

```

380
381
382 Copyright 2018 The pdfcpu Authors.
383
384 Licensed under the Apache License, Version 2.0 (the "License");
385 you may not use this file except in compliance with the License.
386 You may obtain a copy of the License at
387
388     http://www.apache.org/licenses/LICENSE-2.0
389
390 Unless required by applicable law or agreed to in writing, software
391 distributed under the License is distributed on an "AS IS" BASIS,
392 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
393 See the License for the specific language governing permissions and
394 limitations under the License.
395
396
397 package pdfcpu
398
399 import (
400     "bufio"
401     "bytes"
402     "io"
403     "log"
404     "strings"
405     "github.com/pdfcpu/pdfcpu/pkg/fontset"
406     "github.com/pdfcpu/pdfcpu/pkg/generator"
407     "github.com/pdfcpu/pdfcpu/pkg/output"
408 )
409
410 // Read reads a PDF file and builds an internal structure holding its cross
411 // reference information and the objects.
412 func Read(filePath string, conf *Configuration) (*Context, error) {
413     log.Infof("Reading %s (%v)", filePath, infofile)
414     if err := os.Open(filePath)
415     if err == nil {
416         return nil, errors.Errorf("can't open %s", infofile)
417     }
418
419     defer func() {
420         f.Close()
421     }()
422
423     return Read(f, conf)
424 }
425
426 // Read takes a reader/writer and generates a Context,
427 // which can be used for reconstruction containing a cross reference table.
428 func Read(w io.Writer, 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.Print("parseObjectTableEntry: Object %d is in use at offset%0d,
2317             generation%0d", objNumber, offset, generation)
2318             if offset == 0 {
2319                 log.Info.Print("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.Print("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.Print("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 }

```

```

443 // https://github.com/GoogleCloudPlatform/google-cloud-go/blob/master/storage/googlecloudstorage.go#L100
444 for i := 0; i < len(objects); i++ {
445     objectNumber := xsd_objects[i]
446
447     // Read object
448     listStart := i + 1
449     c2 := bufToUint64(listStart + 1252star11)
450     c3 := bufToUint64(listStart+12 + 1252star12+13)
451
452     var xbfTableEntry xbfTableEntry
453
454     switch object[i] {
455     case 0x00:
456         // Read object
457         log.Read_Print("object,xbfTableEntryFromRookStream: Object #", i)
458         // must, first get c2, generation, and uN
459         s = int(c2)
460
461         xbfTableEntry =
462             xbfTableEntry {
463                 Free: true,
464                 Compressed: false,
465                 Offset: 8c2,
466                 Generation: 9s
467             }
468     case 0x01:
469         // in object array
470         log.Read_Read("object,xbfTableEntryFromRookStream: Object # is in
471             list of offset, s, generation,uN", objectNumber, c2, c3)
472         s = int(c2)
473
474         xbfTableEntry =
475             xbfTableEntry {
476                 Free: false,
477                 Compressed: false,
478                 Offset: 8c2,
479                 Generation: 9s
480             }
481     case 0x02:
482         // compressed object
483         log.Read_Read("object,xbfTableEntryFromRookStream: Object # is
484             compressed", objectNumber, c2, c3)
485         objectNumber = int(c2)
486         objName = int(c3)
487
488         xbfTableEntry =
489             xbfTableEntry {
490                 Free: false,
491                 Compressed: true,
492                 ObjectStream: objNumberIndex,
493                 ObjectStreamIndex: objNameIndex
494             }
495     case ccs_Read_ObjectStream[objNumber] == true
496     }
497 }
498
499 if cts.XbfTable.Exists(objectNumber) {
500     log.Read_Read("object,xbfTableEntryFromRookStream: skip entry M -

```

```

672         if desttable == null {
673             // return error: how? (popup: parameterInfo: missing entry "Root"? )
674             return errors.New(popup: parameterInfo: missing entry "Root"? )
675         }
676         xhefTable.Root = desttableRef
677         log.Debug.Printf("parameterInfo: Root object: %s\n", xhefTable.Root)
678     }
679     if xhefTable.Info == nil {
680         // info: how? (popup: parameterInfo: missing entry "Info"? )
681         infoRef := d.LookupEntry("Info")
682         if infoRef == nil {
683             // Info: Info = InfoRef
684             xhefTable.Info = infoRef
685             log.Debug.Printf("parameterInfo: Info object: %s\n", xhefTable.Info)
686         }
687     }
688     if xhefTable.ID == nil {
689         idEntry = d.LookupEntry("ID")
690         if idEntry == nil {
691             // ID: ID = IDEntry
692             log.Debug.Printf("parameterInfo: ID object: %s\n", xhefTable.ID)
693         } else if xhefTable.IDEntry == nil {
694             // IDEntry: IDEntry = IDEntry
695             return errors.New(popup: parameterInfo: missing entry "ID"? )
696         }
697         return errors.New(popup: parameterInfo: missing entry "ID"? )
698     }
699     log.Debug.Printf("parameterInfo end")
700 }
701
702 // return nil, err
703 func parameterInfo(trailerDict Dict, ctx *Context) (xhefData, error) {
704     // log.Debug.Printf("parameterInfo begin")
705     xhefTable = ctx.XhefTable
706     err = parameterInfoFor(trailerDict, xhefTable)
707     if err == nil {
708         return nil, err
709     }
710     if err != nil {
711         return nil, err
712     }
713     if err != nil {
714         log.Debug.Printf("parameterInfo: found AdditionalStreams: %s\n", err)
715         a := value
716         for a := value {
717             a = value (infoRef, a)
718             a = append(a, infoRef)
719         }
720         xhefTable.AdditionalStreams = a
721     }
722     offset = trailerDict.Previous()
723     if offset == nil {
724         log.Debug.Printf("parameterInfo: previous xref table section offset: %s\n", offset)
725         offset = trailerDict.Previous()
726     }
727     offset = trailerDict.Previous()
728 }

```

```

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             return nil, err
9914         }
9915     }
9916
9917     // trailer or another area cable subsection
9918     if !line == " " {
9919         return nil, err
9920     }
9921
9922     // if empty line try next line for trailer
9923     if !line == " " {
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 cxx.NewReader(rsin = br
1345 cxx.NewReader(rsout = rdCount + br)
1346
1347 for offset := nil {
1348
1349     rd, err = newPositionReader(rs, offset)
1350     if err != nil {
1351         return err
1352     }
1353
1354     s := bufio.NewScanner(rd)
1355     s.Split(scanLines)
1356
1357     line, err := s.Scan()
1358     if err != nil {
1359         return err
1360     }
1361
1362     log.Read.Printf("line: %s\n", line)
1363
1364     if strings.TrimSpace(line) == "ref" {
1365         log.Read.Printf("builderFragmentsStartingAt: found seek section")
1366         if offset, err = parseSeekSections(s, cxx); err != nil {
1367             return err
1368         }
1369     } else {
1370         log.Read.Printf("builderFragmentsStartingAt: found seek stream")
1371
1372         rd, err := newPositionReader(rs, offset)
1373         if err != nil {
1374             return err
1375         }
1376
1377         if offset, err = parseSeekStream(rd, offset, cxx); err != nil {
1378             return err
1379         }
1380         // try to find a correct valid seek section.
1381         return hypothesisSection(cxx)
1382     }
1383 }
1384
1385 log.Read.Printf("builderFragmentsStartingAt: end")
1386
1387 return nil
1388 }
1389
1390 // Populate the cross reference table for this PDF file.
1391 // Note: offset of first seek table entry.
1392 // Can be "ref" or indirect object reference or "is a obj"
1393 // Can be "is a obj" or indirect object reference or "is a obj"
1394 // and build up the seek table along the way.
1395 func readHeaderTable(cxx *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 1.3 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 // ScanLine is a multi-function for a Scanner that returns each line of
588 // text, stripped of any trailing end-of-line marker. The returned line may
589 // be empty, may contain any number of carriage returns followed
590 // by one newline or no carriage return or one newline.
591 // If the returned line is empty, it will be returned even if it was an error.
592 func scanLine(data []byte, startIdx int) (string, bool) {
593     if startIdx == len(data) {
594         return "", true
595     }
596     if startIdx == len(data) - 1 {
597         return "", true
598     }
599     if startIdx == len(data) - 2 {
600         return "", true
601     }
602     if startIdx == len(data) - 3 {
603         return "", true
604     }
605     if startIdx == len(data) - 4 {
606         return "", true
607     }
608     if startIdx == len(data) - 5 {
609         return "", true
610     }
611     if startIdx == len(data) - 6 {
612         return "", true
613     }
614     if startIdx == len(data) - 7 {
615         return "", true
616     }
617     if startIdx == len(data) - 8 {
618         return "", true
619     }
620     if startIdx == len(data) - 9 {
621         return "", true
622     }
623     if startIdx == len(data) - 10 {
624         return "", true
625     }
626     if startIdx == len(data) - 11 {
627         return "", true
628     }
629     if startIdx == len(data) - 12 {
630         return "", true
631     }
632     if startIdx == len(data) - 13 {
633         return "", true
634     }
635     if startIdx == len(data) - 14 {
636         return "", true
637     }
638     if startIdx == len(data) - 15 {
639         return "", true
640     }
641     if startIdx == len(data) - 16 {
642         return "", true
643     }
644     if startIdx == len(data) - 17 {
645         return "", true
646     }
647     if startIdx == len(data) - 18 {
648         return "", true
649     }
650     if startIdx == len(data) - 19 {
651         return "", true
652     }
653     if startIdx == len(data) - 20 {
654         return "", true
655     }
656     if startIdx == len(data) - 21 {
657         return "", true
658     }
659     if startIdx == len(data) - 22 {
660         return "", true
661     }
662     if startIdx == len(data) - 23 {
663         return "", true
664     }
665     if startIdx == len(data) - 24 {
666         return "", true
667     }
668     if startIdx == len(data) - 25 {
669         return "", true
670     }
671     if startIdx == len(data) - 26 {
672         return "", true
673     }
674     if startIdx == len(data) - 27 {
675         return "", true
676     }
677     if startIdx == len(data) - 28 {
678         return "", true
679     }
680     if startIdx == len(data) - 29 {
681         return "", true
682     }
683     if startIdx == len(data) - 30 {
684         return "", true
685     }
686     if startIdx == len(data) - 31 {
687         return "", true
688     }
689     if startIdx == len(data) - 32 {
690         return "", true
691     }
692     if startIdx == len(data) - 33 {
693         return "", true
694     }
695     if startIdx == len(data) - 34 {
696         return "", true
697     }
698     if startIdx == len(data) - 35 {
699         return "", true
700     }
701     if startIdx == len(data) - 36 {
702         return "", true
703     }
704     if startIdx == len(data) - 37 {
705         return "", true
706     }
707     if startIdx == len(data) - 38 {
708         return "", true
709     }
710     if startIdx == len(data) - 39 {
711         return "", true
712     }
713     if startIdx == len(data) - 40 {
714         return "", true
715     }
716     if startIdx == len(data) - 41 {
717         return "", true
718     }
719     if startIdx == len(data) - 42 {
720         return "", true
721     }
722     if startIdx == len(data) - 43 {
723         return "", true
724     }
725     if startIdx == len(data) - 44 {
726         return "", true
727     }
728     if startIdx == len(data) - 45 {
729         return "", true
730     }
731     if startIdx == len(data) - 46 {
732         return "", true
733     }
734     if startIdx == len(data) - 47 {
735         return "", true
736     }
737     if startIdx == len(data) - 48 {
738         return "", true
739     }
740     if startIdx == len(data) - 49 {
741         return "", true
742     }
743     if startIdx == len(data) - 50 {
744         return "", true
745     }
746     if startIdx == len(data) - 51 {
747         return "", true
748     }
749     if startIdx == len(data) - 52 {
750         return "", true
751     }
752     if startIdx == len(data) - 53 {
753         return "", true
754     }
755     if startIdx == len(data) - 54 {
756         return "", true
757     }
758     if startIdx == len(data) - 55 {
759         return "", true
760     }
761     if startIdx == len(data) - 56 {
762         return "", true
763     }
764     if startIdx == len(data) - 57 {
765         return "", true
766     }
767     if startIdx == len(data) - 58 {
768         return "", true
769     }
770     if startIdx == len(data) - 59 {
771         return "", true
772     }
773     if startIdx == len(data) - 60 {
774         return "", true
775     }
776     if startIdx == len(data) - 61 {
777         return "", true
778     }
779     if startIdx == len(data) - 62 {
780         return "", true
781     }
782     if startIdx == len(data) - 63 {
783         return "", true
784     }
785     if startIdx == len(data) - 64 {
786         return "", true
787     }
788     if startIdx == len(data) - 65 {
789         return "", true
790     }
791     if startIdx == len(data) - 66 {
792         return "", true
793     }
794     if startIdx == len(data) - 67 {
795         return "", true
796     }
797     if startIdx == len(data) - 68 {
798         return "", true
799     }
800     if startIdx == len(data) - 69 {
801         return "", true
802     }
803     if startIdx == len(data) - 70 {
804         return "", true
805     }
806     if startIdx == len(data) - 71 {
807         return "", true
808     }
809     if startIdx == len(data) - 72 {
810         return "", true
811     }
812     if startIdx == len(data) - 73 {
813         return "", true
814     }
815     if startIdx == len(data) - 74 {
816         return "", true
817     }
818     if startIdx == len(data) - 75 {
819         return "", true
820     }
821     if startIdx == len(data) - 76 {
822         return "", true
823     }
824     if startIdx == len(data) - 77 {
825         return "", true
826     }
827     if startIdx == len(data) - 78 {
828         return "", true
829     }
830     if startIdx == len(data) - 79 {
831         return "", true
832     }
833     if startIdx == len(data) - 80 {
834         return "", true
835     }
836     if startIdx == len(data) - 81 {
837         return "", true
838     }
839     if startIdx == len(data) - 82 {
840         return "", true
841     }
842     if startIdx == len(data) - 83 {
843         return "", true
844     }
845     if startIdx == len(data) - 84 {
846         return "", true
847     }
848     if startIdx == len(data) - 85 {
849         return "", true
850     }
851     if startIdx == len(data) - 86 {
852         return "", true
853     }
854     if startIdx == len(data) - 87 {
855         return "", true
856     }
857     if startIdx == len(data) - 88 {
858         return "", true
859     }
860     if startIdx == len(data) - 89 {
861         return "", true
862     }
863     if startIdx == len(data) - 90 {
864         return "", true
865     }
866     if startIdx == len(data) - 91 {
867         return "", true
868     }
869     if startIdx == len(data) - 92 {
870         return "", true
871     }
872     if startIdx == len(data) - 93 {
873         return "", true
874     }
875     if startIdx == len(data) - 94 {
876         return "", true
877     }
878     if startIdx == len(data) - 95 {
879         return "", true
880     }
881     if startIdx == len(data) - 96 {
882         return "", true
883     }
884     if startIdx == len(data) - 97 {
885         return "", true
886     }
887     if startIdx == len(data) - 98 {
888         return "", true
889     }
890     if startIdx == len(data) - 99 {
891         return "", true
892     }
893     if startIdx == len(data) - 100 {
894         return "", true
895     }
896     if startIdx == len(data) - 101 {
897         return "", true
898     }
899     if startIdx == len(data) - 102 {
900         return "", true
901     }
902     if startIdx == len(data) - 103 {
903         return "", true
904     }
905     if startIdx == len(data) - 104 {
906         return "", true
907     }
908     if startIdx == len(data) - 105 {
909         return "", true
910     }
911     if startIdx == len(data) - 106 {
912         return "", true
913     }
914     if startIdx == len(data) - 107 {
915         return "", true
916     }
917     if startIdx == len(data) - 108 {
918         return "", true
919     }
920     if startIdx == len(data) - 109 {
921         return "", true
922     }
923     if startIdx == len(data) - 110 {
924         return "", true
925     }
926     if startIdx == len(data) - 111 {
927         return "", true
928     }
929     if startIdx == len(data) - 112 {
930         return "", true
931     }
932     if startIdx == len(data) - 113 {
933         return "", true
934     }
935     if startIdx == len(data) - 114 {
936         return "", true
937     }
938     if startIdx == len(data) - 115 {
939         return "", true
940     }
941     if startIdx
```

```

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("pdfproc: compressedObject(s) stream objects are not to be
302         stored in an object's stream")
303 }

```

```

347 // Create a new object table entry
348 [stream assign(obj, objectNumber)
349   ] else {
350     ct.table(objectNumber) = <obj>TableEntry
351   }
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.read.Hybrid
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.read.Hybrid {
223     ctx.read.Hybrid = true
224     ctx.read.bindingStream = true
225 }
226
227 // I/O constant readers process hidden objects contained
228 // in %$hex$ before continuing to process any previous %$hex$.
229 // Previous %$hex$ is expected to have free entries for hidden entries.
230 // No matter in %$hex$ format only.
231 if !ctx.read.Hex() {
232     err = parseHiddenBindingStream(offsetHexStream, ctx); err == nil {
233         return nil, errors.Errorf("no cross reference stream")
234     }
235 }
236
237 log.Debug.Println("parse(file=%s) end")
238
239 return offset, nil
240 }
241 }
242
243 func scanIndexNew(s bufio.Scanner) (string, error) {
244     if s == nil || s.Err() != nil {
245         if s.Err() == nil {
246             return "", s.Err()
247         }
248         return "", errors.New("pdfcpu: scanIndexNew: returning nothing")
249     }
250     return s.Text(), nil
251 }
252 }
253
254 func scanIndex(s bufio.Scanner) (string, error) {
255     for i := 0; i <= i; i++ {
256         if s == nil || s.Err() != nil {
257             if s == nil {
258                 return "", s.Err()
259             }
260             if len(s) > 0 {
261                 break
262             }
263         }
264         i = strings.Index(s, "%")
265         if i > 0 {
266             s = s[:i]
267         }
268     }
269 }

```

[illegible]

```

1190 // Read Read.Print("readFile: begin");
1191
1192 offset, err = offsetLastFileSection(ctx)
1193 if err != nil {
1194     return
1195 }
1196
1197 err = buildNewFileStartingAt(ctx, offset)
1198 if err != io.EOF {
1199     return errors.Wrapferr, "readFile: failed: unexpected eof")
1200 }
1201
1202 if err != nil {
1203     return
1204 }
1205
1206 // Log list of free objects (not the "free list").
1207 //log.Read.Print("freeList: %v", cty.FreeObjects)
1208
1209 // Ensure valid freeList of objects.
1210 err = cty.EnsureValidFreeList()
1211 if err != nil {
1212     return
1213 }
1214
1215 log.Read.Print("readFile: end")
1216
1217 return
1218 }
1219
1220 func growBuf(buf []byte, size int, rd io.Reader) ([]byte, error) {
1221
1222     // no more (large) size.
1223     if err := rd.Read(buf);
1224     if err != nil {
1225         return nil, err
1226     }
1227     //log.Read.Print("growBuf: Read %d bytes", n)
1228
1229     return append(buf, 0...), nil
1230 }
1231
1232 func maxStreamOffset(line string, streamid int) (off int) {
1233
1234     off = streamid + len("stream")
1235
1236     // Skip optional blanks.
1237     // TODO Should be able to optimize whitespace stream?
1238     for i := 0; i < len(off); i++ {
1239         if !isSpace(off[i]) {
1240             return
1241         }
1242     }
1243     if !isLine(off) {
1244         return
1245     }
1246     return
1247 }
1248
1249 // Skip the rest of the line.
1250
1251 // Skip the rest of the line.
1252
1253
1254

```

```

310         return index < 1, data[index], nil
311     }
312     // debug - debug
313     return index < 1, data[index], nil
314 }
315 case index < 0:
316     // We have a full carriage return terminated line.
317     return index + 1, data[index], nil
318 }
319 case index < 0:
320     // We have a full newline-terminated line.
321     return index + 1, data[index], nil
322 }
323 }
324 // If we're at EOF, we have a final, non-terminated line. Return it.
325 if atEOF {
326     return len(data), data, nil
327 }
328 // Request more data.
329 return 0, nil, nil
330 }
331 // bufio.NewReader(rs is ReaderSeeker, offset int64) (*bufio.Reader, error)
332 func newBufioReader(rs io.ReaderSeeker, offset int64) (*bufio.Reader, error) {
333     if rs == rs.Seek(offset, io.SeekStart), err == nil {
334         return nil, err
335     }
336     log.Read.Print("bufio.NewReader: positioned to offset: %d\n", offset)
337     return bufio.NewReader(rs), nil
338 }
339 // Get the file offset of the last R/WSection.
340 // Get the file and search backwards for the first occurrence of startword
341 // (offset)
342 func getLastRWSecction(ctxt *Context) (int64, error) {
343     rs := ctxt.Read.Rs
344     var {
345         prevBuf, wordLen []byte
346         bufSize int64 = 512
347         offset int64
348     }
349     for i := 1; offset < 0; i = {
350         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
351         if err == nil {
352             return lastOff, errors.New("previous can't find last r/w secction")
353         }
354         log.Read.Print("scanning for offsetLastR/WSection starting at %d\n", off)
355         curBuf := make([]byte, bufSize)
356     }
357 }

```

```

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's", v, endObjCount)
344     decompressContent()
345     modObj := decompressContent().objStreamIdc()
346     obj := strings.Fields(string(modObj))
347     if len(obj) % 2 != 0 {
348         return errors.New("pdcpu: parObjStreamIdc corrupt object stream idc")
349     }
350     // e.g., 10 8 11 25 = 2 Objects: 10 @ offset 0, #1 @ 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 += endObjStreamIdcOffset
359         if i % 2 == 1 {
360             dstr := string(decompressContent()[offset:offsetIdc])
361             logDecompressPrint("parObjStreamIdc objectStream = %s\n", dstr)
362             o, err := compressObject(dstr)
363             if err != nil {
364                 return err
365             }
366             logDecompressPrint("parObjStreamIdc: [hd] = obj %s\n", i/2+1, obj[i+1])
367             objArray = append(objArray, o)
368         } else {
369             if i == len(obj)-1 {
370                 dstr := string(decompressContent()[offset:offsetIdc])
371                 logDecompressPrint("parObjStreamIdc objectStream = %s\n", dstr)
372                 o, err := compressObject(dstr)
373                 if err != nil {
374                     return err
375                 }
376                 logDecompressPrint("parObjStreamIdc: [hd] = obj %s\n", i/2+1, obj[i+1])
377                 objArray = append(objArray, o)
378             }
379             offsetIdc = offset
380         }
381     }
382     return objArray
383 }

```

```

630         }
631         return nil, err
632     }
633 }
634
635 // ReadStream reads a stream from the given URL.
636 func ReadStream(url string) (parseRefStream, error) {
637     log.Debug.Printf("parseRefStream: url=%s", url)
638     streamID := 2
639     req, err := http.NewRequest("GET", url, nil)
640     if err != nil {
641         return nil, err
642     }
643     // We use a buffer and therefore "stream" before "endobj" if "endobj" within
644     // the stream. There is no guarantee that "endobj" is contained in this buffer for large
645     // streams.
646     if streamID < 0 || (streamID > 0 && !endobj < streamID) {
647         return nil, errors.New("pdfcpu: parseRefStream: corrupt pdf file")
648     }
649
650     // Init object, parse flow
651     o := &Object{
652         l: line(streamID, buf),
653     }
654     objectNumber, generationNumber, err := parseObjectAttributes(o)
655     if err != nil {
656         return nil, err
657     }
658
659     // Parse stream
660     log.Debug.Printf("parseRefStream: xrefInfo=%d genNum=%d, objectNumber=%d",
661         xrefInfo, genNum, objectNumber)
662     log.Debug.Printf("parseRefStream: referencing object %d\n", objectNumber)
663     o, err = parseObject(o)
664     if err != nil {
665         return nil, errors.Wrap(err, "parseRefStream: no object")
666     }
667
668     // Read stream
669     log.Debug.Printf("parseRefStream: we have an object: %d\n", o)
670
671     streamOffset := o.Offset
672     buf, err := ReadStream(streamID, o, objectNumber, streamOffset)
673     if err != nil {
674         return nil, err
675     }
676
677     // We have an end stream object
678     err = parseRefTableInfo(streamID, ctx.XRefTable)
679     if err != nil {
680         return nil, err
681     }
682
683     // Parse stream and create xrefTable entries for embedded objects.
684     err = extractRefTableEntriesForStream(streamID, content, dx, ctx)
685     if err != nil {
686         return nil, err
687     }
688
689     // Create xrefTable entry for this stream
690     entry = &XRefTableEntry{
691         Index: objectNumber,
692         Offset: offset,
693         Generation: generationNumber,
694         Object: obj,
695     }

```

```

780 return s1, n1
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()
788     return ok, err
789 }
790
791 func scanHeader(s bufio.Scanner, line string) (string, error) {
792
793     var buf bytes.Buffer
794     var err error
795     var i32 int32
796     var i64 int64
797     buf.WriteString(fmt.Sprintf("line: %s\n", line))
798
799     // Scan for dict start tag "\n".
800     for {
801         i32 = strings.Index(line, "\n")
802         if i32 >= 0 {
803             break
804         }
805         line, err = scanLine(s)
806         buf.WriteString(fmt.Sprintf("line: %s\n", line))
807         if err == nil {
808             return "", err
809         }
810     }
811 }
812
813 func line := line(s)
814 buf.WriteString(line)
815 buf.WriteString("\n")
816 buf.WriteString(fmt.Sprintf("scanHeader dictbuf after start tag: %s\n", line))
817
818 // Scan for dict tag "\n" but account for inner dicts.
819 line := line(s)
820
821 for {
822     if len(line) == 0 {
823         line, err = scanLine(s)
824         if err == nil {
825             return "", err
826         }
827     }
828     buf.WriteString(line)
829     buf.WriteString("\n")
830     buf.WriteString(fmt.Sprintf("scanHeader dictbuf next line: %s\n", line))
831 }
832
833 s := strings.Index(line, "\n")
834 if s <= 0 {
835     s := strings.Index(line, "\n")
836     if s >= 0 {

```

```

4820 if s[i] == 'mha' {
4821     colCount = 1
4822 } else if s[i] == 'oah' {
4823     colCount = 1
4824     if s[i] == 'mha' {
4825         colCount = 2
4826     }
4827 } else {
4828     return nil, 0, errorCorruptHeader
4829 }
4830
4831 log.Debug.Printf("headerVersion: %d, found header size: %d", pdfVersion)
4832
4833 return pdfVersion, colCount, nil
4834 }
4835
4836 // popadeparser is a back file digesting content file sections.
4837 // It populates the shiftable by reading in all indirect objects line by line
4838 // and works on the assumption of a single xref section - meaning no incremental
4839 // updates have been made.
4840 func ParsePopadeparser(ctxt *Context) error {
4841     s := bufio.NewScanner(ctxt)
4842     h := FrequencyGeneration
4843     ct := byteTable.NewTableEntry()
4844     Free: true,
4845     Offset: 0,
4846     Generation: 0
4847
4848     rs := ct.ReadRs
4849     rsCount := ct.ReadRs.Count
4850     var off, offset index
4851
4852     rd, err := math.Float64frombits(rs.Offset)
4853     if err == nil {
4854         return err
4855     }
4856
4857     s = bufio.NewReaderScanner(rs)
4858     s.Split(scanLines)
4859
4860     bb := []byte{}
4861     var c byte
4862     withinObj, bool
4863     withinHeader, bool
4864     withinTrailer, bool
4865
4866     for {
4867         line, err := scanLineFrom(s)
4868         if err == nil {
4869             break
4870         }
4871         if withinHeader {
4872             offset += int64(len(line) * colCount)
4873         }
4874         if withinTrailer {
4875             bb = append(bb, ' ')
4876         }
4877         if s == strings.Index(line, "startxref")
4878         if i > 0 {

```

```

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 // lastStreamMarker(streamed <int, int, line, line string) {
267
268     if (streamed > len(line)-len("stream")) {
269         // We found the "stream" stream marker.
270         streamed = -1
271         return
272     }
273
274     // We start searching after this stream marker.
275     bufpos = streamed + len("stream")
276
277     // Search for next stream marker.
278     > in string.IndexOf(line(bufpos, "stream"))
279     if (< 0)
280         // We cannot search within line buffer.
281         streamed = -1
282         return
283     }
284
285     // We found the next stream marker.
286     streamed = len("stream") + 1
287
288     if (ending < 0) do streamed >= 0 end if
289
290     // We found a stream marker of another object
291     streamed = -1
292 }
293
294 // process = PDF file buffer of sufficient size for parsing an object. > stream
295 func readInReader(buf []byte, endOfLine, streamed int, streamOffset int) error {
296     if err :=
297         // process: a gun obj ... obj dict ... stream ... data ... endstream ... endobj
298         // stream
299         // object
300         // object
301         // object
302         // object
303         // object
304         // object
305         // object
306         // object
307         // object
308         // object
309         // object
310         // object
311         // object
312         // object
313         // object
314         // object
315         // object
316         // object
317         // object
318         // object
319         // object
320         // object
321         // object
322         // object
323         // object
324         // object
325         // object
326         // object
327         // object
328         // object
329         // object
330         // object
331         // object
332         // object
333         // object
334         // object
335         // object
336         // object
337         // object
338         // object
339         // object
340         // object
341         // object
342         // object
343         // object
344         // object
345         // object
346         // object
347         // object
348         // object
349         // object
350         // object
351         // object
352         // object
353         // object
354         // object
355         // object
356         // object
357         // object
358         // object
359         // object
360         // object
361         // object
362         // object
363         // object
364         // object
365         // object
366         // object
367         // object
368         // object
369         // object
370         // object
371         // object
372         // object
373         // object
374         // object
375         // object
376         // object
377         // object
378         // object
379         // object
380         // object
381         // object
382         // object
383         // object
384         // object
385         // object
386         // object
387         // object
388         // object
389         // object
390         // object
391         // object
392         // object
393         // object
394         // object
395         // object
396         // object
397         // object
398         // object
399         // object
400         // object
401         // object
402         // object
403         // object
404         // object
405         // object
406         // object
407         // object
408         // object
409         // object
410         // object
411         // object
412         // object
413         // object
414         // object
415         // object
416         // object
417         // object
418         // object
419         // object
420         // object
421         // object
422         // object
423         // object
424         // object
425         // object
426         // object
427         // object
428         // object
429         // object
430         // object
431         // object
432         // object
433         // object
434         // object
435         // object
436         // object
437         // object
438         // object
439         // object
440         // object
441         // object
442         // object
443         // object
444         // object
445         // object
446         // object
447         // object
448         // object
449         // object
450         // object
451         // object
452         // object
453         // object
454         // object
455         // object
456         // object
457         // object
458         // object
459         // object
460         // object
461         // object
462         // object
463         // object
464         // object
465         // object
466         // object
467         // object
468         // object
469         // object
470         // object
471         // object
472         // object
473         // object
474         // object
475         // object
476         // object
477         // object
478         // object
479         // object
480         // object
481         // object
482         // object
483         // object
484         // object
485         // object
486         // object
487         // object
488         // object
489         // object
490         // object
491         // object
492         // object
493         // object
494         // object
495         // object
496         // object
497         // object
498         // object
499         // object
500         // object
501         // object
502         // object
503         // object
504         // object
505         // object
506         // object
507         // object
508         // object
509         // object
510         // object
511         // object
512         // object
513         // object
514         // object
515         // object
516         // object
517         // object
518         // object
519         // object
520         // object
521         // object
522         // object
523         // object
524         // object
525         // object
526         // object
527         // object
528         // object
529         // object
530         // object
531         // object
532         // object
533         // object
534         // object
535         // object
536         // object
537         // object
538         // object
539         // object
540         // object
541         // object
542         // object
543         // object
544         // object
545         // object
546         // object
547         // object
548         // object
549         // object
550         // object
551         // object
552         // object
553         // object
554         // object
555         // object
556         // object
557         // object
558         // object
559         // object
560         // object
561         // object
562         // object
563         // object
564         // object
565         // object
566         // object
567         // object
568         // object
569         // object
570         // object
571         // object
572         // object
573         // object
574         // object
575         // object
576         // object
577         // object
578         // object
579         // object
580         // object
581         // object
582         // object
583         // object
584         // object
585         // object
586         // object
587         // object
588         // object
589         // object
590         // object
591         // object
592         // object
593         // object
594         // object
595         // object
596         // object
597         // object
598         // object
599         // object
600         // object
601         // object
602         // object
603         // object
604         // object
605         // object
606         // object
607         // object
608         // object
609         // object
610         // object
611         // object
612         // object
613         // object
614         // object
615         // object
616         // object
617         // object
618         // object
619         // object
620         // object
621         // object
622         // object
623         // object
624         // object
625         // object
626         // object
627         // object
628         // object
629         // object
630         // object
631         // object
632         // object
633         // object
634         // object
635         // object
636         // object
637         // object
638         // object
639         // object
640         // object
641         // object
642         // object
643         // object
644         // object
645         // object
646         // object
647         // object
648         // object
649         // object
650         // object
651         // object
652         // object
653         // object
654         // object
655         // object
656         // object
657         // object
658         // object
659         // object
660         // object
661         // object
662         // object
663         // object
664         // object
665         // object
666         // object
667         // object
668         // object
669         // object
670         // object
671         // object
672         // object
673         // object
674         // object
675         // object
676         // object
677         // object
678         // object
679         // object
680         // object
681         // object
682         // object
683         // object
684         // object
685         // object
686         // object
687         // object
688         // object
689         // object
690         // object
691         // object
692         // object
693         // object
694         // object
695         // object
696         // object
697         // object
698         // object
699         // object
700         // object
701         // object
702         // object
703         // object
704         // object
705         // object
706         // object
707         // object
708         // object
709         // object
710         // object
711         // object
712         // object
713         // object
714         // object
715         // object
716         // object
717         // object
718         // object
719         // object
720         // object
721         // object
722         // object
723         // object
724         // object
725         // object
726         // object
727         // object
728         // object
729         // object
730         // object
731         // object
732         // object
733         // object
734         // object
735         // object
736         // object
737         // object
738         // object
739         // object
740         // object
741         // object
742         // object
743         // object
744         // object
745         // object
746         // object
747         // object
748         // object
749         // object
750         // object
751         // object
752         // object
753         // object
754         // object
755         // object
756         // object
757         // object
758         // object
759         // object
760         // object
761         // object
762         // object
763         // object
764         // object
765         // object
766         // object
767         // object
768         // object
769         // object
770         // object
771         // object
772         // object
773         // object
774         // object
775         // object
776         // object
777         // object
778         // object
779         // object
780         // object
781         // object
782         // object
783         // object
784         // object
785         // object
786         // object
787         // object
788         // object
789         // object
790         // object
791         // object
792         // object
793         // object
794         // object
795         // object
796         // object
797         // object
798         // object
799         // object
800         // object
801         // object
802         // object
803         // object
804         // object
805         // object
806         // object
807         // object
808         // object
809         // object
810         // object
811         // object
812         // object
813         // object
814         // object
815         // object
816         // object
817         // object
818         //
```

```

363 // https://stackoverflow.com/questions/4913460/using-std-weak-map
374
375         err = r.Read(cursor)
376         if err == nil {
377             return nil, err
378         }
379     }
380
381     workBuf = curBuf
382     if preBuf == nil {
383         workBuf = append(curBuf, preBuf...)
384     }
385
386     j := strings.LastIndex(string(workBuf), "startref")
387     if j == -1 {
388         preBuf = curBuf
389         continue
390     }
391
392     p = workBuf[j+1len(string(workBuf)):]
393     posBuf = strings.Index(string(p), "MEOF")
394     if posBuf == -1 {
395         return nil, errors.New("pfcbuf: no matching MEOF for startref")
396     }
397
398     p = p[posBuf:]
399     offset, err = strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
400     if err == nil {
401         return nil, errors.New("pfcbuf: corrupted last xref section")
402     }
403 }
404
405 log.Read.Print("Offset last xref section: %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) xrefIoScanner, xrefBuf *xrefTable, objectNumber int)
412 {
413     log.Read.Print("parser:TableEntry: begin")
414
415     line, err = scanner()
416     if err == nil {
417         return err
418     }
419
420     if xrefBuf.Exists(objectNumber) {
421         log.Read.Print("parser:TableEntry: 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.Print("parser:TableEntry: end - Skip entry %d - already assigned", objectNumber)
428         return nil
429     }
430     return errors.New("pfcbuf: parser:TableEntry: corrupt xref subsection")
431 }
432
433 }
434
435 }

```

```

380 // @ts-ignore
390 std::ostringstream objArray
391
392     log.Read_Printf("paramsJsonObjectStream")
393
394     return nil
395
396 // for each object embedded in this xdrStream create the corresponding xdr table
397 // that shall be present in the stream
398 // @ts-ignore
399     xdrTableEntry["xdrJsonObjectStream"] = { objArray, xdrJsonObjectStream, xdr
400         JsonObjectStream }
401
402     log.Read_Printf("xdrJsonObjectStreamEntry xdrJsonObjectStream begin")
403
404     // Note:
405     // * A value of zero for an element in the m array indicates that the corresponding
406     //   element shall not be present in the stream
407     // * The default value shall be zero, if there is none
408     // * If an element is zero, the type field shall not be present, and shall
409     //   default to type: int
410
411     int m = xdr.m[0]
412     int n = xdr.m[1]
413     int o = xdr.m[2]
414
415     xdrEntryStream = "1 * 12 + 13"
416     xdrTableEntry["xdrJsonObjectStreamEntry xdrJsonObjectStream: begin xdrEntryStream"] =
417         xdrEntryStream
418
419     if len(xdr.m) > 3 {
420         return errors.New("pdcip: extractJsonObjectEntryFromXdrStream: corrupt
421             stream")
422     }
423
424     objCount = len(xdr.objs)
425     log.Read_Printf("xdrJsonObjectStreamEntry xdrJsonObjectStream: objCount %d",
426         objCount)
427     objCount = len(xdr.objs)
428
429     log.Read_Printf("xdrJsonObjectStreamEntry xdrJsonObjectStream: len(xdr.objs) %d",
430         objCount)
431     if len(xdr.o) < objCount {
432         // Sometimes there is an additional zero entry not accounted for by "index".
433         // This means that there is an error and do not treat this as an error
434         return errors.New("pdcip: extractJsonObjectEntryFromXdrStream: corrupt
435             stream")
436     }
437
438     j = 0
439
440     // bufio.NewReader interprets the content of buf as an int64.
441     // therefore we have to force it to be a float64
442     for i := range buf {
443         // @ts-ignore
444         i = int64(b)
445     }
446 }
447

```

```

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 := newPositionalHeaderReader(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, nil
675
676 // =====
677 // Parse trailer dict and return any offset of a previous shFileable section.
678 func parseTrailerShFileable(Dict, shFileable *shFileable) error {
679
680     log.ReadPrintln("parseTrailerFile begin")
681
682     if _, found = Dict.Fields["encrypt"]; found {
683         // decryptObj := decryptObjShFileable(encrypt)
684         if encryptObjShFileable != nil {
685             shFileable.decrypt = decryptObjShFileable
686             log.ReadPrintln("parseTrailerFile: Encrypt object: %d\n",
687 shFileable.encrypt)
688         }
689     }
690
691     // =====
692     if shFileable.Size == nil {
693         size = d.GetSize()
694     }
695     if size != nil {
696         return errors.New("pdfproc: parseTrailerFile: missing entry \"Size\"")
697     }
698     // Not reliable
699     // /atches after all read in.
700     shFileable.Size = size
701
702     // =====
703     if shFileable.Root == nil {
704         rootObjId = d.IndirectRefEntry("root")
705     }
706 }

```

```

340 //
341 // If k == 0
342 //
343 // Check for err
344 //
345 ok_err = is2(buf.String())
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 = scanline(s)
362 //
363 if err == nil {
364     return "", err
365 }
366 //
367 buf.WriteString(line)
368 buf.WriteString(" ")
369 //
370 log.Reads.Printf("scan trailer dict on next line: %s\n", line)
371 //
372 } else {
373     //
374     // %s is string.Index(line, " ")
375     //
376     if j < 0 {
377         j = 0
378     }
379     //
380     line = line[j+2]
381 //
382 } else {
383     //
384     // Check for dict
385     //
386     ok_err = is1(buf.String())
387     if err == nil && ok {
388         return buf.String(), nil
389     }
390 //
391 } else {
392     k++
393 }
394 //
395 // line = line[j+2]
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 //
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 //
1105
```

```

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 >= 8 {
9789     bb.append(bb, line...)
9790     withIndexof = true
9791     continue
9792 }
9793 l = strings.Index(line, "eof")
9794 if i >= 8 {
9795     offset += int64(int(l)line) + eofCount
9796     withIndexof = true
9797     continue
9798 }
9799 if withIndexof {
9800     s = strings.Index(line, "obj")
9801     if i >= 8 {
9802         withIndexof = true
9803         off += offset
9804         bb += append(bb, line[i:s]...)
9805     }
9806     offset += int64(int(l)line) + eofCount
9807     withIndexof = true
9808 }
9809
9810 // finish
9811 offset += int64(int(l)line) + eofCount
9812 bb += append(bb, s...)
9813 bb += append(bb, line...)
9814 s = strings.Index(line, "eofobj")
9815 if i >= 8 {
9816     l = string(bb)
9817     objMr, generation, err = parseObjectAttributes(s)
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 generation
9836 func buildIteratableByStartingAt(ctx *Context, offset uint64) error {
9837     log.Debug.Println("buildIteratableByStartingAt: begin")
9838 }

```

```

1307 // https://github.com/jefflau/StreamTokenizer
1308
1309 line in string(buf)
1310 endline = strings.Index(line, "\nend")
1311 streamline = strings.Index(line, "stream")
1312
1313 if endline > 0 && (streamline < 0 || streamline > endline) {
1314     // No stream marker in buf detected.
1315     break
1316 }
1317
1318 // For very rare cases where "stream" also occurs within buf dict
1319 // we need to find the last "stream" marker before a position and marker.
1320 // For streamline > 0, we need to find the last occurrence of dict(line, streamline)
1321 // lastStreamMarker(streamline, endline, line)
1322 }
1323
1324 log.Debug.Printf("buffer: endline=%d streamline=%d\n", endline, streamline)
1325
1326 if streamline > 0 {
1327     // streamOffset = the offset where the actual stream data begins.
1328     // In dict after the buf call after "stream"
1329     streamOffset = lastOccurrenceOfDict(line, streamline)
1330
1331     // skip = 1 for optional whitespace + nl (max 2 chars)
1332     next = streamOffset + len("stream") + skip
1333
1334     if len(line) < next {
1335         // to prevent buffer overflow.
1336         buf, err = growBuf(buf, next-len(line), nil)
1337         if err == nil {
1338             return nil, 0, 0, err
1339         }
1340     }
1341     line = string(buf)
1342 }
1343
1344 streamOffset = lastOccurrenceOfDict(line, streamline)
1345
1346 // (1) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1347 // streamOffset)
1348
1349 // (2) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1350 // streamOffset)
1351
1352 return buf, endline, streamline, streamOffset, nil
1353
1354 // (3) buf, endline, and dict, dictIndex, dictIndex
1355 // (4) buf, endline, and dict, dictIndex, dictIndex
1356
1357 func getNextStreamIndexAfterStream(buf string, streamline) bool {
1358     // (1) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1359     // streamOffset)
1360
1361     // (2) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1362     // streamOffset)
1363
1364     // (3) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1365     // streamOffset)
1366
1367     // (4) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1368     // streamOffset)
1369
1370     // (5) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1371     // streamOffset)
1372
1373     // (6) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1374     // streamOffset)
1375
1376     // (7) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1377     // streamOffset)
1378
1379     // (8) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1380     // streamOffset)
1381
1382     // (9) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1383     // streamOffset)
1384
1385     // (10) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1386     // streamOffset)
1387
1388     // (11) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1389     // streamOffset)
1390
1391     // (12) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1392     // streamOffset)
1393
1394     // (13) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1395     // streamOffset)
1396
1397     // (14) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1398     // streamOffset)
1399
1400     // (15) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1401     // streamOffset)
1402
1403     // (16) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1404     // streamOffset)
1405
1406     // (17) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1407     // streamOffset)
1408
1409     // (18) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1410     // streamOffset)
1411
1412     // (19) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1413     // streamOffset)
1414
1415     // (20) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1416     // streamOffset)
1417
1418     // (21) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1419     // streamOffset)
1420
1421     // (22) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1422     // streamOffset)
1423
1424     // (23) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1425     // streamOffset)
1426
1427     // (24) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1428     // streamOffset)
1429
1430     // (25) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1431     // streamOffset)
1432
1433     // (26) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1434     // streamOffset)
1435
1436     // (27) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1437     // streamOffset)
1438
1439     // (28) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1440     // streamOffset)
1441
1442     // (29) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1443     // streamOffset)
1444
1445     // (30) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1446     // streamOffset)
1447
1448     // (31) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1449     // streamOffset)
1450
1451     // (32) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1452     // streamOffset)
1453
1454     // (33) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1455     // streamOffset)
1456
1457     // (34) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1458     // streamOffset)
1459
1460     // (35) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1461     // streamOffset)
1462
1463     // (36) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1464     // streamOffset)
1465
1466     // (37) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1467     // streamOffset)
1468
1469     // (38) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1470     // streamOffset)
1471
1472     // (39) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1473     // streamOffset)
1474
1475     // (40) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1476     // streamOffset)
1477
1478     // (41) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1479     // streamOffset)
1480
1481     // (42) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1482     // streamOffset)
1483
1484     // (43) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1485     // streamOffset)
1486
1487     // (44) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1488     // streamOffset)
1489
1490     // (45) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1491     // streamOffset)
1492
1493     // (46) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1494     // streamOffset)
1495
1496     // (47) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1497     // streamOffset)
1498
1499     // (48) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1500     // streamOffset)
1501
1502     // (49) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1503     // streamOffset)
1504
1505     // (50) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1506     // streamOffset)
1507
1508     // (51) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1509     // streamOffset)
1510
1511     // (52) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1512     // streamOffset)
1513
1514     // (53) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1515     // streamOffset)
1516
1517     // (54) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1518     // streamOffset)
1519
1520     // (55) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1521     // streamOffset)
1522
1523     // (56) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1524     // streamOffset)
1525
1526     // (57) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1527     // streamOffset)
1528
1529     // (58) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1530     // streamOffset)
1531
1532     // (59) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1533     // streamOffset)
1534
1535     // (60) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1536     // streamOffset)
1537
1538     // (61) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1539     // streamOffset)
1540
1541     // (62) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1542     // streamOffset)
1543
1544     // (63) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1545     // streamOffset)
1546
1547     // (64) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1548     // streamOffset)
1549
1550     // (65) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1551     // streamOffset)
1552
1553     // (66) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1554     // streamOffset)
1555
1556     // (67) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1557     // streamOffset)
1558
1559     // (68) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1560     // streamOffset)
1561
1562     // (69) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1563     // streamOffset)
1564
1565     // (70) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1566     // streamOffset)
1567
1568     // (71) buf, endline, streamline, and, returned bufio.Reader streamOffset=>buf, len(buf)
1569     // streamOffset)
1570
1571
```



```

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

```

[illegible]

```

1130 // Save the saveDecodedContentContent to ctx.content, id, saveStreams, objKey, goenv 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 ctx.will nil on ctx.filterKey == nil {
1137     if ctx.filterPolicyName == 1 do do.FilterPolicyName(), Name == "Crypt" {
1138         return nil
1139     }
1140 }
1141
1142 // Special case: If the length of the encoded data is 0, we do not need to decode
1143 anything.
1144 if ctx.len(StdRaw) == 0 {
1145     StdContent = StdRaw
1146     return nil
1147 }
1148
1149 // Std gets created after StdStream parsing.
1150 // StdStreams are not encrypted.
1151 if ctx.will == StdRaw {
1152     StdRaw, err = decryptStream(StdRaw, objKey, goenv, ctx.KeyKey, ctx.AESStreams,
1153         ctx.Ex)
1154     if err == nil {
1155         return err
1156     }
1157     StdRaw =
1158         len(StdRaw)
1159     StdStream.Length = 0
1160 }
1161
1162 // If decode
1163     return nil
1164 }
1165
1166 // Actual decoding of content stream.
1167 err = decodeStream()
1168 if err == filter.StreamSupportFilter {
1169     err = nil
1170 }
1171
1172 if err == nil {
1173     return err
1174 }
1175
1176 Log.Read.Print("saveDecodedContentContent: end")
1177
1178 return nil
1179
1180 // Decode compressed objectTableEntry
1181 func decodeCompressedObjectTableEntry (ctx *Context, objTable *Table, objectNumber int, entry
1182     *ObjectEntry) error {
1183     Log.Read.Print("decodeCompressedObjectTableEntry: compressed object id at %d\n"),
1184         objectNumber, entry.ObjectStream, entry.ObjectStreamLen
1185
1186     // Reading stream entry at reference object stream.
1187     objStreamLen, objTable, findEntry (objStreamLen)
1188     if obj {

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry m, objIdR")
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 obj+objIdR")
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", objIdR)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offset64
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objIdR)
2053             }
2054             offset64 := Int64(offset.Value())
2055             ctx.OffsetOverluminTable = offset64
2056         }
2057     }
2058     return nil
2059 }
2060
2061 func LoadBinaryStream(ctx *Context, s *StreamReader, objIdR, genR int) error {
2062     var err error
2063     if
2064         // Load stream's content and store data into objIdRable entry
2065         err = LoadBinaryStreamFromStreamDataIntoObjIdRableEntry(
2066             ctx, s, objIdR, genR, ctx, objIdR, genR, ctx)
2067         // dereferencing failed: problem dereferencing stream ID"
2068         return errors.Wrapf(err, "dereferencing failed: problem dereferencing stream ID")
2069     }
2070     ctx.Read.BinarySize += s.GetSize()
2071     // Decode stream's content
2072     err = s.DecodeStreamContent(
2073         ctx, s, objIdR, genR, ctx.DecodeAllStreams)
2074     return err
2075 }
2076
2077 func UpdateLinearizationPanicbit(ctx *Context, o Object) {
2078     switch o == o.(Type) {
2079     case StreamDict:
2080         ctx.Read.BinarySize += s.GetSize()
2081     }
2082 }

```

```

290 }
291 // Create a mutable version entry (since it's in the catalog
292 // and record this as rootVersion (as opposed to headerVersion).
293 xhefTable.rootVersion = xhefTable.xHeaderFile.error {
294     log.Read.Println("IdentifyRootVersion: begin")
295     // Copy to get version from xHeaderFile.ParseRootVersion()
296     rootVersionStr = ""
297     if err == nil {
298         return err
299     }
300     if rootVersionStr == nil {
301         return nil
302     }
303     // Validate version and save corresponding constant to xHeFTable.
304     rootVersion, err := PDPVersion(rootVersionStr)
305     if err != nil {
306         return errors.Wrap(err, "IdentifyRootVersion: unknown PDP Root version: "+
307             rootVersionStr)
308     }
309     xhefTable.RootVersion = rootVersion
310     // Since it's the header version we can override by a Version entry in the
311     // manifest.
312     if xhefTable.HeaderVersion < v1 {
313         log.Info.Println("IdentifyRootVersion: PDP version is %s - will ignore root
314             version %s",
315             xhefTable.HeaderVersion, *rootVersionStr)
316     }
317     log.Read.Println("IdentifyRootVersion: end")
318     return nil
319 }
320 // Parse all Objects including stream content from file and save to the corresponding
321 // ManifestEntries.
322 func dereferenceObjectsOfObject(xContext, conf *Configuration) error {
323     log.Read.Println("dereferenceObjects: begin")
324     xhefTable = ctx.XhefTable
325     // Note for unencrypted files.
326     // Mandatory provide users to open & display file.
327     // Access may be restricted (Open access privileges).
328     // Optionally provide comments in order to gain unrestricted access.
329     if err := xhefTable.OpenFile() {

```

```

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

```

```

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

```

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

```

```

1573 // On return object's destructor may be called, problem dereferencing object
1574 stream.Md, no ref table entry, entry.ObjectStreamId)
1575
1576 //
1577 // Object of class entry has to be an ObjectStreamId
1578 //
1579 sd, o = ObjectStreamId(entry.ObjectId, ObjectStreamId)
1580
1581 if !ok {
1582     return errors.Errorf("decompressRefTableEntry: problem dereferencing object stream %d, no object stream", entry.ObjectStreamId)
1583 }
1584
1585 //
1586 // Get index object from ObjectStreamId
1587 //
1588 o, err = sd.IndexObjectFromEntry(ObjectStreamId)
1589 if err != nil {
1590     return errors.Errorf("decompressRefTableEntry: problem dereferencing object stream %d", entry.ObjectStreamId)
1591 }
1592
1593 // Save object to theRefTableEntry.
1594
1595 g := &
1596     entry.Object = o
1597     entry.Compression = 0
1598     entry.Expression = false
1599
1600 //
1601 // Load object's decompressRefTableEntry, end, Obj Id to be 'wcksc/v',
1602 // entry.ObjectStreamId, entry.ObjectStreamId, o)
1603
1604 return nil
1605
1606 //
1607 // Log interesting stream content.
1608 //
1609 func LogStreamContent(i int) {
1610     switch o := o.(type) {
1611     case StreamId:
1612         if o.Content == nil {
1613             log.Printf("logStream: no stream content")
1614         }
1615         if o.IsPageContent {
1616             //log.Printf("logStream: content %s", StreamId.Content)
1617         }
1618     case ObjectStreamId:
1619         if o.Content == nil {
1620             log.Printf("logStream: no object stream content")
1621         }
1622         if o.IsPageContent {
1623             log.Printf("logStream: object stream content %s", o.Content)
1624         }
1625         if o.IsPageEntry {
1626             log.Printf("logStream: no object stream obj arr")
1627         }
1628         if o.IsPageEntry {
1629             log.Printf("logStream: object stream obj arr %s", o.IsPageEntry)
1630         }
1631     }
1632 }
1633
1634 //
1635 // Default:

```

```

2020 // 2. Create a new object to hold the data
2021 case objRead: {
2022     // Read the data from the file
2023     case Read.BinaryToSize + w, Stream.Length
2024     case ReadStream:
2025         // Read the data from the file
2026         case Read.BinaryToSize + w, Stream.Length
2027     }
2028 }
2029 }
2030 }
2031 }
2032 }
2033 }
2034 }
2035 }
2036 }
2037 }
2038 }
2039 }
2040 }
2041 }
2042 }
2043 }
2044 }
2045 }
2046 }
2047 }
2048 }
2049 }
2050 }
2051 }
2052 }
2053 }
2054 }
2055 }
2056 }
2057 }
2058 }
2059 }
2060 }
2061 }
2062 }
2063 }
2064 }
2065 }
2066 }
2067 }
2068 }
2069 }
2070 }
2071 }
2072 }
2073 }
2074 }
2075 }
2076 }
2077 }
2078 }
2079 }
2080 }
2081 }
2082 }
2083 }
2084 }
2085 }
2086 }
2087 }
2088 }
2089 }
2090 }
2091 }
2092 }
2093 }
2094 }
2095 }
2096 }
2097 }
2098 }
2099 }
2100 }
2101 }
2102 }
2103 }
2104 }
2105 }
2106 }
2107 }
2108 }
2109 }
2110 }
2111 }
2112 }
2113 }
2114 }
2115 }
2116 }
2117 }
2118 }
2119 }
2120 }
2121 }
2122 }
2123 }
2124 }
2125 }
2126 }
2127 }
2128 }
2129 }
2130 }
2131 }
2132 }
2133 }
2134 }
2135 }
2136 }
2137 }
2138 }
2139 }
2140 }
2141 }
2142 }
2143 }
2144 }
2145 }
2146 }
2147 }
2148 }
2149 }
2150 }
2151 }
2152 }
2153 }
2154 }
2155 }
2156 }
2157 }
2158 }
2159 }
2160 }
2161 }
2162 }
2163 }
2164 }
2165 }
2166 }
2167 }
2168 }
2169 }
2170 }
2171 }
2172 }
2173 }
2174 }
2175 }
2176 }
2177 }
2178 }
2179 }
2180 }
2181 }
2182 }
2183 }
2184 }
2185 }
2186 }
2187 }
2188 }
2189 }
2190 }
2191 }
2192 }
2193 }
2194 }
2195 }
2196 }
2197 }
2198 }
2199 }
2200 }
2201 }
2202 }
2203 }
2204 }
2205 }
2206 }
2207 }
2208 }
2209 }
2210 }
2211 }
2212 }
2213 }
2214 }
2215 }
2216 }
2217 }
2218 }
2219 }
2220 }
2221 }
2222 }
2223 }
2224 }
2225 }
2226 }
2227 }
2228 }
2229 }
2230 }
2231 }
2232 }
2233 }
2234 }
2235 }
2236 }
2237 }
2238 }
2239 }
2240 }
2241 }
2242 }
2243 }
2244 }
2245 }
2246 }
2247 }
2248 }
2249 }
2250 }
2251 }
2252 }
2253 }
2254 }
2255 }
2256 }
2257 }
2258 }
2259 }
2260 }
2261 }
2262 }
2263 }
2264 }
2265 }
2266 }
2267 }
2268 }
2269 }
2270 }
2271 }
2272 }
2273 }
2274 }
2275 }
2276 }
2277 }
2278 }
2279 }
2280 }
2281 }
2282 }
2283 }
2284 }
2285 }
2286 }
2287 }
2288 }
2289 }
2290 }
2291 }
2292 }
2293 }
2294 }
2295 }
2296 }
2297 }
2298 }
2299 }
2300 }
2301 }
2302 }
2303 }
2304 }
2305 }
2306 }
2307 }
2308 }
2309 }
2310 }
2311 }
2312 }
2313 }
2314 }
2315 }
2316 }
2317 }
2318 }
2319 }
2320 }
2321 }
2322 }
2323 }
2324 }
2325 }
2326 }
2327 }
2328 }
2329 }
2330 }
2331 }
2332 }
2333 }
2334 }
2335 }
2336 }
2337 }
2338 }
2339 }
2340 }
2341 }
2342 }
2343 }
2344 }
2345 }
2346 }
2347 }
2348 }
2349 }
2350 }
2351 }
2352 }
2353 }
2354 }
2355 }
2356 }
2357 }
2358 }
2359 }
2360 }
2361 }
2362 }
2363 }
2364 }
2365 }
2366 }
2367 }
2368 }
2369 }
2370 }
2371 }
2372 }
2373 }
2374 }
2375 }
2376 }
2377 }
2378 }
2379 }
2380 }
2381 }
2382 }
2383 }
2384 }
2385 }
2386 }
2387 }
2388 }
2389 }
2390 }
2391 }
2392 }
2393 }
2394 }
2395 }
2396 }
2397 }
2398 }
2399 }
2400 }
2401 }
2402 }
2403 }
2404 }
2405 }
2406 }
2407 }
2408 }
2409 }
2410 }
2411 }
2412 }
2413 }
2414 }
2415 }
2416 }
2417 }
2418 }
2419 }
2420 }
2421 }
2422 }
2423 }
2424 }
2425 }
2426 }
2427 }
2428 }
2429 }
2430 }
2431 }
2432 }
2433 }
2434 }
2435 }
2436 }
2437 }
2438 }
2439 }
2440 }
2441 }
2442 }
2443 }
2444 }
2445 }
2446 }
2447 }
2448 }
2449 }
2450 }
2451 }
2452 }
2453 }
2454 }
2455 }
2456 }
2457 }
2458 }
2459 }
2460 }
2461 }
2462 }
2463 }
2464 }
2465 }
2466 }
2467 }
2468 }
2469 }
2470 }
2471 }
2472 }
2473 }
2474 }
2475 }
2476 }
2477 }
2478 }
2479 }
2480 }
2481 }
2482 }
2483 }
2484 }
2485 }
2486 }
2487 }
2488 }
2489 }
2490 }
2491 }
2492 }
2493 }
2494 }
2495 }
2496 }
2497 }
2498 }
2499 }
2500 }
2501 }
2502 }
2503 }
2504 }
2505 }
2506 }
2507 }
2508 }
2509 }
2510 }
2511 }
2512 }
2513 }
2514 }
2515 }
2516 }
2517 }
2518 }
2519 }
2520 }
2521 }
2522 }
2523 }
2524 }
2525 }
2526 }
2527 }
2528 }
2529 }
2530 }
2531 }
2532 }
2533 }
2534 }
2535 }
2536 }
2537 }
2538 }
2539 }
2540 }
2541 }
2542 }
2543 }
2544 }
2545 }
2546 }
2547 }
2548 }
2549 }
2550 }
2551 }
2552 }
2553 }
2554 }
2555 }
2556 }
2557 }
2558 }
2559 }
2560 }
2561 }
2562 }
2563 }
2564 }
2565 }
2566 }
2567 }
2568 }
2569 }
2570 }
2571 }
2572 }
2573 }
2574 }
2575 }
2576 }
2577 }
2578 }
2579 }
2580 }
2581 }
2582 }
2583 }
2584 }
2585 }
2586 }
2587 }
2588 }
2589 }
2590 }
2591 }
2592 }
2593 }
2594 }
2595 }
2596 }
2597 }
2598 }
2599 }
2600 }
2601 }
2602 }
2603 }
2604 }
2605 }
2606 }
2607 }
2608 }
2609 }
2610 }
2611 }
2612 }
2613 }
2614 }
2615 }
2616 }
2617 }
2618 }
2619 }
2620 }
2621 }
2622 }
2623 }
2624 }
2625 }
2626 }
2627 }
2628 }
2629 }
2630 }
2631 }
2632 }
2633 }
2634 }
2635 }
2636 }
2637 }
2638 }
2639 }
2640 }
2641 }
2642 }
2643 }
2644 }
2645 }
2646 }
2647 }
2648 }
2649 }
2650 }
2651 }
2652 }
2653 }
2654 }
2655 }
2656 }
2657 }
2658 }
2659 }
2660 }
2661 }
2662 }
2663 }
2664 }
2665 }
2666 }
2667 }
2668 }
2669 }
2670 }
2671 }
2672 }
2673 }
2674 }
2675 }
2676 }
2677 }
2678 }
2679 }
2680 }
2681 }
2682 }
2683 }
2684 }
2685 }
2686 }
2687 }
2688 }
2689 }
2690 }
2691 }
2692 }
```

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObject(object)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shellEntry entry assign a object either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObject(object)
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     if ver = differenceObject(object)
2139     if err != nil {
2140         return err
2141     }
2142
2143     log.Printf("DifferenceCatalogTable: end")
2144
2145     return nil
2146 }
2147
2148 func handleEncryptedFile(cts *Context) error {
2149     cts.Cmd = DECRYPT
2150     if cts.Cmd == SETPERMISSIONS {
2151         return errors.New("pfcpu: this file is not encrypted")
2152     }
2153
2154     if cts.Cmd == DECRYPT {
2155         return nil
2156     }
2157
2158     // Encrypt subcommand found.
2159     if cts.SubCmd == "i" {
2160         return errors.New("pfcpu: please provide owner password and optional user
2161 password")
2162     }
2163     return nil
2164 }
2165
2166 func idBytes(cts *Context) []byte, err error {
2167     if cts.ID == nil {
2168         return nil, errors.New("pfcpu: missing ID entry")
2169     }
2170
2171     N1, ok = cts.ID[0].(float64)
2172     if ok {
2173         id, err = N1.Bytes()
2174         if err != nil {
2175             return nil, err
2176         }
2177     }
2178 }

```

```

1452 // @param {Object} ctx - Context object
1453 if (found) {
1454   decodeParamArr, found = dict.Fall(UNCODEPARAM)
1455 }
1456 if (found) {
1457   decodeParamArr, ok = decodeParam.Array()
1458   if (ok) {
1459     return nil, errors.New("pdcip: pdfFilterPipeline: expected decodeParam
1460 array corrupt")
1461   }
1462 }
1463 // /var Printout("decodeParam: %v", decodeParam)
1464 // /var Printout("decodeParam: %v", decodeParam)
1465 // /var Printout("decodeParam: %v", decodeParam)
1466 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArr,
1467 decodeParam)
1468 if (err != nil) {
1469   log.Read.Println("pdfFilterPipeline: err")
1470 }
1471 return filterPipeline, err
1472 }
1473 func streamDictForObj(c *Context, d Dict, objKey, streamIn int, streamFset
1474 *FileSet, objId int) (StreamDict, error) {
1475   streamLength, streamLengthF = d.Length()
1476   if (streamLength == 0) {
1477     return sd, errors.New("pdcip: streamDictForObj: stream object without
1478 streamFset")
1479   }
1480   filterPipeline, err = pdfFilterPipeline(c, d)
1481   if (err == nil) {
1482     return sd, err
1483   }
1484   streamOffset = offset
1485   // We have a stream object
1486   sd = NewStream(streamDict, streamOffset, streamLength, streamLengthF, filterPipeline
1487   log.Read.Println("streamDictForObj: err, streamFset %v", objId)
1488   return sd, nil
1489 }
1490 func dictCtx(c *Context, d Dict, objKey, err, errIn, streamIn int) (Dict, err
1491 if (err != nil) {
1492   if (ctx.EncKey == nil) {
1493     ctx.EncKey = decryptParamDict(d, objKey, err, ctx.EncKey, ctx.AES4Strings,
1494     "A.S")
1495     if (err == nil) {
1496       return nil, err
1497     }
1498   }
1499   if (errIn == 0 || (errIn < 0 && (errIn < 0 || streamIn < 0))) {
1500     log.Read.Println("dict: err, objId, objKey")
1501     err = errIn
1502   }
1503 }
1504 }

```

```

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

```

```

130 // @see https://github.com/ericniebler/psutil/blob/master/psutil/_psutil_linux.c
131         log.Read.PrintIn("logStream: no objectsReady to copy")
132     }
133 }
134
135 // Decode all object streams to contained objects are ready to be used.
136 void decodeObjectStreams(cts::Context) error {
137     // @see
138     // @entry "externs" 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")
156         }
157     }
158     log.Read.Print("decodeObjectStreams: parsing object stream for objectNumber")
159 }
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 = p.(StreamObject)
169 if !ok {
170     return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
171 }
172
173 // Load decoded stream content to stableTable.
174 if err = loadDecodedStreamContent(cts, sd); err != nil {
175     return errors.Wrap(err, "decodeObjectStreams: problem dereferencing object stream")
176 }
177
178 // Save decoded stream content to stableTable.
179 if err = saveDecodedStreamContent(cts, sd, objectNumber, entry.Generation, true); err != nil {
180     return errors.Wrap(err, "decodeObjectStreams: problem saving object stream")
181 }
182 }

```

```

2342 // err = err + ParseObject(err, 'entry.offset, objNr, entry.generation)
2343 if err == nil {
2344     return errors.Wrap(err, "dereferencedObject: problem dereferencing object id")
2345 }
2346 }
2347
2348 entry.Object = o
2349
2350 // // Linearization objects are validated and removed for stats only.
2351 err = handleLinearizationAndPurge(ctxt, o, objNr)
2352 if err == nil {
2353     return err
2354 }
2355 // // handle stream dict.
2356 if _, ok = o.(StreamDict); ok {
2357     // // Handle stream dict.
2358     err = errors.Errorf("dereferencedObject: object stream should already be
2359     dereferenced at objId=%v", objNr)
2360     if err != nil {
2361         return err
2362     }
2363     if _, ok = o.(StreamDict); ok {
2364         // // Handle stream dict.
2365         return errors.Errorf("dereferencedObject: xref stream should already be
2366         dereferenced at objId=%v", objNr)
2367     }
2368     if sd, ok = o.(StreamDict); ok {
2369         // // Handle stream dict.
2370         err = loadStream(ctxt, sd, objNr, entry.generation)
2371         if err == nil {
2372             return err
2373         }
2374     }
2375     entry.Object = sd
2376 }
2377
2378 // // Log and Print ("dereferencedObject: and objId of %v\n", objNr,
2379 // // objNrDict, entry.Object)
2380 log.Printf("dereferencedObject: and objId of %v\n", objNr,
2381 objNrDict, entry.Object)
2382 logStream(entry.Object)
2383 return nil
2384 }
2385
2386 func processBidiCounts(defTable *XRefTable, D Dict) {
2387     for _, n := range o {
2388         match ok := n.Type() == xrefType
2389         case IndexDict:
2390             entry, ok := defTable.LookupTableEntry(n)
2391             if ok {
2392                 entry.Count++
2393             }
2394         case Dict:
2395             processBidiCounts(defTable, o)
2396         case Array:
2397             processBidiCounts(defTable, o)
2398     }
2399 }
2400
2401 }
2402 }
2403 }

```

```

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

```

[illegible]

```

3570 // Read stream content into the window stream content buffer size
3571 streamContent;
3572 // Load content to readContent context: Context, sd streamContent() [byte, error]
3573 load.ReadContent(readContentContext: Context, sd streamContent() [byte, error])
3574
3575 // Log ReadContent()
3576 log.Read.Print("LoadContentContext: begin(w/v/u/s)",
3577
3578 // Return
3579 // Return saved decoded content.
3580 if sd.Raw == nil {
3581     log.Read.Print("LoadContentContext: end, already in memory.")
3582     return sd.Raw, nil
3583 }
3584
3585 // Read stream content encoded at stream with stream length.
3586 // Difference stream length if stream length is an indirect object.
3587 if sd.StreamLength == nil {
3588     if sd.StreamLength == nil {
3589         return nil, errors.New("pdfcpu: LoadContentContext: missing
3590 streamLength")
3591 }
3592 // sd stream length from indirect object
3593 sd.StreamLength, err = IndirectObject(sd, sd.StreamLengthObj)
3594 if err == nil {
3595     return nil, err
3596 }
3597
3598 // Log Read.Print("LoadContentContext: end indirect streamLength",
3599 sd.StreamLength)
3600
3601 // Read
3602 // Read offset in sd streamContent
3603 rd, err = NewPositionalReaderAt(sd.Raw, 0)
3604 if err != nil {
3605     return nil, err
3606 }
3607
3608 // Log Read.Print("LoadContentContext: seeked to offset:", rd, readContent)
3609
3610 // Buffer stream content.
3611 // Read content from stream
3612 readContent, err = readContentStream(rd, int(sd.StreamLength))
3613 if err == nil {
3614     return nil, err
3615 }
3616
3617 // Log Read.Print("LoadContentContext: buffered:", len(readContent), len(readContent),
3618 // len(readContent), len(readContent))
3619
3620 // Stream content
3621 // Set stream content
3622 sd.Raw = readContent
3623
3624 // Log Read.Print("LoadContentContext: end: len(streamContent)",
3625 // len(sd.Raw))
3626
3627 // Return decoded content
3628 return readContent, nil
3629
3630 // Decode the raw encoded stream content and saves it to streamContent.Context.

```

```

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 SvId",
1980 // objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict objectStream
1984 // If err == nil
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding
1986 object stream svd", objectStream)
1987
1988 // If svd.ObjectArray == nil
1989     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1990
1991 // Log Read, Print("decodeObjectStream: decoded object stream SvId",
1992 // objectStream)
1993
1994 // Save object stream dict to sHeaderEntry.
1995     entry.Object = svd
1996
1997 // Log Read, Print("decodeObjectStream: end")
1998     return nil
1999
2000 func handleLinearizationPanicDict(cxt *Context, obj Object, objStr int) error {
2001     // Log Read, linearized
2002     // Commentation dict already processed.
2003     return nil
2004 }
2005
2006 // handle Linearization panic dict
2007 // If d == c == obj (dict) obj == d, it is linearizationPanicDict()
2008 // Log Read, linearized == true
2009 // c.linearizationPanicDict() == true
2010 // Log Read, Print("handleLinearizationPanicDict: identified linearizationObj
2011 // SvId", c)
2012     a := d.ArrayEntry("sv")
2013

```

```

2020:
2021: func processArrayByCounts(x:Iterable, xObjTable: aXObjTable, a Array) {
2022:   for _ in range a {
2023:     switch o in a.cType() {
2024:     case IndirectType:
2025:       entry, ok = xObjTable.findTableEntryIndirect(o)
2026:       if ok {
2027:         entry.RefCount++
2028:       }
2029:     case DirectType:
2030:       processRefCounts(xObjTable, o)
2031:     case Array:
2032:       processRefCounts(xObjTable, o)
2033:     }
2034:   }
2035: }
2036:
2037: func processRefCounts(xObjTable: aXObjTable, o Object) {
2038:   switch o in o.cType() {
2039:   case DirectType:
2040:     processIndirectCounts(xObjTable, o)
2041:   case StreamIndirect:
2042:     processIndirectCounts(xObjTable, o.Direct)
2043:   case Array:
2044:     processArrayByCounts(xObjTable, o)
2045:   }
2046: }
2047:
2048: // Performance notes: this function iterates over all objects from object stream.
2049: func dereferenceIndirectCounts(c: aContext) error {
2050:   log.Reb.Println("dereferenceIndirectCounts: begin")
2051:   xObjTable = ctx.XObjTable
2052:   // Use sorted list of object numbers.
2053:   // This step sorting for performance gain.
2054:   keys List
2055:   for k in xObjTable.Table {
2056:     keys = append(keys, k)
2057:   }
2058:   sort.Ints(keys)
2059:   for _ , objNr = range keys {
2060:     err = dereferenceIndirectCounts(x, objNr)
2061:     if err != nil {
2062:       return err
2063:     }
2064:   }
2065:   for _ , objNr = range keys {
2066:     entry = xObjTable.Table[objNr]
2067:     if entry.ref != entry.compressed {
2068:       continue
2069:     }
2070:     processRefCounts(xObjTable, entry.Obj)
2071:   }

```

```

2530 // Open the password file
2531 if err := nil {
2532     return err
2533 }
2534 // If the owner password does not match we generally move on if the user password
2535 // errors
2536 // Unless we need to limit on a user's correct password due to the specific
2537 // amount in password
2538 if !ok {
2539     return handlePasswordMismatch(ctxt.Cnd) {
2540         return errors.New("password: please provide the master password with 'opw'")
2541     }
2542 }
2543 // Generally the user password, which is also regarded as the master password or
2544 // pre-password
2545 // is sufficient for moving on. A password change is an exception since it
2546 // needs the master password
2547 if ok {
2548     return handlePasswordMismatch(ctxt.Cnd) {
2549         return errors.New("password: please provide the master password with 'opw'")
2550     }
2551 }
2552 ok, err := validatePermissions(ctxt)
2553 if err == nil {
2554     return err
2555 }
2556 // If ok {
2557     return errors.New("password: corrupted permissions after opw ok")
2558 }
2559 return nil
2560 }
2561 // Validate the user password ok, document open password.
2562 ok, err := validatePermissions(ctxt)
2563 if err == nil {
2564     return err
2565 }
2566 // If ok {
2567     return errors.New("password: please provide the correct password")
2568 }
2569 //fmt.Printf("opw ok: %d\n", ok)
2570 return handlePermissions(ctxt)
2571 }
2572 func checkForEncryption(c *Context) error {
2573     if c.Is.Encrypt
2574     {
2575         if err := nil {
2576             // This file is not encrypted.
2577             return handleEncryptionFailed(c)
2578         }
2579         // This file is encrypted.
2580         log.Read.Printf("Encryption: %v\n", Ir)
2581         if c.Is.Encrypt == ENCRYPT {
2582             // We want to encrypt this file.
2583             return errors.New("password: This file is already encrypted")
2584         }
2585         // Difference encrypted.
2586     }
2587 }

```