

# Work

by Jeffreys Copeland and Haemer



JANE MARINSKY

*In general, the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas.*  
– Donald E Knuth, *The TeXbook*

*Outside of a dog, a book is a man's best friend. Inside of a dog, it's too dark to read.*  
– Groucho Marx

## Compression Practicum

As part of our first column this year (“Reader Filters,” in the January issue of this magazine) we wrote a Perl script to convert text into a form readable on the popular Palm handheld computer by applications such as Bill Clagett’s CSpotRun, (<http://32768.com/cspotrun/>).

One of the features we didn’t develop in `txt2palm` was compression. And, since we discussed compression in last month’s column, it seemed a good time to revisit the topic. We tossed off an explanation of the Palm compression scheme, and told you, “Given that information, you should be able to produce some Perl code to replace the short placeholder,” implying that the code for the compression would be simple. And once the January issue hit the streets, we received a note from Gary Sabot, a finance wonk who is getting tired of moving data from his Sun to his PC in order to convert it for his Palm. Gary asked specifically about the compression code.

Unfortunately, we lied to you. It was not a completely deliberate lie, because we hadn’t fully thought the problem through. Even though the Palm compression scheme is fairly simple and Perl provides wonderful regular expression and string processing features, writing the necessary code in Perl would have been a bit difficult. So, in the Knuth mold, this later chapter contains more reliable information than the earlier one.

### Palm DOC Compression

Typically, the compression scheme for Palm documents results in reducing text to roughly 55% of its original size. Not as good as some of the tools we explored last month, but still pretty good. How does it do that? Well, let’s review. There are four classes of characters in a compressed Palm DOC file.

- Characters that represent themselves, which are just copied from the compressed buffer to the output. This set consists of the ASCII characters from `tab` (0x09) through `DEL` (0x7F), and `NUL` (0x00).

- A single-byte compression. A space followed by a character from 0x40 through 0x7F is compressed into a single character by setting the high bit. Thus two bytes, a space followed by J, are compressed into a single byte of 0xCA.

- Characters that introduce a literal block. The characters from 0x01 through 0x08 introduce a block of that many characters that are copied literally from the compressed buffer. This allows characters that are special codes to appear in the output, which means upper range Latin 1 characters (like `Ë`, which is also encoded as 0xCA) won’t be confused with the space-compressed characters from the previous class.

- Run-length encoding. A run of characters that appears previously in the file can be encoded as two bytes. If we see a repeatable sequence between three and ten bytes, no more than 2047 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)
```

This allows us to store a sequence of up to ten bytes in two. There's a clear tradeoff here: we could make a scheme to store longer sequences but our maximum horizon for finding them would have to be shorter. It's this last class of compressed characters that's a bit difficult to implement in Perl. However, that's just fine, because it allows us a chance to show you how to link a C language routine into a Perl package.

## Perl Packages with C

You might be surprised to realize that we can build Perl packages in C, but it's not quite as complicated as it might seem. In essence we'll be inverting the steps taken when installing a package from the Comprehensive Perl Archive Network (CPAN).

We begin to build our compression package by generating a template with

```
h2xs -A -n PalmComp
```

This creates a directory `PalmComp/` containing a Makefile generator `Makefile.PL`, a module file for the package, `PalmComp.pm`, an extension module, `PalmComp.xs`, and a test template, `test.pl`. We will ignore the test template for the moment, because we'll add some test code directly to the package. Our first step will be to add RCS id strings in the templates, add the name of the routine we want to publicly expose into the exports list in `PalmComp.pm`, and flesh out the POD (Plain Old Documentation) in-line manual page.

```
# $Id: PalmComp.pm,v 1.1 ...
package PalmComp;

use strict;
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK);

require Exporter;
require DynaLoader;
require AutoLoader;

@ISA = qw(Exporter AutoLoader DynaLoader);
# Items to export into callers namespace
# by default.
@EXPORT = qw(compress);
$VERSION = '0.95';

bootstrap PalmComp $VERSION;

1;
__END__

=head1 NAME

PalmComp - Perl extension for
Palm PDB compression

=head1 SYNOPSIS
```

```
use PalmComp;
compress(buffer, length);

=head1 DESCRIPTION

This module compresses a buffer for a Palm PDB
file, as described in our April, 2001 "Work"
column.

=head1 AUTHOR

Jeffrey Copeland C<copeland@alumni.caltech.edu>

Jeffrey S Haemer C<jsh@usenix.org>

=head1 SEE ALSO

perl(1).

=cut
```

Next, we need to flesh out the `PalmComp.xs` template with appropriate C code. We can retain a simple skeleton, for the moment:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* $ID: PalmComp.xs,v 1.2 ... */
#define REAL_MODULE
#include "comp.c"

MODULE = PalmComp      PACKAGE = PalmComp

void
compress( buf, length )
    char *buf;
    int length;
PREINIT:
    int len;
PPCODE:
    len = compress(buf, length);
    EXTEND(SP, 2);
    PUSHs(sv_2mortal(newSVpvn(buf, len)));
    PUSHs(sv_2mortal(newSViv(len)));

The first three include files were provided by h2xs, but we've added the include of comp.c and definition of REAL_MODULE, both of which we'll explain in a little bit. The body of the routine is interpreted by xsubpp, which generates raw C code from the XS file. The simplest possible XS code would be something like

void
hello()
CODE:
```

```
printf("hello, world\n");
```

We've used a slightly more complicated set of tags. `PREINIT` gives us the ability to declare local variables without fear of interference from the code `xsubpp` generates. Because we want to return two values so that the Perl call will be something like

```
($newbuf, $newlen) = compress($buf,$len);
```

and we use the `PPCODE` keyword. This introduces code that won't have return values automatically put on the stack. We put those values there by extending the stack with `EXTEND` and pushing the return values with `PUSH`. The other important calls here are `sv_2mortal`, which instantiates a stack version of a variable that won't be accidentally cleaned up by Perl's garbage collector, and `newSViv` and `newSVpv`, which generate new stack values from an integer and a pointer/length pair, respectively.

You'll notice that we've still avoided talking about the actual compression routine. Our XS routine, called `compress`, calls another routine called `compress`, which is (presumably) included from a file `comp.c`. How can `compress` call `compress`? It's not a recursive call because `xsubpp` converts the "Perl glue" name below the `MODULE` declaration in the XS file using something like C++'s name mangling.

Once we've got the module ready, we can generate a makefile to build and install it with `perl Makefile.PL`.

(Yes, we're covering this quickly and loosely. We recommend you read the *perlXS* and *perlXS* manpages for details.)

## The Real Compression Routine

You've been patient enough with the wrapper. We now get to the contents of the `comp.c` file:

```
static char *id = "$ID: comp.c,v 1.6 ... ";
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int nextmatch(
    unsigned char *, int, int *);
void populate_runs(
    unsigned char *, int *, int);
int memchk(
    unsigned char *, unsigned char *);
```

We begin with our usual debris for a C program, an RCS id line, some include files, and prototypes. Notice that because we're going to be doing a lot of character comparisons, we're using unsigned characters.

```
#define HAVECHARS ((p+1) < (buf+len))
#define COUNTABLE(x) \
    (((x)>=1 && (x)<=8) || ((x)>=0x80))
```

These two macros will prove useful. The first will be true

if we have characters left in the buffer. The second will be true if we are looking at a character which needs to be preceded by a count.

```
int
compress(unsigned char *buf, int len)
{
    unsigned char *new, *p, *np, *t;
    int *runs, rr;
```

The actual compression routine begins fairly routinely, for lack of a better word. We expect the buffer and its length, which we passed from our `txt2palm` program to the Perl glue version of `compress()` in the XS file, and thence here. Our only function return value is the length; we will replace the original buffer with the compressed buffer as part of our return. We also declare a flock of variables, which will be explained in due course. (A narrative disadvantage of C compared to C++ and Perl is that we can only declare variables at the top of a block, and so we have to tell you all of their names before we're ready to explain what they do. It's one reason we occasionally like writing in CWEB.)

The creation of the buffer we'll be compressing into is perhaps a bit tricky. Last month we discussed cases where compression is counterproductive, so we envision where we don't really compress the input. To ensure against this we allocate twice the memory of the existing buffer. If we can't allocate the memory, we return a length of zero indicating a problem in the compression.

```
new = malloc(len*2);
if( new == NULL )
    return 0;
```

The most complicated part of the process is finding the runs of characters that can be expressed as compressed run lengths—those two-byte codes we talked about earlier. We need to find these ahead of any compression pass, because on the decompression pass, the buffer preceding the run length code will already be uncompressed. This means that the run count and distance needs to be expressed in terms of the uncompressed buffer.

```
/* find forward references */
runs = malloc(len * sizeof(int));
if( runs == NULL )
    return 0;
populate_runs(buf, runs, len);
```

We rely on a routine called `populate_runs` to find them, saving them in a dynamically allocated array. Again, we return with an error if there are problems allocating the array. What happens in that first pass through the uncompressed buffer? Patience, please. We'll get there.

Once the information about the contents of the buffer is gathered, we're ready to make the actual compression pass.

```
/* now we run through the text, and
```

# Work

```
do the actual compression */
for( p = buf, np = new;
    p < (buf+len); )
{
```

Our for loop is the obvious one: initialize our pointers into the input and output buffers to their beginnings, and continue until we reach the end of the input buffer.

```
/* if we've got a backreference, use it */
if( (rr=runs[p-buf]) != 0 ) {
    *np++ = 0x80 | ((rr >> 8) & 0x3F);
    *np++ = (rr & 0xFF);
    p += (rr & 0x07) + 3;
    continue;
}
```

The first step is easy. If we've got an entry in the runs array, we have a valid back reference, which can be turned into a two-byte code. Skip over the number of bytes this new code represents, and continue with the next iteration of the loop.

```
/* special case for character we need
   to escape: 1->8 & 0x80->0xFF */
if( COUNTABLE(*p) )
{
    char *bp = np++;
    *bp = 1;
    *np++ = *p++;
    while( COUNTABLE(*p)
        && *bp < 8 && HAVECHARS )
    {
        (*bp)++;
        *np++ = *p++;
    }
    continue;
}
```

Failing a simple run-length compression, there might be characters that need to be escaped because they're in a range that represents one of the special codes. These are characters for which the COUNTABLE macro is true. We save a place for the count, then walk over up to eight of them at a time, backfilling the count as we go.

In the next clause, we compress a space when we can.

```
/* compress a leading space */
if( *p == ' ' &&
    HAVECHARS &&
    runs[p+1-buf] == 0 &&
    p[1] >= 0x40 && p[1] <= 0x7F )
{
    *np++ = *(++p) | 0x80;
    p++;
    continue;
}
```

Notice that we're also checking that there's no run of characters beginning right after this particular space character. If there is, it will provide better compression than the collapsed space, since the minimum run length is three.

```
    *np++ = *p++;
}
```

We close the for loop by copying any other character from the input buffer to the output buffer without change.

```
*np = 0;
if( np-new > len )
    return 0;

memcpy(buf, new, np-new);
free(new);
free(runs);
return np-new;
}
```

We're now done. We return a zero for error if we didn't succeed in making the buffer smaller. Why? Because the PDB file format has a maximum of a 4,096 byte buffer. Typically, that's the size of the uncompressed buffer we'll be handed. If we haven't managed to make it smaller, we will cause buffer size trouble for a program on the Palm device itself. In the normal case, though, we copy the compressed buffer over the old one, and return the length.

That still leaves us without the answer to the question about collecting character runs, which we will now reveal:

```
void
populate_runs(unsigned char *buf,
              int *runs, int len)
{
    int i, d, l;

    memset(runs, 0, len*sizeof(int));
```

We begin in the usual way, making sure to set the freshly allocated runs array to zeroes.

```
for( i = 0; i < (len-6); i++ )
{
    d = nextmatch(buf+i, len-i, &l);
    if( d > 0 )
        runs[d+i] = (d << 3) | (l - 3);
}
}
```

The body of the routine is pretty simple, too, since we've put the hard part off to nextmatch. When we find a run, we save the information about it in the same form in which it will be encoded in the compressed buffer, that is, length shifted left by three bits, OR'ed with the length of the run minus three.

So the problem now boils down to finding the next match

of a character sequence in the input buffer. Given a starting point in the buffer, we want to find how far it is to the same set of between three and ten characters.

```
int
nextmatch(unsigned char *from,
          int buflen, int *matchlen)
{
    unsigned char *to;
    size_t n;
    size_t max;

    buflen = (buflen > 2047) ?
        2047 : buflen;
```

Because we only have 11 bits to store, the maximum distance to the matching string is 2,047. If we're closer than that to the end of the buffer, we'll use the shorter distance.

```
for( to = from+1;
     to < (from+buflen); to++ )
{
    n = 3;
    max = (buflen <= 10) ? buflen : 10;
    if( memcmp(from,to,n) != 0 ) continue;
    while( n <= max &&
```

```
        memcmp(from,to,n) == 0 )
        n++;
    *matchlen = (int) -n;
    return(to - from);
}
return 0;
}
```

Our main loop does a long search, returning the distance to the next forward match, and storing the length of the match in the address passed as 1. If we fall through the bottom of the loop, we then return zero. This is a completely brute force loop. We haven't done any tuning on it at all. How could we improve this routine? If we find a match close by, should we pass it up for a possibly longer match a bit farther up in the buffer? What changes would we need to make to the calling `compress` routine to ensure that runs of Latin-1 characters will be used, and that countable strings of characters will not be used instead. Using one of the more efficient string search algorithms, like Boyer-Moore, is not much help in this circumstance where we are mostly searching for short strings, but we have not done the analysis to prove it.

We finish `comp.c` with a set of testing code. We don't have space to show it to you, but you can find it in the usual places, as detailed at the end of this column.

```
#ifndef REAL_MODULE
/* this portion is for
   stand-alone testing only */
main( int ac, char *av[] )
{
    ...
}
#endif
```

Notice that the test code compiles only if we have not defined `REAL_MODULE`. This allows us to compile without the testing functionality by including `comp.c` after a `REAL_MODULE` as we do in the `XS` file.

## Using the New Function

Now that we are capable of doing compression, some minor changes need to be made to the `txt2palm` script we wrote for January. We'll show these as the output of `diff -u2` and run through them briefly.

```
@@ -4,4 +4,5 @@
     use strict;
     use Getopt::Std;
+use PalmComp;

    ## useful constants

Obviously, we need to include the new module.

@@ -15,10 +16,7 @@
    ## command line flags
```

# Work

```
-use vars qw($opt_c);
-getopts("c") ||
- die "Usage: $0 [-c] title...\n";
-die "compression not supported yet\n"
- if( $opt_c );
-
+use vars qw($opt_c $opt_v);
+getopts("cv") ||
+ die "Usage: $0 [-cv] title...\n";
```

We add a `-v` flag to report on the compression we achieved. We change the usage messages to match the new functionality.

```
@@ -34,4 +32,5 @@
    my $text = <STDIN>;
    my $filesize = length($text);
+my $compsize = 0;

    ## We have a number of steps to do to
```

We add a place to sum the compressed file size.

```
@@ -55,5 +54,10 @@
    $pos += $RECMAX) {
    my $rec = substr($text, $pos, $RECMAX);
-   $rec = compress($rec) if($opt_c);
+   my $len = length($rec);
+   ($rec, $len) = compress($rec, $len)
+   if($opt_c);
+   die "Oops: unable to compress\n"
+   if($opt_c && ! $len);
+   $compsize += $len;
    push(@records, $rec);
}
```

We use the new calling sequence for `compress`, as we've discussed, throwing a fatal error if there was a compression failure. We also sum the compressed size of the data into `$compsize`.

```
@@ -147,8 +151,7 @@

    print @records;

-#####
-
-# we don't provide a real compression
-# routine yet....
-
-sub compress() { $_[0] }
+ # print status if -v
+warn "$filesize compressed to $compsize, ",
+ "compression ratio = ",
+ int($compsize*100/$filesize), "%\n"
+ if( $opt_c && $opt_v );
```

Lastly, we can remove the stub compression routine, and finish off the program with a report on our compression ratio, if the user requested the information.

To install and run this new software, do the same thing you would with any Perl package downloaded from the Net:

```
perl Makefile.PL
make install
```

This will install the package in your designated place for Perl modules. At this point, you can do things like:

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

There you have it. An implementation of a simple compression scheme we've discussed before, some new tricks for Perl packages, and some added functionality to code we showed you before. Not bad for our month's amusement. Some time in the future, we'll show you a way to generate test data for text processing programs such as these, but next time, we'll be back with a discussion of how to organize the epigrammic wisdom of the ages.

Lastly, it's with sadness that we note the death on Feb. 7, as we were working on this column, of Dale Evans, who co-wrote the cowboy standard from which we take our sign off. As usual, we wish you, and especially Dale, 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>.

## Marginal Notes Added to the Proofs:

In the month between writing and proofing this article, we've improved the compression of our PalmComp module in some of the ways suggested in the main text. The improved version (which includes tests in Perl) is bundled with the original at our Web sites.

Also in the intervening month, three interesting bits of data have come our way. First, John Gruenfelder has developed a better compression algorithm for documents to be read on the Palm. He intended it for reading larger works, such as the free books from Project Gutenberg (<http://promo.net/pg>). Because the compression scheme is incompatible, it requires new conversion tools and a new reader, called GutenPalm. They're available at <http://gutenpalm.sourceforge.net>. Second, a company named Agenda Computing is working on a Linux-based hand held device, the Agenda VR3. See their Web site, <http://www.agenda-computing.com/>, for details. It looks like vaporware as we write, but it's clearly got some possibilities. Third, after we submitted this article, the Perl package came to our attention. This package provides a way to drop C (or some other language) code directly in-line in a Perl program, without wrapping it in a file first. For details see <http://www.perl.com/pub/2001/02/inline.html> or the package at CPAN, <http://www.cpan.org/>.