Synthesizing EPUB Files

September 2011

Abstract

The EPUB electronic book format is the current main standard for publication of electronic books and an important format for the interchange of electronic publications. This note explores the format by showing how to synthesize simple publications in the EPUB format.

This is revision 2.43, of 2017/05/02. It was printed 2017/5/2.

Introduction

With the profusion of devices and programs for reading electronic books, there has also been a profusion of e-book formats. Seemingly each device has its own native format. Initially, in the very first years of the current century, a standard called the Open eBook Publication Structure emerged for the preparation of electronic content. Unfortunately, each device or software provider used the OEB format as source material for their own converter to a native format. Two prime examples were the Mobipocket format developed by the French company of the same name for use by readers on various Personal Digital Assistants, and Microsoft's LIT format used by its Pocket PC product. This profusion of native formats — even ones generated from common source material — meant either that consumers of electronic media had to convert their own e-books or producers of electronic media had to make multiple formats available.

As the International Digital Publications Forum replaced OEB with EPUB, many devices have corrected that early native format error. They are able to consume EPUB as a native format, even when (like with the Sony product line) they also have a proprietary format. Additionally, desktop software, (such as Adobe's Digital Editions) and many portable device applications (such as FBReader for Android, Lexcycle Stanza for iPhone, and Freya for Windows Phone) consume the EPUB format.

The exception that proves the universality rule is the Amazon Kindle. Amazon bought Mobipocket in 2005, and used the Mobi format as the basis for the Kindle. However, the Mobipocket Creator software for user-generated Kindle content is capable of accepting both OEB and EPUB books as input.

So what's in an EPUB file? At first glance, it is a ZIP archive containing HTML files with some connective goo. Fortunately, there are two ways to

understand how something is made: you can take one of them apart[1] or you can build one of them.

In a previous piece of software one of us examined a number of common e-book formats for the Palm handheld by deconstructing them. In this case, it makes more sense to generate EPUB files to examine the fine details of how they are composed. We'll do this by converting text files (as we explain below). Even if the full program is not useful as-is, the parts that create EPUB files are likely helpful as stand-alone modules.

How will we know when we've succeeded? Because the EPUB files we generate will be readable on our devices and by the desktop applications. We also can rely on the `epubcheck` program, which provides a very complete syntax and consistency check and is available as a Google Code project, to validate our EPUB structure.

Overview of EPUB

Our description of EPUB files above as "a ZIP archive containing HTML files" is actually a gross oversimplification as we'll see in the next pages.

There are actually three interlocked standards that comprised EPUB:

- **OCF**, the Open Container Format, which describes the layout of files within the EPUB. The EPUB can either be presented wrapped as a ZIP archive, which is the most common form to load into devices or software, or as files on a file system under a central root directory. OCF also describes the restrictions on the names of the files in the container.
- **OPF**, the Open Packaging Format, which describes the control files in the EPUB. These files provide the overall metadata for the book, and describe its internal structure.
- **OPS**, the Open Publication Structure, which describes the structure of the content of the book.

Warning we need to throw in someplace... Many of these devices and applications also use digital rights management hooks in the EPUB format. We will mention these, but a full exploration of the DRM aspects of EPUB is out of scope for this article. *Ditto some notes about font mangling. Ditto some notes about EPUB 2.0.1 vs EPUB 3.0.*

Overview of Literate Programming

Language warning: This program is written using the literate programming tool `noweb`, which allows us to intersperse narrative text with Perl program fragments. (We would have used our normal literate programming tool `CWEB`, but the problem at hand is better suited to the notational density of Perl.) The

[1] "Dad: the Air Force has much cooler things to take apart than bicycles." — Airman James Schwarzin-Copeland, explaining his career choice to a dubious parent.

program chunks are all named, and flow up to a chunk with the name "*". The descriptive names of the chunks are numbered and cross-referenced for ease.

Plan of the Program

We'll generate our EPUBs from flat text files. Why from simple text? Because it saves us having to worry about conversion *from* and only about conversion *to*. Also, because in the past, we've found simple text converters helpful, as with an earlier filter to convert lists of standard RFC822 mail messages into Palm e-book files so we could read long mail threads on our handhelds.

An alternative would be to generate EPUBs from a list of web pages, using Perl's `lwp` package to collect the HTML from the URLs. However, we'd still have to parse the HTML to extract the interesting text and elide the advertisements. Plus, the web site `instapaper.com` already does this quite well.

To do: We could have plug-in procedures for text conversion — means we need to do detection, followed by dispatch. Oh, we've got an RFC 822, so run this routine; no, I don't see a mail header, run this instead. That allows addition of other decoders easily.

So, how would we explain what we're doing in the form of a manual page for the program? We'd start with something like this:

3 ⟨*POD* 3⟩≡

```
=head1 NAME

text2epub -- convert RFC822 files into EPUB documents

=head1 SYNOPSIS

text2epub <flags> <files>

=head1 DESCRIPTION

I<text2epub> processes text files
into a standard EPUB e-book,
with each text file presented as a separate "chapter"
in the book.
```

Assembling

To do: Need to discuss the packaging requirements, especially the ZIP wrapper.

We will construct our program in the inverse of the way we would deconstruct an EPUB file. We looked at the contents of EPUB files by unzipping them, and poring over the individual components. Thus, we begin by writing the code to prepare the ZIP file.

We assume that we have stored the component files for the EPUB in a working directory `$wd` and that we have lists of them in three arrays: one containing the control information `@control`, one containing the content files `@content`, and one containing (optional) image files `@images`.

The EPUB file must start with an uncompressed mimetype file, hence the `-0` flag. There must be no extended header data in the ZIP file for the mimetype file, hence the `-X` flag. The contents of the mimetype file are mandated.

After we've assembled the EPUB, we add the mimetype file to `@control` so that it's included in the list of files for later clean up.

We finish by noticing that we've assembled the EPUB file in the working directory. This means we need to change back to the directory from which we launched this script, and move the EPUB file to its intended full path name.

4 ⟨*utilities to assemble the EPUB* 4⟩≡

```
sub assemble_final_package() {
    my $e = basename($epub);
    (my $launch_dir = qx(pwd)) =~ s/\s//g;
        # change into the working directory
    unless (chdir $wd) {
            die "can't change into $wd?!?!";
    }
        # do we want verbose output from the zip command?
    my $q = ($opt_v) ? "" : "-q";
    ⟨create mimetype file 5a⟩
    system("zip -0 -X $q $e mimetype");
    ⟨capture directory names 5b⟩
    system("zip $q $e " . join(" ", @dirlist));
    system("zip $q $e " . join(" ", reverse @control));
    system("zip $q $e " . join(" ", @content, @images));
    unless (chdir $launch_dir) {
            die "can't change back to $launch_dir?!?!";
    }
    qx(mv $wd/$e $epub);
        # add mimetype to the control file list
        #  so it can be removed later
    push @control, "mimetype";
}
```

Defines:
`assemble_final_package`, used in chunk 35c.

Uses basename 26b, @content 6, @control 6, @dirlist 6, $epub 34b, @images 6, $opt_v 37a, and $wd 6.

As we noted, the mimetype file is dictated:

5a ⟨*create mimetype file* 5a⟩≡

```
unless (open M, "> mimetype") {
    die "can't create mimetype file";
}
print M "application/epub+zip";
close M;
```

Some of the files in our EPUB may be in subdirectories. To make the EPUB consistent, we need to zip the directories as well as their contents. Let's extract the names of the directories from the @control, @content and @images lists. The resulting list of directories needs to be a global to facilitate cleanup later.

5b ⟨*capture directory names* 5b⟩≡

```
@dirlist = ();
my %dirlist = ();
foreach (@control, @content, @images) {
    my $d = dirname($_);
    next  unless (length($d));
    my @d = split /\//, $d;
    my $dd;
    foreach (@d) {
        $dd .= "$_/";
        $dirlist{$dd}++;
    }
}
@dirlist = sort keys %dirlist;
```

Uses @content 6, @control 6, $dirlist 6, @dirlist 6, dirname 26a, and @images 6.

This all assumed some globals, which we should declare. The only confusing one may be `$wd`, the directory in which we have gathered the components of the EPUB, and `$suffix`, the suffix for the working directory name.

6 ⟨*declare global variables* 6⟩≡

```
my @control;
my @content;
my @images;
my @dirlist;
my $wd;
my $suffix = ".work";
```

Defines:
`$content`, used in chunks 12, 25a, and 35b.
`@content`, used in chunks 6, 4, 6, 5b, 6, 9, 10a, 13, 15b, 6, 25a, 35a, and 36b.
`$control`, never used.
`@control`, used in chunks 6, 4, 6, 5–7, 11a, 13, 14a, 23b, and 36b.
`$dirlist`, used in chunk 5b.
`@dirlist`, used in chunks 4, 5b, and 36b.
`$images`, never used.
`@images`, used in chunks 6, 4, 6, 5b, 6, 9, 6, and 21a.
`$suffix`, used in chunks 6, 32, and 34a.
`$wd`, used in chunks 6, 4, 6, 7, 9, 11a, 13–15, 21a, 23b, 25a, 32, 34a, and 36b.

Components

Assembling the pieces is one thing, but we have a lot to create. We have a number of required files, which give structure to the book, and provide pointers to the other pieces. Each of these routines will be responsible for adding the files they create to the `@control`, `@content`, or `@images` lists as appropriate.

We begin with an Open Packaging Format (OPF) file, which contains the wrapper for the entire book. For convenience, we'll derive the OPF file name from the target EPUB's name. We need to return the file name created because we'll use it later.

The OPF file contains the manifest, reading order (also called the spine), and indexing information.

The manifest contains a unique identifier so that our reader can tell this book from other books loaded into it. The standard does not specify what this unique identifier is, but for a book, it would likely be the ISBN. (The reader is not allowed to fail if it encounters two differnt books with the same unique identifier, which rather contradicts the use of "unique" in that definition.) The identifier is named in the package tag, and defined in the identifier tag in the metadata. We'll use the system time as our identifier.

7 ⟨*utilities to assemble the EPUB* 4⟩+≡

```
sub create_opf() {
    my $opf = basename(filename($epub)) . ".opf";
    unless (open F, "> $wd/$opf") {
        die "can't create OPF file $opf";
    }
# header information: XML version & IDPF namespace
    print F '<?xml version="1.0" encoding="UTF-8" ',
        'standalone="no"?>', "\n";
    print F '<package version="2.0" ',
        'xmlns="http://www.idpf.org/2007/opf" ',
        'unique-identifier="BookId">', "\n";
    ⟨OPF metadata 8a⟩
    ⟨OPF manifest creation 9⟩
    ⟨OPF spine creation 10a⟩
    ⟨OPF guide creation 10b⟩
    print F "</package>\n";
    close F;
    push @control, $opf;
    return $opf;
}
```

Defines:
`create_opf`, used in chunk 35b.
Uses `basename` 26b, `@control` 6, `$epub` 34b, `filename` 26c, `$opf` 36a, and `$wd` 6.

The OPF file contains metadata about the book, such as the author and title. The minimal XML namespaces for this particular section are Dublin Core and OPF.

This section contains the definition of the unique identifier we described and declared above. We are allowed to optionally include the "scheme" attribute in the identifier tag to name the system that generated the unique identifier, for example, `scheme="ISBN"`.

It's worth noting that the manifest can contain a `<rights>` tag, which contains information about rights for the content. Rights management is out of scope for this article. **To do:** discuss how this relates to DRM-ness.

To do: Include the "file-as" attribute for the author, by inverting first and last name.

8a ⟨*OPF metadata* 8a⟩≡

```
my $timestamp = strftime("%Y-%m-%d", localtime(time()));
print F " <metadata ",
    'xmlns:dc="http://purl.org/dc/elements/1.1/" ',
    'xmlns:opf="http://www.idpf.org/2007/opf">', "\n";
print F "  <dc:title>$book_title</dc:title>\n";
print F "  <dc:creator opf:role=\"aut\">",
    "$book_author</dc:creator>\n";
print F "  <dc:language>en</dc:language>\n";
print F "  <dc:date>$timestamp</dc:date>\n";
print F "  <dc:identifier id=\"BookId\">",
    $bookid, "</dc:identifier>\n";
print F " </metadata>\n";
```

Uses `$book_author` 34b, `$bookid` 36a, and `$book_title` 34b.

That last part required the POSIX module so we could use the `strftime` function.

8b ⟨*include packages* 8b⟩≡

```
use POSIX;
```

The manifest provides a list of all the content files in the book — the text, the CSS, the images. We would also list any font files or included XML schemas. Lastly, we list the index file here. (More about the index file when we discuss the `<spine>` section.) Each entry in the manifest has to contain an id tag, the file name, and the file type.

To do: need to add cover picture to the manifest, too

9 ⟨*OPF manifest creation* 9⟩≡

```
print F " <manifest>\n";
my $n = 1;
foreach (@content) {
    print F ’  <item id="cc’, $n++, ’" href="’,  $_ ,
        ’" media-type="application/xhtml+xml" />’, "\n";
}
print F ’  <item id="cover" href="’, basename($opt_C),
    ’" media-type="image/’, image_type($opt_C), ’" />’,
    "\n"
        if ($opt_C);
$n = 1;
foreach (@images) {
    print F ’  <item id="im’, $n++, ’" href="’, basename($_),
        ’" media-type="image/’, image_type("$wd/$_"), ’" />’, "\n";
}
print F ’  <item id="css" href="’, $css,
    ’" media-type="text/css" />’, "\n";
print F ’  <item id="ncx" href="’, $ncx,
    ’" media-type="application/x-dtbncx+xml" />’, "\n";
print F " </manifest>\n";
```

Uses `basename` 26b, `@content` 6, `$css` 36a, `@images` 6, `image_type` 31a, `$ncx` 36a, `$opt_C` 37a, and `$wd` 6.

We also need a spine, which tells us the reading order of the components. The spine contains only the list of content files. The items in the spine are designated by referring to the item ids from the manifest. Items in the spine can be designated auxiliary — that is, not read in linear order, for example in a pop up window — by including the `linear="no"` attribute. The default if the "linear" attribute is missing (or is the attribute `linear="yes"` is present) is for the file to be what is called a primary file.

The spine keyword has the required attribute of `toc`, which refers back to the item id in the manifest for the NCX index file. We'll fully flesh out the index file in a moment.

10a ⟨*OPF spine creation* 10a⟩≡

```
print F " <spine toc=\"ncx\">\n";
$n = 1;
foreach (@content) {
    print F ’  <itemref idref="cc’, $n++, ’" />’, "\n";
}
print F " </spine>\n";
```

Uses `@content` 6.

The OPF file also has one more section, the guide, which is optional. This section names the common sections of the book — table of contents, cover — so they can be found easily. We include this section because it gives the EPUB reader a way to easily find the first text page and the table of contents.

→Can we use a single print statement with EOF here?

10b ⟨*OPF guide creation* 10b⟩≡

```
print F " <guide>\n";
print F ’  <reference href="’, $cover, ’" ’,
    ’title="Cover" type="cover" />’, "\n"
            if ($opt_C);
print F ’  <reference href="’, $toc, ’" ’,
    ’title="Table of Contents" type="toc" />’, "\n"
            unless ($opt_t);
print F ’  <reference href="’, $first_text, ’" ’,
    ’title="Text" type="text" />’, "\n";
print F " </guide>\n";
```

Uses `$cover` 36a, `$first_text` 36a, `$opt_C` 37a, `$opt_t` 37a, and `$toc` 36a.

The NCX file is the required index file. NCX was originally defined in ANSI/NISO Z39.86-2005, the Specification for Digital Talking Books, or DTBs. NCX stands for "Navigation Control file for XML."[2]

There are two parts that we are concerned with: the metadata and the navigation map. We'll construct them in the next few sections, and return the name of the NCX file, after adding it to the control file list.

11a ⟨*utilities to assemble the EPUB* 4⟩+≡

```
sub create_ncx() {
    my $ncx = basename(filename($epub)) . ".ncx";
    open F, "> $wd/$ncx";
    print F <<"EOF";
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ncx PUBLIC "-//NISO//DTD ncx 2005-1//EN"
    "http://www.daisy.org/z3986/2005/ncx-2005-1.dtd">
<ncx version="2005-1" xml:lang="en-US"
    xmlns="http://www.daisy.org/z3986/2005/ncx/">
EOF
    ⟨NCX header and metadata 11b⟩
    ⟨NCX navigation map 12⟩
    print F "</ncx>\n";
    close F;
    push @control, $ncx;
    return $ncx;
}
```

Defines:
`create_ncx`, used in chunk 35b.
Uses `basename` 26b, `@control` 6, `$epub` 34b, `filename` 26c, `$ncx` 36a, and `$wd` 6.

The NCX metadata duplicates information that's already in the OPF file, but much of it is required by the NCX chapter of the talking book specification. (Interestingly, even though the metadata is required by the NCX chapter of the DTB specification, it doesn't appear to actually be confirmed by the `epubcheck` program.) We include this data anyway to be compliant with the broader spec.

11b ⟨*NCX header and metadata* 11b⟩≡

```
print F '<head>
  <meta name="dtb:uid" content="', $bookid, '" />
  <meta name="dtb:depth" content="1" />
  <meta name="dtb:totalPageCount" content="0" />
  <meta name="dtb:maxPageNumber" content="0" />
</head>', "\n";
print F "<docTitle><text>$book_title</text></docTitle>\n";
print F "<docAuthor><text>$book_author</text></docAuthor>\n";
```

Uses `$book_author` 34b, `$bookid` 36a, and `$book_title` 34b.

[2] The OPF specification also calls it "Navigation Center eXtended," which appears nowhere else that we can detect.

The navigation map is ... **To do:** more explanation

12 ⟨*NCX navigation map* 12⟩≡

```
print F "<navMap>\n";
foreach (0..$#content) {
    my $n = $_+1;
    print F "<navPoint playOrder=\"$n\" id=\"play_$n\">\n",
        "  <navLabel><text>", $titles[$_],
            "</text></navLabel>\n",
        '  <content src="', $content[$_], '" />', "\n",
        "</navPoint>\n";
}
print F "</navMap>\n";
```

Uses `$content` 6 and `$titles` 14b.

We may have a cover image specified on the command line. If so, we want to add it at the head of the contents array. This means we need to call this after we create the table of contents, which is also inserted at the head of the contents array. We return the name of the file containing the cover reference, which we create here, not the cover image itself.

13 ⟨*utilities to assemble the EPUB* 4⟩+≡

```
sub create_cover() {
    my $cover = basename(filename($epub)) . "-cover.html";
    my $image = basename($opt_C);
    open F, "> $wd/$cover";
    print F <<"EOF";
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Cover</title>
</head>
<body>
 <div class="body">
  <img src="$image" alt="cover image" height="100%" />
 </div>
</body>
</html>
EOF
    close F;
    if (-r $opt_C) {
        qx(cp $opt_C $wd);
    } else {
        warn "can't open cover image $opt_C\n"
    }
    unshift @content, $cover;
    unshift @titles, "Cover";
    push @control, $image;
    return $cover;
}
```

Defines:
`create_cover`, used in chunk 35b.
Uses `basename` 26b, `@content` 6, `@control` 6, `$cover` 36a, `$epub` 34b, `filename` 26c, `$opt_C` 37a, `@titles` 14b, and `$wd` 6.

In addition to the mimetype file, the other mandatory file in the package is container.xml, which provides a pointer to the OPF file. It must appear in the META-INF subdirectory of the EPUB. This routine creates the container file, using the global name of the OPF file.

14a ⟨*utilities to assemble the EPUB* 4⟩+≡

```
sub create_container() {
    mkdir_p "$wd/META-INF";
    unless (open F, "> $wd/META-INF/container.xml") {
        die "can't create container.xml file";
    }
    print F <<"EOF";
<?xml version="1.0"?>
<container
      xmlns="urn:oasis:names:tc:opendocument:xmlns:container"
      version="1.0">
  <rootfiles>
    <rootfile full-path="$opf"
      media-type="application/oebps-package+xml" />
  </rootfiles>
</container>
EOF
    close F;
    push @control, "META-INF/container.xml";
}
```

Defines:
`create_container`, used in chunk 35b.
Uses `@control` 6, `mkdir_p` 27a, `$opf` 36a, and `$wd` 6.

We've posited another global array for generating the table of contents:

14b ⟨*declare global variables* 6⟩+≡

```
my @titles;
```

Defines:
`$titles`, used in chunks 12 and 25a.
`@titles`, used in chunks 13, 25a, and 35a.

Reading and processing the input files

We've deliberately deferred discussing how the input files are processed. If you only cared about what is in an EPUB, you can stop reading now. However, we need to write some code to get the read the data for our book, which is what we'll do now.

The following routines convert the text input to an HTML file, and puts it in the global working directory where we're gathering our components. We return the name of the output file and a "title". The title is either the name of the input file, or (in the case of an e-mail file) the first line.

We'll assemble this routine in chunks over the next few sections.

15a ⟨*utilities to convert the text files* 15a⟩≡

```
sub convert_file($) {
    my $in = shift;
    unless (open IN, "< $in") {
        warn "unable to open input file $in\n";
        return ("", "");
    }
    ⟨body of convert file 15b⟩
}
```

Defines:
`convert_file`, used in chunk 35a.

We define and open the output file right away. This allows us to short-circuit processing if we have no place to put the results. We check to make sure the output file name isn't already used. The standard says names can't be duplicated after case folding, so our check includes that, too. We have a fallback scheme for alternate output file names. (We might also have, for example, the same input file name from different source directories, or want to process the same input file twice. But this is us just being carefully pedantic.)

15b ⟨*body of convert file* 15b⟩≡

```
my $out = basename($in);
my @x;
while (grep(/$out.html/i, @content)) {
    $out .= "-";
}
$out = "$out.html";
unless (open OUT, "> $wd/$out") {
    warn "unable to create output file $out\n";
    return ("", "");
}
```

Uses `basename` 26b, `@content` 6, and `$wd` 6.

One of the compelling uses for this script will be in converting e-mail into EPUBs, so our basic conversion routine does a little checking to see if our input file is mail, and does some supplementary processing on it. To make this check possible, we begin by grabbing the first blank-line-delimited paragraph.

If there is no header block — if we've got a pure text file — we add a "textfile" paragraph at the top as a delimiter, and make the first line of the file the "subject."

16a ⟨*body of convert file* 15b⟩+≡

```
$/ = "\n\n";
my $h = <IN>;
undef $/;
my ($s, $b);
$b = <IN>;
close IN;
⟨insurance for empty text body 16b⟩
⟨get charset header 19a⟩
if ($h =~ m/From:/s && $h =~ m/Subject:/) {
    ($h, $s) = convert_header($h);
} else {
    $h =~ m/^(.*?)\n/s;
    $s = $1;
    $b = $h . $b;
    $h = "<p class=\"textfile\"></p>\n";
}
⟨errant control characters 17a⟩
$b = convert_body($b);
my $p = create_prolog($s);
print OUT $p, "<body>\n", $h, $b, "</body></html>\n";
close OUT;
return ($out, $s);
```

Uses `convert_body` 19c, `convert_header` 18, and `create_prolog` 23a.

Very rarely, we'll have an empty file body. In that case, to prevent downstream complaints, we make sure we define an empty body.

16b ⟨*insurance for empty text body* 16b⟩≡

```
$b = "" unless defined $b;
```

We find, sometimes, that transmission errors or other problems drop control characters into the body text. These cause trouble with our EPUB readers. To prevent that we do a little prophylactic fixup by stripping out the bad control characters. As a special case, rather than stripping form-feeds, we'll replace them with newlines.

17a ⟨*errant control characters* 17a⟩≡

```
if ($b =~ m/[$bad_control]/) {
    warn "errant control characters removed from $in:\n";
    while ($b =~ m/(^.*?[$bad_control].*?$)/mg) {
        my $x = $1;
        $x =~ s/[$bad_control]/sprintf "<0x%02x>", ord($&) /eg;
        warn "  $x\n";
    }
    $b =~ s/\f/\n\n\n/g;
    $b =~ s/[$bad_control]/ /g;
}
```

Some control characters are necessary — newline and tab, for example. Let's start with a string of all of them and strip out the ones that aren't "errant". (Yes, we could just do that directly, but it's harder to understand what's in the list we want to preserve.)

17b ⟨*declare global variables* 6⟩+≡

```
my $bad_control;
foreach (1..0x1f) { $bad_control .= chr($_); }
$bad_control =~ s/[\t\n\r]//sg;
```

If we have a mail file, we only care about keeping a few lines in the header, and want to add some markup to those.

18 ⟨*utilities to convert the text files* 15a⟩+≡

```
sub convert_header($) {
    my $h = shift;
        # convert the header to UTF-8
    $h = text_to_UTF8($h);
        #  prevent confusion about some embedded characters
    $h = escape_html($h);

    my $s = ($h =~ m/^Subject:\s+(.*)$/m) ?
        $1 : "default subject";
    my $hh = "<p class=\"subj\">$s</p>\n";

    $hh .= "<p class=\"from\">From: $1</p>\n"
        if ($h =~ m/^From:\s+(.*)$/m);
    $hh .= "<p class=\"date\">Date: $1</p>\n"
        if ($h =~ m/^Date:\s+(.*)$/m);

    # add a little extra space at the end
    $hh .= "<p class=\"secbrk\"></p>\n";

    return ($hh, $s);
}
```

Defines:
  `convert_header`, used in chunk 16a.
Uses `text_to_UTF8` 27b.

We also need to check the header lines for a "Content-Type:" header. If there is one, we need to capture the "charset" which we can will need to convert the file to UTF-8. Since the Content-Type header is often split across lines, we'll join lines first; we want to check only the specified header line, rather than looking for any instance of "charset" in the header block. This has to happen before we delete the header lines we don't want to preserve for our converted file. We finish by ensuring that we have knowledge of the named charset; we default to `iso-8859-1` and issue a warning otherwise. We'll define the tables later on.

To do: This completely ignores use of "Content-Transfer-Encoding: quoted-printable", in which we transfer in 7-bit ASCII with escapes like `=3D` and `=` at the end of wrapped lines.

19a ⟨*get charset header* 19a⟩≡

```
$h =~ s/\n[ \t]+/ /sg;
$h =~ m/^Content-Type: .*?charset="?([\w-]+)/mi;
$charset = length($1) ? lc($1) : $default_charset;
if (!exists($mapping{$charset})) {
    warn "unrecognized charset $charset in file $in --"
            . " assuming $default_charset\n";
    $charset = $default_charset;
}
```

19b ⟨*declare global variables* 6⟩+≡

```
my $charset;
my $default_charset = "iso-8859-1";
```

Convert the body

Next, want to convert the body text.

19c ⟨*utilities to convert the text files* 15a⟩+≡

```
sub convert_body($) {
    my $b = shift;
    ⟨body of convert body 20⟩
}
```

Defines:
`convert_body`, used in chunk 16a.

We begin by doing some full-body conversion on the text. We remove any e-mail signature blocks, fix up some special characters, and make the line endings all standard.

20 ⟨*body of convert body* 20⟩≡

```
    # convert the text out of the native charset
$b = text_to_UTF8($b);
    # strip off the signature block in the message
$b =~ s/^--\s*\n.*//ms;
    #  prevent confusion about some embedded characters
$b = escape_html($b);
    # fix up funny line endings
$b =~ s/\r//sg; # strip possible DOS line endings
$b =~ s/[\t ]*$//mg;    # strip blanks at line ends
```

Uses `text_to_UTF8` 27b.

We may want to insert images into the text. If that's the case, the image file names will be specified using the Markdown syntax, on a standalone line with the form:

`![Alt text](path/to/image.jpg){size}`

where the "Alt text" and "size" are optional. "size" (which is not part of the Markdown standard) is of the form `<digits>[%][h|w]` – a percentage size, defaulting to height, with an optional percent sign, and an optional `h` or `w` specifier for height (default) or width. For example, `{50%w}` is fifty percent width; `{75}` is three-quarters height. We need to convert those lines into image tags, copy the image files into the working directory, and add the image file names to the `@images` list. (Notice that we're renaming the images to neutral names on the copy so that we don't risk odd formatting problems if the image is named something like `davinci_mona_lisa.jpg` in which case we'd attempt to italicize part of the name.)

21a ⟨*body of convert body* 20⟩+≡

```
while ($b =~ m/^!((\[.*?\])?\((.*?)\)(\{.*?\})?)/msg) {
    my $tag = $1;
    my $im = $3;
    my $imf = (-r $im) ? $im : "$opt_i/$im";
    if (! -r $imf) {
        warn "can't find image file $im\n";
        $b =~ s/^!/ /m;
        next;
    }
    $imagecount++;
    my $seq = "image$imagecount" . extension($im);
    push @images, $seq;
    system("cp $imf $wd/$seq");
    my $alt = ($tag =~ m/.*?\[(.*?)\]/s) ? $1 : $im;
    $alt =~ s/_/-/g;
    $alt = "alt=\"$alt\"";
    my $size = ($tag =~ m/.*?\{(.*?)\%?[hw]?\}/) ? $1 : 100;
    $size = (($tag =~ m/w\}/) ? "width" : "height")
        . "=\"$size\%\"";
    $b =~ s/^!\Q$tag\E$/<p><img src="$seq" $alt $size \/>/sm;
}
        # only one blank line around each image tag
$b =~ s/\n+(<p><img .*?>)\n+/\n\n$1\n\n/sg;
```

Uses `extension` 26d, `@images` 6, `$opt_i` 37a, and `$wd` 6.

And declare the global variable we used:

21b ⟨*declare global variables* 6⟩+≡

```
my $imagecount = 0;
```

Now we'll deal with the paragraph text.

We'll begin by stripping the white space off the the beginning and the end of the text block, so that we don't accidentally insert extra open or close paragraph tags, particularly an open tag at the end.

22a ⟨*body of convert body* 20⟩+≡
```
$b =~ s/^\s+//s;
$b =~ s/\s+$//s;
```

Some of the paragraphs may be broken by blank lines, some designated by indented first lines. We ensure the paragraphs are all broken by blank lines, which will make tagging the paragraphs easier.

22b ⟨*body of convert body* 20⟩+≡
```
$b =~ s/^ +/\n$&/mg;
    # strip out the line-starting blanks, since
    #  they're now redundant
$b =~ s/^ +//mg;
```

Block out the basic paragraphs. Add an open paragraph tag at the beginning of each paragraph, including the one at the very beginning of the text block. That funny negative lookahead assertion, `(?!<p)` prevents a paragraph tag from being inserted in a paragraph that already has one.

In the next line, we add the close paragraph tag, and finally add one at the very bottom of the text block.

22c ⟨*body of convert body* 20⟩+≡
```
$b =~ s/(^|\n\n+(?!<p))/$&<p class="tx">/sg;
$b =~ s!\n\n+!</p>\n!sg;
$b .= "</p>\n";
```

We then have to do one more little bit of fixup: we have some cases where we've got multiple opening `<p>` tags, in particular, instances of `<p class="tx"><p>`, artifacts of inserting image tags.

22d ⟨*body of convert body* 20⟩+≡
```
$b =~ s/(<p.*?>)<p>/$1/sg;
```

Last, we want to handle some basic markup, underlines for italic and asterisks for bold. There are two things we need to be careful about: First, we sometimes we use strings of those characters for section separators. So, we convert paragraphs containing only those to a special section break. Second, we need to take care to not cross italic and bold markup over paragraph breaks, since the resulting markup has to be properly nested.

After that, we return.

22e ⟨*body of convert body* 20⟩+≡
```
$b =~ s!<p class="tx">([* ]+|[_ ]+)</p>!
        <p class="secbrk">*****</p>!sg;
$b =~ s!_([^<_]+?)_!<i>$1</i>!sg;
$b =~ s!\*([^<*]+?)\*!<b>$1</b>!sg;
$b;
```

The prolog for the individual file we've converted needs to provide some boilerplate such as a pointer to the CSS style sheet (which we'll define in a moment) and the title.

23a ⟨*utilities to convert the text files* 15a⟩+≡

```
sub create_prolog($) {
    my $t = shift;
"<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"no\"?>
<html xmlns=\"http://www.w3.org/1999/xhtml\">
<head>
<title>$t</title>
<link href=\"$css\" rel=\"stylesheet\" type=\"text/css\" />
</head>
";
}
```

Defines:
`create_prolog`, used in chunk 16a.
Uses `$css` 36a.

In our markup, we've referred to some styles that we'll now need to define in a cascading style sheet. For simplicity, we define a constant set of CSS markup across all books, even though some of this might not be used in every book. This routine returns the name of the CSS file after adding it to the list of control files.

23b ⟨*utilities to convert the text files* 15a⟩+≡

```
sub create_CSS() {
    my $c = "news2epub.css";
    unless (open F, "> $wd/$c") {
        die "can't create $c file";
    }
    print F <<"EOF";
⟨CSS file contents 23c⟩
EOF
    close F;
    push @control, $c;
    return $c;
}
```

Defines:
`create_CSS`, used in chunk 35a.
Uses `@control` 6 and `$wd` 6.

The individual styles are broken out, with a bit of explanation in the next few sections. We begin with the style for the basic HTML paragraph object. This includes spacing, font size, and widow/orphan prevention.

23c ⟨*CSS file contents* 23c⟩≡

```
p { display: block;  text-indent: 0em;  text-align: left;
    margin-top: 0em;  margin-bottom: 0em;  font-size: .9em;
    widows: 2; orphans: 2 }
```

Next, we want some additional styles for our text layout. We have a style for a standard indented paragraph, and one for a paragraph with block indentation. We'll also provide a section break style for centering little separators with a little extra space above and below. Lastly, we provide a little bit of future-proofing by providing centered text markup.

24a ⟨*CSS file contents* 23c⟩+≡

```
.tx     { text-indent: 2em; }
.txb    { margin-left: 4em;  margin-top: .3em;
            margin-bottom: .3em; }
.secbrk { text-align: center;  margin-top: .5em;
            margin-bottom: .5em; }
.txcb   { text-align: center;  font-weight: bold;
            margin-top: .5em; }
.txc    { text-align: center; }
```

Finally, we want a few styles for the header lines of e-mail messages we have converted.

Notice that our style for "Subject:" lines takes a page break before. According to the CSS definitions, this is redundant, since there is an implied page break at the beginning of each component file, but this makes it explicit. For text files that are not e-mail messages, we can substitute the "textfile" style, which carries the same page break instruction.

24b ⟨*CSS file contents* 23c⟩+≡

```
.subj   { font-weight: bold;  font-size: 125%;
            page-break-before: always;  text-indent: 0em; }
.from   { font-style: italic;  text-indent: 0em; }
.date   { font-style: italic;  text-indent: 0em; }
.textfile { page-break-before: always; }
```

Our last action for the content is to create a table of contents. The resulting TOC should be the first file in `@content`.

25a ⟨*utilities to convert the text files* 15a⟩+≡

```
sub create_toc()
{
    my $toc = "contents.html";
    open F, "> $wd/$toc";
    print F <<"EOF";
<?xml version="1.0" encoding="UTF-8"
    standalone="no"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Table of Contents</title>
<link href="news2epub.css" rel="stylesheet" type="text/css" />
</head>
<body>
<p class="tochdr">Table of Contents</p>
EOF
    foreach (0..$#content) {
        print F '<p class="toc"><a href="', $content[$_], '">',
            $titles[$_], "</a></p>\n";
    }
    print F "</body></html>\n";
    close F;
    unshift @content, $toc;
    unshift @titles, "Table of Contents";
    return $toc;
}
```

Defines:
`create_toc`, used in chunk 35b.
Uses `$content` 6, `@content` 6, `$titles` 14b, `@titles` 14b, `$toc` 36a, and `$wd` 6.

We've referred to a couple of additional styles for our CSS style sheet, and we should add them now. For example, we want a special hanging-indent text style for entries in our table of contents. And, yes, the rendering for `tochdr` is identical to `subj`, but they represent different structural elements, so should have different structural markup.

25b ⟨*CSS file contents* 23c⟩+≡

```
.toc    { padding-left: 2em;  text-indent: -2em;
            margin-bottom: .3em; }
.tochdr { font-weight: bold;  font-size: 125%;
            page-break-before: always;  text-indent: 0em; }
```

Basic utility routines

We've used a number of utility routines through the preceeding sections. We should probably define them.

Let's begin with `dirname`, which returns the part before the last virgule in the path name.

26a ⟨*utilities* 26a⟩≡

```
sub dirname($) {
    my $x = shift;
    return ""  unless ($x =~ m#.*/#);
    $x =~ s!/[^/]*$!!;
    return $x;
}
```

Defines:
`dirname`, used in chunks 5b and 26a.

And, the companion, `basename`.

26b ⟨*utilities* 26a⟩+≡

```
sub basename($) {
    my $x = shift;
    $x =~ s!.*/!!;
    return $x;
}
```

Defines:
`basename`, used in chunks 4, 7, 9, 11a, 13, 15b, 26, 32, 34a, and 35a.

And a similar routine to strip the extension off the end of a file, `filename`.

26c ⟨*utilities* 26a⟩+≡

```
sub filename($) {
    my $x = shift;
    $x =~ s!\..*?$!!;
    return $x;
}
```

Defines:
`filename`, used in chunks 7, 11a, 13, 26, 32, and 34a.

And the inverse of `filename` to capture the extension alone, ignoring possible directory names.

26d ⟨*utilities* 26a⟩+≡

```
sub extension($) {
    my $x = shift;
    $x = basename($x);
    $x =~ s!.*?\.!.!;
    return $x;
}
```

Defines:
`extension`, used in chunks 21a and 33.
Uses `basename` 26b.

Another general-purpose utility we use is the equivalent of the command-line "mkdir -p", which ensures the parents of a directory we are creating are all created.

27a ⟨*utilities* 26a⟩+≡

```
sub mkdir_p($) {
    my $d = shift;
    my $dd = "";
    foreach (split /\//, $d) {
        $dd .= "$_/";
        next   if ( -d $dd );
        unless (mkdir $dd, 0777) {
            die "can't create directory $dd";
        }
    }
}
```

Defines:
`mkdir_p`, used in chunks 14a and 32.

We need to convert text into UTF-8. In general, text will come to us in ISO-8859-1 — Latin-1 — or that special Latin-1 variant, Windows 1252. However, we need a routine that can choose among several options, based on the character set we gleaned from the message header. We use a hash that contains references to the mapping tables for each known character set, (If we're already in UTF-8, we short-circuit our return because there's no mapping table for UTF-8.)

27b ⟨*utilities* 26a⟩+≡

```
sub text_to_UTF8($) {
    my $text = shift;
    return $text  if ($mapping{$charset} eq "");

    my $table = $mapping{$charset};
    $text =~ s/[\x80-\xFF]/
                UTF_to_UTF8($table->[ord($&)-0x80]) /eg;
    $text;
}
```

Defines:
`text_to_UTF8`, used in chunks 18, 20, and 32.
Uses `UTF_to_UTF8` 30a.

Here's the hash we referred to in the last section.

28a ⟨*charset conversion hash* 28a⟩≡

```
my %mapping = (
    "utf-8" => "",
    "iso-8859-1" => \@windows_1252,
    "iso-8859-15" => \@iso_8859_15,
    "us-ascii" => \@windows_1252,
    "windows-1252" => \@windows_1252,
);
```

Uses `@iso_8859_15` 29 and `@windows_1252` 28b.

Now we have the mapping tables for the character sets. Each of these is composed from conversion tables on the Unicode web site.

First, the Windows code page 1252 table, of which 8859-1 is a proper subset.

28b ⟨*charset conversion tables* 28b⟩≡

```
my @windows_1252 = (
        # 0x80 .. 0x87
    0x20AC, 0, 0x201A, 0x0192, 0x201E, 0x2026, 0x2020, 0x2021,
        # 0x88 .. 0x8F
    0x02C6, 0x2030, 0x0160, 0x2039, 0x0152, 0, 0x017D, 0,
        # 0x90 .. 0x97
    0, 0x2018, 0x2019, 0x201C, 0x201D, 0x2022, 0x2013, 0x2014,
        # 0x98 .. 0x9F
    0x02DC, 0x2122, 0x0161, 0x203A, 0x0153, 0, 0x017E, 0x0178,
        # 0xA0 .. 0xFF
    0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6, 0xA7,
    0xA8, 0xA9, 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF,
    0xB0, 0xB1, 0xB2, 0xB3, 0xB4, 0xB5, 0xB6, 0xB7,
    0xB8, 0xB9, 0xBA, 0xBB, 0xBC, 0xBD, 0xBE, 0xBF,
    0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7,
    0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF,
    0xD0, 0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7,
    0xD8, 0xD9, 0xDA, 0xDB, 0xDC, 0xDD, 0xDE, 0xDF,
    0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7,
    0xE8, 0xE9, 0xEA, 0xEB, 0xEC, 0xED, 0xEE, 0xEF,
    0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
    0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF,
);
```

Defines:

`$windows_1252`, never used.
`@windows_1252`, used in chunk 28a.

We rarely have input in character sets other than 8859-1 and UTF-8, but to prove the concept of the table-driven translation, we'll provide a conversion for ISO-8859-15, which is very closely related to 8859-1, differing only in the eight positions 0xA4, 0xA6, 0xA8, 0xB4, 0xB8, 0xBC, 0xBD, 0xBE. However, we remove the high control characters, since the positions from 0x80 to 0x9F won't have to do double duty for their Windows values as in the 1252/Latin-1 table.

29 ⟨*charset conversion tables* 28b⟩+≡

```
my @iso_8859_15 = (
        # 0x80 .. 0x8F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        # 0x90 .. 0x9F
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        # 0xA0 .. 0xFF
    0xA0, 0xA1, 0xA2, 0xA3, 0x20AC, 0xA5, 0x160, 0xA7,
    0x161, 0xA9, 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF,
    0xB0, 0xB1, 0xB2, 0x17D, 0xB4, 0xB5, 0xB6, 0xB7,
    0x17E, 0xB9, 0xBA, 0xBB, 0x152, 0x153, 0x178, 0xBF,
    0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7,
    0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF,
    0xD0, 0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7,
    0xD8, 0xD9, 0xDA, 0xDB, 0xDC, 0xDD, 0xDE, 0xDF,
    0xE0, 0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7,
    0xE8, 0xE9, 0xEA, 0xEB, 0xEC, 0xED, 0xEE, 0xEF,
    0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
    0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF,
);
```

Defines:
`$iso_8859_15`, never used.
`@iso_8859_15`, used in chunk 28a.

This next little utility is not really necessary. We could have obviated it (and made our code more efficient) by populating the conversion tables with the UTF-8 versions of the characters. However, by using the UTF code points instead, the tables is clearer,[3] and this conversion is not computationally expensive. If there is no mapping provided, we return `?`, though it could be argued we should return no character.

30a ⟨*utilities* 26a⟩+≡

```
sub UTF_to_UTF8($) {
    my $x = shift;
    if ($x == 0x0) {
        return '?';
    } elsif ($x <= 0x007F) {
        return $x;
    } elsif ($x <= 0x07FF) {
        my $h = (0xC0 | (($x & 0x07C0) >> 6));
        my $l = (0x80 | ($x & 0x003F));
        return chr($h) . chr($l);
    } elsif ($x <= 0xFFFF) {
        my $h = (0xE0 | (($x & 0xF000) >> 12));
        my $m = (0x80 | (($x & 0x0FC0) >> 6));
        my $l = (0x80 | ($x & 0x003F));
        return chr($h) . chr($m) . chr($l);
    } else {
        return "XXXX";
    }
}
```

Defines:
`UTF_to_UTF8`, used in chunk 27b.

We'd also like a common routine to fixup some common ASCII characters that trip up our HTML rendering:

30b ⟨*utilities* 26a⟩+≡

```
sub escape_html($) {
    my $x = shift;
    $x =~ s/&/&amp;/g;   $x =~ s/</&lt;/g;   $x =~ s/>/&gt;/g;
    return $x;
}
```

[3] "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do. — Donald E Knuth, *Literate Programming*, 1984

Lastly, we need a routine to determine image file type. This is required for the `media-type` tag in the manifest entry for the cover image. We do this by checking the first few bytes of the file for magic character sequences. This is a low-rent, specialized version of the Unix *file* command.

31a ⟨*utilities* 26a⟩+≡

```
sub image_type($) {
    my $image = shift;
    my $line;
    warn "can't open $image to check type\n"
        unless (open I, "< $image");
    sysread I, $line, 32;
    close I;
    my ($m1, $m2, $m3, $m4) = unpack("a2a2a2a4", $line);
    return "jpeg"
        if ($m1 =~ m/\xff\xd8/
            &&  ($m4 =~ m/JFIF/ || $m4 =~ m/Exif/));
    return "gif"
        if ($m1 =~ m/GI/  &&  $m2 =~ m/F8/);
    return "png"
        if ($m1 =~ m/\x89P/  &&  $m2 =~ m/NG/);
    return "???";
}
```

Defines:
`image_type`, used in chunk 9.

We've been dropping other utilities all along the way. Let's collect the rest of them:

31b ⟨*utilities* 26a⟩+≡

```
⟨utilities to convert the text files 15a⟩
⟨utilities to assemble the EPUB 4⟩
```

The main program

Of course, we have to use all this code in an organized fashion. Our main program is not very complicated now that we've posited all the components of conversion and construction.

We begin with some initialization, in which we set some defaults, parse the command line options, and set up a working directory.

32 ⟨*the main program* 32⟩≡

```
getopts("A:C:E:KT:i:ktv") ||
    die "usage: $0 [-A author] [-C cover_image] [-T title] "
        . "[-E epub] [-i dir] [-Kktv] files...";
$epub = ($opt_E) ? $opt_E : "zdefault.epub";
$epub .= ".epub"   unless ($epub =~ m/\.epub$/);
$charset = $default_charset;
$book_title = ($opt_T)
    ? escape_html(text_to_UTF8($opt_T))
    : "what the heck?";
$book_author = ($opt_A)
    ? escape_html(text_to_UTF8($opt_A))
    : "who's on first";
        # working directory
$wd = "," . filename(basename($epub)) . $suffix;
⟨clean up the previous attempt 34a⟩
mkdir_p $wd;
```

Uses basename 26b, $book_author 34b, $book_title 34b, $epub 34b, filename 26c, mkdir_p 27a, $opt_A 37a, $opt_E 37a, $opt_T 37a, $suffix 6, text_to_UTF8 27b, and $wd 6.

We've can now finish out the manual page with descriptions of the command-line flags.

33 ⟨*POD* 3⟩+≡

```
=head1 OPTIONS

=over

=item B<-A>I<author>

Provide the author of the book.

=item B<-C>I<file>

Provide the image for the book cover.

=item B<-E>I<epub_file>

Give the output name of the EPUB file.
The I<.epub> extension is supplied by default.

=item B<-K>

Assume we want to create a Kindle book.
Don't create the final EPUB file,
because we'll run Mobipocket Creator
on the working directory.
Implies I<-k>.

=item B<-T>I<title>

Provide the title for the final book.

=item B<-i>I<dir>

Search for in-line images in I<dir> if they aren't in the current
directory.
(The cover image path is explicit in the argument to the B<-C> flag.)

=item B<-k>

Don't delete the working directory.

=item B<-t>
```

```
Don't include the table of contents.

=item B<-v>

Provide verbose output.

=back
```

Uses `extension` 26d.

We may have a previous attempt to build this book, in which case we want to sequester the results of that effort so we can start fresh. Move the EPUB output file into the working directory if we can, and then rename the working directory to something distinguishable.

34a ⟨*clean up the previous attempt* 34a⟩≡

```
my $alt = filename(basename($epub));
while (-s "$alt.epub" || -d ",$alt$suffix") { $alt .= "~"; }
qx(mv $epub $wd)        if (-s $epub && -d $wd);
qx(mv $epub $alt.epub)  if (-s $epub && ! -d $wd);
warn "$wd moved to ,$alt$suffix\n"
                            if (-d $wd);
rename $wd, ",$alt$suffix"  if (-d $wd);
```

Uses `basename` 26b, `$epub` 34b, `filename` 26c, `$suffix` 6, and `$wd` 6.

We've introduced some global variables.

34b ⟨*declare global variables* 6⟩+≡

```
my $epub;
my $book_title;
my $book_author;
```

Defines:
`$book_author`, used in chunks 8a, 11b, and 32.
`$book_title`, used in chunks 8a, 11b, and 32.
`$epub`, used in chunks 4, 7, 11a, 13, 32, and 34a.

Then, we process and convert the input files. We begin by creating the CSS style sheet, and then loop through the input files and process each one.

35a ⟨*the main program* 32⟩+≡

```
$css = create_CSS();
foreach (@ARGV) {
        # default name & subject
    my ($o, $s) = (basename($_) . ".html", "foo");
    ($o, $s) = convert_file($_);
    warn length($o) ?
        ":: $_ -> $o\n" : "?? $_ not found\n"
            if ($opt_v);
    next  unless (length($o));
    push @content, $o;
    push @titles, $s;
}
```

Uses basename 26b, @content 6, convert_file 15a, create_CSS 23b, $css 36a, $opt_v 37a, and @titles 14b.

We can follow that by creating the wrapper components of the EPUB file, beginning with preparing the table of contents. We use the current time as the unique book id; this should really be something calculated from the contents.

To do: create a UUID via something like the Perl UUID::Tiny module.

To do: When we generate UUIDs for the identifier, we should add the `scheme="uuid"` attribute.

35b ⟨*the main program* 32⟩+≡

```
$first_text = $content[0];
$toc = create_toc()  unless ($opt_t);
$cover = create_cover()  if ($opt_C);
$bookid = time();
$ncx = create_ncx();
$opf = create_opf();
create_container();
```

Uses $bookid 36a, $content 6, $cover 36a, create_container 14a, create_cover 13, create_ncx 11a, create_opf 7, create_toc 25a, $first_text 36a, $ncx 36a, $opf 36a, $opt_C 37a, $opt_t 37a, and $toc 36a.

... and, finally, assembling the pieces into the EPUB.

35c ⟨*the main program* 32⟩+≡

```
assemble_final_package()  unless ($opt_K);
```

Uses assemble_final_package 4.

We had some more globals in those two passages, which we'll add to our raft of declarations.

36a ⟨*declare global variables* 6⟩+≡

```
my $css;
my $first_text;
my $toc;
my $cover;
my $ncx;
my $opf;
my $bookid;
```

Defines:
$bookid, used in chunks 8a, 11b, and 35b.
$cover, used in chunks 10b, 13, and 35b.
$css, used in chunks 9, 23a, and 35a.
$first_text, used in chunks 10b and 35b.
$ncx, used in chunks 9, 11a, and 35b.
$opf, used in chunks 7, 14a, and 35b.
$toc, used in chunks 10b, 25a, and 35b.

We'll finish with some cleanup. If we haven't asked to save the working directory, delete everything we've put in it.

36b ⟨*the main program* 32⟩+≡

```
unless ($opt_k || $opt_K) {
    for (@content, @control) {
        unless (unlink "$wd/$_") {
            warn "can't unlink file $_ from $wd\n";
        }
    }
    for (@dirlist) {
        unless (rmdir "$wd/$_") {
            warn "can't unlink directory $_ from $wd\n";
        }
    }
    unless (rmdir $wd) {
        warn "can't unlink working directory $wd\n";
    }
}
```

Uses @content 6, @control 6, @dirlist 6, $opt_k 37a, and $wd 6.

Over the course of the main program, we needed to include some packages, so let's enumerate them now:

37a ⟨*include packages* 8b⟩+≡

```
use Getopt::Std;
use vars qw($opt_A $opt_C $opt_E $opt_T $opt_i $opt_k $opt_t $opt_v);
```

Defines:
`$opt_A`, used in chunk 32.
`$opt_C`, used in chunks 9, 10b, 13, and 35b.
`$opt_E`, used in chunk 32.
`$opt_T`, used in chunk 32.
`$opt_i`, used in chunk 21a.
`$opt_k`, used in chunk 36b.
`$opt_t`, used in chunks 10b and 35b.
`$opt_v`, used in chunks 4 and 35a.

Finally, to close out our construction, we put all these pieces together as a single Perl script:

37b ⟨* 37b⟩≡

```
#! /usr/bin/perl -w
# $Id: epub.nw,v 2.43 2017/05/02 23:21:58 jeff Exp $

use strict;
⟨include packages 8b⟩
⟨declare global variables 6⟩
⟨charset conversion tables 28b⟩
⟨charset conversion hash 28a⟩
⟨utilities 26a⟩
⟨the main program 32⟩
=pod
⟨POD 3⟩

=cut
```

Stuff to do

- Based on recommendations in the standard, we really need break up the content into separate directories — content, control files, images.
- Obvious follow-on: EPUB to TEX converter.
- Consistent section naming – noun/verb or function/verb.
- Consistent file naming – we have some fixed names, like the CSS, but also variables like cover name and OPF.

List of code chunks

⟨ * 37b⟩
⟨*CSS file contents* 23c⟩
⟨*NCX header and metadata* 11b⟩
⟨*NCX navigation map* 12⟩
⟨*OPF guide creation* 10b⟩
⟨*OPF manifest creation* 9⟩
⟨*OPF metadata* 8a⟩
⟨*OPF spine creation* 10a⟩
⟨*POD* 3⟩
⟨*body of convert body* 20⟩
⟨*body of convert file* 15b⟩
⟨*capture directory names* 5b⟩
⟨*charset conversion hash* 28a⟩
⟨*charset conversion tables* 28b⟩
⟨*clean up the previous attempt* 34a⟩
⟨*create mimetype file* 5a⟩
⟨*declare global variables* 6⟩
⟨*errant control characters* 17a⟩
⟨*get charset header* 19a⟩
⟨*include packages* 8b⟩
⟨*insurance for empty text body* 16b⟩
⟨*the main program* 32⟩
⟨*utilities* 26a⟩
⟨*utilities to assemble the EPUB* 4⟩
⟨*utilities to convert the text files* 15a⟩

Index