

[illegible]

```

2420 offset_err = stream.xAtoi(fields[0], 16, 64)
2421 if err == nil {
2422     return err
2423 }
2424
2425 generation_err = stream.Atoi(fields[1])
2426 if err == nil {
2427     return err
2428 }
2429
2430 entryType = fields[2]
2431 if entryType != "f" || offset_err != "0" {
2432     return errors.New("offset parseXrefTableEntry: corrupt xref subsection
2433 entry")
2434 }
2435
2436 var xrefTableEntry xrefTableEntry
2437
2438 if entryType == "a" {
2439     // no use object
2440
2441     log.Read.Print("parseXrefTableEntry: Object %d is in use at offset%0d,
2442 generation%0d", objectNumber, offset, generation)
2443
2444     if offset == 0 {
2445         // Info: Info("parseXrefTableEntry: Skip entry for in use object %d
2446 with offset %0d", objectNumber)
2447         return nil
2448     }
2449 }
2450
2451 xrefTableEntry.a =
2452     xrefTableEntry{
2453         xrefTableEntry{
2454             Free:      false,
2455             Offset:     0xffff,
2456             Generation: 0xffffffff
2457         }
2458     } else {
2459         // free object
2460
2461         log.Read.Print("parseXrefTableEntry: Object %d is unused, next free is
2462 object%0d", objectNumber, offset, generation)
2463
2464         xrefTableEntry =
2465             xrefTableEntry{
2466                 Free:      true,
2467                 Offset:     offset,
2468                 Generation: 0xffffffff
2469             }
2470     }
2471
2472     log.Read.Print("parseXrefTableEntry: Insert new xrefTable entry for Object %d\n",
2473 objectNumber)
2474
2475     xrefTable[objectNumber] = xrefTableEntry
2476
2477     log.Read.Print("parseXrefTableEntry: end")
2478
2479     return nil
2480 }

```

[illegible]

```

657         if (rootEntry == null)
658             return errors.New("path: %s: parameterInfo: missing entry '%s'()",
659                               xRefTable.Key(), rootEntry.Key());
660
661         xRefTable.Root = rootEntry;
662         log.Root.Print("parameterInfo: Root object: %s", xRefTable.Root);
663
664     }
665
666     if (parameterInfo == nil) {
667         infoEntry := d.indirectEntry("Info")
668         if infoEntry == nil {
669             if infoEntryKey == "Info" {
670                 log.Root.Print("parameterInfo: Info object: %s", xRefTable.Info)
671             }
672         }
673     }
674
675     if xRefTable.ID == nil {
676         idEntry := d.indirectEntry("ID")
677         if idEntry == nil {
678             log.Root.Print("parameterInfo: ID object: %s", xRefTable.ID)
679         } else if xRefTable.ID.Key() == nil {
680             return errors.New("path: %s: parameterInfo: missing entry '%s'()",
681                               xRefTable.Key(), "ID")
682         }
683     }
684
685     log.Root.Print("parameterInfo end")
686
687     return nil
688 }
689
690 func parameterTable(trailerDict Dict, ctx *Context) (xTable, error) {
691     xTable := xTable{}
692     log.Root.Print("parameterTable begin")
693
694     xRefTable := ctx.xRefTable
695
696     err := parameterInfo(&trailerDict, xRefTable)
697     if err == nil {
698         return nil, err
699     }
700
701     if err := trailerDict.ArraysEntry("AdditionalStreams") are == nil {
702         log.Root.Print("parameterInfo: found AdditionalStreams '%s', arr",
703                       ctx.ArraysEntry("AdditionalStreams").Key())
704         for i, value := range arr {
705             if !isObject(value) || !isObject(value) || !is {
706                 a = append(a, index)
707             }
708         }
709         xRefTable.AdditionalStreams = ba
710     }
711
712     offset := trailerDict.Pre()
713     if offset == nil {
714         log.Root.Print("parameterInfo: previous xref table section offset '%s',",
715                       "offset")
716     }
717
718     offsetStr := trailerDict.IndirectEntry("Offset")

```

```

800 // data
801 } data {
802     log_Read.Print("line len: %s\n", line);
803 }
804
805 trailerString := scanTrailer(s, trailerString)
806 // if err == nil {
807     return nil, err
808 // }
809
810 log_Read.Print("processTrailer: trailerString: (len:%d) %s\n",
811     len(trailerString), trailerString)
812
813 // a, err := parseObject(&trailerString)
814 // if err == nil {
815     return nil, err
816 // }
817
818 trailerDict, ok := a.(dict)
819 // if ok {
820     return nil, errors.New("object: processTrailer: corrupt trailer dict")
821 // }
822
823 log_Read.Print("processTrailer: trailerDict(%s)\n", trailerDict)
824
825 return processTrailerDict(trailerDict, ctx)
826 }
827
828 // =====
829
830 // =====
831
832 // =====
833
834 // =====
835
836 // =====
837
838 // =====
839
840 // =====
841
842 // =====
843
844 // =====
845
846 // =====
847
848 // =====
849
850 // =====
851
852 // =====
853
854 // =====
855
856 // =====
857
858 // =====
859
860 // =====
861
862 // =====
863
864 // =====
865
866 // =====
867
868 // =====
869
870 // =====
871
872 // =====
873
874 // =====
875
876 // =====
877
878 // =====
879
880 // =====
881
882 // =====
883
884 // =====
885
886 // =====
887
888 // =====
889
890 // =====
891
892 // =====
893
894 // =====
895
896 // =====
897
898 // =====
899
900 // =====
901
902 // =====
903
904 // =====
905
906 // =====
907
908 // =====
909
910 // =====
911
912 // =====
913
914 // =====
915
916 // =====
917
918 // =====
919
920 // =====
921
922 // =====
923
924 // =====
925
926 // =====
927
928 // =====
929
930 // =====
931
932 // =====
933
934 // =====
935
936 // =====
937
938 // =====
939
940 // =====
941
942 // =====
943
944 // =====
945
946 // =====
947
948 // =====
949
950 // =====
951
952 // =====
953
954 // =====
955
956 // =====
957
958 // =====
959
960 // =====
961
962 // =====
963
964 // =====
965
966 // =====
967
968 // =====
969
970 // =====
971
972 // =====
973
974 // =====
975
976 // =====
977
978 // =====
979
980 // =====
981
982 // =====
983
984 // =====
985
986 // =====
987
988 // =====
989
990 // =====
991
992 // =====
993
994 // =====
995
996 // =====
997
998 // =====
999
1000 // =====
1001
1002 // =====
1003
1004 // =====
1005
1006 // =====
1007
1008 // =====
1009
1010 // =====
1011
1012 // =====
1013
1014 // =====
1015
1016 // =====
1017
1018 // =====
1019
1020 // =====
1021
1022 // =====
1023
1024 // =====
1025
1026 // =====
1027
1028 // =====
1029
1030 // =====
1031
1032 // =====
1033
1034 // =====
1035
1036 // =====
1037
1038 // =====
1039
1040 // =====
1041
1042 // =====
1043
1044 // =====
1045
1046 // =====
1047
1048 // =====
1049
1050 // =====
1051
1052 // =====
1053
1054 // =====
1055
1056 // =====
1057
1058 // =====
1059
1060 // =====
1061
1062 // =====
1063
1064 // =====
1065
1066 // =====
1067
1068 // =====
1069
1070 // =====
1071
1072 // =====
1073
1074 // =====
1075
1076 // =====
1077
1078 // =====
1079
1080 // =====
1081
1082 // =====
1083
1084 // =====
1085
1086 // =====
1087
1088 // =====
1089
1090 // =====
1091
1092 // =====
1093
1094 // =====
1095
1096 // =====
1097
1098 // =====
1099
1100 // =====
1101
1102 // =====
1103
1104 // =====
1105
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 // =====

```

```

1337 rs = ctx.Read(rs
1338
1339 hr, endCount, err = ReaderVersion(rs)
1400 if err == nil {
1401     return err
1402 }
1403
1404 ctx.readHeaderVersion = hr
1405 ctx.readEndCount = endCount
1406
1407 for offset := nil {
1408
1409     rd, err = newPositionedReader(rs, offset)
1410     if err == nil {
1411         return err
1412     }
1413
1414     s := bufio.NewScanner(rd)
1415     s.Split(scanLines)
1416
1417     line, err = s.Scan()
1418     if err == nil {
1419         return err
1420     }
1421
1422     log.Read.Printf("line: %s\n", line)
1423
1424     if strings.TrimSpace(line) == "reset" {
1425         log.Read.Printf("builderTableStarting: find reset stream")
1426         if offset, err = parseResetSection(s, ctx); err == nil {
1427             return err
1428         }
1429     } else {
1430         log.Read.Printf("builderTableStarting: find reset stream")
1431
1432         ctx.readHeaderStreams = true
1433         rd, err = newFullStreamReader(rs, offset)
1434         if err == nil {
1435             return err
1436         }
1437         if offset, err = parseResetStream(r, offset, ctx), err == nil {
1438             log.Read.Printf("builderTableStarting after %s", err)
1439             // for file format single reset section.
1440             return bypassResetSection(s)
1441         }
1442     }
1443 }
1444
1445 log.Read.Printf("builderTableStarting: end")
1446
1447 return nil
1448
1449 // Generate the cross-reference table for this PDF file.
1450 // Data object, object, indirect objects, reference, etc. "A 0 0"
1451 // can be "ref" or indirect object, reference, etc. "A 0 0"
1452 // can be indirect object, reference, etc. as long as there is a defined previous cross section
1453 // and build up the cross table along the way.
1454
1455 func reinterTable(c *Context) (err error) {

```

```

550 // =====
551 log.Read.Print("Read begin!")
552
553 ctx, err := NewContexts, conf)
554 if err != nil {
555     return nil, err
556 }
557
558 if ctx.ReadOnly {
559     log.Info.Print("PDF Version 1.3 conforming reader")
560 } else {
561     log.Info.Print("PDF Version 1.4 conforming reader - no object streams &
562         references allowed")
563 }
564
565 // Populate shuffable
566 if err = readShuffleable(ctx); err != nil {
567     return nil, errors.New("Read: shuffleable failed")
568 }
569
570 // Make all objects explicitly available (load into memory) in corresponding
571 // shuffable entries.
572 // Also makes any involved object streams.
573 if err = dereferenceObjects(ctx, conf); err != nil {
574     return nil, err
575 }
576
577 // Some references write an invariant size into trailer.
578 // cctx.ShuffleableSize <= ctx.ShuffleableTable)
579 // cctx.ShuffleableSize <= len(ctx.ShuffleableTable)
580
581 log.Read.Print("Read end")
582
583 return ctx, nil
584
585 // =====
586
587 // ScanLine is a public function for a Scanner that returns each line of
588 // text, stripped of any trailing end-of-line marker. The returned line may
589 // be empty, may contain carriage returns, or may contain carriage returns followed
590 // by one newline or one carriage return or one newline.
591 // If the scanner is in a state where it will not return even if it has no error
592 // func scanLine(data []byte, startIdx int) (string, bool) {
593 //     if !isEOF {
594 //         if !isEOL {
595 //             return "", nil, nil
596 //         }
597 //         if !isEOL {
598 //             return "", nil, nil
599 //         }
600 //         if !isEOL {
601 //             return "", nil, nil
602 //         }
603 //         if !isEOL {
604 //             return "", nil, nil
605 //         }
606 //         if !isEOL {
607 //             return "", nil, nil
608 //         }
609 //         if !isEOL {
610 //             return "", nil, nil
611 //         }
612 //         if !isEOL {
613 //             return "", nil, nil
614 //         }
615 //         if !isEOL {
616 //             return "", nil, nil
617 //         }
618 //         if !isEOL {
619 //             return "", nil, nil
620 //         }
621 //         if !isEOL {
622 //             return "", nil, nil
623 //         }
624 //         if !isEOL {
625 //             return "", nil, nil
626 //         }
627 //         if !isEOL {
628 //             return "", nil, nil
629 //         }
630 //         if !isEOL {
631 //             return "", nil, nil
632 //         }
633 //         if !isEOL {
634 //             return "", nil, nil
635 //         }
636 //         if !isEOL {
637 //             return "", nil, nil
638 //         }
639 //         if !isEOL {
640 //             return "", nil, nil
641 //         }
642 //         if !isEOL {
643 //             return "", nil, nil
644 //         }
645 //         if !isEOL {
646 //             return "", nil, nil
647 //         }
648 //         if !isEOL {
649 //             return "", nil, nil
650 //         }
651 //         if !isEOL {
652 //             return "", nil, nil
653 //         }
654 //         if !isEOL {
655 //             return "", nil, nil
656 //         }
657 //         if !isEOL {
658 //             return "", nil, nil
659 //         }
660 //         if !isEOL {
661 //             return "", nil, nil
662 //         }
663 //         if !isEOL {
664 //             return "", nil, nil
665 //         }
666 //         if !isEOL {
667 //             return "", nil, nil
668 //         }
669 //         if !isEOL {
670 //             return "", nil, nil
671 //         }
672 //         if !isEOL {
673 //             return "", nil, nil
674 //         }
675 //         if !isEOL {
676 //             return "", nil, nil
677 //         }
678 //         if !isEOL {
679 //             return "", nil, nil
680 //         }
681 //         if !isEOL {
682 //             return "", nil, nil
683 //         }
684 //         if !isEOL {
685 //             return "", nil, nil
686 //         }
687 //         if !isEOL {
688 //             return "", nil, nil
689 //         }
690 //         if !isEOL {
691 //             return "", nil, nil
692 //         }
693 //         if !isEOL {
694 //             return "", nil, nil
695 //         }
696 //         if !isEOL {
697 //             return "", nil, nil
698 //         }
699 //         if !isEOL {
700 //             return "", nil, nil
701 //         }
702 //         if !isEOL {
703 //             return "", nil, nil
704 //         }
705 //         if !isEOL {
706 //             return "", nil, nil
707 //         }
708 //         if !isEOL {
709 //             return "", nil, nil
710 //         }
711 //         if !isEOL {
712 //             return "", nil, nil
713 //         }
714 //         if !isEOL {
715 //             return "", nil, nil
716 //         }
717 //         if !isEOL {
718 //             return "", nil, nil
719 //         }
720 //         if !isEOL {
721 //             return "", nil, nil
722 //         }
723 //         if !isEOL {
724 //             return "", nil, nil
725 //         }
726 //         if !isEOL {
727 //             return "", nil, nil
728 //         }
729 //         if !isEOL {
730 //             return "", nil, nil
731 //         }
732 //         if !isEOL {
733 //             return "", nil, nil
734 //         }
735 //         if !isEOL {
736 //             return "", nil, nil
737 //         }
738 //         if !isEOL {
739 //             return "", nil, nil
740 //         }
741 //         if !isEOL {
742 //             return "", nil, nil
743 //         }
744 //         if !isEOL {
745 //             return "", nil, nil
746 //         }
747 //         if !isEOL {
748 //             return "", nil, nil
749 //         }
750 //         if !isEOL {
751 //             return "", nil, nil
752 //         }
753 //         if !isEOL {
754 //             return "", nil, nil
755 //         }
756 //         if !isEOL {
757 //             return "", nil, nil
758 //         }
759 //         if !isEOL {
760 //             return "", nil, nil
761 //         }
762 //         if !isEOL {
763 //             return "", nil, nil
764 //         }
765 //         if !isEOL {
766 //             return "", nil, nil
767 //         }
768 //         if !isEOL {
769 //             return "", nil, nil
770 //         }
771 //         if !isEOL {
772 //             return "", nil, nil
773 //         }
774 //         if !isEOL {
775 //             return "", nil, nil
776 //         }
777 //         if !isEOL {
778 //             return "", nil, nil
779 //         }
780 //         if !isEOL {
781 //             return "", nil, nil
782 //         }
783 //         if !isEOL {
784 //             return "", nil, nil
785 //         }
786 //         if !isEOL {
787 //             return "", nil, nil
788 //         }
789 //         if !isEOL {
790 //             return "", nil, nil
791 //         }
792 //         if !isEOL {
793 //             return "", nil, nil
794 //         }
795 //         if !isEOL {
796 //             return "", nil, nil
797 //         }
798 //         if !isEOL {
799 //             return "", nil, nil
800 //         }
801 //         if !isEOL {
802 //             return "", nil, nil
803 //         }
804 //         if !isEOL {
805 //             return "", nil, nil
806 //         }
807 //         if !isEOL {
808 //             return "", nil, nil
809 //         }
810 //         if !isEOL {
811 //             return "", nil, nil
812 //         }
813 //         if !isEOL {
814 //             return "", nil, nil
815 //         }
816 //         if !isEOL {
817 //             return "", nil, nil
818 //         }
819 //         if !isEOL {
820 //             return "", nil, nil
821 //         }
822 //         if !isEOL {
823 //             return "", nil, nil
824 //         }
825 //         if !isEOL {
826 //             return "", nil, nil
827 //         }
828 //         if !isEOL {
829 //             return "", nil, nil
830 //         }
831 //         if !isEOL {
832 //             return "", nil, nil
833 //         }
834 //         if !isEOL {
835 //             return "", nil, nil
836 //         }
837 //         if !isEOL {
838 //             return "", nil, nil
839 //         }
840 //         if !isEOL {
841 //             return "", nil, nil
842 //         }
843 //         if !isEOL {
844 //             return "", nil, nil
845 //         }
846 //         if !isEOL {
847 //             return "", nil, nil
848 //         }
849 //         if !isEOL {
850 //             return "", nil, nil
851 //         }
852 //         if !isEOL {
853 //             return "", nil, nil
854 //         }
855 //         if !isEOL {
856 //             return "", nil, nil
857 //         }
858 //         if !isEOL {
859 //             return "", nil, nil
860 //         }
861 //         if !isEOL {
862 //             return "", nil, nil
863 //         }
864 //         if !isEOL {
865 //             return "", nil, nil
866 //         }
867 //         if !isEOL {
868 //             return "", nil, nil
869 //         }
870 //         if !isEOL {
871 //             return "", nil, nil
872 //         }
873 //         if !isEOL {
874 //             return "", nil, nil
875 //         }
876 //         if !isEOL {
877 //             return "", nil, nil
878 //         }
879 //         if !isEOL {
880 //             return "", nil, nil
881 //         }
882 //         if !isEOL {
883 //             return "", nil, nil
884 //         }
885 //         if !isEOL {
886 //             return "", nil, nil
887 //         }
888 //         if !isEOL {
889 //             return "", nil, nil
890 //         }
891 //         if !isEOL {
892 //             return "", nil, nil
893 //         }
894 //         if !isEOL {
895 //             return "", nil, nil
896 //         }
897 //         if !isEOL {
898 //             return "", nil, nil
899 //         }
900 //         if !isEOL {
901 //             return "", nil, nil
902 //         }
903 //         if !isEOL {
904 //             return "", nil, nil
905 //         }
906 //         if !isEOL {
907 //             return "", nil, nil
908 //         }
909 //         if !isEOL {
910 //             return "", nil, nil
911 //         }
912 //         if !isEOL {
913 //             return "", nil, nil
914 //         }
915 //         if !isEOL {
916 //             return "", nil, nil
917 //         }
918 //         if !isEOL {
919 //             return "", nil, nil
920 //         }
921 //         if !isEOL {
922 //             return "", nil, nil
923 //         }
924 //         if !isEOL {
925 //             return "", nil, nil
926 //         }
927 //         if !isEOL {
928 //             return "", nil, nil
929 //         }
930 //         if !isEOL {
931 //             return "", nil, nil
932 //         }
933 //         if !isEOL {
934 //             return "", nil, nil
935 //         }
936 //         if !isEOL {
937 //             return "", nil, nil
938 //         }
939 //         if !isEOL {
940 //             return "", nil, nil
941 //         }
942 //         if !isEOL {
943 //             return "", nil, nil
944 //         }
945 //         if !isEOL {
946 //             return "", nil, nil
947 //         }
948 //         if !isEOL {
949 //             return "", nil, nil
950 //         }
951 //         if !isEOL {
952 //             return "", nil, nil
953 //         }
```

```

2860 // Process all object subsections and create corresponding xref calls
2861 // Process all object subsections and create corresponding xref calls
2862 func processOffsetTableSubsection(x *PdfScanner, xRefTable *XRefTable, fields [string]
2863     error) {
2864     log.Debug.Println("processOffsetTableSubsection: begin")
2865     startObjNumber, err := stream.AtOf(fields[0])
2866     if err != nil {
2867         return err
2868     }
2869     objCount, err := stream.AtOf(fields[1])
2870     if err != nil {
2871         return err
2872     }
2873     log.Debug.Println("detected xref subsection, startObj=%d length=%d\n",
2874         startObjNumber, objCount)
2875     // Process all entries of this subsection into XRefTable entries.
2876     for i := 0; i < objCount; i++ {
2877         if err := parseObjectEntry(fields, xRefTable, startObjNumber+i); err == nil {
2878             return err
2879         }
2880     }
2881     log.Debug.Println("processOffsetTableSubsection: end")
2882     return nil
2883 }
2884
2885 // Parse compressed object.
2886 func parseCompressedObject(s string) (Object, error) {
2887     log.Debug.Println("parseCompressedObject: begin")
2888     o, err := parseObject(s)
2889     if err == nil {
2890         return nil, err
2891     }
2892     d, ok := s.(Dict)
2893     if !ok {
2894         return trivial.Object{Integer, Array, etc}
2895     }
2896     log.Debug.Println("compressedObject: end, any other than dict")
2897     return o, nil
2898 }
2899
2900 streamLength, streamNameTheif = d.Length()
2901 if streamLength == nil || streamNameTheif == nil {
2902     return err
2903 }
2904 log.Debug.Println("compressedObject: end, dict")
2905 return d, nil
2906 }
2907
2908 return nil, errors.New("pdfcpu: compressedObject: stream can be not be
2909 stored in an object stream")

```

[illegible]

```

710 // If offsetReadStream == nil {
711 //     return referenceStream
712 // }
713 // If ctx.ReadHeader() == http.StatusOK, Version() ≥ V1.0.0, ctx.ReadHeader()
714 // == nil, errors.Errorf("parseHeaderInfo: HTTP1.1 compatibility reader
715 // found incompatible version %s, %s", ctxable.VersionString())
716 // }
717 // log-Read.Primals("parseHeaderInfo end")
718 // return offsetReadStream, nil
719 // }
720 // }
721 // }
722 // This file is using cross reference streams.
723 // }
724 // }
725 // If ctx.ReadHeader()
726 // ctx.ReadHeader() == true
727 // ctx.ReadUsing(offsetReadStream) == true
728 // }
729 // }
730 // }
731 // If S-StreamReader readers process header objects contained
732 // // in S-StreamReader before attempting to process any WebSockets.
733 // Previous WebSockets is expected to have true entries for Header entries.
734 // No change in WebSockets will occur.
735 // }
736 // If ctx.ReadHeader()
737 // return nil, errors.Errorf("Stream offsetReadStream, ctx) == nil if nil
738 // }
739 // }
740 // }
741 // }
742 // log-Read.Primals("parseHeaderInfo end")
743 // return offset, nil
744 // }
745 // }
746 // }
747 // }
748 // }
749 // }
750 // }
751 // }
752 // }
753 // }
754 // }
755 // }
756 // }
757 // }
758 // }
759 // }
760 // }
761 // }
762 // }
763 // }
764 // }
765 // }
766 // }
767 // }
768 // }
769 // }
770 // }
771 // }
772 // }
773 // }
774 // }
775 // }
776 // }
777 // }
778 // }
779 // }
780 // }
781 // }
782 // }
783 // }
784 // }
785 // }
786 // }
787 // }
788 // }
789 // }
790 // }
791 // }
792 // }
793 // }
794 // }
795 // }
796 // }
797 // }
798 // }
799 // }
800 // }
801 // }
802 // }
803 // }
804 // }
805 // }
806 // }
807 // }
808 // }
809 // }
810 // }
811 // }
812 // }
813 // }
814 // }
815 // }
816 // }
817 // }
818 // }
819 // }
820 // }
821 // }
822 // }
823 // }
824 // }
825 // }
826 // }
827 // }
828 // }
829 // }
830 // }
831 // }
832 // }
833 // }
834 // }
835 // }
836 // }
837 // }
838 // }
839 // }
840 // }
841 // }
842 // }
843 // }
844 // }
845 // }
846 // }
847 // }
848 // }
849 // }
850 // }
851 // }
852 // }
853 // }
854 // }
855 // }
856 // }
857 // }
858 // }
859 // }
860 // }
861 // }
862 // }
863 // }
864 // }
865 // }
866 // }
867 // }
868 // }
869 // }
870 // }
871 // }
872 // }
873 // }
874 // }
875 // }
876 // }
877 // }
878 // }
879 // }
880 // }
881 // }
882 // }
883 // }
884 // }
885 // }
886 // }
887 // }
888 // }
889 // }
890 // }
891 // }
892 // }
893 // }
894 // }
895 // }
896 // }
897 // }
898 // }
899 // }
900 // }
901 // }
902 // }
903 // }
904 // }
905 // }
906 // }
907 // }
908 // }
909 // }
910 // }
911 // }
912 // }
913 // }
914 // }
915 // }
916 // }
917 // }
918 // }
919 // }
920 // }
921 // }
922 // }
923 // }
924 // }
925 // }
926 // }
927 // }
928 // }
929 // }
930 // }
931 // }
932 // }
933 // }
934 // }
935 // }
936 // }
937 // }
938 // }
939 // }
940 // }
941 // }
942 // }
943 // }
944 // }
945 // }
946 // }
947 // }
948 // }
949 // }
950 // }
951 // }
952 // }
953 // }
954 // }
955 // }
956 // }
957 // }
958 // }
959 // }
960 // }
961 // }
962 // }
963 // }
964 // }
965 // }
966 // }
967 // }
968 // }
969 // }
970 // }
971 // }
972 // }
973 // }
974 // }
975 // }
976 // }
977 // }
978 // }
979 // }
980 // }
981 // }
982 // }
983 // }
984 // }
985 // }
986 // }
987 // }
988 // }
989 // }
990 // }
991 // }
992 // }
993 // }
994 // }
995 // }
996 // }
997 // }
998 // }
999 // }
1000 // }
1001 // }
1002 // }
1003 // }
1004 // }
1005 // }
1006 // }
1007 // }
1008 // }
1009 // }
1010 // }
1011 // }
1012 // }
1013 // }
1014 // }
1015 // }
1016 // }
1017 // }
1018 // }
1019 // }
1020 // }
1021 // }
1022 // }
1023 // }
1024 // }
1025 // }
1026 // }
1027 // }
1028 // }
1029 // }
1030 // }
1031 // }
1032 // }
1033 // }
1034 // }
1035 // }
1036 // }
1037 // }
1038 // }
1039 // }
1040 // }
1041 // }
1042 // }
1043 // }
1044 // }
1045 // }
1046 // }
1047 // }
1048 // }
1049 // }
1050 // }
1051 // }
1052 // }
1053 // }
1054 // }
1055 // }
1056 // }
1057 // }
1058 // }
1059 // }
1060 // }
1061 // }
1062 // }
1063 // }
1064 // }
1065 // }
1066 // }
1067 // }
1068 // }
1069 // }
1070 // }
1071 // }
1072 // }
1073 // }
1074 // }
1075 // }
1076 // }
1077 // }
1078 // }
1079 // }
1080 // }
1081 // }
1082 // }
1083 // }
1084 // }
1085 // }
1086 // }
1087 // }
1088 // }
1089 // }
1090 // }
1091 // }
1092 // }
1093 // }
1094 // }
1095 // }
1096 // }
1097 // }
1098 // }
1099 // }
1100 // }
1101 // }
1102 // }
1103 // }
1104 // }
1105 // }
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 // }
```

```

560 // This is the first trailer, so we can't be sure it's the last one.
561 // fields = strings.Fields(line)
562 //
563 // Look for the "parameters" section. All subsections end
564 // with a "%%EOF" marker.
565 //
566 // If it's the "parameters" section, then it's the "trailer"
567 //
568 // return nil, errors.Errorf("section: missing trailer dict, line = %q",
569 // line)
570 //
571 // return processTrailer(cx, s, line)
572 //
573 // Look for the "parameters" section. All subsections end
574 // with a "%%EOF" marker.
575 //
576 // return processTrailer(cx, s, line)
577 //
578 // Get version from first line. This is the
579 // beginning with PDF 1.1, the version entry in the document's catalog dictionary
580 // is used by the host entry in the file's trailer, as described in 1.5.3, "File
581 // signatures".
582 //
583 // If we can't find the version entry, then the version specified in the header
584 // (PDF Version) is used.
585 //
586 // The header version comes from the first line of the file.
587 //
588 // In the number of characters used for file (s = 2).
589 //
590 // headerVersion is a ReaderVersion (v version, eCount int, err error) {
591 //
592 //     func headerVersion(s string) ReaderVersion {
593 //         // Header version begins
594 //         //
595 //         var err error
596 //         headerVersion := errors.New("pdf: header version: corrupt pdf stream =
597 //         version available")
598 //         //
599 //         // If the file has fields which hold the version of this PDF file,
600 //         // we call this the header version.
601 //         //
602 //         if err := s.Seek(0, io.SeekStart); err == nil {
603 //             return nil, s, err
604 //         }
605 //         //
606 //         buf := make([]byte, 256)
607 //         if err := s.Read(buf); err == nil {
608 //             return nil, s, err
609 //         }
610 //         //
611 //         s = strings.TrimSpace(buf)
612 //         prefix = "%%PDF-"
613 //         //
614 //         if len(s) < 8 || !strings.HasPrefix(prefix, s) {
615 //             return nil, s, errors.Errorf("header version: unknown PDF header Version")
616 //         }
617 //         if err := nil {
618 //             return nil, s, errors.Errorf("header version: unknown PDF header Version")
619 //         }
620 //         //
621 //         s = s[len(prefix):]
622 //         s = strings.TrimLeft(s, "V")
623 //         //
624 //         // Detect the end of each should be 1 (000), 2 (0000), or 3 chars (000000).
625 //         //
626 //         // If the file is a PDF 1.1, the version entry in the document's catalog dictionary

```

```

1197 logMsg.Println("readmeTable: begin")
1198
1199 // Read the first section
1200 offset, err = offSetLastTableSection(cta)
1201 if err != nil {
1202     return
1203 }
1204
1205 err = buildOneTableStartingAt(cta, offset)
1206 if err != io.EOF {
1207     return errors.Wrap(err, "readmeTable: unexpected eof")
1208 }
1209
1210 if err != nil {
1211     return
1212 }
1213
1214 // Log list of free objects (not the "free list")
1215 // /logMsg.Println("freeList: %v", cta.FreeObjIndex)
1216
1217 // Ensure valid freeList of objects.
1218 err = cta.ensureValidFreeList()
1219 if err != nil {
1220     return
1221 }
1222
1223 logMsg.Println("readmeTable: end")
1224
1225 return
1226 }
1227
1228 // =====
1229
1230 func growBuf(off []byte, size int, rd io.Reader) ([]byte, error) {
1231     b := make([]byte, size)
1232
1233     // err = rd.Read(b)
1234     if err != nil {
1235         return nil, err
1236     }
1237
1238     // /logMsg.Println("growBufBy: Read %d bytes", n)
1239
1240     return append(b, b...), nil
1241 }
1242
1243 // =====
1244
1245 func nextStreamOffset(line string, streamIdx int) (off int) {
1246     off = streamIdx + len("stream")
1247
1248     // Skip optional blanks
1249     // /logMsg.Println("nextStreamOffset: skip blanks")
1250     // /logMsg.Println("nextStreamOffset: skip optional whitespace instead")
1251     for ; line[off] == 0x20; off++ {
1252     }
1253
1254     // Skip the end
1255     if line[off] == '\n' {
1256         off++
1257     }
1258     return
1259 }
1260
1261 // /for i := 0; i < len(line); i++ {
1262 //     // Skip the end
1263 //     if line[i] == '\n' {
1264 //         i++
1265 //     }
1266 // }

```

```

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

```

```

344
345 // Parse the object stream into objects: read some data, then into objects: read dict, 0 Array
346 func parseObjectStream(out ObjectStreamDict) error {
347
348     log.Msg.Printf("parseObjectStream begin: decoding %d objects, %d", out.ObjCount,
349
350     decodeContext := out.decodeContext
351     posObj := decodeContext.offset + out.firstObjOffset
352
353     obj := strings.Fields(string(parseObj))
354     if len(obj) % 2 != 0 {
355         return errors.New("parsing parseObjectStream: corrupt object stream dict")
356     }
357
358     // obj, 10 10 12 25 = 2 objects: #10 & offset 0, #12 & offset 25
359
360     var objArray Array
361
362     for i := 0; i < len(obj); i += 2 {
363         offset, err := strconv.Atoi(obj[i+1])
364         if err != nil {
365             return err
366         }
367
368         offset += out.firstObjOffset
369
370         if i % 2 ==
371             dict := strings.Fields(string(parseDict[offset]))
372             log.Msg.Printf("parseObjectStream: objstring = %s\n", dict)
373             o, err := compressObject(dict)
374             if err != nil {
375                 return err
376             }
377
378             log.Msg.Printf("parseObjectStream: [hd] = obj %s\n", obj[i/2], obj[i+1])
379             objArray = append(objArray, o)
380         }
381         if i % 2 == 1 {
382             dict := strings.Fields(string(parseDict[offset]))
383             log.Msg.Printf("parseObjectStream: objstring = %s\n", dict)
384             o, err := compressObject(dict)
385             if err != nil {
386                 return err
387             }
388
389             log.Msg.Printf("parseObjectStream: [hd] = obj %s\n", obj[i/2], obj[i+1])
390             objArray = append(objArray, o)
391         }
392     }
393     offset := offset
394 }

```

```

547         return null, err
548     }
549
550     // We expect a stream and therefore "stream" before "endobj". If "endobj" within
551     // there is guaranteed that "endobj" is contained in this buffer for large
552     // or streaming objects.
553     if !streamed && !len(endobj) > 0 && !len(endobj) < len(buf) {
554         null, errors.New("pdfcorrupt: parse/Stream: corrupt pdf file")
555     }
556
557     // Just object parse buf.
558     buf := line[:streamed]
559
560     obj, checksum, generationNumber, err := parseObjectAttributes(&f)
561     if err != nil {
562         return null, err
563     }
564     return obj, checksum
565 }
566
567 func (p *Parser) parseStream() (obj *Object, checksum uint32, err error) {
568     Log.Debug.Printf("parse/Stream: stream object id %d len %d\n", obj.ObjectID(),
569         streamObjectLen)
570     Log.Debug.Printf("parse/Stream: deferencing object %d\n", obj.ObjectNumber())
571     o, err := parseObject(&f)
572     if err != nil {
573         return null, errors.Wrap(err, "parse/Stream: no object")
574     }
575
576     // We have a real stream object
577     Log.Debug.Printf("parse/Stream: we have an object: %d\n", o.ObjectID())
578
579     streamOffset := offset
580     if err := readObjectContents(&f, obj.ObjectNumber(), streamOffset);
581     if err != nil {
582         return null, err
583     }
584     // We have a real stream object
585     err = readObjectInfo(&f, Dict(&f, &f.Table))
586     if err != nil {
587         return null, err
588     }
589
590     // Parse objectStream and create objectStream entries for embedded objects
591     err = extractObjectTableEntriesForStream(&f, content, &f)
592     if err != nil {
593         return null, err
594     }
595
596     // Create objectTable entry for theStreamStream.
597     entry := &f.ObjectTableEntry{
598         ObjectID: obj.ObjectID(),
599         Offset: offset,
600         Generation: generationNumber,
601         Object: obj,
602     }

```

```

774
775     return nil, nil
776 }
777
778 func initDict(s string) (bool, error) {
779     ok, err := parseDict(s)
780     if err != nil {
781         return false, err
782     }
783     ok, os := os.Dict()
784     return ok, nil
785 }
786
787 func scanTrailer(s *bufio.Scanner, line string, error) {
788     buf := bytes.Buffer{}
789     var err error
790     for _, tag := range s.Tag() {
791         log.Debug.Printf("line %s", line)
792         // Scan for dict start tag "=".
793         for {
794             s = strings.Index(line, "=")
795             if s >= 0 {
796                 break
797             }
798             line, err = scanLine(s)
799             log.Debug.Printf("line %s", line)
800             if err != nil {
801                 return "", err
802             }
803         }
804     }
805     line = line[s:]
806     buf.WriteString(line)
807     log.Debug.Printf("Trailer dictbuf after start tag: %s", line)
808 }
809
810 // Scan for dict end tag "}" but account for inner dicts.
811 line = line[2:]
812
813 for {
814     if len(line) == 0 {
815         err = scanLine(s)
816         if err != nil {
817             return "", err
818         }
819         buf.WriteString(line)
820         log.Debug.Printf("Scanner trailer dictbuf next line: %s", line)
821     }
822     s = strings.Index(line, "=")
823     if s < 0 {
824         // s = strings.Index(line, "}")
825         if s >= 0 {

```

```

3620:         if s[i] == '\n' {
3621:             eolCount++
3622:         } else if s[i] == '0' {
3623:             eolCount++
3624:             if s[i] == '\n' {
3625:                 eolCount++
3626:             }
3627:         }
3628:         if eolCount == 0 {
3629:             return nil, b.Errorf("corrupt header")
3630:         }
3631:     }
3632:     log.Printf("header version: %d, found header version: %d\n", pdfVersion, pdfVersion)
3633:     return pdfVersion, eolCount, nil
3634: }
3635:
3636: // Returns the object of a hash for digested content of pdf sections.
3637: // It populates the assembly by reading in all indirect objects by line
3638: // and works on the assumption of a single entry count - meaning no incremental
3639: // updates are supported.
3640: func generateSectionObject(xContext) error {
3641:     var s string
3642:     g := FreeObjectGeneration
3643:     ctx.InitHash()
3644:     g.Free()
3645:     g.Offset = 0
3646:     g.Generation = 0
3647:
3648:     rs := ctx.Read.rs
3649:     eolCount := ctx.Read.eolCount
3650:     var off offset
3651:
3652:     for {
3653:         rd, err := newHashedIndirectHeader(s, hofast)
3654:         if err == nil {
3655:             return err
3656:         }
3657:
3658:         s = bufio.NewReaderLine(s)
3659:         s.SplitScanLines()
3660:
3661:         bb := [Type]{}
3662:         withIndirect := bool
3663:         withHeader := bool
3664:         withTrailer := bool
3665:
3666:         for {
3667:             line, err := scanIndirect(s)
3668:             if err == nil {
3669:                 break
3670:             }
3671:             if withHeader {
3672:                 offset = append(lineLine) + eolCount
3673:             }
3674:             if withIndirect {
3675:                 bb = append(bb, *)
3676:             }
3677:             bb = append(bb, line...)
3678:             if withTrailer {
3679:                 w := strings.Index(line, "trailer")
3680:                 if w >= 0 {

```

[illegible]

```

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

```

```

630 // https://stackoverflow.com/questions/10467698/using-py-cpp-pybind11-with-gsl
631 #include <array>
632
633 #define M 100
634
635 log_Read_Printf("array object is %s\n", typeid(array).name());
636
637 return nil;
638
639
640 // For each object embedded in this xrefStream create the corresponding xref table
641 // entry and be prepared to be iterated.
642 #define xrefTableEntry(xrefInfo, xrefStream) \
643     { \
644         log_Read_Printf("xrefInfo is %s\n", typeid(xrefInfo).name()); \
645         log_Read_Printf("xrefStream is %s\n", typeid(xrefStream).name()); \
646     }
647
648 // Note: \
649 // * A value of zero for an element in the M array indicates that the corresponding \
650 // * element has not been generated in the stream. \
651 // * The default value shall be zero, if there is none. \
652 // * If the default value shall be non-zero, the type field may not be present, and shall \
653 // * default to type t.
654
655 int i = xrefInfo.M;
656 int j = xrefInfo.N;
657 int k = xrefInfo.O;
658
659 xrefEntryLen = 11 + 12 + 13
660
661 log_Read_Printf("xrefInfo is %s\n", typeid(xrefInfo).name());
662 log_Read_Printf("xrefStream is %s\n", typeid(xrefStream).name());
663
664 if (lenBuf > xrefEntryLen) {
665     return errors.MkPanic("extractor failed to extract xrefStream: corrupt");
666 }
667
668 objCount = lenBuf.xsd.Objects;
669 log_Read_Printf("xrefInfo is %s\n", typeid(xrefInfo).name());
670 log_Read_Printf("xrefStream is %s\n", typeid(xrefStream).name());
671
672 objCount = lenBuf.xsd.Objects;
673
674 log_Read_Printf("xrefInfo is %s\n", typeid(xrefInfo).name());
675 log_Read_Printf("xrefStream is %s\n", typeid(xrefStream).name());
676
677 if (lenBuf < objCount + xrefEntryLen) {
678     // Something there is an additional xref entry not accounted for by "index".
679     // This may be a corrupt stream.
680     return errors.MkPanic("extractor failed to extract xrefStream: corrupt");
681 }
682
683 xrefStream =
684
685 j = 0
686
687 // bufioReader interprets the content of buf as an int64.
688 bufioReader = bufioBuf(buf)
689
690 for i = 0; i < range buf {
691     i = bufioReader.ReadInt64()
692 }
693
694 return
695

```

```

445 //
446 // Log Read/Print("passwordStream: Insert new SHA256 entry for Object UUID,
447 // subjectNumber)
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825 //
826 //
827 //
828 //
829 //
830 //
831 //
832 //
833 //
834 //
835 //
836 //
837 //
838 //
839 //
840 //
841 //
842 //
843 //
844 //
845 //
846 //
847 //
848 //
849 //
850 //
851 //
852 //
853 //
854 //
855 //
856 //
857 //
858 //
859 //
860 //
861 //
862 //
863 //
864 //
865 //
866 //
867 //
868 //
869 //
870 //
871 //
872 //
873 //
874 //
875 //
876 //
877 //
878 //
879 //
880 //
881 //
882 //
883 //
884 //
885 //
886 //
887 //
888 //
889 //
890 //
891 //
892 //
893 //
894 //
895 //
896 //
897 //
898 //
899 //
900 //
901 //
902 //
903 //
904 //
905 //
906 //
907 //
908 //
909 //
910 //
911 //
912 //
913 //
914 //
915 //
916 //
917 //
918 //
919 //
920 //
921 //
922 //
923 //
924 //
925 //
926 //
927 //
928 //
929 //
930 //
931 //
932 //
933 //
934 //
935 //
936 //
937 //
938 //
939 //
940 //
941 //
942 //
943 //
944 //
945 //
946 //
947 //
948 //
949 //
950 //
951 //
952 //
953 //
954 //
955 //
956 //
957 //
958 //
959 //
960 //
961 //
962 //
963 //
964 //
965 //
966 //
967 //
968 //
969 //
970 //
971 //
972 //
973 //
974 //
975 //
976 //
977 //
978 //
979 //
980 //
981 //
982 //
983 //
984 //
985 //
986 //
987 //
988 //
989 //
990 //
991 //
992 //
993 //
994 //
995 //
996 //
997 //
998 //
999 //
1000 //

```

```

640         }
641         if x == 0 {
642             // Check for zero
643             ok, err = isSafeBuf.String()
644             if err == nil || ok {
645                 return buf.String(), nil
646             }
647         } else {
648             k++
649             line = line[i+j:]
650             continue
651         }
652         // No go
653         line, err = scanline(s)
654         if err == nil {
655             return "", err
656         }
657         buf.WriteString(line)
658         buf.WriteString("\n")
659         log.Debug.Printf("scan trailer dictbuf next line: %s\n", line)
660     } else {
661         // No go
662         x = string.Index(line, ">")
663         if x < 0 {
664             k++
665             line = line[i:]
666         } else {
667             if i < j {
668                 // handle ok
669                 k++
670                 line = line[i+j:]
671             } else {
672                 // handle no
673                 if k == 0 {
674                     // Check for dict
675                     ok, err = isSafeBuf.String()
676                     if err == nil || ok {
677                         return buf.String(), nil
678                     }
679                 } else {
680                     k++
681                 }
682             }
683             line = line[i+j:]
684         }
685     }
686 }
687 }
688
689 func processTrailer(s context, s *bufio.Scanner, line string) (uint64, error) {
690     var trailerString string
691     if line == "trailer" {
692         trailerString = line[1:]
693         log.Debug.Printf("process trailer: trailer leftover: %s\n", trailerString)
694     }
695 }

```

```

5300 // Use the given process trailer to generate a new trailer
5301 err = processTrailer(cttr, s, string(bb))
5302 return err
5303 }
5304 }
5305 // Append all until "trailer".
5306 s = string(lineLine, "trailer")
5307 if s > 0 {
5308   bb = append(bb, line...)
5309   withTrailer = true
5310 }
5311 continue
5312 }
5313 s = string(lineLine, "text")
5314 if s > 0 {
5315   offset = int64((mcLine + e) - colCount)
5316   withTrailer = true
5317   continue
5318 }
5319 if withOffset {
5320   s = string(lineLine, "obj")
5321   if s > 0 {
5322     withOffset = true
5323     off = offset
5324     bb = append(bb, line[1:]...)
5325   }
5326   offset = int64((mcLine + e) - colCount)
5327   continue
5328 }
5329 }
5330 // write out
5331 offset = int64((mcLine + e) - colCount)
5332 bb = append(bb, s...)
5333 bb = append(bb, line...)
5334 if s != string(bb, "endobj") {
5335   if s > 0 {
5336     objRef, generation, err = parseObjAttributes(l)
5337     if err == nil {
5338       return err
5339     }
5340     if err == eof {
5341       return err
5342     }
5343     tbl := TableEntry{
5344       frast: frast,
5345       objRef: objRef,
5346       generation: generation
5347     }
5348     bb = nil
5349     withOffset = false
5350   }
5351 }
5352 return nil
5353 }
5354 // Read the trailer for reading. If the trailer is not found,
5355 // log the error.
5356 log.Read.Println("no trailer found")
5357 }

```

```

1010 // https://en.cppreference.com/w/cpp/string/basic/basic_string_view
1011
1012 // line in string(buf)
1013 ending = string::index(line, "ending");
1014 streamend = string::index(line, "streamend");
1015
1016 if (ending > 0 && (streamend < 0 || streamend > ending)) {
1017     // No stream marker, in buf detected.
1018     break;
1019 }
1020
1021 // For very rare cases where "stream" also occurs within buf dict
1022 // we need to find the last "stream" before, for a possible end marker.
1023 // We streamend > 0, the streamend after the last occurrence of (buf[line, streamend])
1024 // lastStreamMarker(streamend, ending, line)
1025
1026 log.Debug.Printf("buffer: ending=%d streamend=%d\n", ending, streamend);
1027
1028 if streamend > 0 {
1029     // streamend == ... the offset where the actual stream begins.
1030     // In time after the call after "stream"
1031     //
1032     // slice in 10 - for optional whitespace + oct (max 2 chars)
1033     need = streamend + len("stream"); // slack
1034     if len(line) < need {
1035         // to prevent buffer overflow.
1036         buf, err = growBuf(buf, need-len(line), nil)
1037         if err != nil {
1038             return nil, 0, 0, err
1039         }
1040         line = string(buf[0:need])
1041     }
1042     streamOffset = bufIndex(lastStreamOffset+line, streamend)
1043
1044 // (log.Debug.Printf("buffer: end, returned beforeIndex streamOffset=%d\n", len(buf)
1045 // streamOffset))
1046
1047 return buf, ending, streamend, streamOffset, nil
1048 }
1049
1050 // func (b *Buffer) GetLine(buf []byte, endLineIndex int) (string, bool) {
1051 // func b.NextStreamOffsetAfterLastOffset(buf []string, streamend) int bool {
1052 //
1053 // (log.Debug.Printf("beforeStreamOffsetAfterLastOffset: begin"))
1054 //
1055 // // Get a slice of the chunk right in front of "stream".
1056 // b := buf[streamIndex:]
1057 //
1058 // // Look for last end of dict marker.
1059 // end := string.LastIndex(b, ">")
1060 // if end < 0 {
1061 //     // end of dict in buf
1062 }

```



```

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=decryptHeaderEnd:NOTDICT: end: %s\n", ok)
1578
1579     return ok
1580 }
1581
1582 func buildFilterPipeline(ctx *Context, filterArray, decodeParsesArr Array, decodeParses
1583     *[]*Filter, *[]*Filter) {
1584     var filterPipeline []*Filter
1585
1586     for i, f := range filterArray {
1587
1588         filterName, ok := f.(Name)
1589         if !ok {
1590             corrupt := true
1591             err := New("pdcgo: buildFilterPipeline: filterArray elements
1592                 corrupt")
1593             return nil, errors.New(err).pdcgo: buildFilterPipeline: filterArray elements
1594                 corrupt
1595         }
1596         if decodeParses == nil || decodeParsesArr[i] == nil {
1597             filterPipeline = append(filterPipeline, *Filter{Name:
1598                 filterName, DecodeParses: nil})
1599             continue
1600         }
1601         dict, ok := decodeParsesArr[i].(Dict)
1602         if !ok {
1603             corrupt := true
1604             err := New("pdcgo: buildFilterPipeline: i.IndefHeader")
1605             return nil, errors.New(err).pdcgo: buildFilterPipeline: corrupt Dict: %s\n",
1606                 dict)
1607         }
1608         if s, err := dereferenceDict(dict, Indef.ObjectNumber.Value());
1609         if err != nil {
1610             return nil, err
1611         }
1612         dict = d
1613     }
1614
1615     filterPipeline = append(filterPipeline, *Filter{Name: filterName.String(),
1616         DecodeParses: dict})
1617 }
1618
1619 //
1620 //
1621 //
1622 //
1623 //
1624 //
1625 //
1626 //
1627 //
1628 //
1629 //
1630 //
1631 //
1632 //
1633 //
1634 //
1635 //
1636 //
1637 //
1638 //
1639 //
1640 //
1641 //
1642 //
1643 //
1644 //
1645 //
1646 //
1647 //
1648 //
1649 //
1650 //
1651 //
1652 //
1653 //
1654 //
1655 //
1656 //
1657 //
1658 //
1659 //
1660 //
1661 //
1662 //
1663 //
1664 //
1665 //
1666 //
1667 //
1668 //
1669 //
1670 //
1671 //
1672 //
1673 //
1674 //
1675 //
1676 //
1677 //
1678 //
1679 //
1680 //
1681 //
1682 //
1683 //
1684 //
1685 //
1686 //
1687 //
1688 //
1689 //
1690 //
1691 //
1692 //
1693 //
1694 //
1695 //
1696 //
1697 //
1698 //
1699 //
1700 //
1701 //
1702 //
1703 //
1704 //
1705 //
1706 //
1707 //
1708 //
1709 //
1710 //
1711 //
1712 //
1713 //
1714 //
1715 //
1716 //
1717 //
1718 //
1719 //
1720 //
1721 //
1722 //
1723 //
1724 //
1725 //
1726 //
1727 //
1728 //
1729 //
1730 //
1731 //
1732 //
1733 //
1734 //
1735 //
1736 //
1737 //
1738 //
1739 //
1740 //
1741 //
1742 //
1743 //
1744 //
1745 //
1746 //
1747 //
1748 //
1749 //
1750 //
1751 //
1752 //
1753 //
1754 //
1755 //
1756 //
1757 //
1758 //
1759 //
1760 //
1761 //
1762 //
1763 //
1764 //
1765 //
1766 //
1767 //
1768 //
1769 //
1770 //
1771 //
1772 //
1773 //
1774 //
1775 //
1776 //
1777 //
1778 //
1779 //
1780 //
1781 //
1782 //
1783 //
1784 //
1785 //
1786 //
1787 //
1788 //
1789 //
1790 //
1791 //
1792 //
1793 //
1794 //
1795 //
1796 //
1797 //
1798 //
1799 //
1800 //
1801 //
1802 //
1803 //
1804 //
1805 //
1806 //
1807 //
1808 //
1809 //
1810 //
1811 //
1812 //
1813 //
1814 //
1815 //
1816 //
1817 //
1818 //
1819 //
1820 //
1821 //
1822 //
1823 //
1824 //
1825 //
1826 //
1827 //
1828 //
1829 //
1830 //
1831 //
1832 //
1833 //
1834 //
1835 //
1836 //
1837 //
1838 //
1839 //
1840 //
1841 //
1842 //
1843 //
1844 //
1845 //
1846 //
1847 //
1848 //
1849 //
1850 //
1851 //
1852 //
1853 //
1854 //
1855 //
1856 //
1857 //
1858 //
1859 //
1860 //
1861 //
1862 //
1863 //
1864 //
1865 //
1866 //
1867 //
1868 //
1869 //
1870 //
1871 //
1872 //
1873 //
1874 //
1875 //
1876 //
1877 //
1878 //
1879 //
1880 //
1881 //
1882 //
1883 //
1884 //
1885 //
1886 //
1887 //
1888 //
1889 //
1890 //
1891 //
1892 //
1893 //
1894 //
1895 //
1896 //
1897 //
1898 //
1899 //
1900 //
1901 //
1902 //
1903 //
1904 //
1905 //
1906 //
1907 //
1908 //
1909 //
1910 //
1911 //
1912 //
1913 //
1914 //
1915 //
1916 //
1917 //
1918 //
1919 //
1920 //
1921 //
1922 //
1923 //
1924 //
1925 //
1926 //
1927 //
1928 //
1929 //
1930 //
1931 //
1932 //
1933 //
1934 //
1935 //
1936 //
1937 //
1938 //
1939 //
1940 //
1941 //
1942 //
1943 //
1944 //
1945 //
1946 //
1947 //
1948 //
1949 //
1950 //
1951 //
1952 //
1953 //
1954 //
1955 //
1956 //
1957 //
1958 //
1959 //
1960 //
1961 //
1962 //
1963 //
1964 //
1965 //
1966 //
1967 //
1968 //
1969 //
1970 //
1971 //
1972 //
1973 //
1974 //
1975 //
1976 //
1977 //
1978 //
1979 //
1980 //
1981 //
1982 //
1983 //
1984 //
1985 //
1986 //
1987 //
1988 //
1989 //
1990 //
1991 //
1992 //
1993 //
1994 //
1995 //
1996 //
1997 //
1998 //
1999 //
2000 //
2001 //
2002 //
2003 //
2004 //
2005 //
2006 //
2007 //
2008 //
2009 //
2010 //
2011 //
2012 //
2013 //
2014 //
2015 //
2016 //
2017 //
2018 //
2019 //
2020 //
2021 //
2022 //
2023 //
2024 //
2025 //
2026 //
2027 //
2028 //
2029 //
2030 //
2031 //
2032 //
2033 //
2034 //
2035 //
2036 //
2037 //
2038 //
2039 //
2040 //
2041 //
2042 //
2043 //
2044 //
2045 //
2046 //
2047 //
2048 //
2049 //
2050 //
2051 //
2052 //
2053 //
2054 //
2055 //
2056 //
2057 //
2058 //
2059 //
2060 //
2061 //
2062 //
2063 //
2064 //
2065 //
2066 //
2067 //
2068 //
2069 //
2070 //
2071 //
2072 //
2073 //
2074 //
2075 //
2076 //
2077 //
2078 //
2079 //
2080 //
2081 //
2082 //
2083 //
2084 //
2085 //
2086 //
2087 //
2088 //
2089 //
2090 //
2091 //
2092 //
2093 //
2094 //
2095 //
2096 //
2097 //
2098 //
2099 //
2100 //
2101 //
2102 //
2103 //
2104 //
2105 //
2106 //
2107 //
2108 //
2109 //
2110 //
2111 //
2112 //
2113 //
2114 //
2115 //
2116 //
2117 //
2118 //
2119 //
2120 //
2121 //
2122 //
2123 //
2124 //
2125 //
2126 //
2127 //
2128 //
2129 //
2130 //
2131 //
2132 //
2133 //
2134 //
2135 //
2136 //
2137 //
2138 //
2139 //
2140 //
2141 //
2142 //
2143 //
2144 //
2145 //
2146 //
2147 //
2148 //
2149 //
2150 //
2151 //
2152 //
2153 //
2154 //
2155 //
2156 //
2157 //
2158 //
2159 //
2160 //
2161 //
2162 //
2163 //
2164 //
2165 //
2166 //
2167 //
2168 //
2169 //
2170 //
2171 //
2172 //
2173 //
2174 //
2175 //
2176 //
2177 //
2178 //
2179 //
2180 //
2181 //
2182 //
2183 //
2184 //
2185 //
2186 //
2187 //
2188 //
2189 //
2190 //
2191 //
2192 //
2193 //
2194 //
2195 //
2196 //
2197 //
2198 //
2199 //
2200 //
```

[illegible]

```

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

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry %d", objId)
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs length 2 or 4", objId)
2043             }
2044             offset, ok = a[0].(Integer)
2045             if !ok {
2046                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs Integer values", objId)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offset64
2050             if len(a) == 4 {
2051                 if !
2052                     return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry %d, needs Integer values", objId)
2053             }
2054             ctx.OffsetOverluminTable = offset64
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(ctx, objId, genNr); err != nil {
2065         return errors.Wrapf(err, "dereferencingContext: problem dereferencing stream %d",
2066             objId)
2067     }
2068     ctx.Read.BinaryFileSize += s.GetSizeLength
2069
2070     // Decode stream's content
2071     err = saveDecodedStreamContent(ctx, s, objId, genNr, ctx.DecodedAllStreams)
2072     return err
2073 }
2074
2075 func updateLinearizationDict(ctx *Context, o Object) {
2076     switch o := o.(type) {
2077     case StreamDict:
2078         ctx.Read.BinaryFileSize += o.GetSizeLength
2079     }
2080 }

```

[illegible]

```

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

```

```

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

```

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

```

```

1573 // On return object's destructor may be called, problem dereferencing object
1574 stream.Md, no ref table entry, entry.ObjectStreamId)
1575
1576 //
1577 // Object of class entry has to be an ObjectStreamId
1578 //
1579 sd, 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.Read.Print("logStream no stream content")
1613         }
1614         if o.IsSpkgContent {
1615             //log.Read.Print("content %v\n", StreamId.Content)
1616         }
1617     case ObjectStreamId:
1618         if o.Content == nil {
1619             //log.Read.Print("logStream no object stream content")
1620         }
1621         if o.IsSpkgContent {
1622             //log.Read.Print("logStream no object stream content %v\n", o.Content)
1623         }
1624         if o.IsObjArray {
1625             //log.Read.Print("logStream no object stream obj array")
1626         }
1627         if o.IsObjArray {
1628             //log.Read.Print("logStream no object stream obj array %v\n", o.ObjArray)
1629         }
1630     }
1631 }
1632
1633 //
1634 // Default:

```

[illegible]

```

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

```

```

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

```

[illegible]

```

130 // @see https://github.com/ericniebler/psutil/blob/master/psutil/_psutil_linux.c
131         log.Read.PrintIn("logStream", 0, OBJECTS_READY) }
132
133
134
135 // Decode all object streams to contained objects are ready to be used.
136 void decodeObjectStreams(cts::Context) error {
137     // @see
138     // @entry "externs" intentionally left out.
139     // No object stream collection validation necessary.
140
141     log.Read.PrintIn("decodeobjectstreams: begin")
142
143     // Get sorted slice of object numbers.
144     // See key [1]
145     for k = range cts.Read.ObjectStreams {
146         keys = append(keys, k)
147     }
148     sort.Int(keys)
149
150     for _ , objectNumber = range keys {
151         // @see ObjectReadyEntry.
152         entry = cts.StableTable.Table(objectNumber)
153         if entry == nil {
154             return errors.Errorf("decodeobjectstreams: missing entry for objectNumber %d",
155                 objectNumber)
156         }
157         log.Read.PrintIn("decodeobjectstreams: parsing object stream for objectNumber",
158             objectNumber)
159
160         // Parse object stream from file.
161         o, err = ParseObjectStream(entry.Offset, objectNumber, entry.Generation)
162         if err != nil || o == nil {
163             return errors.New("pdpcc: decodeobjectstreams: corrupt object stream")
164         }
165
166         // Ensure streamObject
167         sd, ok = o.(StreamObject)
168         if !ok {
169             return errors.New("pdpcc: decodeobjectstreams: corrupt object stream")
170         }
171
172         // Load decoded stream content to stableTable.
173         if err = loadDecodedStreamContent(cts, sd, err) != nil {
174             return errors.Wrap(err, "decodeobjectstreams: problem dereferencing
175                 object stream")
176         }
177
178         // Save decoded stream content to stableTable.
179         if err = saveDecodedStreamContent(cts, sd, objectNumber, entry.Generation,
180             true) != nil {
181             return err
182         }
183     }
184 }

```

```

2342 // err = err + ParseObject(err, 'entry.offset, objNr, entry.generation)
2343 if err == nil {
2344     return errors.Wrap(err, "dereferencedObject: problem dereferencing object id")
2345 }
2346 }
2347
2348 entry.Object = o
2349
2350 // // Linearization objects are validated and removed for stats only.
2351 err = handleLinearizationAndPurge(ctxt, o, objNr)
2352 if err == nil {
2353     return err
2354 }
2355 // // handle stream dict's
2356
2357 if _, ok = o.(ObjectStreamDict); ok {
2358     // Handle errors from 'dereferencedObject: object stream should already be
2359     // dereferenced at objId', objNr'
2360     {
2361         if _, ok = o.(ObjectStreamDict); ok {
2362             // return errors.Errorf("dereferencedObject: xref stream should already be
2363             // dereferenced at objId", objNr)
2364         }
2365     }
2366     if _, ok = o.(ObjectStreamDict); ok {
2367         // return errors.Errorf("dereferencedObject: xref stream should already be
2368         // dereferenced at objId", objNr)
2369     }
2370 }
2371 if sd, ok = o.(StreamDict); ok {
2372     //
2373     err = loadStream(ctxt, sd, objNr, entry.generation)
2374     if err == nil {
2375         return err
2376     }
2377 }
2378 entry.Object = sd
2379
2380 // log.Printf("dereferencedObject: and objId of %v to %v", objNr,
2381 // objId)
2382 // defer objId, entry.Object)
2383 // log.Printf("entry.Object)
2384 // return nil
2385 }
2386 }
2387
2388 func processBidsAndCounts(defTable *XRefTable, D Dict) {
2389     for _, n := range o {
2390         match o1 := x.Table()
2391         case IndexDict:
2392             entry, ok := defTable.LookupTableEntryAndIndex(defTable)
2393             if ok {
2394                 entry.Count++
2395             }
2396         case Dict:
2397             processBidsAndCounts(defTable, o1)
2398         case Array:
2399             processBidsAndCounts(defTable, o1)
2400         }
2401     }
2402 }

```

[illegible]

```

5332 return dc, nil
5333
5334
5335 func object(ctx *Context, offset, objAddr, groupIndex) (Object, error, streamOut,
5336 streamOffset int64, err error) {
5337
5338     var id int64
5339     id, err = readInt64FromHeader(ctx, &readSize, &offset)
5340     if err != nil {
5341         return nil, 0, 0, 0, err
5342     }
5343
5344     //Log.Read_Printf("object: seeked to %d(%d), offset=%d",
5345     //id, groupIndex, offset)
5346
5347     // process: s gen obj ... obj dict ... stream ... data ... endstream endobj
5348     // object ... stream ... stream ... stream ... stream ... stream ... stream ...
5349     var buf [1]byte
5350     if !ctx.readStream(streamOut, streamOffset, err + &buffer(&buf)) {
5351         if err == nil {
5352             return nil, 0, 0, 0, err
5353         }
5354     }
5355
5356     //Log.Read_Printf("stream: %d(%d) %d(%d) streamOffset: %d(%d) endObj: %d(%d)",
5357     //stream, streamOffset, streamOffset, streamOffset, endObj, endObj)
5358     //Log.Printf("streamOffset: %d(%d), endObj: %d(%d)", stream,
5359     //streamOffset, endObj, endObj)
5360
5361     line := string(buf)
5362
5363     var t string
5364
5365     if !bufio & dc { // if streamOut & s, streamOffset
5366         if !bufio & gen obj ... obj dict ... stream ... data
5367             // implies we detected no endobj and a stream starting at streamOffset
5368             // big stream, we parse object until 'stream'
5369             logRead.Printf("object: big stream, we parse object until 'stream'")
5370             t = line
5371         } else if !streamOut & dc { // if dc
5372             if !bufio & gen obj ... obj dict ... stream ... data
5373                 // implies we detected endobj and no stream
5374                 // small stream, we parse object until 'endobj'
5375                 logRead.Printf("object: small object w/o stream, parse until endobj")
5376                 t = line
5377             } else if !streamOut & !dc { // if streamOffset
5378                 if !bufio & gen obj ... obj dict ... stream ... data ... endstream endobj
5379                     // implies we detected endobj and a stream
5380                     // small stream within buffer, parse until 'stream'
5381                     logRead.Printf("object: small stream within buffer, parse until 'stream'")
5382                     t = line
5383                 } else { // if dc
5384                     // implies we detected endobj and a stream
5385                     // small obj w/o stream, parse until 'endobj'
5386                     logRead.Printf("object: small object w/o stream, parse until 'endobj'")
5387                     t = line
5388                 }
5389             } else { // if streamOffset
5390                 // parse object: number and object generation,
5391                 // var objectID, generationID int
5392

```

[illegible]

```

1968 // Use the object stream directly for object stream reads.
1969 // If !sd, !isObject()
1970     return errors.New("pdcps: decodeObjectStream: corrupt object stream")
1971 }
1972
1973 // We have an object stream.
1974 // If !sd, err = objectStreamDict(svd)
1975 // If err == nil
1976     return errors.Wrap(err, "decodeObjectStream: problem dereferencing
1977 object stream svd", objectStream)
1978
1979 // Log Read, Print("decodeObjectStream: decoding object stream SvId",
1980 // objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict. objArray.
1984 // If err = readObjectStreamDict(svd) err == nil {
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding
1986 object stream svd", objectStream)
1987 }
1988
1989 // If svd objArray == nil
1990     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1991 }
1992
1993 // Log Read, Print("decodeObjectStream: decoded object stream SvId",
1994 // objectStream)
1995
1996 // Save object stream dict to sHeaderEntry.
1997     entry.Object = svd
1998
1999     Log, Read, Println("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) ok & d is LinearizationPanicDict {
2011         Log, Read, linearized := true
2012         cxt.LinearizationPanicDict[objIn] = true
2013         Log, Read, Println("handleLinearizationPanicDict: identified LinearizationObj
2014 SvId", c)
2015
2016         a := d.ArrayEntry("sr")
2017
2018

```

```

2020: // 2020
2021: func processArrayByCounts(x:Iterable, y:Iterable, a:Array) {
2022:     for _ in range a {
2023:         switch o in a.toArray() {
2024:             case IndexOf:
2025:                 entry, ok = x.elementAt(IndexOf)
2026:                 if ok {
2027:                     entry, b = Count++
2028:                 }
2029:             case Count:
2030:                 processByCounts(x, y, a, 0)
2031:             case Array:
2032:                 processByCounts(x, y, a, 0)
2033:             }
2034:         }
2035:     }
2036: }
2037:
2038: func processByCounts(x:Iterable, y:Iterable, o:Object) {
2039:     switch o in o.toArray() {
2040:         case Count:
2041:             processByCounts(x, y, a, 0)
2042:         case Stream:
2043:             processByCounts(x, y, a, 0)
2044:         case Array:
2045:             processArrayByCounts(x, y, a, 0)
2046:         }
2047:     }
2048: }
2049:
2050: // 2050
2051: // 2051
2052: // 2052
2053: // 2053
2054: // 2054
2055: // 2055
2056: // 2056
2057: // 2057
2058: // 2058
2059: // 2059
2060: // 2060
2061: // 2061
2062: // 2062
2063: // 2063
2064: // 2064
2065: // 2065
2066: // 2066
2067: // 2067
2068: // 2068
2069: // 2069
2070: // 2070
2071: // 2071
2072: // 2072
2073: // 2073
2074: // 2074
2075: // 2075
2076: // 2076
2077: // 2077
2078: // 2078
2079: // 2079
2080: // 2080
2081: // 2081
2082: // 2082
2083: // 2083
2084: // 2084
2085: // 2085
2086: // 2086
2087: // 2087
2088: // 2088
2089: // 2089
2090: // 2090
2091: // 2091
2092: // 2092
2093: // 2093
2094: // 2094
2095: // 2095
2096: // 2096
2097: // 2097
2098: // 2098
2099: // 2099
2100: // 2100
2101: // 2101
2102: // 2102
2103: // 2103
2104: // 2104
2105: // 2105
2106: // 2106
2107: // 2107
2108: // 2108
2109: // 2109
2110: // 2110
2111: // 2111
2112: // 2112
2113: // 2113
2114: // 2114
2115: // 2115
2116: // 2116
2117: // 2117
2118: // 2118
2119: // 2119
2120: // 2120
2121: // 2121
2122: // 2122
2123: // 2123
2124: // 2124
2125: // 2125
2126: // 2126
2127: // 2127
2128: // 2128
2129: // 2129
2130: // 2130
2131: // 2131
2132: // 2132
2133: // 2133
2134: // 2134
2135: // 2135
2136: // 2136
2137: // 2137
2138: // 2138
2139: // 2139
2140: // 2140
2141: // 2141
2142: // 2142
2143: // 2143
2144: // 2144
2145: // 2145
2146: // 2146
2147: // 2147
2148: // 2148
2149: // 2149
2150: // 2150
2151: // 2151
2152: // 2152
2153: // 2153
2154: // 2154
2155: // 2155
2156: // 2156
2157: // 2157
2158: // 2158
2159: // 2159
2160: // 2160
2161: // 2161
2162: // 2162
2163: // 2163
2164: // 2164
2165: // 2165
2166: // 2166
2167: // 2167
2168: // 2168
2169: // 2169
2170: // 2170
2171: // 2171
2172: // 2172
2173: // 2173
2174: // 2174
2175: // 2175
2176: // 2176
2177: // 2177
2178: // 2178
2179: // 2179
2180: // 2180
2181: // 2181
2182: // 2182
2183: // 2183
2184: // 2184
2185: // 2185
2186: // 2186
2187: // 2187
2188: // 2188
2189: // 2189
2190: // 2190
2191: // 2191
2192: // 2192
2193: // 2193
2194: // 2194
2195: // 2195
2196: // 2196
2197: // 2197
2198: // 2198
2199: // 2199
2200: // 2200
2201: // 2201
2202: // 2202
2203: // 2203
2204: // 2204
2205: // 2205
2206: // 2206
2207: // 2207
2208: // 2208
2209: // 2209
2210: // 2210
2211: // 2211
2212: // 2212
2213: // 2213
2214: // 2214
2215: // 2215
2216: // 2216
2217: // 2217
2218: // 2218
2219: // 2219
2220: // 2220
2221: // 2221
2222: // 2222
2223: // 2223
2224: // 2224
2225: // 2225
2226: // 2226
2227: // 2227
2228: // 2228
2229: // 2229
2230: // 2230
2231: // 2231
2232: // 2232
2233: // 2233
2234: // 2234
2235: // 2235
2236: // 2236
2237: // 2237
2238: // 2238
2239: // 2239
2240: // 2240
2241: // 2241
2242: // 2242
2243: // 2243
2244: // 2244
2245: // 2245
2246: // 2246
2247: // 2247
2248: // 2248
2249: // 2249
2250: // 2250
2251: // 2251
2252: // 2252
2253: // 2253
2254: // 2254
2255: // 2255
2256: // 2256
2257: // 2257
2258: // 2258
2259: // 2259
2260: // 2260
2261: // 2261
2262: // 2262
2263: // 2263
2264: // 2264
2265: // 2265
2266: // 2266
2267: // 2267
2268: // 2268
2269: // 2269
2270: // 2270
2271: // 2271
2272: // 2272
2273: // 2273
2274: // 2274
2275: // 2275
2276: // 2276
2277: // 2277
2278: // 2278
2279: // 2279
2280: // 2280
2281: // 2281
2282: // 2282
2283: // 2283
2284: // 2284
2285: // 2285
2286: // 2286
2287: // 2287
2288: // 2288
2289: // 2289
2290: // 2290
2291: // 2291
2292: // 2292
2293: // 2293
2294: // 2294
2295: // 2295
2296: // 2296
2297: // 2297
2298: // 2298
2299: // 2299
2300: // 2300
2301: // 2301
2302: // 2302
2303: // 2303
2304: // 2304
2305: // 2305
2306: // 2306
2307: // 2307
2308: // 2308
2309: // 2309
2310: // 2310
2311: // 2311
2312: // 2312
2313: // 2313
2314: // 2314
2315: // 2315
2316: // 2316
2317: // 2317
2318: // 2318
2319: // 2319
2320: // 2320
2321: // 2321
2322: // 2322
2323: // 2323
2324: // 2324
2325: // 2325
2326: // 2326
2327: // 2327
2328: // 2328
2329: // 2329
2330: // 2330
2331: // 2331
2332: // 2332
2333: // 2333
2334: // 2334
2335: // 2335
2336: // 2336
2337: // 2337
2338: // 2338
2339: // 2339
2340: // 2340
2341: // 2341
2342: // 2342
2343: // 2343
2344: // 2344
2345: // 2345
2346: // 2346
2347: // 2347
2348: // 2348
2349: // 2349
2350: // 2350
2351: // 2351
2352: // 2352
2353: // 2353
2354: // 2354
2355: // 2355
2356: // 2356
2357: // 2357
2358: // 2358
2359: // 2359
2360
```

```

2530 //
2531 if err == nil {
2532     return err
2533 }
2534 //
2535 // If the owner password does not match we generally move on if the user password
2536 // errors.
2537 //
2538 // Unless we need to limit on a user's owner password due to the specific
2539 // amount in use password.
2540 if tok != newPasswordHandler(password,ctx.Cmd) {
2541     return errors.New("password: please provide the master password with 'opw'")
2542 }
2543 //
2544 //
2545 // Generally the user password, which is also regarded as the master password or
2546 // pre-password.
2547 //
2548 // It is sufficient for moving on. A password change is an occasion since it
2549 // is not.
2550 if ok := newPasswordHandler(password,ctx.Cmd) {
2551     return nil
2552 }
2553 //
2554 // ok,err := validatePermissions(ctx)
2555 //
2556 if err == nil {
2557     return err
2558 }
2559 //
2560 // If ok {
2561 //
2562 //     return errors.New("password: corrupted permissions after opw ok")
2563 //
2564 // }
2565 //
2566 // return nil
2567 //
2568 //
2569 // Validate the user password ok, document opw password.
2570 ok,err := validatePermissions(ctx)
2571 if err == nil {
2572     return err
2573 }
2574 //
2575 // If ok {
2576 //
2577 //     return errors.New("password: please provide the correct password")
2578 //
2579 // }
2580 //
2581 //
2582 // //nc.Print("Type ok: %d\n", ok)
2583 //
2584 // return handlePermissions(ctx)
2585 //
2586 //
2587 //
2588 //
2589 //
2590 //
2591 //
2592 //
2593 //
2594 //
2595 //
2596 //
2597 //
2598 //
2599 //
2600 //
2601 //
2602 //
2603 //
2604 //
2605 //
2606 //
2607 //
2608 //
2609 //
2610 //
2611 //
2612 //
2613 //
2614 //
2615 //
2616 //
2617 //
2618 //
2619 //
2620 //
2621 //
2622 //
2623 //
2624 //
2625 //
2626 //
2627 //
2628 //
2629 //
2630 //
2631 //
2632 //
2633 //
2634 //
2635 //
2636 //
2637 //
2638 //
2639 //
2640 //
2641 //
2642 //
2643 //
2644 //
2645 //
2646 //
2647 //
2648 //
2649 //
2650 //
2651 //
2652 //
2653 //
2654 //
2655 //
2656 //
2657 //
2658 //
2659 //
2660 //
2661 //
2662 //
2663 //
2664 //
2665 //
2666 //
2667 //
2668 //
2669 //
2670 //
2671 //
2672 //
2673 //
2674 //
2675 //
2676 //
2677 //
2678 //
2679 //
2680 //
2681 //
2682 //
2683 //
2684 //
2685 //
2686 //
2687 //
2688 //
2689 //
2690 //
2691 //
2692 //
2693 //
2694 //
2695 //
2696 //
2697 //
2698 //
2699 //
2700 //
2701 //
2702 //
2703 //
2704 //
2705 //
2706 //
2707 //
2708 //
2709 //
2710 //
2711 //
2712 //
2713 //
2714 //
2715 //
2716 //
2717 //
2718 //
2719 //
2720 //
2721 //
2722 //
2723 //
2724 //
2725 //
2726 //
2727 //
2728 //
2729 //
2730 //
2731 //
2732 //
2733 //
2734 //
2735 //
2736 //
2737 //
2738 //
2739 //
2740 //
2741 //
2742 //
2743 //
2744 //
2745 //
2746 //
2747 //
2748 //
2749 //
2750 //
2751 //
2752 //
2753 //
2754 //
2755 //
2756 //
2757 //
2758 //
2759 //
2760 //
2761 //
2762 //
2763 //
2764 //
2765 //
2766 //
2767 //
2768 //
2769 //
2770 //
2771 //
2772 //
2773 //
2774 //
2775 //
2776 //
2777 //
2778 //
2779 //
2780 //
2781 //
2782 //
2783 //
2784 //
2785 //
2786 //
2787 //
2788 //
2789 //
2790 //
2791 //
2792 //
2793 //
2794 //
2795 //
2796 //
2797 //
2798 //
2799 //
2800 //
2801 //
2802 //
2803 //
2804 //
2805 //
2806 //
2807 //
2808 //
2809 //
2810 //
2811 //
2812 //
2813 //
2814 //
2815 //
2816 //
2817 //
2818 //
2819 //
2820 //
2821 //
2822 //
2823 //
2824 //
2825 //
2826 //
2827 //
2828 //
2829 //
2830 //
2831 //
2832 //
2833 //
2834 //
2835 //
2836 //
2837 //
2838 //
2839 //
2840 //
2841 //
2842 //
2843 //
2844 //
2845 //
2846 //
2847 //
2848 //
2849 //
2850 //
2851 //
2852 //
2853 //
2854 //
2855 //
2856 //
2857 //
2858 //
2859 //
2860 //
2861 //
2862 //
2863 //
2864 //
2865 //
2866 //
2867 //
2868 //
2869 //
2870 //
2871 //
2872 //
2873 //
2874 //
2875 //
2876 //
2877 //
2878 //
2879 //
2880 //
2881 //
2882 //
2883 //
2884 //
2885 //
2886 //
2887 //
2888 //
2889 //
2890 //
2891 //
2892 //
2893 //
2894 //
2895 //
2896 //
2897 //
2898 //
2899 //
2900 //
2901 //
2902 //
2903 //
2904 //
2905 //
2906 //
2907 //
2908 //
2909 //
2910 //
2911 //
2912 //
2913 //
2914 //
2915 //
2916 //
2917 //
2918 //
2919 //
2920 //
2921 //
2922 //
2923 //
2924 //
2925 //
2926 //
2927 //
2928 //
2929 //
2930 //
2931 //
2932 //
2933 //
2934 //
2935 //
2936 //
2937 //
2938 //
2939 //
2940 //
2941 //
2942 //
2943 //
2944 //
2945 //
2946 //
2947 //
2948 //
2949 //
2950 //
2951 //
2952 //
2953 //
2954 //
2955 //
2956 //
2957 //
2958 //
2959 //
2960 //
2961 //
2962 //
2963 //
2964 //
2965 //
2966 //
2967 //
2968 //
2969 //
2970 //
2971 //
2972 //
2973 //
2974 //
2975 //
2976 //
2977 //
2978 //
2979 //
2980 //
2981 //
2982 //
2983 //
2984 //
2985 //
2986 //
2987 //
2988 //
2989 //
2990 //
2991 //
2992 //
2993 //
2994 //
2995 //
2996 //
2997 //
2998 //
2999 //
3000 //
3001 //
3002 //
3003 //
3004 //
3005 //
3006 //
3007 //
3008 //
3009 //
3010 //
3011 //
3012 //
3013 //
3014 //
3015 //
3016 //
3017 //
3018 //
3019 //
3020 //
3021 //
3022 //
3023 //
3024 //
3025 //
3026 //
3027 //
3028 //
3029 //
3030 //
3031 //
3032 //
3033 //
3034 //
3035 //
3036 //
3037 //
3038 //
3039 //
3040 //
3041 //
3042 //
3043 //
3044 //
3045 //
3046 //
3047 //
3048 //
3049 //
3050 //
3051 //
3052 //
3053 //
3054 //
3055 //
3056 //
3057 //
3058 //
3059 //
3060 //
3061 //
3062 //
3063 //
3064 //
3065 //
3066 //
3067 //
3068 //
3069 //
3070 //
3071 //
3072 //
3073 //
3074 //
3075 //
3076 //
3077 //
3078 //
3079 //
3080 //
3081 //
3082 //
3083 //
3084 //
3085 //
3086 //
3087 //
3088 //
3089 //
3090 //
3091 //
3092 //
3093 //
3094 //
3095 //
3096 //
3097 //
3098 //
3099 //
3100 //
3101 //
3102 //
3103 //
3104 //
3105 //
3106 //
3107 //
3108 //
3109 //
3110 //
3111 //
3112 //
3113 //
3114 //
3115 //
3116 //
3117 //
3118 //
3119 //
3120 //
3121 //
3122 //
3123 //
3124 //
3125 //
3126 //
3127 //
3128 //
3129 //
3130 //
3131 //
3132 //
3133 //
3134 //
3135 //
3136 //
3137 //
3138 //
3139 //
3140 //
3141 //
3142 //
3143 //
3144 //
3145 //
3146 //
3147 //
3148 //
3149 //
3150 //
3151 //
3152 //
3153 //
3154 //
3155 //
3156 //
3157 //
3158 //
3159 //
3160 //
3161 //
3162 //
3163 //
3164 //
3165 //
3166 //
3167 //
3168 //
3169 //
3170 //
3171 //
3172 //
3173 //
3174 //
3175 //
3176 //
317
```