

[illegible]

```

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

```

```

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

```

[illegible]

```

987 // dict
988 log.Mod.Printf("line (len md) %s\n", len(line), line)
989
990 trailerString, err := scanFields, trailerString)
991 if err == nil {
992     return nil, err
993 }
994
995 log.Mod.Printf("processTrailer: trailerString: (len md) %s\n",
996     len(trailerString), trailerString)
997
998 o, err := paraObject(trailerString)
999 if err == nil {
1000     return nil, err
1001 }
1002
1003 trailerPkt, ok := o.(Dict)
1004 if !ok {
1005     return nil, errors.New("pdpdu: processTrailer: corrupt trailer dict")
1006 }
1007
1008 log.Mod.Printf("processTrailer: trailerDict(%s\n", trailerDict)
1009
1010 return paraTrailerDict(trailerDict, ctx)
1011 }
1012
1013 // paraSubShfSection into corresponding number of shf table entries
1014 func paraSubShfSection(*bufio.Scanner, ctx.Context) (*uint64, error) {
1015
1016     log.Mod.Printf("paraSubShfSection begin")
1017
1018     line, err := scanFields()
1019     if err == nil {
1020         return nil, err
1021     }
1022
1023     log.Mod.Printf("paraSubShfSection: %s\n", line)
1024
1025     fields = strings.Fields(line)
1026
1027 // Process all sub sections of this shf section.
1028 for strings.HasPrefix(line, "trailer") && len(fields) == 2 {
1029     if err := paraTrailer(fields[0], ctx.ShfFields, Fields); err == nil {
1030         return nil, err
1031     }
1032 }
1033
1034 // trailer or another shf table subsection ?
1035 if line[0] == scanFields() {
1036     return nil, err
1037 }
1038
1039 // if many line try next line for trailer
1040 if len(line) == 0 {
1041     if trailer, err := scanFields(); err == nil {
1042         return nil, err
1043     }
1044 }
1045 }

```

```

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

```

```

2520 //
2521 log.Debug.Print("Read begin")
2522
2523 ctx, err := NewContext(s, config)
2524 if err != nil {
2525     return nil, err
2526 }
2527
2528 if ctx.ReadOnly {
2529     // Log info,Println("PDF Version 1.0 conforming reader")
2530
2531     // Log info,Println("PDF Version 1.4 conforming reader - no object streams
2532     // references allowed")
2533 }
2534
2535 // Populate sObjTable.
2536 if err := readObjTable(ctx); err != nil {
2537     return nil, errors.Wrap(err, "objTable failed")
2538 }
2539
2540 // Make all objects explicitly available (load into memory) in corresponding
2541 // sObjTable entry.
2542 // Also decode any indirect object streams.
2543 if err := deferDecodeObjTable(ctx, config); err != nil {
2544     return nil, err
2545 }
2546
2547 // Some scanners write an incorrect file size trailer.
2548 if ctx.ObjTable.Size < len(ctx.ObjTable.Table) {
2549     ctx.ObjTable.Size = len(ctx.ObjTable.Table)
2550 }
2551
2552 log.Debug.Print("Read: end")
2553
2554 return ctx, nil
2555 }
2556
2557 // ScanLines is a split function for a Scanner that returns each line of
2558 // text, stripped of any trailing end-of-line markers. The returned line may
2559 // be a single line, or a carriage return followed by a line, or a line
2560 // by a newline or a carriage return or one newline.
2561 //
2562 // The number of lines will be returned even if it has no newline.
2563 func scanLines(data []byte, offset bool) (advance int, token []byte, err error) {
2564
2565     if atEOF := len(data) == 0 {
2566         return 0, nil, nil
2567     }
2568
2569     indexR := bytes.IndexByte(data, '\r')
2570     indexLF := bytes.IndexByte(data, '\n')
2571
2572     switch {
2573     case indexR >= 0 && indexR > 0:
2574         if indexR < indexLF {
2575             // If indexR < indexLF
2576             return indexR + 1, data[:indexR], nil
2577         }
2578     }
2579     //
2580 }

```

```

6982
6983 // Parse the table header and create corresponding xheffable objects
6984 func parseHeffableTableSubSection() *bufio.Scanner, *heffable.Xheffable, fields []string {
7985     log.Read.Println("parseHeffableTableSubSection: begin")
7986
7987     startOfNumber, err := strconv.Atoi(fields[0])
7988     if err != nil {
7989         return err
7990     }
7991
7992     objCount, err := strconv.Atoi(fields[1])
7993     if err != nil {
7994         return err
7995     }
7996
7997     log.Read.Println("deducted xheff sub-section, startObjId length="+startOfNumber,
7998         startObjCount=objCount)
7999
8000     // Process all entries of this subsection into xheffable entries.
8001     for i := 0; i < objCount; i++ {
8002         if err := parseHeffableEntry(i, xheffable, startObjNumbers); err != nil {
8003             return err
8004         }
8005     }
8006
8007     log.Read.Println("parseHeffableTableSubSection: end")
8008
8009     return nil
8010 }
8011
8012 // Parse compressed object
8013 func compressedObject(s string) (Object, error) {
8014
8015     log.Read.Println("compressedObject: begin")
8016
8017     o, err := parseObject(s)
8018     if err != nil {
8019         return nil, err
8020     }
8021
8022     d, ok := o.(Dict)
8023     if !ok {
8024         // Return trivial Object: Integer, Array, etc.
8025         log.Read.Println("compressedObject: end, any other than dict")
8026         return o, nil
8027     }
8028
8029     streamLength, streamOffset := d.Length()
8030     if streamLength == nil || d.StreamLength() == nil {
8031         // Return dict
8032         log.Read.Println("compressedObject: end, dict")
8033         return s, nil
8034     }
8035
8036     return nil, errors.New("Ofcpu: compressedObject: stream objects are not to be
8037         stored in dict stream")
8038 }

```

```

500 // already initialized, so just reuse it
501     if (value !=
502         ctx.getAddressObjectNumber() < SafeFdsFactory
503             .get())
504     {
505         //++
506     }
507
508     Log.Read_Println("extractSafeFdsFromStreamFromFdsStream: end")
509
510     return nil
511 }
512
513 // safeFdsFromStream(ctxt *Context, o Object, objNr int, streamOffset int64)
514 // *SafeFdsStream error()
515 //
516 //   * must be init
517 //   * do it in dict
518 //   * if nil &
519 //   * return nil, errors.New("affpo: safeFdsFromStream no dict")
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 //

```

```

60 // If offset is null, we will
61 // do a rough reference stream.
62 if (ctx.Reader() != <libfuzzer>.Version0) > VM { ctx.ReadHeader(
63     return ctx.Front(<libfuzzer>.FrontalIndex); } Put a constant scanner:
64 from incompatible version = <libfuzzer>.Version0String();
65 log.Head.Print("parseHeaderError")
66 // Attention: In some previous test functions, if there is any,
67 return offset, nil
68
69 // This file is using cross reference streams.
70
71
72 if ctx.ReadHeader(
73     ctx.ReadHeader() == true
74     ctx.ReadUsingHeaderStreams = true
75 )
76
77 // Is conformant readers process hidden objects contained
78 // in header before attempting to process any previous WebSection.
79 // This operation is important to have few errors for hidden entries.
80 // We may also in WebSections only.
81
82 if err := parseHeaderAndStreamOfHeaderStreams, ctx) == nil {
83     return nil, err
84 }
85
86
87 log.Head.Print("parseHeaderError")
88
89 return offset, nil
90
91
92 func scanLine(s *bufio.Scanner) (string, error) {
93     if s == nil || s.Scan() {
94         if s.Err() == nil {
95             return "", s.Err()
96         }
97         return "", errors.New("defocus: scanLine was returning nothing")
98     }
99     return s.Text(), nil
100 }
101
102
103 func scanLine(s *bufio.Scanner) (sl string, err error) {
104     for i := 0; i <= 1; i++ {
105         sl, err = scanLine(s)
106         if err == nil {
107             break
108         }
109         if !isNil(s) {
110             break
111         }
112     }
113     return sl, err
114 }
115
116 // Remove comment.
117 // Return index, "x"
118 // x >= 0
119 // x = nil()

```

```

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

```

```

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

```

```

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

```

```

340 // Now all objects are on object stream and we can move them into objStream[i].objArray
341 func parseObjStream(oid ObjectsStream) error {
342     log.Read.Printf("parseObjStream begin decoding %d objects\n", oid.objCount)
343     // Read first object
344     decodedContent := oid.Content
345     preDecodedContent := decodedContent[oid.FirstObjOffset:]
346
347     var objArray Array
348     var offset int
349     for i := 0; i < len(objs); i++ {
350         offset, err = stream.At(offset+1)
351         if err != nil {
352             return err.Newf("offset: %d offset: %d, %d offset %d",
353                 // e.g., 10 0 11 25 + 2 Objects: #10 at offset 0, #11 at offset 25
354                 i, objArray.Array,
355                 var offset int
356
357                 for i := 0; i < len(objs); i++ {
358                     offset, err = stream.At(offset+1)
359                     if err != nil {
360                         return err
361                     }
362
363                     offset += oid.FirstObjOffset
364
365                     if i == 0 {
366                         dstr = string(decodedContent[offset:offset])
367                         log.Read.Printf("parseObjStream: objString = '%v'\n", dstr)
368                         o, err = compressObjject(dstr)
369                         if err != nil {
370                             return err
371                         }
372                     }
373
374                     log.Read.Printf("parseObjStream: [%d] = obj %s\n", i/2-1, objStr[i],
375                         objArray = append(objArray, o)
376
377                     if i == len(objs)-2 {
378                         dstr = string(decodedContent[offset:])
379                         log.Read.Printf("parseObjStream: objString = '%v'\n", dstr)
380                         o, err = compressObjject(dstr)
381                         if err != nil {
382                             return err
383                         }
384
385                         log.Read.Printf("parseObjStream: [%d] = obj %s\n", i/2, objStr[i],
386                             objArray = append(objArray, o)
387
388                     }
389
390                     offset += offset
391
392                 }
393             }

```

```

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

```

```

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

```

[illegible][illegible]

```

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

```

```

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

```

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

```

```

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

```

```

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

```

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

```



```

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, decodeParamsArray 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 || decodeParams.Array() == nil {
1593             filterPipeline = append(filterPipeline, HFilter{Name:
1594 filterName, DecodeParams: nil})
1595             continue
1596         }
1597         dict, ok = decodeParams.Array().Dict()
1598         if !ok {
1599             return nil, errors.New("pdcgo: buildFilterPipeline(): IndirectHeader
1600 corrupt")
1601         }
1602         if !ok {
1603             return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s",
1604 dict)
1605         }
1606         if s, err := dereferenceDict(dict, indirect.ObjectNumber.Value());
1607         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 filter pipeline associated with this stream dict.
1621 func buildFilterPipeline(ctx *Context, dict Dict) (*FilterPipeline, error) {
1622     log.Read.Printf("pdcfilterPipeline: begin")
1623
1624     var err error
1625
1626     if found := dict.Find("Filter") {
1627         if found {
1628             // stream is not compressed
1629         }
1630     }

```

[illegible]

```

1130 // Save the saveDecodedContentContent to ctx.Content, id, saveDecodedContent, objName, goenv int, decode
1131 // err error()
1132
1133 // Log.Read.Print("saveDecodedContentContent: begin decode %v", decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if ctx.will nil on ctx.IsKeykey == nil {
1137     if !ctx.filterPolicies().Name == "Crypt" {
1138         return nil
1139     }
1140 }
1141
1142 // Special case: If the length of the encoded data is 0, we do not need to decode anything.
1143 if ctx.Len(StdRaw) == 0 {
1144     StdContent = StdRaw
1145     return nil
1146 }
1147
1148 // Std gets created after StdStream parsing.
1149 // StdStream is not encrypted.
1150 if StdRaw == StdContent {
1151     StdRaw, err = DecryptStdContent(StdRaw, objName, goenv, ctx.IsKeykey, ctx.AESStreamSize,
1152         ctx.IsKeykey)
1153     if err == nil {
1154         return err
1155     }
1156     StdContent = StdRaw
1157     StdStream.Length = 0
1158 }
1159
1160 // If decode
1161     return nil
1162 }
1163
1164 // Actual decoding of content stream.
1165 err = decodeContentStream()
1166 if err == filter.StreamSupportFilter {
1167     err = nil
1168 }
1169
1170 if err == nil {
1171     return err
1172 }
1173
1174 Log.Read.Print("saveDecodedContentContent: end")
1175
1176 return nil
1177
1178 // Decode compressed objectTableEntry
1179 func DecodeCompressedObjectTableEntry (table *Table, objectNumber int, entry
1180     *ObjectEntry) error {
1181     Log.Read.Print("decodeCompressedObjectTableEntry: compressed object id at %d\n",
1182         objectNumber, entry.ObjectStream, entry.ObjectStreamLen)
1183
1184     // Handle stream entry as reference object stream.
1185     objectStreamTableEntry, ok := table.Lookup(entry.ObjectStream)
1186     if !ok {

```

```

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

```

```

2520 }
2521 // Create a mutable version of root (since it's a catalog
2522 // and root is not a rootversion (as opposed to headerVersion).
2523 std::weak_ptr<RootVersion> xRefTable = RootVersion::error {
2524     log.Read.Print("IdentifyRootVersion: begin")
2525     // Copy to get version from xRefTable.
2526     RootVersion* rtr = new RootTable.ParseRootVersion()
2527     if err == nil {
2528         return err
2529     }
2530     if rootVersionStr == nil {
2531         return nil
2532     }
2533     // Validate version and save corresponding constant to xRefTable.
2534     rootVersion, err = PDPVersion.RootVersionStr()
2535     if err == nil {
2536         return err, wrap(err, "IdentifyRootVersion: unknown PDP Root version:
2537             '%s'", rootVersionStr)
2538     }
2539     xRefTable.RootVersion = rootVersion
2540     // Since it's a header version we can override by a version entry in the
2541     // catalog.
2542     if xRefTable.HeaderVersion < v1 {
2543         log.Info.Print("IdentifyRootVersion: PDP version is %s - will ignore root
2544             version %s",
2545             xRefTable.HeaderVersion, rootVersionStr)
2546         log.Read.Print("IdentifyRootVersion: end")
2547         return nil
2548     }
2549     // Parse all Objects including stream content from file and save to the corresponding
2550     // headerTables.
2551     std::weak_ptr<Object> obj = Context, ctx = Configuration::error {
2552     log.Read.Print("DeserializeObjectTable: begin")
2553     xRefTable = ctx.xRefTable
2554     // Note for unencrypted files.
2555     // Mandatory provide users to open & display file.
2556     // Access may be restricted (Open access strategies).
2557     // Optionally provide contexts in order to gain unrestricted access.
2558     if err == ObjectError::Context {
2559         return err
2560     }
2561 }

```

```

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

```

```

3452         }
3453         return nil, nil
3454     }
3455 }
3456 // compressed stream.
3457 //
3458 var filterPipeline []PFFilter
3459 //
3460 if indexOf, ok := o.(IndirectRef); ok {
3461     o, err = derefResourceObject(ctx, indexOf.ObjectNumber.Value())
3462     if err != nil {
3463         return nil, err
3464     }
3465 }
3466 //
3467 //fmt.Printf("Decomposed filter obj: %v\n", o)
3468 //
3469 if name, ok := o.(Name); ok {
3470     //
3471     // single filter.
3472     //
3473     filterName := name.String()
3474     //
3475     o, found = dict.Find("DecodedParam")
3476     if !found {
3477         // w/o decoded parameter.
3478         //
3479         // See Read-Header-PFFilterPipeline and w/o decode param"
3480         return append(filterPipeline, PFFilterName: filterName, DecodedParam:
3481             nil), nil
3482     }
3483     //
3484     d, ok := o.(Dict)
3485     if !ok {
3486         // o, err := o.(IndirectRef)
3487         if !ok {
3488             return nil, errors.Errorf("PFFilterPipeline corrupt Dict: %v\n", o)
3489         }
3490         //
3491         o, err = derefResourceObject(ctx, io.ObjectNumber.Value())
3492         if err != nil {
3493             return nil, err
3494         }
3495     }
3496 }
3497 //
3498 // w/o decoded parameter.
3499 //
3500 log.Printf("PFFilterPipeline: with decode param")
3501 return append(filterPipeline, PFFilterName: filterName, DecodedParam: d),
3502     nil
3503 }
3504 //
3505 // filter pipeline.
3506 //
3507 // Array of filternames
3508 filterArray, ok := o.(Array)
3509 if !ok {
3510     return nil, errors.Errorf("PFFilterPipeline: Expected FilterArray corrupt, %v",
3511         o, o.S)
3512 }
3513 //
3514 // Optional array of decode parameter dicts.
3515 var decodedParam Array

```

```

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 := decompressHeaderTableEntry(ctx, *entry.Table, objectNumber, entry)
3540         if err == nil {
3541             return entry, nil
3542         }
3543     }
3544     if entry.Object == nil {
3545         log.Bad.Printf("dereferenceObject: dereferencing object %d\n", objectNumber)
3546         o, err := ParseObject(ctx, entry.Offset, objectNumber, entry.Generation)
3547         if err == nil {
3548             return nil, errors.Wrap(err, "dereferenceObject: problem dereferencing object %d", objectNumber)
3549         }
3550     }
3551     if o == nil {
3552         return nil, errors.New("p4cpu: dereferenceObject: object is nil")
3553     }
3554     entry.Object = o
3555 }
3556 return entry.Object, nil
3557 }
3558
3559 func dereferenceInteger(ctx *Context, objectNumber int) (Integer, error) {
3560     o, err := dereferenceObject(ctx, objectNumber)
3561     if err == nil {
3562         return nil, err
3563     }
3564     i, ok := o.(Integer)
3565     if !ok {
3566         return nil, errors.New("p4cpu: dereferenceInteger: corrupt integer")
3567     }
3568 }

```

```

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

```

```

2020 // 2. Create a new object to hold the data
2021 case objRefStream:
2022     case Read: Binary.ToInt32 += w * Stream.Length
2023     case Write: Stream.Write
2024     case Read: Binary.ToInt32 += w * Stream.Length
2025     }
2026     }
2027     }
2028     }
2029     }
2030     }
2031     }
2032     }
2033     }
2034     }
2035     }
2036     }
2037     }
2038     }
2039     }
2040     }
2041     }
2042     }
2043     }
2044     }
2045     }
2046     }
2047     }
2048     }
2049     }
2050     }
2051     }
2052     }
2053     }
2054     }
2055     }
2056     }
2057     }
2058     }
2059     }
2060     }
2061     }
2062     }
2063     }
2064     }
2065     }
2066     }
2067     }
2068     }
2069     }
2070     }
2071     }
2072     }
2073     }
2074     }
2075     }
2076     }
2077     }
2078     }
2079     }
2080     }
2081     }
2082     }
2083     }
2084     }
2085     }
2086     }
2087     }
2088     }
2089     }
2090     }
2091     }
2092     }
2093     }
2094     }
2095     }
2096     }
2097     }
2098     }
2099     }
2100     }
2101     }
2102     }
2103     }
2104     }
2105     }
2106     }
2107     }
2108     }
2109     }
2110     }
2111     }
2112     }
2113     }
2114     }
2115     }
2116     }
2117     }
2118     }
2119     }
2120     }
2121     }
2122     }
2123     }
2124     }
2125     }
2126     }
2127     }
2128     }
2129     }
2130     }
2131     }
2132     }
2133     }
2134     }
2135     }
2136     }
2137     }
2138     }
2139     }
2140     }
2141     }
2142     }
2143     }
2144     }
2145     }
2146     }
2147     }
2148     }
2149     }
2150     }
2151     }
2152     }
2153     }
2154     }
2155     }
2156     }
2157     }
2158     }
2159     }
2160     }
2161     }
2162     }
2163     }
2164     }
2165     }
2166     }
2167     }
2168     }
2169     }
2170     }
2171     }
2172     }
2173     }
2174     }
2175     }
2176     }
2177     }
2178     }
2179     }
2180     }
2181     }
2182     }
2183     }
2184     }
2185     }
2186     }
2187     }
2188     }
2189     }
2190     }
2191     }
2192     }
2193     }
2194     }
2195     }
2196     }
2197     }
2198     }
2199     }
2200     }
2201     }
2202     }
2203     }
2204     }
2205     }
2206     }
2207     }
2208     }
2209     }
2210     }
2211     }
2212     }
2213     }
2214     }
2215     }
2216     }
2217     }
2218     }
2219     }
2220     }
2221     }
2222     }
2223     }
2224     }
2225     }
2226     }
2227     }
2228     }
2229     }
2230     }
2231     }
2232     }
2233     }
2234     }
2235     }
2236     }
2237     }
2238     }
2239     }
2240     }
2241     }
2242     }
2243     }
2244     }
2245     }
2246     }
2247     }
2248     }
2249     }
2250     }
2251     }
2252     }
2253     }
2254     }
2255     }
2256     }
2257     }
2258     }
2259     }
2260     }
2261     }
2262     }
2263     }
2264     }
2265     }
2266     }
2267     }
2268     }
2269     }
2270     }
2271     }
2272     }
2273     }
2274     }
2275     }
2276     }
2277     }
2278     }
2279     }
2280     }
2281     }
2282     }
2283     }
2284     }
2285     }
2286     }
2287     }
2288     }
2289     }
2290     }
2291     }
2292     }
2293     }
2294     }
2295     }
2296     }
2297     }
2298     }
2299     }
2300     }
2301     }
2302     }
2303     }
2304     }
2305     }
2306     }
2307     }
2308     }
2309     }
2310     }
2311     }
2312     }
2313     }
2314     }
2315     }
2316     }
2317     }
2318     }
2319     }
2320     }
2321     }
2322     }
2323     }
2324     }
2325     }
2326     }
2327     }
2328     }
2329     }
2330     }
2331     }
2332     }
2333     }
2334     }
2335     }
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     }
2
```

```

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

```

```

1452 // decodeParamArray, found = dict.Fmt.DecodeParamArray }
1453 if found {
1454     decodeParamArray, ok = decodeParamArray(Array)
1455     if !ok {
1456         return nil, errors.New("pdpicp: pdfFilterPipeline: expected decodeParamArray corrupt")
1457     }
1458 }
1459 // /xxx.PdfDict("decodeParamArray: ba1a", decodeParamArray)
1460 // filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArray, decodeParamArray)
1461 // log.Read.FilterPipeline("pdfFilterPipeline: end")
1462 return filterPipeline, err
1463 // func streamDictForPdfObject(c *Context, d Dict, objKey, streamIn int, streamOffset
1464 // int) (uint64, GoStreamDict, error) {
1465 //     streamLength, streamLengthOff = d.Length()
1466 //     if streamLength < 0 {
1467 //         return sd, errors.New("pdpicp: streamDictForPdfObject: stream object without streamLength")
1468 //     }
1469 //     filterPipeline, err = pdfFilterPipeline(c, d)
1470 //     if err == nil {
1471 //         return sd, err
1472 //     }
1473 //     streamOffset = offset
1474 // }
1475 // // We have a stream object
1476 // sd := NewStreamDict(d, streamOffset, streamLength, streamLengthOff, filterPipeline)
1477 // log.Read.Filter("streamDictForPdfObject: end, streamObject %d\n", objKey)
1478 return sd, nil
1479 // }
1480 // func dictCtx *Context, d1 Dict, objKey, err, endId, streamIn int) (d2 Dict, err
1481 // error) {
1482 //     if ctx.EndKey == nil {
1483 //         ctx.EndKey = decryptPdfDict(d1, objKey, ctx.EndKey, ctx.AES4Strings,
1484 //             ctx.AES4B)
1485 //         if err == nil {
1486 //             return nil, err
1487 //         }
1488 //     }
1489 //     if endId == 0 || (streamIn < 0 || streamIn == endId) {
1490 //         log.Read.Print("dict: end, %d\n", objKey)
1491 //         d2 = d1
1492 //     }
1493 // }

```

[illegible]

```

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(ctxs &ctxs) error {
137     // @todo
138     // Entry "streams" intentionally left out.
139     // No object stream collection validation necessary.
140 }
141
142 log.Read.PrintIn("decodeObjectStreams: begin")
143
144 // Get sorted slice of object numbers.
145 void keyList()
146 for k = range cts.Read.ObjectStreams {
147     keys = append(keys, k)
148 }
149 sort.Ints(keys)
150
151 for _ , objectNumber = range keys {
152     // @see ObjectReadyIndex.
153     entry = cts.ReadIndex.Table(objectNumber)
154     if entry == nil {
155         return errors.Errorf("decodeObjectStreams: missing entry for objectNumber %d",
156             objectNumber)
157     }
158     log.Read.PrintIn("decodeObjectStreams: parsing object stream for objectNumber %d",
159         objectNumber)
160 }
161 // Parse object stream from file.
162 o, err = ParseObjectStream(entry.Offset, objectNumber, entry.Generation)
163 if err != nil || o == nil {
164     return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
165 }
166 // Ensure streamObject
167 sd, ok = p.(StreamObject)
168 if !ok {
169     return errors.New("pdpcc: decodeObjectStreams: corrupt object stream")
170 }
171 // @todo
172 // Load decoded stream content to storageTable.
173 if err = loadDecodedStreamContent(ctxs, sd); err != nil {
174     return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing object stream %d",
175         objectNumber)
176 }
177 // Save decoded stream content to storageTable.
178 if err = saveDecodedStreamContent(ctxs, sd, objectNumber, entry.Generation,
179     true); err != nil {
180     return err
181 }
182 }

```

```

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

```

```

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 }
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.Dst) {
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("tname: %s\n user: %s\n id: %s\n", ctx.OwnerName, ctx.UserIDN)
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()) {
3574     log.Read.Print("LoadContent readContent: begin({vLen},{s})");
3575 }
3576 var err = nil
3577 // Return
3578 // Return saved decoded content.
3579 if sd.Raw == nil {
3580     log.Read.Print("LoadContent readContent: end, already in memory.")
3581     return sd.Raw, nil
3582 }
3583 // Read stream content encoded at stream with stream length.
3584 // Difference stream length if stream length is an indirect object.
3585 if sd.StreamLength == nil {
3586     if sd.StreamLength == nil {
3587         return nil, errors.New("pdfcpu: LoadContent readContent: missing streamlength")
3588     }
3589     // sd stream length from indirect object
3590     sd.StreamLength, err = IndirectObj(sd, *sd.StreamLengthObj)
3591     if err == nil {
3592         return nil, err
3593     }
3594 }
3595 log.Read.Print("LoadContent readContent: new indirect streamlength {sdu}, {sd.StreamLength}")
3596 //
3597 //
3598 //
3599 //
3600 //
3601 //
3602 //
3603 //
3604 //
3605 //
3606 //
3607 //
3608 //
3609 //
3610 //
3611 //
3612 //
3613 //
3614 //
3615 //
3616 //
3617 //
3618 //
3619 //
3620 //
3621 //
3622 //
3623 //
3624 //
3625 //
3626 //
3627 //
3628 //
3629 //
3630 //
3631 //
3632 //
3633 //
3634 //
3635 //
3636 //
3637 //
3638 //
3639 //
3640 //
3641 //
3642 //
3643 //
3644 //
3645 //
3646 //
3647 //
3648 //
3649 //
3650 //
3651 //
3652 //
3653 //
3654 //
3655 //
3656 //
3657 //
3658 //
3659 //
3660 //
3661 //
3662 //
3663 //
3664 //
3665 //
3666 //
3667 //
3668 //
3669 //
3670 //
3671 //
3672 //
3673 //
3674 //
3675 //
3676 //
3677 //
3678 //
3679 //
3680 //
3681 //
3682 //
3683 //
3684 //
3685 //
3686 //
3687 //
3688 //
3689 //
3690 //
3691 //
3692 //
3693 //
3694 //
3695 //
3696 //
3697 //
3698 //
3699 //
3700 //
3701 //
3702 //
3703 //
3704 //
3705 //
3706 //
3707 //
3708 //
3709 //
3710 //
3711 //
3712 //
3713 //
3714 //
3715 //
3716 //
3717 //
3718 //
3719 //
3720 //
3721 //
3722 //
3723 //
3724 //
3725 //
3726 //
3727 //
3728 //
3729 //
3730 //
3731 //
3732 //
3733 //
3734 //
3735 //
3736 //
3737 //
3738 //
3739 //
3740 //
3741 //
3742 //
3743 //
3744 //
3745 //
3746 //
3747 //
3748 //
3749 //
3750 //
3751 //
3752 //
3753 //
3754 //
3755 //
3756 //
3757 //
3758 //
3759 //
3760 //
3761 //
3762 //
3763 //
3764 //
3765 //
3766 //
3767 //
3768 //
3769 //
3770 //
3771 //
3772 //
3773 //
3774 //
3775 //
3776 //
3777 //
3778 //
3779 //
3780 //
3781 //
3782 //
3783 //
3784 //
3785 //
3786 //
3787 //
3788 //
3789 //
3790 //
3791 //
3792 //
3793 //
3794 //
3795 //
3796 //
3797 //
3798 //
3799 //
3800 //
3801 //
3802 //
3803 //
3804 //
3805 //
3806 //
3807 //
3808 //
3809 //
3810 //
3811 //
3812 //
3813 //
3814 //
3815 //
3816 //
3817 //
3818 //
3819 //
3820 //
3821 //
3822 //
3823 //
3824 //
3825 //
3826 //
3827 //
3828 //
3829 //
3830 //
3831 //
3832 //
3833 //
3834 //
3835 //
3836 //
3837 //
3838 //
3839 //
3840 //
3841 //
3842 //
3843 //
3844 //
3845 //
3846 //
3847 //
3848 //
3849 //
3850 //
3851 //
3852 //
3853 //
3854 //
3855 //
3856 //
3857 //
3858 //
3859 //
3860 //
3861 //
3862 //
3863 //
3864 //
3865 //
3866 //
3867 //
3868 //
3869 //
3870 //
3871 //
3872 //
3873 //
3874 //
3875 //
3876 //
3877 //
3878 //
3879 //
3880 //
3881 //
3882 //
3883 //
3884 //
3885 //
3886 //
3887 //
3888 //
3889 //
3890 //
3891 //
3892 //
3893 //
3894 //
3895 //
3896 //
3897 //
3898 //
3899 //
3900 //
3901 //
3902 //
3903 //
3904 //
3905 //
3906 //
3907 //
3908 //
3909 //
3910 //
3911 //
3912 //
3913 //
3914 //
3915 //
3916 //
3917 //
3918 //
3919 //
3920 //
3921 //
3922 //
3923 //
3924 //
3925 //
3926 //
3927 //
3928 //
3929 //
3930 //
3931 //
3932 //
3933 //
3934 //
3935 //
3936 //
3937 //
3938 //
3939 //
3940 //
3941 //
3942 //
3943 //
3944 //
3945 //
3946 //
3947 //
3948 //
3949 //
3950 //
3951 //
3952 //
3953 //
3954 //
3955 //
3956 //
3957 //
3958 //
3959 //
3960 //
3961 //
3962 //
3963 //
3964 //
3965 //
3966 //
3967 //
3968 //
3969 //
3970 //
3971 //
3972 //
3973 //
3974 //
3975 //
3976 //
3977 //
3978 //
3979 //
3980 //
3981 //
3982 //
3983 //
3984 //
3985 //
3986 //
3987 //
3988 //
3989 //
3990 //
3991 //
3992 //
3993 //
3994 //
3995 //
3996 //
3997 //
3998 //
3999 //
4000 //
4001 //
4002 //
4003 //
4004 //
4005 //
4006 //
4007 //
4008 //
4009 //
4010 //
4011 //
4012 //
4013 //
4014 //
4015 //
4016 //
4017 //
4018 //
4019 //
4020 //
4021 //
4022 //
4023 //
4024 //
4025 //
4026 //
4027 //
4028 //
4029 //
4030 //
4031 //
4032 //
4033 //
4034 //
4035 //
4036 //
4037 //
4038 //
4039 //
4040 //
4041 //
4042 //
4043 //
4044 //
4045 //
4046 //
4047 //
4048 //
4049 //
4050 //
4051 //
4052 //
4053 //
4054 //
4055 //
4056 //
4057 //
4058 //
4059 //
4060 //
4061 //
4062 //
4063 //
4064 //
4065 //
4066 //
4067 //
4068 //
4069 //
4070 //
4071 //
4072 //
4073 //
4074 //
4075 //
4076 //
4077 //
4078 //
4079 //
4080 //
4081 //
4082 //
4083 //
4084 //
4085 //
4086 //
4087 //
4088 //
4089 //
4090 //
4091 //
4092 //
4093 //
4094 //
4095 //
4096 //
4097 //
4098 //
4099 //
4100 //
4101 //
4102 //
4103 //
4104 //
4105 //
4106 //
4107 //
4108 //
4109 //
4110 //
4111 //
4112 //
4113 //
4114 //
4115 //
4116 //
4117 //
4118 //
4119 //
4120 //
4121 //
4122 //
4123 //
4124 //
4125 //
4126 //
4127 //
4128 //
4129 //
4130 //
4131 //
4132 //
4133 //
4134 //
4135 //
4136 //
4137 //
4138 //
4139 //
4140 //
4141 //
4142 //
4143 //
4144 //
4145 //
4146 //
4147 //
4148 //
4149 //
4150 //
4151 //
4152 //
4153 //
4154 //
4155 //
4156 //
4157 //
4158 //
4159 //
4160 //
4161 //
4162 //
4163 //
4164 //
4165 //
4166 //
4167 //
4168 //
4169 //
4170 //
4171 //
4172 //
4173 //
4174 //
4175 //
4176 //
4177 //
4178 //
4179 //
4180 //
4181 //
4182 //
4183 //
4184 //
4185 //
4186 //
4187 //
4188 //
4189 //
4190 //
4191 //
4192 //
4193 //
4194 //
4195 //
4196 //
4197 //
4198 //
4199 //
4200 //
4201 //
4202 //
4203 //
4204 //
4205 //
4206 //
4207 //
4208 //
4209 //
4210 //
4211 //
4212 //
4213 //
4214 //
4215 //
4216 //
4217 //
421
```

```

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.Println("decodeObjectStream: decoding object stream SvId",
1980 objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict dictionary.
1984 // If err = readObjectStreamDict(svd) err == nil {
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding
1986 object stream SvId", objectStream)
1987 }
1988
1989 // If svd.ObjectArray == nil
1990     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1991 }
1992
1993 log.Read.Println("decodeObjectStream: decoded object stream SvId",
1994 objectStream)
1995
1996 // Save object stream dict to vfileEntryDict.
1997 entry.Object = svd
1998
1999 log.Read.Println("decodeObjectStream: end")
2000
2001 return nil
2002 }
2003
2004 func handleLinearizationPanic(dict *Content, obj Object, objStr int) error {
2005     // Handle linearized
2006     // // linearization dict already processed.
2007     return nil
2008 }
2009
2010 // handle Linearization panic.
2011 // If dict == nil || objStr < 0 || !isLinearizationPanic(dict) {
2012     handleLinearization = true
2013     dict.LinearizationPanic(objStr) + true
2014     log.Read.Println("handleLinearizationPanic: identified LinearizationObj
2015 SvId", objStr)
2016
2017     a := dict.Entries[objStr]
2018 }

```

```

2020:
2021: func processArrayByCounts(x:Iterable, xObjTable: XObjTable, a Array) {
2022:   for _ in range a {
2023:     switch o in a {Type} {
2024:     case IndirectType:
2025:       entry, ok = xObjTable.findTableEntryForIndirect(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: XObjTable, o Object) {
2038:   switch o in o {Type} {
2039:   case DirectType:
2040:     processIndirectCounts(xObjTable, o)
2041:   case StringType:
2042:     processIndirectCounts(xObjTable, o.Dict)
2043:   case Array:
2044:     processArrayByCounts(xObjTable, o)
2045:   }
2046: }
2047:
2048: // Performance notes: this function iterates over all objects from object stream.
2049: func deduplicateRefCounts(cxtx: Context) error {
2050:   log.Red.Println("deduplicateRefCounts: begin")
2051:   xObjTable = cxtx.XObjTable
2052:   // Get sorted list of object numbers.
2053:   // This step sorting for performance gain.
2054:   var keys List
2055:   for k in xObjTable.Table {
2056:     keys = append(keys, k)
2057:   }
2058:   sort.Ints(keys)
2059:
2060:   for _ ,objNr = range keys {
2061:     err = deduplicateRefCounts(cxtx, objNr)
2062:     if err != nil {
2063:       return err
2064:     }
2065:   }
2066:
2067:   for _ ,objNr = range keys {
2068:     entry = xObjTable.Table[objNr]
2069:     if entry.ref != entry.compressed {
2070:       continue
2071:     }
2072:     processRefCounts(xObjTable, entry.obj)
2073:   }
2074: }

```

```

2530 //
2531 if err == nil {
2532     return err
2533 }
2534 //
2535 // If the owner password does not match we generally move on if the user password
2536 // errors.
2537 // Unless we need to limit on a user's owner password due to the specific
2538 // amount in use password.
2539 if tok != newPasswordHandler(passwordCtx.Cmd) {
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 // If it is sufficient for moving on. A password change is an exception since it
2546 // is not.
2547 if ok := newPasswordHandler(passwordCtx.Cmd) {
2548     return nil
2549 }
2550 ok, err = validatePermissions(ctx)
2551 if err == nil {
2552     return err
2553 }
2554 //
2555 if ok {
2556     return errors.New("password: corrupted permissions after opw ok")
2557 }
2558 //
2559 return nil
2560 //
2561 // Validate the user password ok, document open password.
2562 ok, err = validatePermissions(ctx)
2563 if err == nil {
2564     return err
2565 }
2566 //
2567 if ok {
2568     return errors.New("password: please provide the correct password")
2569 }
2570 //
2571 //fmt.Printf("opw ok: %d\n", ok)
2572 //
2573 return handlePermissions(ctx)
2574 //
2575 func checkForEncryption(c *Context) error {
2576     //
2577     ic := ctx.Encrypt
2578     //
2579     if ic == nil {
2580         //
2581         // This file is not encrypted.
2582         return handleEncryptionFailed(ic)
2583     }
2584     //
2585     // This file is encrypted.
2586     log.Read.Printf("Encryption: %v\n", ic)
2587     //
2588     if ctx.Cmd == ENCRYPT {
2589         //
2590         // We want to encrypt this file.
2591         return errors.New("password: This file is already encrypted")
2592     }
2593     //
2594     // Difference encrypted.
2595 }

```