pdbdecode
Extract Palm Electronic Book Contents

revision: 1.25 of 2009/09/03 21:16:31
printed: 18.x.2011 15:08
printed from version being edited!!!

	Section	Page
Introduction	1	1
Implementation language and environment	2	3
PDB structures	13	8
PDB file header	14	8
PDB record headers	26	12
PDB record bodies	33	13
Determine file type	37	15
`TEXt`: Palm DOC decoder	41	16
`TEXt`: Decoding uncompressed text	47	18
`TEXt`: Decoding compressed text	48	19
`zTXT`: Weasel Reader decoder	54	22
`MOBI`: MobiPocket decoder	69	29
`MOBI`: Decoding Huffman compressed data	78	32
`PNRd`: Peanut Reader Decoder	79	33
Utility routines	93	40
Index	105	47

What I cannot create, I do not understand.
— notation left at the top of
Richard Feynman's
blackboard when he died.

1. Introduction.

[[[See TEXbook p106, for a different way to do webref.]]]

There are a number of different electronic book formats for the Palm handheld device, each with a different software application for the Palm. The electronic book files are all wrapped as standard Palm database (PDB) files. The PDB format provides the overall structure of the data (for example, the record sizes and headers); the e-book formats each provide a different encoding of the book data within that structure (for example, how the text is compressed). There are at least six different e-book formats in general use, distinguishable by the type and creator tags in their file headers:

- Traditional Palm DOC format (type `TEXt`, creator `REAd`),
- Weasel Reader compression (type `zTXT`, creator `GPlm`),
- Palm Reader/Peanut Press (type `PNRd`, creator `PPrd`),
- Mobipocket (type `BOOK`, creator `MOBI`),
- iSilo (type `ToGo`, creator `ToGo`),
- Plucker (type `Data`, creator `Plkr`).

[[[Plucker format is described in the Plucker documentation downloadable bundle. In the 1.8 version, it's file `plucker-1.8/manual/DBFormat.html`*.]]]*

This program decodes the first four of these formats and recovers the source text (and sometimes images) for the e-books. Providing a way of decoding the content is a vital step in converting from one e-book format to another. The actual conversion to a new format is beyond the scope of this program — indeed, the reader software packages generally provide adjunct software for the desktop computer to convert text into the appropriate PDB files.

Additional issues out-of-scope for this program:

- *Converting the text markup:* We don't attempt to process or strip the markup data interspersed with the text describing font style and size, paragraph breaks, and so forth. Each of the e-book formats uses a different markup language: Palm Reader uses a very light-weight markup, for example; Weasel expects text with no markup; Mobipocket relies on the Open E-Book markup language, which is a very complete HTML-like language and is the emerging standard used for electronic publishing. Similarly, we don't do line breaking on the text bodies, which often end up with each paragraph in a single long line, nor do we attempt to fold imbedded line-ending characters to their Unix or Windows standard forms.
- *Digital rights management and encryption:* This program as originally written doesn't handle any form of digital rights management (DRM), in which e-books are encrypted or keyed to a particular user or device. E-books for both Mobipocket and Palm Reader can be DRM-locked, either for the exclusive use of the purchaser, or to expire after a pre-set time as with a lending library. At the risk of violating the Digital Millennium Copyright Act, someone could use CWEB's change file mechanism to add code to this program

which breaks or circumvents DRM and encryption schemes. (Breaking the encryption probably involves running reader software under a simulator and debugger.)

- *Comparison of source texts:* If we have multiple versions of a source text — say, the original version and one we just decoded from an e-book form — we may want to compare them. To do this we would need to do a word-by-word comparison of the two versions, probably ignoring possible markup. As useful as this functionality might be (and as much as I expect to need it), it is the purview of a different piece of software.

2. Implementation language and environment. There exist numerous Perl scripts for Palm file manipulation, including at least two modules for `TEXt/REAd` format and two for `zTXT/GPlm`. It would be logical to write modules for the other formats, but writing a C program reduces the dependencies on Perl modules, and the Perl base environment. It also makes a program that will be most-easily portable to other operating systems.

But why CWEB rather than straight C? Because the formats are complicated, and the literate programming model gives us a better way to produce the notes on the formats. Also, changes for particular operating systems can be easily handled by CWEB's *change file* mechanism, which allows modifications to be made to program source via an supplementary source file leaving the original source unchanged. It means we need to ship the raw C and a printable version of the source, since we can't assume everyone has CWEB installed, but I am willing to accept that inconvenience.

And, similarly, why write in C, rather than the trendier C++? While CWEB supports C++, the advantages of C++ for this project are nil. It is unnecessary to build a **class** for each e-book type, since the decoding for the individual types is already separated into its own subroutine. Even if it made sense to build a **class** for each e-book type for data privacy, we already have the necessary isolation by making the data for each type local to its decoding routine. The notion of making each e-book type a derived **class** of a generic `PDB` **class** is likewise silly, since we only decode one type of book at a time, and the data isolation is not necessary in such a small project. The one syntactic advantage of C++ for our purposes, the ability to include declarations anywhere in the code body, is obviated by CWEB features which allow us to explain code in a different order than the computer will see it. By making sections of code that are additions to previous-defined sections, we can freely intermix declaration sections with code sections.

The ultimate CWEB reference is *The CWEB System of Structured Documentation*, by Donald E Knuth and Silvio Levy (Addison-Wesley, 1993, ISBN 0-201-57569-8. The software itself can be downloaded from

`http://www-cs-faculty.stanford.edu/~knuth/cweb.html`.

For further information on literate programming, see Daniel Mall's literate programming web site

`http://www.literateprogramming.com/`.

The list of additional references includes Knuth's original literate programming paper,

`http://www.literateprogramming.com/knuthweb.pdf`.

This software was developed on Microsoft Services for Unix release 3.5, running on Microsoft Windows XP Service Pack 2. *[[[Before final release it should also be built and tested on Linux and as a native Windows program.]]]*

3. The program outline is simple:

```
⟨ include files 4 ⟩
⟨ data types 13 ⟩
⟨ prototypes 23 ⟩
⟨ global data 7 ⟩
⟨ global macros 95 ⟩
⟨ functions 21 ⟩
main(int argc, char *argv[])
{
  ⟨ local data in main 5 ⟩
  ⟨ main program 6 ⟩
  exit(0);
}
```

4. We'll need a number of include files, so let's begin listing the obvious ones. We'll add to this list as we continue.

```
⟨ include files 4 ⟩ ≡
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
```

See also sections 20, 22, 29, and 59.

This code is used in section 3.

5. We begin the body of the program by processing the flags. Let's first stipulate the "usage" message, and some local data.

```
⟨ local data in main 5 ⟩ ≡
  int i;
  const char *usage = "Usage:␣pdbdecode␣[-a]␣[-i]␣[[-h]␣[-r]␣[\
      -O]]␣[-s]␣[-v]␣[-w]␣[-o␣base]␣file";
```

See also sections 16 and 19.

This code is used in section 3.

6. The usage message shows the synopsis of the command line. To enumerate the flags:

`-h` Display the contents of the global `PDB` header, which contains data about the file itself.

`-r` Display all the individual record headers, which contain data such as location of the record within the file.

`-0` Display the data in record 0 of the `PDB` file, which contains data about the contents of the file.

`-a` Turn on all three of `-h`, `-r`, and `-0`.

`-i` Appends non-graphic supplemental data — bookmarks, annotations — to *stdout*. The graphic data is ignored unless there is also a `-o`.

`-o`*loc* Specify the output location. If *loc* is an existing directory, or if the name ends with a slash, we produce the output files in that directory. If *loc* is a simple name, we generate file names of the form *loc*nnnnn, where nnnnn is a sequential five-digit number. If there is no `-o` flag, we only output the base text to *stdout*. See subroutine *next_ofile*( ) for details.

`-s` Only output the supplemental data, such as bookmarks and pictures, and don't output the body text. This is the opposite of running without `-o` to only output the body text. Even if we don't output it, we still extract the body text as a consistency check.

`-v` Display the version and copyright banners and exit.

`-w` Issue extra warnings .....

[[[We should use POSIX *getopt( ) for parsing the command-line options, but I don't think it's available as such on Windows, which would make the Windows port harder.]]]*

```
⟨ main program 6 ⟩ ≡
  i = 1;
  while (i < argc ∧ (argv[i][0]) ≡ '-') {
    switch (argv[i][1]) {
    case 'h': flags |= FLAG_HEADERS;
      break;
    case 'r': flags |= FLAG_RECHDRS;
      break;
    case '0': flags |= FLAG_RECZERO;
      break;
    case 'a': flags |= (FLAG_HEADERS | FLAG_RECHDRS | FLAG_RECZERO);
      break;
    case 's': flags |= FLAG_SUPONLY;
      break;
    case 'i': flags |= FLAG_SUPTEXT;
      break;
    case 'o': o_name = (argv[i][2]) ? (&argv[i][2]) : (argv[++i]);
      break;
    case 'v': ⟨ print version and copyright 8 ⟩
      exit(0);
    default: fatal("bad␣flag␣<%c>\n%s", argv[i][1], usage);
    }
    i++;
  }
  if (i ≥ argc) fatal("no␣filename?\n%s", usage);
```

See also sections 10, 12, 17, 18, 24, 28, 31, and 40.

This code is used in section 3.

7. Now that we've used the command-line flags, let's define the bit flags for them, and the global data we need to support them.

```
#define FLAG_HEADERS #01
#define FLAG_RECHDRS #02
#define FLAG_RECZERO #04
#define FLAG_SUPONLY #08
#define FLAG_SUPTEXT #10
```

```
⟨ global data 7 ⟩ ≡
  int flags = 0;
  char *o_name = NULL;
  FILE *ofp;
```

See also sections 11, 14, 26, 30, 37, and 38.

This code is used in section 3.

8. Of course, if we're printing the version number, we don't need a flag bit: we just display the version, build date, copyright information, and call it quits. We go to some effort to strip just the raw revision number and date out of the strings our revision control system provides, which makes this code a little more baroque than it might be otherwise. In case stripping the data fails, we just print the whole strings.

```
⟨ print version and copyright 8 ⟩ ≡
  {
    char *revisionId = "$Revision:␣1.25␣$";
    char *revisionDate = "$Date:␣2009/09/03␣21:16:31␣$";
    char *idp = strchr(revisionId, '␣');
    char *ide = idp ? strchr(idp + 1, '␣') : NULL;
    char *dap = strchr(revisionDate, '␣');
    char *dae = dap ? strchr(dap + 1, '␣') : NULL;
    int leni = ide − idp − 1;
    int lend = dae − dap − 1;

    if (idp ≡ NULL ∨ ide ≡ NULL ∨ dap ≡ NULL ∨ dae ≡ NULL)
      printf("pdbdecode,␣%s␣%s\n", revisionId, revisionDate);
    else
      printf("pdbdecode,␣version␣%.*s␣(%.*s)\n", leni, idp + 1, lend, dap + 1);
  }
```

See also section 9.

This code is used in section 6.

9. In the case where we displaying the version information, we also want to display the copyright information. We put out the basic copyright here, and if we later have a section discussing the licensing terms, we may make an addition to the section.

```
⟨ print version and copyright 8 ⟩ +≡
  printf("Copyright,␣Jeffrey␣L␣Copeland\n");
```

10. The input file is also straight-forward. Because we intend to port this to Windows, we add the "b" option to *fopen*, which is necessary for that platform.

```
⟨ main program 6 ⟩ +≡
  infilename = argv[i];
  ifp = fopen(infilename, "rb");
  if (ifp ≡ NULL) {
    perror(infilename);
    exit(1);
  }
```

11. Since we have only one input file, we make its **FILE** ∗ global, so that we don't have to pass it in calls to our read utilities.

```
⟨ global data 7 ⟩ +≡
  char *infilename;
  FILE *ifp;
```

12. Open the basic output file if necessary; see subroutine *next_ofile*( ) for the mechanics of supplemental files.

```
⟨ main program 6 ⟩ +≡
  if (o_name ≡ NULL) ofp = stdout;
  else ofp = next_ofile("", "");
```

13. PDB structures.
We'll use the names the Palm documentation uses for basic scalar types. They are stored in the file in network order, most significant byte first. (Except for **BYTE**, which replaces the Palm **unsigned char** type **Byte** to avoid a conflict with the types used by *zlib*.)

```
format Word int
format DWord int
format BYTE int
```

⟨ data types 13 ⟩ ≡
```
typedef unsigned short Word;
typedef unsigned long DWord;
typedef unsigned char BYTE;
```
This code is used in section 3.

14. PDB file header.
We begin by defining the PDB file header block. The format is defined in the Palm developer documentation set at

`http://www.palmos.com/dev/support/docs/fileformats/Intro.html`.
`http://www.palmos.com/dev/tech/overview.html`

The individual fields in the file header are described as comments in the structure below. Additional notes appear in the next section.

```
#define LEN_TTL 32
#define LEN_HDR ((LEN_TTL + 4 + 4) + 3 * sizeof(Word) + 8 * sizeof(DWord))
```

⟨ global data 7 ⟩ +≡
```
struct {
  char title[LEN_TTL];                    /* title, NUL-terminated */
  Word attributes;                        /* flags */
  Word version;                           /* application-specific version tag */
  DWord creationDate;                     /* file dates */
  DWord modificationDate;                 /* ... */
  DWord lastBackupDate;                   /* ... */
  DWord modificationNumber;               /* how many times file was updated */
  DWord appInfoId;                        /* offset within file to appInfo block */
  DWord sortInfoId;                       /* offset within file to sorfInfo block */
  char type[5];                           /* type identification */
  char creator[5];                        /* creator identification */
  DWord uniqueIdSeed;                     /* random seed for record identifiers */
  DWord recordOffset;                     /* offset to record zero */
  Word numRecords;                        /* total number of records in file */
} hdr;
size_t PDBsize;                           /* total size of PDB file */
```

15. Some notes on the data elements in *hdr*:

Numeric data in a PDB file is stored in big-endian format. The title is already expected to be `NUL` terminated, but we include an extra byte in *type* and *creator* so we can add a terminating `NUL`.

The names *appInfoId* and *sortInfoId* are misnomers.

There is a handwave about the number of padding bytes following the header if there is not a record list, however, we ignore this since non-empty e-books always have a record list.

The total size for the PDB file is declared in the previous section, but to be pedantic, it's not part of the header block; we actually get the data from external sources.

The *attributes* field is bit-flags, and the possible flag values are defined here:

```
#define ATTR_RESOURCE #01
#define ATTR_READONLY #02
#define ATTR_DIRTY #04
#define ATTR_BACKUP #08
#define ATTR_OKNEWER #10
#define ATTR_RESET #20
#define ATTR_OPEN #40
#define ATTR_LAUNCHABLE #200
```

16. We need a buffer into which to read the file header.

```
⟨ local data in main 5 ⟩ +≡
  BYTE b[LEN_HDR], *p = b;
```

17. Once defined, we can read the file header as a lump and store it a field at a time.

```
⟨ main program 6 ⟩ +≡
  ck_read(b, LEN_HDR);
  store_String(p, hdr.title, LEN_TTL);
  store_Word(p, hdr.attributes);
  store_Word(p, hdr.version);
  store_DWord(p, hdr.creationDate);
  store_DWord(p, hdr.modificationDate);
  store_DWord(p, hdr.lastBackupDate);
  store_DWord(p, hdr.modificationNumber);
  store_DWord(p, hdr.appInfoId);
  store_DWord(p, hdr.sortInfoId);
  store_ZString(p, hdr.type, 4);
  store_ZString(p, hdr.creator, 4);
  store_DWord(p, hdr.uniqueIdSeed);
  store_DWord(p, hdr.recordOffset);
  store_Word(p, hdr.numRecords);
```

18. We also want to populate the file size, which we need to get from a call to the operating system.

```
⟨ main program 6 ⟩ +≡
  if (stat(infilename, &sb) < 0)
        fatal("unable␣to␣stat␣input␣file");
  PDBsize = (size_t) sb.st_size;
```

19. We need a local structure for capturing the file statistics.

```
⟨ local data in main 5 ⟩ +≡
  struct stat sb;
```

20. Some additional include files declare the data types used by *stat()*.

```
⟨ include files 4 ⟩ +≡
#include <sys/types.h>
#include <sys/stat.h>
```

21. We have a routine to print out our Palm-format date values. The PDB file's dates have an epoch of 1 January 1904, rather than the POSIX standard 1 January 1970. Note that many PDB-generating programs have bugs and use the Unix date, rather than the Palm one. This accounts for early twenty-first-century PDB files apparently being dated in the late 1930s. We try to display the intended date, rather than the actual one, by assuming that the dates in our files should be after the POSIX epoch. If the time is earlier than the POSIX epoch, we append a warning `'??'`. (Just for reference: `DATE_OFFSET` as a POSIX **time_t** is 1 January 2036 at 4pm.)

```
#define DATE_OFFSET 2082844800
#define SCRATCH_BUF_SIZE 64

⟨ functions 21 ⟩ ≡
  char *show_time(DWord t_palm)
  {
    static char buf[SCRATCH_BUF_SIZE];
    int posix_date = (t_palm < DATE_OFFSET) ? 1 : 0;
    time_t tt = (time_t)(posix_date ? t_palm
         : (t_palm − DATE_OFFSET));

    memset(buf, 0, SCRATCH_BUF_SIZE);
    if (t_palm ≡ 0L) return buf;

    strftime(buf, SCRATCH_BUF_SIZE, "%Y/%m/%d@%H:%M:%S", localtime(&tt));
    if (posix_date) strcat(buf, "??");
    return buf;
  }
```

See also sections 33, 34, 35, 41, 50, 54, 70, 79, 88, 94, 97, 98, 99, 100, and 102.

This code is used in section 3.

22. We need the include file declaring time data types:

```
⟨ include files 4 ⟩ +≡
#include <time.h>
```

23. We need to collect the function prototypes for our utility functions. We'll make a habit of declaring them right after their definitions.

```
⟨ prototypes 23 ⟩ ≡
  char *show_time(DWord);
```

See also sections 36, 39, 53, 89, and 103.

This code is used in section 3.

24. If we just wanted to dump the file header data, let's do that.

```
#define EXIT_IF_LAST_FLAG(x) flags &= ~(x);
          if (¬flags) exit(0);
          else printf("\n");

⟨ main program 6 ⟩ +≡
  if (flags & FLAG_HEADERS) {
    ⟨ dump file header 25 ⟩
    EXIT_IF_LAST_FLAG(FLAG_HEADERS);
  }
```

25. This section prints a reasonably formatted rendition of the file header. *[[[We really should decompose attributes and identify what bits are set in text form.]]]*

```
⟨ dump file header 25 ⟩ ≡
  printf("title:␣<%s>\n", hdr.title);
  printf("attributes:␣0x%x,␣version␣0x%x,␣mod␣number␣%d\n", hdr.attributes,
      hdr.version, hdr.modificationNumber);
  printf("Ctime␣0x%x␣=␣%u\t%s\n", hdr.creationDate, hdr.creationDate,
      show_time(hdr.creationDate));
  printf("Mtime␣0x%x␣=␣%u\t%s\n", hdr.modificationDate, hdr.modificationDate,
      show_time(hdr.modificationDate));
  printf("Btime␣0x%x␣=␣%u\t%s\n", hdr.lastBackupDate, hdr.lastBackupDate,
      show_time(hdr.lastBackupDate));
  printf("type/creator:␣%s/%s;␣", hdr.type, hdr.creator);
  printf("app/sort:␣%d/%d;␣", hdr.appInfoId, hdr.sortInfoId);
  printf("seed:␣%d\n", hdr.uniqueIdSeed);
  printf("record␣offset␣0x%lx,␣number␣of␣records␣%d,␣file␣size␣%d\n",
      hdr.recordOffset, hdr.numRecords, PDBsize);
```

This code is used in section 24.

26. PDB record headers.
Now we come to the individual record headers. There is one of these for each one of the *hdr.numRecords* records in the PDB file.

```
⟨ global data 7 ⟩ +≡
  struct PDBrec_header {
    DWord offset;
    BYTE attributes;
    BYTE uniqueID[3];
  };
```

27. Again, the *attributes* byte is a set of bit fields.

```
#define REC_DELETE #80
#define REC_DIRTY #40
#define REC_BUSY #20
#define REC_SECRET #10
#define REC_CATEGORY #0F
```

28. Let's allocate and read the record headers now.

```
⟨ main program 6 ⟩ +≡
  rec_hdrs = (struct PDBrec_header **)
       ck_malloc(sizeof(struct PDBrec_header *) * hdr.numRecords);
  for (i = 0; i < hdr.numRecords; i++) {
    rec_hdrs[i] = (struct PDBrec_header *)
         ck_malloc(sizeof(struct PDBrec_header));
    rec_hdrs[i]→offset = read_DWord( );
    ck_read(&(rec_hdrs[i]→attributes), 1);
    ck_read(rec_hdrs[i]→uniqueID, 3);
  }
```

29. ⟨ include files 4 ⟩ +≡

```
#include <stdlib.h>
```

30. ⟨ global data 7 ⟩ +≡

```
  struct PDBrec_header **rec_hdrs;
```

31. We may also want to dump the record headers:

```
⟨ main program 6 ⟩ +≡
  if (flags & FLAG_RECHDRS) {
    ⟨ dump record headers 32 ⟩
    EXIT_IF_LAST_FLAG(FLAG_RECHDRS);
  }
```

32. *[[[Here, also, we should decompose the attribute bits into text.]]]*

```
⟨ dump record headers 32 ⟩ ≡
  for (i = 0; i < hdr.numRecords; i++) {
    printf("record␣%d:␣offset␣0x%x,␣attr␣0x%x,␣size␣%d,␣id␣0x%02x%02x%02x\n",
        i, rec_hdrs[i]→offset, rec_hdrs[i]→attributes, size_pdb_record(i),
        rec_hdrs[i]→uniqueID[0], rec_hdrs[i]→uniqueID[1], rec_hdrs[i]→uniqueID[2]);
  }
```

This code is used in section 31.

33. PDB record bodies.

We need a utility routine for grabbing a record from the PDB file. We can assume we're reading the records sequentially, but it's safer to plan that we're going to read arbitrary records. We return *NULL* if the record number requested is out of range, or we can't seek to the record position.

Normally, this routine allocates memory to hold the record, and it's the caller's responsibility to free it. Just for insurance, we allocate one extra byte and load it with a `NUL`.

```
⟨ functions 21 ⟩ +≡
  BYTE *read_pdb_record(const Word recnum)
  {
    long recpos;
    size_t recsize;
    BYTE *buf;
    if (recnum ≥ hdr.numRecords) return NULL;
    recpos = rec_hdrs[recnum]→offset;
    if (fseek(ifp, recpos, SEEK_SET) < 0) return NULL;
    recsize = size_pdb_record(recnum);
    buf = malloc(recsize + 1);
    memset(buf, 0, recsize + 1);
    ck_read(buf, recsize);
    return buf;
  }
```

34. We need the postulated routine to get the size of a given PDB record. This depends on the table *rec_hdrs* being global data. And strictly speaking, this should probably return a **DWord** not a **size_t**.

```
⟨ functions 21 ⟩ +≡
  size_t size_pdb_record(const Word recnum)
  {
    size_t recsize;
    if (recnum ≡ (hdr.numRecords − 1))
      recsize = PDBsize − rec_hdrs[recnum]→offset;
    else
      recsize = rec_hdrs[recnum + 1]→offset
          − rec_hdrs[recnum]→offset;
  }
```

35. We also provide a version of the read routine that operates into a pre-allocated buffer. The caller is responsible for ensuring the buffer is big enough to contain the data. We return a pointer to the buffer specified.

```
⟨ functions 21 ⟩ +≡
  BYTE *read_pdb_noalloc(const Word recnum, BYTE *obuf)
  {
    long recpos;
    size_t recsize;
    if (recnum ≥ hdr.numRecords) return NULL;
    recpos = rec_hdrs[recnum]→offset;
    if (fseek(ifp, recpos, SEEK_SET) < 0) return NULL;
    recsize = size_pdb_record(recnum);
    ck_read(obuf, recsize);
    return obuf;
  }
```

36.
```
⟨ prototypes 23 ⟩ +≡
  BYTE *read_pdb_record(const Word);
  BYTE *read_pdb_noalloc(const Word, BYTE *);
  size_t size_pdb_record(const Word);
```

37. Determine file type.
At this point, we'd like to know what file type we have, and dispatch to the appropriate converter.
Let's set up a table of types and creators *vs* decoding routines.

```
⟨ global data 7 ⟩ +≡
  struct dispatch {
    char type[5];
    char creator[5];
    void ((*decode)(void));
  } disp[ ] = {
    {"TEXt", "REAd", TEXt_decode},
    {"zTXT", "GPlm", zTXT_decode},
    {"BOOK", "MOBI", MOBI_decode},
    {"PNRd", "PPrs", PNRd_decode},
  };
#define DISPATCH_SIZE ((sizeof (disp))/(sizeof (disp[0])))
```

38. Also, for convenience, let's remember what kind of book we're decoding.

```
⟨ global data 7 ⟩ +≡
  enum book_formats {
    fmt_ERROR = −1, fmt_TEXt = 0, fmt_zTXT, fmt_MOBI, fmt_PNRd
  };
  enum book_formats current_book_format = fmt_ERROR;
```

39. ⟨ prototypes 23 ⟩ +≡

```
  void TEXt_decode(void);
  void zTXT_decode(void);
  void MOBI_decode(void);
  void PNRd_decode(void);
```

40. ⟨ main program 6 ⟩ +≡

```
  for (i = 0; i < DISPATCH_SIZE; i++) {
    if (memcmp(disp[i].type, hdr.type, 4) ≡ 0
          ∧ memcmp(disp[i].creator, hdr.creator, 4) ≡ 0) {
      current_book_format = i;
      disp[i].decode( );
      break;
    }
  }
  if (i ≥ DISPATCH_SIZE)
        fatal("I␣don't␣know␣how␣to␣decode␣file␣type␣%s/%s.",
      hdr.type, hdr.creator);
```

41. TEXt: Palm DOC decoder.

The Palm DOC format was first used in the TealDoc and Aportis Reader applications. It was reverse-engineered and is now widely understood. For two articles on generating the format, see:

`http://alumnus.caltech.edu/~copeland/work/palm.html`

and

`http://alumnus.caltech.edu/~copeland/work/palmcomp.html`.

Those sources provide further references.

In the uncompressed form, `TEXt` just provides the text broken up into 4096-byte records. In its compressed form, `TEXt` uses a simple, quick-to-compute, run-length encoding to compress the text.

Some versions of readers for this format provide rudimentary bookmark features, but we will ignore those.

```
⟨functions 21⟩ +≡
  void TEXt_decode(void)
  {
    ⟨TEXt: local data 42⟩
    ⟨TEXt: get record 0 43⟩
    ⟨TEXt: dump record 0? 44⟩
    ⟨TEXt: process data 46⟩
  }
```

42. The first thing we need to know about a file in format `TEXt` is the data in record 0. Record 0 for `TEXt` contains a version tag (a flag to tell whether the data is compressed), the total size of the document, the number of records, and the maximum uncompressed record size.

```
#define TEXT_RECO_SIZE (2 * sizeof(DWord) + 4 * sizeof(Word))

⟨TEXt: local data 42⟩ ≡
  BYTE *r0, *p;
  int i;
  int n;
  struct {
    Word version;
    Word reserved;
    DWord doc_size;
    Word num_recs;
    Word rec_size;
    DWord reserved2;
  } rec0;
```

This code is used in section 41.

43. If our record zero is too short, something's wrong. If it's longer than expected, we've got a `MOBI` file with a `TEXt` tag, so we just redirect our processing to *MOBI_decode*.

[[[We should now know enough to identify a `MOBI` *file explicitly, rather than by implication, so we should fix the second* **if** *statement in this section.]]]*

```
⟨ TEXt: get record 0 43 ⟩ ≡
  n = size_pdb_record(0);
  if (n < TEXT_REC0_SIZE) fatal("record␣0␣is␣too␣short!");
  if (n > TEXT_REC0_SIZE) return MOBI_decode( );
  p = r0 = read_pdb_record(0);
  store_Word(p, rec0.version);
  store_Word(p, rec0.reserved);
  store_DWord(p, rec0.doc_size);
  store_Word(p, rec0.num_recs);
  store_Word(p, rec0.rec_size);
  free(r0); r0 = NULL;
```

This code is used in section 41.

44. If we want to display record 0, we do so now, and exit.

```
⟨ TEXt: dump record 0? 44 ⟩ ≡
  if (flags & FLAG_RECZERO) {
    ⟨ TEXt: show record 0 45 ⟩
    EXIT_IF_LAST_FLAG(FLAG_RECZERO);
  }
```

This code is used in section 41.

45. ⟨ TEXt: show record 0 45 ⟩ ≡

```
  printf("Record␣0:\n");
  printf("␣␣version␣0x%x␣(%s)\n",
        rec0.version, ((rec0.version ≡ 1) ? "uncompressed"
        : ((rec0.version ≡ 2) ? "compressed" : "UNKNOWN")));
  printf("␣␣full␣uncompressed␣text␣size␣%d␣bytes\n", rec0.doc_size);
  printf("␣␣contains␣%d␣body␣records,␣with␣maximum␣uncompressed␣size␣%d\n",
        rec0.num_recs, rec0.rec_size);
```

This code is used in section 44.

46. If we're decoding the contents of the file, there is a basic bifurcation in the processing. If *rec0.version* is 1, we have an uncompressed file; if *rec0.version* is 2, it's compressed; other values cause an error.

```
⟨ TEXt: process data 46 ⟩ ≡
  if (rec0.version ≡ 1) {
    ⟨ TEXt: process uncompressed 47 ⟩
  }
  else if (rec0.version ≡ 2) {
    ⟨ TEXt: process compressed 48 ⟩
  }
  else
    fatal("undefined␣version␣in␣TEXt␣file␣0x%x", rec0.version);
```

This code is used in section 41.

47. `TEXt`: Decoding uncompressed text.
Uncompressed processing is painfully simple. Remembering that we append a `NUL` to each record as it is read, we just print the records. We also check if we're only interested in outputing the supplemental data: if that's the case for a `TEXt` format file, we really output no data.

```
⟨ TEXt: process uncompressed 47 ⟩ ≡
  for (i = 1; i ≤ rec0.num_recs; i++) {
    BYTE *buf;

    buf = read_pdb_record(i);
    if (¬(flags & FLAG_SUPONLY)) fprintf(ofp, "%s", buf);
    free(buf);
  }
```

This code is used in sections 46 and 75.

48. TEXt: Decoding compressed text.
Processing compressed data is a little more complicated. We know the uncompressed size of the maximum record from *rec0.rec_size*, so we pre-allocate the decompression buffer. Again, we check if asked to only output supplemental data.

```
⟨ TEXt: process compressed 48 ⟩ ≡
  BYTE *ubuf = malloc(rec0.rec_size + 1);
  for (i = 1; i ≤ rec0.num_recs; i++) {
    BYTE *buf;
    int e;
    size_t n = size_pdb_record(i);
    buf = read_pdb_record(i);
    ⟨ TEXt: decompress 49 ⟩
    free(buf);
    if (¬(flags & FLAG_SUPONLY)) fprintf(ofp, "%s", ubuf);
#if 0
    printf("\n\n==========\n\n");
#endif
  }
  free(ubuf);
```
This code is used in sections 46 and 75.

49. Because TEXt-style decompression is also used by MOBI format files, we'll wrap the decompression as a utility function.

The odd second clause in the **if** statement is a workaround for a bug in the Mobipocket Creator program: The uncompressed buffer ends with junk characters. As a result, we often try to put decompressed characters past the end of the output buffer. However, it appears that these can be safely ignored, so if we've returned from *TEXt_decompress* of a MOBI book with an out-of-bounds error, we can ignore it.

```
⟨ TEXt: decompress 49 ⟩ ≡
  e = TEXt_decompress(buf, n, ubuf, rec0.rec_size + 1);
  if (e ∧ current_book_format ≠ fmt_MOBI)
    fatal("overflowed␣TEXt␣uncompression␣buffer:␣record␣%d,␣error␣%d", i, e);
  if (e ∧ current_book_format ≡ fmt_MOBI) error("record␣%d,␣error␣%d", i, e);
      /* ??? */
```
This code is used in section 48.

50. This is the wrapped decompressor. We provide input and output buffers and their sizes. The output buffer size needs to be one larger than expected for a terminating `NUL`.

`TEXt` format compression is based on a simple run-length encoding. There are four classes of characters in the compressed data block:

- Characters between 0x01 and 0x08 are a byte count introducing a literal block. For example, the Latin 1 sequence "Öôç" (characters 0xD6, 0xF4 0xE7) would be encoded as 0x03, 0xD6, 0xF4, 0xE7.
- Characters from ASCII tab (0x09) through `DEL` (0x7F) and `NUL` (0x00) represent themselves.
- A byte between 0x80 and 0xBF begins a two-byte pair representing a sequence of between three and ten bytes repeated within the previous 2047 bytes of uncompressed text. This is stored by subtracting three from the length, and packing it and the distance into a **Word** as $\#8000 + (distance \ll 3) + (length - 3)$.
- Finally, characters between 0xC0 and 0xFF represent a space followed by an ASCII character between space (0x40) and `DEL` (0x7F), combined by setting the high bit of the second character. Thus, "␣J" (0x40, 0x4A) becomes 0xCA.

We provide differing error returns for each place we can go out of bounds on the decompression buffer, but in the existing callers only check for a non-zero return to indicate errors.

```
⟨ functions 21 ⟩ +≡
  int TEXt_decompress(BYTE *inbuf, const size_t insize, BYTE *outbuf, const
          size_t outsize)
  {
    BYTE *o = outbuf;
    BYTE *p = inbuf;
    BYTE *outend = outbuf + outsize;     /* terminating NUL */
    BYTE *inend = inbuf + insize − 1;
    memset(outbuf, 0, outsize);
    ⟨ TEXt: decode special case for trailing NUL 52 ⟩
    while (p ≤ inend) {
      if (o ≥ outend)
      {     /* ??? */
        ⟨ leftovers 51 ⟩     /* ??? */
        return −1;
      }     /* ??? */
      if (*p ≥ #C0) {
        *o++ = '␣';
        *o++ = (*p++) & #7F;
      }
      else if (*p ≥ #80) {
        int d = (((*p & #3F) ≪ 8) | (*(p + 1) & #F8)) ≫ 3;
        int l = (*(p + 1) & #07) + 3;
        BYTE *pp = (o − d);
        if (pp < outbuf ∨ (o + l) ≥ outend) {
          ⟨ leftovers 51 ⟩   /* ??? */
          return −2;   /* out of bounds */
        }   /* ??? */
        while (l−−) *o++ = *pp++;
        p += 2;
      }
      else if (*p ≥ #09 ∨ *p ≡ #00) {
        *o++ = *p++;
      }
      else {
        int n = *p++;
        if ((o + n) ≥ outend) {
          ⟨ leftovers 51 ⟩   /* ??? */
          return −3;
        }   /* ??? */
        while (n−− > 0) *o++ = *p++;
      }
    }
    return 0;
  }
```

51. ⟨ leftovers 51 ⟩ ≡

```
  fprintf(stderr, "inbuf␣leftovers:␣");
  while (p ≤ inend) fprintf(stderr, "0x%02x␣", *p++);
```

This code is used in section 50.

52. Some MOBI format encoders terminate each input record with a NUL. The NUL is implicit in the output buffer; its presence in the input buffer should not count against the input buffer size. In fact, the buffer size counts assume this. As a result, we ignore the NUL lest we overrun the output buffer.

⟨ TEXt: decode special case for trailing NUL 52 ⟩ ≡

```
  if (*inend ≡ #0) inend−−;
```

This code is used in section 50.

53. ⟨ prototypes 23 ⟩ +≡

```
  int TEXt_decompress(BYTE *, const size_t, BYTE *, const size_t);
```

54. zTXT: Weasel Reader decoder.

Weasel Reader was developed by John Gruenenfelder. (Palm's lawyers objected to the original name, GutenPalm.) While the reader is capable of consuming `TEXt` type files, `zTXT` files were invented for it. The `zTXT` format applies *gzip* compression to the text, providing significantly better compression than DOC format. The format is extremely well-documented at

`http://gutenpalm.sourceforge.net`.

In addition to the reader, that site also includes a `zTXT` encoder/decoder, *makeztxt*; this decoding routine is based on the *ztxt_disect()* routine from that program.

To support *gzip* compression, we require the *zlib* library and header files; sources can be obtained from

`http://www.gzip.org/zlib/`.

```
⟨ functions 21 ⟩ +≡
  void zTXT_decode(void)
  {
    ⟨ zTXT: local data 55 ⟩
    ⟨ zTXT: get record 0 56 ⟩
    ⟨ zTXT: dump record 0? 57 ⟩
    ⟨ zTXT: process data 61 ⟩
  }
```

55. We first read record 0, which contains version, record count, record size, and pointers for bookmarks and annotations. We'll describe each of these as we use them in the following sections. Notice that we don't bother storing the empty bytes specified at the end of the structure to pad *rec0* to an even size.

```
#define ZTXT_RANDOMACCESS #01
#define ZTXT_NONUNIFORM #02
#define ZTXT_REC0_SIZE
        (2 * sizeof(DWord) + 7 * sizeof(Word) + 2 * sizeof(BYTE))
⟨zTXT: local data 55⟩ ≡
  BYTE *r0, *p;
  int i;
  struct {
    Word version;
    Word numRecords;
    DWord size;
    Word recordSize;
    Word numBookmarks;
    Word bookmarkRecord;
    Word numAnnotations;
    Word annotationRecord;
    BYTE flags;
    BYTE reserved;
    DWord crc32;
  } rec0;
```

See also sections 60 and 64.

This code is used in section 54.

56. ⟨ zTXT: get record 0 56 ⟩ ≡

```
p = r0 = read_pdb_record(0);
if (size_pdb_record(0) < ZTXT_REC0_SIZE) fatal("record␣0␣is␣too␣short!");
store_Word(p, rec0.version);
store_Word(p, rec0.numRecords);
store_DWord(p, rec0.size);
store_Word(p, rec0.recordSize);
store_Word(p, rec0.numBookmarks);
store_Word(p, rec0.bookmarkRecord);
store_Word(p, rec0.numAnnotations);
store_Word(p, rec0.annotationRecord);
rec0.flags = *p++;
p++;
store_DWord(p, rec0.crc32);
free(r0); r0 = NULL;
```

This code is used in section 54.

57. If we want to display record 0, now is the time.

⟨ zTXT: dump record 0? 57 ⟩ ≡

```
if (flags & FLAG_RECZERO) {
  ⟨ zTXT: show record 0 58 ⟩
  EXIT_IF_LAST_FLAG(FLAG_RECZERO);
}
```

This code is used in section 54.

58. ⟨ zTXT: show record 0 58 ⟩ ≡

```
printf("Record␣0:\n");
printf("␣␣zTXT␣version␣0x%x␣=␣%d.%d\n", rec0.version, rec0.version ≫ 8,
      rec0.version & #FF);
printf("␣␣%d␣data␣records,␣%d␣bytes␣uncompressed␣data\n", rec0.numRecords,
      rec0.size);
printf("␣␣text␣record␣size␣%d\n", rec0.recordSize);
printf("␣␣%d␣bookmarks,␣listed␣in␣record␣%d\n", rec0.numBookmarks,
      rec0.bookmarkRecord);
printf("␣␣%d␣annotations,␣listed␣in␣record␣%d\n", rec0.numAnnotations,
      rec0.annotationRecord);
printf("␣␣flags␣0x%02x", rec0.flags);
if (rec0.flags & ZTXT_RANDOMACCESS) printf("␣(random␣access)");
if (rec0.flags & ZTXT_NONUNIFORM) printf("␣(non-uniform␣length)");
printf("\n␣␣CRC32␣=␣0x%08x\n", rec0.crc32);
```

This code is used in section 57.

59. ⟨ include files 4 ⟩ +≡

```
#include <zlib.h>
```

60.

```
#define MAXWBITS 15
⟨ zTXT: local data 55 ⟩ +≡
  BYTE *inbuf, *outbuf;
  z_stream zs;
  DWord insize = 0;
```

61. As we've noted, the zTXT format relies on the *zlib* compression library, written by Jean-loup Gailly and Mark Adler, and based on the open-source *gzip* program. Either block or global compression can be used on the zTXT text, depending on whether you want to have to load the whole text into the Palm memory at once. However, for decoding purposes, the compressed text can be treated as though it is a single block. As a result, we begin decompressing by reading all the compressed data into a single buffer.

```
⟨ zTXT: process data 61 ⟩ ≡
  for (i = 1; i ≤ rec0.numRecords; i++)
        insize += size_pdb_record(i);
  p = inbuf = ck_malloc(insize);
  for (i = 1; i ≤ rec0.numRecords; i++) {
    (void *) read_pdb_noalloc(i, p);
    p += size_pdb_record(i);
    if (p > inbuf + insize) fatal("zTXT␣file␣header␣size␣too␣small:␣%d", PDBsize);
  }
```

See also sections 62, 63, and 68.

This code is used in section 54.

62. Then we decompress the whole buffer in one step. This section is a more-or-less direct crib from the *ztxt_disect* routine in John Gruenenfelder's *makeztxt*.

```
⟨zTXT: process data 61⟩ +≡
  zs.zalloc = Z_NULL;
  zs.zfree = Z_NULL;
  zs.opaque = Z_NULL;
  zs.next_in = inbuf;
  zs.avail_in = insize;
  outbuf = zs.next_out = ck_malloc(rec0.size + 1);
  zs.avail_out = rec0.size + 1;
  if (inflateInit2(&zs, MAXWBITS) ≠ Z_OK) fatal("decompression␣init␣failed");
  i = inflate(&zs, Z_SYNC_FLUSH);
  if (zs.msg ≠ NULL) fatal("zlib␣error:␣%s", zs.msg);
  if ((i ≠ Z_STREAM_END) ∧ (i ≠ Z_OK))
    fatal("decompression␣failed:␣error␣from␣inflate");
  if (zs.avail_in > 0)
    fatal("decompression␣failed:␣output␣buffer␣not␣big␣enough");
  *(zs.next_out) = '\0';
  inflateEnd(&zs);
  if (¬(flags & FLAG_SUPONLY)) {
    fprintf(ofp, "%s", outbuf);
    fflush(ofp);
  }
```

63. If we want to save the supplemental information — if we have specified the `-o` flag — we do that now. We'll save both the bookmarks and annotations.

If there are bookmarks in the file, *rec0.numBookmarks* is greater than zero, and the bookmarks themselves are record number *rec0.bookmarkRecord*. Similarly, if there are annotations, *rec0.numAnnotations* is greater than zero, and the annotation titles and locations are specified in *rec0.annotationRecord*. Both records contain an array of offset/title pairs as described in the next section. The actual text of the annotations is in separate records, following the annotation record: the text of the first annotation is in *rec0.annotationRecord* + 1, and so on.

```
⟨zTXT: process data 61⟩ +≡
  if (o_name ≠ NULL ∨ flags & FLAG_SUPTEXT) {
    if (rec0.numBookmarks > 0) {
      ⟨zTXT: show bookmarks 65⟩
    }
    if (rec0.numAnnotations > 0) {
      ⟨zTXT: show annotations 67⟩
    }
  }
```

64. We need the bookmark and annotation structures as local data. The data structure for bookmarks and annotations is the identical. The bytes in either the bookmark record, *rec0.bookmarkRecord* or the annotation one, *rec0.annotationRecord*, are a list of the *tag* structure described here.

```
#define ZTXT_MARK_LEN 20
#define ZTXT_SNIP 60
⟨zTXT: local data 55⟩ +≡
  struct {
    DWord offset;
    BYTE title[ZTXT_MARK_LEN + 1];
  } tag;
  FILE *sup;
  BYTE *index_record;
```

65. We'll read the bookmark record, and loop through the bookmarks, printing the offset, the text of the bookmark, and the surrounding text from the body of the file.

```
⟨zTXT: show bookmarks 65⟩ ≡
  sup = (flags & FLAG_SUPTEXT) ? ofp : next_ofile("", ".bk");
  index_record = read_pdb_record(rec0.bookmarkRecord);
  for (i = 0; i < rec0.numBookmarks; i++) {
    ⟨zTXT: show a tag 66⟩
  }
  free(index_record);
  if (¬(flags & FLAG_SUPTEXT)) fclose(sup);
```

This code is used in section 63.

66. ⟨zTXT: show a tag 66⟩ ≡

```
  BYTE *pp, *ppe, temp;
  p = index_record + i * (sizeof(DWord) + ZTXT_MARK_LEN);
  store_DWord(p, tag.offset);
  store_ZString(p, tag.title, ZTXT_MARK_LEN);
  pp = outbuf + tag.offset;
  ppe = MIN(pp + ZTXT_SNIP, outbuf + rec0.size);
  if (pp < outbuf ∨ pp > (outbuf + rec0.size))
    fatal("bad␣zTXT␣bookmark␣or␣annotation␣pointer");
  temp = *ppe; *ppe = 0;
  if (flags & FLAG_SUPTEXT) fprintf(sup, "\n\n=====\n");
  fprintf(sup, "%s␣@␣%d␣->\n␣␣␣␣<%s>\n", tag.title, tag.offset, pp);
  *ppe = temp;
```

This code is used in sections 65 and 67.

67. And similarly, for annotations. However, since we also want to show the text of the annotations, we'll write them into separate files. Remember that the annotation offset is probably to the top of the screen on which the note appears, which means that the snippet of text we show will probably not contain the text *title*. *[[[I'm not sure I'm happy with the way these are output. Should we be displaying a full screen of text as the "snippet" for each annotation? That would display the text labeling the annotation, and give us context for the annotation text. What does makeztxt do?]]]*

```
⟨zTXT: show annotations 67⟩ ≡
  index_record = read_pdb_record(rec0.annotationRecord);
  for (i = 0; i < rec0.numAnnotations; i++) {
    BYTE *text;
    sup = (flags & FLAG_SUPTEXT) ? ofp : next_ofile("", ".not");
    ⟨zTXT: show a tag 66⟩
    text = read_pdb_record(rec0.annotationRecord + i + 1);
    fprintf(sup, "%s\n", text);
    free(text);
    if (¬(flags & FLAG_SUPTEXT)) fclose(sup);
  }
  free(index_record);
```

This code is used in section 63.

68. For zTXT, we want free the input and output buffers only after we've displayed the bookmarks and annotations.

```
⟨zTXT: process data 61⟩ +≡
  free(inbuf);
  free(outbuf);
```

69. MOBI: MobiPocket decoder.

Since the MobiPocket reader is available on nearly every PDA (both Palm and Pocket PC) and on cell phones, e-books in MobiPocket format are the most portable between devices. It achieves this portability by supporting the Open E-book format as its primary input language. However, since most MOBI-format books are encrypted, they are not easily portable to different formats. The MobiPocket reader is available from

http://mobipocket.com/.

The site also has several programs to convert files into MOBI format.

MOBI files are essentially a superset of TEXt format files, as we mentioned in the TEXt decoder sections. This is taken to its logical absurdity by some MOBI format files actually being tagged as TEXt. In this case, we can distinguish them from a regular TEXt because record 0 is larger for a MOBI.

In addition to the text records accounted for in record 0, a MOBI file may also contain extra records, that is, the record count in the PDB header and in record 0 may differ. Typically, these extra records will be images, and we save these as .bmp files. This means that the text portion of a MOBI file tagged as TEXt can be read by a TEXt reader such as TealDoc without confusion using the record count in record 0. However, the additional data in the extra records can be viewed by the MobiReader.

For ready examples of this format, note that Baen Books uses MOBI format with the TEXt tag for the Palm-format e-books on their web site and CDs included with their physical books.

70. Let's begin:

```
⟨ functions 21 ⟩ +≡
  void MOBI_decode(void)
  {
    ⟨ MOBI: local data 71 ⟩
    ⟨ MOBI: get record 0 72 ⟩
    ⟨ MOBI: dump record 0? 73 ⟩
    ⟨ MOBI: process data 75 ⟩
  }
```

71. The MOBI local data is identical (for the moment) to the record zero structures for TEXt format. *[[[For the moment, we're ignoring the extra information in a* MOBI *record zero, in part because we don't understand what's there. Need to add some notes about the extra stuff once we figure it out.]]]*

```
#define MOBI_REC0_SIZE (2 * sizeof(DWord) + 4 * sizeof(Word))
⟨ MOBI: local data 71 ⟩ ≡
  BYTE *r0, *p;
  int i;
  int n;
  struct {
    Word version;
    Word reserved;
    DWord doc_size;
    Word num_recs;
    Word rec_size;
    Word encrypted;
    Word reserved2;
  } rec0;
```

This code is used in section 70.

72. ⟨ MOBI: get record 0 72 ⟩ ≡

```
  n = size_pdb_record(0);
  if (n < MOBI_REC0_SIZE) fatal("record␣0␣is␣too␣short!");
  p = r0 = read_pdb_record(0);
  store_Word(p, rec0.version);
  store_Word(p, rec0.reserved);
  store_DWord(p, rec0.doc_size);
  store_Word(p, rec0.num_recs);
  store_Word(p, rec0.rec_size);
  store_Word(p, rec0.encrypted);
  free(r0); r0 = NULL;
```

This code is used in section 70.

73. If we want to display record 0, we do so now, and exit.

```
⟨ MOBI: dump record 0? 73 ⟩ ≡
  if (flags & FLAG_RECZERO) {
    ⟨ MOBI: show record 0 74 ⟩
    EXIT_IF_LAST_FLAG(FLAG_RECZERO);
  }
```

This code is used in section 70.

74. ⟨ MOBI: show record 0 74 ⟩ ≡

```
printf("Record␣0:\n");
printf("␣␣version␣0x%x␣(%s,␣%s)\n",
      rec0.version,
      ((rec0.version ≡ 1) ? "uncompressed"
      : ((rec0.version ≡ 2) ? "compressed" : "UNKNOWN")),
      ((rec0.encrypted ≡ 0) ? "unencrypted" : "encrypted"));
printf("␣␣full␣uncompressed␣text␣size␣%d␣bytes\n", rec0.doc_size);
printf("␣␣contains␣%d␣body␣records,␣with␣maximum␣uncompressed␣size␣%d\n",
      rec0.num_recs, rec0.rec_size);
```

This code is used in section 73.

75. Again, processing the data for the MOBI format is a lot like processing the data for the TEXt format. Indeed, we reuse some of the sections for processing TEXt files. However, MOBI also has a “high compression” variant, using Huffman encoding, which we also handle.

⟨ MOBI: process data 75 ⟩ ≡

```
⟨ MOBI: check for encryption 76 ⟩
if (rec0.version ≡ 1) {
   ⟨ TEXt: process uncompressed 47 ⟩
}
else if (rec0.version ≡ 2) {
   ⟨ TEXt: process compressed 48 ⟩
}
else if (rec0.version ≡ #4448) {
   ⟨ MOBI: process Huffman compressed data 78 ⟩
}
else
   fatal("undefined␣version␣in␣MOBI␣file␣0x%x", rec0.version);
if (o_name ≠ NULL) {
   ⟨ MOBI: save ancillary data 77 ⟩
}
```

This code is used in section 70.

76. We used to check for an encrypted MOBI file by trying decompress the first record and conclude it’s encrypted if the decompression fails. However, we now know what word in record 0 stores that information, and we use it instead.

⟨ MOBI: check for encryption 76 ⟩ ≡

```
{
   if (rec0.encrypted ≠ 0) fatal("can't␣process␣encrypted␣MOBI␣format");
}
```

This code is used in section 75.

77. For a MOBI file, we may have extra records at the end containing images. We can tell there are extra records if the number of records specified in record zero is smaller than the number of records in the global file header. Usually — but not always — these are images in Microsoft bmp file format, which our regular Unix image viewer *xv* recognizes. Interestingly, it appears that the images are named sequentially inside the MOBI file, so that our approach of putting the contents of the image files into sequentially-numbered bmp files is exactly right.

```
⟨MOBI: save ancillary data 77⟩ ≡
  for (i = rec0.num_recs + 1; i < hdr.numRecords; i++) {
    BYTE *buf;
    int n;
    ofp = next_ofile("", ".bmp");
    buf = read_pdb_record(i);
    n = size_pdb_record(i);
    fwrite(buf, sizeof(BYTE), n, ofp);
    free(buf); buf = NULL;
    fclose(ofp);
  }
```

This code is used in section 75.

78. MOBI: Decoding Huffman compressed data.

Here we uncompress the Huffman-compressed variant of a MOBI file.

```
⟨MOBI: process Huffman compressed data 78⟩ ≡
  fatal("oops!␣␣there's␣no␣Huffman␣decompression␣code␣here␣yet!");
```

This code is used in section 75.

79. PNRd: Peanut Reader Decoder.

The Peanut Press format is used by the Palm Reader,

`http://ereader.com/product/browse/software`.

As with the MobiReader, the site contains software to convert text into Peanut Press format.

...insert note about the format of the PDB

... structure of rec 0: Word version; 0xFF == encrypted, 0x0A == ztxt, 0x02 == trad palm compression

For the structure of the basic decoding routing, we use the same pattern we've used three times already.

```
⟨ functions 21 ⟩ +≡
  void PNRd_decode(void)
  {
    ⟨ PNRd: local data 80 ⟩
    ⟨ PNRd: get record 0 81 ⟩
    ⟨ PNRd: dump record 0? 83 ⟩
    ⟨ PNRd: process data 86 ⟩
  }
```

80. There are at least three different versions for `PNRd` files, indicated by the first **Word** in record 0.

- values greater than `0xFF` mark an encrypted gzip-compressed file — not supported here;
- `0x02` is an un-encrypted, classic-Palm compressed file — I've never seen one of them in the wild;
- `0x0A` is a gzip-compressed file — this is the most common version in the field, generated by the *DropBook* program from the EReader site.

Each of these versions has a different record 0 layout, none of which are sufficiently documented. Because the information I have about the data in record 0 is cribbed from disparate, contradictory, badly-documented sources, we'll just have variables for the parts we actually care about that we know are present.

```
⟨ PNRd: local data 80 ⟩ ≡
  struct {
    Word version;
    Word txtRecords;
  } rec0;
  BYTE *p;
```

This code is used in section 79.

81. We store the record count based on the version of the file.

```
⟨ PNRd: get record 0 81 ⟩ ≡
  p = read_pdb_record(0);
  store_Word(p, rec0.version);
  ⟨ PNRd: check for valid version 82 ⟩
  if (rec0.version ≡ #0A) p += 10;
  store_Word(p, rec0.txtRecords);
  free(p); p = NULL;
```

This code is used in section 79.

82.

```
⟨ PNRd: check for valid version 82 ⟩ ≡
  if (rec0.version > #FF) fatal("encrypted␣PNRd␣file");
  if (rec0.version ≠ #02 ∧ rec0.version ≠ #0A) fatal("unrecognized␣PNRd␣format");
```

This code is used in section 81.

83. We may also want to show the data we collected about the file.

```
⟨ PNRd: dump record 0? 83 ⟩ ≡
  if (flags & FLAG_RECZERO) {
    ⟨ PNRd: show record 0 84 ⟩
    EXIT_IF_LAST_FLAG(FLAG_RECZERO);
  }
```

This code is used in section 79.

84. We have captured very little data from record 0, but let's show what we have:

```
⟨ PNRd: show record 0 84 ⟩ ≡
  printf("Record␣0:\n");
  printf("␣␣version␣0x%02x,␣text␣records␣%d\n", rec0.version, rec0.txtRecords);
```

See also section 85.

This code is used in section 83.

85. At the same time we show record 0, we can show the metadata for the book in the second to last record, *hdr.numRecords* − 3. This data is present in books generated by the *DropBook* utility, where the last record consists of the string `MeTaInFo`. The metadata consists of the title, the author, the copyright, the publisher, and the ISBN number.

```
#define PNRD_METATAG "MeTaInFo"
⟨ PNRd: show record 0 84 ⟩ +≡
  if (size_pdb_record(hdr.numRecords − 1) ≡ strlen(PNRD_METATAG) + 1) {
    char *pt = read_pdb_record(hdr.numRecords − 1);
    if (strcmp(pt, PNRD_METATAG) ≡ 0) {
      char *p = read_pdb_record(hdr.numRecords − 3);
      char *ps = p;
      int i;
      printf("␣␣␣title:␣␣␣␣␣<%s>\n", p);
      p += strlen(p) + 1;
      printf("␣␣␣author:␣␣␣␣<%s>\n", p);
      p += strlen(p) + 1;
      printf("␣␣␣copyright:␣<%s>\n", p);
      p += strlen(p) + 1;
      printf("␣␣␣publisher:␣<%s>\n", p);
      p += strlen(p) + 1;
      printf("␣␣␣ISBN:␣␣␣␣␣␣<%s>\n", p);
      free(ps);
    }
    free(pt);
  }
```

86. If we don't want to dump the contents of record 0, we proceed to dumping the body text and supplemental records a record at a time. Notice that if we don't want the supplemental data, we skip that step.

```
⟨ PNRd: process data 86 ⟩ ≡
  int rec;
  ⟨ PNRd: process the text records 87 ⟩
  if (o_name ≠ NULL) {
    ⟨ PNRd: process the supplemental records 90 ⟩
  }
```

This code is used in section 79.

87. We walk through the text records, uncompressing and dumping each one. The *gzip* uncompression is wrapped into a separate routine for convenience. The routine allocates and returns a buffer, which the caller must free.

```
⟨PNRd: process the text records 87⟩ ≡
  for (rec = 1; rec < rec0.txtRecords; rec++) {
    BYTE *buf, *ubuf;

    buf = read_pdb_record(rec);
    ubuf = PNRd_uncompress(buf, size_pdb_record(rec));
    if (¬(flags & FLAG_SUPONLY)) fprintf(ofp, "%s", ubuf);
    free(buf);
    free(ubuf);
  }
```

This code is used in section 86.

88. Here's the routine to inflate *gzip*-type PNRd data. This is more-or-less the same process we use to uncompress the text from the zTXT format. Unfortunately, it appears that maximum uncompressed buffer size is not stored in record 0, so we make a guess, and are prepared to increase it iteratively until we have enough room to inflate the current record.

```
⟨functions 21⟩ +≡
  BYTE *PNRd_uncompress(BYTE *inbuf, size_t insize)
  {
    size_t outsize = 8 * 1024;
    BYTE *outbuf = malloc(outsize + 1);
    z_stream zs;
    int status;
    zs.zalloc = Z_NULL;
    zs.zfree = Z_NULL;
    zs.opaque = Z_NULL;
    if (inflateInit2(&zs, MAXWBITS) ≠ Z_OK) fatal("decompression␣init␣failed");
    zs.next_in = inbuf;
    zs.avail_in = insize;
    zs.next_out = outbuf;
    zs.avail_out = outsize;
    do {
      status = inflate(&zs, Z_SYNC_FLUSH);
      if (zs.msg ≠ NULL) fatal("zlib␣error:␣%s", zs.msg);
      if ((status ≠ Z_STREAM_END) ∧ (status ≠ Z_OK))
        fatal("decompression␣failed:␣error␣from␣inflate");
      if (zs.avail_in > 0) {
        size_t l = zs.next_out − outbuf;
        size_t delta = outsize;
        outsize *= 2;
        outbuf = realloc(outbuf, outsize + 1);
        zs.next_out = outbuf + l;
        zs.avail_out += (outsize/2);
      }
    } while (status ≠ Z_STREAM_END);
    *(zs.next_out) = '\0';
    inflateEnd(&zs);
    return outbuf;
  }
```

89.
```
⟨prototypes 23⟩ +≡
  BYTE *PNRd_uncompress(BYTE *, size_t);
```

90. In `PNRd` files, we will sometimes have leftover records — the difference between *rec0.txtRecords* in the `PNRd` header, and *hdr.numRecords* in the global `PDB` header — which will typically contain images and supplementary data such as bookmarks and annotations. Here is where we figure out what to do about them.

It appears that all images in `PNRd` files are `PNG` format. The tag of `PNG␣` is provided as the first four bytes of the record.

If we don't recognize the record as an image, we put out the first couple of characters of the record.

[[[We should be able, through experimentation and before-and-after comparisons, to figure out how the bookmarks and annotations are actually stored.]]]

```
#define PNRd_PNG "PNG␣"

⟨ PNRd: process the supplemental records 90 ⟩ ≡
  for (rec = rec0.txtRecords; rec < hdr.numRecords; rec++) {
    BYTE *buf = read_pdb_record(rec);
    if (strncmp(buf, PNRd_PNG, strlen(PNRd_PNG)) ≡ 0) {
      ⟨ PNRd: save an image 91 ⟩
    }
    else {
      char tag[SCRATCH_BUF_SIZE], *p;
      int i;
      memset(tag, 0, SCRATCH_BUF_SIZE);
      for (p = tag, i = 0; i < 4; i++) {
        snprintf(tag + strlen(tag), SCRATCH_BUF_SIZE − strlen(tag),
            (buf[i] > '␣' ∧ buf[i] < #7F) ? "%c" : "\\x%02x", buf[i]);
      }
      error("didn't␣recognize␣supplemental␣record␣%d:␣%s", rec, tag);
    }
    free(buf);
  }
```

This code is used in section 86.

91. An image record has the file name imbedded in it. We assume the file name is `NUL` terminated. Once we've got the file name, we can open the file for output using *next_ofile*( ) which ensures the file ends up in the right directory.

```
⟨PNRd: save an image 91⟩ ≡
  Word bufsz = size_pdb_record(rec);
  char *filename = buf + strlen(PNRd_PNG);
  FILE *imagef;
  BYTE *imagep;
  if (strlen(filename) > bufsz ∨ strlen(filename) > FILENAME_MAX)
    fatal("badly␣formed␣filename␣in␣PNG␣block");
  imagef = next_ofile(filename, "");
```

See also section 92.

This code is used in section 90.

92. If the image data is bodily a `PNG` file, the data will start with the PNG file signature, so we will search for that string and declare it the beginning of the image data. It appears that the structure of the record containing the image is to have four bytes of record tag, `PNG␣`, the file name padded to 54 bytes, and two **Word**s containing the width and height. This implies that the included PNG image always starts at the sixth-third byte of the record, but explicitly searching is safer.

The decription of the PNG format is available at

`http://www.libpng.org/pub/png/`,

and in particular, the explanation of the file signature from the specification's rationale is instructive:

> This signature both identifies the file as a PNG file and provides for immediate detection of common file-transfer problems. The first two bytes distinguish PNG files on systems that expect the first two bytes to identify the file type uniquely. The first byte is chosen as a non-ASCII value to reduce the probability that a text file may be misrecognized as a PNG file; also, it catches bad file transfers that clear bit 7. Bytes two through four name the format. The CR-LF sequence catches bad file transfers that alter newline sequences. The control-Z character stops file display under MS-DOS. The final line feed checks for the inverse of the CR-LF translation problem.

We postulate a routine *memstr*( ) which is analogous to POSIX *strstr*( ), but which doesn't stop at `NUL` characters in the string being searched.

```
#define PNG_SIG "\x89PNG\r\n\x1A\n"
⟨PNRd: save an image 91⟩ +≡
  imagep = (BYTE *) memstr(buf, PNG_SIG, bufsz);
  if (imagep ≡ NULL ∨ imagep ≥ (buf + bufsz))
    fatal("bad␣PNG␣file␣in␣record␣%d", rec);
  fwrite(imagep, sizeof(BYTE), bufsz − (imagep − buf), imagef);
  fclose(imagef);
```

93. Utility routines.
We have used a number of globally-useful utility routines. Let's finally get around to defining them.

94. We start with the big-endian read routines.

```
⟨ functions 21 ⟩ +≡
  Word read_Word(void)
  {
    Word n = 0;
    BYTE p[2];
    ck_read(p, 2);
    n = (((BYTE)(p[0])) & #FF) ≪ 8;
    n |= ((BYTE)(p[1]) & #FF);
    return n;
  }
  DWord read_DWord(void)
  {
    long n = 0L;
    BYTE p[4];
    ck_read(p, 4);
    n = (((BYTE)(p[0])) & #FF) ≪ 24;
    n |= (((BYTE)(p[1])) & #FF) ≪ 16;
    n |= (((BYTE)(p[2])) & #FF) ≪ 8;
    n |= (((BYTE)(p[3])) & #FF);
    return n;
  }
```

95. We'll also define big-endian storage macros and two string-copiers. Notice that second string-copier assumes the target is at least one character longer than the copy and places a terminating NUL in the target.

```
⟨ global macros 95 ⟩ ≡
#define store_DWord(p, n)
  {
    n = (((BYTE)(p[0])) & #FF) ≪ 24;
    n |= (((BYTE)(p[1])) & #FF) ≪ 16;
    n |= (((BYTE)(p[2])) & #FF) ≪ 8;
    n |= (((BYTE)(p[3])) & #FF);
    p += 4;
  }
#define store_Word(p, n)
  {
    n = (((BYTE)(p[0])) & #FF) ≪ 8;
    n |= ((BYTE)(p[1]) & #FF);
    p += 2;
  }
#define store_String(p, s, n)
  {
    memcpy(s, p, n);
    p += n;
  }
#define store_ZString(p, s, n)
  {
    memcpy(s, p, n);
    *(s + n) = 0;
    p += n;
  }
```

See also section 96.

This code is used in section 3.

96. Two additional utility macros, which are nrmally defined in POSIX headers, but may not be defined in all environments.

```
⟨ global macros 95 ⟩ +≡
#ifndef MAX
#define MAX(a, b)  (((a) > (b)) ? (a) : (b))
#endif
#ifndef MIN
#define MIN(a, b)  (((a) < (b)) ? (a) : (b))
#endif
```

97. We also have a generic error-checking read routine.

```
⟨ functions 21 ⟩ +≡
  void ck_read(void *p, size_t n)
  {
    if (fread(p, sizeof(char), n, ifp) < n) {
      if (feof(ifp))
        fatal("early␣EOF");
      else
        fatal("bad␣read");
    }
  }
```

98. And error reporting.

```
  format error fatal
⟨ functions 21 ⟩ +≡
  void error(const char *fmt, ...)
  {
    va_list ap;
    fflush(stdout);
    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, "\n");
    va_end(ap);
  }
  void fatal(const char *fmt, ...)
  {
    va_list ap;
    fflush(stdout);
    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, "\n");
    va_end(ap);
    exit(1);
  }
```

99. We also need an allocator.

```
⟨ functions 21 ⟩ +≡
  void *ck_malloc(size_t n)
  {
    void *r;
    r = malloc(n);
    if (r ≡ NULL)
          fatal("bad␣allocation");
    return r;
  }
```

100. We need a utility to open an output file name, based on the global base in *o_name*. We will need files for both the base text, but may also need files for supplemental data like pictures and bookmarks. This routine chooses the name of the file based on information supplied and opens it; it is the caller's responsibility to close it. If we call this with a name and extension we prepend the directory name; if we only supply an extension, we concatenate the directory and a sequential file number.

Our strategy for automatically generating an output file name is pretty straight-forward. If the global *o_name* provided by the `-o` flag is a directory, we choose names within the directory sequentially from `00000`. If *o_name* is a file name, we use it as the base for our output name and append a sequence number.

(This is probably too complicated by half.)

```
⟨ functions 21 ⟩ +≡
  FILE *next_ofile(const char *name, const char *ext)
  {
    static int first_call = 1;
    static int dir = 0;
    static int sequence = 0;
    struct stat sb;
    char fullname[FILENAME_MAX];
    FILE *f;
    if (first_call) {
      ⟨ set up next_ofile 101 ⟩
      first_call = 0;
    }
    if (strlen(name)) snprintf(fullname, FILENAME_MAX, dir ? "%s/%s%s" : "%s%s%s",
          o_name ? o_name : "", name, ext);
    else snprintf(fullname, FILENAME_MAX, dir ? "%s/%05d%s" : "%s%05d%s",
          o_name ? o_name : "", sequence++, ext);
    if ((f = fopen(fullname, "wb")) ≡ NULL ) {
      perror(fullname);
      exit(1);
    }
    return f;
  }
```

101. ⟨ set up *next_ofile* 101 ⟩ ≡

```
if (o_name ∧ o_name[strlen(o_name) − 1] ≡ '/') {
   dir++;
   o_name[strlen(o_name) − 1] = 0;
}
if (stat(o_name, &sb) < 0) {
   if (dir ∧ mkdir(o_name, °777) < 0)
      fatal("can't␣create␣output␣directory␣%s", o_name);
}
else {
   if (sb.st_mode & S_IFDIR)
      dir++;
   else
      error("possible␣output␣file␣conflict␣%s", o_name);
}
```

This code is used in section 100.

102. We invented a routine *memstr*( ) while finding PNG images in `PNRd` files. Here it is:

⟨ functions 21 ⟩ +≡

```
void *memstr(char *big, char *little, size_t len)
{
   int l = strlen(little);
   char *e = big + len − 1 − l;
   char *p = big;

   while (p < e) {
      p = memchr(p, *little, len);
      if (p ≡ NULL) return NULL;
      if (memcmp(p, little, l) ≡ 0) return p;
      p++;
   }
   return NULL;
}
```

103.

⟨ prototypes 23 ⟩ +≡
Word *read_Word*(**void**);
DWord *read_DWord*(**void**);
void *ck_read*(**void** ∗, **size_t**);
void *fatal*(**const char** ∗, . . .);
void *error*(**const char** ∗, . . .);
void ∗*ck_malloc*(**size_t**);
FILE ∗*next_ofile*(**const char** ∗, **const char** ∗);
void ∗*memstr*(**char** ∗, **char** ∗, **size_t**);

104.

105. Index.
A number of sections, namely, 1, 2, 6, 25, 32, 43, 67, 71, 90, still contain working notes, which can be recognized because they are set in *slanted sans serif type*.

Just to prevent confusion, this index lists the *section* numbers of the references, not the page numbers.

annotationRecord: 55, 56, 58, 63, 64, 67.
ap: 98.
appInfoId: 14, 15, 17, 25.
argc: 3, 6.
argv: 3, 6, 10.
ATTR_BACKUP: 15.
ATTR_DIRTY: 15.
ATTR_LAUNCHABLE: 15.
ATTR_OKNEWER: 15.
ATTR_OPEN: 15.
ATTR_READONLY: 15.
ATTR_RESET: 15.
ATTR_RESOURCE: 15.
attributes: 14, 15, 17, 25, 26, 27, 28, 32.
avail_in: 62, 88.
avail_out: 62, 88.
b: 16.
big: 102.
book_formats: 38.
bookmarkRecord: 55, 56, 58, 63, 64, 65.
buf: 21, 33, 47, 48, 49, 77, 87, 90, 91, 92.
bufsz: 91, 92.
BYTE: 13, 16, 26, 33, 35, 36, 42, 47, 48, 50, 53, 55, 60, 64, 66, 67, 71, 77, 80, 87, 88, 89, 90, 91, 92, 94, 95.
ck_malloc: 28, 61, 62, 99, 103.
ck_read: 17, 28, 33, 35, 94, 97, 103.
crc32: 55, 56, 58.
creationDate: 14, 17, 25.
creator: 14, 15, 17, 25, 37, 40.
current_book_format: 38, 40, 49.
d: 50.
dae: 8.
dap: 8.
DATE_OFFSET: 21.
decode: 37, 40.
DEL: 50.
delta: 88.
dir: 100, 101.
disp: 37, 40.
dispatch: 37.
DISPATCH_SIZE: 37, 40.
distance: 50.
doc_size: 42, 43, 45, 71, 72, 74.
DWord: 13, 14, 21, 23, 26, 34, 42, 55, 60, 64, 66, 71, 94, 103.
e: 48, 102.
encrypted: 71, 72, 74, 76.
error: 98, 101, 103.
exit: 3, 6, 10, 24, 98, 100.
EXIT_IF_LAST_FLAG: 24, 31, 44, 57, 73, 83.
ext: 100.
f: 100.
fatal: 6, 18, 40, 43, 46, 49, 56, 61, 62, 66, 72, 75, 76, 78, 82, 88, 91, 92, 97, 98, 99, 101, 103.
fclose: 65, 67, 77, 92.
feof: 97.
Feynman, Richard Phillips: 1.
fflush: 62, 98.
filename: 91.
FILENAME_MAX: 91, 100.
first_call: 100.
FLAG_HEADERS: 6, 7, 24.
FLAG_RECHDRS: 6, 7, 31.
FLAG_RECZERO: 6, 7, 44, 57, 73, 83.
FLAG_SUPONLY: 6, 7, 47, 48, 62, 87.
FLAG_SUPTEXT: 6, 7, 63, 65, 66, 67.
flags: 6, 7, 24, 31, 44, 47, 48, 55, 56, 57, 58, 62, 63, 65, 66, 67, 73, 83, 87.
fmt: 98.
fmt_ERROR: 38.
fmt_MOBI: 38, 49.
fmt_PNRd: 38.
fmt_TEXt: 38.

fmt_zTXT: 38.
fopen: 10, 100.
fprintf: 47, 48, 51, 62, 66, 67, 87, 98.
fread: 97.
free: 43, 47, 48, 56, 65, 67, 68, 72, 77, 81, 85, 87, 90.
fseek: 33, 35.
fullname: 100.
fwrite: 77, 92.
getopt: 6.
hdr: 14, 15, 17, 25, 26, 28, 32, 33, 34, 35, 40, 77, 85, 90.
i: 5, 42, 55, 71, 85, 90.
ide: 8.
idp: 8.
ifp: 10, 11, 33, 35, 97.
imagef: 91, 92.
imagep: 91, 92.
inbuf: 50, 60, 61, 62, 68, 88.
index_record: 64, 65, 66, 67.
inend: 50, 51, 52.
infilename: 10, 11, 18.
inflate: 62, 88.
inflateEnd: 62, 88.
inflateInit2: 62, 88.
insize: 50, 60, 61, 62, 88.
l: 50, 88, 102.
lastBackupDate: 14, 17, 25.
len: 102.
`LEN_HDR`: 14, 16, 17.
`LEN_TTL`: 14, 17.
lend: 8.
length: 50.
leni: 8.
little: 102.
localtime: 21.
main: 3.
malloc: 33, 48, 88, 99.
`MAX`: 96.
`MAXWBITS`: 60, 62, 88.
memchr: 102.
memcmp: 40, 102.
memcpy: 95.
memset: 21, 33, 50, 90.
memstr: 92, 102, 103.
`MIN`: 66, 96.
mkdir: 101.
MOBI_decode: 37, 39, 43, 70.
`MOBI_RECO_SIZE`: 71, 72.
modificationDate: 14, 17, 25.
modificationNumber: 14, 17, 25.
msg: 62, 88.
n: 42, 48, 50, 71, 77, 94, 97, 99.
name: 100.
next_in: 62, 88.
next_ofile: 6, 12, 65, 67, 77, 91, 100, 103.
next_out: 62, 88.
`NUL`: 15, 33, 47, 50, 52, 95.
num_recs: 42, 43, 45, 47, 48, 71, 72, 74, 77.
numAnnotations: 55, 56, 58, 63, 67.
numBookmarks: 55, 56, 58, 63, 65.
numRecords: 14, 17, 25, 26, 28, 32, 33, 34, 35, 55, 56, 58, 61, 77, 85, 90.
o: 50.
o_name: 6, 7, 12, 63, 75, 86, 100, 101.
obuf: 35.
offset: 26, 28, 32, 33, 34, 35, 64, 66.
ofp: 7, 12, 47, 48, 62, 65, 67, 77, 87.
opaque: 62, 88.
outbuf: 50, 60, 62, 66, 68, 88.
outend: 50.
outsize: 50, 88.
p: 16, 42, 50, 55, 71, 80, 85, 90, 94, 97, 102.
PDBrec_header: 26, 28, 30.
PDBsize: 14, 18, 25, 34, 61.
perror: 10, 100.
`PNG_SIG`: 92.
PNRd_decode: 37, 39, 79.
`PNRD_METATAG`: 85.
PNRd_PNG: 90, 91.
PNRd_uncompress: 87, 88, 89.
posix_date: 21.
pp: 50, 66.
ppe: 66.
printf: 8, 9, 24, 25, 32, 45, 48, 58, 74, 84, 85.

ps: 85.
pt: 85.
r: 99.
read_DWord: 28, 94, 103.
read_pdb_noalloc: 35, 36, 61.
read_pdb_record: 33, 36, 43, 47, 48, 56, 65, 67, 72, 77, 81, 85, 87, 90.
read_Word: 94, 103.
realloc: 88.
rec: 86, 87, 90, 91, 92.
REC_BUSY: 27.
REC_CATEGORY: 27.
REC_DELETE: 27.
REC_DIRTY: 27.
rec_hdrs: 28, 30, 32, 33, 34, 35.
REC_SECRET: 27.
rec_size: 42, 43, 45, 48, 49, 71, 72, 74.
recnum: 33, 34, 35.
recordOffset: 14, 17, 25.
recordSize: 55, 56, 58.
recpos: 33, 35.
recsize: 33, 34, 35.
rec0: 42, 43, 45, 46, 47, 48, 49, 55, 56, 58, 61, 62, 63, 64, 65, 66, 67, 71, 72, 74, 75, 76, 77, 80, 81, 82, 84, 87, 90.
reserved: 42, 43, 55, 71, 72.
reserved2: 42, 71.
revisionDate: 8.
revisionId: 8.
r0: 42, 43, 55, 56, 71, 72.
S_IFDIR: 101.
sb: 18, 19, 100, 101.
SCRATCH_BUF_SIZE: 21, 90.
SEEK_SET: 33, 35.
sequence: 100.
show_time: 21, 23, 25.
size: 55, 56, 58, 62, 66.
size_pdb_record: 32, 33, 34, 35, 36, 43, 48, 56, 61, 72, 77, 85, 87, 91.
snprintf: 90, 100.
sortInfoId: 14, 15, 17, 25.
st_mode: 101.
st_size: 18.
stat: 18, 19, 100, 101.
status: 88.
stderr: 51, 98.
stdout: 6, 12, 98.
store_DWord: 17, 43, 56, 66, 72, 95.
store_String: 17, 95.
store_Word: 17, 43, 56, 72, 81, 95.
store_ZString: 17, 66, 95.
strcat: 21.
strchr: 8.
strcmp: 85.
strftime: 21.
strlen: 85, 90, 91, 100, 101, 102.
strncmp: 90.
strstr: 92.
sup: 64, 65, 66, 67.
t_palm: 21.
tag: 64, 66, 90.
temp: 66.
text: 67.
TEXt_decode: 37, 39, 41.
TEXt_decompress: 49, 50, 53.
TEXT_RECO_SIZE: 42, 43.
title: 14, 17, 25, 64, 66, 67.
tt: 21.
txtRecords: 80, 81, 84, 87, 90.
type: 14, 15, 17, 25, 37, 40.
ubuf: 48, 49, 87.
uniqueID: 26, 28, 32.
uniqueIdSeed: 14, 17, 25.
usage: 5, 6.
va_end: 98.
va_start: 98.
version: 14, 17, 25, 42, 43, 45, 46, 55, 56, 58, 71, 72, 74, 75, 80, 81, 82, 84.
vfprintf: 98.
Word: 13, 14, 33, 34, 35, 36, 42, 50, 55, 71, 80, 91, 92, 94, 103.
Z_NULL: 62, 88.
Z_OK: 62, 88.
z_stream: 60, 88.
Z_STREAM_END: 62, 88.
Z_SYNC_FLUSH: 62, 88.

zalloc: 62, 88.
zfree: 62, 88.
zs: 60, 62, 88.
zTXT_decode: 37, 39, 54.
`ZTXT_MARK_LEN`: 64, 66.
`ZTXT_NONUNIFORM`: 55, 58.
`ZTXT_RANDOMACCESS`: 55, 58.
`ZTXT_RECO_SIZE`: 55, 56.
`ZTXT_SNIP`: 64, 66.

⟨ MOBI: check for encryption 76 ⟩ Used in section 75.
⟨ MOBI: dump record 0? 73 ⟩ Used in section 70.
⟨ MOBI: get record 0 72 ⟩ Used in section 70.
⟨ MOBI: local data 71 ⟩ Used in section 70.
⟨ MOBI: process Huffman compressed data 78 ⟩ Used in section 75.
⟨ MOBI: process data 75 ⟩ Used in section 70.
⟨ MOBI: save ancillary data 77 ⟩ Used in section 75.
⟨ MOBI: show record 0 74 ⟩ Used in section 73.
⟨ PNRd: check for valid version 82 ⟩ Used in section 81.
⟨ PNRd: dump record 0? 83 ⟩ Used in section 79.
⟨ PNRd: get record 0 81 ⟩ Used in section 79.
⟨ PNRd: local data 80 ⟩ Used in section 79.
⟨ PNRd: process data 86 ⟩ Used in section 79.
⟨ PNRd: process the supplemental records 90 ⟩ Used in section 86.
⟨ PNRd: process the text records 87 ⟩ Used in section 86.
⟨ PNRd: save an image 91, 92 ⟩ Used in section 90.
⟨ PNRd: show record 0 84, 85 ⟩ Used in section 83.
⟨ TEXt: decode special case for trailing `NUL` 52 ⟩ Used in section 50.
⟨ TEXt: decompress 49 ⟩ Used in section 48.
⟨ TEXt: dump record 0? 44 ⟩ Used in section 41.
⟨ TEXt: get record 0 43 ⟩ Used in section 41.
⟨ TEXt: local data 42 ⟩ Used in section 41.
⟨ TEXt: process compressed 48 ⟩ Used in sections 46 and 75.
⟨ TEXt: process data 46 ⟩ Used in section 41.
⟨ TEXt: process uncompressed 47 ⟩ Used in sections 46 and 75.
⟨ TEXt: show record 0 45 ⟩ Used in section 44.
⟨ data types 13 ⟩ Used in section 3.
⟨ dump file header 25 ⟩ Used in section 24.
⟨ dump record headers 32 ⟩ Used in section 31.
⟨ functions 21, 33, 34, 35, 41, 50, 54, 70, 79, 88, 94, 97, 98, 99, 100, 102 ⟩ Used in section 3.
⟨ global data 7, 11, 14, 26, 30, 37, 38 ⟩ Used in section 3.
⟨ global macros 95, 96 ⟩ Used in section 3.
⟨ include files 4, 20, 22, 29, 59 ⟩ Used in section 3.
⟨ leftovers 51 ⟩ Used in section 50.
⟨ local data in main 5, 16, 19 ⟩ Used in section 3.
⟨ main program 6, 10, 12, 17, 18, 24, 28, 31, 40 ⟩ Used in section 3.
⟨ print version and copyright 8, 9 ⟩ Used in section 6.
⟨ prototypes 23, 36, 39, 53, 89, 103 ⟩ Used in section 3.
⟨ set up *next_ofile* 101 ⟩ Used in section 100.
⟨ zTXT: dump record 0? 57 ⟩ Used in section 54.
⟨ zTXT: get record 0 56 ⟩ Used in section 54.
⟨ zTXT: local data 55, 60, 64 ⟩ Used in section 54.
⟨ zTXT: process data 61, 62, 63, 68 ⟩ Used in section 54.

⟨ zTXT: show a tag 66 ⟩ Used in sections 65 and 67.
⟨ zTXT: show annotations 67 ⟩ Used in section 63.
⟨ zTXT: show bookmarks 65 ⟩ Used in section 63.
⟨ zTXT: show record 0 58 ⟩ Used in section 57.