

[illegible]

```

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.parasofttableEntry: corrupt ref subtraction
2311             entry)
2312         }
2313         var xrefTableEntry *XrefTableEntry
2314         if entryType == "in" {
2315             // in use object
2316             log.Debug.Printf("parasofttableEntry: Object %md is in use at offset%md,
2317             generation%md", objNumber, offset, generation)
2318             if offset == 0 {
2319                 log.Infof.Print("parasofttableEntry: Skip entry for in use object %md
2320                 with offset 0", objNumber)
2321                 return nil
2322             }
2323             xrefTableEntry =
2324                 &XrefTableEntry{
2325                     objNumber: objNumber,
2326                     offset: offset,
2327                     Generation: &generation}
2328         } else {
2329             // free object
2330             log.Debug.Printf("parasofttableEntry: Object %md is unused, next free is
2331             objects.parasofttableEntry: objNumber, offset, generation)
2332             xrefTableEntry =
2333                 &XrefTableEntry{
2334                     objNumber: objNumber,
2335                     free: true,
2336                     offset: offset,
2337                     Generation: &generation}
2338         }
2339     }
2340     log.Debug.Printf("parasofttableEntry: Insert new xreftable entry for Object %md",
2341     objNumber)
2342     xrefTable.Table[objNumber] = xrefTableEntry
2343     return nil
2344 }
2345 func (m *Parser) readParasoftTableEntry(cmd *Command) error {
2346     return m.readParasoftTableEntry(cmd)
2347 }

```

```

443 // https://github.com/GoogleCloudPlatform/google-cloud-go/blob/master/storage/googlecloudstorage.go#L104
444 for i := 0; i < len(objects); i++ {
445     objectNumber := xsd_objects[i]
446
447     // Read object
448     iStart := i + 1
449     c2 := bufToInt64(xs[iStart:iStart+12])
450     c3 := bufToInt64(xs[iStart+12:iStart+24+12])
451
452     var xbfFileEntry *XbfFileEntry
453
454     switch object[i] {
455     case 0x00:
456         // Read object
457         log.Read.Println("object,xbfFileEntryFromRssFromStream: Object #", i
458             , numset, firstGen, "objectid, generationNum", objectNumber, c2, c3)
459         s = int(c2)
460         xbfFileEntry =
461             &XbfFileEntry{
462                 Free: true,
463                 Compressed: false,
464                 Offset: 8c2,
465                 Generation: 0d}
466     case 0x01:
467         // In use object
468         log.Read.Println("object,xbfFileEntryFromRssFromStream: Object #", i in
469             , use, s, "offset", generationNum, objectNumber, c2, c3)
470         s = int(c2)
471         xbfFileEntry =
472             &XbfFileEntry{
473                 Free: false,
474                 Compressed: false,
475                 Offset: 8c2,
476                 Generation: 0d}
477     case 0x02:
478         // compressed object
479         log.Read.Println("object,xbfFileEntryFromRssFromStream: Object #", i in
480             , compressed, s, "offset", generationNum, c2, c3)
481         objNumber = int(c2)
482         objName = int(c3)
483         xbfFileEntry =
484             &XbfFileEntry{
485                 Free: false,
486                 Compressed: true,
487                 ObjectStream: objNumberIndex,
488                 ObjectStreamIndex: objNameIndex}
489     case 0x03:
490         // Read object stream
491         log.Read.Println("object,xbfFileEntryFromRssFromStream: skip entry #", i
492             , skip)
493     }
494 }
495
496 if c1x.XbfFile.Exists(objectNumber) {
497     log.Read.Println("object,xbfFileFromRssFromStream: skip entry #", i

```

```

672         if desttable == null {
673             // return error: how? (popup: parameterInfoRef: missing entry "Root"? )
674             return errors.New(popup: parameterInfoRef: missing entry "Root"? )
675         }
676         xhefTable.Root = rootObjRef
677         log.Debug.Printf("parameterInfoRef: Root object: %s\n", *xhefTable.Root)
678     }
679     if xhefTable.Info == nil {
680         // infoObjRef = & indirectEntry("Info")
681         infoObjRef = & indirectEntry("Info")
682         if infoObjRef == nil {
683             // xhefTable.Info = infoObjRef
684             xhefTable.Info = infoObjRef
685         }
686         log.Debug.Printf("parameterInfoRef: Info object: %s\n", *xhefTable.Info)
687     }
688     if xhefTable.ID == nil {
689         idObjRef = & indirectEntry("ID")
690         if idObjRef == nil {
691             // xhefTable.ID = idObjRef
692             log.Debug.Printf("parameterInfoRef: ID object: %s\n", *xhefTable.ID)
693         } else if xhefTable.CryptKey == nil {
694             // return errors.New("parameterInfoRef: missing entry "ID"?")
695             return errors.New("parameterInfoRef: missing entry "ID"?")
696         }
697         log.Debug.Printf("parameterInfoRef: end")
698     }
699     return nil
700 }
701
702 func parameterInfoRef(trailerDict Dict, ctx *Context) (xint64, error) {
703     //
704     log.Debug.Printf("parameterInfoRef: begin")
705     xhefTable = ctx.XhefTable
706     err = parameterInfoRef(trailerDict, xhefTable)
707     if err == nil {
708         return nil, err
709     }
710     //
711     if err == trailerDict.ArrayEntry("AdditionalStreams"); err == nil {
712         log.Debug.Printf("parameterInfoRef: found AdditionalStreams: %s\n", err)
713         a := make([]string, 0)
714         for i, value := range arr {
715             if infoRef, ok := value (indirectEntry); ok {
716                 a = append(a, infoRef)
717             }
718         }
719         xhefTable.AdditionalStreams = &a
720     }
721     offset = trailerDict.Previous()
722     if offset == nil {
723         log.Debug.Printf("parameterInfoRef: previous xref table section offset: %s\n",
724             offset)
725     }
726     offsetObjRefStream = trailerDict.IndirectEntry("objRefStream")
727 }

```

```

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 == "trailer" {
9919         return nil, err
9920     }
9921
9922     // if only line type next line for trailer
9923     if !line == "trailer" {
9924         if line, err := scanLine(s); err == nil {
9925             return nil, err
9926         }
9927     }
9928 }

```

```

1337 // to cxx.NewReader()
1338
1339 br, rdCount, err = reader.SeekStream(rs)
1340 if err != nil {
1341     return err
1342 }
1343
1344 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 current valid seek section.
1381         return hyposisSection(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 // Note that the first seek table entry is a defined previous seek section
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 = dereferenceObjects(ctx, conf); err != nil {
574     return nil, err
575 }
576
577 // Some references write an invariant size into trailer.
578 // i.e. ctx.ShuffleSize < len(ctx.ShuffleTable)
579 if ctx.ShuffleSize < len(ctx.ShuffleTable) {
580     ctx.ShuffleSize = len(ctx.ShuffleTable)
581 }
582
583 log.Read.Print("Read: end")
584
585 return ctx, nil
586 }
587
588 // ScanLine is a multi-function for a Scanner that returns each line of
589 // text, stripped of any trailing end-of-line marker. The returned line may
590 // be empty, may contain carriage returns, or may contain carriage returns followed
591 // by one newline or one carriage return or one newline.
592 // If the returned line is empty, it will be returned even if it was an error.
593 func scanLine(data []byte, startIdx int) (string, bool) {
594     if startIdx > len(data) - 1 {
595         return "", true
596     }
597     if startIdx > len(data) - 1 {
598         return "", true
599     }
600     if startIdx > len(data) - 1 {
601         return "", true
602     }
603     if startIdx > len(data) - 1 {
604         return "", true
605     }
606     if startIdx > len(data) - 1 {
607         return "", true
608     }
609     if startIdx > len(data) - 1 {
610         return "", true
611     }
612     if startIdx > len(data) - 1 {
613         return "", true
614     }
615     if startIdx > len(data) - 1 {
616         return "", true
617     }
618     if startIdx > len(data) - 1 {
619         return "", true
620     }
621     if startIdx > len(data) - 1 {
622         return "", true
623     }
624     if startIdx > len(data) - 1 {
625         return "", true
626     }
627     if startIdx > len(data) - 1 {
628         return "", true
629     }
630     if startIdx > len(data) - 1 {
631         return "", true
632     }
633     if startIdx > len(data) - 1 {
634         return "", true
635     }
636     if startIdx > len(data) - 1 {
637         return "", true
638     }
639     if startIdx > len(data) - 1 {
640         return "", true
641     }
642     if startIdx > len(data) - 1 {
643         return "", true
644     }
645     if startIdx > len(data) - 1 {
646         return "", true
647     }
648     if startIdx > len(data) - 1 {
649         return "", true
650     }
651     if startIdx > len(data) - 1 {
652         return "", true
653     }
654     if startIdx > len(data) - 1 {
655         return "", true
656     }
657     if startIdx > len(data) - 1 {
658         return "", true
659     }
660     if startIdx > len(data) - 1 {
661         return "", true
662     }
663     if startIdx > len(data) - 1 {
664         return "", true
665     }
666     if startIdx > len(data) - 1 {
667         return "", true
668     }
669     if startIdx > len(data) - 1 {
670         return "", true
671     }
672     if startIdx > len(data) - 1 {
673         return "", true
674     }
675     if startIdx > len(data) - 1 {
676         return "", true
677     }
678     if startIdx > len(data) - 1 {
679         return "", true
680     }
681     if startIdx > len(data) - 1 {
682         return "", true
683     }
684     if startIdx > len(data) - 1 {
685         return "", true
686     }
687     if startIdx > len(data) - 1 {
688         return "", true
689     }
690     if startIdx > len(data) - 1 {
691         return "", true
692     }
693     if startIdx > len(data) - 1 {
694         return "", true
695     }
696     if startIdx > len(data) - 1 {
697         return "", true
698     }
699     if startIdx > len(data) - 1 {
700         return "", true
701     }
702     if startIdx > len(data) - 1 {
703         return "", true
704     }
705     if startIdx > len(data) - 1 {
706         return "", true
707     }
708     if startIdx > len(data) - 1 {
709         return "", true
710     }
711     if startIdx > len(data) - 1 {
712         return "", true
713     }
714     if startIdx > len(data) - 1 {
715         return "", true
716     }
717     if startIdx > len(data) - 1 {
718         return "", true
719     }
720     if startIdx > len(data) - 1 {
721         return "", true
722     }
723     if startIdx > len(data) - 1 {
724         return "", true
725     }
726     if startIdx > len(data) - 1 {
727         return "", true
728     }
729     if startIdx > len(data) - 1 {
730         return "", true
731     }
732     if startIdx > len(data) - 1 {
733         return "", true
734     }
735     if startIdx > len(data) - 1 {
736         return "", true
737     }
738     if startIdx > len(data) - 1 {
739         return "", true
740     }
741     if startIdx > len(data) - 1 {
742         return "", true
743     }
744     if startIdx > len(data) - 1 {
745         return "", true
746     }
747     if startIdx > len(data) - 1 {
748         return "", true
749     }
750     if startIdx > len(data) - 1 {
751         return "", true
752     }
753     if startIdx > len(data) - 1 {
754         return "", true
755     }
756     if startIdx > len(data) - 1 {
757         return "", true
758     }
759     if startIdx > len(data) - 1 {
760         return "", true
761     }
762     if startIdx > len(data) - 1 {
763         return "", true
764     }
765     if startIdx > len(data) - 1 {
766         return "", true
767     }
768     if startIdx > len(data) - 1 {
769         return "", true
770     }
771     if startIdx > len(data) - 1 {
772         return "", true
773     }
774     if startIdx > len(data) - 1 {
775         return "", true
776     }
777     if startIdx > len(data) - 1 {
778         return "", true
779     }
780     if startIdx > len(data) - 1 {
781         return "", true
782     }
783     if startIdx > len(data) - 1 {
784         return "", true
785     }
786     if startIdx > len(data) - 1 {
787         return "", true
788     }
789     if startIdx > len(data) - 1 {
790         return "", true
791     }
792     if startIdx > len(data) - 1 {
793         return "", true
794     }
795     if startIdx > len(data) - 1 {
796         return "", true
797     }
798     if startIdx > len(data) - 1 {
799         return "", true
800     }
801     if startIdx > len(data) - 1 {
802         return "", true
803     }
804     if startIdx > len(data) - 1 {
805         return "", true
806     }
807     if startIdx > len(data) - 1 {
808         return "", true
809     }
810     if startIdx > len(data) - 1 {
811         return "", true
812     }
813     if startIdx > len(data) - 1 {
814         return "", true
815     }
816     if startIdx > len(data) - 1 {
817         return "", true
818     }
819     if startIdx > len(data) - 1 {
820         return "", true
821     }
822     if startIdx > len(data) - 1 {
823         return "", true
824     }
825     if startIdx > len(data) - 1 {
826         return "", true
827     }
828     if startIdx > len(data) - 1 {
829         return "", true
830     }
831     if startIdx > len(data) - 1 {
832         return "", true
833     }
834     if startIdx > len(data) - 1 {
835         return "", true
836     }
837     if startIdx > len(data) - 1 {
838         return "", true
839     }
840     if startIdx > len(data) - 1 {
841         return "", true
842     }
843     if startIdx > len(data) - 1 {
844         return "", true
845     }
846     if startIdx > len(data) - 1 {
847         return "", true
848     }
849     if startIdx > len(data) - 1 {
850         return "", true
851     }
852     if startIdx > len(data) - 1 {
853         return "", true
854     }
855     if startIdx > len(data) - 1 {
856         return "", true
857     }
858     if startIdx > len(data) - 1 {
859         return "", true
860     }
861     if startIdx > len(data) - 1 {
862         return "", true
863     }
864     if startIdx > len(data) - 1 {
865         return "", true
866     }
867     if startIdx > len(data) - 1 {
868         return "", true
869     }
870     if startIdx > len(data) - 1 {
871         return "", true
872     }
873     if startIdx > len(data) - 1 {
874         return "", true
875     }
876     if startIdx > len(data) - 1 {
877         return "", true
878     }
879     if startIdx > len(data) - 1 {
880         return "", true
881     }
882     if startIdx > len(data) - 1 {
883         return "", true
884     }
885     if startIdx > len(data) - 1 {
886         return "", true
887     }
888     if startIdx > len(data) - 1 {
889         return "", true
890     }
891     if startIdx > len(data) - 1 {
892         return "", true
893     }
894     if startIdx > len(data) - 1 {
895         return "", true
896     }
897     if startIdx > len(data) - 1 {
898         return "", true
899     }
900     if startIdx > len(data) - 1 {
901         return "", true
902     }
903     if startIdx > len(data) - 1 {
904         return "", true
905     }
906     if startIdx > len(data) - 1 {
907         return "", true
908     }
909     if startIdx > len(data) - 1 {
910         return "", true
911     }
912     if startIdx > len(data) - 1 {
913         return "", true
914     }
915     if startIdx > len(data) - 1 {
916         return "", true
917     }
918     if startIdx > len(data) - 1 {
919         return "", true
920     }
921     if startIdx > len(data) - 1 {
922         return "", true
923     }
924     if startIdx > len(data) - 1 {
925         return "", true
926     }
927     if startIdx > len(data) - 1 {
928         return "", true
929     }
930     if startIdx > len(data)
```

```

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 New")
303 }

```

```

347 // Create a new object table entry
348 | {
349 |     ctta.Table(objectNumber) = <objectTableEntry
350 | }
351 | }
352 |
353 | log, Read, PrintIn("extracted objectTableEntriesFromNewStream: end")
354 |
355 | return nil
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.readHex16be(whofFile.Version()) > 0 || !ctx.readHex4(
214     return nil, errors.Errorf("parseFailure: HDL's constant reader
215 found incompatible version %s, %sFile.versionString()",
216     ctx.readHex4(whofFile.Version())
217 }
218 log.Debug.Println("parseFailure: end")
219 return offset, nil
220 }
221 }
222 // This file is using cross reference streams.
223 }
224 if !ctx.readHex4(
225     ctx.readHex4(whofFile.True
226     ctx.readHex4(whofFile.True == true
227 }
228 // I/O constant readers process hidden objects contained
229 // in whofFile before continuing to process any previous WhofSection.
230 // Previous WhofSection is expected to have free entries for hidden entries.
231 // No matter in whofSection only.
232 if !ctx.readHex4(
233     err = parseHiddenHexStream(offsetHexStream, ctx); err == nil {
234         return nil, errors
235     }
236 }
237 log.Debug.Println("parseFailure: end")
238 return offset, nil
239 }
240 // Return constant readers to process any previous WhofSection.
241 }
242 func scanIndexNew(s whof.Scanner) (string, error) {
243     if !ctx.readHex16be(
244         if s.Err() == nil {
245             return "", s.Err()
246         }
247         return "", errors.New("pdfpc: scanIndexNew: returning nothing")
248     }
249     return s.Text(), nil
250 }
251 }
252 func scanIndex(s whof.Scanner) (string, error) {
253     for i := 0; i <= i; i++ {
254         if err := s.ScanIndex(
255             if err == nil {
256                 return "", err
257             }
258             if len(s) > 0 {
259                 break
260             }
261         }
262         i := strings.Index(s, "\n")
263         if i > 0 {
264             s = s[:i]
265         }
266     }
267 }

```

[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 file objects for 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     // n := max(buf.Len(), size)
1223
1224     // err = rd.Read(buf[:n])
1225
1226     if err != nil {
1227         return nil, err
1228     }
1229
1230     //Log.Read.Print("growBuf: Read %d bytes", n)
1231
1232     return append(buf, ...), nil
1233 }
1234
1235 func maxStreamOffset(line string, stream int) (off int) {
1236
1237     off = stream + len("stream")
1238
1239     // Skip optional blanks.
1240     // TODO Should be skip optional whitespace instead?
1241     for i, lineOff := 0; i < len(line); i++ {
1242         if SkipThe line[i] {
1243             continue
1244         }
1245         if lineOff == '\n' {
1246             return
1247         }
1248     }
1249 }
1250
1251 // Skip 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 (rs ReaderSeeker) NewReader(offset int64) (*bufio.Reader, error) {
333     if rs == rs.Seek(offset, 0).SeekStart() {
334         return nil, err
335     }
336     log.Read.Print("bufio.NewReader: position 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 startchar
341 // (offset)
342 func (rs ReaderSeeker) LastSection(ctx *Context) (int64, error) {
343     rs := ctx.Read.Rs
344     var {
345         startchar, worked, bufSize
346         bufSize int64 = 512
347         offset
348     }
349     for i := 1; offset < 0; i++ {
350         if err := rs.Seek(-index(i)+bufSize, 0).SeekEnd()
351             || err == nil
352             || return nil, errors.New("pdfcpu can't find last r/w section")
353         }
354     log.Read.Print("Scanning for offsetLastR/WSection starting at %d\n", offset)
355     curBuf := make([]byte, bufSize)
356 }

```

```

340 // @ts-ignore
341 // If we call obj instanceof an object stream we have fun, but also instanceofStream()
342 func parObj instanceof ObjectStream? error {
343     logDecompressPrint("parObj instanceofStream: decoding had objects: %s", objDec.Count)
344     decodedContent = decodedContent
345     parObj = decodedContent?.objDec.FirstObjStream()
346 }
347 objDec = strings.Fields(string(objDec))
348 if len(objDec) % 2 != 0 {
349     return errors.New("pdcfg: parObjDec ObjectStream: corrupt object stream dict")
350 }
351 // e.g., 10 8 11 25 = 2 Objects: 810 at offset 0, 111 at offset 25
352 // var objArray Array
353 // var offsetOld int
354 for i := 0; i < len(objDec); i += 2 {
355     offset, err := strconv.Atoi(objDec[i+1])
356     if err != nil {
357         return err
358     }
359     offset += objDec.FIRSTObjOffset
360     if i % 2 == 1 {
361         dict = string(decodedContent[offset:offset+1])
362         logDecompressPrint("parObjDec ObjectStream: objstring = %s\n", dict)
363         obj, err := compressObjObject(dict)
364         if err != nil {
365             return err
366         }
367         logDecompressPrint("parObjDec ObjectStream: [%d] = obj %s\n", i/2+1, obj)
368     } else {
369         objArray = append(objArray, obj)
370     }
371     if i == len(objDec) - 1 {
372         dict = string(decodedContent[offset:offset+1])
373         logDecompressPrint("parObjDec ObjectStream: objstring = %s\n", dict)
374         obj, err := compressObjObject(dict)
375         if err != nil {
376             return err
377         }
378         logDecompressPrint("parObjDec ObjectStream: [%d] = obj %s\n", i/2+1, obj)
379     }
380     objArray = append(objArray, obj)
381     offsetOld = offset
382 }
383 }
384 }

```

```

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     obj := NewObject()
652     l := Line(streamID)
653     if err := obj.Parse(l); err != nil {
654         log.Debug.Printf("parseRefStream: error in parseObjectAttributes()")
655         return nil, err
656     }
657
658     // Parse stream
659     log.Debug.Printf("parseRefStream: xrefInfo=%d genNum=%d, objectNumber=%d",
660         obj.XrefInfo(), obj.GenNum(), obj.ObjectNumber())
661     log.Debug.Printf("parseRefStream: deferencing object %d\n", obj.ObjectNumber())
662     o, err := obj.ParseObject(l)
663     if err != nil {
664         return nil, err
665     }
666     return nil, errors.Wrap(err, "parseRefStream: no object")
667 }
668
669 // ReadStream reads a stream from the given URL.
670 func ReadStream(url string) (parseRefStream, error) {
671     log.Debug.Printf("parseRefStream: we have an object: %d\n", o)
672
673     streamOffset := o.Offset()
674     obj := NewObject()
675     if err := obj.Parse(streamOffset, o.ObjectNumber(), streamOffset); err != nil {
676         return nil, err
677     }
678
679     // We have an end stream object
680     err := parseRefStreamInfo(obj, ctx.XrefTable)
681     if err != nil {
682         return nil, err
683     }
684
685     // Parse stream and create streamable entries for embedded objects.
686     err = extractRefStreamableEntriesFromStream(obj.Content, ctx)
687     if err != nil {
688         return nil, err
689     }
690 }
691
692 // Create streamable entry for XrefStreamEntry.
693 func CreateStreamableEntry(
694     entry *XrefStreamEntry,
695     flags flags,
696     offset Offset,
697     generation int,
698     objectNumber int,
699     object *Object)

```

```

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 s1, s2, s3 string
796
797     buf.WriteString("line: <sk>\n", line)
798
799     // Scan for dict start tag "\n"
800     for {
801         s1 = strings.Index(line, "\n")
802         if s1 >= 0 {
803             break
804         }
805         line, err = scanLine(s)
806         buf.WriteString("line: <sk>\n", line)
807         if err == nil {
808             return "", err
809         }
810     }
811
812     line = line[s1:]
813     buf.WriteString(line)
814     buf.WriteString("\n")
815
816     // Scan for dict start tag ">" but account for inner dicts.
817     line = line[2:]
818     for {
819         if len(line) == 0 {
820             line, err = scanLine(s)
821             if err == nil {
822                 return "", err
823             }
824             buf.WriteString(line)
825             buf.WriteString("\n")
826             line = line[2:]
827             buf.WriteString("scanHeader dictchar next line: <sk>\n", line)
828         }
829         s1 = strings.Index(line, "\n")
830         if s1 <= 0 {
831             s1 = strings.Index(line, ">")
832             if s1 >= 0 {

```

```

4820 if s[i] == 'mha' {
4821     colCount = 1
4822 } else if s[i] == 'o=0' {
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: %s\n", pdfVersion)
4832
4833 return pdfVersion, colCount, nil
4834 }
4835
4836 // popadeparser is a back file digesting content 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(ctt *Content) error {
4841     s := bufio.NewScanner(ctt)
4842     k := FrequencyGeneration
4843     ct := byteIndex + xRefTableEntry{
4844         Free:    true,
4845         Offset:   0,
4846         Generation: 0}
4847
4848     rs := ct.ReadRes
4849     rs.Count = ct.ReadRes.Count
4850     var off, offset index
4851
4852     rd, err := mapIndexToIndex(ctt, offset)
4853     if err == nil {
4854         return err
4855     }
4856
4857     s = bufio.NewReaderString(rs)
4858     s.Split(scanLines)
4859     bo := []byte{
4860         'w', 'r',
4861         'withn0b', 'bool',
4862         'withn0f', 'bool',
4863         'withn0z', 'bool',
4864     }
4865
4866     for {
4867         line, err := scanLineFrom(s)
4868         if err == nil {
4869             break
4870         }
4871         if withn0f {
4872             offset += int64(len(line) * colCount)
4873         }
4874         if withn0b {
4875             bo = append(bo, ' ')
4876         }
4877         if strings.Contains(line, "startover") {
4878             if i > 0 {

```

```

255 if (line[offset] == '\r')
256     offset++;
257 // Valid lines end
258 // If line[offset] = '\n'
259     offset++;
260 }
261 }
262 return
263 }
264
265 func lastStreamMarker(streamed <int, int, line string) {
266
267     if streamEnd != len(line)-streamStart {
268         // We found no another stream marker
269         streamed += 1
270         return
271     }
272
273     // We start searching after this stream marker.
274     bufpos := streamed + len('stream')
275
276     // Search for next stream marker.
277     n in strings.Index(line(bufpos), "stream")
278     if n < 0
279         // No stream marker within line buffer.
280         streamed += 1
281         return
282     }
283
284     // We found the next stream marker.
285     streamed += len("stream") + 1
286
287     if ending != 0 do streamed += ending {
288         // Found a stream marker of another object
289         streamed += 1
290     }
291 }
292
293 // process - PDF file buffer of sufficient size for parsing an object. So stream
294 func findInReader(buf []byte, endOfLine int, streamEnd int, streamOffset int) bool {
295     err := 0
296 }
297
298 // process: a gun obj ... obj dict ... stream ... data ... endstream ... endobj
299 //          stream
300 //          absent      -1 if absent      -1 if
301
302 //Log.Debug.Println("buffer: begin")
303
304 endOfLine, streamEnd = -1, -1
305
306 for ending <= 0 do streamed <- default
307
308     buf, err = greadObjBuf(0, bufio.ReadBufferSize, rd)
309     if err != nil
310         return nil, 0, 0, 0, err

```

```

363 // https://stackoverflow.com/questions/4913464/using-std-weak_ptr-to-avoid-memory-leak
374
375         .. err = r.Read(cursor)
376         if err == nil {
377             return nil, err
378         }
379     }
380
381     workBuf = curBuf
382     if preBuf == nil {
383         workBuf = append(curBuf, preBuf...)
384     }
385
386     j := strings.LastIndex(string(workBuf), "startref")
387     if j == -1 {
388         preBuf = curBuf
389         continue
390     }
391
392     p = workBuf[j+1len(string(workBuf)):]
393     posOff := strings.Index(string(p), "MEOF")
394     if posOff == -1 {
395         return nil, errors.New("pfcpu: no matching MEOF for startref")
396     }
397
398     p = p[posOff:]
399     offset, err := strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
400     if err == nil {
401         return nil, errors.New("pfcpu: corrupted last ref section")
402     }
403 }
404
405 log.Read.Printf("Offset last xrefsection: %d\n", offset)
406
407 return bufOffset, nil
408 }
409
410 // Read next subsection entry and generate corresponding ref table entry.
411 func (parser *ParserTableEntry) xrefScan, xrefBuf *xrefTable, objectNumber int) {
412     log.Read.Printf("xrefScan: %d\n", objectNumber)
413
414     log.Read.Printf("parserTableEntry: begin")
415
416     line, err = bufio.ReadString()
417     if err == nil {
418         return err
419     }
420
421     if xrefBuf.Exists(objectNumber) {
422         log.Read.Printf("xrefTableEntry: end - Skip entry %d - already assigned", objectNumber)
423         return nil
424     }
425
426     fields = strings.Split(line)
427     if len(fields) == 1 {
428         log.Read.Printf("xrefTableEntry: end - Skip entry %d - already assigned", objectNumber)
429         return nil
430     }
431     return errors.New("pfcpu: parserTableEntry: corrupt ref subsection")
432 }
433
434 }
435
436 }

```

```

600   std::ostringstream &objArray
601 {
602     log.Read_Printf("params/objectstream end")
603     return nil
604 }
605 // for each object embedded in this xdrStream create the corresponding xdr table
606 // which shall be present in the stream()
607 static inline void extractXdrTablesFromXdrStream(bf_t *bf, udr.XdrStreamInfo_t, cts
608         .Context) error {
609     log.Read_Printf("extractXdrTablesFromXdrStream begin")
610
611     // Note:
612     // A value of zero for an element in the m array indicates that the corresponding
613     // element shall not be present in the stream().
614     // The default value shall be zero, if there is none.
615     // If the value of an element is non-zero, the type field shall be present, and shall
616     // default to type 0.
617     int i = xsd.m[0]
618     int j = xsd.m[1]
619     int k = xsd.m[2]
620     xdrEntryType = "1 * 12 + 13"
621     xdrTableFromXdrStream = extractXdrTablesFromXdrStream: begin xdrEntryType =
622     "xdrEntryType"
623     return errors.New("pfcip: extractXdrTablesFromXdrStream: corrupt
624             info")
625 }
626 //
627 objCount := len(xsd.Objects)
628 log.Read_Printf("extractXdrTablesFromXdrStream: buf has %d b'v's",
629         objCount,xsd.Objects)
630 //
631 log.Read_Printf("extractXdrTablesFromXdrStream: len(buf)%d
632         ",len(buf))
633 if len(buf)>< objCount+entryLen{
634     // Sometimes there is an additional zero entry not accounted for by "index".
635     // This may happen because a writer and do not treat this as an error.
636     return errors.New("pfcip: extractXdrTablesFromXdrStream: corrupt
637             info")
638 }
639 //
640 j = 0
641 // bufio.Reader interprets the content of buf as an int64.
642 for _, b := range bf.Bytes(1) {
643     buf |= b
644     i += 1
645     if i == len(buf){
646         return
647     }
648 }

```

```

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

```

```

340 //
341 // If k == 0
342 //
343 // Check for err
344 //
345 ok_err = isEOL(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.Read,Print("scanTrailer dicttbl next line: <scan>, line)
371 //
372 } else {
373     //
374     // s = string.Index(line, ">")
375     //
376     if s < 0 {
377         //
378         // No go
379         //
380         k++
381         line = line[j+2]
382     }
383 //
384 } else {
385     //
386     // No go
387     //
388     if s < 0 {
389         //
390         // Check for dict
391         //
392         ok_err = isEOL(buf.String())
393         //
394         if err == nil && ok {
395             return buf.String(), nil
396         }
397     }
398 //
399 } else {
400     k++
401     line = line[j+2]
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 }

```

```

9780 // Issue trailer
9781 // off = processTrailer(ctx, s, string(bb))
9782         return err
9783     }
9784     continue
9785 }
9786 // Ignore all until "trailer".
9787 s = strings.Index(line, "trailer")
9788 if i > s {
9789     bb.append(bb, line...)
9790     withIndexof = true
9791     continue
9792 }
9793 l = strings.Index(line, "eof")
9794 if i > l {
9795     offset += int64(int(line) - eofCount)
9796     withIndexof = true
9797     continue
9798 }
9799 if withIndexof {
9800     s = strings.Index(line, "obj")
9801     if i > s {
9802         withIndexof = true
9803         off += offset
9804         bb.append(bb, line[i:s]...)
9805     }
9806     offset += int64(int(line) - eofCount)
9807     continue
9808 }
9809 // finish
9810 offset += int64(int(line) - eofCount)
9811 bb.append(bb, s)
9812 bb.append(bb, line...)
9813 s = strings.Index(line, "eofobj")
9814 if i > s {
9815     l = string(bb)
9816     objMr, generation, err = parseObjectAttributes(s)
9817     if err == nil {
9818         return err
9819     }
9820     off = off
9821     ctx.tailer[objMr] = &tableEntry{
9822         offset: false,
9823         offset: 0,
9824         generation: generation
9825     }
9826     bb = nil
9827     withIndexof = false
9828 }
9829 }
9830 return nil
9831 }
9832 }
9833 }
9834 // Build an iterator by reading all objects or when specified
9835 func buildIteratorStartingAt(ctx *Context, offset uint64) error {
9836     log.Debug.Print("buildIteratorStartingAt: begin")
9837 }

```

[illegible]


```

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

```

[illegible]

```

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

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry %d", objId)
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs length 2 or 4", objId)
2043             }
2044             offset, ok = a[0].(Integer)
2045             if !ok {
2046                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs Integer values", objId)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offset64
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs Integer values", objId)
2053             }
2054             ctx.OffsetOverLimitInitial = offset64
2055         }
2056     }
2057     return nil
2058 }
2059
2060 func LoadBinaryStream(ctx *Context, s *StreamReader, objId, genR int) error {
2061     var err error
2062
2063     // Load stream's content and store data into offsetable entry
2064     if err = LoadInfiniteStream(ctx, s, objId, genR); err != nil {
2065         return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d", objId)
2066     }
2067     ctx.Read.BinarySize += s.GetSize()
2068
2069     // Decode stream's content
2070     err = saveDecodedStreamContent(ctx, s, objId, genR, ctx.DecodedAllStreams)
2071     return err
2072 }
2073
2074 func updateLinearizationDict(ctx *Context, o Object) {
2075     switch o := o.(type) {
2076     case StreamDict:
2077         ctx.Add.LinearizationList += s.GetSize()
2078     }
2079 }

```

```

2030 }
2031 // Create a mutable version string (since it's in the catalog
2032 // and needs to sit as readonly in class compared to headerVersion).
2033 auto mutableRootVersion = xFileTable.HeaderTable() error {
2034     log.Read.Print("IdentifyRootVersion: begin")
2035     // Copy to get version from mutableHeaderTable
2036     RootVersionStr = mutableHeaderTable.ParseRootVersion()
2037     if err == nil {
2038         return err
2039     }
2040     if rootVersionStr == nil {
2041         return nil
2042     }
2043     // Validate version and save corresponding constant to xFileTable.
2044     rootVersion, err := PDPVersion(rootVersionStr)
2045     if err != nil {
2046         return errors.Wrap(err, "IdentifyRootVersion: unknown PDP Root version: %v", *rootVersionStr)
2047     }
2048     xFileTable.RootVersion = rootVersion
2049     // Since v1.1, the header version may be overridden by a version entry in the
2050     // manifest. If xFileTable.HeaderVersion < v1.1,
2051     log.Info.Print("IdentifyRootVersion: PDP version is %s - will ignore root
2052     version %s", *rootVersionStr, *xFileTable.HeaderVersion)
2053     xFileTable.HeaderVersion = rootVersionStr
2054     log.Read.Print("IdentifyRootVersion: end")
2055     return nil
2056 }
2057 // Parse all Objects including stream content from file and save to the corresponding
2058 // manifests.
2059 func (m *Manifest) processObjOfObjectStreamAndLinearizationDicts(
2060     auto dereference xFileTable *Context, conf *Configuration) error {
2061     log.Read.Print("dereferenceXFileTable: begin")
2062     xFileTable = ctx.XFileTable
2063     // Note for unencrypted files.
2064     // Mandatory provide users to open & display file.
2065     // Access may be restricted (upon access strategies).
2066     // Optionally provide contexts in order to gain unrestricted access.
2067     if err := m.dereferncryptContexts()

```

```

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.DirectIndex(); ok {
3428     // 1433
3429     o, err = deferPipelinesDirect(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("pdpicu: 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("pdpicu: 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("pdpicu: dereferenceInteger: corrupt integer")
3567     }
3568 }

```

```

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

```

[illegible]

```

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

```

```

1452 // @param {Object} ctx - Context object
1453 if found != dict.Fall(DecodParams) )
1454 {
1455     if found != nil
1456     {
1457         decodeParamsArr, ok = decodeParams.(Array)
1458         if !ok
1459             return nil, errors.New("pdcip: pdfFilterPipeline: expected decodeParams
1460             array corrupt")
1461     }
1462 }
1463
1464 // /var Printout("decodeParams: %s", decodeParams)
1465
1466 // filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamsArr,
1467 // decodeParams)
1468
1469 log.Read.FilterPipeline("pdfFilterPipeline: end")
1470
1471 return filterPipeline, err
1472
1473 // =====
1474 func streamDictForObj(c *Context, d Dict, objKey, streamIn int, streamInSet
1475 int) (uint64, GoStreamDict, error) {
1476     streamLength, streamLengthRef = d.Length()
1477
1478     if streamIn == 0 {
1479         return sd, errors.New("pdcip: streamDictForObj: stream object without
1480         streamIn")
1481     }
1482
1483     filterPipeline, err = pdfFilterPipeline(c, d)
1484     if err == nil {
1485         return sd, err
1486     }
1487
1488     streamOffset = offset
1489
1490 // We have a stream object
1491 sd = NewStreamInSet(d, streamOffset, streamLength, streamLengthRef, filterPipeline)
1492
1493 log.Read.Print("streamDictForObj: end, streamInSet %d\n", objKey)
1494
1495 return sd, nil
1496 }
1497
1498 // =====
1499 func dictCtx(Context, d Dict, objKey, err, endIn, streamIn int) (d Dict, err
1500 error) {
1501     if ctx.EncKey == nil {
1502         ctx.EncKey = decryptObjStreamDict(d, objKey, err, ctx.EncKey, ctx.AES4Strings,
1503         ctx.AES)
1504         if err == nil {
1505             return nil, err
1506         }
1507     }
1508
1509     if endIn == 0 || (streamIn < 0 || streamIn == endIn) {
1510         log.Read.Print("dict: end, %d\n", objKey)
1511         d = di
1512     }
1513 }

```

```

3730         return dc, nil
3731     }
3732 }
3733 func dereferenceObject(cxt *Context, objectNumber int) (oic, error) {
3734     oic := dereferenceObject(cxt, objectNumber)
3735     if err == nil {
3736         return nil, err
3737     }
3738     // ok = o.(float)
3739     if tok != "f" {
3740         return nil, errors.New("pdpcc: dereferenceObject corrupt dict")
3741     }
3742     return dc, nil
3743 }
3744 // dereference a Message object representing an IPv6 value.
3745 func int64Object(cxt *Context, objectNumber int) (*int64, error) {
3746     log.Read.Printf("int64Object begin: %d\n", objectNumber)
3747     // ok = o.(int64)
3748     oic := dereferenceInteger(cxt, objectNumber)
3749     if err == nil {
3750         return nil, err
3751     }
3752     id4 := int64(4).Value()
3753     log.Read.Printf("int64Object end: %d\n", objectNumber)
3754     return id4, nil
3755 }
3756 // Reads and returns a File buffer with length + stream length using provided reader
3757 // positioned at offset.
3758 func readStreamFrom(r io.Reader, streamLength int) ([]byte, error) {
3759     log.Read.Printf("readStreamFrom: begin streamLength:%d\n", streamLength)
3760     buf := make([]byte, streamLength)
3761     for totalCount := 0; totalCount < streamLength; {
3762         count, err := r.Read(buf[totalCount:])
3763         if err == nil {
3764             return nil, err
3765         }
3766         log.Read.Printf("readCountFrom: count=%d, bufLen=%d(x)%v", count,
3767             len(buf), hexDump)
3768         totalCount += count
3769     }
3770     log.Read.Printf("readCountFrom: endBuf")
3771     return buf, nil
3772 }

```

```

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

```

```

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 = handleLinearizationAndStats(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         err = loadStream(ctxt, sd, objNr, entry.generation)
2370         if err == nil {
2371             return err
2372         }
2373     }
2374     entry.Object = sd
2375 }
2376
2377 // // Log and Print ("dereferencedObject: and objId of %v\n", objNr,
2378 // // objNrDict, entry.Object)
2379 log.Printf("dereferencedObject: and objId of %v\n", objNr,
2380 objNrDict, entry.Object)
2381 logStream(entry.Object)
2382 return nil
2383 }
2384
2385 func processBidsAndCounts(defTable *XRefTable, D Dict) {
2386     for _, n := range o {
2387         match ok := n.Key()
2388         case IndexDict:
2389             entry, ok := defTable.LookupTableEntry(defIndexDict(n))
2390             if ok {
2391                 entry.Count++
2392             }
2393         case Dict:
2394             processBidsAndCounts(defTable, o)
2395         case Array:
2396             processBidsAndCounts(defTable, o)
2397     }
2398 }
2399
2400 }
2401
2402 }

```

```

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

```

[illegible][illegible]

```

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

```

```

2020: // 2020
2021: func processArrayByCounts(x:Iterable, y:Iterable, a:Array) {
2022:     for _ in range a {
2023:         switch o in a.toArray() {
2024:             case IndexDefect:
2025:                 entry, ok = x.elementAt(i).flatMap{f, _} {
2026:                     if ok {
2027:                         entry, _ = Count++
2028:                     }
2029:                 }
2030:             case Count:
2031:                 processByCounts(x.toIterable(), o)
2032:             case Array:
2033:                 processByCounts(x.toIterable(), o)
2034:             }
2035:         }
2036:     }
2037: }
2038:
2039: func processByCounts(x:Iterable, o:Iterable, o:Object) {
2040:     switch o in o.toArray() {
2041:         case Count:
2042:             processByCounts(x.toIterable(), o)
2043:         case StreamDefect:
2044:             processByCounts(x.toIterable(), o, dict)
2045:         case Array:
2046:             processArrayByCounts(x.toIterable(), o)
2047:         }
2048:     }
2049: }
2050:
2051: // 2051
2052: // 2052
2053: // 2053
2054: // 2054
2055: // 2055
2056: // 2056
2057: // 2057
2058: // 2058
2059: // 2059
2060: // 2060
2061: // 2061
2062: // 2062
2063: // 2063
2064: // 2064
2065: // 2065
2066: // 2066
2067: // 2067
2068: // 2068
2069: // 2069
2070: // 2070
2071: // 2071
2072: // 2072
2073: // 2073
2074: // 2074
2075: // 2075
2076: // 2076
2077: // 2077
2078: // 2078
2079: // 2079
2080: // 2080
2081: // 2081
2082: // 2082
2083: // 2083
2084: // 2084
2085: // 2085
2086: // 2086
2087: // 2087
2088: // 2088
2089: // 2089
2090: // 2090
2091: // 2091
2092: // 2092
2093: // 2093
2094: // 2094
2095: // 2095
2096: // 2096
2097: // 2097
2098: // 2098
2099: // 2099
2100: // 2100
2101: // 2101
2102: // 2102
2103: // 2103
2104: // 2104
2105: // 2105
2106: // 2106
2107: // 2107
2108: // 2108
2109: // 2109
2110: // 2110
2111: // 2111
2112: // 2112
2113: // 2113
2114: // 2114
2115: // 2115
2116: // 2116
2117: // 2117
2118: // 2118
2119: // 2119
2120: // 2120
2121: // 2121
2122: // 2122
2123: // 2123
2124: // 2124
2125: // 2125
2126: // 2126
2127: // 2127
2128: // 2128
2129: // 2129
2130: // 2130
2131: // 2131
2132: // 2132
2133: // 2133
2134: // 2134
2135: // 2135
2136: // 2136
2137: // 2137
2138: // 2138
2139: // 2139
2140: // 2140
2141: // 2141
2142: // 2142
2143: // 2143
2144: // 2144
2145: // 2145
2146: // 2146
2147: // 2147
2148: // 2148
2149: // 2149
2150: // 2150
2151: // 2151
2152: // 2152
2153: // 2153
2154: // 2154
2155: // 2155
2156: // 2156
2157: // 2157
2158: // 2158
2159: // 2159
2160: // 2160
2161: // 2161
2162: // 2162
2163: // 2163
2164: // 2164
2165: // 2165
2166: // 2166
2167: // 2167
2168: // 2168
2169: // 2169
2170: // 2170
2171: // 2171
2172: // 2172
2173: // 2173
2174: // 2174
2175: // 2175
2176: // 2176
2177: // 2177
2178: // 2178
2179: // 2179
2180: // 2180
2181: // 2181
2182: // 2182
2183: // 2183
2184: // 2184
2185: // 2185
2186: // 2186
2187: // 2187
2188: // 2188
2189: // 2189
2190: // 2190
2191: // 2191
2192: // 2192
2193: // 2193
2194: // 2194
2195: // 2195
2196: // 2196
2197: // 2197
2198: // 2198
2199: // 2199
2200: // 2200
2201: // 2201
2202: // 2202
2203: // 2203
2204: // 2204
2205: // 2205
2206: // 2206
2207: // 2207
2208: // 2208
2209: // 2209
2210: // 2210
2211: // 2211
2212: // 2212
2213: // 2213
2214: // 2214
2215: // 2215
2216: // 2216
2217: // 2217
2218: // 2218
2219: // 2219
2220: // 2220
2221: // 2221
2222: // 2222
2223: // 2223
2224: // 2224
2225: // 2225
2226: // 2226
2227: // 2227
2228: // 2228
2229: // 2229
2230: // 2230
2231: // 2231
2232: // 2232
2233: // 2233
2234: // 2234
2235: // 2235
2236: // 2236
2237: // 2237
2238: // 2238
2239: // 2239
2240: // 2240
2241: // 2241
2242: // 2242
2243: // 2243
2244: // 2244
2245: // 2245
2246: // 2246
2247: // 2247
2248: // 2248
2249: // 2249
2250: // 2250
2251: // 2251
2252: // 2252
2253: // 2253
2254: // 2254
2255: // 2255
2256: // 2256
2257: // 2257
2258: // 2258
2259: // 2259
2260: // 2260
2261: // 2261
2262: // 2262
2263: // 2263
2264: // 2264
2265: // 2265
2266: // 2266
2267: // 2267
2268: // 2268
2269: // 2269
2270: // 2270
2271: // 2271
2272: // 2272
2273: // 2273
2274: // 2274
2275: // 2275
2276: // 2276
2277: // 2277
2278: // 2278
2279: // 2279
2280: // 2280
2281: // 2281
2282: // 2282
2283: // 2283
2284: // 2284
2285: // 2285
2286: // 2286
2287: // 2287
2288: // 2288
2289: // 2289
2290: // 2290
2291: // 2291
2292: // 2292
2293: // 2293
2294: // 2294
2295: // 2295
2296: // 2296
2297: // 2297
2298: // 2298
2299: // 2299
2300: // 2300
2301: // 2301
2302: // 2302
2303: // 2303
2304: // 2304
2305: // 2305
2306: // 2306
2307: // 2307
2308: // 2308
2309: // 2309
2310: // 2310
2311: // 2311
2312: // 2312
2313: // 2313
2314: // 2314
2315: // 2315
2316: // 2316
2317: // 2317
2318: // 2318
2319: // 2319
2320: // 2320
2321: // 2321
2322: // 2322
2323: // 2323
2324: // 2324
2325: // 2325
2326: // 2326
2327: // 2327
2328: // 2328
2329: // 2329
2330: // 2330
2331: // 2331
2332: // 2332
2333: // 2333
2334: // 2334
2335: // 2335
2336: // 2336
2337: // 2337
2338: // 2338
2339: // 2339
2340: // 2340
2341: // 2341
2342: // 2342
2343: // 2343
2344: // 2344
2345: // 2345
2346: // 2346
2347: // 2347
2348: // 2348
2349: // 2349
2350: // 2350
2351: // 2351
2352: // 2352
2353: // 2353
2354: // 2354
2355: // 2355
2356: // 2356
2357: // 2357
2358: // 2358
2359: // 2359
2360: // 2360

```

```

2530 //
2531 if err == nil {
2532     return err
2533 }
2534 //
2535 // If the owner password does not match we generally move on if the user password
2536 // errors.
2537 //
2538 // Unless we need to limit on a user's owner password due to the specific
2539 // amount in use password.
2540 if tok != newPasswordHandler(password,ctx.Cmd) {
2541     return errors.New("password: please provide the master password with 'opw'")
2542 }
2543 //
2544 //
2545 // Generally the user password, which is also regarded as the master password or
2546 // pre-password.
2547 //
2548 // It is sufficient for moving on. A password change is an occasion since it
2549 // is not.
2550 if ok := newPasswordHandler(password,ctx.Cmd) {
2551     return nil
2552 }
2553 //
2554 // ok,err := validatePermissions(ctx)
2555 //
2556 if err == nil {
2557     return err
2558 }
2559 //
2560 // If ok {
2561 //
2562 //     return errors.New("password: corrupted permissions after opw ok")
2563 //
2564 // }
2565 //
2566 // return nil
2567 //
2568 //
2569 // Validate the user password ok, document opw password.
2570 ok,err := validatePermissions(ctx)
2571 if err == nil {
2572     return err
2573 }
2574 //
2575 // If ok {
2576 //
2577 //     return errors.New("password: please provide the correct password")
2578 //
2579 // }
2580 //
2581 //
2582 // //fmt.Printf("opw ok: %d\n", ok)
2583 //
2584 // return handlePermissions(ctx)
2585 //
2586 //
2587 //
2588 //
2589 //
2590 //
2591 //
2592 //
2593 //
2594 //
2595 //
2596 //
2597 //
2598 //
2599 //
2600 //
2601 //
2602 //
2603 //
2604 //
2605 //
2606 //
2607 //
2608 //
2609 //
2610 //
2611 //
2612 //
2613 //
2614 //
2615 //
2616 //
2617 //
2618 //
2619 //
2620 //
2621 //
2622 //
2623 //
2624 //
2625 //
2626 //
2627 //
2628 //
2629 //
2630 //
2631 //
2632 //
2633 //
2634 //
2635 //
2636 //
2637 //
2638 //
2639 //
2640 //
2641 //
2642 //
2643 //
2644 //
2645 //
2646 //
2647 //
2648 //
2649 //
2650 //
2651 //
2652 //
2653 //
2654 //
2655 //
2656 //
2657 //
2658 //
2659 //
2660 //
2661 //
2662 //
2663 //
2664 //
2665 //
2666 //
2667 //
2668 //
2669 //
2670 //
2671 //
2672 //
2673 //
2674 //
2675 //
2676 //
2677 //
2678 //
2679 //
2680 //
2681 //
2682 //
2683 //
2684 //
2685 //
2686 //
2687 //
2688 //
2689 //
2690 //
2691 //
2692 //
2693 //
2694 //
2695 //
2696 //
2697 //
2698 //
2699 //
2700 //
2701 //
2702 //
2703 //
2704 //
2705 //
2706 //
2707 //
2708 //
2709 //
2710 //
2711 //
2712 //
2713 //
2714 //
2715 //
2716 //
2717 //
2718 //
2719 //
2720 //
2721 //
2722 //
2723 //
2724 //
2725 //
2726 //
2727 //
2728 //
2729 //
2730 //
2731 //
2732 //
2733 //
2734 //
2735 //
2736 //
2737 //
2738 //
2739 //
2740 //
2741 //
2742 //
2743 //
2744 //
2745 //
2746 //
2747 //
2748 //
2749 //
2750 //
2751 //
2752 //
2753 //
2754 //
2755 //
2756 //
2757 //
2758 //
2759 //
2760 //
2761 //
2762 //
2763 //
2764 //
2765 //
2766 //
2767 //
2768 //
2769 //
2770 //
2771 //
2772 //
2773 //
2774 //
2775 //
2776 //
2777 //
2778 //
2779 //
2780 //
2781 //
2782 //
2783 //
2784 //
2785 //
2786 //
2787 //
2788 //
2789 //
2790 //
2791 //
2792 //
2793 //
2794 //
2795 //
2796 //
2797 //
2798 //
2799 //
2800 //
2801 //
2802 //
2803 //
2804 //
2805 //
2806 //
2807 //
2808 //
2809 //
2810 //
2811 //
2812 //
2813 //
2814 //
2815 //
2816 //
2817 //
2818 //
2819 //
2820 //
2821 //
2822 //
2823 //
2824 //
2825 //
2826 //
2827 //
2828 //
2829 //
2830 //
2831 //
2832 //
2833 //
2834 //
2835 //
2836 //
2837 //
2838 //
2839 //
2840 //
2841 //
2842 //
2843 //
2844 //
2845 //
2846 //
2847 //
2848 //
2849 //
2850 //
2851 //
2852 //
2853 //
2854 //
2855 //
2856 //
2857 //
2858 //
2859 //
2860 //
2861 //
2862 //
2863 //
2864 //
2865 //
2866 //
2867 //
2868 //
2869 //
2870 //
2871 //
2872 //
2873 //
2874 //
2875 //
2876 //
2877 //
2878 //
2879 //
2880 //
2881 //
2882 //
2883 //
2884 //
2885 //
2886 //
2887 //
2888 //
2889 //
2890 //
2891 //
2892 //
2893 //
2894 //
2895 //
2896 //
2897 //
2898 //
2899 //
2900 //
2901 //
2902 //
2903 //
2904 //
2905 //
2906 //
2907 //
2908 //
2909 //
2910 //
2911 //
2912 //
2913 //
2914 //
2915 //
2916 //
2917 //
2918 //
2919 //
2920 //
2921 //
2922 //
2923 //
2924 //
2925 //
2926 //
2927 //
2928 //
2929 //
2930 //
2931 //
2932 //
2933 //
2934 //
2935 //
2936 //
2937 //
2938 //
2939 //
2940 //
2941 //
2942 //
2943 //
2944 //
2945 //
2946 //
2947 //
2948 //
2949 //
2950 //
2951 //
2952 //
2953 //
2954 //
2955 //
2956 //
2957 //
2958 //
2959 //
2960 //
2961 //
2962 //
2963 //
2964 //
2965 //
2966 //
2967 //
2968 //
2969 //
2970 //
2971 //
2972 //
2973 //
2974 //
2975 //
2976 //
2977 //
2978 //
2979 //
2980 //
2981 //
2982 //
2983 //
2984 //
2985 //
2986 //
2987 //
2988 //
2989 //
2990 //
2991 //
2992 //
2993 //
2994 //
2995 //
2996 //
2997 //
2998 //
2999 //
3000 //
3001 //
3002 //
3003 //
3004 //
3005 //
3006 //
3007 //
3008 //
3009 //
3010 //
3011 //
3012 //
3013 //
3014 //
3015 //
3016 //
3017 //
3018 //
3019 //
3020 //
3021 //
3022 //
3023 //
3024 //
3025 //
3026 //
3027 //
3028 //
3029 //
3030 //
3031 //
3032 //
3033 //
3034 //
3035 //
3036 //
3037 //
3038 //
3039 //
3040 //
3041 //
3042 //
3043 //
3044 //
3045 //
3046 //
3047 //
3048 //
3049 //
3050 //
3051 //
3052 //
3053 //
3054 //
3055 //
3056 //
3057 //
3058 //
3059 //
3060 //
3061 //
3062 //
3063 //
3064 //
3065 //
3066 //
3067 //
3068 //
3069 //
3070 //
3071 //
3072 //
3073 //
3074 //
3075 //
3076 //
3077 //
3078 //
3079 //
3080 //
3081 //
3082 //
3083 //
3084 //
3085 //
3086 //
3087 //
3088 //
3089 //
3090 //
3091 //
3092 //
3093 //
3094 //
3095 //
3096 //
3097 //
3098 //
3099 //
3100 //
3101 //
3102 //
3103 //
3104 //
3105 //
3106 //
3107 //
3108 //
3109 //
3110 //
3111 //
3112 //
3113 //
3114 //
3115 //
3116 //
3117 //
3118 //
3119 //
3120 //
3121 //
3122 //
3123 //
3124 //
3125 //
3126 //
3127 //
3128 //
3129 //
3130 //
3131 //
3132 //
3133 //
3134 //
3135 //
3136 //
3137 //
3138 //
3139 //
3140 //
3141 //
3142 //
3143 //
3144 //
3145 //
3146 //
3147 //
3148 //
3149 //
3150 //
3151 //
3152 //
3153 //
3154 //
3155 //
3156 //
3157 //
3158 //
3159 //
3160 //
3161 //
3162 //
3163 //
3164 //
3165 //
3166 //
3167 //
3168 //
3169 //
3170 //
3171 //
3172 //
3173 //
3174 //
3175 //
3176 //
31
```