

```

38 2019
39
40 Copyright 2018 The pdfcpu authors.
41
42 Licensed under the Apache License, Version 2.0 (the "License");
43 you may not use this file except in compliance with the License.
44 You may obtain a copy of the License at
45
46     http://www.apache.org/licenses/LICENSE-2.0
47
48 Unless required by applicable law or agreed to in writing, software
49 distributed under the License is distributed on an "AS IS" BASIS,
50 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
51 See the License for the specific language governing permissions and
52 limitations under the License.
53
54
55 package pdfcpu
56
57 import (
58     "bufio"
59     "bytes"
60     "fmt"
61     "io"
62     "log"
63     "os"
64     "strings"
65     "time"
66
67     "github.com/pdfcpu/pdfcpu/pkg/api"
68     "github.com/pdfcpu/pdfcpu/pkg/font"
69     "github.com/pdfcpu/pdfcpu/pkg/generic"
70     "github.com/pdfcpu/pdfcpu/pkg/graphics"
71     "github.com/pdfcpu/pdfcpu/pkg/output"
72
73     "github.com/pkg/errors"
74
75     "golang.org/x/net/context"
76 )
77
78 // Read reads a PDF file and builds an internal structure holding its cross
79 // reference table and the objects.
80 func Read(filename string, conf Configuration) (*Context, error) {
81     log.Infof("Reading %s (%v)", filename)
82
83     f, err := os.Open(filename)
84     if err != nil {
85         return nil, errors.Wrap(err, "can't open %s", filename)
86     }
87
88     defer func() {
89         f.Close()
90     }()
91
92     return Read(f, conf)
93 }
94
95 // Read takes a reader/seeker and generates a Context,
96 // or an error in case of a non-readable or corrupt reference table.
97 func Read(reader io.Reader, conf Configuration) (*Context, error) {

```

```

220 //offset, err = stream.read(fields[0], 10, 64)
221 if err != nil {
222     return err
223 }
224
225 generation, err = stream.Async(fields[1])
226 if err != nil {
227     return err
228 }
229
230 entryType = fields[2]
231 if entryType == "0" {
232     return errors.New("offset: parseOffsetTable: corrupt ref subsection
233 entry")
234 }
235
236 var xrefOffsetTable *xrefOffsetTable
237
238 if entryType == "1" {
239     // in one object
240
241     log.Read.Printf("parseOffsetTable: Object %d is in use at offset%0d,
242 generation%0d", objectNumber, offset, generation)
243
244     if offset == 0 {
245         log.Info.Printf("parseOffsetTable: Skip entry for in use object %d
246 with offset 0", objectNumber)
247         return nil
248     }
249
250     xrefOffsetTable =
251         xrefOffsetTable{
252             From:      false,
253             Offset:      &offset,
254             Generation: &generation}
255
256     // free object
257
258     log.Read.Printf("parseOffsetTable: Object %d is unused, next free is
259 object%0d", objectNumber, offset, generation)
260
261     xrefOffsetTable =
262         xrefOffsetTable{
263             From:      true,
264             Offset:      &offset,
265             Generation: &generation}
266 }
267
268 log.Read.Printf("parseOffsetTable: Insert new xrefable table for Object %0d",
269 objectNumber)
270
271 objTable.Table[objectNumber] = xrefOffsetTable
272
273 log.Read.Printf("parseOffsetTable: end")
274
275 return nil
276 }

```

```

443         // b = b * (a < b) ? a : b * (a <= 0 ? 1 : 0) + refTableEntry[i]
444     }
445
446     objectNumber = new Object[objectSize]
447
448     //start = i + 1
449     // b = bufTable[i] * (start - iStart + 1)
450     // c = bufTable[i] * (start - i2 + 1)
451     var objRefTableEntry = new ObjectTableEntry(i, start - iStart + 1, start - i2 + 1)
452     var objRefTableEntry = objRefTableEntry
453
454     switch buf[i] {
455     case 0:
456         case 0:
457             log.Read.Print("extraObjectTableEntryFromObjRefStream: Object obj is
458             unused, next free is objNumber, generationNumber, objectNumber, c, 3)
459             c = int(c)
460             objRefTableEntry =
461                 objRefTableEntry
462             // Free, true,
463             // Compressed: false,
464             // Offset: 80,
465             // Generation: 0
466
467         case 0:
468             log.Read.Print("extraObjectTableEntryFromObjRefStream: Object obj is
469             used at offset=0, generationNumber, objectNumber, c, 3)
470             c = int(c)
471             objRefTableEntry =
472                 objRefTableEntry
473             // Free: false,
474             // Compressed: false,
475             // Offset: 80,
476             // Generation: 0
477
478         case 0:
479             log.Read.Print("extraObjectTableEntryFromObjRefStream: Object obj is
480             compressed at offset=0, generationNumber, objectNumber, c, 3)
481             c = int(c)
482             objRefTableEntry =
483                 objRefTableEntry
484             // Free: false,
485             // Compressed: false,
486             // Offset: 80,
487             // Generation: 0
488
489         case 0:
490             // compressed object
491             // generation change &
492             log.Read.Print("extraObjectTableEntryFromObjRefStream: Object obj is
493             compressed at offset=0, generationNumber, c, 3)
494             objNumber = int(c2)
495             objIndex = int(c3)
496
497             objRefTableEntry =
498                 objRefTableEntry
499             // Free: false,
500             // Compressed: true,
501             // ObjectStream: 808NumberRef,
502             // Generation: objIndex
503
504             ctx.objRefTableEntry[objNumber] = true
505
506             }
507
508     if ctx.objRefTable.Exists(objectNumber) {
509         log.Read.Print("extraObjectTableEntryFromObjRefStream: Skip obj id =
510         objNumber")
511     }

```

[illegible]

```

907 // dict {
908   log.Mod.Printf("line (%mM) %v\n", len(line), line)
909 }
910 trailerString, err := scanFields(), trailerString()
911 if err == nil {
912     return nil, err
913 }
914
915 log.Mod.Printf("processTrailer: trailerString: (%mM) %v\n",
916     len(trailerString), trailerString)
917
918 o, err := paraObject(&trailerString)
919 if err == nil {
920     return nil, err
921 }
922 trailerPkt, ok = o.Dict()
923 if !ok {
924     return nil, errors.New("pdpca: processTrailer: corrupt trailer dict")
925 }
926
927 log.Mod.Printf("processTrailer: trailerDict(%v)\n", trailerDict)
928 return para.TrailerDict(trailerDict, ctx)
929 }
930
931 // Para sub section into corresponding number of subF table entries
932 func paraSubSectionToSubFI(scanner, ctx context.Context) (*uint64, error) {
933     log.Mod.Printf("paraSubSectionToSubFI begin")
934
935     line, err := scanLine()
936     if err == nil {
937         return nil, err
938     }
939
940     log.Mod.Printf("paraSubSectionToSubFI: %v\n", line)
941     fields = strings.Fields(line)
942
943     // Process all sub sections of this subF section.
944     for strings.HasPrefix(line, "trialer") && len(fields) == 2 {
945         if err := para.SubSectionToSubFI(ctx, &fields, Fields); err == nil {
946             return nil, err
947         }
948     }
949
950     // trailer or another test table subsection ?
951     if line[0] == scanLine() {
952         return nil, err
953     }
954
955     // If many line try next line for trailer
956     if len(line) == 0 {
957         if trailer, err := scanLine(); err == nil {
958             return nil, err
959         }
960     }

```

```

113  rs = ctx.NewReader()
114
115  hw, wCount, err = headerVersion(rs)
116  if err != nil {
117      return err
118  }
119
120  ctx.HeaderVersion = hw
121  ctx.HeaderLength = wCount
122
123  for offset := nil {
124      {
125          rd, err = newPositionalReader(rs, offset)
126          if err != nil {
127              return err
128          }
129
130          s := bufio.NewScanner(rd)
131          s.Split(scanLines)
132
133          line, err = scanLine(s)
134          if err == nil {
135              return err
136          }
137
138          log.Read.Printf("line: %s\n", line)
139      }
140
141      if strings.TrimSpace(line) == "read" {
142          log.Read.Printf("builderForTableStarting: found xref section")
143          if offset, err = parseRefSection(s, ctx); err == nil {
144              return err
145          }
146      } else {
147          log.Read.Printf("builderForTableStarting: found xref stream")
148          ctx.Read.IngesterStream = true
149          rd, err = newPositionalReader(rs, offset)
150          if err != nil {
151              return err
152          }
153          if offset, err = parseRefStream(rd, offset, ctx); err == nil {
154              log.Read.Printf("builderForTableStarting: found xref section")
155              // fix fix for current xref in xref section.
156              return xrefParser(ctx).Fix()
157          }
158      }
159  }
160
161  log.Read.Printf("builderForTableStarting: end")
162
163  return nil
164 }
165
166 // Populate the cross reference table for this PDF file.
167 // Note offset of first xref table must be
168 // "Can be 'xref' or indirect object reference eg. '11 0 obj'"
169 // and build the xref table along with the map.
170 func readCrossRef(rs io.Reader) (err error) {

```

```

2520 //
2521 log.Debug.Print("Read begin")
2522
2523 ctx, err := NewContext(s, config)
2524 if err != nil {
2525     return nil, err
2526 }
2527
2528 if ctx.ReadOnly {
2529     // Log info,PrintInfo("PDF Version 1.0 conforming reader")
2530
2531     // Log info,PrintInfo("PDF Version 1.4 conforming reader - no object streams
2532     // (refStreams allowed)")
2533 }
2534
2535 // Populate metaTable.
2536 if err := readMetaTable(ctx); err != nil {
2537     return nil, errors.Wrap(err, "metaTable failed")
2538 }
2539
2540 // Make all objects explicitly available (load into memory) in corresponding
2541 // metaTable entry.
2542 // Also decode any indirect object streams.
2543 if err := deferDecodeMetaTable(ctx, config); err != nil {
2544     return nil, err
2545 }
2546
2547 // Some scanners write an incorrect file size trailer.
2548 if ctx.MetaTable.Size < len(ctx.MetaTable.Table) {
2549     ctx.MetaTable.Size = len(ctx.MetaTable.Table)
2550 }
2551
2552 log.Debug.Print("Read: end")
2553
2554 return ctx, nil
2555 }
2556
2557 // ScanLines is a split function for a Scanner that returns each line of
2558 // text, stripped of any trailing end-of-line markers. The returned line may
2559 // be a single line, or a carriage return followed by a carriage return followed
2560 // by one newline or one carriage return or one newline.
2561 // The number of lines will be returned even if it has no newline.
2562 func scanLines(data []byte, offset bool) (advance int, token []byte, err error) {
2563     if atEOF := len(data) == 0 {
2564         return 0, nil, nil
2565     }
2566
2567     indexR := bytes.IndexByte(data, '\r')
2568     indexLF := bytes.IndexByte(data, '\n')
2569
2570     switch {
2571     case indexR >= 0 && indexR > 0:
2572         if indexR < indexLF {
2573             return indexR + 1, data[:indexR], nil
2574         }
2575         return indexR + 1, data[:indexR], nil
2576     }
2577     return 0, nil, nil
2578 }

```

```

6982
6983 // Parse the start of stream and create corresponding table objects
6984 func parseTablesSubSection() *bufio.Scanner, *byteTable, *byteTable {
6985     log.Read.Println("parseTablesSubSection: begin")
6986
6987     startOfNumber, err := stream.Atol(fields[8])
6988     if err != nil {
6989         return err, nil
6990     }
6991
6992     objCount, err := stream.Atol(fields[1])
6993     if err != nil {
6994         return err, nil
6995     }
6996
6997     log.Read.Println("deducted stream subsection, startOfNumber length="+startOfNumber,
6998         startOfNumber=objCount)
6999
7000     // Process all entries of this subsection into xheffable entries.
7001     for i := 0; i < objCount; i++ {
7002         if err := parseTablesEntry(i, xheffTable, startOfNumber); err != nil {
7003             return err, nil
7004         }
7005     }
7006
7007     log.Read.Println("parseTablesSubSection: end")
7008
7009     return nil, nil
7010 }
7011
7012 // Parse compressed object
7013 func compressedObject(i string) (Object, error) {
7014
7015     log.Read.Println("compressedObject: begin")
7016
7017     o, err := parseObject(i)
7018     if err != nil {
7019         return nil, err
7020     }
7021
7022     d, ok := o.(Dict)
7023     if !ok {
7024         // Return trivial Object: Integer, Array, etc.
7025         log.Read.Println("compressedObject: end, any other than dict")
7026         return o, nil
7027     }
7028
7029     streamLength, streamEntryLen := d.Length()
7030     if streamLength == nil || d.StreamLength() == nil {
7031         // Return dict
7032         log.Read.Println("compressedObject: end, dict")
7033         return o, nil
7034     }
7035
7036     return nil, errors.New("Ofcpu: compressedObject: stream objects are not to be
7037         stored in dict stream")
7038 }

```

[illegible][illegible]

```

945         fields = strings.Fields(line)
946     }
947
948     log.Header.Println("parsingSection: All subsections read")
949
950     if strings.HasPrefix(line, "trailer") {
951         return nil, errors.Errorf("parsingSection: missing trailer dict, line = %s",
952             line)
953     }
954
955     log.Header.Println("parsingSection: parsing trailer dict.")
956
957     return processTrailerDict(s, line)
958 }
959
960 // get version from first line of file.
961 // Beginning with PDF 1.4, the version entry in the document's catalog dictionary
962 // is the only place where the file's trailer, as described in 7.2.5, "File
963 // Trailer", may be used instead of the version specified in the header.
964 // See PDF version from header to shiftable.
965 // See PDF version from first line of file.
966 // nil/empty is the number of characters used for m0 (1 or 2).
967 func headersVer(s io.Reader) (v Version, s0 int, err error) {
968     return headersVer(s, "headersBegin")
969 }
970
971 var errHeaderVerFromFile = errors.New("pdf:header version: corrupt pdf stream - no
972     headersBegin")
973
974 // get first line of file which holds the version of the PDF file.
975 // we call this the header version.
976 func headersVerFromFile(s io.Reader) (v Version, s0 int, err error) {
977     return nil, 0, err
978 }
979
980 buf := make([]byte, 25)
981 if err := s.Read(buf); err != nil {
982     return nil, 0, err
983 }
984
985 n := strings.Index(
986     prefix + "supp.",
987
988     if len(s) < 8 || !strings.HasPrefix(s, prefix) {
989         return nil, 0, errors.Errorf("header version: unknown PDF Header Version")
990     }
991     return nil, 0, errors.New(err)
992 }
993
994 s = s[8:]
995 s = strings.TrimLeft(s, "\t \r")
996
997 // Detect the used end token which should be 1 (endb, endd) or 2 chars (endbdl/long,
998     enddlong)

```

```

1157 log.Debug.Printf("readofftable: begin")
1158
1159 // Read offstart bytes from section
1160 offset, err = offstartbytesInSection(ctx)
1161 if err != nil {
1162     return
1163 }
1164
1165 err = bufio.NewReaderAtStartingAt(ctx, offset)
1166 if err != io.EOF {
1167     return errors.Wrap(err, "readstarttable: unexpected eof")
1168 }
1169
1170 if err != nil {
1171     return
1172 }
1173
1174 // Log list of free objects out the "free list"
1175 // Log.Read.Printf("FreeList: %v", ctx.FreeObjects)
1176
1177 // Ensure valid FreeList of objects.
1178 err = ctx.EnsureValidFreeList()
1179 if err != nil {
1180     return
1181 }
1182
1183 log.Debug.Printf("readstarttable: end")
1184
1185 return
1186 }
1187
1188 func growBuf(buf []byte, size, int, rd io.Reader) ([]byte, error) {
1189     b := make([]byte, size)
1190     _, err = rd.Read(b)
1191     if err != nil {
1192         return nil, err
1193     }
1194     return buf[:len(buf)+len(b)], nil
1195 }
1196
1197 // Log.Read.Printf("growBuf: Read %d bytes", n)
1198
1199 return append(buf, b..., nil)
1200 }
1201
1202 func nextStreamOffsetInString, streamEnd(int) (off int) {
1203     off = streamEnd + len(string)
1204
1205     // Skip next null bytes
1206     // TODO Should be skip optional whitespace instead
1207     for ; lineOffset == 0; off++ {
1208     }
1209
1210     // Skip 80 col.
1211     if lineOffset == "0" {
1212         off =
1213     }
1214
1215     // Skip 80 col.
1216 }

```

```

302         return index + 1, data[index&S], nil
303     }
304     }
305     return index + 1, data[0:index&S], nil
306 }
307
308 case index >= 0:
309     // we have a full carriage return terminated line.
310     return index + 1, data[0:index&S], nil
311 }
312
313 case index >= 0:
314     // we have a full newline-terminated line.
315     return index + 1, data[0:index&S], nil
316 }
317
318 // If we're at EOF, we have a final, non-terminated line. Return it.
319 if !atEOF {
320     return nil(data), data, nil
321 }
322
323 // Return more data.
324 return 0, nil, nil
325 }
326
327 func newPositionedReader(rs io.Reader, offset int64) (*bufio.Reader, error) {
328     if _, err := rs.Seek(offset, io.SeekStart); err != nil {
329         return nil, err
330     }
331     log.Printf("newPositionedReader: positioned to offset: %d\n", offset)
332     return bufio.NewReader(rs), nil
333 }
334
335 // Get the file offset of the last WriteSection.
336 // Go to end of file and search backwards for the first occurrence of StartStr
337 // offset must be a file offset
338 func rsToLastWriteSection(ctxt *Context) (*int64, error) {
339     rs := ctxt.Read-rs
340
341     var (
342         prevBuf, workBuf [byte]
343         bufSize         int64 = 512
344         offset          int64
345     )
346
347     for i := 0; offset == 0; i++ {
348         if err := rs.Seek(-int64(i)*bufSize, io.SeekEnd);
349             err == nil ||
350             rs.Err() != nil {
351             return nil, errors.New("pdirpos: can't find last write section")
352         }
353         log.Printf("scanning for offsetLastWriteSection starting at %d\n", off)
354         curBuf := make([]byte, bufSize)
355     }
356 }

```

[illegible]

```

556         return nil, err
557     }
558
559     log.Debug.Printf("parseStream: offset=0x%04x | stream=0x%04x", offset, streamId)
560     endId, streamId :=
561         W.Line + string(buf)
562     // We expect a stream and therefore "stream before "endId" if "endId" within
563     // "endId". Is there a guarantee that "endId" is contained in this buffer for large
564     // streams?
565     if streamId < 0 || (endId > 0 & endId < streamId) {
566         log.Warn.Printf("offset: parseStream: corrupt dfp file")
567         return nil, err
568     }
569
570     // Build object parse buf
571     // If nil, streamId = 0
572     obj := nil
573     objNumber, generationNumber, err := parseObjectAttributes(&obj)
574     // If err != nil, return
575     return nil, err
576 }
577
578 // Parse this object
579 func (p *Parser) parseObject(
580     log *Log,
581     logID, streamId, xrefId objId, objIdNil bool, objNumber,
582     generationNumber int,
583     err error) (obj *Object, err error) {
584     // If err != nil
585     if err != nil {
586         log.Warn.Printf("err: 'parseStream: no object')")
587         return nil, err
588     }
589
590     log.Debug.Printf("parseStream: We have an object: %04x", obj)
591
592     streamOffset := xrefId
593     id, err := streamIdToObjId(objNumber, streamOffset)
594     if err != nil
595         return nil, err
596     // If we have an xref stream object
597     if id == nil {
598         // Parse trailer stream object, dx, dx:sha1
599         id, err = parseTrailerStreamObject(dx, dx:sha1)
600         if err != nil
601             return nil, err
602     }
603     // Parse stream object and create sha1able entries for embedded objects.
604     obj = extractObjectTailBytesFromDfPStream(dx.Content, dx, dx)
605     if err != nil
606         return nil, err
607 }
608
609 // Create sha1able entry for SHA1Streamid.
610 entry =
611     Metadata{
612         Free: false,
613         Offset: offset,
614         Generation: generationNumber,
615         Object: *obj
616     }

```

```

789         return s1, nil
790     }
791 }
792
793 func IsIndex(s string) (bool, error) {
794     s, err := para(s)
795     if err != nil {
796         return false, err
797     }
798     s, ok := s.(int)
799     return ok, nil
800 }
801
802 func scanTrailer(s bufio.Scanner, line string) (string, error) {
803     //
804     var buf bytes.Buffer
805     var err error
806     var i, j int
807     loopRead.Print("line: %s\n", line)
808     // Scan for disc start tag "=="
809     i = strings.Index(line, "=")
810     if i >= 0 {
811         break
812     }
813     line, err = scan(s)
814     loopRead.Print("line: %s\n", line)
815     if err != nil {
816         return "", err
817     }
818     //
819     line = line[i:]
820     buf.WriteString(line)
821     buf.WriteString("\n")
822     loopRead.Print("scanTrailer dictbuf after start tag: %s\n", line)
823     // Scan for disc and tag "==" but account for inner discs.
824     line = line[i:]
825     for {
826         if !isLine() {
827             line, err = scanLine(s)
828             if err != nil {
829                 return "", err
830             }
831             buf.WriteString(line)
832             buf.WriteString("\n")
833             loopRead.Print("scanTrailer dictbuf max: line: %s\n", line)
834             //
835             i = strings.Index(line, "=")
836             if i < 0 {
837                 break
838             }
839             j = strings.Index(line, "\n")
840             if j >= 0 {

```

[illegible][illegible]

```

267 // err = err.Append(curbuf)
268 if err != nil {
269     return nil, err
270 }
271
272
273
274 wordBuf := curbuf
275 if predefn == nil {
276     wordBuf = append(curbuf, predefn...)
277 }
278
279
280 j := strings.LastIndex(string(wordBuf), "startref")
281 if j == -1 {
282     predefn = curbuf
283     continue
284 }
285
286
287 p := wordBuf[j+len("startref"):]
288 posdef := string.Index(string(p), "MEMOF")
289 if posdef == -1 {
290     return nil, errors.New("pdefpos: no matching MEMOF for startref")
291 }
292
293
294 p = p[posdef:]
295 offset, err := strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
296 if err != nil {
297     return nil, errors.New("pdefpos: corrupted last xref section")
298 }
299
300
301 log.Red.Printf("offset last xrefsection: %d\n", offset)
302
303
304 return boffset, nil
305 }
306
307
308 // Read next subsection entry and generate corresponding xref table entry.
309 func parseSubtableEntry(s bufio.Scanner, xrefTable xrefTable, objectIndex int) error {
310     log.Red.Printf("parseSubtableEntry: begin")
311
312     line, err := scanLine(s)
313     if err != nil {
314         return err
315     }
316
317     obj := objTable.Exists(objectIndex) {
318         log.Red.Printf("parseSubtableEntry: end - Skip entry %d - already assigned", objectIndex)
319     }
320     return nil
321 }
322
323
324 fields := strings.Split(line)
325 if (len(fields)) != 5 || !isInt(fields[0]) || !isInt(fields[1]) || !isInt(fields[2]) || !isInt(fields[3]) || !isInt(fields[4]) {
326     return errors.New("pdefpos: parseSubtableEntry: corrupt xref subsection")
327 }
328
329

```

```

440         @SuppressWarnings("unchecked")
441         obj.putArray = obj.putArray
442         log.Read_PrintfLn("para:ObjectStream end")
443     }
444     return nil
445 }
446
447 // For each object embedded in this stream create the corresponding obj table entry
448 // This is done by creating an object table entry and then creating an object stream
449 // object
450 func (obj *ObjectStream) readObjectStream() {
451     log.Read_Printf("extractObjectTableEntryFromObjectStream begin")
452 }
453
454 // Note:
455 // A value of zero for an element in the W array indicates that the corresponding
456 // field shall not be present in the stream
457 // The default value shall be zero, if there is one.
458 // If the first element is zero, the type field shall not be present, and shall
459 // default to type 1.
460
461 // Read the object table entry
462 func (obj *ObjectStream) readObjectTableEntry() {
463     w := make([]int, 16)
464     w[0] = 0
465     w[1] = 0
466     w[2] = 0
467     w[3] = 0
468     w[4] = 0
469     w[5] = 0
470     w[6] = 0
471     w[7] = 0
472     w[8] = 0
473     w[9] = 0
474     w[10] = 0
475     w[11] = 0
476     w[12] = 0
477     w[13] = 0
478     w[14] = 0
479     w[15] = 0
480     w[16] = 0
481     w[17] = 0
482     w[18] = 0
483     w[19] = 0
484     w[20] = 0
485     w[21] = 0
486     w[22] = 0
487     w[23] = 0
488     w[24] = 0
489     w[25] = 0
490     w[26] = 0
491     w[27] = 0
492     w[28] = 0
493     w[29] = 0
494     w[30] = 0
495     w[31] = 0
496     w[32] = 0
497     w[33] = 0
498     w[34] = 0
499     w[35] = 0
500     w[36] = 0
501     w[37] = 0
502     w[38] = 0
503     w[39] = 0
504     w[40] = 0
505     w[41] = 0
506     w[42] = 0
507     w[43] = 0
508     w[44] = 0
509     w[45] = 0
510     w[46] = 0
511     w[47] = 0
512     w[48] = 0
513     w[49] = 0
514     w[50] = 0
515     w[51] = 0
516     w[52] = 0
517     w[53] = 0
518     w[54] = 0
519     w[55] = 0
520     w[56] = 0
521     w[57] = 0
522     w[58] = 0
523     w[59] = 0
524     w[60] = 0
525     w[61] = 0
526     w[62] = 0
527     w[63] = 0
528     w[64] = 0
529     w[65] = 0
530     w[66] = 0
531     w[67] = 0
532     w[68] = 0
533     w[69] = 0
534     w[70] = 0
535     w[71] = 0
536     w[72] = 0
537     w[73] = 0
538     w[74] = 0
539     w[75] = 0
540     w[76] = 0
541     w[77] = 0
542     w[78] = 0
543     w[79] = 0
544     w[80] = 0
545     w[81] = 0
546     w[82] = 0
547     w[83] = 0
548     w[84] = 0
549     w[85] = 0
550     w[86] = 0
551     w[87] = 0
552     w[88] = 0
553     w[89] = 0
554     w[90] = 0
555     w[91] = 0
556     w[92] = 0
557     w[93] = 0
558     w[94] = 0
559     w[95] = 0
560     w[96] = 0
561     w[97] = 0
562     w[98] = 0
563     w[99] = 0
564     w[100] = 0
565     w[101] = 0
566     w[102] = 0
567     w[103] = 0
568     w[104] = 0
569     w[105] = 0
570     w[106] = 0
571     w[107] = 0
572     w[108] = 0
573     w[109] = 0
574     w[110] = 0
575     w[111] = 0
576     w[112] = 0
577     w[113] = 0
578     w[114] = 0
579     w[115] = 0
580     w[116] = 0
581     w[117] = 0
582     w[118] = 0
583     w[119] = 0
584     w[120] = 0
585     w[121] = 0
586     w[122] = 0
587     w[123] = 0
588     w[124] = 0
589     w[125] = 0
590     w[126] = 0
591     w[127] = 0
592     w[128] = 0
593     w[129] = 0
594     w[130] = 0
595     w[131] = 0
596     w[132] = 0
597     w[133] = 0
598     w[134] = 0
599     w[135] = 0
600     w[136] = 0
601     w[137] = 0
602     w[138] = 0
603     w[139] = 0
604     w[140] = 0
605     w[141] = 0
606     w[142] = 0
607     w[143] = 0
608     w[144] = 0
609     w[145] = 0
610     w[146] = 0
611     w[147] = 0
612     w[148] = 0
613     w[149] = 0
614     w[150] = 0
615     w[151] = 0
616     w[152] = 0
617     w[153] = 0
618     w[154] = 0
619     w[155] = 0
620     w[156] = 0
621     w[157] = 0
622     w[158] = 0
623     w[159] = 0
624     w[160] = 0
625     w[161] = 0
626     w[162] = 0
627     w[163] = 0
628     w[164] = 0
629     w[165] = 0
630     w[166] = 0
631     w[167] = 0
632     w[168] = 0
633     w[169] = 0
634     w[170] = 0
635     w[171] = 0
636     w[172] = 0
637     w[173] = 0
638     w[174] = 0
639     w[175] = 0
640     w[176] = 0
641     w[177] = 0
642     w[178] = 0
643     w[179] = 0
644     w[180] = 0
645     w[181] = 0
646     w[182] = 0
647     w[183] = 0
648     w[184] = 0
649     w[185] = 0
650     w[186] = 0
651     w[187] = 0
652     w[188] = 0
653     w[189] = 0
654     w[190] = 0
655     w[191] = 0
656     w[192] = 0
657     w[193] = 0
658     w[194] = 0
659     w[195] = 0
660     w[196] = 0
661     w[197] = 0
662     w[198] = 0
663     w[199] = 0
664     w[200] = 0
665     w[201] = 0
666     w[202] = 0
667     w[203] = 0
668     w[204] = 0
669     w[205] = 0
670     w[206] = 0
671     w[207] = 0
672     w[208] = 0
673     w[209] = 0
674     w[210] = 0
675     w[211] = 0
676     w[212] = 0
677     w[213] = 0
678     w[214] = 0
679     w[215] = 0
680     w[216] = 0
681     w[217] = 0
682     w[218] = 0
683     w[219] = 0
684     w[220] = 0
685     w[221] = 0
686     w[222] = 0
687     w[223] = 0
688     w[224] = 0
689     w[225] = 0
690     w[226] = 0
691     w[227] = 0
692     w[228] = 0
693     w[229] = 0
694     w[230] = 0
695     w[231] = 0
696     w[232] = 0
697     w[233] = 0
698     w[234] = 0
699     w[235] = 0
700     w[236] = 0
701     w[237] = 0
702     w[238] = 0
703     w[239] = 0
704     w[240] = 0
705     w[241] = 0
706     w[242] = 0
707     w[243] = 0
708     w[244] = 0
709     w[245] = 0
710     w[246] = 0
711     w[247] = 0
712     w[248] = 0
713     w[249] = 0
714     w[250] = 0
715     w[251] = 0
716     w[252] = 0
717     w[253] = 0
718     w[254] = 0
719     w[255] = 0
720     w[256] = 0
721     w[257] = 0
722     w[258] = 0
723     w[259] = 0
724     w[260] = 0
725     w[261] = 0
726     w[262] = 0
727     w[263] = 0
728     w[264] = 0
729     w[265] = 0
730     w[266] = 0
731     w[267] = 0
732     w[268] = 0
733     w[269] = 0
734     w[270] = 0
735     w[271] = 0
736     w[272] = 0
737     w[273] = 0
738     w[274] = 0
739     w[275] = 0
740     w[276] = 0
741     w[277] = 0
742     w[278] = 0
743     w[279] = 0
7
```

```

545         Log.Read_Printf("parseStream: Insert new vtable entry for object %Min",
546             objNameNumber)
547     }
548     ctx.table[objNameNumber] = &entry
549     Log.Read_Printf("parseStream: objNameNumber == TRUE
550     prevOffset == id.PrevOffsetFrom
551     621
552     Log.Read_Printf("parseStream: end")
553     return prevOffsetFrom, nil
554 }
555
556 // returns vtableEntry and a 32-bit error flag
557 func parseVtableStream(offset int64, ctx *Context) error {
558     Log.Read_Printf("parseStream: begin")
559     id, err := readVariableHeaderFrom(ctx.ReadR, offset)
560     if err != nil {
561         return err
562     }
563     return parseVtableStream(id, offset, ctx)
564 }
565
566 // returns err
567 func parseVtableStream(id int64, offset int64, ctx *Context) error {
568     Log.Read_Printf("parseVtableStream: end")
569     return nil
570 }
571
572 // returns vtableEntry and a 32-bit error flag
573 func parseVtableEntry(id int64, objName *VtableEntry) error {
574     Log.Read_Printf("parseVtableEntry: begin")
575     if found := findInCtx("vtableEntry"); found {
576         ctx.vtableEntry = &id.vtableEntry["vtableEntry"]
577     } else {
578         ctx.vtableEntry = nil
579     }
580     vtableEntry, err := ctx.vtableEntry[id]
581     Log.Read_Printf("parseVtableEntry: object objName",
582         objName)
583     if err != nil {
584         return err
585     }
586     if vtableEntry.Size == nil {
587         size := &id.size
588         if size == nil {
589             return errors.New("object: parseVtableEntry: missing entry (%v)",
590                 objName)
591         }
592         // new variable
593         // returns after all read in.
594         vtableEntry.Size = size
595     }
596     if vtableEntry.Root == nil {
597         rootObj := &id.rootEntry["vtableEntry"]
598     }

```

```

847         }
848         if k == 0
849             // Check for diff
850             ok, err = isDiff(buf.String())
851             if err == nil || ok {
852                 return buf.String(), nil
853             }
854         } else {
855             k++
856         }
857     }
858     continue
859 }
860 // Append
861 line, err = scanLine(s)
862 if err == nil || ok {
863     return "", err
864 }
865 buf.WriteString(line)
866 buf.WriteString("\n")
867 log.Printf("Trailer diff: %s\n", buf.String())
868 } else {
869     // Append
870     line, err = scanLine(s, "no")
871     if j < 0 {
872         // ok
873         k++
874         line = line[j+1:]
875     } else {
876         // diff
877         if s < 0 {
878             // handle ok
879             k++
880             line = line[j+1:]
881         } else {
882             // handle no
883             if k == 0 {
884                 // Check for diff
885                 ok, err = isDiff(buf.String())
886                 if err == nil || ok {
887                     return buf.String(), nil
888                 }
889             } else {
890                 k++
891             }
892             line = line[j+1:]
893         }
894     }
895 }
896 }
897 }
898 }
899
900 func processTrailer(tc Context, s bufio.Scanner, line string) (error, []byte) {
901     var trailerString string
902     if line == "Trailer" {
903         trailerString = line[7:]
904         log.Printf("Trailer: %s\n", trailerString)
905     }
906 }

```

```

1004 //err = processTrailerLine(x, string(bb)
1005 //return err
1006 }
1007 continue
1008 }
1009 }
1010 //count all units "trailer"
1011 //l = string.LineLine, "trailer"
1012 if l > 0 {
1013     be = append(bb, line...)
1014     withinTrailer = true
1015 }
1016 }
1017 continue
1018 }
1019 //l = string.LineLine, "start"
1020 if l > 0 {
1021     offset = lastIdx(endLine) + eoCount
1022     withinHeader = true
1023 }
1024 }
1025 continue
1026 }
1027 //l = string.Line, "obj"
1028 if l > 0 {
1029     withinObj = true
1030 }
1031 }
1032 offset = offset, lineIdx-1}
1033 }
1034 be = append(bb, lineIdx-1}
1035 }
1036 offset = lastIdx(endLine) + eoCount
1037 }
1038 }
1039 }
1040 //return obj
1041 offset = append(endLine) + eoCount
1042 be = append(bb, "")
1043 be = append(bb, line...)
1044 }
1045 }
1046 }
1047 //l = string.Line, "endobj"
1048 if l > 0 {
1049     l = string(bb)
1050     if !objectGeneration, err = parseObjectAttributes(l)
1051     if err == nil {
1052         return err
1053     }
1054 }
1055 }
1056 }
1057 }
1058 }
1059 }
1060 }
1061 }
1062 }
1063 }
1064 }
1065 }
1066 }
1067 }
1068 }
1069 }
1070 }
1071 }
1072 }
1073 }
1074 }
1075 }
1076 }
1077 }
1078 }
1079 }
1080 }
1081 }
1082 }
1083 }
1084 }
1085 }
1086 }
1087 }
1088 }
1089 }
1090 }
1091 }
1092 }
1093 }
1094 }
1095 }
1096 }
1097 }
1098 }
1099 }
1100 }
1101 }
1102 }
1103 }
1104 }
1105 }
1106 }
1107 }
1108 }
1109 }
1110 }
1111 }
1112 }
1113 }
1114 }
1115 }
1116 }
1117 }
1118 }
1119 }
1120 }
1121 }
1122 }
1123 }
1124 }
1125 }
1126 }
1127 }
1128 }
1129 }
1130 }
1131 }
1132 }
1133 }
1134 }
1135 }
1136 }
1137 }
1138 }
1139 }
1140 }
1141 }
1142 }
1143 }
1144 }
1145 }
1146 }
1147 }
1148 }
1149 }
1150 }
1151 }
1152 }
1153 }
1154 }
1155 }
1156 }
1157 }
1158 }
1159 }
1160 }
1161 }
1162 }
1163 }
1164 }
1165 }
1166 }
1167 }
1168 }
1169 }
1170 }
1171 }
1172 }
1173 }
1174 }
1175 }
1176 }
1177 }
1178 }
1179 }
1180 }
1181 }
1182 }
1183 }
1184 }
1185 }
1186 }
1187 }
1188 }
1189 }
1190 }
1191 }
1192 }
1193 }
1194 }
1195 }
1196 }
1197 }
1198 }
1199 }
1200 }
1201 }
1202 }
1203 }
1204 }
1205 }
1206 }
1207 }
1208 }
1209 }
1210 }
1211 }
1212 }
1213 }
1214 }
1215 }
1216 }
1217 }
1218 }
1219 }
1220 }
1221 }
1222 }
1223 }
1224 }
1225 }
1226 }
1227 }
1228 }
1229 }
1230 }
1231 }
1232 }
1233 }
1234 }
1235 }
1236 }
1237 }
1238 }
1239 }
1240 }
1241 }
1242 }
1243 }
1244 }
1245 }
1246 }
1247 }
1248 }
1249 }
1250 }
1251 }
1252 }
1253 }
1254 }
1255 }
1256 }
1257 }
1258 }
1259 }
1260 }
1261 }
1262 }
1263 }
1264 }
1265 }
1266 }
1267 }
1268 }
1269 }
1270 }
1271 }
1272 }
1273 }
1274 }
1275 }
1276 }
1277 }
1278 }
1279 }
1280 }
1281 }
1282 }
1283 }
1284 }
1285 }
1286 }
1287 }
1288 }
1289 }
1290 }
1291 }
1292 }
1293 }
1294 }
1295 }
1296 }
1297 }
1298 }
1299 }
1300 }
1301 }
1302 }
1303 }
1304 }
1305 }
1306 }
1307 }
1308 }
1309 }
1310 }
1311 }
1312 }
1313 }
1314 }
1315 }
1316 }
1317 }
1318 }
1319 }
1320 }
1321 }
1322 }
1323 }
1324 }
1325 }
1326 }
1327 }
1328 }
1329 }
1330 }
1331 }
1332 }
1333 }
1334 }
1335 }
1336 }
1337 }
1338 }
1339 }
1340 }
1341 }
1342 }
1343 }
1344 }
1345 }
1346 }
1347 }
1348 }
1349 }
1350 }
1351 }
1352 }
1353 }
1354 }
1355 }
1356 }
1357 }
1358 }
1359 }
1360 }
1361 }
1362 }
1363 }
1364 }
1365 }
1366 }
1367 }
1368 }
1369 }
1370 }
1371 }
1372 }
1373 }
1374 }
1375 }
1376 }
1377 }
1378 }
1379 }
1380 }
1381 }
1382 }
1383 }
1384 }
1385 }
1386 }
1387 }
1388 }
1389 }
1390 }
1391 }
1392 }
1393 }
1394 }
1395 }
1396 }
1397 }
1398 }
1399 }
1400 }
1401 }
1402 }
1403 }
1404 }
1405 }
1406 }
1407 }
1408 }
1409 }
1410 }
1411 }
1412 }
1413 }
1414 }
1415 }
1416 }
1417 }
1418 }
1419 }
1420 }
1421 }
1422 }
1423 }
1424 }
1425 }
1426 }
1427 }
1428 }
1429 }
1430 }
1431 }
1432 }
1433 }
1434 }
1435 }
1436 }
1437 }
1438 }
1439 }
1440 }
1441 }
1442 }
1443 }
1444 }
1445 }
1446 }
1447 }
1448 }
1449 }
1450 }
1451 }
1452 }
1453 }
1454 }
1455 }
1456 }
1457 }
1458 }
1459 }
1460 }
1461 }
1462 }
1463 }
1464 }
1465 }
1466 }
1467 }
1468 }
1469 }
1470 }
1471 }
1472 }
1473 }
1474 }
1475 }
1476 }
1477 }
1478 }
1479 }
1480 }
1481 }
1482 }
1483 }
1484 }
1485 }
1486 }
1487 }
1488 }
1489 }
1490 }
1491 }
1492 }
1493 }
1494 }
1495 }
1496 }
1497 }
1498 }
1499 }
1500 }
1501 }
1502 }
1503 }
1504 }
1505 }
1506 }
1507 }
1508 }
1509 }
1510 }
1511 }
1512 }
1513 }
1514 }
1515 }
1516 }
1517 }
1518 }
1519 }
1520 }
1521 }
1522 }
1523 }
1524 }
1525 }
1526 }
1527 }
1528 }
1529 }
1530 }
1531 }
1532 }
1533 }
1534 }
1535 }
1536 }
1537 }
1538 }
1539 }
1540 }
1541 }
1542 }
1543 }
1544 }
1545 }
1546 }
1547 }
1548 }
1549 }
1550 }
1551 }
1552 }
1553 }
1554 }
1555 }
1556 }
1557 }
1558 }
1559 }
1560 }
1561 }
1562 }
1563 }
1564 }
1565 }
1566 }
1567 }
1568 }
1569 }
1570 }
1571 }
1572 }
1573 }
1574 }
1575 }
1576 }
1577 }
1578 }
1579 }
1580 }
1581 }
1582 }
1583 }
1584 }
1585 }
1586 }
1587 }
1588 }
1589 }
1590 }
1591 }
1592 }
1593 }
1594 }
1595 }
1596 }
1597 }
1598 }
1599 }
1600 }
1601 }
1602 }
1603 }
1604 }
1605 }
1606 }
1607 }
1608 }
1609 }
1610 }
1611 }
1612 }
1613 }
1614 }
1615 }
1616 }
1617 }
1618 }
1619 }
1620 }
1621 }
1622 }
1623 }
1624 }
1625 }
1626 }
1627 }
1628 }
1629 }
1630 }
1631 }
1632 }
1633 }
1634 }
1635 }
1636 }
1637 }
1638 }
1639 }
1640 }
1641 }
1642 }
1643 }
1644 }
1645 }
1646 }
1647 }
1648 }
1649 }
1650 }
1651 }
1652 }
1653 }
1654 }
1
```

```

101 // https://en.cppreference.com/w/cpp/string/basic/basic_stringbuf
102
103 line = string(buf);
104 endOfLine = string(line.find("\n"), endOfLine);
105 streamoff = string(line.find("stream", 1));
106
107 if endOfLine > 0 && (streamoff < 0 || streamoff > endOfLine) {
108     // no stream marker in buf detected.
109     break;
110 }
111
112 // For very rare cases where "stream" also occurs within obj dict
113 // (e.g. "stream" is a key in the dict), we need to find the
114 // first occurrence of "stream" after the endOfLine.
115 streamoff = 0;
116 if (auto it = find_if(streamMarker, obj.findEndOfLine(), streamoff);
117     !it.is_err()) {
118     logDebugPrint("buffer: offset: stream: ", streamoff, line);
119 }
120
121 if streamoff < 0 {
122     // streamOffset ... the offset where the actual stream data begins.
123     // It is right after the \n of last "stream".
124     streamoff = 0;
125 }
126
127 // max 32 bit unsigned whitespace + eof (max 2 chars)
128 slash = 32;
129 need = streamoff + (len(stream) * slash);
130
131 if (len(line) < need) {
132     // to prevent buffer overflow.
133     buf.eref = growBuf(buf, need-(len(line), rd)
134         if err == nil {
135             return nil, 0, 0, err
136         }
137     )
138     line = string(buf)
139     streamoff = len(line)-streamoff+(len(line), streamoff)
140 }
141
142 //Log-Debug-Print("buffer: end, returned buf: len: ", len(buf), len(buf),
143 //streamoff)
144
145 return buf, endOfLine, streamoff, streamOffset, nil
146 }
147
148 // returns true if "stream" follows end of dict: multi-line stream
149 func isStreamEnd(buf []byte, stream string, streamoff int) bool {
150     //Log-Debug-Print("isStreamEnd: stream: ", stream, "offset: ", streamoff)
151     // get a slice of the chunk right in front of "stream".
152     b := buf[streamoff:]
153     // look for last word of dict marker.
154     if !strings.LastIndex(b, ">>") {
155         return false
156     }
157     // no end of dict in buf.
158     return false
159 }

```



```

1570         }
1571         return false
1572     }
1573
1574     // We found the last zero (end1) just after end of dict only whitespace,
1575     ok = strings.TrimSpace(end1) == ">"
1576
1577     // Log Read.Printf("keywords:dictHeader:dictHeaderNDICT: end: %s\n", ok)
1578
1579     return ok
1580 }
1581
1582 func buildFilterPipeline(ctx *Context, filterArray, decodeParamsArr Array, decodeParams
1583 *dict *jioFilter *array) {
1584     var filterPipeline []jioFilter
1585
1586     for i, f := range filterArray {
1587
1588         filterName, ok := f.(Name)
1589         if !ok {
1590             corrupt := true, errors.New("pdcgo: buildFilterPipeline: filterArray elements
1591 corrupt")
1592             return corrupt
1593         }
1594         if decodeParams == nil || decodeParamsArr[i] == nil {
1595             filterPipeline = append(filterPipeline, jioFilter{Name:
1596 filterName, decodeParams: nil})
1597             continue
1598         }
1599         dict, ok := decodeParamsArr[i].(Dict)
1600         if !ok {
1601             corrupt := true, errors.New("pdcgo: buildFilterPipeline: dictHeader:dict
1602 dict")
1603             return corrupt, errors.Errorf("buildFilterPipeline: corrupt Dict: %s\n",
1604 dict)
1605         }
1606         if err := deferenceDict(dict, indirectObjectHeader.value()) {
1607             if err != nil {
1608                 return nil, err
1609             }
1610             dict = d
1611         }
1612     }
1613
1614     filterPipeline = append(filterPipeline, jioFilter{Name: filterName.String(),
1615 decodeParams: dict})
1616 }
1617
1618 return filterPipeline, nil
1619 }
1620
1621 // Begin the after pipeline associated with this source dict:
1622 func buildFilterPipeline2(*Context, dict Dict) (*jioFilter, error) {
1623
1624     log.Read.Printf("pdcfilterPipeline: begin")
1625
1626     var err error
1627
1628     o, found := dict.Find("filter")
1629     if !found {
1630         // stream is not compressed
1631     }

```

[illegible]

```

1130 // Save the saveDecodedContentStream to cksContent, sd, sdsReadIndex, objKey, govt, int,
1131 // err (or error)
1132
1133 // Log Read, Print("saveDecodedContentStream: begin decode\n"), decode)
1134
1135 // If the "isEmpty" crypt filter is used we do not need to decode.
1136 if cks == nil || cks.FilterKey == nil {
1137     // If sd is FilterPolicy(s), s = 1 do sd.FilterPolicy(s).Name = "Crypt"
1138     sd.Content = sd.Raw
1139     return nil
1140 }
1141
1142 // Log
1143
1144 // Special case: If the length of the encoded data is 0, we do not need to decode
1145 // anything.
1146 if (sd.Length() == 0) {
1147     sd.Content = sd.Raw
1148     return nil
1149 }
1150
1151 // cks gets created after objStream parsing.
1152 // objStream may not be encrypted.
1153 if cks == nil || cks.FilterKey == nil {
1154     // sd.Raw, obj = decryptRead(sd.Raw, objKey, govt, cks.FilterKey, cksAESStream,
1155     // cks.FK)
1156     if err == nil {
1157         return err
1158     }
1159     // If sd is nil, then
1160     // 1 = len(sd.Content)
1161     sd.Content.Length() = 0
1162     return nil
1163 }
1164
1165 // If decode
1166 // return nil
1167
1168 // Actual decoding of content stream.
1169 err = decodeContentStream()
1170 if err == filter.ErrUnsupportedFilter {
1171     // err = nil
1172     return nil
1173 }
1174
1175 if err == nil {
1176     return err
1177 }
1178
1179 // Log Read, Print("saveDecodedContentStream: end")
1180
1181 return nil
1182
1183 // Decode compressed objTableEntry
1184 func decodeCompressedObjTableEntry(
1185     obj: decompressedObjTableEntry, objTable *objTable, objStream int, entry
1186     *objStream, err error) error {
1187     // Log Read, Print("decodeCompressedObjTableEntry: compressed object sd at %d\n",
1188     // objStream, entry.objStream, entry.objStreamLen)
1189
1190     // Missing streamName entry in referenced object stream.
1191     objStreamLen = objTable[objStream].objStreamLen
1192     if objLen {

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry m, objId: %v", objId)
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs length 2 or 4", objId)
2043             }
2044             offset, ok = a[0].(Integer)
2045             if !ok {
2046                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objId)
2047             }
2048             offset4 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offsets4
2050         }
2051         if len(a) == 4 {
2052             if !
2053                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objId)
2054             ctx.OffsetOverluminTable = offsets4
2055         }
2056     }
2057     return nil
2058 }
2059
2060 func LoadLinearizationDict(ctx *Context, s *StreamReader, objId, genNr int) error {
2061     var err error
2062
2063     // Load stream's content and store data into offsetable entry
2064     if err = LoadLinearizationDictFromStreamData(s, objId, ctx); err != nil {
2065         return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d", objId)
2066     }
2067     ctx.Read.BinaryFileSize += s.GetSizeLength()
2068
2069     // Decode stream's content
2070     err = saveDecodedStreamContent(ctx, s, objId, genNr, ctx.DecodedAllStreams)
2071     return err
2072 }
2073
2074 func updateLinearizationDict(ctx *Context, o Object) {
2075     switch o := o.(type) {
2076     case StreamDict:
2077         ctx.Read.BinaryFileSize += o.GetSizeLength()
2078     }
2079 }

```

[illegible]

```

2487 d, err := differenceCdc(c1x, ifObjectNumber.Value())
2488 if err != nil {
2489     return err
2490 }
2491 log.Read.Printf("%s\n", d)
2492
2493 // We need to decrypt this file in order to read it.
2494 return setupEncryptionKey(c1x, d)
2495
2496

```

```

3420 // return nil, nil
3421 // 1426
3422 // 1426
3423 // compressed stream.
3424 // 1428
3425 var filterPipeline []PFFilter
3426 // 1431
3427 if indirOf, ok := o.Directref(ctx); ok {
3428     // 1432
3429     o, err = derefResourceDirect(ctx, indirOf.ObjectNumber.Value())
3430     // 1433
3431     if err != nil {
3432         // 1435
3433         return nil, err
3434     }
3435 // 1437
3436 // 1437
3437 // 1437
3438 // 1437
3439 // 1437
3440 // 1437
3441 // 1437
3442 // 1437
3443 // 1437
3444 // 1437
3445 // 1437
3446 // 1437
3447 // 1437
3448 // 1437
3449 // 1437
3450 // 1437
3451 // 1437
3452 // 1437
3453 // 1437
3454 // 1437
3455 // 1437
3456 // 1437
3457 // 1437
3458 // 1437
3459 // 1437
3460 // 1437
3461 // 1437
3462 // 1437
3463 // 1437
3464 // 1437
3465 // 1437
3466 // 1437
3467 // 1437
3468 // 1437
3469 // 1437
3470 // 1437
3471 // 1437
3472 // 1437
3473 // 1437
3474 // 1437
3475 // 1437
3476 // 1437
3477 // 1437
3478 // 1437
3479 // 1437
3480 // 1437
3481 // 1437
3482 // 1437
3483 // 1437
3484 // 1437
3485 // 1437
3486 // 1437
3487 // 1437
3488 // 1437
3489 // 1437
3490 // 1437
3491 // 1437
3492 // 1437
3493 // 1437
3494 // 1437
3495 // 1437
3496 // 1437
3497 // 1437
3498 // 1437
3499 // 1437
3500 // 1437
3501 // 1437
3502 // 1437
3503 // 1437
3504 // 1437
3505 // 1437
3506 // 1437
3507 // 1437
3508 // 1437
3509 // 1437
3510 // 1437
3511 // 1437
3512 // 1437
3513 // 1437
3514 // 1437
3515 // 1437
3516 // 1437
3517 // 1437
3518 // 1437
3519 // 1437
3520 // 1437
3521 // 1437
3522 // 1437
3523 // 1437
3524 // 1437
3525 // 1437
3526 // 1437
3527 // 1437
3528 // 1437
3529 // 1437
3530 // 1437
3531 // 1437
3532 // 1437
3533 // 1437
3534 // 1437
3535 // 1437
3536 // 1437
3537 // 1437
3538 // 1437
3539 // 1437
3540 // 1437
3541 // 1437
3542 // 1437
3543 // 1437
3544 // 1437
3545 // 1437
3546 // 1437
3547 // 1437
3548 // 1437
3549 // 1437
3550 // 1437
3551 // 1437
3552 // 1437
3553 // 1437
3554 // 1437
3555 // 1437
3556 // 1437
3557 // 1437
3558 // 1437
3559 // 1437
3560 // 1437
3561 // 1437
3562 // 1437
3563 // 1437
3564 // 1437
3565 // 1437
3566 // 1437
3567 // 1437
3568 // 1437
3569 // 1437
3570 // 1437
3571 // 1437
3572 // 1437
3573 // 1437
3574 // 1437
3575 // 1437
3576 // 1437
3577 // 1437
3578 // 1437
3579 // 1437
3580 // 1437
3581 // 1437
3582 // 1437
3583 // 1437
3584 // 1437
3585 // 1437
3586 // 1437
3587 // 1437
3588 // 1437
3589 // 1437
3590 // 1437
3591 // 1437
3592 // 1437
3593 // 1437
3594 // 1437
3595 // 1437
3596 // 1437
3597 // 1437
3598 // 1437
3599 // 1437
3600 // 1437
3601 // 1437
3602 // 1437
3603 // 1437
3604 // 1437
3605 // 1437
3606 // 1437
3607 // 1437
3608 // 1437
3609 // 1437
3610 // 1437
3611 // 1437
3612 // 1437
3613 // 1437
3614 // 1437
3615 // 1437
3616 // 1437
3617 // 1437
3618 // 1437
3619 // 1437
3620 // 1437
3621 // 1437
3622 // 1437
3623 // 1437
3624 // 1437
3625 // 1437
3626 // 1437
3627 // 1437
3628 // 1437
3629 // 1437
3630 // 1437
3631 // 1437
3632 // 1437
3633 // 1437
3634 // 1437
3635 // 1437
3636 // 1437
3637 // 1437
3638 // 1437
3639 // 1437
3640 // 1437
3641 // 1437
3642 // 1437
3643 // 1437
3644 // 1437
3645 // 1437
3646 // 1437
3647 // 1437
3648 // 1437
3649 // 1437
3650 // 1437
3651 // 1437
3652 // 1437
3653 // 1437
3654 // 1437
3655 // 1437
3656 // 1437
3657 // 1437
3658 // 1437
3659 // 1437
3660 // 1437
3661 // 1437
3662 // 1437
3663 // 1437
3664 // 1437
3665 // 1437
3666 // 1437
3667 // 1437
3668 // 1437
3669 // 1437
3670 // 1437
3671 // 1437
3672 // 1437
3673 // 1437
3674 // 1437
3675 // 1437
3676 // 1437
3677 // 1437
3678 // 1437
3679 // 1437
3680 // 1437
3681 // 1437
3682 // 1437
3683 // 1437
3684 // 1437
3685 // 1437
3686 // 1437
3687 // 1437
3688 // 1437
3689 // 1437
3690 // 1437
3691 // 1437
3692 // 1437
3693 // 1437
3694 // 1437
3695 // 1437
3696 // 1437
3697 // 1437
3698 // 1437
3699 // 1437
3700 // 1437
3701 // 1437
3702 // 1437
3703 // 1437
3704 // 1437
3705 // 1437
3706 // 1437
3707 // 1437
3708 // 1437
3709 // 1437
3710 // 1437
3711 // 1437
3712 // 1437
3713 // 1437
3714 // 1437
3715 // 1437
3716 // 1437
3717 // 1437
3718 // 1437
3719 // 1437
3720 // 1437
3721 // 1437
3722 // 1437
3723 // 1437
3724 // 1437
3725 // 1437
3726 // 1437
3727 // 1437
3728 // 1437
3729 // 1437
3730 // 1437
3731 // 1437
3732 // 1437
3733 // 1437
3734 // 1437
3735 // 1437
3736 // 1437
3737 // 1437
3738 // 1437
3739 // 1437
3740 // 1437
3741 // 1437
3742 // 1437
3743 // 1437
3744 // 1437
3745 // 1437
3746 // 1437
3747 // 1437
3748 // 1437
3749 // 1437
3750 // 1437
3751 // 1437
3752 // 1437
3753 // 1437
3754 // 1437
3755 // 1437
3756 // 1437
3757 // 1437
3758 // 1437
3759 // 1437
3760 // 1437
3761 // 1437
3762 // 1437
3763 // 1437
3764 // 1437
3765 // 1437
3766 // 1437
3767 // 1437
3768 // 1437
3769 // 1437
3770 // 1437
3771 // 1437
3772 // 1437
3773 // 1437
3774 // 1437
3775 // 1437
3776 // 1437
3777 // 1437
3778 // 1437
3779 // 1437
3780 // 1437
3781 // 1437
3782 // 1437
3783 // 1437
3784 // 1437
3785 // 1437
3786 // 1437
3787 // 1437
3788 // 1437
3789 // 1437
3790 // 143
```

```

3520 // test: (x:R)
3521 if err == nil {
3522     return nil, err
3523 }
3524 return StringLiteral(string(bb)), nil
3525 }
3526
3527 default:
3528     return o, nil
3529 }
3530 }
3531 }
3532
3533 func dereferenceObject(ctx *Context, objectNumber int) (Object, error) {
3534     entry, ok := cts.Find(objectNumber)
3535     if !ok {
3536         return nil, errors.New("p4cpu: dereferenceObject: unregistered object")
3537     }
3538     if entry.Compressed {
3539         err := decompressHeaderTableEntry(ctx, &entry.Table, objectNumber, entry)
3540         if err == nil {
3541             return entry, nil
3542         }
3543     }
3544     if entry.Object == nil {
3545         log.Bad.Printf("dereferenceObject: dereferencing object %d\n", objectNumber)
3546         o, err := ParseObject(ctx, entry.Offset, objectNumber, entry.Generation)
3547         if err == nil {
3548             return nil, errors.Wrap(err, "dereferenceObject: problem dereferencing object %d", objectNumber)
3549         }
3550     }
3551     if o == nil {
3552         return nil, errors.New("p4cpu: dereferenceObject: object is nil")
3553     }
3554     entry.Object = o
3555 }
3556 return entry.Object, nil
3557 }
3558
3559 func dereferenceInteger(ctx *Context, objectNumber int) (Integer, error) {
3560     o, err := dereferenceObject(ctx, objectNumber)
3561     if err == nil {
3562         return nil, err
3563     }
3564     i, ok := o.(Integer)
3565     if !ok {
3566         return nil, errors.New("p4cpu: dereferenceInteger: corrupt integer")
3567     }
3568 }

```

```

1573 // On return object's destructor may be called, problem dereferencing object
1574 stream.Md, no ref table entry, entry.ObjectStreamId)
1575
1576 //
1577 // Object of class entry has to be an ObjectStreamId
1578 //
1579 sd, obj = ObjectStreamId.ObjectStreamId(ObjectStreamId)
1580
1581 if !ok {
1582     return errors.Errorf("decompressRefTableEntry: problem dereferencing objectStreamMd, no object stream", entry.ObjectStreamId)
1583 }
1584
1585 //
1586 // Get IndexAndObject from ObjectStreamId
1587 //
1588 o, err = sd.IndexAndObject.ObjectStreamId()
1589 if err != nil {
1590     return errors.Errorf("decompressRefTableEntry: problem dereferencing object stream Md", entry.ObjectStreamId)
1591 }
1592
1593 // Save object to theRefTableEntry.
1594
1595 g := &entry.Object.o
1596 entry.Compression = g.Compression
1597 entry.Decompression = false
1598
1599 //
1600 // Load object's decompressRefTableEntry, end, obj MdId: %v\n",
1601 // entry.ObjectStreamId, entry.ObjectStreamId, o)
1602
1603 return nil
1604
1605 //
1606 // Log interesting stream content.
1607 //
1608 func LogStreamContent(i int) {
1609     switch o := o.(type) {
1610     case StreamId:
1611         if o.Content == nil {
1612             log.Printf("logStream no stream content")
1613         }
1614         if o.IsSpkgContent {
1615             //log.Printf("content %v\n", StreamId.Content)
1616         }
1617     case ObjectStreamId:
1618         if o.Content == nil {
1619             log.Printf("logStream no object stream content")
1620         }
1621         if o.IsSpkgContent {
1622             log.Printf("logStream no object stream content %v\n", o.Content)
1623         }
1624         if o.IsObjArray {
1625             log.Printf("logStream no object stream obj array")
1626         }
1627         if o.IsObjArray {
1628             log.Printf("logStream no object stream obj array %v\n", o.ObjArray)
1629         }
1630     }
1631 }
1632
1633 //
1634 // Default:

```

[illegible]

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObject(objs[ctx])
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shFileEntry object assign a object either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObject(objs[ctx])
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     err = decodeObject(objs[ctx])
2139     if err != nil {
2140         return err
2141     }
2142
2143     log.Root.Println("referenceCatalog: end")
2144
2145     return nil
2146 }
2147
2148 func handleEncryptedFile(ctxs *Context) error {
2149     // ctx.Cmd = DECRYPT
2150     if ctx.Cmd == DECRYPT || ctx.Cmd == SETPERMISSIONS {
2151         return errors.New("pfcpu: this file is not encrypted")
2152     }
2153
2154     if ctx.Cmd == DECRYPT {
2155         return nil
2156     }
2157
2158     // Encrypt subcommand found.
2159     // ctx.SubCmd = "e"
2160     return errors.New("pfcpu: please provide owner password and optional user
2161 password")
2162 }
2163
2164 func init() {
2165     return nil
2166 }
2167
2168 func initBytes(ctx *Context) (id []byte, err error) {
2169     if ctx.ID == nil {
2170         return nil, errors.New("pfcpu: missing ID entry")
2171     }
2172
2173     N1, ok := ctx.ID[0].(uint64)
2174     if ok {
2175         id, err = N1.Bytes()
2176         if err != nil {
2177             return nil, err
2178         }
2179     }
2180 }

```

```

1452 // If we have a stream object, found = dict.Fmt.DecodeParamArray
1453 if found != 0 {
1454     decodeParamArray, ok = decodeParamArray(Array)
1455     if !ok {
1456         return nil, errors.New("pdcip: pdfFilterPipeline: expected decodeParamArray corrupt")
1457     }
1458 }
1459
1460 // /xref.PdfDict("decodeParms: /s", dict.decodeParamArray)
1461
1462 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArray, decodeParamArray)
1463 if err != nil {
1464     log.Read.FilterPipeline("pdfFilterPipeline: err")
1465     return filterPipeline, err
1466 }
1467
1468 func streamDictForObjObject(c *Context, d Dict, objKey, streamIn int, streamOffset int) (
1469     streamLength, streamLengthOff = d.Length())
1470
1471 if streamIn < 0 {
1472     return sd, errors.New("pdcip: streamDictForObjObject: stream object without streamIn")
1473 }
1474
1475 filterPipeline, err = pdfFilterPipeline(c, d)
1476 if err == nil {
1477     return sd, err
1478 }
1479
1480 streamOffset = offset
1481
1482 // We have a stream object
1483 sd = NewStream(streamDict, streamOffset, streamLength, streamLengthOff, filterPipeline)
1484 log.Read.Filter("streamDictForObjObject: end, streamObject %d\n", objKey)
1485
1486 return sd, nil
1487
1488 func dictDict(c *Context, d Dict, objKey, err, endId, streamIn int) (d2 Dict, err
1489     error) {
1490     if c.Fmt.Encode == nil {
1491         return nil, errors.New("pdcip: dictDict: objKey, err, endId, streamIn")
1492     }
1493     if err == nil {
1494         return nil, err
1495     }
1496 }
1497
1498 if endId >= d.Length() && c.Streaming == c.Ended() {
1499     log.Read.PdfDict("dict: end, %d\n", objKey)
1500     d2 = d1
1501 }
1502

```

```

3730         return dc, nil
3731     }
3732 }
3733
3734 func dereferenceObject(cxt *Context, objectNumber int) (oic, error) {
3735     oic := dereferenceObject(cxt, objectNumber)
3736     if err == nil {
3737         return nil, err
3738     }
3739     if cxt != nil {
3740         dc, ok := oic.(Context)
3741         if !ok {
3742             return nil, errors.New("pdcpu: dereferenceObject: corrupt dict")
3743         }
3744     }
3745     return dc, nil
3746 }
3747
3748 // dereference a Message object representing an object value.
3749 func intObject(cxt *Context, objectNumber int) (uint64, error) {
3750     log.Read.Print("intObject begin: %d\n", objectNumber)
3751     oic := dereferenceObject(cxt, objectNumber)
3752     if err == nil {
3753         return nil, err
3754     }
3755     if cxt != nil {
3756         dc, ok := oic.(Context)
3757         if !ok {
3758             return 0, errors.New("pdcpu: intObject: corrupt dict")
3759         }
3760     }
3761     return dc, nil
3762 }
3763
3764 func id4(cxt *Context, objectNumber int) (uint64, error) {
3765     log.Read.Print("id4 begin: %d\n", objectNumber)
3766     oic := dereferenceObject(cxt, objectNumber)
3767     if err == nil {
3768         return 0, err
3769     }
3770     if cxt != nil {
3771         dc, ok := oic.(Context)
3772         if !ok {
3773             return 0, errors.New("pdcpu: id4: corrupt dict")
3774         }
3775     }
3776     return dc, nil
3777 }
3778
3779 // Reads and returns a file buffer with length = stream length using provided reader
3780 // positioned at offset.
3781 func readStreamStream(r io.Reader, streamLength int) ([]byte, error) {
3782     log.Read.Print("readStreamStream: begin streamLength:%d\n", streamLength)
3783     buf := make([]byte, streamLength)
3784     for totalCount := 0; totalCount < streamLength; {
3785         count, err := r.Read(buf[totalCount:])
3786         if err == nil {
3787             return nil, err
3788         }
3789         totalCount += count
3790     }
3791     log.Read.Print("readStreamStream: count=%d, bufLen=%d(x)%v", count,
3792         len(buf), buf[:count])
3793     return buf, nil
3794 }

```

```

130 // @see https://github.com/ericniebler/psutil/blob/master/psutil/_psutil_linux.c
131         log.Read.PrintIn("logStream: no objectsReady to copy")
132     }
133 }
134
135 // Decode all object streams to contained objects are ready to be used.
136 func decodeObjectStreams(ctxs []Context) error {
137     // @todo
138     // Entry "streams" intentionally left out.
139     // No object stream collection validation necessary.
140     log.Read.PrintIn("decodeObjectStreams: begin")
141
142     // Get sorted slice of object numbers.
143     objNums := ctxs[0].objNumbers
144     for k := range ctxs.Read.ObjectStreams {
145         keys := append(objNums, k)
146     }
147     sort.Ints(keys)
148
149     for _, objectNumber := range keys {
150         // @see StoreHeaderInfo.
151         entry := ctxs[0].table.Table(objectNumber)
152         if entry == nil {
153             return errors.Errorf("decodeObjectStreams: missing entry for objNum%d",
154                 objectNumber)
155         }
156         log.Read.PrintIn("decodeObjectStreams: parsing object stream for objNum%d",
157             objectNumber)
158
159         // Parse object stream from file.
160         o, err := ParseObjectStream(entry.Offset, objectNumber, entry.Generation)
161         if err != nil || o == nil {
162             return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
163         }
164
165         // Ensure streamObject
166         sd, ok := o.(StreamObject)
167         if !ok {
168             return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
169         }
170
171         if err := loadDecodedStreamContent(ctxs, sd); err != nil {
172             // @todo
173         }
174
175         // Load decoded stream content to storeTable.
176         if err := loadDecodedStreamContent(ctxs, sd); err != nil {
177             return errors.New("pdpcc: decodeObjectStreams: problem dereferencing
178                 object stream")
179         }
180     }
181
182     // Save decoded stream content to storeTable.
183     if err := SaveDecodedStreamContent(ctxs, sd, objectNumber, entry.Generation,
184         true); err != nil {
185         log.Read.PrintIn("obj %d %s", objectNumber, err)
186     }
187 }

```

```

2342 // err = ParseObject(ctxt, entry.Offset, objNr, entry.Generation)
2343 if err == nil {
2344     return errors.Wrapf(err, "dereferencedObject: problem dereferencing object %d", objNr)
2345 }
2346
2347 entry.Object = o
2348
2349 // // Linearization objects are validated and removed for stats only.
2350 err = handleLinearizationHash(ctxt, o, objNr)
2351 if err == nil {
2352     return err
2353 }
2354 // // handle stream dict.
2355 if _, ok = o.(StreamDict); ok {
2356     return errors.Errorf("dereferencedObject: object stream should already be dereferenced at objId=%d", objNr)
2357 }
2358
2359 if _, ok = o.(IndirectStreamDict); ok {
2360     return errors.Errorf("dereferencedObject: xref stream should already be dereferenced at objId=%d", objNr)
2361 }
2362
2363 if sd, ok = o.(StreamDict); ok {
2364     err = loadStream(ctxt, sd, objNr, entry.Generation)
2365     if err == nil {
2366         return err
2367     }
2368 }
2369
2370 entry.Object = sd
2371
2372 log.Root().Printf("dereferencedObject: and objId of %d\n", objNr,
2373     objNrDict, entry.Object)
2374
2375 logStream(entry.Object)
2376
2377 return nil
2378
2379 func processCountsOfCounts(refTable *XRefTable, D Dict) {
2380     for _, n := range o {
2381         match o1 := x.Table(n)
2382         case IndirectRef:
2383             entry, ok := refTable.LookupIndirectRefOfIndex(n)
2384             if ok {
2385                 entry.RefCount++
2386             }
2387         case Dict:
2388             processCountsOfCounts(refTable, o1)
2389         case Array:
2390             processCountsOfCounts(refTable, o1)
2391     }
2392 }

```

```

2370 }
2371 | else {
2372   id, ok := ctx.ID().ID(StringLiteral)
2373   if !ok {
2374     return nil, error.New("pdpoc: ID must contain hex literals or string
2375       literals");
2376   }
2377   id, err = Unescape(id.Value());
2378   if err != nil {
2379     return nil, err
2380   }
2381 }
2382
2383 return id, nil
2384
2385 func needsOwnerAndPasswd(cnd CommandNode) bool {
2386   return cnd == CHANGEOPT || cnd == CHANGEUSER || cnd == SETPERMISSIONS
2387 }
2388
2389 func handlePermissions(ctx *Context) error {
2390   // AE255 Validate permissions
2391   ok, err = validatePermissions(ctx)
2392   if err != nil {
2393     return err
2394   }
2395   if !ok {
2396     return errors.New("pdpoc: corrupted permissions after upw ok")
2397   }
2398   // Double check existing permissions for pdpoc processing.
2399   if hasWritePermissions(ctx.Cmd, Ctx) {
2400     return errors.New("pdpoc: insufficient access permissions")
2401   }
2402   return nil
2403 }
2404
2405 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2406   ctx.t, err = supportGetEncryption(ctx, d)
2407   if err != nil {
2408     return err
2409   }
2410   ctx.t.ID, err = idbytes(ctx)
2411   if err != nil {
2412     return err
2413   }
2414   var ok bool
2415   //fmt.Printf("pwpe: %s\n", ctx.t);
2416   // Validate the owner password sha_permissions/master_password.
2417   ok, err = ValidationPassword(Ctx)

```

[illegible]

```

3570 // Read stream content into the window stream content buffer size
3571 streamContent;
3572 // Load content to readContent context: Context, sd streamContent() [byte, error]
3573 load.ReadContent(readContentContext: Context, sd streamContent()) {
3574     log.Read.Print("LoadContent readContent: begin(wvLen)", wvLen)
3575 }
3576 var err = nil
3577 // Return saved decoded content.
3578 if sd.Raw == nil {
3579     // Read stream content encoded at stream with stream length.
3580     // Difference stream length if stream length is an indirect object.
3581     if sd.StreamLength == nil {
3582         if sd.StreamLength == nil {
3583             return nil, errors.New("pdfcpu: LoadContent readContent: missing streamLength")
3584         }
3585         // sd stream length from indirect object
3586         sd.StreamLength, err = int64(binary.BigEndian.Uint64(sd.StreamLengthObj))
3587         if err == nil {
3588             return nil, err
3589         }
3590     }
3591     log.Read.Print("LoadContent readContent: one indirect streamLengthObj",
3592         sd.StreamLengthObj)
3593 }
3594 // Read stream content
3595 readOffset := sd.StreamOffset
3596 rd, err := newBufferedReader(&Context, read, rd, readOffset)
3597 if err == nil {
3598     return nil, err
3599 }
3600 log.Read.Print("LoadContent readContent: seeked to offset: %d", readOffset)
3601 // Buffer stream content.
3602 readContentContext, err = readContentStream(rd, int64(sd.StreamLength))
3603 if err == nil {
3604     return nil, err
3605 }
3606 // Read stream content
3607 // Load readContentContext bufferLen(sha1Len), len(readContentContext),
3608 // readContentContext, sha1(readContentContext)
3609 // Set readContentContext
3610 sd.Raw = readContentContext
3611 log.Read.Print("LoadContent readContent: len(readContentContext): %d",
3612     len(sd.Raw))
3613 // Return decoded content.
3614 return readContentContext, nil
3615 }
3616 // Decode the raw encoded stream content and saves it to streamContent.Context.

```

[illegible]

```

2020: // 2020:
2021: func processArrayByCounts(x:Iterable, xObjTable, a Array) {
2022:     for _ in range a {
2023:         switch o in a {xObj} {
2024:             case IndexDef:
2025:                 entry, ok = xObjTable.FindTableEntryDef(o)
2026:                 if ok {
2027:                     entry.RefCount++
2028:                 }
2029:             case Cmt:
2030:                 processRefCounts(xObjTable, o)
2031:             case Array:
2032:                 processCounts(xObjTable, o)
2033:             }
2034:         }
2035:     }
2036: }
2037:
2038: func processRefCounts(xObjTable *XRefTable, o Object) {
2039:     switch o in o.(type) {
2040:     case Dict:
2041:         processDictCounts(xObjTable, o)
2042:     case StreamDict:
2043:         processDictCounts(xObjTable, o.Dict)
2044:     case Array:
2045:         processArrayByCounts(xObjTable, o)
2046:     }
2047: }
2048:
2049: // performance note: this function includes unprocessed objects from object streams.
2050: func deduplicateRefCounts(cxtx *Context) error {
2051:     log.Red.Println("deduplicateRefCounts: begin")
2052:     xRefTable := cxtx.XRefTable
2053:     // do not deduplicate counts of object numbers.
2054:     // TODO: skip sorting for performance gain.
2055:     var keys List
2056:     for k in xRefTable.Table {
2057:         keys = append(keys, k)
2058:     }
2059:     sort.Ints(keys)
2060:
2061:     for _ ,objNr := range keys {
2062:         err = deduplicateRefCounts(cxtx, objNr)
2063:         if err != nil {
2064:             return err
2065:         }
2066:     }
2067:
2068:     for _ ,objNr := range keys {
2069:         entry = xRefTable.Table[objNr]
2070:         if entry.ref != entry.compressed {
2071:             continue
2072:         }
2073:         processRefCounts(xRefTable, entry.obj.Secl)
2074:     }
2075: }

```

```

2530 //
2531 if err == nil {
2532     return err
2533 }
2534 //
2535 // If the owner password does not match we generally move on if the user password
2536 // errors.
2537 // Unless we need to limit on a user's owner password due to the specific
2538 // amount in use password.
2539 if tok != newPasswordHandler(password,ctx.Cmd) {
2540     return errors.New("password: please provide the master password with 'opw'")
2541 }
2542 //
2543 //
2544 // Generally the user password, which is also regarded as the master password or
2545 // pre-password.
2546 // If it is sufficient for moving on. A password change is an occasion since it
2547 // is not.
2548 if ok := newPasswordHandler(password,ctx.Cmd) {
2549     //
2550     ok,err := validatePermissions(ctx)
2551     if err == nil {
2552         return err
2553     }
2554     if ok {
2555         return errors.New("password: corrupted permissions after opw ok")
2556     }
2557     return nil
2558 }
2559 //
2560 // Validate the user password ok, document opw password.
2561 ok,err := validatePermissions(ctx)
2562 if err == nil {
2563     return err
2564 }
2565 if ok {
2566     return errors.New("password: please provide the correct password")
2567 }
2568 //
2569 //
2570 //
2571 //
2572 //
2573 //
2574 //
2575 //
2576 //
2577 //
2578 //
2579 //
2580 //
2581 //
2582 //
2583 //
2584 //
2585 //
2586 //
2587 //
2588 //
2589 //
2590 //
2591 //
2592 //
2593 //
2594 //
2595 //
2596 //
2597 //
2598 //
2599 //
2600 //
2601 //
2602 //
2603 //
2604 //
2605 //
2606 //
2607 //
2608 //
2609 //
2610 //
2611 //
2612 //
2613 //
2614 //
2615 //
2616 //
2617 //
2618 //
2619 //
2620 //
2621 //
2622 //
2623 //
2624 //
2625 //
2626 //
2627 //
2628 //
2629 //
2630 //
2631 //
2632 //
2633 //
2634 //
2635 //
2636 //
2637 //
2638 //
2639 //
2640 //
2641 //
2642 //
2643 //
2644 //
2645 //
2646 //
2647 //
2648 //
2649 //
2650 //
2651 //
2652 //
2653 //
2654 //
2655 //
2656 //
2657 //
2658 //
2659 //
2660 //
2661 //
2662 //
2663 //
2664 //
2665 //
2666 //
2667 //
2668 //
2669 //
2670 //
2671 //
2672 //
2673 //
2674 //
2675 //
2676 //
2677 //
2678 //
2679 //
2680 //
2681 //
2682 //
2683 //
2684 //
2685 //
2686 //
2687 //
2688 //
2689 //
2690 //
2691 //
2692 //
2693 //
2694 //
2695 //
2696 //
2697 //
2698 //
2699 //
2700 //
2701 //
2702 //
2703 //
2704 //
2705 //
2706 //
2707 //
2708 //
2709 //
2710 //
2711 //
2712 //
2713 //
2714 //
2715 //
2716 //
2717 //
2718 //
2719 //
2720 //
2721 //
2722 //
2723 //
2724 //
2725 //
2726 //
2727 //
2728 //
2729 //
2730 //
2731 //
2732 //
2733 //
2734 //
2735 //
2736 //
2737 //
2738 //
2739 //
2740 //
2741 //
2742 //
2743 //
2744 //
2745 //
2746 //
2747 //
2748 //
2749 //
2750 //
2751 //
2752 //
2753 //
2754 //
2755 //
2756 //
2757 //
2758 //
2759 //
2760 //
2761 //
2762 //
2763 //
2764 //
2765 //
2766 //
2767 //
2768 //
2769 //
2770 //
2771 //
2772 //
2773 //
2774 //
2775 //
2776 //
2777 //
2778 //
2779 //
2780 //
2781 //
2782 //
2783 //
2784 //
2785 //
2786 //
2787 //
2788 //
2789 //
2790 //
2791 //
2792 //
2793 //
2794 //
2795 //
2796 //
2797 //
2798 //
2799 //
2800 //
2801 //
2802 //
2803 //
2804 //
2805 //
2806 //
2807 //
2808 //
2809 //
2810 //
2811 //
2812 //
2813 //
2814 //
2815 //
2816 //
2817 //
2818 //
2819 //
2820 //
2821 //
2822 //
2823 //
2824 //
2825 //
2826 //
2827 //
2828 //
2829 //
2830 //
2831 //
2832 //
2833 //
2834 //
2835 //
2836 //
2837 //
2838 //
2839 //
2840 //
2841 //
2842 //
2843 //
2844 //
2845 //
2846 //
2847 //
2848 //
2849 //
2850 //
2851 //
2852 //
2853 //
2854 //
2855 //
2856 //
2857 //
2858 //
2859 //
2860 //
2861 //
2862 //
2863 //
2864 //
2865 //
2866 //
2867 //
2868 //
2869 //
2870 //
2871 //
2872 //
2873 //
2874 //
2875 //
2876 //
2877 //
2878 //
2879 //
2880 //
2881 //
2882 //
2883 //
2884 //
2885 //
2886 //
2887 //
2888 //
2889 //
2890 //
2891 //
2892 //
2893 //
2894 //
2895 //
2896 //
2897 //
2898 //
2899 //
2900 //
2901 //
2902 //
2903 //
2904 //
2905 //
2906 //
2907 //
2908 //
2909 //
2910 //
2911 //
2912 //
2913 //
2914 //
2915 //
2916 //
2917 //
2918 //
2919 //
2920 //
2921 //
2922 //
2923 //
2924 //
2925 //
2926 //
2927 //
2928 //
2929 //
2930 //
2931 //
2932 //
2933 //
2934 //
2935 //
2936 //
2937 //
2938 //
2939 //
2940 //
2941 //
2942 //
2943 //
2944 //
2945 //
2946 //
2947 //
2948 //
2949 //
2950 //
2951 //
2952 //
2953 //
2954 //
2955 //
2956 //
2957 //
2958 //
2959 //
2960 //
2961 //
2962 //
2963 //
2964 //
2965 //
2966 //
2967 //
2968 //
2969 //
2970 //
2971 //
2972 //
2973 //
2974 //
2975 //
2976 //
2977 //
2978 //
2979 //
2980 //
2981 //
2982 //
2983 //
2984 //
2985 //
2986 //
2987 //
2988 //
2989 //
2990 //
2991 //
2992 //
2993 //
2994 //
2995 //
2996 //
2997 //
2998 //
2999 //
3000 //
3001 //
3002 //
3003 //
3004 //
3005 //
3006 //
3007 //
3008 //
3009 //
3010 //
3011 //
3012 //
3013 //
3014 //
3015 //
3016 //
3017 //
3018 //
3019 //
3020 //
3021 //
3022 //
3023 //
3024 //
3025 //
3026 //
3027 //
3028 //
3029 //
3030 //
3031 //
3032 //
3033 //
3034 //
3035 //
3036 //
3037 //
3038 //
3039 //
3040 //
3041 //
3042 //
3043 //
3044 //
3045 //
3046 //
3047 //
3048 //
3049 //
3050 //
3051 //
3052 //
3053 //
3054 //
3055 //
3056 //
3057 //
3058 //
3059 //
3060 //
3061 //
3062 //
3063 //
3064 //
3065 //
3066 //
3067 //
3068 //
3069 //
3070 //
3071 //
3072 //
3073 //
3074 //
3075 //
3076 //
3077 //
3078 //
3079 //
3080 //
3081 //
3082 //
3083 //
3084 //
3085 //
3086 //
3087 //
3088 //
3089 //
3090 //
3091 //
3092 //
3093 //
3094 //
3095 //
3096 //
3097 //
3098 //
3099 //
3100 //
3101 //
3102 //
3103 //
3104 //
3105 //
3106 //
3107 //
3108 //
3109 //
3110 //
3111 //
3112 //
3113 //
3114 //
3115 //
3116 //
3117 //
3118 //
3119 //
3120 //
3121 //
3122 //
3123 //
3124 //
3125 //
3126 //
3127 //
3128 //
3129 //
3130 //
3131 //
3132 //
3133 //
3134 //
3135 //
3136 //
3137 //
3138 //
3139 //
3140 //
3141 //
3142 //
3143 //
3144 //
3145 //
3146 //
3147 //
3148 //
3149 //
3150 //
3151 //
3152 //
3153 //
3154 //
3155 //
3156 //
3157 //
3158 //
3159 //
3160 //
3161 //
3162 //
3163 //
3164 //
3165 //
3166 //
3167 //
3168 //
3169 //
3170 //
3171 //
3172 //
3173 //
3174 //
3175 //
3176 //
3177 //
3178 //
3179 //
3180 //
```