Understanding Projector Databases

by Ron Karr

(with modifications by Greg Branche)

Apple Computer, Inc.

gregb@apple.com

Understanding Projector Databases

Purpose of this document

This document describes the structure of the information in Projector databases. The goal is to provide enough information to help someone to repair broken Projector databases.

Possible causes of corruption are discussed.

There are several tools which can be helpful in diagnosing problems. A section of this document describes these, where they can be found, and how they can be used to repair Projector databases.

The structure of Projector databases

A Projector database consists of a single file called ProjectorDB of type `'MPSP'` and creator `'MPS '`. This section describes how some of the information in the database is structured and what it means.

The files recMgr.h, recMgrPriv.h, and nameTable.h from the Projector sources are important references. The necessary information from them is at the end of this document.

Internally, Projector divides projects into pages of size 2K. There are two special pages, the **zero page** ($0- $7FF) and **bitmap page** ($800-$FFF). The remaining pages are used to store the "records" in which Projector's data are kept. There are 12 types of records; each page contains one or more records of a given type.

Each page contains header information. See "recMgrPriv.h" for a description of the headers for each page. The tool DumpDB (described below in the section on "Tools") is useful for looking at page-level information.

Page-level data

The following is sample output from DumpDB, followed by descriptions of the fields. The ProjectorDB that was dumped was the “Projector Example” supplied with MPW.

Page Zero

```
PAGE #0             HEADER      EOF: 008800      PAGESIZE: 0800

CheckSum:     5A69F06B
PageDiskAdr:    000000
Stamp:            REPP
Version:          0002
ModCount:       000016
PageSize:         0800
FirstRecord:    00101A
eof:            008800
FreePages:      000001
RecTypeCount:     000C
FreeRec[0]:     001000      Project
FreeRec[1]:     003000      File
FreeRec[2]:     003800      Rev
FreeRec[3]:     002000      Comment
FreeRec[4]:     007800      Data
FreeRec[5]:     004800      SymbolicNames
FreeRec[6]:     005000      FileNames
FreeRec[7]:     006000      RevNames
FreeRec[8]:     001800      Authors
FreeRec[9]:     004000      Resource
FreeRec[10]:    008000      Delta
FreeRec[11]:    002800      Log
RecoverID:      000000
```

The fields on page zero are described by the struct DBHeader.

The header of each page contains a checksum, which is a 32-bit value obtained by summing each long word in the page. The checksum field of all pages beyond the bitmap page is not used and is always set to zero. The zero and bitmap page checksums are used. If you change any fields on those pages manually, the checksums must be adjusted to reflect the change.

PageDiskAdr is the relative disk address of the page. It's always a multiple of $800 (2K). PageSize is always $800.

Stamp is always "REPP". **Version** may be 2 or 3. (Version 3 databases were introduced with MPW Shell/SourceServer 3.5, primarily to address limitations in the size of the NameTable.)

ModCount is not relevant.

FirstRecord is always $101A (address of the first record, i.e. the Project record).

eof is the physical size of the ProjectorDB file. It should be the same for all pages, and should match the actual EOF of the file.

FreePages is not relevant.

RecTypeCount should always be $C (12): the number of record types.

The values in the **FreeRec**[i] array are pointers to the first page that contains a free record slot of type i. (The names of the record types are displayed above: i.e. type 0 is the Project Record, type 11 is the Log Record, etc.) When assigning a new record of a given type, Projector looks in the FreeRec array to see where there is a free slot for that type of record. If the value is zero, it means there are no free slots, and a new full page of that record type must be assigned.

RecoveryID is normally zero, unless a recovery operation is taking place (see "Recovery" below).

Page 1 (Bitmap)

```
PAGE #1            BITMAP      EOF: 008800      PAGESIZE: 0800

 CheckSum:     800107EF
 PageDiskAdr:    000800
 RecordSize:       0000

 PAGE#                              PAGES
 000000        FFEF  8000  0000  0000  0000  0000  0000  0000
```

The bitmap page contains a short header (struct PageID) followed by a bitmap. Each page in the DB is represented by a single bit in the bitmap. A page in use is represented by a "1". Bits are read from left to right. For example, page zero is represented by bit 7 of the first byte in the bitmap (which is actually the 11th byte on the page), etc. In the above example, each "F" (or binary 1111) represents 4 pages that are in use. (An unused page would be represented by a zero, so the value $E (1110) would mean that the 4th page was free.) The above example shows 11 consecutive used pages, followed by 1 free page, followed by 5 more used pages. This is consistent with the file size of $8800, since each page is $800 in size.

There is room on the bitmap page for 16304 bits, representing 16304 pages, or a size of 33,390,592 bytes ($1FD8000). Note that this is arrived at by calculating 8*(PageSize-sizeof(PageID)). If the file exceeds that size, an additional bitmap page is assigned at the end of the file, and then further allocation of pages proceeds. If that one gets used up, another one is assigned, etc.

Page 2 & Beyond

```
   PAGE #2             RECORD      EOF: 008800      PAGESIZE: 0800

   CheckSum:       000000
   PageDiskAdr:    001000
   RecordSize:       002C
   CurRecCount:      0001
   MaxRecCount:      002D
   RecordType:         00
   filler:             00
   filler1:          0000
   RecvrID:        000000
   NextFreePage:   000000

   00101A        Project Record
        PrevRec:      000000
        NextRec:      000000
        SubRec0:      00201A         Comment
        SubRec1:      00481A         SymbolicNames
        SubRec2:      00501A         FileNames
        SubRec3:      00301A         File
        SubRec4:      00181A         Authors
        SubRec5:      00281A         Log
```

Each of the remaining pages contains one or more records of a given type. Each page, of whatever type, begins with a standard header (shown above), whose fields have the following meaning:

The number of records on that page should be equal to the value **CurRecCount**. **MaxRecCount** is the maximum number of records that will fit on a page, which is determined by **RecordSize**.

filler, **filler1** & **RecvrID** are not relevant. **CheckSum** should be zero.

NextFreePage refers to a page containing a free record of this type, if non-zero. Recall that in page zero, there is a pointer to the first page containing a free record of the given type. That page can, in turn, point to another page with a free slot.

The records themselves start at an offset of $1A into the page. Each record contains a **pointer section** and a **data section**. The pointer section (shown in the output above) contains a pointer to the *next* & *previous* records of a given type. (Since the listing above shows a Project Record, and a project contains only one of these records, these pointers are NULL.) For those records containing sub-records, the pointer section also contains pointers to the first record of each of the sub-record types. For example, a project record contains a pointer to each of its sub-records (Comment, SymbolicNames, FileNames, File, Authors & Log). The File Record it points to represents the "first" file in the project. That File Record will have NULL as its *previous* pointer, and its *next* pointer will point to the next File Record in the project. The File Records in turn contain pointers to their sub-records. The value of the pointers are the file offsets of the beginning of the record.

The hierarchy of record types

Project : *There is 1 and only 1 project record in a project and it's always on the first available page (location $1000)*

----**Comment**

----**SymbolicNames** (*nametable*)

----**FileNames** (*nametable*)

----**Files**: *There is 1 file record for each file in the project.*

    ----**Comment**

    ----**Revision**: *There is one revision record for each revision in each file in the project.*

        ----**Comment**

        ----**Data**

        ----**Resource**

        ----**Delta**

    ----**RevisionNames** (*nametable*)

----**Authors** (*nametable*)

----**Log**

Note that Comment records may belong either to the Project, a File, or a Revision. SymbolicNames, FileNames, Files, Authors and Log records belong to the Project; Revisions and RevisionNames belong to a File; Data, Resource and Delta records belong to a Revision.

The *data section* of each record is described by its structure (as outlined in recMgr.h). Only the data contents of NameTables is shown in the output of DumpDB, and then only with the –d or –adr options. (See the DumpDB description later in this document.) The command DumpProject (an undocumented built-in Shell command) shows some of this information for Project, File and Revision records.

Data, Delta and Resource records contain data which is indeterminate in size. Therefore there can be any number of, say, Data Records for a given revision; the Data Records are linked by their *previous* & *next* pointers.

Record types 5-8 (symbolicNames, fileNames, revisionNames and authors) have an internal structure known as a *nametable* (see "nameTable.h"). All the instances of a given record are linked together to form a single table, containing all the relevant names. For example, all the FileName records for a project are concatenated to make up its FileName table.

The file "nametable.h" defines the structure of a "struct TheNameTable", and its substructures. This chart, showing the number of bytes for each item, may be more helpful in actually debugging nametables:

The structure of name tables

<table>
<tr><th>Bytes
(v2/v3)</th><th colspan="2">Description</th></tr>
<tr><td>4/4</td><td colspan="2">size of entire table in bytes</td></tr>
<tr><td>2/2</td><td colspan="2">largest ID ever used in offset table</td></tr>
<tr><td>2/4</td><td colspan="2">offset of next name; used by nmNextName</td></tr>
<tr><td>2/2</td><td colspan="2">unused</td></tr>
<tr><td>2/2</td><td colspan="2">flag to indicate if table has been modified</td></tr>
<tr><td>1/1</td><td colspan="2">record type</td></tr>
<tr><td>1/1</td><td colspan="2">unused</td></tr>
<tr><td>2/2</td><td colspan="2">number of elements (numElements) <i>(determines # of entries to follow)</i></td></tr>
<tr><td>numElements
* 4/6</td><td colspan="2"><b>offsetTable:</b> series of entries with the following structure (struct OffsetEntry)</td></tr>
<tr><td></td><td>2/2</td><td>ID corresponding to name</td></tr>
</table>

	2/4	offset of ID from start of nameList
numElements elements of varying size	**nameList** begins here**:** contains *numElements* elements; each element has a fixed part & variable part, as follows	
	2/4	offset of next element from start of nameList
	2/2	ID corresponding to this element
	1/1	isDummy/isObsolete flag (for RevisionNames, true if it's a dummy revision; for FileNames, if file is obsolete)
	1/1	*not relevant*
	*/1	isLocked flag, true if name cannot be changed (version 3 only)
	*/1	isObsoleteName flag, true if name is obsolete (version 3 only)
	C string	this is the name itself
	C string	comment associated with name (version 3 only)
	C string	password (version 3 only)
	4 * number of IDElements	list of **IDElements**
		2: fileID
		2: revID

note: for all record types except SymbolicNames, there is only 1 IDElement, consisting of 2 NULLs. For SymbolicNames, there is a series of them, corresponding to the fileID-revID pairs that make up the SymbolicName, and terminated by an element containing 2 NULLs.

Occasionally there have been problems where nametables have been corrupted. This problem is very likely to show up with ProjVerify (using the -verbose option will indicate exactly where the problem is). For example, there have been cases where offsets were wrong, thus causing some internal inconsistency. It's fairly straightforward to determine what the offsets should be by looking at the data.

Recovery

In order to insure the integrity of its databases, Projector uses a technique of copying all pages that are to be changed to the resource fork of the ProjectorDB. If anything goes wrong with the update process (such as a crash), the project can be recovered by writing the old pages back from the resource fork to the data fork.

Therefore it is important not to attempt to "fix" a ProjectorDB if it has a resource fork. The project should be mounted first; then opened (by selecting the project in the Check Out window); this will cause the recovery to take place and remove the resource fork.

It is also important not to install any standard resources, thereby producing a valid resource fork, into a ProjectorDB file. Doing so will cause Projector to assume that a recovery operation is required.

It may be useful to save a copy of "before" and compare it with "after", in case of problems.

Tools useful in diagnosing problems

DumpDB

Syntax:

```
   DumpDB [-f] [-r] [-rec 0x# | -page #[,#]] [-d] [-adr] projectfile
```

Description

DumpDB is an unsupported tool written early in the development of Projector. It displays a summary of all the information regarding each page of the ProjectorDB.

DumpDB displays

- the header fields for each page
- the bitmap on page 1
- the pointer sections of each record in the project. This is useful for tracing the record hierarchy.

Parameters

projectfile

Full or partial pathname; the name "ProjectorDB" is optional.

Options

-adr

Display file address of data, along with record data.

-d

Display record data (currently, only shows data for NameTable records)

-f

Show free records

-r

Show resource fork of ProjectorDB

-rec *num*

Display page containing record *num*, where *num* is the disk address of the record. The supplied number is interpreted as hexadecimal (as is the display of the record numbers in the tool's output)

-page *#[,#]*

Display pages starting with first #, and continuing on through page ,#. If the ending page number is not supplied, only the page specified will be displayed.

The following is some sample output from DumpDB

```
PAGE #0             HEADER      EOF: 008800      PAGESIZE: 0800

CheckSum:     5A69F06B
PageDiskAdr:    000000
Stamp:            REPP
Version:          0002
ModCount:       000016
PageSize:         0800
FirstRecord:    00101A
eof:            008800
FreePages:      000001
RecTypeCount:     000C
FreeRec[0]:     001000      Project
FreeRec[1]:     003000      File
FreeRec[2]:     003800      Rev
FreeRec[3]:     002000      Comment
FreeRec[4]:     007800      Data
FreeRec[5]:     004800      SymbolicNames
FreeRec[6]:     005000      FileNames
FreeRec[7]:     006000      RevNames
FreeRec[8]:     001800      Authors
FreeRec[9]:     004000      Resource
FreeRec[10]:    008000      Delta
FreeRec[11]:    002800      Log
RecoverID:      000000


PAGE #1             BITMAP      EOF: 008800      PAGESIZE: 0800

CheckSum:     800107EF
PageDiskAdr:    000800
RecordSize:       0000

PAGE#                              PAGES
000000        FFEF  8000  0000  0000  0000  0000  0000  0000
```

```
PAGE #2             RECORD      EOF: 008800      PAGESIZE: 0800

CheckSum:       000000
PageDiskAdr:    001000
RecordSize:       002C
CurRecCount:      0001
MaxRecCount:      002D
RecordType:         00
filler:             00
filler1:          0000
RecvrID:        000000
NextFreePage:   000000

00101A        Project Record
     PrevRec:       000000
     NextRec:       000000
     SubRec0:       00201A        Comment
     SubRec1:       00481A        SymbolicNames
     SubRec2:       00501A        FileNames
     SubRec3:       00301A        File
     SubRec4:       00181A        Authors
     SubRec5:       00281A        Log


PAGE #3             RECORD      EOF: 008800      PAGESIZE: 0800

CheckSum:       000000
PageDiskAdr:    001800
RecordSize:       01F4
CurRecCount:      0001
MaxRecCount:      0004
RecordType:         08
filler:             00
filler1:          0000
RecvrID:        000000
NextFreePage:   000000

00181A        Authors Record
     PrevRec:       000000
     NextRec:       000000
```

```
PAGE #4             RECORD      EOF: 008800      PAGESIZE: 0800

CheckSum:       000000
PageDiskAdr:    002000
RecordSize:       007C
CurRecCount:      0007
MaxRecCount:      0010
RecordType:         03
filler:             00
filler1:          0000
RecvrID:        000000
NextFreePage:   000000

00201A        Comment Record
     PrevRec:      000000
     NextRec:      002096

002096        Comment Record
     PrevRec:      00201A
     NextRec:      000000

002112        Comment Record
     PrevRec:      000000
     NextRec:      00218E

00218E        Comment Record
     PrevRec:      002112
     NextRec:      000000

00220A        Comment Record
     PrevRec:      000000
     NextRec:      002286

002286        Comment Record
     PrevRec:      00220A
     NextRec:      000000

002302        Comment Record
     PrevRec:      000000
     NextRec:      000000
```

```
PAGE #5             RECORD      EOF: 008800      PAGESIZE: 0800

CheckSum:       000000
PageDiskAdr:    002800
RecordSize:       01F4
CurRecCount:      0001
MaxRecCount:      0004
RecordType:         0B
filler:             00
filler1:          0000
RecvrID:        000000
NextFreePage:   000000

00281A        Log Record
     PrevRec:       000000
     NextRec:       000000
```

DumpProject

Description

DumpProject is an undocumented built-in MPW command.

The default output of DumpProject is a listing of the following information

- the data in the project record
- the data in each file record
- the data in each revision record for each file record

Syntax:

```
DumpProject [-t] | ([-p] [-f] [-c] [-n] [-rmID number] [-fix] project)
```

Parameters

project

Full or partial pathname; must include filename, i.e. "ProjectorDB")

Options

-t

List project tree (shows file info about all mounted projects and their checkoutdirs)

-p

List project info only

-f

Show only file info, not revision info

-c

List comment where applicable

-n

List all the name tables (author, filename, revname, not symbolic names)

-rmID *num*

Remove author ID *num* from project

-fix

Attempt to fix the project (unimplemented)

ProjVerify

Syntax:

```
ProjVerify [projectfile] [-comp (compressedfile | -nc ) ] [-verbose]
```

Description

ProjVerify can help determine the integrity of a Projector database. It will find many, but not all, errors. In "-verbose" mode, ProjVerify provides diagnostic information which can be used to track down & fix problems.

ProjVerify also provides an option to "compact" a ProjectorDB. This can useful after using the "deleteRevisions" command. When pages are freed up within a Projector database, the file does not normally get smaller; there is simply space available to be used in the future. ProjVerify can compact a Projector database by removing the unused pages.

Status

ProjVerify can return the following status codes:

0	no errors
1	syntax error
2	error found in database

Parameters

projectfile

Full or partial pathname. If not supplied, the file ProjectorDB in the current directory will be used.

Options

-comp *compressedfile* | -nc

Compress the ProjectorDB file. The compressed file will be written to *compressedfile* or will be compressed in place if the -nc option is supplied.

-verbose

Display progress of validation sequence.

What the program checks

- The main thing that ProjVerify does is to check for the consistency of pointers. This is important, because if any of these are wrong, conceivably the entire database could be unusable. Record pointers are checked in hierarchical order. (See above for description of record hierarchy). All of the pointers are checked to make sure the records they point to are indeed there.

- Various fields in the DB Header (page 0) and bitmap page (page 1) are checked.

- Those record types which are designated as nametables have additional checking to verify consistency of data within the nametables.

- The database keeps track of pages which contain free slots for each type of record, by means of a linked list. The pointers in this list are checked for consistency.

What the program does not check

- The data sections of the records themselves are not checked. Therefore, it would be possible to corrupt a database by changing some bytes within a Data Record, or some other type of record, and this would not show up in ProjVerify. Usually in such cases the damaged will be localized to a particular revision or file.

Diagnostic messages

- The number of bytes that will be saved if compaction is done is displayed.
- A status of 2 is returned if there are any errors in the validation. Compaction never proceeds if there have been any errors.
- In non-verbose mode, the error messages are not terribly informative, certainly not to the user. They simply indicate whether or not there are any problems in the database.
- In "-verbose" mode, the entire record hierarchy is displayed (i.e. the address of each record with its sub-records). Information about each name table is also displayed.
- There are certain errors which abort the checking , because the remaining data is not necessarily valid. In other cases, the checking proceeds all the way even though an error has been found.

DumpFile

A standard MPW tool. Sometimes needed to dump the exact contents of a page, or part thereof, once the addresses of the needed data have been determined. It is useful to use the "-r" option which displays a range of bytes, e.g.

`dumpfile ProjectorDB -r $3800,$3FFF` --This displays the data on page 7 of the project. (0x3800 ÷ 0x800 = 7)

ProjectInfo

The built-in Projector command, with various options. Can be used when the project is mounted. Corresponds somewhat to the data in dumpProject.

A disk editor (e.g. FEdit, Norton Utilities)

Needed for actually changing data in the files.

Causes of corruption

The main types of corruption that have been encountered are as follows:

- There have been known bugs—now fixed—where a database could be corrupted when operating under low-disk-space conditions.

- In 1990 some experimental versions of the MPW Shell, containing erroneously generated code due to bugs in the C compiler, were inadvertently released to members of the system software team. As a result, some of their databases became corrupt.

- There was a bug in the "recovery" code which wrote some garbage to the 2nd bitmap page in a file over 32 Meg in size, thereby corrupting the project.

- Any kind of read/write error, whether as a result of network problems, SCSI problems, or other, can potentially have devastating consequences. At best, one or more characters in a file could be wrong. At worst, changing a pointer or other value in the database could make it impossible to check out certain files or even use the project at all. (The ProjVerify tool is designed to detect such conditions).

Header files

The following source listings are current as of MPW Shell version 3.5d5, 11/24/98.

ProjID.h

```
/*
 * ProjID.h - structure definition for a ProjectorID
 *
 * Copyright © 1994
 * Apple Computer, Inc.
 * All Rights Reserved
 *
 * Extracted from various places by
 * Greg Branche, 3/11/94
 *
 * The reason I extracted this was so that the definition
 * of the ProjID structure would be in one, and only one
 * place, which could then be #include'd where needed.
 */

#ifndef __PROJID__
#define __PROJID__

/*
 * This structure is used to uniquely identify a project.
 */
struct ProjID {
  unsigned long crDateTime;
  unsigned long crTickCount;
};

typedef struct ProjID ProjID, *pProjID, **hProjID;

#endif /* __PROJID__ */
```

recmgr.h

```
/* RecMgr.h  */
/*******-------------------------------------------------------------------------
  NAME
           RecMgr.h - public header file for the Record Manager (Projector).

  AUTHOR
           Copyright Apple Computer, Inc. 1987-1998
           All Rights Reserved
-------------------------------------------------------------------------*******/
#ifndef RECMGR_H
#define RECMGR_H

#include <Types.h>
#include "shareDefs.h"
#include "characters.h"
#include "ProjID.h"

typedef unsigned char       uchar;
typedef unsigned short  ushort;
typedef unsigned long       ulong;
typedef ulong           DiskAdr;

// all structures within a ProjectorDB must conform to 68k architecture alignment

#if defined(powerc) || defined (__powerc)
#pragma options align=mac68k
#endif

/*******-------------------------------------------------------------------------
  Define record types - Add new record types to the end of the list.
-------------------------------------------------------------------------*******/
typedef enum {
  ProjectRecType=0,
  FileRecType,
  RevRecType,
  CommentRecType,
  DataRecType,
  SymbolicNamesRecType,
  FileNamesRecType,
  RevNamesRecType,
  AuthorsRecType,
  ResourceRecType,
  DeltaRecType,
  LogRecType,
  NoParent=255
  } RecordType;
#define MaxRecType (LogRecType+1)

/*******-------------------------------------------------------------------------
  Define the data section for each record type
-------------------------------------------------------------------------*******/
typedef struct ProjectRecord {               /* Project Record */
  short    authorID;
  ProjID   projectID;
  } ProjectRecord;

/*-*-*-*
 Extra Description of the fields of the FileRecord:
  checkOutAuthorID:     If this is > 0 then the latest revision on main trunk is checked
                        out for modification by this person.  Otherwise (file is free)
                        the value should be < 0.  In this case the negative is the
                        ID of the author that created (i.e. checked in) the latest
                        revision.  In order to maintain project compatibility with
                        old project the value can be == 0.  In this case the file is
                        also free, but I can't tell who made the latest revision on the
                        main trunk.
 *-*-*-*/
typedef struct FileRecord {                  /* File Record */
  short            fileID;
  short            authorID;                 // of person who added file to the project
  short            checkOutAuthorID;
  short            latestRevID;              // id of latest rev (excluding dummy rev)
  unsigned long    modDate;                  // last time any revision was created
  short            compressionFormat;
  } FileRecord;

/*-*-*-*
 Extra Description of the fields of the RevRecord:
  checkOutFlag:         If this is > 0 then this revision is checked out “checkOutFlag”
                        number of times.  If it is == 0 then the revision is free.  If it
                        is < 0 then this is a dummy revision and the parent revision id
                        is -checkOutFlag.
  dateTime:             If this is a dummy revision then dateTime is the time that it
```

```
                        was checked out.  Otherwise this is the time that the revision was
                        checked in.
 *-*-*-*/
typedef struct RevRecord {                   /* Revision Record */
  short            revID;
  short            authorID;                 // person who created this revision
  short            checkOutFlag;
  unsigned long    dateTime;
  short            compressionFormat;
  character        task[40];
  } RevRecord;

typedef struct CommentRecord {               /* Comment Record */
  character    data[114];
  } CommentRecord;

/*
 *The LogHeader is the format for each item in the LogRecord.
 *The LogRecord.theData is just a bunch of bytes, so this record
 *imposes a structure over the individual sections of the LogRecord.
 *The very last item in the log will be padded with 0's to signify
 *end of log.
 *
 *The MyLogRecord is a structure to impose a format over the
 *LogRecord.  The first two bytes are always an offset to the
 *beginning of the data in the remaining section of the record.
 *
 *For example:
 *    myRec.offset = 100
 *    myRec.theData[100] = start of logging data
 *    logHeader = &myRec.theData[100];
 *
 *The cmdLine can spill over different LogRecords, but currently, the
 *authorID, timeStamp, and cmdLen are not split.
*/

struct LogHeader {
  short            authorID;
  unsigned long    timeStamp;
  int              cmdLen;
  character        cmdLine[UNBOUNDED_ANSI];
};

typedef struct LogHeader LogHeader, *LogHeaderP;

#define GENERICRECORDSIZE 490

struct LogRecord {
  short        offset;
  character    theData[GENERICRECORDSIZE-sizeof(short)];
};

typedef struct LogRecord LogRecord, MyLogRecord, *MyLogRecordP;

typedef struct {                             /* SymNames, FileNames,   */
  chardata[GENERICRECORDSIZE];           /* Authors, Resource,...  */
  }   SymbolicNamesRecord,               /*  records               */
      FileNamesRecord,
      RevNamesRecord,
      AuthorsRecord,
      ResourceRecord,
      DeltaRecord;

typedef struct {                             /* Data Record            */
  char    data[980];
  }   DataRecord;

/*******--------------------------------------------------------------------------
  Define current state size.  This is the minimum length (in bytes) of the buffer
  in GetState(character *buffer).
--------------------------------------------------------------------------*******/
#define StateSize (MaxRecType*(4*sizeof(long)))

typedef enum {
  DefaultK =0,
  OneK     =1024,
  TwoK     =2048,
  FourK    =4096,
  EightK   =8192,
  SixteenK =16384
  } RMPage ;                             /* Page Size     */


#if defined(powerc) || defined(__powerc)
#pragma options align=reset
#endif

/*******--------------------------------------------------------------------------
  Record Manager version number - stored in the file when the project file is
  created and must be incremented everytime the record definitions are changed.
--------------------------------------------------------------------------*******/
#define RM_Version2 2
```

```
#define RM_Version3 3
#define RM_VersionLatest RM_Version3
// name of environment variable user can set to default to a specific version number
#define RM_ProjectorVersString  ((character *) "ProjectorVersion")

#endif
```

recMgrPriv.h

```
/* RecMgrPriv.h version 13 */
/*******-------------------------------------------------------------------------
  NAME
           RecMgrPriv.h - private header file for the Record Manager (Projector).

  AUTHOR
           Copyright Apple Computer, Inc. 1987-1998
           All Rights Reserved
-------------------------------------------------------------------------*******/
#ifndef RECMGRPRIV_H
#define RECMGRPRIV_H


#include "recmgr.h"


// all structures within a ProjectorDB must conform to 68k architecture alignment

#if defined(powerc) || defined (__powerc)
#pragma options align=mac68k
#endif

/*******-------------------------------------------------------------------------
  Define the DB file's Header format.  This data structure lives
  at the beginning of page zero of the DB file.
-------------------------------------------------------------------------*******/
typedef struct DBHeader {
  ulong    checkSum;            /* the checksum for the page                            */
  DiskAdr  pageDiskAdr;         /* always set to 0                                      */
  ulong    stamp;               /* a unique string identifying the file as a project file */
  ushort   version;             /* version the project file format                      */
  ulong    modCount;            /* cummulative count of all page writes                 */
  RMPage   pageSize;            /* size in bytes of each page                           */
  DiskAdr  firstRecord;         /* the address of the Project record                    */
  ulong    eof;                 /* the number of bytes in the data fork                 */
  ulong    freePages;           /* number of free pages in the bit map                  */
  ushort   recTypeCount;        /* the number of record types                           */
  DiskAdr  freeRec[MaxRecType]; /* address of a page containing a free record slot      */
  long     recoveryID;          /* id of all non zero/bitmap pages in the resource fork */
  }  DBHeader;

/*******-------------------------------------------------------------------------
  Define the DB file's page header format.  The first header format (PageID) is
  common to all pages excepting page zero.  The second header format (PageHeader)
  is common to all pages excepting page zero and the bit map pages.
  All pages except for page zero begin with a page header, followed by one or
  more fixed size record slots.  Record slots are an even number of bytes
  in length.
-------------------------------------------------------------------------*******/
typedef struct PageID {
  ulong        checkSum;        /* the checksum for the page                   */
  DiskAdr      pageDiskAdr;     /* the page's disk address                     */
  ushort       recSize;         /* record size (bytes) of records on the page  */
  } PageID;

typedef struct PageHeader {
  ulong        checkSum;        /* the checksum for the page                   */
  DiskAdr      pageDiskAdr;     /* the page's disk address                     */
  ushort       recSize;         /* record size (bytes) of records on the page  */
  ushort       curRecCount;     /* the number of records currently on the page */
  ushort       maxRecCount;     /* maximum number of records on this page      */
  RecordType   recType;         /* record type of records on this page         */
  uchar        filler;          /* future expansion                            */
  ushort       filler1;         /* future expansion                            */
  long         recoveryID;      /* id of all non z/bpages in the resource fork */
  DiskAdr      nextFreePage;    /* adr of next page containing a free record   */
  } PageHeader;

/*******-------------------------------------------------------------------------
  Define the record header.  Each record in the DB consists of three components.
               Record
                   Header                fixed size - same format for all records.
                   PointerSection        depends on record type, defined in record
                                             descriptor table below.
                   DataSection           depends on record type, defined in RecMgr.h
-------------------------------------------------------------------------*******/
typedef struct RecHeader {
  uchar        inUse;           /* 1 = in use, 0 = free                    */
  RecordType   recType;         /* record's type, e.g. project,file,rev,   */
  DiskAdr      prev;            /* adr of previous record                  */
  DiskAdr      next;            /* adr of next record                      */
  } RecHeader;
```

```
/*******------------------------------------------------------------------------
  Define record descriptors.  Each descriptor contains the list of subordinate
  records associated with that record type.

  Important:  The record's type is the index into the array of record
  descriptors.
------------------------------------------------------------------------*******/
typedef struct {
  RecordType   subRecs[MaxRecType];
  } RecDescr;

/*******------------------------------------------------------------------------
  Define the record State structure.  The current state (i.e. position) of each
  record type is maintained using an array of struct RecState.
------------------------------------------------------------------------*******/
typedef struct {
  RecordType   parent;          /* parent record type or 255 "NoParent"    */
  uchar        filler;
  DiskAdr      current;         /* current record                          */
  DiskAdr      next;            /* next record                             */
  }   RecState;

/*
 * special definition for SpeedMount stuff (found in forest.c)
 * Extracted from forest.c by Greg Branche, 3/11/94, so that this struct
 * can be included within the 68k structure alignment pragmas, instead of
 * being hidden within a source file where it originally received default
 * alignment by whatever compiler was being used at the time.
 */

struct Page2 {
  PageHeader   page2Header;              /* the standard page header                    */
      // ulong         checkSum;         /* the checksum for the page                   */
      // DiskAdr       pageDiskAdr;      /* the page's disk address                     */
      // ushort        recSize;          /* record size (bytes) of records on the page  */
      // ushort        curRecCount;      /* the number of records currently on the page */
      // ushort        maxRecCount;      /* maximum number of records on this page      */
      // RecordType    recType;          /* record type of records on this page         */
      // uchar     filler;
      // ushort    filler1;              /* future expansion                            */
  RecHeader    page2RecHeader;           /* standard record header (pointer section)    */
      // uchar         inUse;            /* 1 = in use, 0 = free                        */
      // RecordType    recType;          /* record's type, e.g. project,file,rev,       */
      // DiskAdr       prev;             /* adr of previous record                      */
      // DiskAdr       next;             /* adr of next record                          */
  DiskAdr      commentRecAdr;            /* disk address of first comment record        */
  DiskAdr      symbolicNamesRecAdr;      /* disk address of first symbolic names record */
  DiskAdr      filenamesRecAdr;          /* disk address of first FileNames record      */
  DiskAdr      fileRecAdr;               /* disk address of first File record           */
  DiskAdr      authorsRecAdr;            /* disk address of first Authors record        */
  DiskAdr      logRecAdr;                /* disk address of first Log record            */
  ProjectRecord        page2ProjRec;     /* the data section of the Project Record      */
      // short         authorID;
      // ProjID        projectID;
};

typedef struct Page2 Page2, *pPage2, **hPage2;


#if defined(powerc) || defined(__powerc)
#pragma options align=reset
#endif

/*******------------------------------------------------------------------------
  Miscl definitions.
------------------------------------------------------------------------*******/
enum {ZPage=1,BPage=2};
#define NumOfBuffers 3
typedef enum NextField {ZeroNextFields=0,SetNextFields} NextField;
#define BitMapDiskAdr (1*pageSize)
#define ProjFileCr 'MPS '
#define ProjFileType    'MPSP'
#define ProjStamp  ((ulong)'REPP')

#define DebugRMCalls    (debugFlags & 1)
#define DebugPageIO(debugFlags & 2)
#define DebugRecovery  (debugFlags & 4)

#define ResForkHdrSize 256

#endif /* RECMGRPRIV_H */
```

nametable.h

```
/*******--------------------------------------------------------------------------

NAME
      nametable.h  --  Name table commands header

AUTHOR
      Copyright Apple Computer, Inc. 1987-1998
      All Rights Reserved.


--------------------------------------------------------------------------*******/
#ifndef NAMETABLE_H
#define NAMETABLE_H

#include <StdDef.h>
#include <Files.h>

#include "characters.h"
#include "defs.pj.h"
#include "recmgr.h"
#include "shareDefs.h"

#define MAX_BRANCH          8            /* num chars in branch e.g. abb == 3                    */
#define MAX_BRANCH_NAME     MAX_BRANCH+2 /* num chars in valid branch string e.g, "abb1" == 5    */

#define isObsolete isDummy
// defining isObsolete = same field as isDummy in NameElement & TableName

typedef enum { NoError,
               SyntaxError,
               ProcessError,
               MemoryError,
               DefineNameError,
               TooManyEntriesError,
               DeleteFailed} CmdResult;

// all structures within a ProjectorDB must conform to 68k architecture alignment

#if defined(powerc) || defined (__powerc)
#pragma options align=mac68k
#endif

typedef struct TableName {
  short            id;              /* id for name: {1:SHRT_MAX}           */
  unsigned char    *name;           /* : string                            */
  unsigned char    *extra;          /* comment or password: string         */
  char             isDummy;         /* is record a “dummy”: {false:true}   */
  char             isLocked;        /* is name locked:  {false:true}       */
  char             isObsoleteName;  /* is name obsolete:  {false:true}     */
} TableName, *PTableName;

typedef struct IdElement {
  short            fileId;          /* : {SHRT_MIN:SHRT_MAX}  */
  short            revId;           /* : {0:SHRT_MAX}         */
} IdElement, *PIdElement, **HIdElement;

typedef struct OffsetEntry2 {
  short            id;              /* id for name: {1:SHRT_MAX}                                */
  unsigned short   offset2;         /* offset of element from start of nameList: {0:USHRT_MAX}  */
} OffsetEntry2, *POffsetEntry2;

typedef struct OffsetEntry3 {
  short            id;              /* id for name: {1:SHRT_MAX}                                */
  unsigned long    offset3;         /* offset of element from start of nameList: {0:ULONG_MAX}  */
} OffsetEntry3, *POffsetEntry3;

typedef struct OffsetEntry
{
  union
  {
      OffsetEntry2 offsetEntry2;
      OffsetEntry3 offsetEntry3;
  };
} OffsetEntry, *POffsetEntry;

typedef struct NameElement2 {
  unsigned short   next2;                   /* byte offset of next element from start of nameList: {0:USHRT_MAX} */
  short            id;                      /* name’s id in offsetTable: {1:SHRT_MAX}          */
  char             isDummy;                 /* is this name for a dummy record: {false:true}   */
  char             referenced;              /* used to detect cycles when expanding name lists */
  unsigned char    name[UNBOUNDED_ANSI];    /* name: string                                    */
  IdElement        idList[UNBOUNDED_ANSI];  /* list of fileId,revId pairs: (0,0) terminated    */
} NameElement2, *PNameElement2;
```

```
typedef struct NameElement3 {
  unsigned long    next3;                    /* byte offset of next element from start of nameList: {0:ULONG_MAX} */
  short            id;                       /* name’s id in offsetTable: {1:SHRT_MAX}          */
  char             isDummy;                  /* is this name for a dummy record: {false:true}   */
  char             referenced;               /* used to detect cycles when expanding name lists */
  char             isLocked;                 /* is name locked:  {false:true}                   */
  char             isObsoleteName;           /* is name obsolete:  {false:true}                 */
  unsigned char    name[UNBOUNDED_ANSI];     /* name: string                                    */
  unsigned char    comment[UNBOUNDED_ANSI];  /* comment: string                                 */
  unsigned char    password[UNBOUNDED_ANSI]; /* password: string                                */
  IdElement        idList[UNBOUNDED_ANSI];   /* list of fileId,revId pairs: (0,0) terminated    */
} NameElement3, *PNameElement3;

typedef struct NameElement
{
  union
  {
      NameElement2 nameElement2;
      NameElement3 nameElement3;
  };
} NameElement, *PNameElement;

typedef struct TheNameTable2 {
  unsigned long    size;            /* size of entire table in bytes: {sizeof(NameTable):ULONG_MAX} */
  unsigned short   lastId;          /* largest id ever used in offsetTable: {1:SHRT_MAX}           */
  unsigned short   nextName2;       /* offset of next name; used by nmNextName: {0:USHRT_MAX}      */
  short            unused;          /* unused                                                       */
  short            tableDirty;      /* flag to indicate if table has been modified: {false:true}    */
  RecordType       recType;         /* specific type of data record: {SymbolicNamesRecType:AuthorsRecType} */
  unsigned short   numElements;     /* #elements in list, also #entries in table: {0:SHRT_MAX}      */
  OffsetEntry2     offsetTable2[UNBOUNDED_ANSI];
  NameElement2     nameList2[UNBOUNDED_ANSI];
} TheNameTable2;

typedef struct TheNameTable3 {
  unsigned long    size;            /* size of entire table in bytes: {sizeof(NameTable):ULONG_MAX} */
  unsigned short   lastId;          /* largest id ever used in offsetTable: {1:SHRT_MAX}           */
  unsigned long    nextName3;       /* offset of next name; used by nmNextName: {0:ULONG_MAX}      */
  short            unused;          /* unused                                                       */
  short            tableDirty;      /* flag to indicate if table has been modified: {false:true}    */
  RecordType       recType;         /* specific type of data record: {SymbolicNamesRecType:AuthorsRecType} */
  unsigned short   numElements;     /* #elements in list, also #entries in table: {0:SHRT_MAX}      */
  OffsetEntry3     offsetTable3[UNBOUNDED_ANSI];
  NameElement3     nameList3[UNBOUNDED_ANSI];
} TheNameTable3;

typedef struct TheNameTable
{
  union
  {
      TheNameTable2 nameTable2;
      TheNameTable3 nameTable3;
  };
} TheNameTable, *PNameTable, **HNameTable;

#if defined(powerc) || defined(__powerc)
#pragma options align=reset
#endif

/*
  NOTE:   The number of entries in a NameTable will really be limited to about
          (USHRT_MAX ÷ avgsizeof(NameElement)). For example: a minimum size NameElement
          is 12 bytes which means that after only 5.4K entries the offset field in the
          offsetTable will not be able to reach the next NameElement. A more realistic
          case is where the average NameElement is 16 bytes (6 character revision name
          plus overhead), which turns out to allow about 3640 revisions per file (or 10
          revisions per day for a year).

  Note:   This limitation is now removed.  The NameTable size is now ULONG_MAX
          instead of USHRT_MAX.  However, the count of name table entries is still
          limited to USHRT_MAX.
*/

typedef enum
{
  kNothingExtra,
  kCommentExtra,
  kPasswordExtra
} NameExtras;


#endif
```