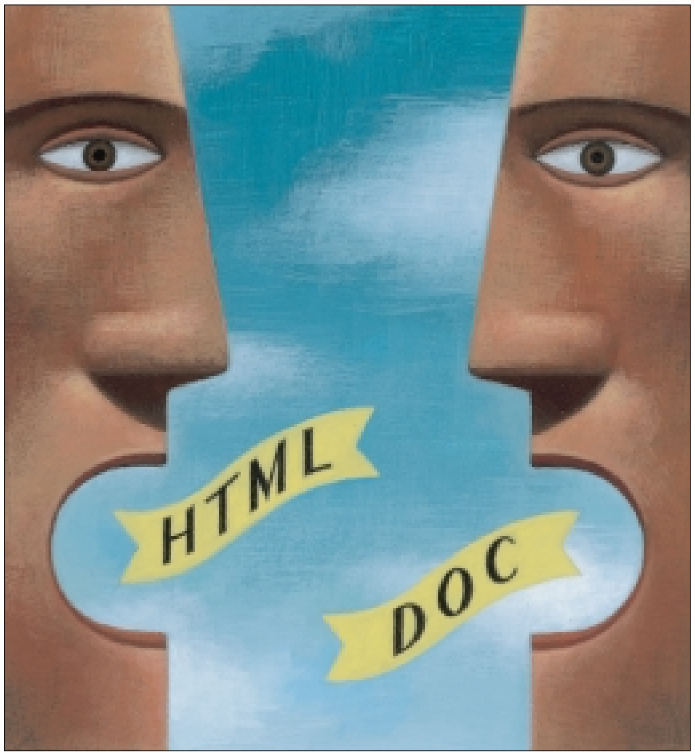
by Jeffreys Copeland and Haemer



SCOTT ROBERTS

Whenever a person is called upon to make a speech, the first question that enters his mind is "What shall I talk about?"
– Gerald R. Ford

[Mr. Protocol] continues to believe that words on paper are so antique by the time he sees them that they couldn't possibly be of any use. Not only are the classics a closed book to him, books are a closed book to him.
– Michael O'Brien

Reader Filters

Welcome to the real beginning of the new millennium. We are glad you weren't taken in by any of that frivolity associated with the year rolling over to an even thousand. What better way to celebrate the new century than by discussing an interesting calendar problem? We even advertised such a column last time. Unfortunately, when we sat down to write it, we realized that in nearly 70 work columns, we've done eight on calendar calculations. We simply couldn't think of anything new to say. Certainly, we could have built a Perl module to handle Easter or phases of the moon, but that's been done–check out the modules available at `http://www.cpan.org/`. We could have done a survey of date and time Perl modules, but you can probably do that just as well yourself. Instead, we'll continue our thought from last month, and talk about reading some more.

As we mentioned last month, one of the problems with electronic book readers is that there are many different formats. When we wrote a text reader two years ago, we based it on simple ASCII files, the *lingua franca* of textual interchange. (See `http://swexpert.com/C9/SE.C9.DEC.98.pdf` and `http://swexpert.com/C9/SE.C9.JAN.99.pdf`.) The profusion of new devices means that there is a wealth of new formats to contend with. Many of them are based on HTML or XML, but still require some translation into the device's internal format.

This month, we'll show you a converter from text to Palm's `DOC` format.

Why this format? Because it's simple, it's common (there are about seven million Palm devices out there), and it shows techniques for doing a conversion for your own device. Oh, yes: Copeland's wife recently got a Palm, and wanted just such a converter.

Even though there are bits of software out there that do this conversion, we've chosen to write our own. We do this because we have a different notion of what constitutes a paragraph than most of the other software, and because we can express the important concepts a little more clearly in Perl. However, we will point out that the Linux software choice for text to Palm `DOC` format is `txt2pdbdoc`, written by Paul Lucas, `http://www.best.com/~pjl/software.html`. There's a corresponding Windows program, `MakeDocW`, by Mark Pierce, `http://www.pierce.de/`. By checking the output of both programs, and comparing them with the documentation, we have a good idea of what the file format looks like.

Format

The first thing to realize is most data files downloaded to the Palm are in the form of a Palm database file: a `PDB` file. The `DOC` format, developed to circumvent the Palm's reluctance to access text blocks larger than 4,096 bytes, is encapsulated inside a `PDB`.

What does a `PDB` file look like? It's documented in the Palm File Format Specification, on the Palm Web site at http://www.palm.com. In addition, there's documentation for the DOC format at http://pyrite.linuxave.net/doc/, which also has pointers to the details of the DOC compression algorithm described by its designer, Pat Beirne. Of course, having Paul Lucas' code to read helped, too. Let's dig into our code, and we'll explain the formats as we go along.

We start with our usual prolog: a shebang line, an RCS id string, and appropriate use statements.

```
#! /usr/local/bin/perl -w
# $ID: txt2palm,v 1.7 2000/11/08 ...

use strict;
use Getopt::Std;
```

Next, we need to set up some important constants.

```
## useful constants
   # maximum record size in PDB file
my $RECMAX = 4096;
   # size of PDB header
my $PDBHDRSZ = 0x4E;
   # size of record header
my $RECHDRSZ = 0x08;
```

The record size in a PDB file has to be less than 4 K, the maximum amount of memory you're guaranteed to find free in the Palm's dynamic memory. We also supply the sizes of header blocks, which we'll need to calculate file offsets.

```
## command line flags
use vars qw($opt_c);
getopts("c")  ||
  die usage "$0 [-c] title...\n";
die "compression not supported yet\n"
  if( $opt_c );
```

We parse the single command-line flag, and then promptly yell if it was used. We aren't going to show code for the compression algorithm here due to lack of space and time.

```
## the document title is taken from
## the command line:  if missing, we
##  just use the current date.
my $fulltitle = join(" ", @ARGV);
```

We grab the rest of the command-line arguments and use them for the document title. This is the name under which the document gets remembered by the Palm OS and the reader software.

It's traditional, in this sort of program, to read the text to be converted a little bit at a time, do the conversion and then write the converted file a little at a time, too. However, since we're using Perl, we don't bother with that charade. Instead, we just read the whole file at a gulp and process it with a burp. Notice that we're reading the file from standard input, so we have a command line of the form:

```
txt2palm work columns <work.txt >work.pdb
```

Later we'll disgorge the PDB file to standard output.

```
## begin by reading the whole text file
undef $/;
my $text = <STDIN>;
my $filesize = length($text);
```

Since we have the file as a lump, we can make some transformations on it as a whole that we couldn't if we were processing it a bit at a time. One of the failures of existing programs to do text to DOC conversion–both for UNIX and Windows–is that they believe a paragraph in the DOC file is generated from a single line in the text file, and that the only allowable paragraph separator is a blank line in both the text file and the DOC file. Not so here: We accept either a blank line or an indented line as the start of a paragraph, and then turn each paragraph into the single line that DOC expects.

```
## We have a number of steps to do to
## process the text into a useful lump.
   # First, we indent each paragraph.
$text =~ s@\n\n+@\n    @g;
   # Next, we elide the newlines inside
   # a paragraph
$text =~ s@\n(?=\S)@ @g;
   # (This has the happy side-effect that
   # when we have pars marked by just an
   # indented first line, we don't lose
   # the paragraph breaks.)
```

Notice that we're using Perl's zero-width look-ahead to remove the newlines. This is more efficient than the equivalent s@\n(\S)@ $1@.

Our next step is to make an array of chunks of 4 K, each of which will become a record in our PDB file.

```
## Now that we have the text in a lump,
## we want to break it up into chunks of
## the maximum size for Palm PDB records.
my @records;
for( my $pos = 0;
   $pos < length($text);
   $pos += $RECMAX) {
 my $rec = substr($text,$pos,$RECMAX);
 $rec = compress($rec) if($opt_c);
 push(@records, $rec);
}
```

We next set up the data to be stuffed into the PDB header block. These include the document title, which is a maximum of 31 bytes long, and the current time. Notice that the Palm uses an epoch of January 1, 1904, instead of our familiar UNIX epoch of January 1, 1970. "But," you say, "the UNIX time_t type only covers 68 years. How does the Palm manage to still have bits left in its time values?" Because time_t is a signed arithmetic quantity–usually a signed 32-bit integer–and the Palm uses an unsigned 32-bit double word, which covers twice as many years.

```
## prepare the PDB header
my $header;
my $title = length($fulltitle) > 31 ?
  substr($fulltitle,0,31) :
  $fulltitle;
my $timestamp = time() + 2082844800;
   # Palm time epoch is 1/1/1904;
   # offset to Unix epoch 1/1/1970
```

Header blocks

The PDB header block and embedded DOC headers are the closest part of this process to magic. The PDB header begins with the NUL-padded title string, includes some time stamps, and some known values for which reader programs look. We add the number of records.

```
# we prepare the (mostly empty)
# header for the PDB file
$header = pack("a32nnN6A4A4NN",
   $title, # 32-bytes null padded
   0, # attributes = 0
   0, # version = 0
   $timestamp, # creationDate
   $timestamp, # modificationDate
   0, # lastBackupDate
   0, # modificationNumber
   0, # appInfoID
   0, # sortInfoID
   "TEXt",  # type
   "REAd", # creator
   0,          # uniqueIDSeed
   0  # (offset to) recordList
  );

 # add record count to header block
$header .= pack("n", @records+1);
```

Note that we're using Perl's pack function to assemble these strings of bytes. The unsigned shorts and unsigned longs that make up the header block are packed in "network"–that is, bigendian–order.

We also need to create an array of record headers. Each header includes a pair of unsigned longs, the index, and offset. The index is an arbitrary value that includes some incompletely documented status bits. All the programs use the same starting value and increment it by one for each record. The offset is a little easier: It's the number of bytes from the beginning of the file where the record actually begins, which makes it fairly easy to calculate.

We begin with the header for the special "record zero," which contains important data for the whole document:

```
# now we create a list of record
 # headers, for each record we have
my @rechdrs;
my $index;
my $offset;
 # the zeroth record header contains
 # the offset to the point after the
 # last header block, that is to the
 # body of record zero itself
$index = 0x406f8000;
$offset = $PDBHDRSZ +
 ($RECHDRSZ * (@records+1));
push(@rechdrs,
 pack("N2", $offset, $index));
```

Again, pack is a useful ally. Once we've created the header for record zero, we can create that record itself. The assembly order doesn't really matter, since we're buffering information to be dumped in a flurry later, but this makes logical sense.

```
# also need to create the body of
 # record zero, which contains some
 # global information
my $reczero = pack("nnNnnN",
  $opt_c ? 2 : 1,
    # version:
    #  1 = uncompressed,
    #  2 = compressed
  0, # unused short
  $filesize,      # uncompressed size
  $#records+1, # nr of records
  $RECMAX,  # max record
  0  # current position in doc
  );
$offset += length($reczero);
```

Notice that we need to use $#records+1 for the size of the array rather than simply @records because we need a scaler for pack. Also notice that we are keeping careful track of the offset through the PDB file as we go along.

```
  # record headers after the first
  # contain the offset to the actual
  # data record
foreach my $rec (@records) {
  $index++;
  push(@rechdrs,
    pack("N2", $offset, $index));
  $offset += length($rec);
}
```

Once we've assembled record zero, assembling the headers for the text records is a fairly simple task. We just walk through the list of records we prepared earlier, increment the index value, and accumulate the offset for the placement of the record in the file.

Output and Compression

That's basically it. Our last step is to dump the file to `stdout`.

```
## now drop the whole file in a
## quick series of prints...
  # the header block
print $header;
  # the record headers
print @rechdrs;
  # PDB record 0
print $reczero;
  # the text records
print @records;
```

Well, we're not quite done: We need to provide a compression routine. For purposes of this article, we'll provide a very simple placeholder that just returns the string it's sent.

```
# we don't provide a real compression
# routine yet....

sub compress { $_[0] }
```

How does the compression in a `DOC` file work? Good question. Remember in the discussion that follows the Palm uses ISO Latin-1 as its basic character set. (You've probably got a man page entitled `iso 8859 1` on your system, so you can read along.)

We begin by collapsing spaces into the following character if the character is between @ and tilde. We do this by setting the character's high bit. Thus, a space followed by "J" becomes `0xCA`. This commits the range from `0xC0` through `0xFF`.

This means that many of the high-range ISO Latin-1characters can't be represented as themselves. To solve this problem (remember they're not as common as the ASCII characters) we preceed any of those characters with a byte count. In fact, we can proceed any string of from one to eight characters with a byte count to produce the raw string. In this case, a sequence of `0x02 0xB6 0xC4` would be the uncompressed pair of bytes *paragraph sign, A-umlaut.*

The most complicated feature allows repeating previous sequences in the (uncompressed) file. If we see a repeatable sequence of from three to 10 bytes, no more than 2,047 bytes previously, we subtract three from the length of the sequence, and pack it and the distance into two bytes:

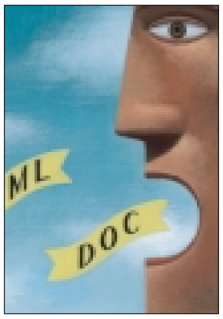
```
0x8000 + (distance<<3) + ((length-3)& 0x07)
```

For example, if "Jeffrey" appeared twice in our text, 500 bytes apart, the second occurrence would be encoded as `0x8000 + 0x1F4 << 3 + 4` or `0x8FA4`, replacing seven bytes with two. This covers lead bytes from `0x80` through `0xBF`.Every other byte–that is, `0x00` plus `0x09` through `0x7F`–represents itself.

Given that information, you should be able to produce some Perl code to replace the short placeholder above. In the software bundle on the Web site, you'll find a sample input file along with compressed and uncompressed output results. (Thanks to loyal reader Mark Jackson at Xerox for reminding us to include sample data with our software.)

Readers

You'll notice that we haven't attempted to write a program for the Palm to act as a reader. There are two reasons for that. One reason is that we don't know how to program the Palm yet. A second is that even if we did, it would be unnecessary since there is a variety of reader programs already available. TealDoc and AportisDoc are the main commercial contenders; Boulder-based NetLibrary provides a reader through its PeanutPress subsidiary; but the GNU entry is CSpotRun by Bill Clagett (`http://www.32768.com/bill/palmos/cspotrun`).



If starting from scratch, we would drop the ugly Palm system font and build our reader around a version of a font like Bell Centennial or Verdana.

On the other hand, if we were starting from scratch, we'd drop the ugly Palm system font and build our reader around a version of a font like Bell Centennial or Verdana, two typefaces designed by Matthew Carter that are highly readable at small sizes or low resolutions. We'd use Douglas Henke's `pbm2pfnt` (see `http://www.insync.net/~henke/pbm2pfnt/`) to build the loadable bitmaps for the reader. Then, because every reader uses a slightly different, incompatible set of extensions, we'd ensure that we had some portable way of signifying font changes in our `DOC` files, for example, using HTML tags such as `<I>...</I>`. We'd also build in on-the-fly hyphenation based on the algorithm Frank Liang developed for his Ph.D. work at Stanford, and line filling based on either Mike Plass' Stanford Ph.D. work or Kurt Shoens' and Liz Allen's version of the BSD `fmt` program. However, those are all nits. In general, we're amazed to discover much of the functionality we want is already squeezed into a device with a screen the size of a Post-It note.

That's all for now. Until next month, happy trails.

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer (`jsh@usenix.org`) *works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.