

```

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/big"
66     "math/rand"
67     "strings"
68     "time"
69     "github.com/pdfcpu/pdfcpu/pkg/pdfwriter"
70     "github.com/pdfcpu/pdfcpu/pkg/rasterizer"
71     "github.com/pdfcpu/pdfcpu/pkg/transform"
72 )
73
74 // Read reads a PDF file and builds an internal structure holding its cross
75 // reference table and the objects.
76 func Read(filename string, conf Configuration) (*Context, error) {
77     log.Infof("Reading %s (%v, %v)", filename,
78         &conf, &conf)
79     f, err := os.Open(filename)
80     if err != nil {
81         return nil, errors.Wrap(err, "can't open %s", filename)
82     }
83
84     defer func() { f.Close() }()
85
86     return ReadFile(f, conf)
87 }
88
89 // Read takes a reader/writer and generates a Context,
90 // or an error in case of a non-readable or corrupt reference table.
91 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.parseObjectTableEntry: corrupt ref subobject
2311             entry)
2312         }
2313         var xrefTableEntry *XrefTableEntry
2314         if entryType == "in" {
2315             // in use object
2316             log.Debug.Printf("parseObjectTableEntry: Object %d is in use at offset%0d,
2317             generation%0d", objNumber, offset, generation)
2318             if offset == 0 {
2319                 log.Info.Printf("parseXrefTableEntry: 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.Printf("parseObjectTableEntry: Object %d is unused, next free is
2331             object %d", objNumber, objectNumber, offset, generation)
2332             xrefTableEntry =
2333                 &XrefTableEntry{
2334                     free: true,
2335                     offset: offset,
2336                     Generation: &generation}
2337         }
2338     }
2339     log.Debug.Printf("ParseObjectTableEntry: Insert new xrefTable entry for Object %d",
2340     objNumber)
2341     xrefTable.Table[objNumber] = xrefTableEntry
2342     return nil
2343 }
2344 func (p *Parser) parseObjectTableEntryEnd() error {
2345     return nil
2346 }

```

[illegible]

```

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     xRefTable = ctx.XRefTable
693     err = parameterInfoRef(trailerDict, xRefTable)
694     if err == nil {
695         return nil, err
696     }
697     if err == trailerEntryArrayEntry("AdditionalStreams"); err == nil {
698         log.Debug.Printf("parameterInfoRef: found AdditionalStreams: %s\n", err)
699         a := value
700         for a.value == range arr {
701             if infoRef := & value (indirectEntry), a {
702                 a = append(a, indirect)
703             }
704         }
705         xRefTable.AdditionalStreams = &a
706     }
707     offset = trailerDict.Prev()
708     if offset == nil {
709         log.Debug.Printf("parameterInfoRef: previous xref table section offset: %s\n",
710             offset)
711     }
712     offset := trailerDict.InfoEntry("xref")
713 }

```

```

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 cable subsection
9918     if !line == "trailer" {
9919         return nil, err
9920     }
9921
9922     // if only line type next line for trailer
9923     if !line == "trailer" {
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.NewReader(rd)
1355 s.Split(ScanLines)
1356
1357 line, err = s.ReadString()
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 = readStream(ctx, offset)
1373     if err != nil {
1374         return err
1375     }
1376     if offset, err = parseStream(ctx, offset, ctx); err != nil {
1377         log.Read.Printf("builderFragmentsStartingAt: error", err)
1378         // fix for corrupt input section.
1379         return hypothesisSection(ctx)
1380     }
1381 }
1382
1383 log.Read.Printf("builderFragmentsStartingAt: end")
1384
1385 return nil
1386 }
1387
1388 // Populate the cross reference table for this PDF file.
1389 // Note offset of first ref table entry.
1390 // Can be "ref" or indirect object reference or "is a obj"
1391 // Note that the ref table contains as many as there is a defined previous ref section
1392 // and build up the ref table along the way.
1393 func readRefTable(ctx *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 = dereferenceShuffleable(ctx, conf); err != nil {
574     return nil, err
575 }
576
577 // Some references write an invariant size into trailer.
578 // cctx.ShuffleableSize <= len(ctx.Shuffleable.Table) &
579 // cctx.Shuffleable.Size <= len(ctx.Shuffleable.Table)
580
581 log.Read.Print("Read: end")
582
583 return ctx, nil
584
585 // =====
586
587 // ScanIndex 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 newline or one carriage return or one newline.
591 // If the returned line is empty, it will be returned even if it was an error.
592 func scanIndex(data []byte, startIdx int64) (string, bool) {
593     if startIdx > len(data) - 1 {
594         return "", true
595     }
596     if startIdx < 0 {
597         return "", true
598     }
599     startIdx = bytes.IndexAny(data, "\r\n")
600     if startIdx < 0 {
601         return data[startIdx:], false
602     }
603     switch {
604     case startIdx >= 0 && index > 0:
605         if index < startIdx {
606             if index <= index {
607                 return index, false
608             }
609             return index, false
610         }
611         return index, false
612     }
613     return index, false
614 }

```

```

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.readHybrid & true
225 ) {
226     // I/O constant readers process hidden objects contained
227     // in %$%$ before continuing to process any previous %$%$.
228     // Previous %$%$Stream is expected to have free entries for hidden entries.
229     // No reason in %$%$Stream only.
230     if !ctx.readHex(0) {
231         err = parseHiddenHexStream(offsetHexStream, ctx); err == nil {
232             return nil, errors.Errorf("offsetHexStream: %s", err)
233         }
234     }
235     log.Debug.Println("parse(file=%s) end")
236     return offset, nil
237 }
238 // Return constant readers.
239
240 func scanIndexNew(s bufio.Scanner) (string, error) {
241     if s == nil || s.Err() != nil {
242         if s.Err() == nil {
243             return "", s.Err()
244         }
245         return "", errors.New("pdfcpu: scanIndexNew: returning nothing")
246     }
247     return s.Text(), nil
248 }
249
250 func scanIndex(s bufio.Scanner) (string, error) {
251     for i := 0; i <= i; i++ {
252         if s == nil || s.Err() != nil {
253             if s == nil {
254                 return "", s.Err()
255             }
256             if len(s) > 0 {
257                 break
258             }
259         }
260     }
261     i := strings.Index(s, "%$")
262     if i > 0 {
263         s := s[i:]
264     }
265 }

```

```

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 := m.parseDict(m.trailerDict, 1, 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 7.5.3, "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     m.headerVersion = headerVersion
996
997     var errPDFVersionError = errors.New("pdfdoc: headerVersion: corrupt pdf stream =
998         header version available")
999
1000     // Get the version of the file which holds the version of this document.
1001     // If we call this the header version.
1002     if err := m.seek(0, io.SeekStart), err == nil {
1003         m.headerVersion, err = m.readHeaderVersion()
1004         if err != nil {
1005             return
1006         }
1007     }
1008     buf := make([]byte, 20)
1009     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1010         return nil, &err.
1011     }
1012
1013     // Get the version of the file which holds the version of this document.
1014     // If we call this the header version.
1015     if err := m.seek(0, io.SeekStart), err == nil {
1016         m.headerVersion, err = m.readHeaderVersion()
1017         if err != nil {
1018             return
1019         }
1020     }
1021     buf := make([]byte, 20)
1022     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1023         return nil, &err.
1024     }
1025
1026     // Get the version of the file which holds the version of this document.
1027     // If we call this the header version.
1028     if err := m.seek(0, io.SeekStart), err == nil {
1029         m.headerVersion, err = m.readHeaderVersion()
1030         if err != nil {
1031             return
1032         }
1033     }
1034     buf := make([]byte, 20)
1035     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1036         return nil, &err.
1037     }
1038
1039     // Get the version of the file which holds the version of this document.
1040     // If we call this the header version.
1041     if err := m.seek(0, io.SeekStart), err == nil {
1042         m.headerVersion, err = m.readHeaderVersion()
1043         if err != nil {
1044             return
1045         }
1046     }
1047     buf := make([]byte, 20)
1048     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1049         return nil, &err.
1050     }
1051
1052     // Get the version of the file which holds the version of this document.
1053     // If we call this the header version.
1054     if err := m.seek(0, io.SeekStart), err == nil {
1055         m.headerVersion, err = m.readHeaderVersion()
1056         if err != nil {
1057             return
1058         }
1059     }
1060     buf := make([]byte, 20)
1061     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1062         return nil, &err.
1063     }
1064
1065     // Get the version of the file which holds the version of this document.
1066     // If we call this the header version.
1067     if err := m.seek(0, io.SeekStart), err == nil {
1068         m.headerVersion, err = m.readHeaderVersion()
1069         if err != nil {
1070             return
1071         }
1072     }
1073     buf := make([]byte, 20)
1074     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1075         return nil, &err.
1076     }
1077
1078     // Get the version of the file which holds the version of this document.
1079     // If we call this the header version.
1080     if err := m.seek(0, io.SeekStart), err == nil {
1081         m.headerVersion, err = m.readHeaderVersion()
1082         if err != nil {
1083             return
1084         }
1085     }
1086     buf := make([]byte, 20)
1087     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1088         return nil, &err.
1089     }
1090
1091     // Get the version of the file which holds the version of this document.
1092     // If we call this the header version.
1093     if err := m.seek(0, io.SeekStart), err == nil {
1094         m.headerVersion, err = m.readHeaderVersion()
1095         if err != nil {
1096             return
1097         }
1098     }
1099     buf := make([]byte, 20)
1100     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1101         return nil, &err.
1102     }
1103
1104     // Get the version of the file which holds the version of this document.
1105     // If we call this the header version.
1106     if err := m.seek(0, io.SeekStart), err == nil {
1107         m.headerVersion, err = m.readHeaderVersion()
1108         if err != nil {
1109             return
1110         }
1111     }
1112     buf := make([]byte, 20)
1113     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1114         return nil, &err.
1115     }
1116
1117     // Get the version of the file which holds the version of this document.
1118     // If we call this the header version.
1119     if err := m.seek(0, io.SeekStart), err == nil {
1120         m.headerVersion, err = m.readHeaderVersion()
1121         if err != nil {
1122             return
1123         }
1124     }
1125     buf := make([]byte, 20)
1126     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1127         return nil, &err.
1128     }
1129
1130     // Get the version of the file which holds the version of this document.
1131     // If we call this the header version.
1132     if err := m.seek(0, io.SeekStart), err == nil {
1133         m.headerVersion, err = m.readHeaderVersion()
1134         if err != nil {
1135             return
1136         }
1137     }
1138     buf := make([]byte, 20)
1139     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1140         return nil, &err.
1141     }
1142
1143     // Get the version of the file which holds the version of this document.
1144     // If we call this the header version.
1145     if err := m.seek(0, io.SeekStart), err == nil {
1146         m.headerVersion, err = m.readHeaderVersion()
1147         if err != nil {
1148             return
1149         }
1150     }
1151     buf := make([]byte, 20)
1152     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1153         return nil, &err.
1154     }
1155
1156     // Get the version of the file which holds the version of this document.
1157     // If we call this the header version.
1158     if err := m.seek(0, io.SeekStart), err == nil {
1159         m.headerVersion, err = m.readHeaderVersion()
1160         if err != nil {
1161             return
1162         }
1163     }
1164     buf := make([]byte, 20)
1165     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1166         return nil, &err.
1167     }
1168
1169     // Get the version of the file which holds the version of this document.
1170     // If we call this the header version.
1171     if err := m.seek(0, io.SeekStart), err == nil {
1172         m.headerVersion, err = m.readHeaderVersion()
1173         if err != nil {
1174             return
1175         }
1176     }
1177     buf := make([]byte, 20)
1178     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1179         return nil, &err.
1180     }
1181
1182     // Get the version of the file which holds the version of this document.
1183     // If we call this the header version.
1184     if err := m.seek(0, io.SeekStart), err == nil {
1185         m.headerVersion, err = m.readHeaderVersion()
1186         if err != nil {
1187             return
1188         }
1189     }
1190     buf := make([]byte, 20)
1191     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1192         return nil, &err.
1193     }
1194
1195     // Get the version of the file which holds the version of this document.
1196     // If we call this the header version.
1197     if err := m.seek(0, io.SeekStart), err == nil {
1198         m.headerVersion, err = m.readHeaderVersion()
1199         if err != nil {
1200             return
1201         }
1202     }
1203     buf := make([]byte, 20)
1204     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1205         return nil, &err.
1206     }
1207
1208     // Get the version of the file which holds the version of this document.
1209     // If we call this the header version.
1210     if err := m.seek(0, io.SeekStart), err == nil {
1211         m.headerVersion, err = m.readHeaderVersion()
1212         if err != nil {
1213             return
1214         }
1215     }
1216     buf := make([]byte, 20)
1217     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1218         return nil, &err.
1219     }
1220
1221     // Get the version of the file which holds the version of this document.
1222     // If we call this the header version.
1223     if err := m.seek(0, io.SeekStart), err == nil {
1224         m.headerVersion, err = m.readHeaderVersion()
1225         if err != nil {
1226             return
1227         }
1228     }
1229     buf := make([]byte, 20)
1230     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1231         return nil, &err.
1232     }
1233
1234     // Get the version of the file which holds the version of this document.
1235     // If we call this the header version.
1236     if err := m.seek(0, io.SeekStart), err == nil {
1237         m.headerVersion, err = m.readHeaderVersion()
1238         if err != nil {
1239             return
1240         }
1241     }
1242     buf := make([]byte, 20)
1243     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1244         return nil, &err.
1245     }
1246
1247     // Get the version of the file which holds the version of this document.
1248     // If we call this the header version.
1249     if err := m.seek(0, io.SeekStart), err == nil {
1250         m.headerVersion, err = m.readHeaderVersion()
1251         if err != nil {
1252             return
1253         }
1254     }
1255     buf := make([]byte, 20)
1256     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1257         return nil, &err.
1258     }
1259
1260     // Get the version of the file which holds the version of this document.
1261     // If we call this the header version.
1262     if err := m.seek(0, io.SeekStart), err == nil {
1263         m.headerVersion, err = m.readHeaderVersion()
1264         if err != nil {
1265             return
1266         }
1267     }
1268     buf := make([]byte, 20)
1269     if err := m.seek(m.headerVersion, io.SeekStart), err == nil {
1270         return nil, &err.
1271     }
1272
1273     // Get the version of the file which holds the version of this
```

```

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) ReadFile(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) ReadFile(offset int, string, 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("readFile: Read %d bytes", i)
1242         }
1243         if i % 100 == 0 {
1244             log.Printf("readFile: Read %d bytes", i)
1245         }
1246         if i % 100 == 0 {
1247             log.Printf("readFile: Read %d bytes", i)
1248         }
1249         if i % 100 == 0 {
1250             log.Printf("readFile: Read %d bytes", i)
1251         }
1252         if i % 100 == 0 {
1253             log.Printf("readFile: Read %d bytes", i)
1254         }
1255         if i % 100 == 0 {
1256             log.Printf("readFile: Read %d bytes", i)
1257         }
1258         if i % 100 == 0 {
1259             log.Printf("readFile: Read %d bytes", i)
1260         }
1261         if i % 100 == 0 {
1262             log.Printf("readFile: Read %d bytes", i)
1263         }
1264         if i % 100 == 0 {
1265             log.Printf("readFile: Read %d bytes", i)
1266         }
1267         if i % 100 == 0 {
1268             log.Printf("readFile: Read %d bytes", i)
1269         }
1270         if i % 100 == 0 {
1271             log.Printf("readFile: Read %d bytes", i)
1272         }
1273         if i % 100 == 0 {
1274             log.Printf("readFile: Read %d bytes", i)
1275         }
1276         if i % 100 == 0 {
1277             log.Printf("readFile: Read %d bytes", i)
1278         }
1279         if i % 100 == 0 {
1280             log.Printf("readFile: Read %d bytes", i)
1281         }
1282         if i % 100 == 0 {
1283             log.Printf("readFile: Read %d bytes", i)
1284         }
1285         if i % 100 == 0 {
1286             log.Printf("readFile: Read %d bytes", i)
1287         }
1288         if i % 100 == 0 {
1289             log.Printf("readFile: Read %d bytes", i)
1290         }
1291         if i % 100 == 0 {
1292             log.Printf("readFile: Read %d bytes", i)
1293         }
1294         if i % 100 == 0 {
1295             log.Printf("readFile: Read %d bytes", i)
1296         }
1297         if i % 100 == 0 {
1298             log.Printf("readFile: Read %d bytes", i)
1299         }
1300         if i % 100 == 0 {
1301             log.Printf("readFile: Read %d bytes", i)
1302         }
1303         if i % 100 == 0 {
1304             log.Printf("readFile: Read %d bytes", i)
1305         }
1306         if i % 100 == 0 {
1307             log.Printf("readFile: Read %d bytes", i)
1308         }
1309         if i % 100 == 0 {
1310             log.Printf("readFile: Read %d bytes", i)
1311         }
1312         if i % 100 == 0 {
1313             log.Printf("readFile: Read %d bytes", i)
1314         }
1315         if i % 100 == 0 {
1316             log.Printf("readFile: Read %d bytes", i)
1317         }
1318         if i % 100 == 0 {
1319             log.Printf("readFile: Read %d bytes", i)
1320         }
1321         if i % 100 == 0 {
1322             log.Printf("readFile: Read %d bytes", i)
1323         }
1324         if i % 100 == 0 {
1325             log.Printf("readFile: Read %d bytes", i)
1326         }
1327         if i % 100 == 0 {
1328             log.Printf("readFile: Read %d bytes", i)
1329         }
1330         if i % 100 == 0 {
1331             log.Printf("readFile: Read %d bytes", i)
1332         }
1333         if i % 100 == 0 {
1334             log.Printf("readFile: Read %d bytes", i)
1335         }
1336         if i % 100 == 0 {
1337             log.Printf("readFile: Read %d bytes", i)
1338         }
1339         if i % 100 == 0 {
1340             log.Printf("readFile: Read %d bytes", i)
1341         }
1342         if i % 100 == 0 {
1343             log.Printf("readFile: Read %d bytes", i)
1344         }
1345         if i % 100 == 0 {
1346             log.Printf("readFile: Read %d bytes", i)
1347         }
1348         if i % 100 == 0 {
1349             log.Printf("readFile: Read %d bytes", i)
1350         }
1351         if i % 100 == 0 {
1352             log.Printf("readFile: Read %d bytes", i)
1353         }
1354         if i % 100 == 0 {
1355             log.Printf("readFile: Read %d bytes", i)
1356         }
1357         if i % 100 == 0 {
1358             log.Printf("readFile: Read %d bytes", i)
1359         }
1360         if i % 100 == 0 {
1361             log.Printf("readFile: Read %d bytes", i)
1362         }
1363         if i % 100 == 0 {
1364             log.Printf("readFile: Read %d bytes", i)
1365         }
1366         if i % 100 == 0 {
1367             log.Printf("readFile: Read %d bytes", i)
1368         }
1369         if i % 100 == 0 {
1370             log.Printf("readFile: Read %d bytes", i)
1371         }
1372         if i % 100 == 0 {
1373             log.Printf("readFile: Read %d bytes", i)
1374         }
1375         if i % 100 == 0 {
1376             log.Printf("readFile: Read %d bytes", i)
1377         }
1378         if i % 100 == 0 {
1379             log.Printf("readFile: Read %d bytes", i)
1380         }
1381         if i % 100 == 0 {
1382             log.Printf("readFile: Read %d bytes", i)
1383         }
1384         if i % 100 == 0 {
1385             log.Printf("readFile: Read %d bytes", i)
1386         }
1387         if i % 100 == 0 {
1388             log.Printf("readFile: Read %d bytes", i)
1389         }
1390         if i % 100 == 0 {
1391             log.Printf("readFile: Read %d bytes", i)
1392         }
1393         if i % 100 == 0 {
1394             log.Printf("readFile: Read %d bytes", i)
1395         }
1396         if i % 100 == 0 {
1397             log.Printf("readFile: Read %d bytes", i)
1398         }
1399         if i % 100 == 0 {
1400             log.Printf("readFile: Read %d bytes", i)
1401         }
1402         if i % 100 == 0 {
1403             log.Printf("readFile: Read %d bytes", i)
1404         }
1405         if i % 100 == 0 {
1406             log.Printf("readFile: Read %d bytes", i)
1407         }
1408         if i % 100 == 0 {
1409             log.Printf("readFile: Read %d bytes", i)
1410         }
1411         if i % 100 == 0 {
1412             log.Printf("readFile: Read %d bytes", i)
1413         }
1414         if i % 100 == 0 {
1415             log.Printf("readFile: Read %d bytes", i)
1416         }
1417         if i % 100 == 0 {
1418             log.Printf("readFile: Read %d bytes", i)
1419         }
1420         if i % 100 == 0 {
1421             log.Printf("readFile: Read %d bytes", i)
1422         }
1423         if i % 100 == 0 {
1424             log.Printf("readFile: Read %d bytes", i)
1425         }
1426         if i % 100 == 0 {
1427             log.Printf("readFile: Read %d bytes", i)
1428         }
1429         if i % 100 == 0 {
1430             log.Printf("readFile: Read %d bytes", i)
1431         }
1432         if i % 100 == 0 {
1433             log.Printf("readFile: Read %d bytes", i)
1434         }
1435         if i % 100 == 0 {
1436             log.Printf("readFile: Read %d bytes", i)
1437         }
1438         if i % 100 == 0 {
1439             log.Printf("readFile: Read %d bytes", i)
1440         }
1441         if i % 100 == 0 {
1442             log.Printf("readFile: Read %d bytes", i)
1443         }
1444         if i % 100 == 0 {
1445             log.Printf("readFile: Read %d bytes", i)
1446         }
1447         if i % 100 == 0 {
1448             log.Printf("readFile: Read %d bytes", i)
1449         }
1450         if i % 100 == 0 {
1451             log.Printf("readFile: Read %d bytes", i)
1452         }
1453         if i % 100 == 0 {
1454             log.Printf("readFile: Read %d bytes", i)
1455         }
1456         if i % 100 == 0 {
1457             log.Printf("readFile: Read %d bytes", i)
1458         }
1459         if i % 100 == 0 {
1460             log.Printf("readFile: Read %d bytes", i)
1461         }
1462         if i % 100 == 0 {
1463             log.Printf("readFile: Read %d bytes", i)
1464         }
1465         if i % 100 == 0 {
1466             log.Printf("readFile: Read %d bytes", i)
1467         }
1468         if i % 100 == 0 {
1469             log.Printf("readFile: Read %d bytes", i)
1470         }
1471         if i % 100 == 0 {
1472             log.Printf("readFile: Read %d bytes", i)
1473         }
1474         if i % 100 == 0 {
1475             log.Printf("readFile: Read %d bytes", i)
1476         }
1477         if i % 100 == 0 {
1478             log.Printf("readFile: Read %d bytes", i)
1479         }
1480         if i % 100 == 0 {
1481             log.Printf("readFile: Read %d bytes", i)
1482         }
1483         if i % 100 == 0 {
1484             log.Printf("readFile: Read %d bytes", i)

```

```

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 offsetLastSection: positioned to offset: %d\n", offset)
346     return bufio.NewReader(rs), nil
347 }
348
349 // Get the file offset of the last R/WSection.
350 // Get the file and search backwards for the first occurrence of startword
351 // (offset)
352 func offsetLastSection(ctx *Context) (int64, error) {
353     rs := ctx.Read.Rs
354     var {
355         prevBuf, worked, bufSize,
356         bufSize, index = 512
357         offset
358     }
359     for i := 1; offset < 0; i++ {
360         if, err := rs.Seek(-index(i), io.SeekEnd)
361         if, err == nil {
362             return nil, errors.New("offset can't find last read section")
363         }
364         log.Read.Print("Scanning for offsetLastSection starting at %d\n", offset)
365         curBuf = make([]byte, bufSize)
366     }
367 }

```

```

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(strings.Trim(obj, " "))
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 := strings.Repeat(objectStreamIdcOffset, offset)
361             logDecompressPrint("parObjStreamIdc objectStreamIdc objStreamIdc = %s\n", dst)
362             obj := 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 := strings.Repeat(objectStreamIdcOffset, offset)
371             logDecompressPrint("parObjStreamIdc objectStreamIdc objStreamIdc = %s\n", dst)
372             obj := 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     return objArray
382 }

```

```

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     // We expect a stream and therefore "stream" before "endInfo" if "endInfo" within
641     // the buffer.
642     // There is no guarantee that "endInfo" is contained in this buffer for large
643     // streams.
644     if streamID < 0 || (streamID > 0 && !endInfoContained) {
645         return nil, errors.New("pdpic: parseRefStream: corrupt pdf file")
646     }
647
648     // Init object, parse buf
649     l := lineStream(streamID)
650
651     objectNumber, generationNumber, err := parseObjectAttributes(l)
652     if err != nil {
653         return nil, err
654     }
655
656     // Parse stream
657     log.Debug.Printf("parseRefStream: xrefInfo=%d genNum=%d, objectNumber=%d",
658         streamID, generationNumber)
659     log.Debug.Printf("parseRefStream: deferencing object %d\n", objectNumber)
660     o, err := parseObject(l)
661     if err != nil {
662         return nil, errors.Wrap(err, "parseRefStream: no object")
663     }
664
665     log.Debug.Printf("parseRefStream: we have an object: %d\n", o)
666
667     streamOffset := o.Offset
668     buf, err := s.readStreamContents(o, objectNumber, streamOffset)
669     if err != nil {
670         return nil, err
671     }
672
673     // We have an end stream object
674     err = s.parseRefInfo(buf, dict, ctx.XRefTable)
675     if err != nil {
676         return nil, err
677     }
678
679     // Parse stream and create streamable entries for embedded objects.
680     err = extractRefTableEntriesForStream(streamID, Content, dict, ctx)
681     if err != nil {
682         return nil, err
683     }
684
685     // Create streamable entry for this stream
686     entry := XRefTableEntry{
687         Index:      streamID,
688         Offset:     Offset,
689         Generation: generationNumber,
690         Object:     obj
691     }

```

```

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 < 0
280 // We search marker within line buffer.
281 streamlined = -1;
282 return
283 }
284
285 // We find the next stream marker.
286 streamlined = len('stream') + 1
287
288 // We find a stream marker of another object
289 if (ending > 0) do streamlined = ending + 1
290
291 // We find a stream marker of another object
292 streamlined = -1
293 }
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, streamid] (lineid
298 // error) {
299
300 // process: a gun obj ... obj dict ... stream ... data ... endstream ... endobj
301 // streamid ... streamid ...
302 // -1 if absent -1 if absent
303 absent
304
305 //Log-Mode.Println("buffer: begin")
306
307 endline, streamlined = -1, -1
308
309 for (ending < 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+len("startref"):]
393     posOfP := strings.Index(string(p), "MDOP")
394     if posOfP == -1 {
395         return nil, errors.New("pdpico: no matching MDOP for startref")
396     }
397
398     p = p[posOfP:]
399     offset, err := strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
400     if err == nil {
401         return nil, errors.New("pdpico: corrupted last xref section")
402     }
403 }
404
405 log.Read.Printf("Offset last xrefsection: %d\n", offset)
406
407 return bufOffset, nil
408 }
409
410 // Read next subsection entry and generate corresponding xref table entry.
411 func (parser *ParserTableEntry) xrefInfoScanner, xrefInfo *xrefInfo, objectNumber int)
412 {
413     log.Read.Printf("parser:xrefTableEntry: begin")
414
415     line, err = scanner()
416     if err == nil {
417         return err
418     }
419
420     if xrefInfo.Exists(objectNumber) {
421         log.Read.Printf("xrefTableEntry: end - Skip entry %d - already assigned", objectNumber)
422         return nil
423     }
424
425     fields = strings.Split(line)
426     if len(fields) == 1 {
427         log.Read.Printf("xrefTableEntry: end - Skip entry %d - already assigned", objectNumber)
428         return nil
429     }
430     return errors.New("pdpico: parser:xrefTableEntry: corrupt xref section")
431 }
432
433 }
434
435 }

```

```

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[0]
414     // @ts-ignore
415     i1 := xsd.m[1]
416     // @ts-ignore
417     i2 := xsd.m[2]
418
419     xdefEntryLen = "1" * i2 + "13"
420
421     // @ts-ignore
422     xdefEntryLen = "1" * xdefEntryLen + "13"
423
424     // @ts-ignore
425     log.Read_Printf("extract:objectfile:entriesFromObjectStream: begin xdefEntryLen = %s\n",
426         xdefEntryLen)
427
428     // @ts-ignore
429     if len(buf) > xdefEntryLen > 0 {
430         // @ts-ignore
431         return errors.New("pdcip: extract:objectfile:entriesFromObjectStream: corrupt
432             refstream")
433     }
434
435     // @ts-ignore
436     obfCount = len(xsd.Objects)
437
438     log.Read_Printf("extract:objectfile:entriesFromObjectStream: obfCount %d bufLen %d\n",
439         obfCount, xsd.bufLen)
440
441     // @ts-ignore
442     log.Read_Printf("extract:objectfile:entriesFromObjectStream: len(buf):%d\n",
443         len(buf))
444
445     // @ts-ignore
446     if len(buf) < obfCount * xdefEntryLen {
447         // Sometimes there is an additional zero entry not accounted for by "index".
448         // This is not a problem, but it is not clear if this is an error.
449         // @ts-ignore
450         return errors.New("pdcip: extract:objectfile:entriesFromObjectStream: corrupt
451             refstream")
452     }
453
454     // @ts-ignore
455     i := 0
456
457     // @ts-ignore
458     // bufLenLen interprets the content of buf as an int64.
459     // @ts-ignore
460     bufLen := func(buf []byte) (int64) {
461         // @ts-ignore
462         for i := range buf {
463             // @ts-ignore
464             i |= int64(buf[i])
465         }
466     }
467 }

```

```

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 *table) error {
679
680     log.ReadPrintln("parseTrailer begin")
681
682     if _, found = dict.FindEntry("root"); found {
683         if encryptObj := dict["root"]
684             if encryptObj != nil {
685                 shFileable.decrypt = decryptObjObjId
686                 log.ReadPrintln("parseTrailer: Encrypt object: %d\n",
687 shFileable.encrypt)
688             }
689         }
690
691         // =====
692         if shFileable.Size == nil {
693             size = d.GetSize()
694             if size != nil
695                 return errors.New("pdfproc: parseTrailerInfo: missing entry \"%s\"")
696             }
697         }
698         // Not reliable
699         // / Because after all read in.
700         shFileable.Size = size
701     }
702
703     if shFileable.Root == nil {
704         rootObjId = d.IndirectEntry("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.ReadPrint("scanner failed to read line: %s\n", line)
370 //
371 } else {
372     //
373     // s = string.Index(line, "\n")
374     //
375     if s < 0 {
376         //
377         // No go
378         //
379         k++
380         line = line[j+2]
381     }
382 //
383 } else {
384     //
385     // No go
386     //
387     if s < 0 {
388         //
389         // Check for dict
390         //
391         ok_err = bufio.ReadString()
392         //
393         if err == nil || ok {
394             return buf.String(), nil
395         }
396     }
397 //
398 } else {
399     k++
400     line = line[j+2]
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 //
1001 }
1002 //
1003 }
1004 //
1005 }
1006 //
1007 }
1008 //
1009 }
1010 //
1011 }
1012 //
1013 }
1014 //
1015 }
1016 //
1017 }
1018 //
1019 }
1020 //
1021 }
1022 //
1023 }
1024 //
1025 }
1026 //
1027 }
1028 //
1029 }
1030 //
1031 }
1032 //
1033 }
1034 //
1035 }
1036 //
1037 }
1038 //
1039 }
1040 //
1041 }
1042 //
1043 }
1044 //
1045 }
1046 //
1047 }
1048 //
1049 }
1050 //
1051 }
1052 //
1053 }
1054 //
1055 }
1056 //
1057 }
1058 //
1059 }
1060 //
1061 }
1062 //
1063 }
1064 //
1065 }
1066 //
1067 }
1068 //
1069 }
1070 //
1071 }
1072 //
1073 }
1074 //
1075 }
1076 //
1077 }
1078 //
1079 }
1080 //
1081 }
1082 //
1083 }
1084 //
1085 }
1086 //
1087 }
1088 //
1089 }
1090 //
1091 }
1092 //
1093 }
1094 //
1095 }
1096 //
1097 }
1098 //
1099 }
1100 //
1101 }
1102 //
1103 }
1104 //
11
```

```

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     withIndexof = false
9808 }
9809
9810 // finish obj
9811 offset += int64(int(line) - eofCount)
9812 bb = append(bb, s...)
9813 bb = append(bb, line...)
9814 s = strings.Index(line, "endobj")
9815 if i > s {
9816     l = string(bb)
9817     objMr, generation, err = parseObjectAttributes(s[l])
9818     if err == nil {
9819         return err
9820     }
9821 }
9822 if off == 0 {
9823     ctx.tailer[objMr] = &tableEntry{
9824         offset: false,
9825         offset: 0,
9826         generation: generation
9827     }
9828     bb = nil
9829     withIndexof = false
9830 }
9831 }
9832 return nil
9833 }
9834
9835 // Build an iterable by reading all objects or when specified
9836 func buildIterablesStartingAt(ctx *Context, offset uint64) error {
9837     log.Debug.Print("buildIterablesStartingAt: begin")
9838 }

```

[illegible]


```

1570         }
1571         return false
1572     }
1573
1574     // We found the last so, return true if after end of dict only whitespace,
1575     ok = strings.TrimSpace(dict) == ""
1576
1577     // Log, Read, Print if keyword is right after ENDNOTICE: end, %s\n", ok)
1578     return ok
1579 }
1580
1581 func buildFilterPipeline(ctx *Context, filterArray, decodePassesArray Array, decodePasses
1582 []string) (*PFFilter, error) {
1583     var filterPipeline []PFFilter
1584
1585     for i, f := range filterArray {
1586
1587         filterName, ok := f.(Name)
1588         if !ok {
1589             return nil, errors.New("pffco: buildFilterPipeline: filterArray elements
1590 [corrupt]")
1591         }
1592         if decodePasses == nil || decodePassesArray[i] == nil {
1593             filterPipeline = append(filterPipeline, PFFilter{Name:
1594                 filterName, decodePasses: nil})
1595             continue
1596         }
1597
1598         dict, ok := decodePassesArray[i].(Dict)
1599         if !ok {
1600             return nil, errors.New("pffco: buildFilterPipeline: i,IndirectRef)
1601         }
1602         if !ok := decodePassesArray[i].(IndirectRef) {
1603             return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s\n",
1604                 dict)
1605         }
1606         if s, err := dereferenceDict(dict, indirect.ObjectName.Value());
1607         if err == nil {
1608             return nil, err
1609         }
1610         dict = d
1611     }
1612
1613     filterPipeline = append(filterPipeline, PFFilter{Name: filterName.String(),
1614         decodePasses: dict})
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 //

```

[illegible]

```

1130 // Save the saveDecodedContentContent to ctx.content, id, saveStreams, objKey, govt, int,
1131 // err (error)
1132
1133 // Log Read, Print("saveDecodedContentContent: begin decode\n"), decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if cty == nil || cty.IsKey == nil {
1137     // If id is FilterPolicyIdName = 1 do so, FilterPolicyIdName = "Crypt":
1138     idContent = id.Raw
1139     return nil
1140 }
1141
1142 // Log
1143
1144 // Special case: If the length of the encoded data is 0, we do not need to decode
1145 // anything.
1146 if len(id.Raw) == 0 {
1147     idContent = id.Raw
1148     return nil
1149 }
1150
1151 // cty gets created after theStream parsing.
1152 // theStream is not encrypted.
1153 if cty == nil || cty.IsKey == nil {
1154     id.Raw, err = decryptRaw(id.Raw, objKey, govt, cty.IsKey, cty.AESStreams,
1155 cty.Ex)
1156     if err == nil {
1157         return err
1158     }
1159     // If id is not nil
1160     id = &idContent[idContent.Len() - 1]
1161     id.Raw.Length = id
1162 }
1163
1164 // If decode
1165     return nil
1166 }
1167
1168 // Actual decoding of content stream.
1169 err = decodeStream(id)
1170 if err == filter.StreamUnsupported {
1171     // err == nil
1172     return err
1173 }
1174
1175 // Log Read, Print("saveDecodedContentContent: end")
1176
1177 return nil
1178
1179 // Decode compressed objectTableEntry
1180 func decodeCompressedObjectTableEntry (table *Table, objectNumber int, entry
1181 *ObjectEntry) error {
1182     // Log Read, Print("decodeCompressedObjectTableEntry: compressed object id at %d\n",
1183 objectNumber, entry.ObjectStream, entry.ObjectStreamLen)
1184
1185 // Missing stream entry in reference object stream.
1186 // objectStreamEntryTableEntry, obj = table[id, find entry, ObjectStream]
1187 if obj ==

```

```

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

```

```

2350 }
2351 // Create a mutable version of the file (since it's in the catalog
2352 // and record this as rootVersion (as opposed to headerVersion).
2353 // Use identifyRootVersion (seeTable & rootFile) error {
2354     log.Read.Print('IdentifyRootVersion: begin')
2355     // Copy to get version from rootFile
2356     rootVersionStr = vHeaderTable.ParseRootVersion()
2357     if err == nil {
2358         return err
2359     }
2360     if rootVersionStr == nil {
2361         return nil
2362     }
2363     // Validate version and save corresponding constant to vHeaderTable.
2364     rootVersion, err = PDPVersion(rootVersionStr)
2365     if err == nil {
2366         return errors.Wrap(err, 'IdentifyRootVersion: unknown PDP Root version:
2367             '+rootVersionStr)
2368     }
2369     vHeaderTable.RootVersion = rootVersion
2370     // Since v1.1, the header version may be overridden by a version entry in the
2371     // vHeaderTable. The header version < v1.1
2372     log.Info.Print('IdentifyRootVersion: PDP version is %s - will ignore root
2373         version %s',
2374             vHeaderTable.HeaderVersion, rootVersionStr)
2375     log.Read.Print('IdentifyRootVersion: end')
2376     return nil
2377 }
2378 // Parse all Objects including stream content from file and save to the corresponding
2379 // headerFields.
2380 // Parse all Objects including object stream and Linearization dicts.
2381 // Use vHeaderTable.ParseObject (ctx <Context, conf <Configuration) error {
2382     log.Read.Print('ParseObject: begin')
2383     vHeaderTable = ctx.vHeaderTable
2384     // Note for unencrypted files.
2385     // Mandatory provide users to open & display file.
2386     // Access may be restricted (Open access privileges).
2387     // Optionally provide comments in order to gain unrestricted access.
2388     if err == vHeaderTable.Open() {
2389         return err
2390     }

```

```

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 := deferPreambleDictObj(ctx, indirRef.ObjectNumber.Value())
5050     if err != nil {
5051         return nil, err
5052     }
5053 }
5054
5055 //fmt.Printf("Decomposed filter obj: %v\n", o)
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         // Use local filterPipeline and w/o decode param
5064         return append(filterPipeline, PFFilterName: filterName, DecodedParam:
5065             nil), nil
5066     }
5067     d, ok := o.(Dict)
5068     if !ok {
5069         // o, err := o.(IndirectRef)
5070         if !ok {
5071             return nil, errors.Errorf("PFFilterPipeline corrupt Dict: %v\n", o)
5072         }
5073         o, err := deferPreambleDict(ctx, ir.ObjectNumber.Value())
5074         if err != nil {
5075             return nil, err
5076         }
5077     }
5078 }
5079
5080 // with decoded parameter.
5081 log.Printf("PFFilterPipeline: with decode param")
5082 return append(filterPipeline, PFFilterName: filterName, DecodedParam: d),
5083     nil
5084 }
5085 // filter pipeline.
5086
5087 // Array of filternames
5088 filterArray, ok := o.([]string)
5089 if !ok {
5090     return nil, errors.Errorf("PFFilterPipeline: Expected FilterArray corrupt, %v",
5091         o, s)
5092 }
5093
5094 // Optional array of decode 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 // On return object's destructor may potentially produce dereferencing object's
1574 stream Md, so need table entry, entry.Object(stream)
1575
1576 //
1577 // Object of class entry has to be Object(stream)
1578 // so, obj = Object(stream) & entry.Object(stream)
1579 if !ok {
1580     return errors.Errorf("decompressorNoTableEntry: problem dereferencing object's
1581 stream Md, no object stream", entry.Object(stream)
1582 )
1583 }
1584
1585 // Get indexmd object from Object(stream)
1586 o, err = sd.IndexmdObject(entry.Object(stream))
1587 if err != nil {
1588     return errors.Errorf("decompressorNoTableEntry: problem dereferencing
1589 object stream Md, entry.Object(stream)
1590 )
1591 }
1592
1593 // Save object to theTableEntry.
1594
1595 g := &
1596 entry.Object & o
1597 entry.Compression = 0
1598 entry.Expression = false
1599
1600 //
1601 // Table entry's decompressorNoTableEntry: end, obj &obj's "objMd",
1602 entry.Object(stream), entry.Object(stream), o)
1603
1604 return nil
1605
1606 //
1607 // Log interesting stream content.
1608 func LogStreamContent(i int) {
1609     switch o := o.(type) {
1610     case StreamMd:
1611         if o.Content == nil {
1612             log.Read.Printf("logStream no stream content")
1613         }
1614         if o.IsSpkgContent {
1615             //log.Read.Printf("content %s\n", StreamMd.Content)
1616         }
1617     case ObjectStreamMd:
1618         if o.Content == nil {
1619             log.Read.Printf("logStream no object stream content")
1620         }
1621         if o.IsSpkgContent {
1622             log.Read.Printf("logStream no object stream content %s\n", o.Content)
1623         }
1624         if o.IsObjArray {
1625             log.Read.Printf("logStream no object stream obj array")
1626         }
1627         if o.IsObjArray {
1628             log.Read.Printf("logStream no object stream obj array %s\n", o.ObjArray)
1629         }
1630     }
1631 }
1632
1633 //
1634 // Default:

```

[illegible]

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObjectObject(ctxs)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shFileEntry object assign a object either by parsing from file or pass
2131     // a decrypted object.
2132     err = 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 // If we have a stream object, found = dict.Fmt(DecompParams)
1453 if found != 0 {
1454     decodeParamsArr, ok = decodeParams.Array()
1455     if !ok {
1456         return nil, errors.New("pdcip: pdfFilterPipeline: expected decompParams
1457 array corrupt")
1458     }
1459 }
1460
1461 // /var.PdfDict("decompParams: ba1u", decodeParams)
1462
1463 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamsArr,
1464 decodeParams)
1465 if err != nil {
1466     log.Read.FilterPipeline("pdfFilterPipeline: err")
1467     return filterPipeline, err
1468 }
1469
1470 func streamDictForObj(c *Context, d Dict, objKey, streamIn int, streamFset
1471 *FileSet, ba1u *dict.StreamDict, err error) (StreamDict, error) {
1472     streamLength, streamLengthF = d.Length()
1473
1474     if streamLength < 0 {
1475         return sd, errors.New("pdcip: streamDictForObj: stream object without
1476 streamLength")
1477     }
1478
1479     filterPipeline, err = pdfFilterPipeline(c, d)
1480     if err == nil {
1481         return sd, err
1482     }
1483
1484     streamOffset = offset
1485
1486 // We have a stream object
1487 sd = NewStream(streamDict, streamOffset, streamLength, streamLengthF, filterPipeline)
1488
1489 log.Read.Print("streamDictForObj: end, streamObject %d\n", objKey)
1490
1491 return sd, nil
1492 }
1493
1494 func dictCtx *Context, d Dict, objKey, err, endId, streamIn int (d Dict, err
1495 error) {
1496     if ctx.EncKey == nil {
1497         ctx.EncKey = decryptPdfDict(d, objKey, err, ctx.EncKey, ctx.AES4Strings,
1498         ctx.B)
1499     }
1500     if err == nil {
1501         return nil, err
1502     }
1503 }
1504
1505 if condStr >= 0 && (streamCond < 0 || streamCond == condId) {
1506     log.Read.Print("dict: end, %d\n", objKey)
1507     d2 = d1
1508 }
1509 }

```

```

3730         return dc, nil
3731     }
3732 }
3733
3734 func dereferenceObject(cctx *Context, objectNumber int) (dict, error) {
3735     // 5.11.1
3736     // 5.11.1.1
3737     // 5.11.1.2
3738     // 5.11.1.3
3739     // 5.11.1.4
3740     // 5.11.1.5
3741     // 5.11.1.6
3742     // 5.11.1.7
3743     // 5.11.1.8
3744     // 5.11.1.9
3745     // 5.11.1.10
3746     // 5.11.1.11
3747     // 5.11.1.12
3748     // 5.11.1.13
3749     // 5.11.1.14
3750     // 5.11.1.15
3751     // 5.11.1.16
3752     // 5.11.1.17
3753     // 5.11.1.18
3754     // 5.11.1.19
3755     // 5.11.1.20
3756     // 5.11.1.21
3757     // 5.11.1.22
3758     // 5.11.1.23
3759     // 5.11.1.24
3760     // 5.11.1.25
3761     // 5.11.1.26
3762     // 5.11.1.27
3763     // 5.11.1.28
3764     // 5.11.1.29
3765     // 5.11.1.30
3766     // 5.11.1.31
3767     // 5.11.1.32
3768     // 5.11.1.33
3769     // 5.11.1.34
3770     // 5.11.1.35
3771     // 5.11.1.36
3772     // 5.11.1.37
3773     // 5.11.1.38
3774     // 5.11.1.39
3775     // 5.11.1.40
3776     // 5.11.1.41
3777     // 5.11.1.42
3778     // 5.11.1.43
3779     // 5.11.1.44
3780     // 5.11.1.45
3781     // 5.11.1.46
3782     // 5.11.1.47
3783     // 5.11.1.48
3784     // 5.11.1.49
3785     // 5.11.1.50
3786     // 5.11.1.51
3787     // 5.11.1.52
3788     // 5.11.1.53
3789     // 5.11.1.54
3790     // 5.11.1.55
3791     // 5.11.1.56
3792     // 5.11.1.57
3793     // 5.11.1.58
3794     // 5.11.1.59
3795     // 5.11.1.60
3796     // 5.11.1.61
3797     // 5.11.1.62
3798     // 5.11.1.63
3799     // 5.11.1.64
3800     // 5.11.1.65
3801     // 5.11.1.66
3802     // 5.11.1.67
3803     // 5.11.1.68
3804     // 5.11.1.69
3805     // 5.11.1.70
3806     // 5.11.1.71
3807     // 5.11.1.72
3808     // 5.11.1.73
3809     // 5.11.1.74
3810     // 5.11.1.75
3811     // 5.11.1.76
3812     // 5.11.1.77
3813     // 5.11.1.78
3814     // 5.11.1.79
3815     // 5.11.1.80
3816     // 5.11.1.81
3817     // 5.11.1.82
3818     // 5.11.1.83
3819     // 5.11.1.84
3820     // 5.11.1.85
3821     // 5.11.1.86
3822     // 5.11.1.87
3823     // 5.11.1.88
3824     // 5.11.1.89
3825     // 5.11.1.90
3826     // 5.11.1.91
3827     // 5.11.1.92
3828     // 5.11.1.93
3829     // 5.11.1.94
3830     // 5.11.1.95
3831     // 5.11.1.96
3832     // 5.11.1.97
3833     // 5.11.1.98
3834     // 5.11.1.99
3835     // 5.11.1.100
3836     // 5.11.1.101
3837     // 5.11.1.102
3838     // 5.11.1.103
3839     // 5.11.1.104
3840     // 5.11.1.105
3841     // 5.11.1.106
3842     // 5.11.1.107
3843     // 5.11.1.108
3844     // 5.11.1.109
3845     // 5.11.1.110
3846     // 5.11.1.111
3847     // 5.11.1.112
3848     // 5.11.1.113
3849     // 5.11.1.114
3850     // 5.11.1.115
3851     // 5.11.1.116
3852     // 5.11.1.117
3853     // 5.11.1.118
3854     // 5.11.1.119
3855     // 5.11.1.120
3856     // 5.11.1.121
3857     // 5.11.1.122
3858     // 5.11.1.123
3859     // 5.11.1.124
3860     // 5.11.1.125
3861     // 5.11.1.126
3862     // 5.11.1.127
3863     // 5.11.1.128
3864     // 5.11.1.129
3865     // 5.11.1.130
3866     // 5.11.1.131
3867     // 5.11.1.132
3868     // 5.11.1.133
3869     // 5.11.1.134
3870     // 5.11.1.135
3871     // 5.11.1.136
3872     // 5.11.1.137
3873     // 5.11.1.138
3874     // 5.11.1.139
3875     // 5.11.1.140
3876     // 5.11.1.141
3877     // 5.11.1.142
3878     // 5.11.1.143
3879     // 5.11.1.144
3880     // 5.11.1.145
3881     // 5.11.1.146
3882     // 5.11.1.147
3883     // 5.11.1.148
3884     // 5.11.1.149
3885     // 5.11.1.150
3886     // 5.11.1.151
3887     // 5.11.1.152
3888     // 5.11.1.153
3889     // 5.11.1.154
3890     // 5.11.1.155
3891     // 5.11.1.156
3892     // 5.11.1.157
3893     // 5.11.1.158
3894     // 5.11.1.159
3895     // 5.11.1.160
3896     // 5.11.1.161
3897     // 5.11.1.162
3898     // 5.11.1.163
3899     // 5.11.1.164
3900     // 5.11.1.165
3901     // 5.11.1.166
3902     // 5.11.1.167
3903     // 5.11.1.168
3904     // 5.11.1.169
3905     // 5.11.1.170
3906     // 5.11.1.171
3907     // 5.11.1.172
3908     // 5.11.1.173
3909     // 5.11.1.174
3910     // 5.11.1.175
3911     // 5.11.1.176
3912     // 5.11.1.177
3913     // 5.11.1.178
3914     // 5.11.1.179
3915     // 5.11.1.180
3916     // 5.11.1.181
3917     // 5.11.1.182
3918     // 5.11.1.183
3919     // 5.11.1.184
3920     // 5.11.1.185
3921     // 5.11.1.186
3922     // 5.11.1.187
3923     // 5.11.1.188
3924     // 5.11.1.189
3925     // 5.11.1.190
3926     // 5.11.1.191
3927     // 5.11.1.192
3928     // 5.11.1.193
3929     // 5.11.1.194
3930     // 5.11.1.195
3931     // 5.11.1.196
3932     // 5.11.1.197
3933     // 5.11.1.198
3934     // 5.11.1.199
3935     // 5.11.1.200
3936     // 5.11.1.201
3937     // 5.11.1.202
3938     // 5.11.1.203
3939     // 5.11.1.204
3940     // 5.11.1.205
3941     // 5.11.1.206
3942     // 5.11.1.207
3943     // 5.11.1.208
3944     // 5.11.1.209
3945     // 5.11.1.210
3946     // 5.11.1.211
3947     // 5.11.1.212
3948     // 5.11.1.213
3949     // 5.11.1.214
3950     // 5.11.1.215
3951     // 5.11.1.216
3952     // 5.11.1.217
3953     // 5.11.1.218
3954     // 5.11.1.219
3955     // 5.11.1.220
3956     // 5.11.1.221
3957     // 5.11.1.222
3958     // 5.11.1.223
3959     // 5.11.1.224
3960     // 5.11.1.225
3961     // 5.11.1.226
3962     // 5.11.1.227
3963     // 5.11.1.228
3964     // 5.11.1.229
3965     // 5.11.1
```

```

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

```

```

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 = handleLinearizationAndStats(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 logStream := entry.Object
2382 logStream := entry.Object
2383 logStream := entry.Object
2384 logStream := entry.Object
2385 logStream := entry.Object
2386 logStream := entry.Object
2387
2388 func processBidsAndCounts(defTable *XRefTable, D Dict) {
2389     for _, n := range o {
2390         match o1 := x.Table(
2391             case IndexNotFit:
2392                 entry, ok := defTable.LookupTableEntryForIndexDef(nal)
2393                 if ok {
2394                     entry.Count++
2395                 }
2396             case Dict:
2397                 processBidsAndCounts(defTable, o1)
2398             case Array:
2399                 processBidsAndCounts(defTable, o1)
2400         }
2401     }
2402 }

```

```

2079 //
2080 } else {
2081   id, ok := ctx.ID().ID(StringLiteral)
2082   if !ok {
2083     return nil, error.New("pdpctx: ID must contain hex literals or string
2084       literals");
2085   }
2086   id, err = Unescape(id.Value());
2087   if err != nil {
2088     return nil, err
2089   }
2090 }
2091
2092 return id, nil
2093 }
2094
2095 func needsOwnerAndSessionPassword(cnd CommandMode) bool {
2096   cnd == CHANGEOPT || cnd == CHANGEUSER || cnd == SETPERMISSIONS
2097 }
2098
2099 func handlePermissions(ctx *Context) error {
2100   // AE255 Validate parameters
2101   ok, err = validatePermissions(ctx)
2102   if err != nil {
2103     return err
2104   }
2105   if tok {
2106     return errors.New("pdpctx: corrupted permissions after upw ok")
2107   }
2108   // Only check session permissions for pdpctx processing.
2109   if hasWritePermissions(ctx.cmd.Ctx, Ctx_Owner) {
2110     return errors.New("pdpctx: insufficient access permissions")
2111   }
2112   return nil
2113 }
2114
2115 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2116   //
2117   ctx.t, err = supportGetEncryption(ctx, d)
2118   if err != nil {
2119     return err
2120   }
2121   ctx.t.ID, err = idbytes(ctx)
2122   if err != nil {
2123     return err
2124   }
2125   var ok bool
2126   //for Prefix("pw:"): kfo := ctx.kfo; ctx.OwnerNew, ctx.SessionNew
2127   // Validate the owner password aka_permissions/master_password
2128   ok, err = ValidationOwnerPassword(ctx)

```

[illegible][illegible]

```

1968 // Use the object stream directly for object stream reads.
1969 // If !sd, !isObject()
1970     return errors.New("pdcps: decodeObjectStream: corrupt object stream")
1971 }
1972
1973 // We have an object stream.
1974 // If !sd, err = objectStreamDict(svd)
1975 // If err == nil
1976     return errors.Wrap(err, "decodeObjectStream: problem dereferencing
1977 object stream svd", objectStream)
1978
1979 // Log Read, Print("decodeObjectStream: decoding object stream %d\n",
1980 // objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict object.
1984 // If err = readObjectStreamDict(svd) err == nil {
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding
1986 object stream svd", objectStream)
1987 }
1988
1989 // If ssvd.ObjectArray == nil {
1990     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1991 }
1992
1993 // Log Read, Print("decodeObjectStream: decoded object stream %d\n",
1994 // objectStream)
1995
1996 // Save object stream dict to ssvdEntryDict.
1997     entry.Object = ssvd
1998
1999     Log Read, Print("decodeObjectStream: end")
2000     return nil
2001 }
2002
2003 func handleLinearizationPanicDict(cxt *Context, obj Object, objStr int) error {
2004     // Log Read, linearized {
2005     // Commentation dict already processed.
2006     return nil
2007 }
2008
2009 // handle Linearization panic dict.
2010 // If d == c == obj (dict) obj == d, itLinearizationPanicDict() {
2011     handleLinearization := true
2012     cxt.LinearizationPanicDict = true
2013     Log Read, Print("handleLinearizationPanicDict: identified LinearizationOb
2014 ject")
2015
2016     a := d.ArrayEntry("pr")
2017
2018 }

```

```

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

```

```

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