

```

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

```

```

2300         offset, err := strconv.ParseInt(fields[0], 10, 64)
2301         if err != nil {
2302             return err
2303         }
2304         generation, err := strconv.Atoi(fields[1])
2305         if err != nil {
2306             return err
2307         }
2308         entryType := fields[2]
2309         if entryType != "in use" || entryType != "in" {
2310             return errors.New(objects.parseableTableEntry: corrupt ref subobject
2311             entry)
2312         }
2313         var xrefTableEntry *xrefTableEntry
2314         if entryType == "in" {
2315             // in use object
2316             log.Debug.Print("parseableTableEntry: Object %d is in use at offset%0d,
2317             generation%0d", objNumber, offset, generation)
2318             if offset == 0 {
2319                 log.Info.Print("parseableTableEntry: Skip entry for in use object %d
2320                 with offset 0", objNumber)
2321                 return nil
2322             }
2323             xrefTableEntry =
2324                 &xrefTableEntry{
2325                     objNumber: objNumber,
2326                     offset: offset,
2327                     Generation: &generation}
2328         } else {
2329             // free object
2330             log.Debug.Print("parseableTableEntry: Object %d is unused, next free is
2331             object %d", objNumber, offset, generation)
2332             xrefTableEntry =
2333                 &xrefTableEntry{
2334                     free: true,
2335                     offset: offset,
2336                     Generation: &generation}
2337         }
2338         // free object
2339         log.Debug.Print("parseableTableEntry: Insert new xrefTable entry for Object %d",
2340         objNumber)
2341         xrefTable.Table[objNumber] = xrefTableEntry
2342         return nil
2343     }
2344     return err
2345 }

```

```

443 // https://github.com/GoogleCloudPlatform/google-cloud-go/blob/master/storage/googlecloudstorage.go#L100
444 for i := 0; i < len(objects); i++ {
445     objectNumber := xsd_objects[i]
446
447     // Read object
448     i32start := i + 1
449     c2 := bufToInt64(int32start + 1252start)
450     c3 := bufToInt64(int32start + 1252start + 12)
451
452     var xbfFileEntry xbfFileEntry
453
454     switch object {
455     case 0x00:
456         // Read object
457         log.Read_Print("object,xbfFileEntryFromRookFSStream: Object #0 is
458             \n, must get file c2,c3, generation\n", objectNumber, c2, c3)
459         s := int(c2)
460         xbfFileEntry =
461             xbfFileEntry{
462                 Free: true,
463                 Compressed: false,
464                 Offset: 8c2,
465                 Generation: 0s}
466     case 0x01:
467         // In use object
468         log.Read_Read("object,xbfFileEntryFromRookFSStream: Object #1 is in
469             use c2,c3, generation\n", objectNumber, c2, c3)
470         s := int(c2)
471         xbfFileEntry =
472             xbfFileEntry{
473                 Free: false,
474                 Compressed: false,
475                 Offset: 8c2,
476                 Generation: 0s}
477     case 0x02:
478         // Compressed object
479         log.Read_Read("object,xbfFileEntryFromRookFSStream: Object #2 is
480             compressed c2,c3, generation\n", objectNumber, c2, c3)
481         objNumberIn := int(c2)
482         objName := int(c3)
483         xbfFileEntry =
484             xbfFileEntry{
485                 Free: false,
486                 Compressed: true,
487                 ObjectStream: objNumberIn,
488                 ObjectStreamIndex: objName}
489     case ccs_Read_ObjectStream[objNumberIn] == true
490     }
491 }
492
493 if ctx.RbfFile.Exists(objectNumber) {
494     log.Read_Read("object,xbfFileEntryFromRookFSStream: skip entry M -

```

```

657         if desttable == null {
658             // return error: how? (perhaps: parameterInfoRef: missing entry "Root"? )
659             return errors.New("parameterInfoRef: missing entry \"Root\"")
660         }
661         xRefTable.Root = desttableRef
662         log.Debug.Printf("parameterInfoRef: Root object: %s\n", xRefTable.Root)
663     }
664     if xRefTable.Info == nil {
665         // infoRef := & indirectEntry("Info")
666         if infoRef := & indirectEntry("Info")
667         if infoRef != nil {
668             // xRefTable.Info = infoRefInfo
669             log.Debug.Printf("parameterInfoRef: Info object: %s\n", xRefTable.Info)
670         }
671     }
672     if xRefTable.ID == nil {
673         idEntry := & indirectEntry("ID")
674         if idEntry == nil {
675             // xRefTable.ID = idEntry
676             log.Debug.Printf("parameterInfoRef: ID object: %s\n", xRefTable.ID)
677         } else if xRefTable.IDEntry == nil {
678             // return errors.New("parameterInfoRef: missing entry \"ID\"")
679             return errors.New("parameterInfoRef: missing entry \"ID\"")
680         }
681     }
682     return errors.New("parameterInfoRef: end")
683 }
684
685 log.Debug.Printf("parameterInfoRef: end")
686
687 return nil
688 }
689
690 func parameterInfoRef(trailerDict Dict, ctx *Context) (xint64, error) {
691     // log.Debug.Printf("parameterInfoRef: begin")
692     log.Debug.Printf("parameterInfoRef: begin")
693     xRefTable = ctx.XRefTable
694     err = parameterInfoRef(trailerDict, xRefTable)
695     if err == nil {
696         return nil, err
697     }
698     if err == nil {
699         //
700     }
701     if err == trailerDict.ArrayEntry("AdditionalStreams"); err == nil {
702         log.Debug.Printf("parameterInfoRef: found AdditionalStreams: %s\n", err)
703         a := err
704         for a.value == range arr {
705             if infoRef := & value(indirectRef), a {
706                 a = append(a, indirect)
707             }
708         }
709         xRefTable.AdditionalStreams = &a
710     }
711     offset = trailerDict.Previous()
712     if offset == nil {
713         log.Debug.Printf("parameterInfoRef: previous xref table section offset: %s\n",
714             offset)
715     }
716     offset := trailerDict.InfoEntry("offset")
717 }

```

```

9870 // else if
9871 // log.Read.Printf("line %d\n", len(line), line)
9872 }
9873
9874 trailerString, err := scanTrailer(s, trailerString)
9875 if err == nil {
9876     return nil, err
9877 }
9878
9879 // log.Read.Printf("processTrailer: trailerDict: %v\n",
9880 // len(trailerString), trailerString)
9881
9882 // o, err := parseObject(trailerString)
9883 // if err == nil {
9884 //     return nil, err
9885 // }
9886
9887 trailerDict, ok := o.(Dict)
9888 if !ok {
9889     return nil, errors.New("pdpic: processTrailer: corrupt trailer dict")
9890 }
9891
9892 log.Read.Printf("processTrailer: trailerDict: %v\n", trailerDict)
9893
9894 return parseTrailerDict(trailerDict, ctx)
9895 }
9896
9897 // Parse sub section into corresponding number of sub table entries.
9898 func parseSubSection(s *bufio.Scanner, ctx *Content) (*uint64, error) {
9899     // log.Read.Printf("parseSubSection begin")
9900
9901     line, err := scanLine(s)
9902     if err == nil {
9903         return nil, err
9904     }
9905
9906     // log.Read.Printf("parseSubSection: %v\n", line)
9907
9908     fields := strings.Fields(line)
9909
9910     // Process all sub sections of this sub section
9911     for strings.HasPrefix(line, "trailer") && len(fields) == 2 {
9912         if err := parseSubTableSubSection(s, ctx.SubTable, fields); err == nil {
9913             // log.Read.Printf("subSection: %v\n", fields)
9914         }
9915     }
9916
9917     // trailer or another area table subsection
9918     if !line == "trailer" && len(fields) == 1 {
9919         return nil, err
9920     }
9921
9922     // if only line type next line for trailer
9923     if !line == "trailer" && len(fields) == 1 {
9924         if line, err := scanLine(s); err == nil {
9925             return nil, err
9926         }
9927     }
9928 }

```

```

1337 // to cxx.NewReader()
1338
1339 br, rdCount, err = reader.SeekStream(rs)
1340 if err != nil {
1341     return err
1342 }
1343
1344 ctx.NewReaderFrom = br
1345 ctx.ReadEdtCount = rdCount
1346
1347 for offset := nil {
1348     rd, err = newPositionedReader(ctx, offset)
1349     if err != nil {
1350         return err
1351     }
1352 }
1353
1354 s = bufio.NewScanner(rd)
1355 s.Split(func(lines)
1356
1357     line, err := s.Scan()
1358     if err != nil {
1359         return err
1360     }
1361
1362     log.Read.Printf("line: %s\n", line)
1363
1364     if strings.TrimSpace(line) == "ref" {
1365         log.Read.Printf("builderFragmentsStartingAt: found section")
1366         if offset, err = parseSection(ctx, line); err != nil {
1367             return err
1368         }
1369     } else {
1370         log.Read.Printf("builderFragmentsStartingAt: found error stream")
1371         log.Read.Printf("builderFragmentsStartingAt: error", err)
1372         if err, err = newPositionedReader(ctx, offset)
1373         if err != nil {
1374             return err
1375         }
1376         if offset, err = parseStream(rd, offset, ctx); err != nil {
1377             log.Read.Printf("builderFragmentsStartingAt: error", err)
1378             // try fix for corrupt input section.
1379             return hyposizeSection(ctx)
1380         }
1381         log.Read.Printf("builderFragmentsStartingAt: end")
1382     }
1383 }
1384
1385 log.Read.Printf("builderFragmentsStartingAt: end")
1386
1387 return nil
1388 }
1389
1390 // Populate the cross reference table for this PDF file.
1391 // Note offset of first ref table entry.
1392 // Can be "ref" or indirect object reference or "is a obj"
1393 // Note that the ref table contains as many as there is a defined previous ref section
1394 // and build up the ref table along the way.
1395 func readRefTable(cctx *Context) (err error) {

```

```

550 // =====
551 log.Read.Print("Read: begin!")
552
553 ctx, err := NewContexts, conf)
554 if err != nil {
555     return nil, err
556 }
557
558 if ctx.ReadOnly {
559     log.Info.Print("PDF Version 1.5 conforming reader")
560 } else {
561     log.Info.Print("PDF Version 1.4 conforming reader - no object streams &
562         references allowed")
563 }
564
565 // Populate shuffable
566 if err = readShuffleable(ctx); err != nil {
567     return nil, errors.New("Read: shuffleable failed")
568 }
569
570 // Make all objects explicitly available (load into memory) in corresponding
571 // shuffable entries.
572 // Also makes any involved object streams.
573 if err = dereferenceCrossTable(ctx, conf); err != nil {
574     return nil, err
575 }
576
577 // Some references write an invariant size into trailer.
578 // cctx.ShuffleableSize <= ctx.ShuffleableTable)
579 // cctx.ShuffleableSize <= len(ctx.ShuffleableTable)
580
581 log.Read.Print("Read: end")
582
583 return ctx, nil
584
585 // =====
586
587 // ScanLine is a multi-function for a Scanner that returns each line of
588 // text, stripped of any trailing end-of-line marker. The returned line may
589 // be empty, may contain any number of carriage returns followed
590 // by one or more line or one carriage return or one newline.
591 // If the line is empty, the returned line will be returned empty. If it is not,
592 // the line will be returned with the trailing carriage return or one newline.
593 func scanLines(data []byte, offset int64) (string, error) {
594     if ATEOF == len(data) == 0 {
595         return "", nil, nil
596     }
597
598     index := bytes.IndexByte(data, '\n')
599     indexL := bytes.IndexByte(data, '\r')
600
601     switch {
602     case index >= 0 && indexL >= 0:
603         if index < indexL {
604             if indexL == 0 {
605                 return indexL + 1, data[0:indexL], nil
606             }
607             // Both == -1
608         }
609     }
610 }

```

```

245
246
247 // Print out the object's location and create corresponding log entry within
248 func parseSubtableSubsection(subIn Scanner, subTable *SubTable, fields []string)
249 error {
250     log.Read.Println("parseSubtableSubsection: begin")
251
252     startObjNumber, err := stream.Atol(fields[0])
253     if err == nil {
254         return err
255     }
256
257     objCount, err := stream.Atol(fields[1])
258     if err == nil {
259         return err
260     }
261
262     log.Read.Println("detected err subsection, startObj=Obj length=ObjIn",
263         startObjNumber, objCount)
264
265     // Process all entries of this subsection into subtable entries
266     for i := 0; i < objCount; i++ {
267         if err := parseSubtableEntry(s, subTable, startObjNumber+i); err == nil {
268             return err
269         }
270     }
271
272     log.Read.Println("parseSubtableSubsection: end")
273
274     return nil
275 }
276
277 // Parse compressed object
278 func parseCompressedObject(s *string) (Object, error) {
279
280     log.Read.Println("parseCompressedObject: begin")
281
282     o, err := parseObject(s)
283     if err == nil {
284         return nil, err
285     }
286
287     d, ok := o.(Dict)
288     if !ok {
289         // return trivial Object: Integer, Array, etc.
290         log.Read.Println("compressedObject: end, any other than dict")
291         return o, nil
292     }
293
294     streamLength, streamLengthRef := d.Length()
295     if streamLength == nil || streamLengthRef == nil {
296         // return dict
297         log.Read.Println("compressedObject: end, dict")
298         return d, nil
299     }
300
301     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to
302         be stored in an object's stream")
303 }

```

```

347 // Create a new table entry
348 [stream assign(mo, objectNumber)
349   ] {
350     } else {
351       ct.table(objectNumber) = <objectTableEntry
352     }
353   }
354   }
355   }
356   }
357   }
358   }
359   }
360   }
361   }
362   }
363   }
364   }
365   }
366   }
367   }
368   }
369   }
370   }
371   }
372   }
373   }
374   }
375   }
376   }
377   }
378   }
379   }
380   }
381   }
382   }
383   }
384   }
385   }
386   }
387   }
388   }
389   }
390   }
391   }
392   }
393   }
394   }
395   }
396   }
397   }
398   }
399   }
400   }
401   }
402   }
403   }
404   }
405   }
406   }
407   }
408   }
409   }
410   }
411   }
412   }
413   }
414   }
415   }
416   }
417   }
418   }
419   }
420   }
421   }
422   }
423   }
424   }
425   }
426   }
427   }
428   }
429   }
430   }
431   }
432   }
433   }
434   }
435   }
436   }
437   }
438   }
439   }
440   }
441   }
442   }
443   }
444   }
445   }
446   }
447   }
448   }
449   }
450   }
451   }
452   }
453   }
454   }
455   }
456   }
457   }
458   }
459   }
460   }
461   }
462   }
463   }
464   }
465   }
466   }
467   }
468   }
469   }
470   }
471   }
472   }
473   }
474   }
475   }
476   }
477   }
478   }
479   }
480   }
481   }
482   }
483   }
484   }
485   }
486   }
487   }
488   }
489   }
490   }
491   }
492   }
493   }
494   }
495   }
496   }
497   }
498   }
499   }
500   }
501   }
502   }
503   }
504   }
505   }
506   }
507   }
508   }
509   }
510   }
511   }
512   }
513   }
514   }
515   }
516   }
517   }
518   }
519   }
520   }
521   }
522   }
523   }
524   }
525   }
526   }
527   }
528   }
529   }
530   }
531   }
532   }
533   }
534   }
535   }
536   }
537   }
538   }
539   }
540   }
541   }
542   }
543   }
544   }
545   }
546   }
547   }
548   }
549   }
550   }
551   }
552   }
553   }
554   }
555   }
556   }
557   }
558   }
559   }
560   }
561   }
562   }
563   }
564   }
565   }
566   }
567   }
568   }
569   }
570   }
571   }
572   }
573   }
574   }
575   }
576   }
577   }
578   }
579   }
580   }
581   }
582   }
583   }
584   }
585   }
586   }
587   }
588   }
589   }
590   }
591   }
592   }
593   }
594   }
595   }
596   }
597   }
598   }
599   }
600   }
601   }
602   }
603   }
604   }
605   }
606   }
607   }
608   }
609   }
610   }
611   }
612   }
613   }
614   }
615   }
616   }
617   }
618   }
619   }
620   }
621   }
622   }
623   }
624   }
625   }
626   }
627   }
628   }
629   }
630   }
631   }
632   }
633   }
634   }
635   }
636   }
637   }
638   }
639   }
640   }
641   }
642   }
643   }
644   }
645   }
646   }
647   }
648   }
649   }
650   }
651   }
652   }
653   }
654   }
655   }
656   }
657   }
658   }
659   }
660   }
661   }
662   }
663   }
664   }
665   }
666   }
667   }
668   }
669   }
670   }
671   }
672   }
673   }
674   }
675   }
676   }
677   }
678   }
679   }
680   }
681   }
682   }
683   }
684   }
685   }
686   }
687   }
688   }
689   }
690   }
691   }
692   }
693   }
694   }
695   }
696   }
697   }
698   }
699   }
700   }
701   }
702   }
703   }
704   }
705   }
706   }
707   }
708   }
709   }
710   }
711   }
712   }
713   }
714   }
715   }
716   }
717   }
718   }
719   }
720   }
721   }
722   }
723   }
724   }
725   }
726   }
727   }
728   }
729   }
730   }
731   }
732   }
733   }
734   }
735   }
736   }
737   }
738   }
739   }
740   }
741   }
742   }
743   }
744   }
745   }
746   }
747   }
748   }
749   }
750   }
751   }
752   }
753   }
754   }
755   }
756   }
757   }
758   }
759   }
760   }
761   }
762   }
763   }
764   }
765   }
766   }
767   }
768   }
769   }
770   }
771   }
772   }
773   }
774   }
775   }
776   }
777   }
778   }
779   }
780   }
781   }
782   }
783   }
784   }
785   }
786   }
787   }
788   }
789   }
790   }
791   }
792   }
793   }
794   }
795   }
796   }
797   }
798   }
799   }
800   }
801   }
802   }
803   }
804   }
805   }
806   }
807   }
808   }
809   }
810   }
811   }
812   }
813   }
814   }
815   }
816   }
817   }
818   }
819   }
820   }
821   }
822   }
823   }
824   }
825   }
826   }
827   }
828   }
829   }
830   }
831   }
832   }
833   }
834   }
835   }
836   }
837   }
838   }
839   }
840   }
841   }
842   }
843   }
844   }
845   }
846   }
847   }
848   }
849   }
850   }
851   }
852   }
853   }
854   }
855   }
856   }
857   }
858   }
859   }
860   }
861   }
862   }
863   }
864   }
865   }
866   }
867   }
868   }
869   }
870   }
871   }
872   }
873   }
874   }
875   }
876   }
877   }
878   }
879   }
880   }
881   }
882   }
883   }
884   }
885   }
886   }
887   }
888   }
889   }
890   }
891   }
892   }
893   }
894   }
895   }
896   }
897   }
898   }
899   }
900   }
901   }
902   }
903   }
904   }
905   }
906   }
907   }
908   }
909   }
910   }
911   }
912   }
913   }
914   }
915   }
916   }
917   }
918   }
919   }
920   }
921   }
922   }
923   }
924   }
925   }
926   }
927   }
928   }
929   }
930   }
931   }
932   }
933   }
934   }
935   }
936   }
937   }
938   }
939   }
940   }
941   }
942   }
943   }
944   }
945   }
946   }
947   }
948   }
949   }
950   }
951   }
952   }
953   }
954   }
955   }
956   }
957   }
958   }
959   }
960   }
961   }
962   }
963   }
964   }
965   }
966   }
967   }
968   }
969   }
970   }
971   }
972   }
973   }
974   }
975   }
976   }
977   }
978   }
979   }
980   }
981   }
982   }
983   }
984   }
985   }
986   }
987   }
988   }
989   }
990   }
991   }
992   }
993   }
994   }
995   }
996   }
997   }
998   }
999   }
1000  }

```

```

210 if offsetHexStream == nil {
211     // no cross reference stream.
212 }
213 if !ctx.readHex(0x00, func(file, version) { v := v & ctx.readHybrid(
214     return nil, errors.Errorf("parse(file=%s, version=%s) not a constant reader:
215 found incompatible version %s, fileName=%s", versionString())
216 }) {
217     log.Debug.Println("parse(file=%s) end")
218     return offset, nil
219 }
220 // This file is using cross reference streams.
221
222 if !ctx.readHybrid(
223     ctx.readHybrid & true
224     ctx.readShadingStream & true
225 ) {
226     // I/O constant readers process hidden objects contained
227     // in %shInfo before continuing to process any previous %shSection.
228     // Previous %shSection is expected to have free entries for hidden entries.
229     // No reason in %shSection only.
230     if !ctx.readHex(1) {
231         err = fmt.Errorf("readShadingStream of %s(%s, %s); err == nil %s",
232             return nil, errors
233         )
234     }
235     log.Debug.Println("parse(file=%s) end")
236     return offset, nil
237 }
238 // Return constant readers.
239 func scanIndexMap(%s, %s, %s) (string, error) {
240     if !ctx.readHex(0x00, func(file, version) {
241         if !ctx.readHex(0x00) == nil {
242             return "", S.Err()
243         }
244         return "", errors.New("pdfpc: scanIndexMap: returning nothing")
245     }) {
246         return s.Text(), nil
247     }
248 }
249
250 func scanIndex(%s, %s) (string, error) {
251     for i := 0; i <= i; i++ {
252         if !ctx.readHex(0x00, func(file, version) {
253             if !ctx.readHex(0x00) == nil {
254                 return "", S.Err()
255             }
256             if !ctx.readHex(0x00) > 0 {
257                 break
258             }
259         }) {
260             return "", errors
261         }
262         i := strings.Index(s, "%s")
263         if i > 0 {
264             s = s[:i]
265         }
266     }
267 }

```

```

960         fields = strings.Fields(line)
961     }
962 }
963
964 log.Read.Println("parseObjectSections: All subsections read!")
965
966 // Return a PdfHeader trailer.
967 func (m *Reader) trailer() {
968     if m.trailerNil, errors.Errorf("trailerSection missing trailer dict, line = %d",
969         Line), != nil {
970         return
971     }
972     log.Read.Println("parseObjectSections: parsing trailer dict.")
973
974     // Parse the trailer dictionary.
975     v, err := processTrailerDict(m, s, line)
976     if err != nil {
977         return
978     }
979
980     // Get version from first line of file.
981     // Beginning with PDF 1.4, the version entry in the document's catalog dictionary
982     // is optional. The PDF file may contain no "file" trailer, as described in 7.3.5, "File
983     // Trailer".
984     // If no version is found, we will assume the version specified in the header.
985     // Use PDF Version from header to start file.
986     // The header version comes as the first line of the file.
987     // Note that the number of header lines may not be 2.
988     func headerVersion() int {
989         v, err := m.headerVersion()
990         if err != nil {
991             return 0
992         }
993         return v
994     }
995     var errCodePdfHeader = errors.New("pdfdoc: headerVersion: corrupt pdf stream =
996     header version available")
997
998     // Get the version of the file which holds the version of this document.
999     // If we call this the header version.
1000     if err := m.seek(0, io.SeekStart), err == nil {
1001         buf := make([]byte, 8)
1002         s, err := m.read(buf)
1003         if err != nil {
1004             return nil, 0, err
1005         }
1006         if len(s) != 8 {
1007             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1008         }
1009         if len(s) != 8 {
1010             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1011         }
1012         if len(s) != 8 {
1013             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1014         }
1015         if len(s) != 8 {
1016             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1017         }
1018         if len(s) != 8 {
1019             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1020         }
1021         if len(s) != 8 {
1022             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1023         }
1024         if len(s) != 8 {
1025             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1026         }
1027         if len(s) != 8 {
1028             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1029         }
1030         if len(s) != 8 {
1031             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1032         }
1033         if len(s) != 8 {
1034             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1035         }
1036         if len(s) != 8 {
1037             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1038         }
1039         if len(s) != 8 {
1040             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1041         }
1042         if len(s) != 8 {
1043             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1044         }
1045         if len(s) != 8 {
1046             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1047         }
1048         if len(s) != 8 {
1049             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1050         }
1051         if len(s) != 8 {
1052             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1053         }
1054         if len(s) != 8 {
1055             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1056         }
1057         if len(s) != 8 {
1058             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1059         }
1060         if len(s) != 8 {
1061             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1062         }
1063         if len(s) != 8 {
1064             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1065         }
1066         if len(s) != 8 {
1067             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1068         }
1069         if len(s) != 8 {
1070             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1071         }
1072         if len(s) != 8 {
1073             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1074         }
1075         if len(s) != 8 {
1076             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1077         }
1078         if len(s) != 8 {
1079             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1080         }
1081         if len(s) != 8 {
1082             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1083         }
1084         if len(s) != 8 {
1085             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1086         }
1087         if len(s) != 8 {
1088             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1089         }
1090         if len(s) != 8 {
1091             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1092         }
1093         if len(s) != 8 {
1094             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1095         }
1096         if len(s) != 8 {
1097             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1098         }
1099         if len(s) != 8 {
1100             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1101         }
1102         if len(s) != 8 {
1103             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1104         }
1105         if len(s) != 8 {
1106             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1107         }
1108         if len(s) != 8 {
1109             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1110         }
1111         if len(s) != 8 {
1112             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1113         }
1114         if len(s) != 8 {
1115             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1116         }
1117         if len(s) != 8 {
1118             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1119         }
1120         if len(s) != 8 {
1121             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1122         }
1123         if len(s) != 8 {
1124             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1125         }
1126         if len(s) != 8 {
1127             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1128         }
1129         if len(s) != 8 {
1130             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1131         }
1132         if len(s) != 8 {
1133             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1134         }
1135         if len(s) != 8 {
1136             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1137         }
1138         if len(s) != 8 {
1139             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1140         }
1141         if len(s) != 8 {
1142             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1143         }
1144         if len(s) != 8 {
1145             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1146         }
1147         if len(s) != 8 {
1148             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1149         }
1150         if len(s) != 8 {
1151             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1152         }
1153         if len(s) != 8 {
1154             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1155         }
1156         if len(s) != 8 {
1157             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1158         }
1159         if len(s) != 8 {
1160             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1161         }
1162         if len(s) != 8 {
1163             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1164         }
1165         if len(s) != 8 {
1166             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1167         }
1168         if len(s) != 8 {
1169             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1170         }
1171         if len(s) != 8 {
1172             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1173         }
1174         if len(s) != 8 {
1175             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1176         }
1177         if len(s) != 8 {
1178             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1179         }
1180         if len(s) != 8 {
1181             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1182         }
1183         if len(s) != 8 {
1184             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1185         }
1186         if len(s) != 8 {
1187             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1188         }
1189         if len(s) != 8 {
1190             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1191         }
1192         if len(s) != 8 {
1193             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1194         }
1195         if len(s) != 8 {
1196             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1197         }
1198         if len(s) != 8 {
1199             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1200         }
1201         if len(s) != 8 {
1202             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1203         }
1204         if len(s) != 8 {
1205             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1206         }
1207         if len(s) != 8 {
1208             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1209         }
1210         if len(s) != 8 {
1211             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1212         }
1213         if len(s) != 8 {
1214             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1215         }
1216         if len(s) != 8 {
1217             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1218         }
1219         if len(s) != 8 {
1220             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1221         }
1222         if len(s) != 8 {
1223             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1224         }
1225         if len(s) != 8 {
1226             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1227         }
1228         if len(s) != 8 {
1229             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1230         }
1231         if len(s) != 8 {
1232             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1233         }
1234         if len(s) != 8 {
1235             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1236         }
1237         if len(s) != 8 {
1238             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1239         }
1240         if len(s) != 8 {
1241             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1242         }
1243         if len(s) != 8 {
1244             return nil, 0, errors.Errorf("trailerSection: corrupt pdf stream")
1245         }
1246         if len(s) != 8
```

```

1190 // Read the first section of the file
1191 log.Read_Printf("readFile: begin")
1192
1193 // Read the first section of the file
1194 offset, err = offsetLastFileSection(cta)
1195 if err != nil {
1196     return
1197 }
1198
1199 // Build the file starting at (cta, offset)
1200 if err != io.EOF {
1201     return errors.Wrapferr, "readFile: unexpected eof")
1202 }
1203
1204 // Read the first section of the file
1205 if err != nil {
1206     return
1207 }
1208
1209 // Log list of free objects for the "Free list"
1210 //log.Read_Printf("FreeList: %v", cta.FreeObjects)
1211
1212 // Ensure valid FreeList of objects.
1213 err, cta.EnsureValidFreeList()
1214 if err != nil {
1215     return
1216 }
1217
1218 log.Read_Printf("readFile: end")
1219
1220 return
1221 }
1222
1223 func (m *Memory) Read(buf []byte, size int, rd io.Reader) (lbyte, error) {
1224     // Read the first section of the file
1225     if err := m.Read(buf, size); err != nil {
1226         return 0, err
1227     }
1228     if err != nil {
1229         return 0, err
1230     }
1231     //log.Read_Printf("readFile: Read %d bytes", n)
1232     return append(buf, ...), nil
1233 }
1234
1235 func (m *Memory) ReadOffset(offset int, stream int) (off int) {
1236     off = stream + len("stream")
1237     // Read the first section of the file
1238     //log.Read_Printf("readFile: Read %d bytes", n)
1239     for i := 0; i < n; i++ {
1240         if i%100 == 0 {
1241             log.Printf("Read %d bytes", i)
1242         }
1243         if i%100 == 0 {
1244             log.Printf("Read %d bytes", i)
1245         }
1246         if i%100 == 0 {
1247             log.Printf("Read %d bytes", i)
1248         }
1249         if i%100 == 0 {
1250             log.Printf("Read %d bytes", i)
1251         }
1252         if i%100 == 0 {
1253             log.Printf("Read %d bytes", i)
1254         }
1255         if i%100 == 0 {
1256             log.Printf("Read %d bytes", i)
1257         }
1258         if i%100 == 0 {
1259             log.Printf("Read %d bytes", i)
1260         }
1261         if i%100 == 0 {
1262             log.Printf("Read %d bytes", i)
1263         }
1264         if i%100 == 0 {
1265             log.Printf("Read %d bytes", i)
1266         }
1267         if i%100 == 0 {
1268             log.Printf("Read %d bytes", i)
1269         }
1270         if i%100 == 0 {
1271             log.Printf("Read %d bytes", i)
1272         }
1273         if i%100 == 0 {
1274             log.Printf("Read %d bytes", i)
1275         }
1276         if i%100 == 0 {
1277             log.Printf("Read %d bytes", i)
1278         }
1279         if i%100 == 0 {
1280             log.Printf("Read %d bytes", i)
1281         }
1282         if i%100 == 0 {
1283             log.Printf("Read %d bytes", i)
1284         }
1285         if i%100 == 0 {
1286             log.Printf("Read %d bytes", i)
1287         }
1288         if i%100 == 0 {
1289             log.Printf("Read %d bytes", i)
1290         }
1291         if i%100 == 0 {
1292             log.Printf("Read %d bytes", i)
1293         }
1294         if i%100 == 0 {
1295             log.Printf("Read %d bytes", i)
1296         }
1297         if i%100 == 0 {
1298             log.Printf("Read %d bytes", i)
1299         }
1300         if i%100 == 0 {
1301             log.Printf("Read %d bytes", i)
1302         }
1303         if i%100 == 0 {
1304             log.Printf("Read %d bytes", i)
1305         }
1306         if i%100 == 0 {
1307             log.Printf("Read %d bytes", i)
1308         }
1309         if i%100 == 0 {
1310             log.Printf("Read %d bytes", i)
1311         }
1312         if i%100 == 0 {
1313             log.Printf("Read %d bytes", i)
1314         }
1315         if i%100 == 0 {
1316             log.Printf("Read %d bytes", i)
1317         }
1318         if i%100 == 0 {
1319             log.Printf("Read %d bytes", i)
1320         }
1321         if i%100 == 0 {
1322             log.Printf("Read %d bytes", i)
1323         }
1324         if i%100 == 0 {
1325             log.Printf("Read %d bytes", i)
1326         }
1327         if i%100 == 0 {
1328             log.Printf("Read %d bytes", i)
1329         }
1330         if i%100 == 0 {
1331             log.Printf("Read %d bytes", i)
1332         }
1333         if i%100 == 0 {
1334             log.Printf("Read %d bytes", i)
1335         }
1336         if i%100 == 0 {
1337             log.Printf("Read %d bytes", i)
1338         }
1339         if i%100 == 0 {
1340             log.Printf("Read %d bytes", i)
1341         }
1342         if i%100 == 0 {
1343             log.Printf("Read %d bytes", i)
1344         }
1345         if i%100 == 0 {
1346             log.Printf("Read %d bytes", i)
1347         }
1348         if i%100 == 0 {
1349             log.Printf("Read %d bytes", i)
1350         }
1351         if i%100 == 0 {
1352             log.Printf("Read %d bytes", i)
1353         }
1354         if i%100 == 0 {
1355             log.Printf("Read %d bytes", i)
1356         }
1357         if i%100 == 0 {
1358             log.Printf("Read %d bytes", i)
1359         }
1360         if i%100 == 0 {
1361             log.Printf("Read %d bytes", i)
1362         }
1363         if i%100 == 0 {
1364             log.Printf("Read %d bytes", i)
1365         }
1366         if i%100 == 0 {
1367             log.Printf("Read %d bytes", i)
1368         }
1369         if i%100 == 0 {
1370             log.Printf("Read %d bytes", i)
1371         }
1372         if i%100 == 0 {
1373             log.Printf("Read %d bytes", i)
1374         }
1375         if i%100 == 0 {
1376             log.Printf("Read %d bytes", i)
1377         }
1378         if i%100 == 0 {
1379             log.Printf("Read %d bytes", i)
1380         }
1381         if i%100 == 0 {
1382             log.Printf("Read %d bytes", i)
1383         }
1384         if i%100 == 0 {
1385             log.Printf("Read %d bytes", i)
1386         }
1387         if i%100 == 0 {
1388             log.Printf("Read %d bytes", i)
1389         }
1390         if i%100 == 0 {
1391             log.Printf("Read %d bytes", i)
1392         }
1393         if i%100 == 0 {
1394             log.Printf("Read %d bytes", i)
1395         }
1396         if i%100 == 0 {
1397             log.Printf("Read %d bytes", i)
1398         }
1399         if i%100 == 0 {
1400             log.Printf("Read %d bytes", i)
1401         }
1402         if i%100 == 0 {
1403             log.Printf("Read %d bytes", i)
1404         }
1405         if i%100 == 0 {
1406             log.Printf("Read %d bytes", i)
1407         }
1408         if i%100 == 0 {
1409             log.Printf("Read %d bytes", i)
1410         }
1411         if i%100 == 0 {
1412             log.Printf("Read %d bytes", i)
1413         }
1414         if i%100 == 0 {
1415             log.Printf("Read %d bytes", i)
1416         }
1417         if i%100 == 0 {
1418             log.Printf("Read %d bytes", i)
1419         }
1420         if i%100 == 0 {
1421             log.Printf("Read %d bytes", i)
1422         }
1423         if i%100 == 0 {
1424             log.Printf("Read %d bytes", i)
1425         }
1426         if i%100 == 0 {
1427             log.Printf("Read %d bytes", i)
1428         }
1429         if i%100 == 0 {
1430             log.Printf("Read %d bytes", i)
1431         }
1432         if i%100 == 0 {
1433             log.Printf("Read %d bytes", i)
1434         }
1435         if i%100 == 0 {
1436             log.Printf("Read %d bytes", i)
1437         }
1438         if i%100 == 0 {
1439             log.Printf("Read %d bytes", i)
1440         }
1441         if i%100 == 0 {
1442             log.Printf("Read %d bytes", i)
1443         }
1444         if i%100 == 0 {
1445             log.Printf("Read %d bytes", i)
1446         }
1447         if i%100 == 0 {
1448             log.Printf("Read %d bytes", i)
1449         }
1450         if i%100 == 0 {
1451             log.Printf("Read %d bytes", i)
1452         }
1453         if i%100 == 0 {
1454             log.Printf("Read %d bytes", i)
1455         }
1456         if i%100 == 0 {
1457             log.Printf("Read %d bytes", i)
1458         }
1459         if i%100 == 0 {
1460             log.Printf("Read %d bytes", i)
1461         }
1462         if i%100 == 0 {
1463             log.Printf("Read %d bytes", i)
1464         }
1465         if i%100 == 0 {
1466             log.Printf("Read %d bytes", i)
1467         }
1468         if i%100 == 0 {
1469             log.Printf("Read %d bytes", i)
1470         }
1471         if i%100 == 0 {
1472             log.Printf("Read %d bytes", i)
1473         }
1474         if i%100 == 0 {
1475             log.Printf("Read %d bytes", i)
1476         }
1477         if i%100 == 0 {
1478             log.Printf("Read %d bytes", i)
1479         }
1480         if i%100 == 0 {
1481             log.Printf("Read %d bytes", i)
1482         }
1483         if i%100 == 0 {
1484             log.Printf("Read %d bytes", i)
1485         }
1486         if i%100 == 0 {
1487             log.Printf("Read %d bytes", i)
1488         }
1489         if i%100 == 0 {
1490             log.Printf("Read %d bytes", i)
1491         }
1492         if i%100 == 0 {
1493             log.Printf("Read %d bytes", i)
1494         }
1495         if i%100 == 0 {
1496             log.Printf("Read %d bytes", i)
1497         }
1498         if i%100 == 0 {
1499             log.Printf("Read %d bytes", i)
1500         }
1501         if i%100 == 0 {
1502             log.Printf("Read %d bytes", i)
1503         }
1504         if i%100 == 0 {
1505             log.Printf("Read %d bytes", i)
1506         }
1507         if i%100 == 0 {
1508             log.Printf("Read %d bytes", i)
1509         }
1510         if i%100 ==
```

```

315         return index < 1, data[index], nil
316     }
317 }
318 // debug - debug
319 return index < 1, data[index], nil
320 }
321 case index < 0:
322     // We have a full carriage return terminated line.
323     return index + 1, data[index], nil
324 }
325 case index < 0:
326     // We have a full newline-terminated line.
327     return index + 1, data[index], nil
328 }
329 }
330 // If we're at EOF, we have a final, non-terminated line. Return it.
331 if atEOF {
332     return len(data), data, nil
333 }
334 return len(data), data, nil
335 }
336 // Request more data.
337 return 0, nil, nil
338 }
339
340 func newOffsetReader(rs io.Reader, offset int64) (*bufio.Reader, error) {
341     if rs == rs.Seek(offset, io.SeekStart), err == nil {
342         // If we're at EOF, we have a final, non-terminated line. Return it.
343         return nil, err
344     }
345     log.Read.Print("Scanning for offsetLineSection: positioned to offset: %d\n", offset)
346     return bufio.NewReader(rs), nil
347 }
348
349 // Get the file offset of the last offsetSection.
350 // Get the file and search backwards for the first occurrence of startLine
351 // offset
352 func findLastOffsetSection(ctx *Context) (int64, error) {
353     rs := ctx.Read.Rs
354     var {
355         prevBuf, worked, bufSize,
356         bufSize, int64 = 512
357         offset,
358     }
359     for i := 1; offset > 0; i = {
360         // If we're at rs.Seek(-int64(i)+bufSize, io.SeekEnd)
361         if, err := rs.Seek(-int64(i)+bufSize, io.SeekEnd)
362         if err == nil {
363             return nil, errors.New("previous can't find last offset section")
364         }
365         log.Read.Print("Scanning for offsetLineSection starting at %d\n", offset)
366         curBuf := make([]byte, bufSize)
367     }
368 }

```

```

340 // @ts-ignore
341 // If we call obj instanceof an object stream we have fun, but into objectStreamIdc()
342 func parObjStreamIdc() objectStreamIdc error {
343     logDecompressPrint("parObjStreamIdc begin: decoding hd object:\n", obj, objCount)
344     decompressContent()
345     modObj := decompressContent().objStreamIdc()
346     obj := strings.Fields(string(modObj))
347     if len(obj) % 2 != 0 {
348         return errors.New("pdcpu: parObjStreamIdc corrupt object stream idc")
349     }
350     // e.g., 10 8 11 25 = 2 Objects: 810 at offset 0, 111 at offset 25
351     var objArray Array
352     var offsetIdc int
353     for i := 0; i < len(obj); i += 2 {
354         offset, err := strconv.Atoi(obj[i+1])
355         if err != nil {
356             return err
357         }
358         offset += objStreamIdcOffset
359         if obj[i] != '1' {
360             dst := string(decompressContent().offsetIdc)
361             logDecompressPrint("parObjStreamIdc objectStreamIdc objStreamIdc = %s\n", dst)
362             obj, err := compressObject(dst)
363             if err != nil {
364                 return err
365             }
366             logDecompressPrint("parObjStreamIdc [hd] = obj %s\n", obj, i/2, objCount)
367             objArray = append(objArray, obj)
368         }
369         if i == len(obj) - 1 {
370             dst := string(decompressContent().offsetIdc)
371             logDecompressPrint("parObjStreamIdc objectStreamIdc objStreamIdc = %s\n", dst)
372             obj, err := compressObject(dst)
373             if err != nil {
374                 return err
375             }
376             logDecompressPrint("parObjStreamIdc [hd] = obj %s\n", obj, i/2, objCount)
377             objArray = append(objArray, obj)
378         }
379         offsetIdc = offset
380     }
381 }

```

```

630         }
631         return nil, err
632     }
633 }
634
635 // ReadStream reads a stream from the given streamID.
636 func (s *Server) ReadStream(streamID int32) (streamInfo, io.Reader) {
637     log.Debug.Printf("parseRefStream: endInfo=%d[id=%d]", streamID, s2b(streamID))
638     log.Debug.Printf("id = %v\n", buf)
639 }
640
641 // We expect a stream and therefore "stream" before "endobj" if "endobj" within
642 // the stream.
643 // There is no guarantee that "endobj" is contained in this buffer for large
644 // streams.
645 func (s *Server) ReadStream(streamID int32) (streamInfo, io.Reader) {
646     return nil, errors.New("pdfcpu: parseRefStream: corrupt pdf file")
647 }
648
649 // Init object array buf
650 func (s *Server) InitObjectArrayBuf() {
651     l := lineStream(streamID)
652     objectNumber, generationNumber, err := parseObjectAttributes(l)
653     if err != nil {
654         return nil, err
655     }
656 }
657
658 // Parse object array buf
659 func (s *Server) ParseObjectArrayBuf(objectNumber int, generationNumber int) {
660     log.Debug.Printf("parseRefStream: xrefInfo=%d genNum=%d, objectNumber=%d",
661         objectNumber, generationNumber)
662     log.Debug.Printf("parseRefStream: referencing object %d\n", objectNumber)
663     o, err := parseObject(l)
664     if err != nil {
665         return nil, errors.Wrap(err, "parseRefStream: no object")
666     }
667 }
668
669 // Parse object array buf
670 func (s *Server) ParseObjectArrayBuf(objectNumber int, generationNumber int) {
671     log.Debug.Printf("parseRefStream: we have an object: %d\n", o)
672 }
673
674 // Parse object array buf
675 func (s *Server) ParseObjectArrayBuf(objectNumber int, generationNumber int) {
676     streamOffset := -offset
677     obj := objectStream(streamID, o, objectNumber, streamOffset)
678     if err := nil {
679         return nil, err
680     }
681 }
682
683 // We have an end stream object
684 func (s *Server) ParseObjectArrayBuf(objectNumber int, generationNumber int) {
685     err := parseRefInfo(bufInfo(s, ctx.XRefTable))
686     if err != nil {
687         return nil, err
688     }
689 }
690
691 // Parse object array buf
692 func (s *Server) ParseObjectArrayBuf(objectNumber int, generationNumber int) {
693     err := extractObjectStreamContent(streamID, content, s, ctx)
694     if err != nil {
695         return nil, err
696     }
697 }
698
699 // Create object array buf
700 func (s *Server) CreateObjectArrayBuf(objectNumber int, generationNumber int) {
701     entry =
702         XRefTableEntry{
703             Free: false,
704             Offset: offset,
705             Generation: generationNumber,
706             Object: obj
707         }
708 }

```

```

780         return s1.nil
781     }
782     func isDict(s string) (bool, error) {
783         ok, err = parseDict(s)
784         if err == nil {
785             return false, err
786         }
787         ok, err = o.Dict(s)
788         return ok, nil
789     }
790     func scanHeader(s bufio.Scanner, line string) (string, error) {
791         var buf bytes.Buffer
792         var err error
793         var i32 int32
794         var i32s int32
795         buf.WriteString(fmt.Sprintf("line: %s\n", line))
796         // Scan for dict start tag "\n".
797         for {
798             i32 = strings.Index(line, "\n")
799             if i32 >= 0 {
800                 break
801             }
802             line, err = scanLine(s)
803             buf.WriteString(fmt.Sprintf("line: %s\n", line))
804             if err == nil {
805                 return "", err
806             }
807         }
808     }
809     line := line[s]
810     buf.WriteString(line)
811     buf.WriteString("\n")
812     buf.WriteString(fmt.Sprintf("scanner dictbuf after start tag: %s\n", line))
813     // Scan for dict tag "\n" but account for inner dicts.
814     line = line[2:]
815     for {
816         if len(line) == 0 {
817             line, err = scanLine(s)
818             if err == nil {
819                 return "", err
820             }
821             buf.WriteString(line)
822             buf.WriteString("\n")
823             buf.WriteString(fmt.Sprintf("scanner dictbuf next line: %s\n", line))
824         }
825         i32 = strings.Index(line, "\n")
826         if i32 <= 0 {
827             i32 = strings.Index(line, "\n")
828             if i32 >= 0 {

```

[illegible]

```

255 if (line[offset] == '\r')
256     offset++;
257 // Valid lines only.
258 // If line[offset] == '\n'
259     offset++;
260 }
261 }
262
263 return
264 }
265
266 // Use lastStreamMarker (streamed = limit, endline, line, string) {
267
268     if (streamed > len(line[lineStreamMarker]) {
269         // We move to another stream marker.
270         streamlined = -1;
271         return
272     }
273
274 // We start searching after this stream marker.
275 bufpos = streamlined + len('stream')
276
277 // Search for next stream marker.
278 // In strings: Index(line[offset], "stream")
279 if (i < 0)
280     // We search marker within line buffer.
281     streamlined = -1;
282     return
283
284 // We find the next stream marker.
285 streamlined = len('stream') + 1
286
287 // We find the next stream marker.
288 streamlined = len('stream') + 1
289
290 if (ending > 0) do streamlined >= ending {
291     // We find a stream marker of another object
292     streamlined = -1;
293     return
294 }
295
296 // process = PDF file buffer of sufficient size for parsing an object. > stream
297 // endline = in Reader) buf[byte, endline len, stream len, streamOffset (index
298 // error)]
299
300 // process: a gun obj ... obj dict ... stream ... data ... endstream ... endobj
301 // stream ... stream ...
302 // -1 if absent -1 if absent
303 absent
304
305 //Log.Debug.WriteLine("buffer: begin")
306
307 endline, streamlined = -1, -1
308
309 for (endline < 0) do streamlined < 0 {
310     buf, err = g.Read(buf, defaultBufferSize, rd)
311     if err == nil
312         return nil, 0, 0, 0, err
313 }

```

```

363 // https://stackoverflow.com/questions/40480000/using-std-weak-map
374
375         .. err = r.Read(cursor)
376         if err == nil {
377             return nil, err
378         }
379     }
380
381     workBuf = curBuf
382     if preBuf == nil {
383         workBuf = append(curBuf, preBuf...)
384     }
385
386     j := strings.LastIndex(string(workBuf), "startref")
387     if j == -1 {
388         preBuf = curBuf
389         continue
390     }
391
392     p = workBuf[j+1len(string(workBuf)):]
393     posOff := strings.Index(string(p), "MEOF")
394     if posOff == -1 {
395         return nil, errors.New("pfcpu: no matching MEOF for startref")
396     }
397
398     p = p[posOff:]
399     offset, err := strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
400     if err == nil {
401         return nil, errors.New("pfcpu: corrupted last ref section")
402     }
403 }
404
405 log.Read.Printf("Offset last ref section: %d\n", offset)
406
407 return bufOffset, nil
408 }
409
410 // Read next subsection entry and generate corresponding ref table entry.
411 func (parser *ParserTableEntry) sbufIoScanner, sbufIo *sbufIo, objectNumber int) {
412     log.Read.Printf("parser:TableEntry: begin")
413
414     line, err = scanner()
415     if err == nil {
416         return nil, err
417     }
418
419     if sbufIo.Exists(objectNumber) {
420         log.Read.Printf("parser:TableEntry: end - Skip entry %d - already assigned", objectNumber)
421         return nil, nil
422     }
423
424     fields = strings.Split(line)
425     if len(fields) == 1 {
426         log.Read.Printf("parser:TableEntry: end - Skip entry %d - already assigned", objectNumber)
427         return errors.New("pfcpu: parser:TableEntry: corrupt ref subsection")
428     }
429 }

```

```

380 // @ts-ignore
391 @std.obufArray = obfArray
392
393 log.Read_Printf("params:objectstream\n")
394
395 return nil
396
397 // for each object embedded in this objectStream create the corresponding vdef table
398 // that shall be present in the stream
399 func (this *ObjectStream) createVdefTables() {
400     // @ts-ignore
401     log.Read_Printf("params:objectstream\n")
402
403     log.Read_Printf("extract:objectfile:entriesFromObjectStream begin")
404
405     // Note:
406     // * A value of zero for an element in the m array indicates that the corresponding
407     //   element shall not be present in the stream.
408     // * The default value shall be zero, if there is one.
409     // * If an element is zero, the type field shall not be present, and shall
410     //   default to type: 0
411
412     // @ts-ignore
413     m := xsd.M{
414         11: xsd.M{0},
415         12: xsd.M{1},
416         13: xsd.M{2},
417     }
418
419     xdefEntryLen = "11 * 12 + 13"
420
421     log.Read_Printf("extract:objectfile:entriesFromObjectStream: begin xdefEntryLen = %s",
422         xdefEntryLen)
423
424     if len(buf) > xdefEntryLen + 8 {
425         return errors.New("pdcip: extract:objectfile:entriesFromObjectStream: corrupt
426             refstream")
427     }
428
429     obfCount = len(xsd.Objects)
430
431     log.Read_Printf("extract:objectfile:entriesFromObjectStream: obfCount %d bufLen",
432         obfCount, xsd.Objects)
433
434     log.Read_Printf("extract:objectfile:entriesFromObjectStream: len(buf):%d",
435         len(buf))
436
437     if len(buf) < obfCount + xdefEntryLen {
438         // Sometimes there is an additional zero entry not accounted for by "index".
439         // This is not a problem, but it is not clear if this is an error.
440         return errors.New("pdcip: extract:objectfile:entriesFromObjectStream: corrupt
441             refstream")
442     }
443
444     // @ts-ignore
445     j := 0
446
447     // bufToIndex interprets the content of buf as an index.
448     // bufToIndex = func(buf []byte) (int) {
449         for i := range buf {
450             // @ts-ignore
451             i |= int(buf[i])
452         }
453     }
454
455     // @ts-ignore
456     }
457
458 }

```

```

645 log.ReadPrintln("parseHeader: Insert new shFileable entry for Object %d\n",
646 objId, objName);
647
648 ctx.Table<objId>name> = Entry
649 (ctx, new parseHeaderShFileable(objId)); // true
650 prevOffset = id.PrevioalOffset
651
652 log.ReadPrintln("parseHeaderStream: end")
653
654 return prevOffset, nil
655
656 // =====
657 // Parse an shFileable as a typical PDF file.
658 func parseHeaderShFileableStream(offset int64, ctx Context) error {
659
660     log.ReadPrintln("parseHeaderStream: begin")
661
662     rd, err := newPositionalReader(ctx, offset)
663     if err != nil
664         return err
665     }
666
667     // err = parseHeaderStream(rd, offset, ctx)
668     if err != nil
669         return err
670     }
671
672     log.ReadPrintln("parseHeaderStream: end")
673
674 return nil
675
676 // =====
677 // Parse trailer dict and return any offset of a previous shFileable.
678 func parseTrailerShFileable(Dict, shFileable shFileable) error {
679
680     log.ReadPrintln("parseTrailerFile begin")
681
682     if _, found = dict.FindEntry("Root"); found {
683         if encryptObj := dict["Root"].(*dictEntry).value.(dictEntry)
684             if encryptObj != nil {
685                 shFileable.decrypt = decryptObj.dict["shFileable"]
686                 log.ReadPrintln("parseTrailerFile: Encrypt object: %d\n",
687 shFileable.encrypt)
688             }
689     }
690
691     // =====
692     if shFileable.Size == nil {
693         size = d.GetSize()
694     }
695     if size != nil {
696         return errors.New("pdfproc: parseTrailerFile: missing entry \"%s\"")
697     }
698     // Not reliable
699     // /atches after all read in.
700     shFileable.Size = size
701
702     // =====
703     if shFileable.Root == nil {
704         rootObjId = d.IndirectRefEntry("Root")
705     }
706 }

```

```

340 //
341 // If k == 0
342 //
343 // Check for err
344 //
345 ok_err = bufio.ReadString()
346 //
347 if err == nil || ok {
348     return buf.String(), nil
349 }
350 //
351 } else {
352     k++
353 }
354 //
355 // line = line[j+2]
356 //
357 continue
358 //
359 // No go
360 //
361 line, err = scanner(s)
362 //
363 if err == nil {
364     return "", err
365 }
366 //
367 bufio.WriteString(line)
368 bufio.WriteString(" ")
369 log.Reads.Printf("scanner failed to find next line: %s\n", line)
370 //
371 } else {
372     //
373     // %v %s %f
374     //
375 s = string.Index(line, "\n")
376 //
377 // k++
378 //
379 // line = line[j+2]
380 //
381 } else {
382     //
383     // %v %s %f
384     //
385 if j < 3 {
386     //
387     // handle ok
388     //
389     line = line[j+2]
390 } else {
391     //
392     // handle go
393     //
394     if k == 0 {
395         //
396         // Check for dict
397         //
398         ok_err = bufio.ReadString()
399         //
400         if err == nil || ok {
401             return buf.String(), nil
402         }
403     } else {
404         k++
405     }
406     //
407     line = line[j+2]
408 }
409 //
410 }
411 //
412 }
413 //
414 }
415 //
416 }
417 //
418 }
419 //
420 }
421 //
422 }
423 //
424 }
425 //
426 }
427 //
428 }
429 //
430 }
431 //
432 }
433 //
434 }
435 //
436 }
437 //
438 }
439 //
440 }
441 //
442 }
443 //
444 }
445 //
446 }
447 //
448 }
449 //
450 }
451 //
452 }
453 //
454 }
455 //
456 }
457 //
458 }
459 //
460 }
461 //
462 }
463 //
464 }
465 //
466 }
467 //
468 }
469 //
470 }
471 //
472 }
473 //
474 }
475 //
476 }
477 //
478 }
479 //
480 }
481 //
482 }
483 //
484 }
485 //
486 }
487 //
488 }
489 //
490 }
491 //
492 }
493 //
494 }
495 //
496 }
497 //
498 }
499 //
500 }
501 //
502 }
503 //
504 }
505 //
506 }
507 //
508 }
509 //
510 }
511 //
512 }
513 //
514 }
515 //
516 }
517 //
518 }
519 //
520 }
521 //
522 }
523 //
524 }
525 //
526 }
527 //
528 }
529 //
530 }
531 //
532 }
533 //
534 }
535 //
536 }
537 //
538 }
539 //
540 }
541 //
542 }
543 //
544 }
545 //
546 }
547 //
548 }
549 //
550 }
551 //
552 }
553 //
554 }
555 //
556 }
557 //
558 }
559 //
560 }
561 //
562 }
563 //
564 }
565 //
566 }
567 //
568 }
569 //
570 }
571 //
572 }
573 //
574 }
575 //
576 }
577 //
578 }
579 //
580 }
581 //
582 }
583 //
584 }
585 //
586 }
587 //
588 }
589 //
590 }
591 //
592 }
593 //
594 }
595 //
596 }
597 //
598 }
599 //
600 }
601 //
602 }
603 //
604 }
605 //
606 }
607 //
608 }
609 //
610 }
611 //
612 }
613 //
614 }
615 //
616 }
617 //
618 }
619 //
620 }
621 //
622 }
623 //
624 }
625 //
626 }
627 //
628 }
629 //
630 }
631 //
632 }
633 //
634 }
635 //
636 }
637 //
638 }
639 //
640 }
641 //
642 }
643 //
644 }
645 //
646 }
647 //
648 }
649 //
650 }
651 //
652 }
653 //
654 }
655 //
656 }
657 //
658 }
659 //
660 }
661 //
662 }
663 //
664 }
665 //
666 }
667 //
668 }
669 //
670 }
671 //
672 }
673 //
674 }
675 //
676 }
677 //
678 }
679 //
680 }
681 //
682 }
683 //
684 }
685 //
686 }
687 //
688 }
689 //
690 }
691 //
692 }
693 //
694 }
695 //
696 }
697 //
698 }
699 //
700 }
701 //
702 }
703 //
704 }
705 //
706 }
707 //
708 }
709 //
710 }
711 //
712 }
713 //
714 }
715 //
716 }
717 //
718 }
719 //
720 }
721 //
722 }
723 //
724 }
725 //
726 }
727 //
728 }
729 //
730 }
731 //
732 }
733 //
734 }
735 //
736 }
737 //
738 }
739 //
740 }
741 //
742 }
743 //
744 }
745 //
746 }
747 //
748 }
749 //
750 }
751 //
752 }
753 //
754 }
755 //
756 }
757 //
758 }
759 //
760 }
761 //
762 }
763 //
764 }
765 //
766 }
767 //
768 }
769 //
770 }
771 //
772 }
773 //
774 }
775 //
776 }
777 //
778 }
779 //
780 }
781 //
782 }
783 //
784 }
785 //
786 }
787 //
788 }
789 //
790 }
791 //
792 }
793 //
794 }
795 //
796 }
797 //
798 }
799 //
800 }
801 //
802 }
803 //
804 }
805 //
806 }
807 //
808 }
809 //
810 }
811 //
812 }
813 //
814 }
815 //
816 }
817 //
818 }
819 //
820 }
821 //
822 }
823 //
824 }
825 //
826 }
827 //
828 }
829 //
830 }
831 //
832 }
833 //
834 }
835 //
836 }
837 //
838 }
839 //
840 }
841 //
842 }
843 //
844 }
845 //
846 }
847 //
848 }
849 //
850 }
851 //
852 }
853 //
854 }
855 //
856 }
857 //
858 }
859 //
860 }
861 //
862 }
863 //
864 }
865 //
866 }
867 //
868 }
869 //
870 }
871 //
872 }
873 //
874 }
875 //
876 }
877 //
878 }
879 //
880 }
881 //
882 }
883 //
884 }
885 //
886 }
887 //
888 }
889 //
890 }
891 //
892 }
893 //
894 }
895 //
896 }
897 //
898 }
899 //
900 }
901 //
902 }
903 //
904 }
905 //
906 }
907 //
908 }
909 //
910 }
911 //
912 }
913 //
914 }
915 //
916 }
917 //
918 }
919 //
920 }
921 //
922 }
923 //
924 }
925 //
926 }
927 //
928 }
929 //
930 }
931 //
932 }
933 //
934 }
935 //
936 }
937 //
938 }
939 //
940 }
941 //
942 }
943 //
944 }
945 //
946 }
947 //
948 }
949 //
950 }
951 //
952 }
953 //
954 }
955 //
956 }
957 //
958 }
959 //
960 }
961 //
962 }
963 //
964 }
965 //
966 }
967 //
968 }
969 //
970 }
971 //
972 }
973 //
974 }
975 //
976 }
977 //
978 }
979 //
980 }
981 //
982 }
983 //
984 }
985 //
986 }
987 //
988 }
989 //
990 }
991 //
992 }
993 //
994 }
995 //
996 }
997 //
998 }
999 //
1000 }

```

```

9780 // Issue trailer
9781 // off = processTrailer(ctx, s, string(bb))
9782         return err
9783     }
9784     continue
9785 }
9786 // Ignore all until "trailer".
9787 s = strings.Index(line, "trailer")
9788 if i > s {
9789     bb.append(bb, line...)
9790     withIndexof = true
9791     continue
9792 }
9793 l = strings.Index(line, "eof")
9794 if i > l {
9795     offset += int64(int(line) - eofCount)
9796     withIndexof = true
9797     continue
9798 }
9799 if withIndexof {
9800     s = strings.Index(line, "obj")
9801     if i > s {
9802         withIndexof = true
9803         off += offset
9804         bb = append(bb, line[i:s]...)
9805     }
9806     offset += int64(int(line) - eofCount)
9807     continue
9808 }
9809 // finish
9810 offset += int64(int(line) - eofCount)
9811 bb = append(bb, s...)
9812 bb = append(bb, line...)
9813 s = strings.Index(line, "endobj")
9814 if i > s {
9815     l = string(bb)
9816     objMr, generation, err = parseObjectAttributes(s[l])
9817     if err == nil {
9818         return err
9819     }
9820     off = off
9821     ctx.tailer[objMr] = &TableEntry{
9822         offset:      false,
9823         offset2:     false,
9824         generation:  generation,
9825         ba: nil,
9826         withIndexof: false
9827     }
9828 }
9829 return nil
9830 }
9831
9832 // bufio.NewReader is reading from a stream or a file
9833 func bufio.NewReaderStartingAt(ctx *Context, offset *int64) error {
9834     log.Debug.Println("bufio.NewReaderStartingAt: begin")
9835 }

```

[illegible]



```

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=decryptHeader(ENDDICT: end: %s)", ok)
1578
1579     return ok
1580 }
1581
1582 func buildFilterPipeline(ctx *Context, filterArray, decodeParamsArr Array, decodeParams
1583 *dict) (*Pfilter, error) {
1584     var filterPipeline []Pfilter
1585
1586     for i, f := range filterArray {
1587
1588         filterName, ok := f.(Name)
1589         if !ok {
1590             corrupt := true
1591             return nil, errors.New("pfilter: buildFilterPipeline: filterArray elements
1592 corrupt")
1593         }
1594         if decodeParams == nil || decodeParamsArr[i] == nil {
1595             filterPipeline = append(filterPipeline, PFilter{Name:
1596 filterName, decodeParams: nil})
1597             continue
1598         }
1599         dict, ok := decodeParamsArr[i].(Dict)
1600         if !ok {
1601             corrupt := true
1602             return nil, errors.New("pfilter: buildFilterPipeline: corrupt Dict: %s",
1603 dict)
1604         }
1605         d, err := dereferenceDicts(dict, indirectObjectNumber.Value())
1606         if err != nil {
1607             return nil, err
1608         }
1609         dict = d
1610     }
1611
1612     filterPipeline = append(filterPipeline, PFilter{Name: filterName.String(),
1613 decodeParams: dict})
1614 }
1615
1616 return filterPipeline, nil
1617 }
1618
1619 // Decode the filter pipeline associated with this stream dict:
1620 func buildFilterPipelines(*Context, dict Dict) (*[]Pfilter, error) {
1621     log.Read.Printf("pfilterPipeline: begin")
1622
1623     var err error
1624
1625     o, found := dict.Find("Filter")
1626     if !found {
1627         // stream is not compressed
1628     }

```

[illegible]

```

1130 // Save the saveDecodedContentContext to ctx.content, id, saveStreams, objKey, goenv int, decoded
1131 // err error()
1132
1133 // Log.Read.Print("saveDecodedContentContext: begin decode\n"), decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if ctx.will nil on ctx.filterKey == nil {
1137     if ctx.filterPolicyName == 1 do ctx.FilterPolicyName(),Name == "Crypt" {
1138         return nil
1139     }
1140 }
1141
1142 // Special case: If the length of the encoded data is 0, we do not need to decode anything.
1143 if ctx.len(StdRaw) == 0 {
1144     StdContent = StdRaw
1145     return nil
1146 }
1147
1148 // ctx gets created after StdStream parsing.
1149 // StdStream is not encrypted.
1150 if ctx.will == StdRaw {
1151     StdRaw, err = decryptStdStream(StdRaw, objKey, goenv, ctx.KeyKey, ctx.AESStreamSize,
1152 ctx.Ex)
1153     if err == nil {
1154         return err
1155     }
1156     l = len(StdRaw)
1157     StdStream.Length = l
1158 }
1159
1160 // If decode {
1161     return nil
1162 }
1163
1164 // Actual decoding of content stream.
1165 err = decodeStream()
1166 if err == filter.StreamSupportFilter {
1167     err = nil
1168 }
1169 if err == nil {
1170     return err
1171 }
1172
1173 // Log.Read.Print("saveDecodedContentContext: end")
1174
1175 return nil
1176
1177 // Decode compressed objTableEntry
1178 func decodeCompressedObjTableEntry (objTable *objTable, objStream int, entry
1179 int) error {
1180     // Log.Read.Print("decodeCompressedObjTableEntry: compressed object id at %d\n"),
1181     objStream, entry, objStream, entry, objStream)
1182
1183     // Message stream is not in reference object stream.
1184     objStreamEntryTable, obj = objTable.FindEntry(objStream)
1185     if obj {

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry %d", 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 %d, 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 %d, needs Integer values", objId)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offsets64
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs Integer values", objId)
2053             }
2054             ctx.OffsetOverluminTable = offsets64
2055         }
2056     }
2057     return nil
2058 }
2059
2060 func LoadLinearizationDict(ctx *Context, s *StreamReader, objId, genNr int) error {
2061     var err error
2062     if !
2063         // Load stream's content and store data into offsetable entry
2064         if err = LoadLinearizationDictFromStream(s, objId, ctx, genNr); err != nil {
2065             return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d", objId)
2066         }
2067         ctx.Read.BinaryFileSize += s.GetSizeLength()
2068         // Decode stream's content
2069         err = saveDecodedStreamContent(ctx, s, objId, genNr, ctx.DecodedAllStreams)
2070         if err != nil {
2071             return err
2072         }
2073     }
2074     return nil
2075 }
2076
2077 func updateLinearizationDict(ctx *Context, o Object) {
2078     switch o := o.(type) {
2079     case StreamDict:
2080         ctx.Add.BinaryFileSize += o.GetSizeLength()
2081     }
2082 }

```

[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

```

```

5042         return nil, nil
5043     }
5044 }
5045 // compressed stream.
5046 var filterPipeline []PFFilter
5047
5048 if !defer, ok := o.(IndirectRef); ok {
5049     o, err := deferPipelineDecodeObj(ctx, indirRef.ObjectNumber.Value())
5050     if err != nil {
5051         return nil, err
5052     }
5053 }
5054
5055 //fmt.Printf("Decomposed filter obj: %v\n", obj)
5056
5057 if name, ok := o.(Name); ok {
5058     // single filter.
5059     filterName := name.String()
5060     o, found := dict.Find("DecodedParam")
5061     if !found {
5062         // w/o decoded parameter.
5063         log.Bad-Print("pffilterPipeline: w/o decoded param")
5064         return append(filterPipeline, PFFilterName: filterName, DecodedParam:
5065             nil), nil
5066     }
5067     d, ok := o.(Dict)
5068     if !ok {
5069         // ok
5070         if ok := o.(IndirectRef) {
5071             if !ok {
5072                 return nil, errors.Errorf("pffilterPipeline: corrupt Dict: %v", o)
5073             }
5074             o, err := deferPipelineDecodeObj(ctx, ir.ObjectNumber.Value())
5075             if err != nil {
5076                 return nil, err
5077             }
5078         }
5079     }
5080 }
5081 // with decoded parameter.
5082 log.Bad-Print("pffilterPipeline: with decoded param")
5083 return append(filterPipeline, PFFilterName: filterName, DecodedParam: d),
5084     nil
5085 }
5086 // filter pipeline.
5087
5088 // Array of filternames
5089 filterArray, ok := o.([]array)
5090 if !ok {
5091     return nil, errors.Errorf("pffilterPipeline: Expected FilterArray corrupt, %v",
5092         o, o.S)
5093 }
5094 // Optional array of decoded parameter dicts.
5095 var decodedParamArray

```

```

3540 // (ct, <E>)
3541 if err == nil {
3542     return nil, err
3543 }
3544 return StringLiteral(string(bb)), nil
3545 }
3546
3547 // default:
3548 return o, nil
3549 }
3550 }
3551
3552 func dereferenceObject(ctx <Context, objectNumber int> (Object, error) {
3553     entry, ok := ctx.Find(objectNumber)
3554     if !ok {
3555         return nil, errors.New("p4cpu: dereferenceObject: unregistered object")
3556     }
3557     if entry.Compressed {
3558         err := decompressHeaderTable(entry.Ctx, <HeaderTable, objectNumber, entry>)
3559         if err == nil {
3560             return nil, err
3561         }
3562     }
3563     if entry.Ref == nil {
3564         log.Bad.Printf("dereferenceObject: dereferencing object %d\n", objectNumber)
3565         o, err := ParseObject(ctx, entry.Offset, objectNumber, entry.Generation)
3566         if err == nil {
3567             return nil, errors.Wrap(err, "dereferenceObject: problem dereferencing object %d", objectNumber)
3568         }
3569     }
3570     if o == nil {
3571         return nil, errors.New("p4cpu: dereferenceObject: object is nil")
3572     }
3573     entry.Object = o
3574 }
3575
3576 return entry.Object, nil
3577 }
3578
3579 func dereferenceInteger(ctx <Context, objectNumber int> (Integer, error) {
3580     o, err := dereferenceObject(ctx, objectNumber)
3581     if err == nil {
3582         return nil, err
3583     }
3584     i, ok := o.(Integer)
3585     if !ok {
3586         return nil, errors.New("p4cpu: dereferenceInteger: corrupt integer")
3587     }
3588 }

```

```

1573 // Do not access object's decompressHeader attribute: problem dereferencing object
1574 streamMd, no ref table entry", entry.objectsStreamMd);
1575
1576 //1574
1577 //1575
1578 //1576 // Object of class entry has no objectStreamMd
1579 //1577 sd, o = objectStreamMd.objectsStreamMd (ObjectStreamMd)
1580 //1578 if (ok {
1581 //1579 return errors.Errorf("decompressHeaderTableEntry: problem dereferencing objectStreamMd, no object stream", entry.objectsStreamMd)
1582 //1580 }
1583 //1581
1584 //1582 // Get IndexMd object from ObjectStreamMd
1585 //1583 o, err = sd.IndexMdObjectFromObjectStreamMd
1586 //1584 if err != nil {
1587 //1585 return errors.Errorf("decompressHeaderTableEntry: problem dereferencing object stream Md", entry.objectsStreamMd)
1588 //1586 }
1589 //1587 // Save object to theHeaderTableEntry.
1590 //1588
1591 //1589 g :=
1592 //1590 entry.Object + o
1593 //1591 entry.Compression = HG
1594 //1592 entry.ExtraData = false
1595 //1593
1596 //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 //1655 //1656 //1657 //1658 //1659 //1660 //1661 //1662 //1663 //1664 //1665 //1666 //1667 //1668 //1669 //1670 //1671 //1672 //1673 //1674 //1675 //1676 //1677 //1678 //1679 //1680 //1681 //1682 //1683 //1684 //1685 //1686 //1687 //1688 //1689 //1690 //1691 //1692 //1693 //1694 //1695 //1696 //1697 //1698 //1699 //1700 //1701 //1702 //1703 //1704 //1705 //1706 //1707 //1708 //1709 //1710 //1711 //1712 //1713 //1714 //1715 //1716 //1717 //1718 //1719 //1720 //1721 //1722 //1723 //1724 //1725 //1726 //1727 //1728 //1729 //1730 //1731 //1732 //1733 //1734 //1735 //1736 //1737 //1738 //1739 //1740 //1741 //1742 //1743 //1744 //1745 //1746 //1747 //1748 //1749 //1750 //1751 //1752 //1753 //1754 //1755 //1756 //1757 //1758 //1759 //1760 //1761 //1762 //1763 //1764 //1765 //1766 //1767 //1768 //1769 //1770 //1771 //1772 //1773 //1774 //1775 //1776 //1777 //1778 //1779 //1780 //1781 //1782 //1783 //1784 //1785 //1786 //1787 //1788 //1789 //1790 //1791 //1792 //1793 //1794 //1795 //1796 //1797 //1798 //1799 //1800 //1801 //1802 //1803 //1804 //1805 //1806 //1807 //1808 //1809 //1810 //1811 //1812 //1813 //1814 //1815 //1816 //1817 //1818 //1819 //1820 //1821 //1822 //1823 //1824 //1825 //1826 //1827 //1828 //1829 //1830 //1831 //1832 //1833 //1834 //1835 //1836 //1837 //1838 //1839 //1840 //1841 //1842 //1843 //1844 //1845 //1846 //1847 //1848 //1849 //1850 //1851 //1852 //1853 //1854 //1855 //1856 //1857 //1858 //1859 //1860 //1861 //1862 //1863 //1864 //1865 //1866 //1867 //1868 //1869 //1870 //1871 //1872 //1873 //1874 //1875 //1876 //1877 //1878 //1879 //1880 //1881 //1882 //1883 //1884 //1885 //1886 //1887 //1888 //1889 //1890 //1891 //1892 //1893 //1894 //1895 //1896 //1897 //1898 //1899 //1900 //1901 //1902 //1903 //1904 //1905 //1906 //1907 //1908 //1909 //1910 //1911 //1912 //1913 //1914 //1915 //1916 //1917 //1918 //1919 //1920 //1921 //1922 //1923 //1924 //1925 //1926 //1927 //1928 //1929 //1930 //1931 //1932 //1933 //1934 //1935 //1936 //1937 //1938 //1939 //1940 //1941 //1942 //1943 //1944 //1945 //1946 //1947 //1948 //1949 //1950 //1951 //1952 //1953 //1954 //1955 //1956 //1957 //1958 //1959 //1960 //1961 //1962 //1963 //1964 //1965 //1966 //1967 //1968 //1969 //1970 //1971 //1972 //1973 //1974 //1975 //1976 //1977 //1978 //1979 //1980 //1981 //1982 //1983 //1984 //1985 //1986 //1987 //1988 //1989 //1990 //1991 //1992 //1993 //1994 //1995 //1996 //1997 //1998 //1999 //2000 //2001 //2002 //2003 //2004 //2005 //2006 //2007 //2008 //2009 //2010 //2011 //2012 //2013 //2014 //2015 //2016 //2017 //2018 //2019 //2020 //2021 //2022 //2023 //2024 //2025 //2026 //2027 //2028 //2029 //2030 //2031 //2032 //2033 //2034 //2035 //2036 //2037 //2038 //2039 //2040 //2041 //2042 //2043 //2044 //2045 //2046 //2047 //2048 //2049 //2050 //2051 //2052 //2053 //2054 //2055 //2056 //2057 //2058 //2059 //2060 //2061 //2062 //2063 //2064 //2065 //2066 //2067 //2068 //2069 //2070 //2071 //2072 //2073 //2074 //2075 //2076 //2077 //2078 //2079 //2080 //2081 //2082 //2083 //2084 //2085 //2086 //2087 //2088 //2089 //2090 //2091 //2092 //2093 //2094 //2095 //2096 //2097 //2098 //2099 //2100 //2101 //2102 //2103 //2104 //2105 //2106 //2107 //2108 //2109 //2110 //2111 //2112 //2113 //2114 //2115 //2116 //2117 //2118 //2119 //2120 //2121 //2122 //2123 //2124 //2125 //2126 //2127 //2128 //2129 //2130 //2131 //2132 //2133 //2134 //2135 //2136 //2137 //2138 //2139 //2140 //2141 //2142 //2143 //2144 //2145 //2146 //2147 //2148 //2149 //2150 //2151 //2152 //2153 //2154 //2155 //2156 //2157 //2158 //2159 //2160 //2161 //2162 //2163 //2164 //2165 //2166 //2167 //2168 //2169 //2170 //2171 //2172 //2173 //2174 //2175 //2176 //2177 //2178 //2179 //2180 //2181 //2182 //2183 //2184 //2185 //2186 //2187 //2188 //2189 //2190 //2191 //2192 //2193 //2194 //2195 //2196 //2197 //2198 //2199 //2200 //2201 //2202 //2203 //2204 //2205 //2206 //2207 //2208 //2209 //2210 //2211 //2212 //2213 //2214 //2215 //2216 //2217 //2218 //2219 //2220 //2221 //2222 //2223 //2224 //2225 //2226 //2227 //2228 //2229 //2230 //2231 //2232 //2233 //2234 //2235 //2236 //2237 //2238 //2239 //2240 //2241 //2242 //2243 //2244 //2245 //2246 //2247 //2248 //2249 //2250 //2251 //2252 //2253 //2254 //2255 //2256 //2257 //2258 //2259 //2260 //2261 //2262 //2263 //2264 //2265 //2266 //2267 //2268 //2269 //2270 //2271 //2272 //2273 //2274 //2275 //2276 //2277 //2278 //2279 //2280 //2281 //2282 //2283 //2284 //2285 //2286 //2287 //2288 //2289 //2290 //2291 //2292 //2293 //2294 //2295 //2296 //2297 //2298 //2299 //2300 //2301 //2302 //2303 //2304 //2305 //2306 //2307 //2308 //2309 //2310 //2311 //2312 //2313 //2314 //2315 //2316 //2317 //2318 //2319 //2320 //2321 //2322 //2323 //2324 //2325 //2326 //2327 //2328 //2329 //2330 //2331 //2332 //2333 //2334 //2335 //
```

[illegible]

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObjectObject(ctxs)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shEntryEntry object assign either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObjectObject(ctxs)
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     err = decodeObjectObject(ctxs)
2139     if err != nil {
2140         return err
2141     }
2142
2143     log.Root.Println("referenceCatalogTable: end")
2144
2145     return nil
2146 }
2147
2148 func handleEncryptedFile(ctxs *Context) error {
2149     err := ctxs.Cmd == DECRYPT || ctxs.Cmd == SETPERMISSIONS ||
2150         return errors.New("pfcpu: this file is not encrypted")
2151 }
2152
2153 if ctxs.Cmd == DECRYPT {
2154     return nil
2155 }
2156
2157 // Encrypt subcommand found.
2158
2159 if ctxs.SubCmd == "i" {
2160     return errors.New("pfcpu: please provide owner password and optional user
2161 password")
2162 }
2163
2164 return nil
2165 }
2166
2167 func lshBytes(ctxs *Context) (id []byte, err error) {
2168     if ctxs.ID == nil {
2169         return nil, errors.New("pfcpu: missing ID entry")
2170     }
2171
2172     N1, ok := ctxs.ID[0].(uint64)
2173     if ok {
2174         id, err = n1.Bytes()
2175         if err != nil {
2176             return nil, err
2177         }
2178     }
2179 }

```

```

1452 // decodeParam, found = dict.Fmt.DecodeParam(s)
1453 if found {
1454     decodeParamArr, ok = decodeParam(s.Array)
1455     if !ok {
1456         return nil, errors.New("pdpic: pdfFilterPipeline: expected decodeParam
1457 array corrupt")
1458     }
1459 }
1460
1461 // /xx.PdfDict("decodeParms: ba1", decodeParamArr)
1462
1463 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArr,
1464 decodeParam)
1465 if err != nil {
1466     log.Read.PdfFilterPipeline("pdfFilterPipeline: err")
1467     return filterPipeline, err
1468 }
1469
1470 func streamDictForObj(c *Context, d Dict, objKey, streamIn int, streamFset
1471 *FileSet, objId int) (StreamDict, error) {
1472     streamLength, streamLengthF = d.Length()
1473     if streamLength == 0 {
1474         return sd, errors.New("pdpic: streamDictForObj: stream object without
1475 streamFset")
1476     }
1477     filterPipeline, err = pdfFilterPipeline(ctx, d)
1478     if err == nil {
1479         return sd, err
1480     }
1481     streamOffset = offset
1482
1483 // We have a stream object
1484 sd = NewStream(streamDict, streamOffset, streamLength, streamLengthF, filterPipeline)
1485 log.Read.PdfStream("streamDictForObj: end, streamObject %d\n", objId)
1486 return sd, nil
1487 }
1488
1489 func dictCt(c *Context, d Dict, objKey, err, objId, streamIn int) (Dict Dict, err
1490 error) {
1491     if c.IsEncKey == nil {
1492         objKey = decryptObjStreamDict(d, objKey, err, c.IsEncKey, c.AES4Strings,
1493         c.AES4B)
1494         if err == nil {
1495             return nil, err
1496         }
1497     }
1498     if objId == 0 || IsStream(c == 0 || streamIn == undefined) {
1499         log.Read.PdfDict("dict: end, %d\n", objId)
1500         d2 = d1
1501     }
1502 }

```

```

3730         return dc, nil
3731     }
3732 }
3733
3734 func dereferenceObject(dict *Context, objectNumber int) (dict, error) {
3735     // 1. ok = dereferenceObject(dict, objectNumber)
3736     // 2. if err == nil {
3737         return nil, err
3738     }
3739     // 3. ok = ok.(dict)
3740     // 4. if !ok {
3741         return nil, errors.New("pdpcc: dereferenceObject: corrupt dict")
3742     }
3743 }
3744
3745 return dc, nil
3746 }
3747
3748 // dereference a message object representing an object value.
3749 func intObject(dict *Context, objectNumber int) (uint64, error) {
3750     // 1. log.Read.Print("intObject begin: %d\n", objectNumber)
3751     // 2.
3752     // 3. 1. err = dereferenceInteger(dict, objectNumber)
3753     // 4. if err == nil {
3754         return nil, err
3755     }
3756     // 5.
3757     // 6. id = int64(i.Value())
3758     // 7. log.Read.Print("intObject end: %d\n", objectNumber)
3759     // 8.
3760     return id64, nil
3761 }
3762
3763 // Reads and returns a file buffer with length = streamLength using provided reader
3764 // positioned at offset.
3765 func readStreamStream(read io.Reader, streamLength int) ([]byte, error) {
3766     // 1. log.Read.Print("readStreamStream: begin streamLength=%d\n", streamLength)
3767     // 2.
3768     buf := make([]byte, streamLength)
3769     // 3.
3770     for totalCount := 0; totalCount < streamLength; {
3771         // 4. count, err = d.Read(buf[totalCount:])
3772         // 5. if err == nil {
3773             return nil, err
3774         }
3775         // 6.
3776         log.Read.Print("readStreamStream: count=%d, bufLen=%d(x)%v", count,
3777             len(buf), buf[totalCount:])
3778         totalCount += count
3779     }
3780     // 7.
3781     log.Read.Print("readStreamStream: end\n")
3782     // 8.
3783     return buf, nil
3784 }

```

```

130 // @see https://github.com/ericniebler/psutil/blob/master/psutil/_psutil_linux.c
131         log.Read.PrintIn("logStream", 0, OBJECTS_READY) }
132
133
134
135 // Decode all object streams to contained objects are ready to be used.
136 void decodeObjectStreams(cts::Context) error {
137     // @see
138     // @entry "externs" intentionally left out.
139     // No object stream collection validation necessary.
140
141     log.Read.PrintIn("decodeobjectstreams: begin")
142
143     // Get sorted slice of object numbers.
144     // See key List
145     for k = range cts.Read.ObjectStreams {
146         keys = append(keys, k)
147     }
148     sort.Int(keys)
149
150     for _ , objectNumber = range keys {
151         // @see ObjectReadyIndex.
152         entry = cts.StableTable.Table(objectNumber)
153         if entry == nil {
154             return errors.Errorf("decodeobjectstreams: missing entry for objid%u",
155                 objectNumber)
156         }
157         log.Read.PrintIn("decodeobjectstreams: parsing object stream for objid%u",
158             objectNumber)
159
160         // Parse object stream from file.
161         o, err = ParseObjectStream(entry.Offset, objectNumber, entry.Generation)
162         if err != nil || o == nil {
163             return errors.New("pdpcc: decodeobjectstreams: corrupt object stream")
164         }
165
166         // Ensure streamObject
167         sd, ok = o.(StreamObject)
168         if !ok {
169             return errors.New("pdpcc: decodeobjectstreams: corrupt object stream")
170         }
171
172         // Load decoded stream content to stableTable.
173         if err = loadDecodedStreamContent(cts, sd); err != nil {
174             return errors.Wrapf(err, "decodeobjectstreams: problem dereferencing
175                 object stream %u", objectNumber)
176         }
177
178         // Save decoded stream content to stableTable.
179         if err = saveDecodedStreamContent(cts, sd, objectNumber, entry.Generation,
180             true); err != nil {
181             return err
182         }
183     }
184 }

```

```

2342 // err = ParseObject(ctxt, entry.Offset, objNr, entry.Generation)
2343 if err == nil {
2344     return errors.Wrapf(err, "dereferencedObject: problem dereferencing object %d",
2345         objNr)
2346 }
2347
2348 entry.Object = o
2349
2350 // Linearization objects are validated and removed for stats only.
2351 err = handleLinearizationStats(ctxt, o, objNr)
2352 if err == nil {
2353     return err
2354 }
2355
2356 // Handle stream dics.
2357 if objNr == 0 {
2358     if err := o.(ObjectStreamDict).ok {
2359         // Stream errors from a dereferencedObject: object stream should already be
2360         // referenced at objId, objNr
2361         return err
2362     }
2363     if err := o.(ObjectStreamDict).ok {
2364         // return errors from a dereferencedObject: xref stream should already be
2365         // referenced at objId, objNr
2366         return err
2367     }
2368     if sd, ok := o.(StreamDict).ok {
2369         if err := loadStream(ctxt, sd, objNr, entry.Generation)
2370         if err == nil {
2371             return err
2372         }
2373     }
2374     entry.Object = sd
2375 }
2376
2377 log.Root().Printf("dereferencedObject: and objId of %v to %v", objNr,
2378     objNrDict, entry.Object)
2379
2380 logStream(entry.Object)
2381 return nil
2382 }
2383
2384 func processBidsAndCounts(defTable *XRefTable, D Dict) {
2385     for _, n := range o {
2386         match ok := xEqTable(
2387             case IndexNotSet:
2388                 entry, ok := defTable.LookupTableEntryForIndexDef(nal)
2389                 if ok {
2390                     entry.Count++
2391                 }
2392             case Dict:
2393                 processBidsAndCounts(defTable, o)
2394             case Array:
2395                 processBidsAndCounts(defTable, o)
2396         }
2397     }
2398 }

```

```

2317 }
2318 // else {
2319 //     id = ctx.ID().ID().StringLiteral()
2320 //     if id != ""
2321 //         return nil, error.New("pdpoc: ID must contain hex literals or string
2322 //             literals");
2323 // }
2324 // id, err = Unescape(id.Value());
2325 // if err != nil {
2326 //     return nil, err
2327 // }
2328 // }
2329 // }
2330 // return id, nil
2331 }
2332
2333 func needsOwnerAndAdminPassword(cnd CommandNode) bool {
2334     return cnd == CHANGELOGS || cnd == CHANGELOGS || cnd == SETPERMISSIONS
2335 }
2336
2337 func handlePermissions(ctx *Context) error {
2338     // AE255 Validate permissions
2339     ok, err = validatePermissions(ctx)
2340     if err != nil {
2341         return err
2342     }
2343     if !ok {
2344         return errors.New("pdpoc: corrupted permissions after upw ok")
2345     }
2346     // Double check existing permissions for pdpoc processing.
2347     if hasWritePermissions(ctx, ctx.Cmd) {
2348         return errors.New("pdpoc: insufficient access permissions")
2349     }
2350     return nil
2351 }
2352
2353 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2354     //
2355     ctx.t, err = supportGetEncryption(ctx, d)
2356     if err != nil {
2357         return err
2358     }
2359     //
2360     ctx.t.ID, err = idbytes(ctx)
2361     if err != nil {
2362         return err
2363     }
2364     //
2365     var ok bool
2366     // //for:Prefix("poc: k2o: upw: k2o: k1: ", ctx.OwnerNs, ctx.IDStr99)
2367     // // Validate the owner password sha_permissions/master_password
2368     ok, err = validateOwnerPassword(ctx)

```

[illegible][illegible][illegible]

```

2020: // 2020:
2021: func processArrayByCounts(x:Iterable, y:Iterable, a Array) {
2022:     for _ in range a {
2023:         switch o in a {x,y} {
2024:             case IndexDefect:
2025:                 entry, ok = x.elementAt(i).flatMap(Iterable::of)(fail)
2026:                 if ok {
2027:                     entry, bcCount++ =
2028:                         case Count:
2029:                             processByCounts(x:Iterable, o)
2030:                         case Array:
2031:                             processByCounts(x:Iterable, o)
2032:                     }
2033:                 }
2034:             }
2035:         }
2036:     }
2037: }
2038:
2039: func processByCounts(x:Iterable, o:Iterable, o:Object) {
2040:     switch o in o {type} {
2041:         case Count:
2042:             processByCounts(x:Iterable, o)
2043:         case StreamDefect:
2044:             processByCounts(x:Iterable, o:Unit)
2045:         case Array:
2046:             processArrayByCounts(x:Iterable, o)
2047:         }
2048:     }
2049: }
2050:
2051: // 2051:
2052: // 2052:
2053: // 2053:
2054: // 2054:
2055: // 2055:
2056: // 2056:
2057: // 2057:
2058: // 2058:
2059: // 2059:
2060: // 2060:
2061: // 2061:
2062: // 2062:
2063: // 2063:
2064: // 2064:
2065: // 2065:
2066: // 2066:
2067: // 2067:
2068: // 2068:
2069: // 2069:
2070: // 2070:
2071: // 2071:
2072: // 2072:
2073: // 2073:
2074: // 2074:
2075: // 2075:
2076: // 2076:
2077: // 2077:
2078: // 2078:
2079: // 2079:
2080: // 2080:
2081: // 2081:
2082: // 2082:
2083: // 2083:
2084: // 2084:
2085: // 2085:
2086: // 2086:
2087: // 2087:
2088: // 2088:
2089: // 2089:
2090: // 2090:
2091: // 2091:
2092: // 2092:
2093: // 2093:
2094: // 2094:
2095: // 2095:
2096: // 2096:
2097: // 2097:
2098: // 2098:
2099: // 2099:
2100: // 2100:
2101: // 2101:
2102: // 2102:
2103: // 2103:
2104: // 2104:
2105: // 2105:
2106: // 2106:
2107: // 2107:
2108: // 2108:
2109: // 2109:
2110: // 2110:
2111: // 2111:
2112: // 2112:
2113: // 2113:
2114: // 2114:
2115: // 2115:
2116: // 2116:
2117: // 2117:
2118: // 2118:
2119: // 2119:
2120: // 2120:
2121: // 2121:
2122: // 2122:
2123: // 2123:
2124: // 2124:
2125: // 2125:
2126: // 2126:
2127: // 2127:
2128: // 2128:
2129: // 2129:
2130: // 2130:
2131: // 2131:
2132: // 2132:
2133: // 2133:
2134: // 2134:
2135: // 2135:
2136: // 2136:
2137: // 2137:
2138: // 2138:
2139: // 2139:
2140: // 2140:
2141: // 2141:
2142: // 2142:
2143: // 2143:
2144: // 2144:
2145: // 2145:
2146: // 2146:
2147: // 2147:
2148: // 2148:
2149: // 2149:
2150: // 2150:
2151: // 2151:
2152: // 2152:
2153: // 2153:
2154: // 2154:
2155: // 2155:
2156: // 2156:
2157: // 2157:
2158: // 2158:
2159: // 2159:
2160: // 2160:
2161: // 2161:
2162: // 2162:
2163: // 2163:
2164: // 2164:
2165: // 2165:
2166: // 2166:
2167: // 2167:
2168: // 2168:
2169: // 2169:
2170: // 2170:
2171: // 2171:
2172: // 2172:
2173: // 2173:
2174: // 2174:
2175: // 2175:
2176: // 2176:
2177: // 2177:
2178: // 2178:
2179: // 2179:
2180: // 2180:
2181: // 2181:
2182: // 2182:
2183: // 2183:
2184: // 2184:
2185: // 2185:
2186: // 2186:
2187: // 2187:
2188: // 2188:
2189: // 2189:
2190: // 2190:
2191: // 2191:
2192: // 2192:
2193: // 2193:
2194: // 2194:
2195: // 2195:
2196: // 2196:
2197: // 2197:
2198: // 2198:
2199: // 2199:
2200: // 2200:
2201: // 2201:
2202: // 2202:
2203: // 2203:
2204: // 2204:
2205: // 2205:
2206: // 2206:
2207: // 2207:
2208: // 2208:
2209: // 2209:
2210: // 2210:
2211: // 2211:
2212: // 2212:
2213: // 2213:
2214: // 2214:
2215: // 2215:
2216: // 2216:
2217: // 2217:
2218: // 2218:
2219: // 2219:
2220: // 2220:
2221: // 2221:
2222: // 2222:
2223: // 2223:
2224: // 2224:
2225: // 2225:
2226: // 2226:
2227: // 2227:
2228: // 2228:
2229: // 2229:
2230: // 2230:
2231: // 2231:
2232: // 2232:
2233: // 2233:
2234: // 2234:
2235: // 2235:
2236: // 2236:
2237: // 2237:
2238: // 2238:
2239: // 2239:
2240: // 2240:
2241: // 2241:
2242: // 2242:
2243: // 2243:
2244: // 2244:
2245: // 2245:
2246: // 2246:
2247: // 2247:
2248: // 2248:
2249: // 2249:
2250: // 2250:
2251: // 2251:
2252: // 2252:
2253: // 2253:
2254: // 2254:
2255: // 2255:
2256: // 2256:
2257: // 2257:
2258: // 2258:
2259: // 2259:
2260: // 2260:
2261: // 2261:
2262: // 2262:
2263: // 2263:
2264: // 2264:
2265: // 2265:
2266: // 2266:
2267: // 2267:
2268: // 2268:
2269: // 2269:
2270: // 2270:
2271: // 2271:
2272: // 2272:
2273: // 2273:
2274: // 2274:
2275: // 2275:
2276: // 2276:
2277: // 2277:
2278: // 2278:
2279: // 2279:
2280: // 2280:
2281: // 2281:
2282: // 2282:
2283: // 2283:
2284: // 2284:
2285: // 2285:
2286: // 2286:
2287: // 2287:
2288: // 2288:
2289: // 2289:
2290: // 2290:
2291: // 2291:
2292: // 2292:
2293: // 2293:
2294: // 2294:
2295: // 2295:
2296: // 2296:
2297: // 2297:
2298: // 2298:
2299: // 2299:
2300: // 2300:
2301: // 2301:
2302: // 2302:
2303: // 2303:
2304: // 2304:
2305: // 2305:
2306: // 2306:
2307: // 2307:
2308: // 2308:
2309: // 2309:
2310: // 2310:
2311: // 2311:
2312: // 2312:
2313: // 2313:
2314: // 2314:
2315: // 2315:
2316: // 2316:
2317: // 2317:
2318: // 2318:
2319: // 2319:
2320: // 2320:
2321: // 2321:
2322: // 2322:
2323: // 2323:
2324: // 2324:
2325: // 2325:
2326: // 2326:
2327: // 2327:
2328: // 2328:
2329: // 2329:
2330: // 2330:
2331: // 2331:
2332: // 2332:
2333: // 2333:
2334: // 2334:
2335: // 2335:
2336: // 233
```

```

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 //
2538 // Unless we need to limit on a user's owner password due to the specific
2539 // amount in use password.
2540 if tok != newPasswordOwnerPassword(ctx.Cnd) {
2541     return errors.New("password: please provide the master password with 'opw'")
2542 }
2543 //
2544 //
2545 // Generally the user password, which is also regarded as the master password or
2546 // pre-password.
2547 //
2548 // It is sufficient for moving on. A password change is an occasion since it
2549 // is not.
2550 if ok != newPasswordOwnerPassword(ctx.Cnd) {
2551     return err
2552 }
2553 ok, err = validatePermissions(ctx)
2554 if err == nil {
2555     return err
2556 }
2557 //
2558 // If ok {
2559     return errors.New("password: corrupted permissions after opw ok")
2560 }
2561 //
2562 //
2563 // return nil
2564 //
2565 //
2566 // Validate the user password ok, document opw password.
2567 ok, err = validatePermissions(ctx)
2568 if err == nil {
2569     return err
2570 }
2571 //
2572 // If ok {
2573     return errors.New("password: please provide the correct password")
2574 }
2575 //
2576 //
2577 // //nc.Print("opw ok: %d\n", ok)
2578 //
2579 return handlePermissions(ctx)
2580 }
2581 //
2582 //
2583 // func checkForEncryption(c *Context) error {
2584 //
2585 //     ic := ctx.Encrypt
2586 //
2587 //     if ic == nil {
2588 //         // This file is not encrypted.
2589 //         return handleEncryptionFailed(ic)
2590 //     }
2591 //
2592 //     // This file is encrypted.
2593 //     log.ReadPrint("Encryption: %v\n", ic)
2594 //
2595 //     if ctx.Cnd == ENCRYPT {
2596 //         // We want to encrypt this file.
2597 //         return errors.New("password: This file is already encrypted")
2598 //     }
2599 //
2600 //     // Difference encryptFile.

```