Table of Contents

Proteus Archive Tool

The Proteus Archive Tool, a.k.a PAT, archives and optionally compresses the contents of a directory into two files. One file contains all the data, the second file is the file allocation table. The file allocation table contains all the required data to locate a file within the archive

This was designed to hide files from basic users to prevent modding of files. It does not encrypt files, it merely makes the applications files harder to change. Determined users are impossible to stop. Look into server based data if this is what you are after

Files in the archive are alphabetically sorted and use a simple numerical hash which is designed to allow simple binary searching implementations.

The tool is normally run with asset conversion, but can be run at anytime as required.

Normal usage

Normally in the Proteus Game Engine one or more file archives are registered on game or application startup. This was designed this way to allow for downloadable content or simply for easily updating game data and game scripts. This was supported via the file manager which would search through all registered archives for newer versions of any file requested. You can of course ignore all that and use it as you see fit

As indicated above the .fat files were loaded for rapid searching plus all the registered archive files are were then opened. Once opened, stored files were found by simply seeking to the offset and reading the data

Decompression is of course optional

Finding files

The .fat file can be loaded and then searched to find the offset into the .arc file. The following two pieces of code can be used to locate a file. They are also in the project

```
// ----------------------------------------------------------------------
// Looks for some test files
// ----------------------------------------------------------------------
void Find()
{
    FILE *fp = fopen("arc_00.fat", "rb");
    if (fp)
    {
        // Get file size
        fseek(fp, 0, SEEK_END);
        s32 filesize = (s32)ftell(fp);
        rewind(fp);

        // Read data
        u8 *buffer = (u8*)malloc(filesize);
        fread(buffer, 1, filesize, fp);
        fclose(fp);

        // Point to data
        FatHeader *pHeader     = (FatHeader*)(buffer);
        ArcEntry  *pArcEntries = (ArcEntry*)(buffer + sizeof(FatHeader));

        u32 count = pHeader->entries;
        printf("Entries: %i\n", count);

        ArcEntry *pEntry1 = Locate(StringHash("abc/a.txt"), count, pArcEntries);
        ArcEntry *pEntry2 = Locate(StringHash("textures/dialog.png"), count, pArcEntries);
        ArcEntry *pEntry3 = Locate(StringHash("textures/logo.png"), count, pArcEntries);

        if (pEntry1)
        {
            printf("Found '%s'\n", pEntry1->filename);
        }
        if (pEntry2)
        {
            printf("Found '%s'\n", pEntry2->filename);
        }
        if (pEntry3)
        {
            printf("Found '%s'\n", pEntry3->filename);
        }

        free(buffer);
    }
}
```

```
// ---------------------------------------------------------------------------
// This function looks in an archive for a file using a binary search.
// ---------------------------------------------------------------------------
ArcEntry *Locate(u32 hash, u32 entryCount, ArcEntry *pEntries)
{
    ArcEntry *result = nullptr;

    for (u32 i=0; i<entryCount; i++)
    {
        s32  lower = 0;
        s32  upper = entryCount;

        // Get table start
        ArcEntry *pStart = pEntries;

        while(lower <= upper)
        {
            s32 mid = (lower + upper) / 2;

            // Get mid entry
            ArcEntry *pEntry = pStart;
            pEntry += mid;

            if (hash > pEntry->hash)
            {
                lower = mid + 1;
            }
            else if (hash < pEntry->hash)
            {
                upper = mid - 1;
            }
            else
            {
                result = pEntry;
                break;
            }
        }
    }

    return result;
}
```

Running pat

To run the pat tool on the command line use the basic commands as follows;

To archive a folders contents. Does not compress. Mixed case filenames

pat -i **folderToArchive** -o **outFilename**

To archive a folders contents with compression

pat -i **folderToArchive** -o **outFilename** -c

To archive a folders contents with compression, lowercased filenames and verbose output

pat -i **folderToArchive** -o **outFilename** -c -lc -verb

To archive a folders contents with compression, uppercased filenames and verbose output

pat -i **folderToArchive** -o **outFilename** -c -lc -verb

Notes

The output filename shouldn't contain an extension as the .fat and .arc extensions are added to the output filename by the pat tool as it creates two files, so any extension added will be used but is not necessary

Notes

The order of parameters is not important

Notes

The exported filenames use forward slashes and exclude the **folderToArchive** directory name. Use the -verb parameters to view outputted names. See example output. **Note the 4 byte alignment**

```
-----------------------------------------------------------------------
    Offset Compressed     Actual
in archive       size   Filesize : File
-----------------------------------------------------------------------
         0      18859      47523 : dictionary/c_6.dic
     18860       9827      26110 : dictionary/s_13.dic
     36708       6007      15690 : dictionary/v_9.dic
     42716       4822      10923 : dictionary/k_10.dic
     47540      17470      44520 : dictionary/m_6.dic
```

Data types

The proteus archive tool uses some of the following basic types. Your application should have equivalent data types to read the data correctly. These types are used to define the structs

```
// Basic types
typedef   signed char       s8;             // Signed 8 bit
typedef unsigned char       u8;             // Unsigned 8 bit
typedef   signed short      s16;            // Signed 16 bit
typedef unsigned short      u16;            // Unsigned 16 bit
typedef   signed int        s32;            // Signed 32 bit
typedef unsigned int        u32;            // Unsigned 32 bit
```

The .arc file

The .arc file is a simple binary file containing all of the data aligned to a 4 byte offset. The .fat file is required to determine offsets to data, the format of the data, etc

The .fat file

The .fat file is a simple binary file with the following file structure. The application is currently windows only, Therefore tool uses the windows PC byte ordering. Though the data reads fine on Linux, Android and iOS

```
FatHeader                   // Always contains the header
ArcEntry                    // 1 or more entries
```

A description of both structs follow

The fat file header

The header is also quite simple

```
typedef struct FatHeader
{
    u32 entries;                    // The number of archive entries
    u32 size;                       // The size of the binary file (Including header)
    u32 magic1;                     // File identifier
    u32 magic2;                     // File identifier

} FatHeader;
```

The ArcEntry struct

Each entry in the fat file is stored using this struct. You can of course customize this struct as you see fit, for example the filename isn't really used, except by the archive viewer tool. The expansion variables are just to align data and can also bre removed if required

```
// Each archive entry is store as this block of data
typedef struct ArcEntry
{
    u32     hash;                       // The hashed filename of the entry
    u32     offset;                     // Offset into the archive
    u32     filesize;                   // The uncompressed filesize of the entry
    u32     compressedSize;             // The compressed filesize of the entry
    u8      compressed;                 // 0 == uncompressed 1 == compressed
    u8      compressionType;            // COMPRESSION_TYPE_NONE or COMPRESSION_TYPE_ZLIB
    u8      exp0;                       // Expansion purposes (Free to use)
    u8      exp1;                       // Expansion purposes (Free to use)
    char    filename[260];              // The filename

} ArcEntry;
```

The hash function

This function was designed to be fast, simple and looks like this. It also produces a hash which is suitable for binary searches. A more complex hash may not produce the desired result, though it may be more unique. Please do not change

```
// ---------------------------------------------------------------------------
// Turns a string into a number.
// ---------------------------------------------------------------------------
u32 StringHash(const char* string)
{
    u32 hash = 0;

    while(*string)
    {
        hash += *string;
        hash *= *string++;
    }

    return hash;
}
```

This code works well with the file manager and is required to generate hashes that are used in the file allocation table

Tools

There is also a support tool which lets you view the contents of an archive. Its called the ArchiveManager. Its a .NET 4.5 tool