

```

34 2 Copyright 2018 The pdfcpu authors.
35
36 Licensed under the Apache License, Version 2.0 (the "License");
37 you may not use this file except in compliance with the License.
38 You may obtain a copy of the License at
39
40     http://www.apache.org/licenses/LICENSE-2.0
41
42 Unless required by applicable law or agreed to in writing, software
43 distributed under the License is distributed on an "AS IS" BASIS,
44 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
45 See the License for the specific language governing permissions and
46 limitations under the License.
47
48 package pdfcpu
49
50 import (
51     "bytes"
52     "io"
53     "log"
54     "strings"
55     "github.com/pdfcpu/pdfcpu/pkg/fontset"
56     "github.com/pdfcpu/pdfcpu/pkg/output"
57     "github.com/pdfcpu/pdfcpu/pkg/reader"
58     "github.com/pdfcpu/pdfcpu/pkg/transform"
59 )
60
61 // Read reads a PDF file and builds an internal structure holding its cross
62 // reference table and the objects.
63 func Read(filePath string, conf *Configuration) (*Context, error) {
64     log.Infof("Reading %s (%v)", filePath, conf)
65
66     f, err := os.Open(filePath)
67     if err != nil {
68         return nil, errors.Wrap(err, "can't open PDF", filePath)
69     }
70
71     defer func() {
72         f.Close()
73     }()
74
75     reader := Read(f, conf)
76
77     // Read takes a reader and generates a Context,
78     // which is a representation containing a cross reference table.
79     return reader, conf.Configuration()(*Context, error) {

```

```

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 //

```

[illegible]

```

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/eng.
998 // endd)

```

```

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 }

```

[illegible]

```

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 = extractObjectTailorTrailerStreamObjectStreamIdContent, dx, dx:sha1
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["vtableEntry"]
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, stream string, streamoff int) bool {
150     //Log-Debug-Print("isStreamEnd: stream: ", stream, "offset: ", 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("keyWords=string(HeaderDict)+NDICT: end: %s", ok)
1578
1579     return ok
1580 }
1581
1582 func buildFilterPipeline(ctx *Context, filterArray, decodeParamsArr Array, decodeParams
1583 *dict) (*Pfilter, error) {
1584     var filterPipeline []Pfilter
1585
1586     for i, f := range filterArray {
1587
1588         filterName, ok := f.(Name)
1589         if !ok {
1590             corrupt := true
1591             return nil, errors.New("pfilter: buildFilterPipeline: filterArray elements
1592 corrupt")
1593         }
1594         if decodeParams == nil || decodeParamsArr[i] == nil {
1595             filterPipeline = append(filterPipeline, PFilter{Name:
1596 filterName, decodeParams: nil})
1597             continue
1598         }
1599         dict, ok := decodeParamsArr[i].(Dict)
1600         if !ok {
1601             corrupt := true
1602             return nil, errors.New("pfilter: buildFilterPipeline: dict: %s",
1603 dict)
1604         }
1605         d, err := dereferenceDict(dict, indirectObjectNumberValue())
1606         if err != nil {
1607             return nil, err
1608         }
1609         dict = d
1610     }
1611
1612     filterPipeline = append(filterPipeline, PFilter{Name: filterName.String(),
1613 decodeParams: dict})
1614 }
1615
1616 return filterPipeline, nil
1617 }
1618
1619 // Decode the filter pipeline associated with this stream dict:
1620 func pfilterPipeline(ctx *Context, dict dict) (*Pfilter, error) {
1621     log.Read.Printf("pfilterPipeline: begin")
1622
1623     var err error
1624
1625     o, found := dict.Find("Filter")
1626     if !found {
1627         // stream is not compressed
1628     }

```

[illegible]

```

1130 // Save the saveDecodedContentContent to ctx.content, id, saveStreams, objKey, goenv int,
1131 // err error()
1132
1133 // Log.Read.Print("saveDecodedContentContent: begin decode\n"), decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if ctx.will nil on ctx.filterKey == nil {
1137     if ctx.filterPolicyName == 1 do do.FilterPolicyName(), Name == "Crypt" {
1138         return nil
1139     }
1140 }
1141
1142 // Special case: If the length of the encoded data is 0, we do not need to decode
1143 anything.
1144 if ctx.len(StdRaw) == 0 {
1145     StdContent = StdRaw
1146     return nil
1147 }
1148
1149 // Std gets created after StdStream parsing.
1150 // StdStreams are not encrypted.
1151 if ctx.will == StdRaw {
1152     StdRaw, err = decryptStream(StdRaw, objKey, goenv, ctx.KeyEncry, ctx.AESStreams,
1153         ctx.Ex)
1154     if err == nil {
1155         return err
1156     }
1157     StdRaw =
1158         len(StdRaw)
1159     StdStream.Length = 0
1160 }
1161
1162 // If decode
1163     return nil
1164 }
1165
1166 // Actual decoding of content stream.
1167 err = decodeStream()
1168 if err == filter.StreamSupportFilter {
1169     err = nil
1170 }
1171
1172 if err == nil {
1173     return err
1174 }
1175
1176 Log.Read.Print("saveDecodedContentContent: end")
1177
1178 return nil
1179
1180 // Decode compressed objTableEntry.
1181 func decodeCompressedObjTableEntry(
1182     obj: decompressedObjTableEntry, objTable *objTable, objStream int, entry
1183     *objStream, *objStream) error {
1184     Log.Read.Print("decompressObjTableEntry: compressed object id at %d\n",
1185         objStream, *objStream, entry.ObjStream, entry.ObjStream)
1186
1187     // Missing stream entry in reference object stream.
1188     objStreamTableEntry, obj = nil, nil, find entry.ObjStream)
1189     if obj {

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry m, objIdR")
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs length 2 obj+o objNR")
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", objNR)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offset64
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objNR)
2053             }
2054             offset64 := Int64(offset.Value())
2055             ctx.OffsetOverluminTable = offset64
2056         }
2057     }
2058     return nil
2059 }
2060
2061 func LoadBinaryStream(ctx *Context, s *StreamReader, objNR, genNr int) error {
2062     var err error
2063     if !
2064         // Load stream's content and store data into offsetable entry
2065         if err = LoadInfiniteStream(ctx, s, objNR, genNr, m); err != nil {
2066             return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d", objNR)
2067         }
2068         ctx.Read.BinarySize += s.GetSize()
2069         // Decode stream's content
2070         err = saveDecodedStreamContent(ctx, s, objNR, genNr, ctx.DecodedAllStreams)
2071         if err != nil {
2072             return err
2073         }
2074     }
2075     return nil
2076 }
2077
2078 func updateLinearizationDict(ctx *Context, o Object) {
2079     switch o := o.(type) {
2080     case StreamDict:
2081         ctx.AddBinarySize(s + s.GetSize())
2082     }
2083 }

```

```

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 < v14 {
2543         log.Info.Print("IdentifyRootVersion: PDP version is %s - will ignore root
2544             version '%s'",
2545             xRefTable.HeaderVersion, rootVersionStr)
2546     }
2547     log.Read.Print("IdentifyRootVersion: end")
2548     return nil
2549 }
2550 // Parse all Objects including stream content from file and save to the corresponding
2551 // headerTables.
2552 std::weak_ptr<Object> processObjectOfObjectStreamAndLinearizationDicts(
2553     std::weak_ptr<Object> obj, Context, const <Configuration> err) {
2554     log.Read.Print("ProcessObjectOfObjectStream: begin")
2555     xRefTable = ctx.xRefTable
2556     // Note for unencrypted files:
2557     // Mandatory provide users to open & display file.
2558     // Access may be restricted (Open access privileges).
2559     // Optionally provide comments in order to gain unrestricted access.
2560     if err == ObjectFormatError {
2561         return err
2562     }
2563 }

```

```

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

```

```

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

```

3520 // test: (ts:R)
3521 if err == nil {
3522     return nil, err
3523 }
3524 return StringLiteral(string(bb)), nil
3525 }
3526
3527 default:
3528     return o, nil
3529 }
3530 }
3531 }
3532
3533 func dereferenceObject(ctx *Context, objectNumber int) (Object, error) {
3534     entry, ok := cts.Find(objectNumber)
3535     if !ok {
3536         return nil, errors.New("p4cpu: dereferenceObject: unregistered object")
3537     }
3538     if entry.Compressed {
3539         err := 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, o = ObjectStreamId(entry.ObjectId, ObjectStreamId)
1580
1581 if !ok {
1582     return errors.Errorf("decompressRefTableEntry: problem dereferencing object stream %d, no object stream", entry.ObjectStreamId)
1583 }
1584
1585 //
1586 // Get index object from ObjectStreamId
1587 //
1588 o, err = sd.IndexObjectFromEntry(ObjectStreamId)
1589 if err != nil {
1590     return errors.Errorf("decompressRefTableEntry: problem dereferencing object stream %d", entry.ObjectStreamId)
1591 }
1592
1593 // Save object to theRefTableEntry.
1594
1595 g := &
1596     entry.Object = o
1597     entry.Compression = 0
1598     entry.Expression = false
1599
1600 //
1601 // Load object's decompressRefTableEntry, end, do &sd[0]:Vec&sd[0],
1602 // entry.ObjectStreamId, entry.ObjectStreamId, o)
1603
1604 return nil
1605
1606 //
1607 // Log interesting stream content.
1608 //
1609 func LogStreamContent(i int) {
1610     switch o := o.(type) {
1611     case StreamId:
1612         if o.Content == nil {
1613             log.Printf("logStream: no stream content")
1614         }
1615         if o.IsPageContent {
1616             //log.Printf("logStream: content %s", StreamId.Content)
1617         }
1618     case ObjectStreamId:
1619         if o.Content == nil {
1620             log.Printf("logStream: no object stream content")
1621         }
1622         if o.IsPageContent {
1623             log.Printf("logStream: object stream content %s", o.Content)
1624         }
1625         if o.IsPageEntry {
1626             log.Printf("logStream: no object stream &sd[0] array")
1627         }
1628         if o.IsPageEntry {
1629             log.Printf("logStream: object stream &sd[0] %s", o.IsPageEntry)
1630         }
1631     }
1632 }
1633
1634 //
1635 // Default:

```

```

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

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObject(object)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shellEntry entry assign a object either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObject(object)
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     if ver := differenceObject(object)
2139     if err != nil {
2140         return err
2141     }
2142
2143     log.Printf("DifferenceCatalogTable: end")
2144
2145     return nil
2146 }
2147
2148 func handleEncryptedFile(cts *Context) error {
2149     err :=
2150     if cts.Cmd == DECRYPT {
2151         if cts.Cmd == SETPERMISSIONS {
2152             return errors.New("pfcpu: this file is not encrypted")
2153         }
2154     }
2155     if cts.Cmd == DECRYPT {
2156         return nil
2157     }
2158     // Encrypt subcommand found.
2159     if cts.SubCmd == "e" {
2160         return errors.New("pfcpu: please provide owner password and optional user
2161         password")
2162     }
2163     return nil
2164 }
2165
2166 func idBytes(cts *Context) []byte { id []byte, err error {
2167     if cts.ID == nil {
2168         return nil, errors.New("pfcpu: missing ID entry")
2169     }
2170     N1, ok := cts.ID[0].(float64)
2171     if ok {
2172         id, err = n1.Bytes()
2173         if err != nil {
2174             return nil, err
2175         }
2176     }
2177 }

```

```

1452 // @param {Object} ctx - Context object
1453 if (found) {
1454   decodeParamArr, found = dict.Fall(UNCODEPARAM)
1455 }
1456 if (found) {
1457   decodeParamArr, ok = decodeParam.Array()
1458   if (ok) {
1459     return nil, errors.New("pdcip: pdfFilterPipeline: expected decodeParam
1460 array corrupt")
1461   }
1462 }
1463 // /var Printout("decodeParam: %v", decodeParam)
1464 // /var Printout("decodeParam: %v", decodeParam)
1465 // /var Printout("decodeParam: %v", decodeParam)
1466 filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParamArr,
1467 decodeParam)
1468 if (err != nil) {
1469   log.Read.Println("pdfFilterPipeline: err")
1470 }
1471 return filterPipeline, err
1472 }
1473 func streamDictForObj(c *Context, d Dict, objKey, streamIn int, streamFset
1474 *FileSet, objId int) (StreamDict, error) {
1475   streamLength, streamLengthF = d.Length()
1476   if (streamLength == 0) {
1477     return sd, errors.New("pdcip: streamDictForObj: stream object without
1478 streamFset")
1479   }
1480   filterPipeline, err = pdfFilterPipeline(c, d)
1481   if (err != nil) {
1482     return sd, err
1483   }
1484   streamOffset = offset
1485   // We have a stream object
1486   sd = NewStream(streamDict, streamOffset, streamLength, streamLengthF, filterPipeline
1487   log.Read.Println("streamDictForObj: end, streamObject %v", objId)
1488   return sd, nil
1489 }
1490 func dict(c *Context, d Dict, objKey, objKey, objId, streamIn int) (Dict, err
1491 if (c.IncKey(d, objKey) != nil) {
1492   return nil, errors.New("pdcip: dict: objKey, objKey, objId, streamIn: %v", objKey)
1493 }
1494 if (c.IncKey(d, objKey) != nil) {
1495   return nil, err
1496 }
1497 if (objId == 0 || (streamIn < 0 || streamIn > streamIn)) {
1498   log.Read.Println("dict: end, objId, objKey")
1499   d2 = d1
1500 }
1501 }
1502 }

```

```

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

```

```

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

```

[illegible]

```

2370 }
2371 | else {
2372   id, ok := ctx.ID().(StringLiteral)
2373   if !ok {
2374     return nil, error.New("pdpoc: ID must contain hex literals or string
2375       literals").(Error)
2376   }
2377   id, err = Unescape(id.Value())
2378   if err != nil {
2379     return nil, err
2380   }
2381 }
2382
2383 return id, nil
2384
2385 func needsOwnerAndNamespace(cmd CommandMode) bool {
2386   cmd == CHANGEOID || 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.id) {
2400     return errors.New("pdpoc: insufficient access permissions")
2401   }
2402   return nil
2403 }
2404
2405 func setupEncryptionKey(ctx *Context, d Dict) (err error) {
2406   ctx.t, err = supportGetEncryption(ctx, d)
2407   if err != nil {
2408     return err
2409   }
2410   ctx.t.ID, err = idbytes(cctx)
2411   if err != nil {
2412     return err
2413   }
2414   var ok bool
2415   //fmt.Printf("cctx: %s\n", cctx)
2416   // Validate the owner password aka .permissions/master.password
2417   ok, err = ValidationPassword(ctx)

```

[illegible]

```

3570 // Read stream content into the window stream content buffer size
3571 streamContent;
3572 // Load content to readContent context: Context, sd streamContent() [byte, error]
3573 load.ReadContent(readContentContext: Context, sd streamContent() [byte, error])
3574
3575 // Log ReadContent()
3576 log.Read.Print("LoadContentContext: begin(w/v/u/s)",
3577
3578 // Return
3579 // Return saved decoded content.
3580 if sd.Raw == nil {
3581     log.Read.Print("LoadContentContext: end, already in memory.")
3582     return sd.Raw, nil
3583 }
3584
3585 // Read stream content encoded at stream with stream length.
3586 // Difference stream length if stream length is an indirect object.
3587 if sd.StreamLength == nil {
3588     if sd.StreamLength == nil {
3589         return nil, errors.New("pdfcpu: LoadContentContext: missing
3590 streamLength")
3591 }
3592 // sd stream length from indirect object
3593 sd.StreamLength, err = IndirectObject(sd, sd.StreamLengthObj)
3594 if err == nil {
3595     return nil, err
3596 }
3597
3598 // Log Read.Print("LoadContentContext: end indirect streamLength",
3599 sd.StreamLength)
3600
3601 // Read
3602 // Read offset in sd streamContent
3603 rd, err = NewPositionalReaderAt(sd.Raw, 0)
3604 if err != nil {
3605     return nil, err
3606 }
3607
3608 // Log Read.Print("LoadContentContext: seeked to offset:", rd, 0)
3609
3610 // Buffer stream content.
3611 // Read content from stream
3612 readContent, err = readContentStream(rd, int64(sd.StreamLength))
3613 if err == nil {
3614     return nil, err
3615 }
3616
3617 // Log Read.Print("LoadContentContext: bufferLen()", len(readContent),
3618 len(sd.Raw))
3619
3620 // Stream content
3621 // Set stream content
3622 sd.Raw = readContent
3623
3624 // Log Read.Print("LoadContentContext: end: len(streamContent)",
3625 len(sd.Raw))
3626
3627 // Return decoded content
3628 return readContent, nil
3629 }
3630
3631 // Decode the raw encoded stream content and saves it to streamContent.Context.

```

```

1968 // Use the object stream directly for object stream reads.
1969 // If !sd, !isObject()
1970     return errors.New("pdcps: decodeObjectStream: corrupt object stream")
1971 }
1972
1973 // We have an object stream.
1974 // If !sd, err = objectStreamDict(svd)
1975 // If err == nil
1976     return errors.Wrap(err, "decodeObjectStream: problem dereferencing
1977 object stream svd", objectStream)
1978
1979 // Log Read, Print("decodeObjectStream: decoding object stream %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(cxt *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 == c == obj (dict) obj != d, it is linearizationPanicDict() {
2011         Log Read, linearized := true
2012         cxt.LinearizationPanicDict(obj) == true
2013         Log Read, Print("handleLinearizationPanicDict: identified LinearizationObj
2014 %d\n", obj)
2015
2016         a := d.ArrayEntry("pr")
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
271
```

```

2020:
2021: func processArrayByCounts(x:Iterable, xObjTable: aXObjTable, a Array) {
2022:   for _ in range a {
2023:     switch o in a.cType() {
2024:     case IndirectType:
2025:       entry, ok = xObjTable.findTableEntryIndirect(o)
2026:       if !ok {
2027:         entry, ok = Count++
2028:       }
2029:       case Count:
2030:         processByCounts(xObjTable, o)
2031:       case Array:
2032:         processByCounts(xObjTable, o)
2033:       }
2034:     }
2035:   }
2036: }
2037:
2038: func processByCounts(xObjTable: aXObjTable, o Object) {
2039:   switch o in o.cType() {
2040:   case Dict:
2041:     processDictByCounts(xObjTable, o)
2042:   case StreamDict:
2043:     processDictByCounts(xObjTable, o.Dict)
2044:   case Array:
2045:     processArrayByCounts(xObjTable, o)
2046:   }
2047: }
2048:
2049: // Performance analysis: counts including unprocessed objects from object streams.
2050: func debugPrintCounts(c: a Context) error {
2051:   log.Red.Println("counts: begin")
2052:   xObjTable = ctx.XObjTable
2053:   // Do sorted counts for object numbers.
2054:   // This step sorting for performance gain.
2055:   var keys List
2056:   for k in xObjTable.Table {
2057:     keys = append(keys, k)
2058:   }
2059:   sort.Ints(keys)
2060:
2061:   for _ , objNr = range keys {
2062:     err = deferencedict(c,ctx,objNr)
2063:     if err != nil {
2064:       return err
2065:     }
2066:   }
2067:
2068:   for _ , objNr = range keys {
2069:     entry = xObjTable.Table[objNr]
2070:     if entry.ref != nil {
2071:       continue
2072:     }
2073:     processByCounts(xObjTable, entry.obj)
2074:   }
2075: }

```

```

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

```