The MoPaQ Archive Format

From /dev/klog Wiki

Contents

Introduction to the MoPaQ Format

The MoPaQ (or MPQ) format is an archive file format designed by Mike O'Brien (hence the name Mike O'brien PaCK) at Blizzard Entertainment. The format has been used in all Blizzard games since (and including) Diablo. It is heavily optimized to be a read-only game archive format, and excels at this role.

The Blizzard MoPaQ-reading functions are contained in the Storm module, which my be either statically or dynamically linked. Traditionally, the Blizzard MoPaQ-writing functions are contained in the MPQAPI module, which is always statically linked; in Starcraft 2, however, the MoPaQ-writing functions are now contained in Storm.

StormLib - mentioned several times in this specification - is an open-source MoPaQ reading and writing library written by Ladislav Zezula (no affiliation with Blizzard Entertainment). While it's a bit dated, and does not support all of the newer MoPaQ features, it contains source code to the more exotic compression methods used by MoPaQ, such as the PKWare implode algorithm, MoPaQ's huffman compression algorithm, and the IMA ADPCM compression used by MoPaQ.

The MoPaQ Format

All numbers in the MoPaQ format are in little endian byte order; signed numbers use the two's complement system. Data types are listed either as int (integer, the number of bits specified), byte (8 bits), or char (bytes which contain ASCII characters). All sizes and offsets are in bytes, unless specified otherwise. Structure members are listed in the following general form: offset from the beginning of the structure: data type(array size) member name : member description

General Archive Layout

- Archive Header
- File Data
- File Data - Special Files
- Hash Table
- Block Table
- Extended Block Table
- Strong Digital signature

This is the usual archive layout, but it is not mandatory. Some archives have been observed placing the hash table and file table after the archive header, and before the file data.

However, beginning with Starcraft 2, the block table must immediately follow the hash table.

Archive Header

```
00h: char(4) Magic             Indicates that the file is a MoPaQ archive. Must be ASCII "MPQ" 1Ah.
04h: int32 HeaderSize          Size of the archive header.
08h: int32 ArchiveSize         Size of the whole archive, including the header. Does not include the strong digital signature,
                               if present. This size is used, among other things, for determining the region to hash in computing
                               the digital signature. This field is deprecated in the Burning Crusade MoPaQ format, and the size
                               of the archive is calculated as the size from the beginning of the archive to the end of the
                               hash table, block table, or extended block table (whichever is largest).
0Ch: int16 FormatVersion       MoPaQ format version. MPQAPI will not open archives where this is negative. Known versions:
        0000h                  Original format. HeaderSize should be 20h, and large archives are not supported.
        0001h                  Burning Crusade format. Header size should be 2Ch, and large archives are supported.
0Eh: int8 SectorSizeShift      Power of two exponent specifying the number of 512-byte disk sectors in each logical sector
                               in the archive. The size of each logical sector in the archive is 512 * 2^SectorSizeShift.
                               Bugs in the Storm library dictate that this should always be 3 (4096 byte sectors).
10h: int32 HashTableOffset     Offset to the beginning of the hash table, relative to the beginning of the archive.
14h: int32 BlockTableOffset    Offset to the beginning of the block table, relative to the beginning of the archive.
18h: int32 HashTableEntries    Number of entries in the hash table. Must be a power of two, and must be less than 2^16
                               for the original MoPaQ format, or less than 2^20 for the Burning Crusade format.
1Ch: int32 BlockTableEntries   Number of entries in the block table.

Fields only present in the Burning Crusade format and later:

20h: int64 ExtendedBlockTableOffset   Offset to the beginning of the extended block table, relative to the beginning of the archive.
28h: int16 HashTableOffsetHigh        High 16 bits of the hash table offset for large archives.
2Ah: int16 BlockTableOffsetHigh       High 16 bits of the block table offset for large archives.
```

The archive header is the first structure in the archive, at archive offset 0; however, the archive does not need to be at offset 0 of the containing file. The offset of the archive in the file is referred to here as ArchiveOffset. If the archive is not at the beginning of the file, it must begin at a disk sector boundary (512 bytes). Early versions of Storm require that the archive be at the end of the containing file (ArchiveOffset + ArchiveSize = file size), but this is not required in newer versions (due to the strong digital signature not being considered a part of the archive).

Block Table

The block table contains entries for each region in the archive. Regions may be either files, empty space, which may be overwritten by new files (typically this space is from deleted file data), or unused block table entries. Empty space entries should have BlockOffset and BlockSize nonzero, and FileSize and Flags zero; unused block table entries should have BlockSize, FileSize, and Flags zero. The block table is encrypted, using the hash of "(block table)" as the key. Each entry is structured as follows:

```
00h: int32 BlockOffset   Offset of the beginning of the block, relative to the beginning of the archive.
04h: int32 BlockSize     Size of the block in the archive.
08h: int32 FileSize      Size of the file data stored in the block. Only valid if the block is a file; otherwise
                         meaningless, and should be 0. If the file is compressed, this is the size of the uncompressed
                         file data.
0Ch: int32 Flags         Bit mask of the flags for the block. The following values are conclusively identified:
        80000000h        Block is a file, and follows the file data format; otherwise, block is free space or unused.
                         If the block is not a file, all other flags should be cleared, and FileSize should be 0.
        04000000h        File has checksums for each sector (explained in the File Data section). Ignored if file is not
                         compressed or imploded.
        02000000h        File is a deletion marker, indicating that the file no longer exists. This is used to allow
                         patch archives to delete files present in lower-priority archives in the search chain.
        01000000h        File is stored as a single unit, rather than split into sectors.
        00020000h        The file's encryption key is adjusted by the block offset and file size (explained in detail in the
                         File Data section). File must be encrypted.
        00010000h        File is encrypted.
        00000200h        File is compressed. File cannot be imploded.
        00000100h        File is imploded. File cannot be compressed.
```

Extended Block Table

The extended block table was added to support archives larger than 4 gigabytes (2^32 bytes). The table contains the upper bits of the archive offsets for each block in the block table. It is simply an array of int16s, which become bits 32-47 of the archive offsets for each block, with bits 48-63 being zero. Individual blocks in the archive are still limited to 4 gigabytes in size. This table is only present in Burning Crusade format archives that exceed 4 gigabytes size.

Unlike the hash and block tables, the extended block table is not encrypted nor compressed.

Hash Table

Instead of storing file names, for quick access MoPaQs use a fixed, power of two-size hash table of files in the archive. A file is uniquely identified by its file path, its language, and its platform.

The home entry for a file in the hash table is computed as a hash of the file path. In the event of a collision (the home entry is occupied by another file), progressive overflow is used, and the file is placed in the next available hash table entry. Searches for a desired file in the hash table proceed from the home entry for the file until either the file is found, the entire hash table is searched, or an empty hash table entry (FileBlockIndex of FFFFFFFFh) is encountered.

The hash table is always encrypted, using the hash of "(hash table)" as the key.

Prior to Starcraft 2, the hash table is stored uncompressed. In Starcraft 2, however, the table may optionally be compressed. If the offset of the block table is not equal to the offset of the hash table plus the uncompressed size, Starcraft 2 interprets the hash table as being compressed (not imploded). This calculation assumes that the block table immediately follows the hash table, and will fail or crash otherwise.

Each entry is structured as follows:

```
00h: int32 FilePathHashA    The hash of the file path, using method A.
04h: int32 FilePathHashB    The hash of the file path, using method B.
08h: int16 Language         The language of the file. This is a Windows LANGID data type, and uses the same values.
                            0 indicates the default language (American English), or that the file is language-neutral.
0Ah: int8 Platform          The platform the file is used for. 0 indicates the default platform. No other values
                            have been observed.
0Ch: int32 FileBlockIndex   If the hash table entry is valid, this is the index into the block table of the file.
                            Otherwise, one of the following two values:
       FFFFFFFFh            Hash table entry is empty, and has always been empty. Terminates searches for a given file.
       FFFFFFFEh            Hash table entry is empty, but was valid at some point (in other words, the file was deleted).
                            Does not terminate searches for a given file.
```

Extended Attributes

The extended attributes are optional file attributes for files in the block table. These attributes were added at times after the MoPaQ format was already finalized, and it is not necessary for every archive to have all (or any) of the extended attributes. If an archive contains a given attribute, there will be an instance of that attribute for every block in the block table, although the attribute will be meaningless if the block is not a file. The order of the attributes for blocks correspond to the order of the blocks in the block table, and are of the same number. The attributes are stored in parallel arrays in the "(attributes)" file (default language and platform), in the archive. The attributes corresponding to this file need not be valid (and logically cannot be). Unlike all the other structures in the MoPaQ format, entries in the extended attributes are NOT guaranteed to be aligned. Also note that in some archives, malicious zeroing of the attributes has been observed, perhaps with the intent of breaking archive viewers. This file is structured as follows:

```
00h: int32 Version :           Specifies the extended attributes format version. For now, must be 100.
04h: int32 AttributesPresent : Bit mask of the extended attributes present in the archive:
       00000001h: File CRC32s.
       00000002h: File timestamps.
       00000004h: File MD5s.
08h: int32(BlockTableEntries) CRC32s :   CRC32s of the (uncompressed) file data for each block in the archive.
                                         Omitted if the archive does not have CRC32s.
FILETIME(BlockTableEntries) Timestamps : Timestamps for each block in the archive. The format is that of the
                                         Windows FILETIME structure. Omitted if the archive does not have timestamps.
MD5(BlockTableEntries) MD5s :            MD5s of the (uncompressed) file data for each block in the archive.
                                         Omitted if the archive does not have MD5s.
```

File Data

The data for each file is composed of the following structure:

```
int32(SectorsInFile* + 1) SectorOffsetTable  Offsets to the start of each sector, relative to the beginning
                                             of the file data. The last entry contains the total compressed file
                                             size, making it possible to easily calculate the size of any given
                                             sector by simple subtraction. This table is not present or necessary
                                             if the file is not compressed.
SECTOR Sectors(SectorsInFile)                Data of each sector in the file, packed end to end (see details below).
```

Normally, file data is split up into sectors, for simple streaming. All sectors, save for the last, will contain as many bytes of file data as specified in the archive header's SectorSizeShift; the last sector may contain less than this, depending on the size of the entire file's data. If the file is compressed or imploded, the sector will be smaller or the same size as the file data it contains. Individual sectors in a compressed or imploded file may be stored uncompressed; this occurs if and only if the file data the sector contains could not be compressed by the algorithm(s) used (if the compressed sector size was greater than or equal to the size of the file data), and is indicated by the sector's size in SectorOffsetTable being equal to the size of the file data in the sector (which may be calculated from the FileSize).

Sector checksums, when present, are stored as an additional sector at the end of the file, and such files consequently have an additional entry in the sector table; however, unlike the other sectors in the file, this sector is always compressed (not imploded) and unencrypted. This sector consists of an array of checksums, one for each sector in the file minus the checksum sector itself. Each checksum is computed from a compressed but unencrypted sector; for further details of the implementation of sector checksums, see VerifySectorChecksum below.

The format of each sector depends on the kind of sector it is. Uncompressed sectors are simply the the raw file data contained in the sector. Imploded sectors are the raw compressed data following compression with the implode algorithm (these sectors can only be in imploded files). Compressed sectors (only found in compressed - not imploded - files) are compressed with one or more compression algorithms, and have the following structure:

```
byte CompressionMask : Mask of the compression types applied to this sector.
byte(SectorSize - 1) SectorData : The compressed data for the sector.
```

CompressionMask indicates which compression algorithm(s) to apply to the compressed sector. This byte counts towards the total sector size, and the sector will be stored uncompressed if the data cannot be compressed by at least two bytes; in other words, there must be an overall gain of at least one byte through compression. As well, this byte is encrypted with the sector data, if applicable. The following compression algorithms are defined (for implementations of these algorithms, see StormLib):

```
20h: Sparse compressed. Added in Starcraft 2.
40h: IMA ADPCM mono
80h: IMA ADPCM stereo
01h: Huffman encoded
02h: Deflated (see ZLib). Added in Warcraft 3.
08h: Imploded (see PKWare Data Compression Library)
10h: BZip2 compressed (see BZip2). Added in World of Warcraft: The Burning Crusade.
```

Prior to Starcraft 2, this byte is interpreted as a bit mask of algorithms to apply. Algorithms, which may be combined arbitrarily, are applied in the order entries are shown above during compression, and in the opposite order during decompression.

In Starcraft 2, this byte is now processed via a single large lookup table of values, which translates each possible value into either one or two different compression algorithms. Algorithms may no longer be arbitrarily combined, but must come from a list of valid combinations; specifically, most compression types (e.g. Deflate and Implode) are now mutually exclusive, and may only be combined with a small number of other algorithms (e.g. IMA ADPCM), which themselves are mutually exclusive. Furthermore, there is one exception (12h) where the value is not two algorithms combined, but a single, new algorithm (LZMA). The list of valid combinations:

```
12h: LZMA compression. Added in Starcraft 2.
22h: Sparse compression + Deflate
30h: Sparse compression + BZip2
41h: Mono IMA ADPCM + Huffman encoding
48h: Mono IMA ADPCM + Implode
81h: Stereo IMA ADPCM + Huffman encoding
88h: Stereo IMA ADPCM + Implode
```

If the file is stored as a single unit (indicated in the file's Flags), there is effectively only a single sector, which contains the entire file data.

If the file is encrypted, each sector (after compression/implosion, if applicable) is encrypted with the file's key. The base key for a file is determined by a hash of the file name stripped of the directory (i.e. the key for a file named "directory\file" would be computed as the hash of "file"). If this key is adjusted, as indicated in the file's Flags, the final key is calculated as ((base key + BlockOffset) XOR FileSize). Each sector is encrypted using the key + the 0-based index of the sector in the file. The SectorOffsetTable, if present, is encrypted using the key - 1.

The SectorOffsetTable is omitted when the sizes and offsets of all sectors in the file are calculatable from the FileSize. This can happen in several circumstances. If the file is not compressed/imploded, then the size and offset of all sectors is known, based on the archive's SectorSizeShift. If the file is stored as a single unit compressed/imploded, then the SectorOffsetTable is omitted, as the single file "sector" corresponds to BlockSize and FileSize, as mentioned previously. However, the SectorOffsetTable will be present if the file is compressed/imploded and the file is not stored as a single unit, even if there is only a single sector in the file (the size of the file is less than or equal to the archive's sector size).

Listfile

The listfile is a very simple extension to the MoPaQ format that contains the file paths of (most) files in the archive. The languages and platforms of the files are not stored in the listfile. The listfile is contained in the file "(listfile)" (default language and platform), and is simply a text file with file paths separated by ';', 0Dh, 0Ah, or some combination of these. The file "(listfile)" may not be listed in the listfile.

Weak Digital Signature

The weak digital signature is a RSASSA-PKCS1-v1_5 digital signature, using the MD5 hashing algorithm and a 512-bit (weak) RSA key (for more information about this protocol, see the RSA Labs PKCS1 specification). The public key and exponent are stored in a resource in Storm, the private key is stored in a separate file, whose filename is passed to MPQAPI (the private key is not stored in MPQAPI). The signature is stored uncompressed, unencrypted in the file "(signature)" (default language and platform) in the archive. The archive is hashed from the beginning of the archive (ArchiveOffset in the containing file) to the end of the archive (the length indicated by ArchiveSize, or calculated in the Extended MoPaQ format); the signature file is added to the archive before signing, and the space occupied by the file is considered to be all binary 0s during signing/verification. This file is structured as follows:

```
00h: int32 Unknown : Must be 0.
04h: int32 Unknown : Must be 0.
08h: int512 Signature : The digital signature. Like all other numbers in the MoPaQ format, this is stored in little-endian order.
```

The structure of the signature, when decrypted, follows the RSASSA-PKCS1-v1_5 specification; this format is rather icky to work with (Quantam wrote a program to verify this signature using nothing but an MD5 function and huge integer functions; it wasn't pleasant), and best left to an encryption library, such as OpenSSL as shown below:

```
int mpq_verify_weak_signature(RSA *public_key, const unsigned char *signature, const unsigned char *digest) {
    unsigned char reversed_signature[MPQ_WEAK_SIGNATURE_SIZE];
    memcpy(reversed_signature, signature + 8, MPQ_WEAK_SIGNATURE_SIZE);
    memrev(reversed_signature, MPQ_WEAK_SIGNATURE_SIZE);

    return RSA_verify(NID_md5, digest, MD5_DIGEST_LENGTH, reversed_signature, MPQ_WEAK_SIGNATURE_SIZE, public_key);
}
```

MPQKit includes the weak signature public RSA key in PEM format since r73.

Strong Digital Signature - Generics

The strong digital signature consists of a SHA-1 digest with extremely simple padding, encrypted using straight RSA encryption. All known Blizzard keys are 2048-bit (strong) RSA keys; a default key is stored in Storm. Obviously, any RSA key may be used; in fact, an archive signed with the default key has never been seen in the wild. The strong digital signature is stored immediately after the archive, in the containing file. The entire archive (ArchiveSize bytes, starting at ArchiveOffset in the containing file) is hashed as a single block (there is one known exception to that algorithm, see below). In addition, a signature tail may be appended to the SHA-1 digest before it is finalized; this can be any arbitrary blob of data. The signature has the following format:

```
00h: char(4) Magic : Indicates the presence of a digital signature. Must be "NGIS" ("SIGN" backwards).
04h: int2048 Signature : The digital signature, stored in little-endian format.
```

When the Signature field is decrypted with the public key, and the resulting large integer is stored in little-endian order, it is structured as follows:

```
00h: byte Padding : Must be 0Bh.
01h: byte(235) Padding : Must be BBh.
ECh: byte(20) SHA-1 : SHA-1 digest of the archive, in standard SHA-1 byte order.
```

Strong Digital Signature - Specifics

This section aims at giving specific information about the usage of the various known Blizzard public keys, as well as known SHA-1 digest tails.

Warcraft 3 maps

Warcraft 3 maps (.w3m and .w3x) are composed of a map header, followed by an MPQ archive at offset 512, followed by a strong digital signature. The Warcraft 3 Map key is used for Warcraft 3 map signatures; the SHA-1 digest is constructed from the content of the entire file, including the map header, up to the end of the archive. That is, (MapHeaderSize + ArchiveSize bytes, starting at 0 in the containing file). Once the map file has been digested, a tail is appended to the SHA-1 digest; the tail is the uppercased file name of the map.

World of Warcraft Macintosh patches

World of Warcraft Macintosh patches are embodied by a patch program ("patcher"), which contains in its bundle resources a standalone.MPQ archive, which stores the patch's content. standalone.MPQ archives are signed with the World of Warcraft Macintosh Patch key; the SHA-1 digest is constructed normally as described in the general case. Once the archive has been digested, a tail is appended to the SHA-1 digest; the tail is "ARCHIVE".

It should be noted that World of Warcraft Windows patches are signed using the weak signature scheme.

World of Warcraft survey

World of Warcraft stores a Survey.MPQ archive in its WDB directory. Survey.MPQ archives are signed with the World of Warcraft Survey key; the SHA-1 digest is constructed normally as described in the general case. Once the archive has been digested, a tail is appended to the SHA-1 digest; the tail is "ARCHIVE".

It should be noted that the Survey.MPQ archive has not yet been observed in publicly available builds of Burning Crusade.

User Data

The second version of the MoPaQ format, first used in Burning Crusade, features a mechanism to store some amount of data outside the archive proper, though the reason for this mechanism is not known. This is implemented by means of a shunt block that precedes the archive itself. The format of this block is as follows:

```
00h: char(4) Magic                  Indicates that this is a shunt block. ASCII "MPQ" 1Bh.
04h: int32 UserDataSize             The number of bytes that have been allocated in this archive for user data. This does not
                                    need to be the exact size of the data itself, but merely the maximum amount of data which
                                    may be stored in this archive.
08h: int32 ArchiveHeaderOffset      The offset in the file at which to continue the search for the archive header.
0Ch: byte(UserDataSize) UserData    The block to store user data in.
```

When Storm encounters this block in its search for the archive header, it saves the location of the shunt block and resumes its search for the archive header at the offset specified in the shunt.

Blizzard-generated archives place the shunt at the beginning of the file, and begin the archive itself at the next 512-byte boundary after the end of the shunt block.

User Data File

Some archives, based on criteria that are not yet known, are unable to contain a shunt block. In this case, user data is stored in a normal file inside the archive named "(user data)".

Algorithm Source Code

All of the sample code here assumes little endian machine byte order, that the short type is 16 bits, that the long type is 32 bits, and that the long long type is 64 bits. Adjustments must be made if these assumptions are not correct on a given platform. All code not credited otherwise was written by myself in the writing of this specification.

Encryption/Decryption

Based on code from StormLib.

```
unsigned long dwCryptTable[0x500];

// The encryption and hashing functions use a number table in their procedures. This table must be initialized before the functions are called the first time.
void InitializeCryptTable()
{
    unsigned long seed   = 0x00100001;
    unsigned long index1 = 0;
    unsigned long index2 = 0;
    int   i;

    for (index1 = 0; index1 < 0x100; index1++)
    {
        for (index2 = index1, i = 0; i < 5; i++, index2 += 0x100)
        {
            unsigned long temp1, temp2;

            seed  = (seed * 125 + 3) % 0x2AAAAB;
            temp1 = (seed & 0xFFFF) << 0x10;

            seed  = (seed * 125 + 3) % 0x2AAAAB;
            temp2 = (seed & 0xFFFF);

            dwCryptTable[index2] = (temp1 | temp2);
        }
    }
}

void EncryptData(void *lpbyBuffer, unsigned long dwLength, unsigned long dwKey)
{
    assert(lpbyBuffer);

    unsigned long *lpdwBuffer = (unsigned long *)lpbyBuffer;
    unsigned long seed = 0xEEEEEEEE;
    unsigned long ch;

    dwLength /= sizeof(unsigned long);
```

```
    while(dwLength-- > 0)
    {
        seed += dwCryptTable[0x400 + (dwKey & 0xFF)];
        ch = *lpdwBuffer ^ (dwKey + seed);

        dwKey = ((~dwKey << 0x15) + 0x11111111) | (dwKey >> 0x0B);
        seed = *lpdwBuffer + seed + (seed << 5) + 3;

                *lpdwBuffer++ = ch;
    }
}

void DecryptData(void *lpbyBuffer, unsigned long dwLength, unsigned long dwKey)
{
    assert(lpbyBuffer);

    unsigned long *lpdwBuffer = (unsigned long *)lpbyBuffer;
    unsigned long seed = 0xEEEEEEEEL;
    unsigned long ch;

    dwLength /= sizeof(unsigned long);

    while(dwLength-- > 0)
    {
        seed += dwCryptTable[0x400 + (dwKey & 0xFF)];
        ch = *lpdwBuffer ^ (dwKey + seed);

        dwKey = ((~dwKey << 0x15) + 0x11111111L) | (dwKey >> 0x0B);
        seed = ch + seed + (seed << 5) + 3;

                *lpdwBuffer++ = ch;
    }
}
```

Hashing And File Key Computation

These functions may have been derived from StormLib code at some point in the very distant past. It was so long ago that I don't remember for certain.

```
// Different types of hashes to make with HashString
#define MPQ_HASH_TABLE_OFFSET   0
#define MPQ_HASH_NAME_A 1
#define MPQ_HASH_NAME_B 2
#define MPQ_HASH_FILE_KEY       3

// Based on code from StormLib.
unsigned long HashString(const char *lpszString, unsigned long dwHashType)
{
    assert(lpszString);
    assert(dwHashType <= MPQ_HASH_FILE_KEY);

    unsigned long  seed1 = 0x7FED7FEDL;
    unsigned long  seed2 = 0xEEEEEEEEL;
    int    ch;

    while (*lpszString != 0)
    {
        ch = toupper(*lpszString++);

        seed1 = dwCryptTable[(dwHashType * 0x100) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
    }
    return seed1;
}

#define BLOCK_OFFSET_ADJUSTED_KEY 0x00020000L

unsigned long ComputeFileKey(const char *lpszFilePath, const BlockTableEntry &blockEntry, unsigned long nArchiveOffset)
{
        assert(lpszFilePath);

        // Find the file name part of the path
        const char *lpszFileName = strrchr(lpszFilePath, '\\');
        if (lpszFileName)
                lpszFileName++; // Skip the \
        else
                lpszFileName = lpszFilePath;

        // Hash the name to get the base key
        unsigned long nFileKey = HashString(lpszFileName, MPQ_HASH_FILE_KEY);

        // Offset-adjust the key if necessary
        if (blockEntry.Flags & BLOCK_OFFSET_ADJUSTED_KEY)
                nFileKey = (nFileKey + blockEntry.BlockOffset) ^ blockEntry.FileSize;

        return nFileKey;
}
```

Conversion of FILETIME And time_t

This code assumes that the base ("zero") date for time_t is 01/01/1970. This is true on Windows, Unix System V systems, and Mac OS X. It is unknown whether this is true on all other platforms. You'll need to research this yourself, if you plan on porting it somewhere else.

THIS CODE MAY BE INCORRECT, AND HAS NOT BEEN TESTED

```
#define EPOCH_OFFSET 116444736000000000ULL      // Number of 100 ns units between 01/01/1601 and 01/01/1970

bool GetTimeFromFileTime(const FILETIME &fileTime, time_t &time)
{
        // The FILETIME represents a 64-bit integer: the number of 100 ns units since January 1, 1601
        unsigned long long nTime = ((unsigned long long)fileTime.dwHighDateTime << 32) + fileTime.dwLowDateTime;
```

```
	if (nTime < EPOCH_OFFSET)
		return false;

	nTime -= EPOCH_OFFSET;  // Convert the time base from 01/01/1601 to 01/01/1970
	nTime /= 10000000ULL;   // Convert 100 ns to sec

	time = (time_t)nTime;

	// Test for overflow (FILETIME is 64 bits, time_t is 32 bits)
	if ((nTime - (unsigned long long)time) > 0)
		return false;

	return true;
}

void GetFileTimeFromTime(const time_t &time, FILETIME &fileTime)
{
	unsigned long long nTime = (unsigned long long)time;

	nTime *= 10000000ULL;
	nTime += EPOCH_OFFSET;

	fileTime.dwLowDateTime = (DWORD)nTime;
	fileTime.dwHighDateTime = (DWORD)(nTime >> 32);
}
```

Conversion of FILETIME And NSDate

MPQKit includes a category on [NSDate (https://web.archive.org/web/20120222093346/http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSDate_Class/Reference/Reference.html)] to convert to and from NTFS FILETIME, and has been properly tested for correctness.

Forming a 64-Bit Large Archive Offset From 32-Bit And 16-Bit Components

```
unsigned long long MakeLargeArchiveOffset(unsigned long nOffsetLow, unsigned short nOffsetHigh)
{
	return ((unsigned long long)nOffsetHigh << 32) + (unsigned long long)nOffsetLow;
}
```

Verifying a Sector Checksum

```
bool VerifySectorChecksum(const void *buffer, unsigned int length, unsigned long checksum)
{
	if (checksum == 0)
		return true;    // Ignore the actual checksum

	unsigned long bufferChecksum = adler32(0, buffer, length);
	if (bufferChecksum == 0)
		// Can't deal with a 0 checksum
		bufferChecksum = (unsigned long)-1;

	return (bufferChecksum == checksum);
}
```

Finding Files

```
#define MPQ_HASH_ENTRY_EMPTY 0xFFFFFFFFL
#define MPQ_HASH_ENTRY_DELETED 0xFFFFFFFEL

bool FindFileInHashTable(const HashTableEntry *lpHashTable, unsigned long nHashTableSize, const char *lpszFilePath, unsigned short nLang, unsigned char nPlatform, unsigned long &iFileHash
{
	assert(lpHashTable);
	assert(nHashTableSize);
	assert(lpszFilePath);

	// Find the home entry in the hash table for the file
	unsigned long iInitEntry = HashString(lpszFilePath, MPQ_HASH_TABLE_OFFSET) & (nHashTableSize - 1);

	// Is there anything there at all?
	if (lpHashTable[iInitEntry].FileBlockIndex == MPQ_HASH_ENTRY_EMPTY)
		return false;

	// Compute the hashes to compare the hash table entry against
	unsigned long nNameHashA = HashString(lpszFilePath, MPQ_HASH_NAME_A),
		nNameHashB = HashString(lpszFilePath, MPQ_HASH_NAME_B),
		iCurEntry = iInitEntry;

	// Check each entry in the hash table till a termination point is reached
	do
	{
		if (lpHashTable[iCurEntry].FileBlockIndex != MPQ_HASH_ENTRY_DELETED)
		{
			if (lpHashTable[iCurEntry].FilePathHashA == nNameHashA
				&& lpHashTable[iCurEntry].FilePathHashB == nNameHashB
				&& lpHashTable[iCurEntry].Language == nLang
				&& lpHashTable[iCurEntry].Platform == nPlatform)
			{
				iFileHashEntry = iCurEntry;

				return true;
			}
		}

		iCurEntry = (iCurEntry + 1) & (nHashTableSize - 1);
	} while (iCurEntry != iInitEntry && lpHashTable[iCurEntry].FileBlockIndex != MPQ_HASH_ENTRY_EMPTY);

	return false;
}
```

Deleting Files

```
bool DeleteFile(HashTableEntry *lpHashTable, unsigned long nHashTableSize, BlockTableEntry *lpBlockTable, const char *lpszFilePath, unsigned short nLang, unsigned char nPlatform)
{
	assert(lpHashTable);
	assert(nHashTableSize);
	assert(lpBlockTable);

	// Find the file in the hash table
	unsigned long iFileHashEntry;

	if (!FindFileInHashTable(lpHashTable, nHashTableSize, lpszFilePath, nLang, nPlatform, iFileHashEntry))
		return false;

	// Get the block table index before we nuke the hash table entry
	unsigned long iFileBlockEntry = lpHashTable[iFileHashEntry].FileBlockIndex;

	// Delete the file's entry in the hash table
	memset(&lpHashTable[iFileHashEntry], 0xFF, sizeof(HashTableEntry));

	// If the next entry is empty, mark this one as empty; otherwise, mark this as deleted.
	if (lpHashTable[(iFileHashEntry + 1) & (nHashTableSize - 1)].FileBlockIndex == MPQ_HASH_ENTRY_EMPTY)
		lpHashTable[iFileHashEntry].FileBlockIndex = MPQ_HASH_ENTRY_EMPTY;
	else
		lpHashTable[iFileHashEntry].FileBlockIndex = MPQ_HASH_ENTRY_DELETED;

	// If the block occupies space, mark the block as free space; otherwise, clear the block table entry.
	if (lpBlockTable[iFileBlockEntry].BlockSize > 0)
	{
		lpBlockTable[iFileBlockEntry].FileSize = 0;
		lpBlockTable[iFileBlockEntry].Flags = 0;
	}
	else
		memset(&lpBlockTable[iFileBlockEntry], 0, sizeof(BlockTableEntry);

	return true;
}
```

Constants

Locales

```
   MPQNeutral      = 0,
   MPQChinese      = 0x404,
   MPQCzech        = 0x405,
   MPQGerman       = 0x407,
   MPQEnglish      = 0x409,
   MPQSpanish      = 0x40a,
   MPQFrench       = 0x40c,
   MPQItalian      = 0x410,
   MPQJapanese     = 0x411,
   MPQKorean       = 0x412,
   MPQDutch        = 0x413,
   MPQPolish       = 0x415,
   MPQPortuguese   = 0x416,
   MPQRusssian     = 0x419,
   MPQEnglishUK    = 0x809
```