

```
1  /*
2  Copyright 2018 The pdfcpu Authors.
3
4  Licensed under the Apache License, Version 2.0 (the "License");
5  you may not use this file except in compliance with the License.
6  You may obtain a copy of the License at
7
8      http://www.apache.org/licenses/LICENSE-2.0
9
10 Unless required by applicable law or agreed to in writing, software
11 distributed under the License is distributed on an "AS IS" BASIS,
12 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 See the License for the specific language governing permissions and
14 limitations under the License.
15 */
16
17 package pdfcpu
18
19 import (
20     "bufio"
21     "bytes"
22     "io"
23     "os"
24     "sort"
25     "strconv"
26     "strings"
27
28     "github.com/pdfcpu/pdfcpu/pkg/filter"
29     "github.com/pdfcpu/pdfcpu/pkg/log"
30     "github.com/pkg/errors"
31 )
32
33 const (
34     defaultBufSize = 1024
35     //unknownDelimiter = byte(0)
36 )
37
38 // ReadFile reads in a PDF file and builds an internal structure holding its cross
39 // reference table aka the Context.
40 func ReadFile(inFile string, conf *Configuration) (*Context, error) {
41     log.Info.Printf("reading %s..\n", inFile)
42
43     f, err := os.Open(inFile)
44     if err != nil {
45         return nil, errors.Wrapf(err, "can't open %q", inFile)
46     }
47
48     defer func() {
49         f.Close()
50     }()
51
52     return Read(f, conf)
53 }
54
55 // Read takes a readSeeker and generates a Context,
56 // an in-memory representation containing a cross reference table.
57 func Read(rs io.ReadSeeker, conf *Configuration) (*Context, error) {
58
```

```

59     log.Read.Println("Read: begin")
60
61     ctx, err := NewContext(rs, conf)
62     if err != nil {
63         return nil, err
64     }
65
66     if ctx.Reader15 {
67         log.Info.Println("PDF Version 1.5 conforming reader")
68     } else {
69         log.Info.Println("PDF Version 1.4 conforming reader - no object streams or
xrefstreams allowed")
70     }
71
72     // Populate xRefTable.
73     if err = readXRefTable(ctx); err != nil {
74         return nil, errors.Wrap(err, "Read: xRefTable failed")
75     }
76
77     // Make all objects explicitly available (load into memory) in corresponding
xRefTable entries.
78     // Also decode any involved object streams.
79     if err = dereferenceXRefTable(ctx, conf); err != nil {
80         return nil, err
81     }
82
83     // Some PDFWriters write an incorrent Size into trailer.
84     if *ctx.XRefTable.Size < len(ctx.XRefTable.Table) {
85         *ctx.XRefTable.Size = len(ctx.XRefTable.Table)
86     }
87
88     log.Read.Println("Read: end")
89
90     return ctx, nil
91 }
92
93 // ScanLines is a split function for a Scanner that returns each line of
94 // text, stripped of any trailing end-of-line marker. The returned line may
95 // be empty. The end-of-line marker is one carriage return followed
96 // by one newline or one carriage return or one newline.
97 // The last non-empty line of input will be returned even if it has no newline.
98 func scanLines(data []byte, atEOF bool) (advance int, token []byte, err error) {
99
100     if atEOF && len(data) == 0 {
101         return 0, nil, nil
102     }
103
104     indCR := bytes.IndexByte(data, '\r')
105     indLF := bytes.IndexByte(data, '\n')
106
107     switch {
108
109     case indCR ≥ 0 && indLF ≥ 0:
110         if indCR < indLF {
111             if indLF == indCR+1 {
112                 // 0x0D0A
113                 return indLF + 1, data[0:indCR], nil
114             }
115             // 0x0D ... 0x0A

```

```
116         return indCR + 1, data[0:indCR], nil
117     }
118     // 0x0A ... 0x0D
119     return indLF + 1, data[0:indLF], nil
120
121     case indCR ≥ 0:
122         // We have a full carriage return terminated line.
123         return indCR + 1, data[0:indCR], nil
124
125     case indLF ≥ 0:
126         // We have a full newline-terminated line.
127         return indLF + 1, data[0:indLF], nil
128
129 }
130
131 // If we're at EOF, we have a final, non-terminated line. Return it.
132 if atEOF {
133     return len(data), data, nil
134 }
135
136 // Request more data.
137 return 0, nil, nil
138 }
139
140 func newPositionedReader(rs io.ReadSeeker, offset *int64) (*bufio.Reader, error) {
141
142     if _, err := rs.Seek(*offset, io.SeekStart); err != nil {
143         return nil, err
144     }
145
146     log.Read.Printf("newPositionedReader: positioned to offset: %d\n", *offset)
147
148     return bufio.NewReader(rs), nil
149 }
150
151 // Get the file offset of the last XRefSection.
152 // Go to end of file and search backwards for the first occurrence of startxref
153 // {offset} %%EOF
154 func offsetLastXRefSection(ctx *Context) (*int64, error) {
155     rs := ctx.Read.rs
156
157     var (
158         prevBuf, workBuf []byte
159         bufSize          int64 = 512
160         offset           int64
161     )
162
163     for i := 1; offset == 0; i++ {
164
165         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
166         if err != nil {
167             return nil, errors.New("pdfcpu: can't find last xref section")
168         }
169
170         log.Read.Printf("scanning for offsetLastXRefSection starting at %d\n", off)
171
172         curBuf := make([]byte, bufSize)
173     }
```

```
174     _, err = rs.Read(curBuf)
175     if err != nil {
176         return nil, err
177     }
178
179     workBuf = curBuf
180     if prevBuf != nil {
181         workBuf = append(curBuf, prevBuf ... )
182     }
183
184     j := strings.LastIndex(string(workBuf), "startxref")
185     if j == -1 {
186         prevBuf = curBuf
187         continue
188     }
189
190     p := workBuf[j+len("startxref"):]
191     posEOF := strings.Index(string(p), "%EOF")
192     if posEOF == -1 {
193         return nil, errors.New("pdfcpu: no matching %EOF for startxref")
194     }
195
196     p = p[:posEOF]
197     offset, err = strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
198     if err != nil {
199         return nil, errors.New("pdfcpu: corrupted last xref section")
200     }
201
202 }
203
204 log.Read.Printf("Offset last xrefsection: %d\n", offset)
205
206 return &offset, nil
207 }
208
209 // Read next subsection entry and generate corresponding xref table entry.
210 func parseXRefTableEntry(s *bufio.Scanner, xRefTable *XRefTable, objectNumber int)
error {
211
212     log.Read.Println("parseXRefTableEntry: begin")
213
214     line, err := scanLine(s)
215     if err != nil {
216         return err
217     }
218
219     if xRefTable.Exists(objectNumber) {
220         log.Read.Printf("parseXRefTableEntry: end - Skip entry %d - already
assigned\n", objectNumber)
221         return nil
222     }
223
224     fields := strings.Fields(line)
225     if len(fields) != 3 ||
226         len(fields[0]) != 10 || len(fields[1]) != 5 || len(fields[2]) != 1 {
227         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
header")
228     }
229
```

```
230 offset, err := strconv.ParseInt(fields[0], 10, 64)
231 if err != nil {
232     return err
233 }
234
235 generation, err := strconv.Atoi(fields[1])
236 if err != nil {
237     return err
238 }
239
240 entryType := fields[2]
241 if entryType != "f" && entryType != "n" {
242     return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
entry")
243 }
244
245 var xRefTableEntry XRefTableEntry
246
247 if entryType == "n" {
248     // in use object
249
250     log.Read.Printf("parseXRefTableEntry: Object #%d is in use at offset=%d,
generation=%d\n", objectNumber, offset, generation)
251
252     if offset == 0 {
253         log.Info.Printf("parseXRefTableEntry: Skip entry for in use object #%d
with offset 0\n", objectNumber)
254         return nil
255     }
256
257     xRefTableEntry =
258         XRefTableEntry{
259             Free:      false,
260             Offset:    &offset,
261             Generation: &generation}
262
263 } else {
264     // free object
265
266     log.Read.Printf("parseXRefTableEntry: Object #%d is unused, next free is
object#%d, generation=%d\n", objectNumber, offset, generation)
267
268     xRefTableEntry =
269         XRefTableEntry{
270             Free:      true,
271             Offset:    &offset,
272             Generation: &generation}
273
274 }
275
276 log.Read.Printf("parseXRefTableEntry: Insert new xref table entry for Object %d\n",
objectNumber)
277
278 xRefTable.Table[objectNumber] = &xRefTableEntry
279
280 log.Read.Println("parseXRefTableEntry: end")
281
282 return nil
```

```
285 }
286
287 // Process xRef table subsection and create corresponding xRef table entries.
288 func parseXRefTableSubSection(s *bufio.Scanner, xRefTable *XRefTable, fields []string)
error {
289
290     log.Read.Println("parseXRefTableSubSection: begin")
291
292     startObjNumber, err := strconv.Atoi(fields[0])
293     if err != nil {
294         return err
295     }
296
297     objCount, err := strconv.Atoi(fields[1])
298     if err != nil {
299         return err
300     }
301
302     log.Read.Printf("detected xref subsection, startObj=%d length=%d\n",
startObjNumber, objCount)
303
304     // Process all entries of this subsection into xRefTable entries.
305     for i := 0; i < objCount; i++ {
306         if err = parseXRefTableEntry(s, xRefTable, startObjNumber+i); err != nil {
307             return err
308         }
309     }
310
311     log.Read.Println("parseXRefTableSubSection: end")
312
313     return nil
314 }
315
316 // Parse compressed object.
317 func compressedObject(s string) (Object, error) {
318
319     log.Read.Println("compressedObject: begin")
320
321     o, err := parseObject(&s)
322     if err != nil {
323         return nil, err
324     }
325
326     d, ok := o.(Dict)
327     if !ok {
328         // return trivial Object: Integer, Array, etc.
329         log.Read.Println("compressedObject: end, any other than dict")
330         return o, nil
331     }
332
333     streamLength, streamLengthRef := d.Length()
334     if streamLength == nil && streamLengthRef == nil {
335         // return Dict
336         log.Read.Println("compressedObject: end, dict")
337         return d, nil
338     }
339
340     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
stored in an object stream")
}
```

```

341 }
342
343 // Parse all objects of an object stream and save them into objectStreamDict.ObjArray.
344 func parseObjectStream(osd *ObjectStreamDict) error {
345
346     log.Read.Printf("parseObjectStream begin: decoding %d objects.\n", osd.ObjCount)
347
348     decodedContent := osd.Content
349     prolog := decodedContent[:osd.FirstObjOffset]
350
351     objs := strings.Fields(string(prolog))
352     if len(objs)%2 > 0 {
353         return errors.New("pdfcpu: parseObjectStream: corrupt object stream dict")
354     }
355
356     // e.g., 10 0 11 25 = 2 Objects: #10 @ offset 0, #11 @ offset 25
357
358     var objArray Array
359
360     var offsetOld int
361
362     for i := 0; i < len(objs); i += 2 {
363
364         offset, err := strconv.Atoi(objs[i+1])
365         if err != nil {
366             return err
367         }
368
369         offset += osd.FirstObjOffset
370
371         if i > 0 {
372             dstr := string(decodedContent[offsetOld:offset])
373             log.Read.Printf("parseObjectStream: objString = %s\n", dstr)
374             o, err := compressedObject(dstr)
375             if err != nil {
376                 return err
377             }
378
379             log.Read.Printf("parseObjectStream: [%d] = obj %s:\n%s\n", i/2-1, objs[i-
380 2], o)
381             objArray = append(objArray, o)
382         }
383
384         if i == len(objs)-2 {
385             dstr := string(decodedContent[offset:])
386             log.Read.Printf("parseObjectStream: objString = %s\n", dstr)
387             o, err := compressedObject(dstr)
388             if err != nil {
389                 return err
390             }
391
392             log.Read.Printf("parseObjectStream: [%d] = obj %s:\n%s\n", i/2, objs[i],
393 o)
394             objArray = append(objArray, o)
395         }
396
397         offsetOld = offset
398     }
399 }

```

```

398     osd.ObjArray = objArray
399
400     log.Read.Println("parseObjectStream end")
401
402     return nil
403 }
404
405 // For each object embedded in this xRefStream create the corresponding xRef table
entry.
406 func extractXRefTableEntriesFromXRefStream(buf []byte, xsd *XRefStreamDict, ctx
    *Context) error {
407
408     log.Read.Printf("extractXRefTableEntriesFromXRefStream begin")
409
410     // Note:
411     // A value of zero for an element in the W array indicates that the corresponding
field shall not be present in the stream,
412     // and the default value shall be used, if there is one.
413     // If the first element is zero, the type field shall not be present, and shall
default to type 1.
414
415     i1 := xsd.W[0]
416     i2 := xsd.W[1]
417     i3 := xsd.W[2]
418
419     xrefEntryLen := i1 + i2 + i3
420     log.Read.Printf("extractXRefTableEntriesFromXRefStream: begin xrefEntryLen =
    %d\n", xrefEntryLen)
421
422     if len(buf)%xrefEntryLen > 0 {
423         return errors.New("pdfcpu: extractXRefTableEntriesFromXRefStream: corrupt
    xrefstream")
424     }
425
426     objCount := len(xsd.Objects)
427     log.Read.Printf("extractXRefTableEntriesFromXRefStream: objCount:%d %v\n",
    objCount, xsd.Objects)
428
429     log.Read.Printf("extractXRefTableEntriesFromXRefStream: len(buf):%d
    objCount*xrefEntryLen:%d\n", len(buf), objCount*xrefEntryLen)
430     if len(buf) < objCount*xrefEntryLen {
431         // Sometimes there is an additional xref entry not accounted for by "Index".
432         // We ignore such a entries and do not treat this as an error.
433         return errors.New("pdfcpu: extractXRefTableEntriesFromXRefStream: corrupt
    xrefstream")
434     }
435
436     j := 0
437
438     // bufToInt64 interprets the content of buf as an int64.
439     bufToInt64 := func(buf []byte) (i int64) {
440
441         for _, b := range buf {
442             i <<= 8
443             i |= int64(b)
444         }
445
446         return
447     }
448

```



```
449     for i := 0; i < len(buf) && j < len(xsd.Objects); i += xrefEntryLen {
450
451         objectNumber := xsd.Objects[j]
452
453         i2Start := i + i1
454         c2 := bufToInt64(buf[i2Start : i2Start+i2])
455         c3 := bufToInt64(buf[i2Start+i2 : i2Start+i2+i3])
456
457         var xRefTableEntry XRefTableEntry
458
459         switch buf[i] {
460
461             case 0x00:
462                 // free object
463                 log.Read.Printf("extractXRefTableEntriesFromXRefStream: Object #%d is
unused, next free is object#%d, generation=%d\n", objectNumber, c2, c3)
464                 g := int(c3)
465
466                 xRefTableEntry =
467                     XRefTableEntry{
468                         Free:      true,
469                         Compressed: false,
470                         Offset:    &c2,
471                         Generation: &g}
472
473             case 0x01:
474                 // in use object
475                 log.Read.Printf("extractXRefTableEntriesFromXRefStream: Object #%d is in
use at offset=%d, generation=%d\n", objectNumber, c2, c3)
476                 g := int(c3)
477
478                 xRefTableEntry =
479                     XRefTableEntry{
480                         Free:      false,
481                         Compressed: false,
482                         Offset:    &c2,
483                         Generation: &g}
484
485             case 0x02:
486                 // compressed object
487                 // generation always 0.
488                 log.Read.Printf("extractXRefTableEntriesFromXRefStream: Object #%d is
compressed at obj %5d[%d]\n", objectNumber, c2, c3)
489                 objNumberRef := int(c2)
490                 objIndex := int(c3)
491
492                 xRefTableEntry =
493                     XRefTableEntry{
494                         Free:      false,
495                         Compressed: true,
496                         ObjectStream: &objNumberRef,
497                         ObjectStreamInd: &objIndex}
498
499                 ctx.Read.ObjectStreams[objNumberRef] = true
500
501             }
502
503             if ctx.XRefTable.Exists(objectNumber){
504                 log.Read.Printf("extractXRefTableEntriesFromXRefStream: Skip entry %d -
```

```
already assigned\n", objectNumber)
505     } else {
506         ctx.Table[objectNumber] = &xRefTableEntry
507     }
508
509     j++
510 }
511
512 log.Read.Println("extractXRefTableEntriesFromXRefStream: end")
513
514 return nil
515 }
516
517 func xRefStreamDict(ctx *Context, o Object, objNr int, streamOffset int64)
518 (*XRefStreamDict, error) {
519     // must be Dict
520     d, ok := o.(Dict)
521     if !ok {
522         return nil, errors.New("pdfcpu: xRefStreamDict: no dict")
523     }
524
525     // Parse attributes for stream object.
526     streamLength, streamLengthObjNr := d.Length()
527     if streamLength == nil && streamLengthObjNr == nil {
528         return nil, errors.New("pdfcpu: xRefStreamDict: no \"Length\" entry")
529     }
530
531     filterPipeline, err := pdfFilterPipeline(ctx, d)
532     if err != nil {
533         return nil, err
534     }
535
536     // We have a stream object.
537     log.Read.Printf("xRefStreamDict: streamobject %#d\n", objNr)
538     sd := NewStreamDict(d, streamOffset, streamLength, streamLengthObjNr,
539 filterPipeline)
540
541     if _, err = loadEncodedStreamContent(ctx, &sd); err != nil {
542         return nil, err
543     }
544
545     // Decode xrefstream content
546     if err = saveDecodedStreamContent(nil, &sd, 0, 0, true); err != nil {
547         return nil, errors.Wrapf(err, "xRefStreamDict: cannot decode stream for
548 obj#:%d\n", objNr)
549     }
550
551     return parseXRefStreamDict(&sd)
552 }
553
554 // Parse xRef stream and setup xrefTable entries for all embedded objects and the xref
555 stream dict.
556 func parseXRefStream(rd io.Reader, offset *int64, ctx *Context) (prevOffset *int64,
557 err error) {
558     log.Read.Printf("parseXRefStream: begin at offset %d\n", *offset)
559
560     buf, endInd, streamInd, streamOffset, err := buffer(rd)
561     if err != nil {
```

```
559     return nil, err
560 }
561
562     log.Read.Printf("parseXRefStream: endInd=%[1]d(%[1]x) streamInd=%[2]d(%[2]x)\n",
endInd, streamInd)
563
564     line := string(buf)
565
566     // We expect a stream and therefore "stream" before "endobj" if "endobj" within
buffer.
567     // There is no guarantee that "endobj" is contained in this buffer for large
streams!
568     if streamInd < 0 || (endInd > 0 && endInd < streamInd) {
569         return nil, errors.New("pdfcpu: parseXRefStream: corrupt pdf file")
570     }
571
572     // Init object parse buf.
573     l := line[:streamInd]
574
575     objectNumber, generationNumber, err := parseObjectAttributes(&l)
576     if err != nil {
577         return nil, err
578     }
579
580     // parse this object
581     log.Read.Printf("parseXRefStream: xrefstm obj#:%d gen:%d\n", *objectNumber,
*generationNumber)
582     log.Read.Printf("parseXRefStream: dereferencing object %d\n", *objectNumber)
583     o, err := parseObject(&l)
584     if err != nil {
585         return nil, errors.Wrapf(err, "parseXRefStream: no object")
586     }
587
588     log.Read.Printf("parseXRefStream: we have an object: %s\n", o)
589
590     streamOffset += *offset
591     sd, err := xRefStreamDict(ctx, o, *objectNumber, streamOffset)
592     if err != nil {
593         return nil, err
594     }
595     // We have an xref stream object
596
597     err = parseTrailerInfo(sd.Dict, ctx.XRefTable)
598     if err != nil {
599         return nil, err
600     }
601
602     // Parse xRefStream and create xRefTable entries for embedded objects.
603     err = extractXRefTableEntriesFromXRefStream(sd.Content, sd, ctx)
604     if err != nil {
605         return nil, err
606     }
607
608     // Create xRefTableEntry for XRefStreamDict.
609     entry :=
610         XRefTableEntry{
611             Free:      false,
612             Offset:    offset,
613             Generation: generationNumber,
614             Object:    *sd}
```

```
615
616     log.Read.Printf("parseXRefStream: Insert new xRefTable entry for Object %d\n",
*objectNumber)
617
618     ctx.Table[*objectNumber] = &entry
619     ctx.Read.XRefStreams[*objectNumber] = true
620     prevOffset = sd.PreviousOffset
621
622     log.Read.Println("parseXRefStream: end")
623
624     return prevOffset, nil
625 }
626
627 // Parse an xRefStream for a hybrid PDF file.
628 func parseHybridXRefStream(offset *int64, ctx *Context) error {
629
630     log.Read.Println("parseHybridXRefStream: begin")
631
632     rd, err := newPositionedReader(ctx.Read.rs, offset)
633     if err != nil {
634         return err
635     }
636
637     _, err = parseXRefStream(rd, offset, ctx)
638     if err != nil {
639         return err
640     }
641
642     log.Read.Println("parseHybridXRefStream: end")
643
644     return nil
645 }
646
647 // Parse trailer dict and return any offset of a previous xref section.
648 func parseTrailerInfo(d Dict, xRefTable *XRefTable) error {
649
650     log.Read.Println("parseTrailerInfo begin")
651
652     if _, found := d.Find("Encrypt"); found {
653         encryptObjRef := d.IndirectRefEntry("Encrypt")
654         if encryptObjRef != nil {
655             xRefTable.Encrypt = encryptObjRef
656             log.Read.Printf("parseTrailerInfo: Encrypt object: %s\n",
*xRefTable.Encrypt)
657         }
658     }
659
660     if xRefTable.Size == nil {
661         size := d.Size()
662         if size == nil {
663             return errors.New("pdfcpu: parseTrailerInfo: missing entry \"Size\"")
664         }
665         // Not reliable!
666         // Patched after all read in.
667         xRefTable.Size = size
668     }
669
670     if xRefTable.Root == nil {
671         rootObjRef := d.IndirectRefEntry("Root")
```

```
672     if rootObjRef == nil {
673         return errors.New("pdfcpu: parseTrailerInfo: missing entry \"Root\"")
674     }
675     xRefTable.Root = rootObjRef
676     log.Read.Printf("parseTrailerInfo: Root object: %s\\n", *xRefTable.Root)
677 }
678
679 if xRefTable.Info == nil {
680     infoObjRef := d.IndirectRefEntry("Info")
681     if infoObjRef != nil {
682         xRefTable.Info = infoObjRef
683         log.Read.Printf("parseTrailerInfo: Info object: %s\\n", *xRefTable.Info)
684     }
685 }
686
687 if xRefTable.ID == nil {
688     idArray := d.ArrayEntry("ID")
689     if idArray != nil {
690         xRefTable.ID = idArray
691         log.Read.Printf("parseTrailerInfo: ID object: %s\\n", xRefTable.ID)
692     } else if xRefTable.Encrypt != nil {
693         return errors.New("pdfcpu: parseTrailerInfo: missing entry \"ID\"")
694     }
695 }
696
697 log.Read.Println("parseTrailerInfo end")
698
699 return nil
700 }
701
702 func parseTrailerDict(trailerDict Dict, ctx *Context) (*int64, error) {
703
704     log.Read.Println("parseTrailerDict begin")
705
706     xRefTable := ctx.XRefTable
707
708     err := parseTrailerInfo(trailerDict, xRefTable)
709     if err != nil {
710         return nil, err
711     }
712
713     if arr := trailerDict.ArrayEntry("AdditionalStreams"); arr != nil {
714         log.Read.Printf("parseTrailerInfo: found AdditionalStreams: %s\\n", arr)
715         a := Array{}
716         for _, value := range arr {
717             if indRef, ok := value.(IndirectRef); ok {
718                 a = append(a, indRef)
719             }
720         }
721         xRefTable.AdditionalStreams = &a
722     }
723
724     offset := trailerDict.Prev()
725     if offset != nil {
726         log.Read.Printf("parseTrailerDict: previous xref table section offset:%d\\n",
727 *offset)
728     }
729
730     offsetXRefStream := trailerDict.Int64Entry("XRefStm")
```

```

730     if offsetXRefStream == nil {
731         // No cross reference stream.
732         if !ctx.Reader15 && xRefTable.Version() ≥ V14 && !ctx.Read.Hybrid {
733             return nil, errors.Errorf("parseTrailerDict: PDF1.4 conformant reader:
found incompatible version: %s", xRefTable.VersionString())
734         }
735         log.Read.Println("parseTrailerDict end")
736         // continue to parse previous xref section, if there is any.
737         return offset, nil
738     }
739
740     // This file is using cross reference streams.
741
742     if !ctx.Read.Hybrid {
743         ctx.Read.Hybrid = true
744         ctx.Read.UsingXRefStreams = true
745     }
746
747     // 1.5 conformant readers process hidden objects contained
748     // in XRefStm before continuing to process any previous XRefSection.
749     // Previous XRefSection is expected to have free entries for hidden entries.
750     // May appear in XRefSections only.
751     if ctx.Reader15 {
752         if err := parseHybridXRefStream(offsetXRefStream, ctx); err ≠ nil {
753             return nil, err
754         }
755     }
756
757     log.Read.Println("parseTrailerDict end")
758
759     return offset, nil
760 }
761
762 func scanLineRaw(s *bufio.Scanner) (string, error) {
763     if ok := s.Scan(); !ok {
764         if s.Err() ≠ nil {
765             return "", s.Err()
766         }
767         return "", errors.New("pdfcpu: scanLineRaw: returning nothing")
768     }
769     return s.Text(), nil
770 }
771
772 func scanLine(s *bufio.Scanner) (s1 string, err error) {
773     for i := 0; i ≤ 1; i++ {
774         s1, err = scanLineRaw(s)
775         if err ≠ nil {
776             return "", err
777         }
778         if len(s1) > 0 {
779             break
780         }
781     }
782
783     // Remove comment.
784     i := strings.Index(s1, "%")
785     if i ≥ 0 {
786         s1 = s1[:i]
787     }

```

```
788
789     return s1, nil
790 }
791
792 func isDict(s string) (bool, error) {
793     o, err := parseObject(&s)
794     if err != nil {
795         return false, err
796     }
797     _, ok := o.(Dict)
798     return ok, nil
799 }
800
801 func scanTrailer(s *bufio.Scanner, line string) (string, error) {
802
803     var buf bytes.Buffer
804     var err error
805     var i, j, k int
806
807     log.Read.Printf("line: <%s>\n", line)
808
809     // Scan for dict start tag "<<".
810     for {
811         i = strings.Index(line, "<<")
812         if i ≥ 0 {
813             break
814         }
815         line, err = scanLine(s)
816         log.Read.Printf("line: <%s>\n", line)
817         if err != nil {
818             return "", err
819         }
820     }
821
822     line = line[i:]
823     buf.WriteString(line)
824     buf.WriteString(" ")
825     log.Read.Printf("scanTrailer dictBuf after start tag: <%s>\n", line)
826
827     // Scan for dict end tag ">>" but account for inner dicts.
828     line = line[2:]
829
830     for {
831
832         if len(line) == 0 {
833             line, err = scanLine(s)
834             if err != nil {
835                 return "", err
836             }
837             buf.WriteString(line)
838             buf.WriteString(" ")
839             log.Read.Printf("scanTrailer dictBuf next line: <%s>\n", line)
840         }
841
842         i = strings.Index(line, "<<")
843         if i < 0 {
844             // No <<
845             j = strings.Index(line, ">>")
846             if j ≥ 0 {
```

```

847         // Yes >>
848         if k == 0 {
849             // Check for dict
850             ok, err := isDict(buf.String())
851             if err == nil && ok {
852                 return buf.String(), nil
853             }
854         } else {
855             k--
856         }
857         line = line[j+2:]
858         continue
859     }
860     // No >>
861     line, err = scanLine(s)
862     if err != nil {
863         return "", err
864     }
865     buf.WriteString(line)
866     buf.WriteString(" ")
867     log.Read.Printf("scanTrailer dictBuf next line: <%s>\n", line)
868 } else {
869     // Yes <<
870     j = strings.Index(line, ">>")
871     if j < 0 {
872         // No >>
873         k++
874         line = line[i+2:]
875     } else {
876         // Yes >>
877         if i < j {
878             // handle <<
879             k++
880             line = line[i+2:]
881         } else {
882             // handle >>
883             if k == 0 {
884                 // Check for dict
885                 ok, err := isDict(buf.String())
886                 if err == nil && ok {
887                     return buf.String(), nil
888                 }
889             } else {
890                 k--
891             }
892             line = line[j+2:]
893         }
894     }
895 }
896 }
897 }
898
899 func processTrailer(ctx *Context, s *bufio.Scanner, line string) (*int64, error) {
900
901     var trailerString string
902
903     if line != "trailer" {
904         trailerString = line[7:]
905         log.Read.Printf("processTrailer: trailer leftover: <%s>\n", trailerString)

```



```
906     } else {
907         log.Read.Printf("line (len %d) <%s>\n", len(line), line)
908     }
909
910     trailerString, err := scanTrailer(s, trailerString)
911     if err != nil {
912         return nil, err
913     }
914
915     log.Read.Printf("processTrailer: trailerString: (len:%d) <%s>\n",
916 len(trailerString), trailerString)
917
918     o, err := parseObject(&trailerString)
919     if err != nil {
920         return nil, err
921     }
922
923     trailerDict, ok := o.(Dict)
924     if !ok {
925         return nil, errors.New("pdfcpu: processTrailer: corrupt trailer dict")
926     }
927
928     log.Read.Printf("processTrailer: trailerDict:\n%s\n", trailerDict)
929
930     return parseTrailerDict(trailerDict, ctx)
931 }
932
933 // Parse xRef section into corresponding number of xRef table entries.
934 func parseXRefSection(s *bufio.Scanner, ctx *Context) (*int64, error) {
935
936     log.Read.Println("parseXRefSection begin")
937
938     line, err := scanLine(s)
939     if err != nil {
940         return nil, err
941     }
942
943     log.Read.Printf("parseXRefSection: <%s>\n", line)
944
945     fields := strings.Fields(line)
946
947     // Process all sub sections of this xRef section.
948     for !strings.HasPrefix(line, "trailer") && len(fields) == 2 {
949
950         if err = parseXRefTableSubSection(s, ctx.XRefTable, fields); err != nil {
951             return nil, err
952         }
953
954         // trailer or another xref table subsection ?
955         if line, err = scanLine(s); err != nil {
956             return nil, err
957         }
958
959         // if empty line try next line for trailer
960         if len(line) == 0 {
961             if line, err = scanLine(s); err != nil {
962                 return nil, err
963             }
964         }
965     }
966 }
```

```
964     fields = strings.Fields(line)
965 }
966
967 log.Read.Println("parseXRefSection: All subsections read!")
968
969 if !strings.HasPrefix(line, "trailer") {
970     return nil, errors.Errorf("xrefsection: missing trailer dict, line = <%s>",
971 line)
972 }
973
974 log.Read.Println("parseXRefSection: parsing trailer dict..")
975
976 return processTrailer(ctx, s, line)
977 }
978
979 // Get version from first line of file.
980 // Beginning with PDF 1.4, the Version entry in the document's catalog dictionary
981 // (located via the Root entry in the file's trailer, as described in 7.5.5, "File
982 // Trailer"),
983 // if present, shall be used instead of the version specified in the Header.
984 // Save PDF Version from header to xRefTable.
985 // The header version comes as the first line of the file.
986 // eolCount is the number of characters used for eol (1 or 2).
987 func headerVersion(rs io.ReadSeeker) (v *Version, eolCount int, err error) {
988     log.Read.Println("headerVersion begin")
989
990     var errCorruptHeader = errors.New("pdfcpu: headerVersion: corrupt pdf stream - no
991 header version available")
992
993     // Get first line of file which holds the version of this PDFFile.
994     // We call this the header version.
995     if _, err = rs.Seek(0, io.SeekStart); err != nil {
996         return nil, 0, err
997     }
998
999     buf := make([]byte, 20)
1000     if _, err = rs.Read(buf); err != nil {
1001         return nil, 0, err
1002     }
1003
1004     s := string(buf)
1005     prefix := "%PDF-"
1006
1007     if len(s) < 8 || !strings.HasPrefix(s, prefix) {
1008         return nil, 0, errCorruptHeader
1009     }
1010
1011     pdfVersion, err := PDFVersion(s[len(prefix) : len(prefix)+3])
1012     if err != nil {
1013         return nil, 0, errors.Wrapf(err, "headerVersion: unknown PDF Header Version")
1014     }
1015
1016     s = s[8:]
1017     s = strings.TrimLeft(s, "\t\f ")
1018
1019     // Detect the used eol which should be 1 (0x00, 0x0D) or 2 chars (0x0D0A)long.
1020     // %PDF-1.x{whiteSpace}{eol}
```

```
1020     if s[0] == 0x0A {
1021         eolCount = 1
1022     } else if s[0] == 0x0D {
1023         eolCount = 1
1024         if s[9] == 0x0A {
1025             eolCount = 2
1026         }
1027     } else {
1028         return nil, 0, errCorruptHeader
1029     }
1030
1031     log.Read.Printf("headerVersion: end, found header version: %s\n", pdfVersion)
1032
1033     return &pdfVersion, eolCount, nil
1034 }
1035
1036 // bypassXrefSection is a hack for digesting corrupt xref sections.
1037 // It populates the xrefTable by reading in all indirect objects line by line
1038 // and works on the assumption of a single xref section - meaning no incremental
1039 // updates have been made.
1040 func bypassXrefSection(ctx *Context) error {
1041     var z int64
1042     g := FreeHeadGeneration
1043     ctx.Table[0] = &XRefTableEntry{
1044         Free:      true,
1045         Offset:     &z,
1046         Generation: &g
1047     }
1048     rs := ctx.Read.rs
1049     eolCount := ctx.Read.EolCount
1050     var off, offset int64
1051     rd, err := newPositionedReader(rs, &offset)
1052     if err != nil {
1053         return err
1054     }
1055     s := bufio.NewScanner(rd)
1056     s.Split(scanLines)
1057
1058     bb := []byte{}
1059     var (
1060         withinObj      bool
1061         withinXref     bool
1062         withinTrailer bool
1063     )
1064
1065     for {
1066         line, err := scanLineRaw(s)
1067         if err != nil {
1068             break
1069         }
1070         if withinXref {
1071             offset += int64(len(line) + eolCount)
1072             if withinTrailer {
1073                 bb = append(bb, ' ')
1074                 bb = append(bb, line...)
1075                 i := strings.Index(line, "startxref")
1076                 if i >= 0 {
```

```
1078         // Parse trailer.
1079         _, err = processTrailer(ctx, s, string(bb))
1080         return err
1081     }
1082     continue
1083 }
1084 // Ignore all until "trailer".
1085 i := strings.Index(line, "trailer")
1086 if i ≥ 0 {
1087     bb = append(bb, line...)
1088     withinTrailer = true
1089 }
1090 continue
1091 }
1092 i := strings.Index(line, "xref")
1093 if i ≥ 0 {
1094     offset += int64(len(line) + eolCount)
1095     withinXref = true
1096     continue
1097 }
1098 if !withinObj {
1099     i := strings.Index(line, "obj")
1100     if i ≥ 0 {
1101         withinObj = true
1102         off = offset
1103         bb = append(bb, line[i+3:]...)
1104     }
1105     offset += int64(len(line) + eolCount)
1106     continue
1107 }
1108
1109 // within obj
1110 offset += int64(len(line) + eolCount)
1111 bb = append(bb, ' ')
1112 bb = append(bb, line...)
1113 i = strings.Index(line, "endobj")
1114 if i ≥ 0 {
1115     l := string(bb)
1116     objNr, generation, err := parseObjectAttributes(&l)
1117     if err ≠ nil {
1118         return err
1119     }
1120     of := off
1121     ctx.Table[*objNr] = &XRefTableEntry{
1122         Free:      false,
1123         Offset:    &of,
1124         Generation: generation}
1125     bb = nil
1126     withinObj = false
1127 }
1128 }
1129 return nil
1130 }
1131
1132 // Build XRefTable by reading XRef streams or XRef sections.
1133 func buildXRefTableStartingAt(ctx *Context, offset *int64) error {
1134
1135     log.Read.Println("buildXRefTableStartingAt: begin")
1136 }
```

```
1137     rs := ctx.Read.rs
1138
1139     hv, eolCount, err := headerVersion(rs)
1140     if err != nil {
1141         return err
1142     }
1143
1144     ctx.HeaderVersion = hv
1145     ctx.Read.EolCount = eolCount
1146
1147     for offset != nil {
1148
1149         rd, err := newPositionedReader(rs, offset)
1150         if err != nil {
1151             return err
1152         }
1153
1154         s := bufio.NewScanner(rd)
1155         s.Split(scanLines)
1156
1157         line, err := scanLine(s)
1158         if err != nil {
1159             return err
1160         }
1161
1162         log.Read.Printf("line: <%s>\n", line)
1163
1164         if strings.TrimSpace(line) == "xref" {
1165             log.Read.Println("buildXRefTableStartingAt: found xref section")
1166             if offset, err = parseXRefSection(s, ctx); err != nil {
1167                 return err
1168             }
1169         } else {
1170
1171             log.Read.Println("buildXRefTableStartingAt: found xref stream")
1172             ctx.Read.UsingXRefStreams = true
1173             rd, err = newPositionedReader(rs, offset)
1174             if err != nil {
1175                 return err
1176             }
1177             if offset, err = parseXRefStream(rd, offset, ctx); err != nil {
1178                 log.Read.Printf("bypassXRefSection after %v\n", err)
1179                 // Try fix for corrupt single xref section.
1180                 return bypassXRefSection(ctx)
1181             }
1182         }
1183     }
1184
1185     log.Read.Println("buildXRefTableStartingAt: end")
1186
1187     return nil
1188 }
1189
1190 // Populate the cross reference table for this PDF file.
1191 // Goto offset of first xref table entry.
1192 // Can be "xref" or indirect object reference eg. "34 0 obj"
1193 // Keep digesting xref sections as long as there is a defined previous xref section
1194 // and build up the xref table along the way.
1195 func readXRefTable(ctx *Context) (err error) {
```

```
1196
1197     log.Read.Println("readXRefTable: begin")
1198
1199     offset, err := offsetLastXRefSection(ctx)
1200     if err != nil {
1201         return
1202     }
1203
1204     err = buildXRefTableStartingAt(ctx, offset)
1205     if err == io.EOF {
1206         return errors.Wrap(err, "readXRefTable: unexpected eof")
1207     }
1208     if err != nil {
1209         return
1210     }
1211
1212     // Log list of free objects (not the "free list").
1213     //log.Read.Printf("freelist: %v\n", ctx.FreeObjects)
1214
1215     // Ensure valid freelist of objects.
1216     err = ctx.EnsureValidFreeList()
1217     if err != nil {
1218         return
1219     }
1220
1221     log.Read.Println("readXRefTable: end")
1222
1223     return
1224 }
1225
1226 func growBufBy(buf []byte, size int, rd io.Reader) ([]byte, error) {
1227
1228     b := make([]byte, size)
1229
1230     _, err := rd.Read(b)
1231     if err != nil {
1232         return nil, err
1233     }
1234     //log.Read.Printf("growBufBy: Read %d bytes\n", n)
1235
1236     return append(buf, b...), nil
1237 }
1238
1239 func nextStreamOffset(line string, streamInd int) (off int) {
1240
1241     off = streamInd + len("stream")
1242
1243     // Skip optional blanks.
1244     // TODO Should be skip optional whitespace instead?
1245     for ; line[off] == 0x20; off++ {
1246     }
1247
1248     // Skip 0A eol.
1249     if line[off] == '\n' {
1250         off++
1251         return
1252     }
1253
1254     // Skip 0D eol.
```

```
1255     if line[off] == '\r' {
1256         off++
1257         // Skip 0D0A eol.
1258         if line[off] == '\n' {
1259             off++
1260         }
1261     }
1262
1263     return
1264 }
1265
1266 func lastStreamMarker(streamInd *int, endInd int, line string) {
1267
1268     if *streamInd > len(line)-len("stream") {
1269         // No space for another stream marker.
1270         *streamInd = -1
1271         return
1272     }
1273
1274     // We start searching after this stream marker.
1275     bufpos := *streamInd + len("stream")
1276
1277     // Search for next stream marker.
1278     i := strings.Index(line[bufpos:], "stream")
1279     if i < 0 {
1280         // No stream marker within line buffer.
1281         *streamInd = -1
1282         return
1283     }
1284
1285     // We found the next stream marker.
1286     *streamInd += len("stream") + i
1287
1288     if endInd > 0 && *streamInd > endInd {
1289         // We found a stream marker of another object
1290         *streamInd = -1
1291     }
1292 }
1293
1294 // Provide a PDF file buffer of sufficient size for parsing an object w/o stream.
1295 func buffer(rd io.Reader) (buf []byte, endInd int, streamInd int, streamOffset int64,
1296 err error) {
1297
1298     // process: # gen obj ... obj dict ... {stream ... data ... endstream} ... endobj
1299     //                                     streamInd           endInd
1300     //                                     -1 if absent           -1 if
1301     absent
1302
1303     //log.Read.Println("buffer: begin")
1304
1305     endInd, streamInd = -1, -1
1306
1307     for endInd < 0 && streamInd < 0 {
1308
1309         buf, err = growBufBy(buf, defaultBufSize, rd)
1310         if err != nil {
1311             return nil, 0, 0, 0, err
1312         }
1313     }
1314 }
```

```
1312
1313     line := string(buf)
1314     endInd = strings.Index(line, "endobj")
1315     streamInd = strings.Index(line, "stream")
1316
1317     if endInd > 0 && (streamInd < 0 || streamInd > endInd) {
1318         // No stream marker in buf detected.
1319         break
1320     }
1321
1322     // For very rare cases where "stream" also occurs within obj dict
1323     // we need to find the last "stream" marker before a possible end marker.
1324     for streamInd > 0 && !keywordStreamRightAfterEndOfDict(line, streamInd) {
1325         lastStreamMarker(&streamInd, endInd, line)
1326     }
1327
1328     log.Read.Printf("buffer: endInd=%d streamInd=%d\n", endInd, streamInd)
1329
1330     if streamInd > 0 {
1331
1332         // streamOffset ... the offset where the actual stream data begins.
1333         // is right after the eol after "stream".
1334
1335         slack := 10 // for optional whitespace + eol (max 2 chars)
1336         need := streamInd + len("stream") + slack
1337
1338         if len(line) < need {
1339
1340             // to prevent buffer overflow.
1341             buf, err = growBufBy(buf, need-len(line), rd)
1342             if err != nil {
1343                 return nil, 0, 0, 0, err
1344             }
1345
1346             line = string(buf)
1347         }
1348
1349         streamOffset = int64(nextStreamOffset(line, streamInd))
1350     }
1351 }
1352
1353 //log.Read.Printf("buffer: end, returned bufsize=%d streamOffset=%d\n", len(buf),
streamOffset)
1354
1355 return buf, endInd, streamInd, streamOffset, nil
1356 }
1357
1358 // return true if 'stream' follows end of dict: >>{whitespace}stream
1359 func keywordStreamRightAfterEndOfDict(buf string, streamInd int) bool {
1360
1361     //log.Read.Println("keywordStreamRightAfterEndOfDict: begin")
1362
1363     // Get a slice of the chunk right in front of 'stream'.
1364     b := buf[:streamInd]
1365
1366     // Look for last end of dict marker.
1367     eod := strings.LastIndex(b, ">>")
1368     if eod < 0 {
1369         // No end of dict in buf.
```



```

1370     return false
1371 }
1372
1373 // We found the last >>. Return true if after end of dict only whitespace.
1374 ok := strings.TrimSpace(b[eod:]) == ">>"
1375
1376 //log.Read.Printf("keywordStreamRightAfterEndOfDict: end, %v\n", ok)
1377
1378 return ok
1379 }
1380
1381 func buildFilterPipeline(ctx *Context, filterArray, decodeParamsArr Array, decodeParams
Object) ([]PDFFilter, error) {
1382
1383     var filterPipeline []PDFFilter
1384
1385     for i, f := range filterArray {
1386         filterName, ok := f.(Name)
1387         if !ok {
1388             return nil, errors.New("pdfcpu: buildFilterPipeline: filterArray elements
corrupt")
1389         }
1390         if decodeParams == nil || decodeParamsArr[i] == nil {
1391             filterPipeline = append(filterPipeline, PDFFilter{Name:
filterName.Value(), DecodeParams: nil})
1392             continue
1393         }
1394
1395         dict, ok := decodeParamsArr[i].(Dict)
1396         if !ok {
1397             indRef, ok := decodeParamsArr[i].(IndirectRef)
1398             if !ok {
1399                 return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s\n",
dict)
1400             }
1401             d, err := dereferencedDict(ctx, indRef.ObjectNumber.Value())
1402             if err != nil {
1403                 return nil, err
1404             }
1405             dict = d
1406         }
1407
1408         filterPipeline = append(filterPipeline, PDFFilter{Name: filterName.String(),
DecodeParams: dict})
1409     }
1410
1411     return filterPipeline, nil
1412 }
1413
1414 // Return the filter pipeline associated with this stream dict.
1415 func pdfFilterPipeline(ctx *Context, dict Dict) ([]PDFFilter, error) {
1416
1417     log.Read.Println("pdfFilterPipeline: begin")
1418
1419     var err error
1420
1421     o, found := dict.Find("Filter")
1422     if !found {
1423         // stream is not compressed.

```

```

1425     return nil, nil
1426 }
1427
1428 // compressed stream.
1429
1430 var filterPipeline []PDFFilter
1431
1432 if indRef, ok := o.(IndirectRef); ok {
1433     o, err = dereferencedObject(ctx, indRef.ObjectNumber.Value())
1434     if err != nil {
1435         return nil, err
1436     }
1437 }
1438
1439 //fmt.Printf("dereferenced filter obj: %s\n", obj)
1440
1441 if name, ok := o.(Name); ok {
1442     // single filter.
1443
1444     filterName := name.String()
1445
1446     o, found := dict.Find("DecodeParms")
1447     if !found {
1448         // w/o decode parameters.
1449         log.Read.Println("pdfFilterPipeline: end w/o decode parms")
1450         return append(filterPipeline, PDFFilter{Name: filterName, DecodeParms:
1451 nil}), nil
1452     }
1453
1454     d, ok := o.(Dict)
1455     if !ok {
1456         ir, ok := o.(IndirectRef)
1457         if !ok {
1458             return nil, errors.Errorf("pdfFilterPipeline: corrupt Dict: %s\n", o)
1459         }
1460         d, err = dereferencedDict(ctx, ir.ObjectNumber.Value())
1461         if err != nil {
1462             return nil, err
1463         }
1464     }
1465
1466     // with decode parameters.
1467     log.Read.Println("pdfFilterPipeline: end with decode parms")
1468     return append(filterPipeline, PDFFilter{Name: filterName, DecodeParms: d}),
1469 nil
1470 }
1471
1472 // filter pipeline.
1473
1474 // Array of filternames
1475 filterArray, ok := o.(Array)
1476 if !ok {
1477     return nil, errors.Errorf("pdfFilterPipeline: Expected filterArray corrupt, %v
1478 %T", o, o)
1479 }
1480
1481 // Optional array of decode parameter dicts.
1482 var decodeParmsArr Array

```

```

1481     decodeParms, found := dict.Find("DecodeParms")
1482     if found {
1483         decodeParmsArr, ok = decodeParms.(Array)
1484         if !ok {
1485             return nil, errors.New("pdfcpu: pdfFilterPipeline: expected decodeParms
array corrupt")
1486         }
1487     }
1488
1489     //fmt.Printf("decodeParmsArr: %s\n", decodeParmsArr)
1490
1491     filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParmsArr,
decodeParms)
1492
1493     log.Read.Println("pdfFilterPipeline: end")
1494
1495     return filterPipeline, err
1496 }
1497
1498 func streamDictForObject(ctx *Context, d Dict, objNr, streamInd int, streamOffset,
offset int64) (sd StreamDict, err error) {
1499
1500     streamLength, streamLengthRef := d.Length()
1501
1502     if streamInd ≤ 0 {
1503         return sd, errors.New("pdfcpu: streamDictForObject: stream object without
streamOffset")
1504     }
1505
1506     filterPipeline, err := pdfFilterPipeline(ctx, d)
1507     if err ≠ nil {
1508         return sd, err
1509     }
1510
1511     streamOffset += offset
1512
1513     // We have a stream object.
1514     sd = NewStreamDict(d, streamOffset, streamLength, streamLengthRef, filterPipeline)
1515
1516     log.Read.Printf("streamDictForObject: end, Streamobject %#d\n", objNr)
1517
1518     return sd, nil
1519 }
1520
1521 func dict(ctx *Context, d1 Dict, objNr, genNr, endInd, streamInd int) (d2 Dict, err
error) {
1522
1523     if ctx.EncKey ≠ nil {
1524         _, err := decryptDeepObject(d1, objNr, genNr, ctx.EncKey, ctx.AES4Strings,
ctx.E.R)
1525         if err ≠ nil {
1526             return nil, err
1527         }
1528     }
1529
1530     if endInd ≥ 0 && (streamInd < 0 || streamInd > endInd) {
1531         log.Read.Printf("dict: end, %#d\n", objNr)
1532         d2 = d1
1533     }
1534

```

```

1535     return d2, nil
1536 }
1537
1538 func object(ctx *Context, offset int64, objNr, genNr int) (o Object, endInd, streamInd
int, streamOffset int64, err error) {
1539
1540     var rd io.Reader
1541     rd, err = newPositionedReader(ctx.Read.rs, &offset)
1542     if err != nil {
1543         return nil, 0, 0, 0, err
1544     }
1545
1546     //log.Read.Printf("object: seeked to offset:%d\n", offset)
1547
1548     // process: # gen obj ... obj dict ... {stream ... data ... endstream} endobj
1549     //                               streamInd                               endInd
1550     //                               -1 if absent                           -1 if absent
1551     var buf []byte
1552     buf, endInd, streamInd, streamOffset, err = buffer(rd)
1553     if err != nil {
1554         return nil, 0, 0, 0, err
1555     }
1556
1557     //log.Read.Printf("streamInd:%d(0x) streamOffset:%d(0x) endInd:%d(0x)\n",
streamInd, streamInd, streamOffset, streamOffset, endInd, endInd)
1558     //log.Read.Printf("buflen=%d\n%s", len(buf), hex.Dump(buf))
1559
1560     line := string(buf)
1561
1562     var l string
1563
1564     if endInd < 0 { // && streamInd ≥ 0, streamdict
1565         // buf: # gen obj ... obj dict ... stream ... data
1566         // implies we detected no endobj and a stream starting at streamInd.
1567         // big stream, we parse object until "stream"
1568         log.Read.Println("object: big stream, we parse object until stream")
1569         l = line[:streamInd]
1570     } else if streamInd < 0 { // dict
1571         // buf: # gen obj ... obj dict ... endobj
1572         // implies we detected endobj and no stream.
1573         // small object w/o stream, parse until "endobj"
1574         log.Read.Println("object: small object w/o stream, parse until endobj")
1575         l = line[:endInd]
1576     } else if streamInd < endInd { // streamdict
1577         // buf: # gen obj ... obj dict ... stream ... data ... endstream endobj
1578         // implies we detected endobj and stream.
1579         // small stream within buffer, parse until "stream"
1580         log.Read.Println("object: small stream within buffer, parse until stream")
1581         l = line[:streamInd]
1582     } else { // dict
1583         // buf: # gen obj ... obj dict ... endobj # gen obj ... obj dict ... stream
1584         // small obj w/o stream, parse until "endobj"
1585         // stream in buf belongs to subsequent object.
1586         log.Read.Println("object: small obj w/o stream, parse until endobj")
1587         l = line[:endInd]
1588     }
1589
1590     // Parse object number and object generation.
1591     var objectNr, generationNr *int

```

```
1592 objectNr, generationNr, err = parseObjectAttributes(&l)
1593 if err != nil {
1594     return nil, 0, 0, 0, err
1595 }
1596
1597 if objNr != *objectNr || genNr != *generationNr {
1598     return nil, 0, 0, 0, errors.Errorf("object: non matching objNr(%d) or
generationNumber(%d) tags found.", *objectNr, *generationNr)
1599 }
1600
1601 o, err = parseObject(&l)
1602
1603 return o, endInd, streamInd, streamOffset, err
1604 }
1605
1606 // ParseObject parses an object from file at given offset.
1607 func ParseObject(ctx *Context, offset int64, objNr, genNr int) (Object, error) {
1608
1609     log.Read.Printf("ParseObject: begin, obj#%d, offset:%d\n", objNr, offset)
1610
1611     obj, endInd, streamInd, streamOffset, err := object(ctx, offset, objNr, genNr)
1612     if err != nil {
1613         return nil, err
1614     }
1615
1616     switch o := obj.(type) {
1617
1618     case Dict:
1619         d, err := dict(ctx, o, objNr, genNr, endInd, streamInd)
1620         if err != nil || d != nil {
1621             // Dict
1622             return d, err
1623         }
1624         // StreamDict.
1625         return streamDictForObject(ctx, o, objNr, streamInd, streamOffset, offset)
1626
1627     case Array:
1628         if ctx.EncKey != nil {
1629             if _, err = decryptDeepObject(o, objNr, genNr, ctx.EncKey,
ctx.AES4Strings, ctx.E.R); err != nil {
1630                 return nil, err
1631             }
1632         }
1633         return o, nil
1634
1635     case StringLiteral:
1636         if ctx.EncKey != nil {
1637             s1, err := decryptString(o.Value(), objNr, genNr, ctx.EncKey,
ctx.AES4Strings, ctx.E.R)
1638             if err != nil {
1639                 return nil, err
1640             }
1641             return StringLiteral(*s1), nil
1642         }
1643         return o, nil
1644
1645     case HexLiteral:
1646         if ctx.EncKey != nil {
1647             bb, err := decryptHexLiteral(o, objNr, genNr, ctx.EncKey, ctx.AES4Strings,
```

```
ctx.E.R)
1648     if err ≠ nil {
1649         return nil, err
1650     }
1651     return StringLiteral(string(bb)), nil
1652 }
1653 return o, nil
1654
1655 default:
1656     return o, nil
1657 }
1658 }
1659
1660 func dereferencedObject(ctx *Context, objectNumber int) (Object, error) {
1661
1662     entry, ok := ctx.Find(objectNumber)
1663     if !ok {
1664         return nil, errors.New("pdfcpu: dereferencedObject: unregistered object")
1665     }
1666
1667     if entry.Compressed {
1668         err := decompressXRefTableEntry(ctx.XRefTable, objectNumber, entry)
1669         if err ≠ nil {
1670             return nil, err
1671         }
1672     }
1673
1674     if entry.Object = nil {
1675
1676         log.Read.Printf("dereferencedObject: dereferencing object %d\n", objectNumber)
1677
1678         o, err := ParseObject(ctx, *entry.Offset, objectNumber, *entry.Generation)
1679         if err ≠ nil {
1680             return nil, errors.Wrapf(err, "dereferencedObject: problem dereferencing
object %d", objectNumber)
1681         }
1682
1683         if o = nil {
1684             return nil, errors.New("pdfcpu: dereferencedObject: object is nil")
1685         }
1686
1687         entry.Object = o
1688     }
1689
1690     return entry.Object, nil
1691 }
1692
1693 func dereferencedInteger(ctx *Context, objectNumber int) (*Integer, error) {
1694
1695     o, err := dereferencedObject(ctx, objectNumber)
1696     if err ≠ nil {
1697         return nil, err
1698     }
1699
1700     i, ok := o.(Integer)
1701     if !ok {
1702         return nil, errors.New("pdfcpu: dereferencedInteger: corrupt integer")
1703     }
1704 }
```

```
1705     return &i, nil
1706 }
1707
1708 func dereferencedDict(ctx *Context, objectNumber int) (Dict, error) {
1709
1710     o, err := dereferencedObject(ctx, objectNumber)
1711     if err != nil {
1712         return nil, err
1713     }
1714
1715     d, ok := o.(Dict)
1716     if !ok {
1717         return nil, errors.New("pdfcpu: dereferencedDict: corrupt dict")
1718     }
1719
1720     return d, nil
1721 }
1722
1723 // dereference a Integer object representing an int64 value.
1724 func int64Object(ctx *Context, objectNumber int) (*int64, error) {
1725
1726     log.Read.Printf("int64Object begin: %d\n", objectNumber)
1727
1728     i, err := dereferencedInteger(ctx, objectNumber)
1729     if err != nil {
1730         return nil, err
1731     }
1732
1733     i64 := int64(i.Value())
1734
1735     log.Read.Printf("int64Object end: %d\n", objectNumber)
1736
1737     return &i64, nil
1738 }
1739 }
1740
1741 // Reads and returns a file buffer with length = stream length using provided reader
1742 // positioned at offset.
1743 func readContentStream(rd io.Reader, streamLength int) ([]byte, error) {
1744
1745     log.Read.Printf("readContentStream: begin streamLength:%d\n", streamLength)
1746
1747     buf := make([]byte, streamLength)
1748
1749     for totalCount := 0; totalCount < streamLength; {
1750         count, err := rd.Read(buf[totalCount:])
1751         if err != nil {
1752             return nil, err
1753         }
1754         log.Read.Printf("readContentStream: count=%d, buflen=%d(%X)\n", count,
1755             len(buf), len(buf))
1756         totalCount += count
1757     }
1758
1759     log.Read.Printf("readContentStream: end\n")
1760
1761     return buf, nil
1762 }
```

```

1762 // LoadEncodedStreamContent loads the encoded stream content from file into
1763 // StreamDict.
1764 func loadEncodedStreamContent(ctx *Context, sd *StreamDict) ([]byte, error) {
1765     log.Read.Printf("LoadEncodedStreamContent: begin\n%v\n", sd)
1766
1767     var err error
1768
1769     // Return saved decoded content.
1770     if sd.Raw != nil {
1771         log.Read.Println("LoadEncodedStreamContent: end, already in memory.")
1772         return sd.Raw, nil
1773     }
1774
1775     // Read stream content encoded at offset with stream length.
1776
1777     // Dereference stream length if stream length is an indirect object.
1778     if sd.StreamLength == nil {
1779         if sd.StreamLengthObjNr == nil {
1780             return nil, errors.New("pdfcpu: loadEncodedStreamContent: missing
1781 streamLength")
1782         }
1783         // Get stream length from indirect object
1784         sd.StreamLength, err = int64Object(ctx, *sd.StreamLengthObjNr)
1785         if err != nil {
1786             return nil, err
1787         }
1788         log.Read.Printf("LoadEncodedStreamContent: new indirect streamLength:%d\n",
1789 *sd.StreamLength)
1790
1791         newOffset := sd.StreamOffset
1792         rd, err := newPositionedReader(ctx.Read.rs, &newOffset)
1793         if err != nil {
1794             return nil, err
1795         }
1796
1797         log.Read.Printf("LoadEncodedStreamContent: seeked to offset:%d\n", newOffset)
1798
1799         // Buffer stream contents.
1800         // Read content from disk.
1801         rawContent, err := readContentStream(rd, int(*sd.StreamLength))
1802         if err != nil {
1803             return nil, err
1804         }
1805
1806         //log.Read.Printf("rawContent buflen=%d(0x%x)\n%s", len(rawContent),
1807 len(rawContent), hex.Dump(rawContent))
1808
1809         // Save encoded content.
1810         sd.Raw = rawContent
1811
1812         log.Read.Printf("LoadEncodedStreamContent: end: len(streamDictRaw)=%d\n",
1813 len(sd.Raw))
1814
1815         // Return encoded content.
1816         return rawContent, nil
1817 }
1818 // Decodes the raw encoded stream content and saves it to streamDict.Content.

```



```

1817 func saveDecodedStreamContent(ctx *Context, sd *StreamDict, objNr, genNr int, decode
1818 bool) (err error) {
1819     log.Read.Printf("saveDecodedStreamContent: begin decode=%t\n", decode)
1820
1821     // If the "Identity" crypt filter is used we do not need to decrypt.
1822     if ctx != nil && ctx.EncKey != nil {
1823         if len(sd.FilterPipeline) == 1 && sd.FilterPipeline[0].Name == "Crypt" {
1824             sd.Content = sd.Raw
1825             return nil
1826         }
1827     }
1828
1829     // Special case: If the length of the encoded data is 0, we do not need to decode
1830     // anything.
1831     if len(sd.Raw) == 0 {
1832         sd.Content = sd.Raw
1833         return nil
1834     }
1835
1836     // ctx gets created after XRefStream parsing.
1837     // XRefStreams are not encrypted.
1838     if ctx != nil && ctx.EncKey != nil {
1839         sd.Raw, err = decryptStream(sd.Raw, objNr, genNr, ctx.EncKey, ctx.AES4Streams,
1840 ctx.E.R)
1841         if err != nil {
1842             return err
1843         }
1844         l := int64(len(sd.Raw))
1845         sd.StreamLength = &l
1846     }
1847
1848     if !decode {
1849         return nil
1850     }
1851
1852     // Actual decoding of content stream.
1853     err = decodeStream(sd)
1854     if err == filter.ErrUnsupportedFilter {
1855         err = nil
1856     }
1857     if err != nil {
1858         return err
1859     }
1860
1861     log.Read.Println("saveDecodedStreamContent: end")
1862
1863     return nil
1864 }
1865
1866 // Resolve compressed xRefTableEntry
1867 func decompressXRefTableEntry(xRefTable *XRefTable, objectNumber int, entry
1868 *XRefTableEntry) error {
1869     log.Read.Printf("decompressXRefTableEntry: compressed object %d at %d[%d]\n",
1870 objectNumber, *entry.ObjectStream, *entry.ObjectStreamInd)
1871
1872     // Resolve xRefTable entry of referenced object stream.
1873     objectStreamXRefTableEntry, ok := xRefTable.Find(*entry.ObjectStream)
1874     if !ok {

```

```
1872     return errors.Errorf("decompressXRefTableEntry: problem dereferencing object
1873     stream %d, no xref table entry", *entry.ObjectStream)
1874 }
1875 // Object of this entry has to be a ObjectStreamDict.
1876 sd, ok := objectStreamXRefTableEntry.Object.(ObjectStreamDict)
1877 if !ok {
1878     return errors.Errorf("decompressXRefTableEntry: problem dereferencing object
1879     stream %d, no object stream", *entry.ObjectStream)
1880 }
1881 // Get indexed object from ObjectStreamDict.
1882 o, err := sd.IndexedObject(*entry.ObjectStreamInd)
1883 if err != nil {
1884     return errors.Wrapf(err, "decompressXRefTableEntry: problem dereferencing
1885     object stream %d", *entry.ObjectStream)
1886 }
1887 // Save object to XRefTableEntry.
1888 g := 0
1889 entry.Object = o
1890 entry.Generation = &g
1891 entry.Compressed = false
1892
1893 log.Read.Printf("decompressXRefTableEntry: end, Obj %d[%d]:\n<%s>\n",
1894 *entry.ObjectStream, *entry.ObjectStreamInd, o)
1895 return nil
1896 }
1897
1898 // Log interesting stream content.
1899 func logStream(o Object) {
1900
1901     switch o := o.(type) {
1902
1903     case StreamDict:
1904
1905         if o.Content == nil {
1906             log.Read.Println("logStream: no stream content")
1907         }
1908
1909         if o.IsPageContent {
1910             //log.Read.Printf("content <%s>\n", StreamDict.Content)
1911         }
1912
1913     case ObjectStreamDict:
1914
1915         if o.Content == nil {
1916             log.Read.Println("logStream: no object stream content")
1917         } else {
1918             log.Read.Printf("logStream: objectStream content = %s\n", o.Content)
1919         }
1920
1921         if o.ObjArray == nil {
1922             log.Read.Println("logStream: no object stream obj arr")
1923         } else {
1924             log.Read.Printf("logStream: objectStream objArr = %s\n", o.ObjArray)
1925         }
1926
1927     default:
```

```
1928     log.Read.Println("logStream: no ObjectStreamDict")
1929
1930 }
1931
1932 }
1933
1934 // Decode all object streams so contained objects are ready to be used.
1935 func decodeObjectStreams(ctx *Context) error {
1936
1937     // Note:
1938     // Entry "Extends" intentionally left out.
1939     // No object stream collection validation necessary.
1940
1941     log.Read.Println("decodeObjectStreams: begin")
1942
1943     // Get sorted slice of object numbers.
1944     var keys []int
1945     for k := range ctx.Read.ObjectStreams {
1946         keys = append(keys, k)
1947     }
1948     sort.Ints(keys)
1949
1950     for _, objectNumber := range keys {
1951
1952         // Get XRefTableEntry.
1953         entry := ctx.XRefTable.Table[objectNumber]
1954         if entry == nil {
1955             return errors.Errorf("decodeObjectStream: missing entry for obj#%d\n",
objectNumber)
1956         }
1957
1958         log.Read.Printf("decodeObjectStreams: parsing object stream for obj#%d\n",
objectNumber)
1959
1960         // Parse object stream from file.
1961         o, err := ParseObject(ctx, *entry.Offset, objectNumber, *entry.Generation)
1962         if err != nil || o == nil {
1963             return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
1964         }
1965
1966         // Ensure StreamDict
1967         sd, ok := o.(StreamDict)
1968         if !ok {
1969             return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
1970         }
1971
1972         // Load encoded stream content to xRefTable.
1973         if _, err = loadEncodedStreamContent(ctx, &sd); err != nil {
1974             return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing
object stream %d", objectNumber)
1975         }
1976
1977         // Save decoded stream content to xRefTable.
1978         if err = saveDecodedStreamContent(ctx, &sd, objectNumber, *entry.Generation,
true); err != nil {
1979             log.Read.Printf("obj %d: %s", objectNumber, err)
1980             return err
1981         }
1982     }
```

```

1983 // Ensure decoded objectArray for object stream dicts.
1984 if !sd.IsObjStm() {
1985     return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
1986 }
1987
1988 // We have an object stream.
1989 log.Read.Printf("decodeObjectStreams: object stream %#d\n", objectNumber)
1990
1991 ctx.Read.UsingObjectStreams = true
1992
1993 // Create new object stream dict.
1994 osd, err := objectStreamDict(&sd)
1995 if err != nil {
1996     return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing
object stream %d", objectNumber)
1997 }
1998
1999 log.Read.Printf("decodeObjectStreams: decoding object stream %d:\n",
objectNumber)
2000
2001 // Parse all objects of this object stream and save them to
ObjectStreamDict.ObjArray.
2002 if err = parseObjectStream(osd); err != nil {
2003     return errors.Wrapf(err, "decodeObjectStreams: problem decoding object
stream %d\n", objectNumber)
2004 }
2005
2006 if osd.ObjArray == nil {
2007     return errors.Wrap(err, "decodeObjectStreams: objArray should be set!")
2008 }
2009
2010 log.Read.Printf("decodeObjectStreams: decoded object stream %d:\n",
objectNumber)
2011
2012 // Save object stream dict to xRefTableEntry.
2013 entry.Object = *osd
2014 }
2015
2016 log.Read.Println("decodeObjectStreams: end")
2017
2018 return nil
2019 }
2020
2021 func handleLinearizationParmDict(ctx *Context, obj Object, objNr int) error {
2022
2023     if ctx.Read.Linearized {
2024         // Linearization dict already processed.
2025         return nil
2026     }
2027
2028     // handle linearization parm dict.
2029     if d, ok := obj.(Dict); ok && d.IsLinearizationParmDict() {
2030
2031         ctx.Read.Linearized = true
2032         ctx.LinearizationObjs[objNr] = true
2033         log.Read.Printf("handleLinearizationParmDict: identified linearizationObj
#%d\n", objNr)
2034
2035         a := d.ArrayEntry("H")
2036

```

```
2037     if a == nil {
2038         return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - missing array entry H", objNr)
2039     }
2040
2041     if len(a) != 2 && len(a) != 4 {
2042         return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - corrupt array entry H, needs length 2 or 4", objNr)
2043     }
2044
2045     offset, ok := a[0].(Integer)
2046     if !ok {
2047         return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - corrupt array entry H, needs Integer values", objNr)
2048     }
2049
2050     offset64 := int64(offset.Value())
2051     ctx.OffsetPrimaryHintTable = &offset64
2052
2053     if len(a) == 4 {
2054
2055         offset, ok := a[2].(Integer)
2056         if !ok {
2057             return errors.Errorf("handleLinearizationParmDict: corrupt
linearization dict at obj:%d - corrupt array entry H, needs Integer values", objNr)
2058         }
2059
2060         offset64 := int64(offset.Value())
2061         ctx.OffsetOverflowHintTable = &offset64
2062     }
2063 }
2064
2065 return nil
2066 }
2067
2068 func loadStreamDict(ctx *Context, sd *StreamDict, objNr, genNr int) error {
2069
2070     var err error
2071
2072     // Load encoded stream content for stream dicts into xRefTable entry.
2073     if _, err = loadEncodedStreamContent(ctx, sd); err != nil {
2074         return errors.Wrapf(err, "dereferenceObject: problem dereferencing stream %d",
objNr)
2075     }
2076
2077     ctx.Read.BinaryTotalSize += *sd.StreamLength
2078
2079     // Decode stream content.
2080     err = saveDecodedStreamContent(ctx, sd, objNr, genNr, ctx.DecodeAllStreams)
2081
2082     return err
2083 }
2084
2085 func updateBinaryTotalSize(ctx *Context, o Object) {
2086
2087     switch o := o.(type) {
2088
2089     case StreamDict:
2090         ctx.Read.BinaryTotalSize += *o.StreamLength
2091     }
```

```

2092     case ObjectStreamDict:
2093         ctx.Read.BinaryTotalSize += *o.StreamLength
2094
2095     case XRefStreamDict:
2096         ctx.Read.BinaryTotalSize += *o.StreamLength
2097
2098     }
2099
2100 }
2101
2102 func dereferenceObject(ctx *Context, objNr int) error {
2103
2104     xRefTable := ctx.XRefTable
2105     xRefTableSize := len(xRefTable.Table)
2106
2107     log.Read.Printf("dereferenceObject: begin, dereferencing object %d\n", objNr)
2108
2109     entry := xRefTable.Table[objNr]
2110
2111     if entry.Free {
2112         log.Read.Printf("free object %d\n", objNr)
2113         return nil
2114     }
2115
2116     if entry.Compressed {
2117         err := decompressXRefTableEntry(xRefTable, objNr, entry)
2118         if err != nil {
2119             return err
2120         }
2121         //log.Read.Printf("dereferenceObject: decompressed entry,
Compressed=%v\n%s\n", entry.Compressed, entry.Object)
2122         return nil
2123     }
2124
2125     // entry is in use.
2126     log.Read.Printf("in use object %d\n", objNr)
2127
2128     if entry.Offset == nil || *entry.Offset == 0 {
2129         log.Read.Printf("dereferenceObject: already decompressed or used object w/o
offset → ignored")
2130         return nil
2131     }
2132
2133     o := entry.Object
2134
2135     // Already dereferenced stream dict.
2136     if o != nil {
2137         logStream(entry.Object)
2138         updateBinaryTotalSize(ctx, o)
2139         log.Read.Printf("handleCachedStreamDict: using cached object %d of
%d\n<%=s>\n", objNr, xRefTableSize, entry.Object)
2140         return nil
2141     }
2142
2143     // Dereference (load from disk into memory).
2144
2145     log.Read.Printf("dereferenceObject: dereferencing object %d\n", objNr)
2146
2147     // Parse object from file: anything goes dict, array, integer, float,

```

```

streamdicts...
2148     o, err := ParseObject(ctx, *entry.Offset, objNr, *entry.Generation)
2149     if err != nil {
2150         return errors.Wrapf(err, "dereferenceObject: problem dereferencing object %d",
objNr)
2151     }
2152
2153     entry.Object = o
2154
2155     // Linearization dicts are validated and recorded for stats only.
2156     err = handleLinearizationParmDict(ctx, o, objNr)
2157     if err != nil {
2158         return err
2159     }
2160
2161     // Handle stream dicts.
2162
2163     if _, ok := o.(ObjectStreamDict); ok {
2164         return errors.Errorf("dereferenceObject: object stream should already be
dereferenced at obj:%d", objNr)
2165     }
2166
2167     if _, ok := o.(XRefStreamDict); ok {
2168         return errors.Errorf("dereferenceObject: xref stream should already be
dereferenced at obj:%d", objNr)
2169     }
2170
2171     if sd, ok := o.(StreamDict); ok {
2172
2173         err = loadStreamDict(ctx, &sd, objNr, *entry.Generation)
2174         if err != nil {
2175             return err
2176         }
2177
2178         entry.Object = sd
2179     }
2180
2181     log.Read.Printf("dereferenceObject: end obj %d of %d\n<%s>\n", objNr,
xRefTableSize, entry.Object)
2182
2183     logStream(entry.Object)
2184
2185     return nil
2186 }
2187
2188 func processDictRefCounts(xRefTable *XRefTable, d Dict) {
2189     for _, e := range d {
2190         switch o1 := e.(type) {
2191             case IndirectRef:
2192                 entry, ok := xRefTable.FindTableEntryForIndRef(&o1)
2193                 if ok {
2194                     entry.RefCount++
2195                 }
2196             case Dict:
2197                 processRefCounts(xRefTable, o1)
2198             case Array:
2199                 processRefCounts(xRefTable, o1)
2200         }
2201     }
2202 }

```

```
2203
2204 func processArrayRefCounts(xRefTable *XRefTable, a Array) {
2205     for _, e := range a {
2206         switch o1 := e.(type) {
2207             case IndirectRef:
2208                 entry, ok := xRefTable.FindTableEntryForIndRef(&o1)
2209                 if ok {
2210                     entry.RefCount++
2211                 }
2212             case Dict:
2213                 processRefCounts(xRefTable, o1)
2214             case Array:
2215                 processRefCounts(xRefTable, o1)
2216         }
2217     }
2218 }
2219
2220 func processRefCounts(xRefTable *XRefTable, o Object) {
2221
2222     switch o := o.(type) {
2223     case Dict:
2224         processDictRefCounts(xRefTable, o)
2225
2226     case StreamDict:
2227         processDictRefCounts(xRefTable, o.Dict)
2228
2229     case Array:
2230         processArrayRefCounts(xRefTable, o)
2231     }
2232 }
2233
2234 // Dereferences all objects including compressed objects from object streams.
2235 func dereferenceObjects(ctx *Context) error {
2236
2237     log.Read.Println("dereferenceObjects: begin")
2238
2239     xRefTable := ctx.XRefTable
2240
2241     // Get sorted slice of object numbers.
2242     // TODO Skip sorting for performance gain.
2243     var keys []int
2244     for k := range xRefTable.Table {
2245         keys = append(keys, k)
2246     }
2247     sort.Ints(keys)
2248
2249     for _, objNr := range keys {
2250         err := dereferenceObject(ctx, objNr)
2251         if err != nil {
2252             return err
2253         }
2254     }
2255
2256     for _, objNr := range keys {
2257         entry := xRefTable.Table[objNr]
2258         if entry.Free || entry.Compressed {
2259             continue
2260         }
2261         processRefCounts(xRefTable, entry.Object)
```



```
2262     }
2263
2264     log.Read.Println("dereferenceObjects: end")
2265
2266     return nil
2267 }
2268
2269 // Locate a possible Version entry (since V1.4) in the catalog
2270 // and record this as rootVersion (as opposed to headerVersion).
2271 func identifyRootVersion(xRefTable *XRefTable) error {
2272
2273     log.Read.Println("identifyRootVersion: begin")
2274
2275     // Try to get Version from Root.
2276     rootVersionStr, err := xRefTable.ParseRootVersion()
2277     if err != nil {
2278         return err
2279     }
2280
2281     if rootVersionStr == nil {
2282         return nil
2283     }
2284
2285     // Validate version and save corresponding constant to xRefTable.
2286     rootVersion, err := PDFVersion(*rootVersionStr)
2287     if err != nil {
2288         return errors.Wrapf(err, "identifyRootVersion: unknown PDF Root version:
2289 %s\n", *rootVersionStr)
2290     }
2291
2292     xRefTable.RootVersion = &rootVersion
2293
2294     // since V1.4 the header version may be overridden by a Version entry in the
2295     // catalog.
2296     if *xRefTable.HeaderVersion < V14 {
2297         log.Info.Printf("identifyRootVersion: PDF version is %s - will ignore root
2298 version: %s\n",
2299             xRefTable.HeaderVersion, *rootVersionStr)
2300     }
2301
2302     log.Read.Println("identifyRootVersion: end")
2303
2304     return nil
2305 }
2306
2307 // Parse all Objects including stream content from file and save to the corresponding
2308 xRefTableEntries.
2309 // This includes processing of object streams and linearization dicts.
2310 func dereferenceXRefTable(ctx *Context, conf *Configuration) error {
2311
2312     log.Read.Println("dereferenceXRefTable: begin")
2313
2314     xRefTable := ctx.XRefTable
2315
2316     // Note for encrypted files:
2317     // Mandatory provide userpw to open & display file.
2318     // Access may be restricted (Decode access privileges).
2319     // Optionally provide ownerpw in order to gain unrestricted access.
2320     err := checkForEncryption(ctx)
2321     if err != nil {
```

```
2318     return err
2319 }
2320 //fmt.Println("pw authenticated")
2321
2322 // Prepare decompressed objects.
2323 err = decodeObjectStreams(ctx)
2324 if err != nil {
2325     return err
2326 }
2327
2328 // For each xRefTableEntry assign a Object either by parsing from file or pointing
to a decompressed object.
2329 err = dereferenceObjects(ctx)
2330 if err != nil {
2331     return err
2332 }
2333
2334 // Identify an optional Version entry in the root object/catalog.
2335 err = identifyRootVersion(xRefTable)
2336 if err != nil {
2337     return err
2338 }
2339
2340 log.Read.Println("dereferenceXRefTable: end")
2341
2342 return nil
2343 }
2344
2345 func handleUnencryptedFile(ctx *Context) error {
2346
2347     if ctx.Cmd == DECRYPT || ctx.Cmd == SETPERMISSIONS {
2348         return errors.New("pdfcpu: this file is not encrypted")
2349     }
2350
2351     if ctx.Cmd != ENCRYPT {
2352         return nil
2353     }
2354
2355     // Encrypt subcommand found.
2356
2357     if ctx.OwnerPW == "" {
2358         return errors.New("pdfcpu: please provide owner password and optional user
password")
2359     }
2360
2361     return nil
2362 }
2363
2364 func idBytes(ctx *Context) (id []byte, err error) {
2365
2366     if ctx.ID == nil {
2367         return nil, errors.New("pdfcpu: missing ID entry")
2368     }
2369
2370     hl, ok := ctx.ID[0].(HexLiteral)
2371     if ok {
2372         id, err = hl.Bytes()
2373         if err != nil {
2374             return nil, err

```

```
2375     }
2376 } else {
2377     sl, ok := ctx.ID[0].(StringLiteral)
2378     if !ok {
2379         return nil, errors.New("pdfcpu: ID must contain hex literals or string
literals")
2380     }
2381     id, err = Unescape(sl.Value())
2382     if err != nil {
2383         return nil, err
2384     }
2385 }
2386
2387 return id, nil
2388 }
2389
2390 func needsOwnerAndPassword(cmd CommandMode) bool {
2391
2392     return cmd == CHANGEOPW || cmd == CHANGEUPW || cmd == SETPERMISSIONS
2393 }
2394
2395 func handlePermissions(ctx *Context) error {
2396
2397     // AES256 Validate permissions
2398     ok, err := validatePermissions(ctx)
2399     if err != nil {
2400         return err
2401     }
2402
2403     if !ok {
2404         return errors.New("pdfcpu: corrupted permissions after upw ok")
2405     }
2406
2407     // Double check minimum permissions for pdfcpu processing.
2408     if !hasNeededPermissions(ctx.Cmd, ctx.E) {
2409         return errors.New("pdfcpu: insufficient access permissions")
2410     }
2411
2412     return nil
2413 }
2414
2415 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2416
2417     ctx.E, err = supportedEncryption(ctx, d)
2418     if err != nil {
2419         return err
2420     }
2421
2422     ctx.E.ID, err = idBytes(ctx)
2423     if err != nil {
2424         return err
2425     }
2426
2427     var ok bool
2428
2429     //fmt.Printf("opw: <%s> upw: <%s> \n", ctx.OwnerPW, ctx.UserPW)
2430
2431     // Validate the owner password aka. permissions/master password.
2432     ok, err = validateOwnerPassword(ctx)
```

```
2433     if err ≠ nil {
2434         return err
2435     }
2436
2437     // If the owner password does not match we generally move on if the user password
2438     // is correct
2439     // unless we need to insist on a correct owner password due to the specific
2440     // command in progress.
2441     if !ok && needsOwnerAndPassword(ctx.Cmd) {
2442         return errors.New("pdfcpu: please provide the owner password with -opw")
2443     }
2444
2445     // Generally the owner password, which is also regarded as the master password or
2446     // set permissions password
2447     // is sufficient for moving on. A password change is an exception since it
2448     // requires both current passwords.
2449     if ok && !needsOwnerAndPassword(ctx.Cmd) {
2450         // AES256 Validate permissions
2451         ok, err = validatePermissions(ctx)
2452         if err ≠ nil {
2453             return err
2454         }
2455         if !ok {
2456             return errors.New("pdfcpu: corrupted permissions after opw ok")
2457         }
2458         return nil
2459     }
2460
2461     // Validate the user password aka. document open password.
2462     ok, err = validateUserPassword(ctx)
2463     if err ≠ nil {
2464         return err
2465     }
2466     if !ok {
2467         return errors.New("pdfcpu: please provide the correct password")
2468     }
2469
2470     //fmt.Printf("upw ok: %t\n", ok)
2471
2472     return handlePermissions(ctx)
2473 }
2474
2475 func checkForEncryption(ctx *Context) error {
2476
2477     ir := ctx.Encrypt
2478
2479     if ir = nil {
2480         // This file is not encrypted.
2481         return handleUnencryptedFile(ctx)
2482     }
2483
2484     // This file is encrypted.
2485     log.Read.Printf("Encryption: %v\n", ir)
2486
2487     if ctx.Cmd == ENCRYPT {
2488         // We want to encrypt this file.
2489         return errors.New("pdfcpu: this file is already encrypted")
2490     }
2491
2492     // Dereference encryptDict.
```

```
2489     d, err := dereferencedDict(ctx, ir.ObjectNumber.Value())
2490     if err != nil {
2491         return err
2492     }
2493     log.Read.Printf("%s\n", d)
2494
2495     // We need to decrypt this file in order to read it.
2496     return setupEncryptionKey(ctx, d)
2497 }
2498
```