-preceding cabinet file that contains the initial instance of a file entry. This might not be the immediately previous cabinet file, when the most recent file spans multiple cabinet files. If searching in reverse for a specific file entry, or trying to extract a file which is reported to begin in the "previous cabinet", `szCabinetPrev` would give the name of the cabinet to examine.

**u1 szDiskPrev[]** (optional)

If `flags.cfhdrPREV_CABINET` is not set, then this field is not present. NUL-terminated ASCII string containing a descriptive name for the media containing the file named in `szCabinetPrev`, such as the text on the diskette label. This string can be used when prompting the user to insert a diskette. May contain up to 255 bytes plus the NUL byte.

**u1 szCabinetNext[]** (optional)

If `flags.cfhdrNEXT_CABINET` is not set, then this field is not present. NUL-terminated ASCII string containing the file name of the next cabinet file in a set. May contain up to 255 bytes plus the NUL byte. Files extending beyond the end of the current cabinet file are continued in the named cabinet file.

**u1 szDiskNext[]** (optional)

If `flags.cfhdrNEXT_CABINET` is not set, then this field is not present. NUL-terminated ASCII string containing a descriptive name for the media containing the file named in `szCabinetNext`, such as the text on the diskette label. May contain up to 255 bytes plus the NUL byte. This string can be used when prompting the user to insert a diskette.

## 2.3.2

## CFFOLDER

Each CFFOLDER structure contains information about one of the folders or partial folders stored in this cabinet file. The first CFFOLDER entry immediately follows the CFHEADER entry. CFHEADER.cFolders indicates how many CFFOLDER entries are present.

Folders may start in one cabinet, and continue on to one or more succeeding cabinets. When the cabinet file creator detects that a folder has been continued into another cabinet, it will complete that folder as soon as the current file has been completely compressed. Any additional files will be placed in the next folder. Generally, this means that a folder would span at most two cabinets, but if the file is large enough, it could span more than two cabinets.

CFFOLDER entries actually refer to folder fragments, not necessarily complete folders. A CFFOLDER structure is the beginning of a folder if the iFolder value in the first file referencing the folder does not indicate the folder is continued from the previous cabinet file.

The typeCompress field may vary from one folder to the next, unless the folder is continued from a previous cabinet file.

```
struct CFFOLDER
{
    u4  coffCabStart;       /* offset of the first CFDATA block in this folder */
    u2  cCFData;            /* number of CFDATA blocks in this folder */
    u2  typeCompress;       /* compression type indicator */
    u1  abReserve[];        /* (optional) per-folder reserved area */
};
```

**u4 coffCabStart**

Absolute file offset of first CFDATA block for this folder.

**u2 cCFData**

Number of CFDATA structures for this folder that are actually in this cabinet. A folder can continue into another cabinet and have more CFDATA blocks in that cabinet, and a folder may have started in a previous cabinet. This number represents only the CFDATA structures for this folder that are at least partially recorded in this cabinet.

**u2 typeCompress**

Indicates the compression method used for all CFDATA entries in this folder. The valid values are defined in each compression format's specification.

**u1 abReserve[CFHEADER.cbCFFolder]** (optional)

If CFHEADER.flags.cfhdrRESERVE_PRESENT is set and cbCFFolder is non-zero, then this field contains per-folder application information. This field is defined by the application and used for application-defined purposes.

### 2.3.3 CFFILE

Each CFFILE entry contains information about one of the files stored (or at least partially stored) in this cabinet. The first CFFILE entry in each cabinet is found at absolute offset CFHEADER.coffFiles. CFHEADER.cFiles indicates how many of these entries are in the cabinet. The CFFILE entries in a cabinet are ordered by iFolder value, then by uoffFolderStart. Entries for files continued from the previous cabinet will be first, and entries for files continued to the next cabinet will be last.

```
struct CFFILE
{
    u4  cbFile;             /* uncompressed size of this file in bytes */
    u4  uoffFolderStart;    /* uncompressed offset of this file in the folder */
    u2  iFolder;            /* index into the CFFOLDER area */
    u2  date;               /* date stamp for this file */
    u2  time;               /* time stamp for this file */
    u2  attribs;            /* attribute flags for this file */
    u1  szName[];           /* name of this file */
};
```

**u4 cbFile**

Uncompressed size of this file in bytes.

**u4 uoffFolderStart**

Uncompressed byte offset of the start of this file's data. For the first file in each folder, this value will usually be zero. Subsequent files in the folder will have offsets that are typically the running sum of the cbFile values.

**u2 iFolder**

Index of the folder containing this file's data. A value of zero indicates this is the first folder in this cabinet file. The special iFolder values ifoldCONTINUED_FROM_PREV and ifoldCONTINUED_PREV_AND_NEXT indicate that the folder index is actually zero, but that extraction of this file would have to begin with the cabinet named in CFHEADER.szCabinetPrev. The special iFolder values ifoldCONTINUED_PREV_AND_NEXT and ifoldCONTINUED_TO_NEXT indicate that the folder index is actually one less than CFHEADER.cFolders, and that extraction of this file will require continuation to the cabinet named in CFHEADER.szCabinetNext.

```
#define ifoldCONTINUED_FROM_PREV      (0xFFFD)
#define ifoldCONTINUED_TO_NEXT        (0xFFFE)
#define ifoldCONTINUED_PREV_AND_NEXT  (0xFFFF)
```

**u2 date**

Date of this file, in the format ((year-1980) << 9)+(month << 5)+(day), where month={1..12} and day={1..31}. This "date" is typically considered the "last modified" date in local time, but the actual definition is application-defined.

**u2 time**

Time of this file, in the format `(hour << 11)+(minute << 5)+(seconds/2)`, where `hour`={0..23}. This "time" is typically considered the "last modified" time in local time, but the actual definition is application-defined.

**u2 attribs**

Attributes of this file; may be used in any combination:

```
#define _A_RDONLY       (0x01) /* file is read-only */
#define _A_HIDDEN       (0x02) /* file is hidden */
#define _A_SYSTEM       (0x04) /* file is a system file */
#define _A_ARCH         (0x20) /* file modified since last backup */
#define _A_EXEC         (0x40) /* run after extraction */
#define _A_NAME_IS_UTF  (0x80) /* szName[] contains UTF */
```

All other attribute bit values are reserved.

**char szName[]**

NUL-terminated name of this file. Note that this string may include path separator characters. When `attribs._A_NAME_IS_UTF` is set, this string can be converted directly to Unicode, avoiding locale-specific dependencies. See "UTF Encoding" for more information. When `attribs._A_NAME_IS_UTF` is not set, this string is subject to interpretation depending on locale.

## 2.3.4

## CFDATA

Each CFDATA record describes some amount of compressed data. The first CFDATA entry for each folder is located using CFFOLDER.coffCabStart. Subsequent CFDATA records for this folder are contiguous.

```
struct CFDATA
{
    u4  csum;               /* checksum of this CFDATA entry */
    u2  cbData;             /* number of compressed bytes in this block */
    u2  cbUncomp;           /* number of uncompressed bytes in this block */
    u1  abReserve[];        /* (optional) per-datablock reserved area */
    u1  ab[cbData];         /* compressed data bytes */
};
```

**u4 csum**

Checksum of this CFDATA structure, from CFDATA.cbData through CFDATA.ab[cbData-1]. See "Checksum Method" for more information. May be set to zero if the checksum is not supplied.

**u2 cbData**

Number of bytes of compressed data in this CFDATA record. When cbUncomp is zero, this field indicates only the number of bytes which fit into this cabinet file.

**u2 cbUncomp**

The uncompressed size of the data in this CFDATA entry. When this CFDATA entry is continued in the next cabinet file, cbUncomp will be zero, and cbUncomp in the first CFDATA entry in the next cabinet file will report the total uncompressed size of the data from both CFDATA blocks.

**u1 abReserve[CFHEADER.cbCFData]** (optional)

If CFHEADER.flags.cfhdrRESERVE_PRESENT is set and cbCFHeader is non-zero, then this field contains per-datablock application information. This field is defined by the application and used for application-defined purposes.

**u1 ab[cbData]**

The compressed data bytes, compressed using the CFFOLDER.typeCompress method. When cbUncomp is zero, these data bytes must be combined with the data bytes from the next cabinet's first CFDATA entry before decompression.

When CFFOLDER.typeCompress indicates that the data is not compressed, this field contains the uncompressed data bytes. In this case, cbData and cbUncomp will be equal unless this CFDATA entry crosses a cabinet file boundary.

# 3

## A Sample Cabinet File

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
000   4D 53 43 46 00 00 00 00-FD 00 00 00 00 00 00 00   MSCF
010   2C 00 00 00 00 00 00 00-03 01 01 00 02 00 00 00
020   22 06 00 00 5E 00 00 00-01 00 00 00 4D 00 00 00
030   00 00 00 00 00 00 6C 22-BA 59 20 00 68 65 6C 6C              hell
040   6F 2E 63 00 4A 00 00 00-4D 00 00 00 00 00 6C 22   o.c
050   E7 59 20 00 77 65 6C 63-6F 6D 65 2E 63 00 BD 5A       welcome.c
060   A6 30 97 00 97 00 23 69-6E 63 6C 75 64 65 20 3C         #include <
070   73 74 64 69 6F 2E 68 3E-0D 0A 0D 0A 76 6F 69 64   stdio.h>    void
080   20 6D 61 69 6E 28 76 6F-69 64 29 0D 0A 7B 0D 0A    main(void)  {
090   20 20 20 20 70 72 69 6E-74 66 28 22 48 65 6C 6C       printf("Hell
0A0   6F 2C 20 77 6F 72 6C 64-21 5C 6E 22 29 3B 0D 0A   o, world!\n");
0B0   7D 0D 0A 23 69 6E 63 6C-75 64 65 20 3C 73 74 64   }  #include <std
0C0   69 6F 2E 68 3E 0D 0A 0D-0A 76 6F 69 64 20 6D 61   io.h>    void ma
0D0   69 6E 28 76 6F 69 64 29-0D 0A 7B 0D 0A 20 20 20   in(void)  {
0E0   20 70 72 69 6E 74 66 28-22 57 65 6C 63 6F 6D 65    printf("Welcome
0F0   21 5C 6E 22 29 3B 0D 0A-7D 0D 0A 0D 0A           !\n");  }
```

This is a very simple example of a cabinet file which contains two small text files, stored uncompressed for clarity.

```
     Offset       Description
00..23            CFHEADER
         00..03   signature = 0x4D, 0x53, 0x43, 0x46
         04..07   reserved1
         08..0B   cbCabinet = 0x000000FD (253)
         0C..0F   reserved2
         10..13   coffFiles = 0x0000002C
         14..17   reserved3
         18..19   versionMinor, Major = 1.3
         1A..1B   cFolders = 1
         1C..1D   cFiles = 2
         1E..1F   flags = 0 (no reserve, no previous or next cabinet)
         20..21   setID = 0x0622
         22..23   iCabinet = 0

24..2B            CFFOLDER[0]
         24..27   coffCabStart = 0x0000005E
         28..29   cCFData = 1
         2A..2B   typeCompress = 0 (none)

2C..43            CFFILE[0]
         2C..2F   cbFile = 0x0000004D (77 bytes)
         30..33   uoffFolderStart = 0x00000000
         34..35   iFolder = 0
         36..37   date = 0x226C = 0010001 0011 01100 = March 12, 1997
         38..39   time = 0x59BA = 01011 001101 11010 = 11:13:52 AM
         3A..3B   attribs = 0x0020 = _A_ARCHIVE
         3C..43   szName = “hello.c” + NUL

44..5D            CFFILE[1]
         44..47   cbFile = 0x0000004A (74 bytes)
         48..4B   uoffFolderStart = 0x0000004D
         4C..4D   iFolder = 0
         4E..4F   date = 0x226C = 0010001 0011 01100 = March 12, 1997
         50..51   time = 0x59E7 = 01011 001111 00111 = 11:15:14 AM
         52..53   attribs = 0x0020 = _A_ARCHIVE
         54..5D   szName = “welcome.c” + NUL
```

```
5E..FD            CFDATA[0]
         5E..61   csum = 0x30A65ABD
         62..63   cbData = 0x0097 (151 bytes)
         64..65   cbUncomp = 0x0097 (151 bytes)
         66..FD   ab[0x0097] = uncompressed file data
```

**4**

# Notes

## 4.1 Checksum Method

The computation and verification of checksums found in CFDATA entries cabinet files is done using a function named CSUMCompute. Its actual source code is provided for reference. When checksums are not supplied by the cabinet file creating application, the checksum field is set to zero. Cabinet extracting applications do not compute or verify the checksum if the field is set to zero.

```
CHECKSUM CSUMCompute(void *pv, UINT cb, CHECKSUM seed)
{
    int         cUlong;                 // Number of ULONGs in block
    CHECKSUM    csum;                   // Checksum accumulator
    BYTE       *pb;
    ULONG       ul;

    cUlong = cb / 4;                    // Number of ULONGs
    csum = seed;                        // Init checksum
    pb = pv;                            // Start at front of data block

    //** Checksum integral multiple of ULONGs
    while (cUlong-- > 0) {
        //** NOTE: Build ULONG in big/little-endian independent manner
        ul = *pb++;                     // Get low-order byte
        ul |= (((ULONG)(*pb++)) <<  8); // Add 2nd byte
        ul |= (((ULONG)(*pb++)) << 16); // Add 3nd byte
        ul |= (((ULONG)(*pb++)) << 24); // Add 4th byte

        csum ^= ul;                     // Update checksum
    }

    //** Checksum remainder bytes
    ul = 0;
    switch (cb % 4) {
        case 3:
            ul |= (((ULONG)(*pb++)) << 16); // Add 3nd byte
        case 2:
            ul |= (((ULONG)(*pb++)) <<  8); // Add 2nd byte
        case 1:
            ul |= *pb++;                    // Get low-order byte
        default:
            break;
    }
    csum ^= ul;                         // Update checksum

    //** Return computed checksum
    return csum;
}
```

The checksums for non-split CFDATA blocks are computed first on the compressed data bytes, then on the CFDATA header area, starting at the CFDATA.cbData field:

```
CFDATA.cbData = cbCompressed;
CFDATA.cbUncomp = cbUncompressed;
csumPartial = CSUMCompute(&CFDATA.ab[0],CFDATA.cbData,0);
CFDATA.csum = CSUMCompute(&CFDATA.cbData,sizeof(CFDATA) -
                    sizeof(CFDATA.csum),csumPartial);
```

When blocks are split across cabinet file boundaries, the checksum for the partial block at the end of a cabinet file is computed first on the partial field of compressed data bytes, then on the header:

```
CFDATA.cbData = cbPartialData;
CFDATA.cbUncomp = 0;
csumPartial = CSUMCompute(&CFDATA.ab[0],cbPartialData,0);
CFDATA.csum = CSUMCompute(&CFDATA.cbData,sizeof(CFDATA) -
```

```
                        sizeof(CFDATA.csum),csumPartial);
```

The checksum for the residual block in the next cabinet file is computed first on the remainder of the field of compressed data bytes, then on the header:

```
CFDATA.cbData = cbResidualData;
CFDATA.cbUncomp = cbUncompressed;
csumPartial = CSUMCompute(&CFDATA.ab[cbPartialData],cbResidualData,0);
CFDATA.csum = CSUMCompute(&CFDATA.cbData,sizeof(CFDATA) -
                        sizeof(CFDATA.csum),csumPartial);
```

## *4.2*

### *UTF Encoding Method*

UTF (universal text format) is used to compactly represent a broad range of Unicode characters while favoring size for the most common characters. Unicode characters are translated to sequences of one, two, or three bytes per character.

When a string containing Unicode characters larger than 0x007F are encoded in the `CFFILE.szName` field, the `_A_NAME_IS_UTF` attribute should be included in the file's attributes. When no characters larger than 0x007F are in the name, the `_A_NAME_IS_UTF` attribute should not be set. If byte values larger than 0x7F are found in `CFFILE.szName`, but the `_A_NAME_IS_UTF` attribute is not set, the characters should be interpreted according to the current locale.

Unicode characters with values 0x0000 through 0x007F are represented by a single byte of the same value.

The first byte emitted for Unicode characters 0x0080 through 0x07FF is `0xC0+(unicodevalue >> 6)`, and the second byte is `0x80+(unicodevalue & 0x003F)`.

Unicode characters 0x0800 through 0xFFFF are represented by byte1 = `0xE0+(unicodevalue >> 12)`, byte2 = `0x80+((unicodevalue >> 6) & 0x3F)`, and byte3 = `0x80+(unicodevalue & 0x3F)`.