MIX Format

MIX Format is a simple group file used in 1995-2001. It only stores file name hashes so some file names will result in the same hash and as a result will not be able to be stored together if they differ in content. Position and size of files are stored so size does not need to be inferred. It does not support compression. There are 2 versions of the header, basic and advanced. The basic one only stores the file count and the total data size (not including the header or file index). The advanced one supports additional optional features such as encryption of the header and index and a sha1 checksum tailer to verify the data integrity. An additional flag field is used to indicate which features are present.

File format

Signature

MIX files have no signature. Usually they will have the extension .mix

File Header

The header consists of an optional flags section 4 bytes in size. The first 2 bytes are 0 in this case to distinguish between a mix without feature flags where this stores the file count as UNIT16LE. The flags are stored in the 3rd byte as a bit field with bit 0 indicating a checksum is present and bit 1 indicating that the header is encrypted.

If the flags indicate encryption, an 80 byte block follows the flag section. This block is encrypted with an RSA private key with the public key used to decrypt being embedded in the games code. This block decrypts to a 56 byte Blowfish key that is used to decrypt the rest of the header.

Data type	Name	Description
UINT32LE	Flags	An optional field that specifies additional optional features that the archive uses.

Data type	Name	Description
BYTE[80]	Key	An optional 80 byte block containing an RSA encrypted Blowfish key to use to decode the rest of the header.
UINT16LE	FileCount	The number of files in the archive. Can be used to calculate the size of the file entry index.
UINT32LE	DataSize	Size of the data section.

File Entry

The indexes file entries contain a hash of the filename, the offset of the file within the data section (header size needs to be added for absolute position in the archive) and the size of the file in bytes.

Data type	Name	Description
INT32LE	Id	Hash of the filename. CRC32 in this specific file.
UINT32LE	Offset	File's starting offset within the data section.
UINT32LE	Size	File's size.

In order to quickly locate files from their file name hashes, the index is sorted according to the hashes taken as signed integers.

Encryption

Header encryption can be detected by checking Flags & 0x00020000 does not equal zero. If it doesn't, then the next 80 bytes are two RSA encrypted blocks that need to be decrypted with the Westwood public key. For the purposes of the RSA algorithm, the blocks should be treated as 2 40 byte "big" integers in little endian byte order. 0x10001 should be used as the public exponent for the decryption.

The result is a 56 byte block padded with zeros to use with the Blowfish algorithm in ECB mode. Blocks are 8 bytes long and you need to decode the first block to get the standard header. From that you can calculate the length of the index and thus how many additional blocks you need to

decode to get the rest of the index. If the header and index together are not perfectly divisible by 8, then it is padded with zeros since blowfish operates on 8 byte blocks.

Creating an encrypted header is the reverse of this process, a Blowfish key is generated somehow and used to encrypt the header and index and is itself padded with zeros and encrypted using the Westwood private key.

Key	Value
Public	AihRvNoIbTn85FZRYNZRcT+i6KpU+maCsEqr3Q5q+LDB5tH7Tz2qQ38V
Private	AigKVje8mROcR8QixnxUEF5b29Curkq01DNDWCdOG99XBqH79OaCiTCB

The keys are Base64 encoded and DER encoded big endian "big" integers and represent the modulus and the private exponent respectively in the RSA encryption scheme.

Checksum

Data checksum presence can be detected by checking Flags & 0x00010000 does not equal zero. If it doesn't, then the last 20 bytes of the file are a SHA1 hash of the data contents of the file, not including the header.

Code Snippets

```
// For all data, before hashing it, you need to convert it into uppercase and then apply this snippet:
std::string CRC_PreProcess(std::string s)
{
        const int l = s.length();
        int a = l >> 2;
        if (l & 3)
        {
                s += static_cast<char>(l - (a << 2));
                int i = 3 - (l & 3);
                while (i--)
                        s += s[a << 2];
        }
        return s;
}
```

```
// We somehow stolen some code for you to take a reference



/***********************************************************************************************
 * MixFileClass::MixFileClass -- Constructor for mixfile object.                               *
 *                                                                                             *
 *    This is the constructor for the mixfile object. It takes a filename and a memory         *
 *    handler object and registers the mixfile object with the system. The index block is      *
 *    allocated and loaded from disk by this routine.                                          *
 *                                                                                             *
 * INPUT:   filename -- Pointer to the filename of the mixfile object.                         *
 *                                                                                             *
 * OUTPUT:  none                                                                               *
 *                                                                                             *
 * WARNINGS:   none                                                                            *
 *                                                                                             *
 * HISTORY:                                                                                    *
 *   08/08/1994 JLB : Created.                                                                 *
 *   07/12/1996 JLB : Handles compressed file header.                                          *
 *=============================================================================================*/
template<class T>
MixFileClass<T>::MixFileClass(char const * filename, PKey const * key) :
        IsDigest(false),
        IsEncrypted(false),
        IsAllocated(false),
        Filename(0),
        Count(0),
        DataSize(0),
        DataStart(0),
        HeaderBuffer(0),
        Data(0)
{
        if (filename == NULL) return;   // ST - 5/9/2019

        /*
        **      Check to see if the file is available. If it isn't, then
```

```
**      no further processing is needed or possible.
*/
if (!Force_CD_Available(RequiredCD)) {
        Prog_End("MixFileClass Force_CD_Available failed", true);
        if (!RunningAsDLL) {    //PG
                Emergency_Exit(EXIT_FAILURE);
        }
}

T file(filename);               // Working file object.
Filename = strdup(file.File_Name());
FileStraw fstraw(file);
PKStraw pstraw(PKStraw::DECRYPT, CryptRandom);
Straw * straw = &fstraw;

if (!file.Is_Available()) return;

/*
**      Stuctures used to hold the various file headers.
*/
FileHeader fileheader;
struct {
        short First;            // Always zero for extended mixfile format.
        short Second;           // Bitfield of extensions to this mixfile.
} alternate;

/*
**      Fetch the first bit of the file. From this bit, it is possible to detect
**      whether this is an extended mixfile format or the plain format. An
**      extended format may have extra options or data layout.
*/
int got = straw->Get(&alternate, sizeof(alternate));

/*
**      Detect if this is an extended mixfile. If so, then see if it is encrypted
**      and/or has a message digest attached. Otherwise, just retrieve the
**      plain mixfile header.
```

```
        */
        if (alternate.First == 0) {
                IsDigest = ((alternate.Second & 0x01) != 0);
                IsEncrypted = ((alternate.Second & 0x02) != 0);

                if (IsEncrypted) {
                        pstraw.Key(key);
                        pstraw.Get_From(&fstraw);
                        straw = &pstraw;
                }
                straw->Get(&fileheader, sizeof(fileheader));

        } else {
                memmove(&fileheader, &alternate, sizeof(alternate));
                straw->Get(((char*)&fileheader)+sizeof(alternate), sizeof(fileheader)-sizeof(alternate));
        }

        Count = fileheader.count;
        DataSize = fileheader.size;
//BGMono_Printf("Mixfileclass %s DataSize: %08x   \n",filename,DataSize);Get_Key();
        /*
        **      Load up the offset control array. If RAM is exhausted, then the mixfile is invalid.
        */
        HeaderBuffer = new SubBlock [Count];
        if (HeaderBuffer == NULL) return;
        straw->Get(HeaderBuffer, Count * sizeof(SubBlock));

        /*
        **      The start of the embedded mixfile data will be at the current file offset.
        **      This should be true even if the file header has been encrypted because the file
        **      header was cleverly written with just the sufficient number of padding bytes so
        **      that this condition would be true.
        */
        DataStart = file.Seek(0, SEEK_CUR) + file.BiasStart;
//      DataStart = file.Seek(0, SEEK_CUR);

        /*
```

```
        **      Attach to list of mixfiles.
        */
        List.Add_Tail(this);
}

/***********************************************************************************************
 * MixFileClass::Offset -- Determines the offset of the requested file from the mixfile system.*
 *                                                                                             *
 *    This routine will take the filename specified and search through the mixfile system      *
 *    looking for it. If the file was found, then the mixfile it was found in, the offset      *
 *    from the start of the mixfile, and the size of the embedded file will be returned.       *
 *    Using this method it is possible for the CCFileClass system to process it as a normal    *
 *    file.                                                                                    *
 *                                                                                             *
 * INPUT:   filename    -- The filename to search for.                                         *
 *                                                                                             *
 *          realptr     -- Stores a pointer to the start of the file in memory here. If the    *
 *                         file is not in memory, then NULL is stored here.                    *
 *                                                                                             *
 *          mixfile     -- The pointer to the corresponding mixfile is placed here. If no      *
 *                         mixfile was found that contains the file, then NULL is stored here. *
 *                                                                                             *
 *          offset      -- The starting offset from the beginning of the parent mixfile is     *
 *                         stored here.                                                        *
 *                                                                                             *
 *          size        -- The size of the embedded file is stored here.                       *
 *                                                                                             *
 * OUTPUT:  bool; Was the file found? The file may or may not be resident, but it does exist   *
 *                and can be opened.                                                            *
 *                                                                                             *
 * WARNINGS:   none                                                                            *
 *                                                                                             *
 * HISTORY:                                                                                    *
 *   10/17/1994 JLB : Created.                                                                 *
 *=============================================================================================*/
template<class T>
bool MixFileClass<T>::Offset(char const * filename, void ** realptr, MixFileClass ** mixfile, long * offset, long * size)
```

```
{
        MixFileClass<T> * ptr;

        if (filename == NULL) {
assert(filename != NULL);//BG
                return(false);
        }

        /*
        **      Create the key block that will be used to binary search for the file.
        */
        long crc = Calculate_CRC(strupr((char *)filename), strlen(filename));
        SubBlock key;
        key.CRC = crc;

        /*
        **      Sweep through all registered mixfiles, trying to find the file in question.
        */
        ptr = List.First();
        while (ptr->Is_Valid()) {
                SubBlock * block;

                /*
                **      Binary search for the file in this mixfile. If it is found, then extract the
                **      appropriate information and store it in the locations provided and then return.
                */
                block = (SubBlock *)bsearch(&key, ptr->HeaderBuffer, ptr->Count, sizeof(SubBlock), compfunc);
                if (block != NULL) {
                        if (mixfile != NULL) *mixfile = ptr;
                        if (size != NULL) *size = block->Size;
                        if (realptr != NULL) *realptr = NULL;
                        if (offset != NULL) *offset = block->Offset;
                        if (realptr != NULL && ptr->Data != NULL) {
                                *realptr = (char *)ptr->Data + block->Offset;
                        }
                        if (ptr->Data == NULL && offset != NULL) {
                                *offset += ptr->DataStart;
```

```
                        }
                        return(true);
                }

                /*
                **      Advance to next mixfile.
                */
                ptr = ptr->Next();
        }

        /*
        **      All the mixfiles have been examined but no match was found. Return with the non success flag.
        */
assert(1);//BG
        return(false);
}
```