

[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         a: 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.a =
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 (object instanceof Info) {
658             return entries.New<Info>(parametersInfo: missing entry, "Root(" +
659                 object.ToString() + ")");
660         }
661         if (object is Root) {
662             return entries.New<Root>(parametersInfo: Root object: Null, "Root");
663         }
664         if (object is Info) {
665             return entries.New<Info>(parametersInfo: Info object: Null, "Info");
666         }
667         if (object is ID) {
668             return entries.New<ID>(parametersInfo: ID object: Null, "ID");
669         }
670         if (object is Entry) {
671             return entries.New<Entry>(parametersInfo: Entry object: Null, "Entry");
672         }
673         if (object is MissingEntry) {
674             return entries.New<MissingEntry>(parametersInfo: missing entry "ID");
675         }
676         log.Debug.Print("parametersInfo end");
677     }
678     return nil
679 }
680
681 func parametersInfo(trailerDict Dict, ctx *Context) (*Info, error) {
682     log.Debug.Print("parametersInfo begin")
683     ifHeader := ctx.IfHeader
684     err := parametersInfoOf(trailerDict, ifHeader)
685     if err == nil {
686         return nil, err
687     }
688     if err := trailerDict.ArraysEntry("AdditionalStreams") are == nil {
689         return parametersInfoOf(parametersInfo: found AdditionalStreams "Null", arr
690             := arr["AdditionalStreams"],
691             for := value := range arr {
692                 if !index {
693                     a = append(a, index)
694                 }
695             }
696         ifHeader.AdditionalStreams = &a
697     }
698     offset := trailerDict.Pre()
699     if offset == nil {
700         return parametersInfoOf(parametersInfo: previous stream table section offset "Null",
701             offset)
702     }
703     offsetOfStream := trailerDict.IndexEntry("Offset")

```

[illegible]

```

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.readBuilderStreams = true
1433         rd, err = newPositionedReader(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 of cross reference table will call this func.
1451 // Can be "ref" or indirect object reference e.g. "34 0 obj"
1452 // If no indirect object definition as long as there is a defined previous cross section
1453 // and build up the cross table along the way.
1454 func reinterTable(c *Context) (err error) {

```

```

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

```

```

2820 // Process all entries of this subsection into xSubTable entries.
2821 // Process all entries of this subsection into xSubTable entries.
2822 func parseCompressedTableSubSection(x *bufio.Scanner, xSubTable *xSubTable, fields []string) error {
2823     log.Read.Println("parseCompressedTableSubSection: begin")
2824     startOfNumber, err := strconv.Atoi(fields[0])
2825     if err == nil {
2826         return err
2827     }
2828     objCount, err := strconv.Atoi(fields[1])
2829     if err == nil {
2830         return err
2831     }
2832     log.Read.Printf("detected xref subsection, startOfObj=%d length=%d\n",
2833         startOfNumber, objCount)
2834     // Process all entries of this subsection into xSubTable entries.
2835     for i := 0; i < objCount; i++ {
2836         if err := parseObjTableEntry(x, xSubTable, startOfNumber); err == nil {
2837             return err
2838         }
2839     }
2840     log.Read.Println("parseCompressedTableSubSection: end")
2841     return nil
2842 }
2843 // Parse compressed object.
2844 func (p *PDFParser) parseCompressedObject(Object, error) {
2845     log.Read.Println("compressedObject: begin")
2846     o, err := parseObject(p)
2847     if err == nil {
2848         return nil, err
2849     }
2850     if ok := o.IsDict();
2851     if ok {
2852         // Return trivial object: Integer, Array, etc.
2853         log.Read.Println("compressedObject: end, any other than dict")
2854         return o, nil
2855     }
2856     streamLength, streamOffset := p.Length()
2857     if streamLength == nil || streamOffset == nil {
2858         // Return dict.
2859         log.Read.Println("compressedObject: end, dict")
2860         return o, nil
2861     }
2862     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
2863         stored in an object stream")
2864 }

```

```

587 // @throws IOException, UnsupportedOperationException
588 | else {
589 |     ctx.tableId(objectNumber) = BaseTableEntry
590 | }
591 | }
592 | }
593 | }
594 | }
595 | }
596 | }
597 | }
598 | }
599 | }
600 | }
601 | }
602 | }
603 | }
604 | }
605 | }
606 | }
607 | }
608 | }
609 | }
610 | }
611 | }
612 | }
613 | }
614 | }
615 | }
616 | }
617 | }
618 | }
619 | }
620 | }
621 | }
622 | }
623 | }
624 | }
625 | }
626 | }
627 | }
628 | }
629 | }
630 | }
631 | }
632 | }
633 | }
634 | }
635 | }
636 | }
637 | }
638 | }
639 | }
640 | }
641 | }
642 | }
643 | }
644 | }
645 | }
646 | }
647 | }
648 | }
649 | }
650 | }
651 | }
652 | }
653 | }
654 | }
655 | }
656 | }
657 | }
658 | }
659 | }
660 | }
661 | }
662 | }
663 | }
664 | }
665 | }
666 | }
667 | }
668 | }
669 | }
670 | }
671 | }
672 | }
673 | }
674 | }
675 | }
676 | }
677 | }
678 | }
679 | }
680 | }
681 | }
682 | }
683 | }
684 | }
685 | }
686 | }
687 | }
688 | }
689 | }
690 | }
691 | }
692 | }
693 | }
694 | }
695 | }
696 | }
697 | }
698 | }
699 | }
700 | }
701 | }
702 | }
703 | }
704 | }
705 | }
706 | }
707 | }
708 | }
709 | }
710 | }
711 | }
712 | }
713 | }
714 | }
715 | }
716 | }
717 | }
718 | }
719 | }
720 | }
721 | }
722 | }
723 | }
724 | }
725 | }
726 | }
727 | }
728 | }
729 | }
730 | }
731 | }
732 | }
733 | }
734 | }
735 | }
736 | }
737 | }
738 | }
739 | }
740 | }
741 | }
742 | }
743 | }
744 | }
745 | }
746 | }
747 | }
748 | }
749 | }
750 | }
751 | }
752 | }
753 | }
754 | }
755 | }
756 | }
757 | }
758 | }
759 | }
760 | }
761 | }
762 | }
763 | }
764 | }
765 | }
766 | }
767 | }
768 | }
769 | }
770 | }
771 | }
772 | }
773 | }
774 | }
775 | }
776 | }
777 | }
778 | }
779 | }
780 | }
781 | }
782 | }
783 | }
784 | }
785 | }
786 | }
787 | }
788 | }
789 | }
790 | }
791 | }
792 | }
793 | }
794 | }
795 | }
796 | }
797 | }
798 | }
799 | }
800 | }
801 | }
802 | }
803 | }
804 | }
805 | }
806 | }
807 | }
808 | }
809 | }
810 | }
811 | }
812 | }
813 | }
814 | }
815 | }
816 | }
817 | }
818 | }
819 | }
820 | }
821 | }
822 | }
823 | }
824 | }
825 | }
826 | }
827 | }
828 | }
829 | }
830 | }
831 | }
832 | }
833 | }
834 | }
835 | }
836 | }
837 | }
838 | }
839 | }
840 | }
841 | }
842 | }
843 | }
844 | }
845 | }
846 | }
847 | }
848 | }
849 | }
850 | }
851 | }
852 | }
853 | }
854 | }
855 | }
856 | }
857 | }
858 | }
859 | }
860 | }
861 | }
862 | }
863 | }
864 | }
865 | }
866 | }
867 | }
868 | }
869 | }
870 | }
871 | }
872 | }
873 | }
874 | }
875 | }
876 | }
877 | }
878 | }
879 | }
880 | }
881 | }
882 | }
883 | }
884 | }
885 | }
886 | }
887 | }
888 | }
889 | }
890 | }
891 | }
892 | }
893 | }
894 | }
895 | }
896 | }
897 | }
898 | }
899 | }
900 | }
901 | }
902 | }
903 | }
904 | }
905 | }
906 | }
907 | }
908 | }
909 | }
910 | }
911 | }
912 | }
913 | }
914 | }
915 | }
916 | }
917 | }
918 | }
919 | }
920 | }
921 | }
922 | }
923 | }
924 | }
925 | }
926 | }
927 | }
928 | }
929 | }
930 | }
931 | }
932 | }
933 | }
934 | }
935 | }
936 | }
937 | }
938 | }
939 | }
940 | }
941 | }
942 | }
943 | }
944 | }
945 | }
946 | }
947 | }
948 | }
949 | }
950 | }
951 | }
952 | }
953 | }
954 | }
955 | }
956 | }
957 | }
958 | }
959 | }
960 | }
961 | }
962 | }
963 | }
964 | }
965 | }
966 | }
967 | }
968 | }
969 | }
970 | }
971 | }
972 | }
973 | }
974 | }
975 | }
976 | }
977 | }
978 | }
979 | }
980 | }
981 | }
982 | }
983 | }
984 | }
985 | }
986 | }
987 | }
988 | }
989 | }
990 | }
991 | }
992 | }
993 | }
994 | }
995 | }
996 | }
997 | }
998 | }
999 | }
1000 | }

```

[illegible]

```

5820         files = strings.Join(line)
5821     }
5822     log.Printf("parseSection: All subsection read")
5823 }
5824 // Parse the trailer section
5825 func (p *PDF) parseTrailer() error {
5826     if !strings.HasPrefix(trailer, "trailer") {
5827         return nil, errors.Errorf("section missing trailer dict, line = %d",
5828             p.currentLine)
5829     }
5830     log.Printf("parseSection: parsing trailer dict.")
5831     return processTrailerDict(p, line)
5832 }
5833 // Get version from first line of file
5834 func (p *PDF) getPDFVersion() (int, error) {
5835     // Beginning with PDF 1.0, the version entry in the document's catalog dictionary
5836     // is the first entry in the file's trailer as described in 8.5.3, "File
5837     // format".
5838     // The version, shall be used instead of the version specified in the header.
5839     // Save PDF version from header to offsetFile.
5840     // If the version is not found, return the version from the trailer.
5841     // If the version is the number of characters used for file (1 to 3).
5842     // If the version is not found, return the version from the trailer.
5843     headerVersion := p.getHeaderVersion()
5844     if headerVersion != 0 {
5845         return headerVersion, nil
5846     }
5847     log.Printf("headerVersion is empty")
5848     // Get the version from the trailer
5849     err := p.parseTrailer()
5850     if err != nil {
5851         return 0, errors.Errorf("headerVersion: corrupt pdf stream = %v",
5852             err)
5853     }
5854     // Get first line of file which holds the version of this PDF file.
5855     // We call this the header version.
5856     // If we find a %EOF, %EOFstart, err = nil
5857     // If we find a %EOF, err = nil
5858     // If we find a %EOF, err = nil
5859     // If we find a %EOF, err = nil
5860     // If we find a %EOF, err = nil
5861     // If we find a %EOF, err = nil
5862     // If we find a %EOF, err = nil
5863     // If we find a %EOF, err = nil
5864     // If we find a %EOF, err = nil
5865     // If we find a %EOF, err = nil
5866     // If we find a %EOF, err = nil
5867     // If we find a %EOF, err = nil
5868     // If we find a %EOF, err = nil
5869     // If we find a %EOF, err = nil
5870     // If we find a %EOF, err = nil
5871     // If we find a %EOF, err = nil
5872     // If we find a %EOF, err = nil
5873     // If we find a %EOF, err = nil
5874     // If we find a %EOF, err = nil
5875     // If we find a %EOF, err = nil
5876     // If we find a %EOF, err = nil
5877     // If we find a %EOF, err = nil
5878     // If we find a %EOF, err = nil
5879     // If we find a %EOF, err = nil
5880     // If we find a %EOF, err = nil
5881     // If we find a %EOF, err = nil
5882     // If we find a %EOF, err = nil
5883     // If we find a %EOF, err = nil
5884     // If we find a %EOF, err = nil
5885     // If we find a %EOF, err = nil
5886     // If we find a %EOF, err = nil
5887     // If we find a %EOF, err = nil
5888     // If we find a %EOF, err = nil
5889     // If we find a %EOF, err = nil
5890     // If we find a %EOF, err = nil
5891     // If we find a %EOF, err = nil
5892     // If we find a %EOF, err = nil
5893     // If we find a %EOF, err = nil
5894     // If we find a %EOF, err = nil
5895     // If we find a %EOF, err = nil
5896     // If we find a %EOF, err = nil
5897     // If we find a %EOF, err = nil
5898     // If we find a %EOF, err = nil
5899     // If we find a %EOF, err = nil
5900     // If we find a %EOF, err = nil
5901     // If we find a %EOF, err = nil
5902     // If we find a %EOF, err = nil
5903     // If we find a %EOF, err = nil
5904     // If we find a %EOF, err = nil
5905     // If we find a %EOF, err = nil
5906     // If we find a %EOF, err = nil
5907     // If we find a %EOF, err = nil
5908     // If we find a %EOF, err = nil
5909     // If we find a %EOF, err = nil
5910     // If we find a %EOF, err = nil
5911     // If we find a %EOF, err = nil
5912     // If we find a %EOF, err = nil
5913     // If we find a %EOF, err = nil
5914     // If we find a %EOF, err = nil
5915     // If we find a %EOF, err = nil
5916     // If we find a %EOF, err = nil
5917     // If we find a %EOF, err = nil
5918     // If we find a %EOF, err = nil
5919     // If we find a %EOF, err = nil
5920     // If we find a %EOF, err = nil
5921     // If we find a %EOF, err = nil
5922     // If we find a %EOF, err = nil
5923     // If we find a %EOF, err = nil
5924     // If we find a %EOF, err = nil
5925     // If we find a %EOF, err = nil
5926     // If we find a %EOF, err = nil
5927     // If we find a %EOF, err = nil
5928     // If we find a %EOF, err = nil
5929     // If we find a %EOF, err = nil
5930     // If we find a %EOF, err = nil
5931     // If we find a %EOF, err = nil
5932     // If we find a %EOF, err = nil
5933     // If we find a %EOF, err = nil
5934     // If we find a %EOF, err = nil
5935     // If we find a %EOF, err = nil
5936     // If we find a %EOF, err = nil
5937     // If we find a %EOF, err = nil
5938     // If we find a %EOF, err = nil
5939     // If we find a %EOF, err = nil
5940     // If we find a %EOF, err = nil
5941     // If we find a %EOF, err = nil
5942     // If we find a %EOF, err = nil
5943     // If we find a %EOF, err = nil
5944     // If we find a %EOF, err = nil
5945     // If we find a %EOF, err = nil
5946     // If we find a %EOF, err = nil
5947     // If we find a %EOF, err = nil
5948     // If we find a %EOF, err = nil
5949     // If we find a %EOF, err = nil
5950     // If we find a %EOF, err = nil
5951     // If we find a %EOF, err = nil
5952     // If we find a %EOF, err = nil
5953     // If we find a %EOF, err = nil
5954     // If we find a %EOF, err = nil
5955     // If we find a %EOF, err = nil
5956     // If we find a %EOF, err = nil
5957     // If we find a %EOF, err = nil
5958     // If we find a %EOF, err = nil
5959     // If we find a %EOF, err = nil
5960     // If we find a %EOF, err = nil
5961     // If we find a %EOF, err = nil
5962     // If we find a %EOF, err = nil
5963     // If we find a %EOF, err = nil
5964     // If we find a %EOF, err = nil
5965     // If we find a %EOF, err = nil
5966     // If we find a %EOF, err = nil
5967     // If we find a %EOF, err = nil
5968     // If we find a %EOF, err = nil
5969     // If we find a %EOF, err = nil
5970     // If we find a %EOF, err = nil
5971     // If we find a %EOF, err = nil
5972     // If we find a %EOF, err = nil
5973     // If we find a %EOF, err = nil
5974     // If we find a %EOF, err = nil
5975     // If we find a %EOF, err = nil
5976     // If we find a %EOF, err = nil
5977     // If we find a %EOF, err = nil
5978     // If we find a %EOF, err = nil
5979     // If we find a %EOF, err = nil
5980     // If we find a %EOF, err = nil
5981     // If we find a %EOF, err = nil
5982     // If we find a %EOF, err = nil
5983     // If we find a %EOF, err = nil
5984     // If we find a %EOF, err = nil
5985     // If we find a %EOF, err = nil
5986     // If we find a %EOF, err = nil
5987     // If we find a %EOF, err = nil
5988     // If we find a %EOF, err = nil
5989     // If we find a %EOF, err = nil
5990     // If we find a %EOF, err = nil
5991     // If we find a %EOF, err = nil
5992     // If we find a %EOF, err = nil
5993     // If we find a %EOF, err = nil
5994     // If we find a %EOF, err = nil
5995     // If we find a %EOF, err = nil
5996     // If we find a %EOF, err = nil
5997     // If we find a %EOF, err = nil
5998     // If we find a %EOF, err = nil
5999     // If we find a %EOF, err = nil
6000     // If we find a %EOF, err = nil
6001     // If we find a %EOF, err = nil
6002     // If we find a %EOF, err = nil
6003     // If we find a %EOF, err = nil
6004     // If we find a %EOF, err = nil
6005     // If we find a %EOF, err = nil
6006     // If we find a %EOF, err = nil
6007     // If we find a %EOF, err = nil
6008     // If we find a %EOF, err = nil
6009     // If we find a %EOF, err = nil
6010     // If we find a %EOF, err = nil
6011     // If we find a %EOF, err = nil
6012     // If we find a %EOF, err = nil
6013     // If we find a %EOF, err = nil
6014     // If we find a %EOF, err = nil
6015     // If we find a %EOF, err = nil
6016     // If we find a %EOF, err = nil
6017     // If we find a %EOF, err = nil
6018     // If we find a %EOF, err = nil
6019     // If we find a %EOF, err = nil
6020     // If we find a %EOF, err = nil
6021     // If we find a %EOF, err = nil
6022     // If we find a %EOF, err = nil
6023     // If we find a %EOF, err = nil
6024     // If we find a %EOF, err = nil
6025     // If we find a %EOF, err = nil
6026     // If we find a %EOF, err = nil
6027     // If we find a %EOF, err = nil
6028     // If we find a %EOF, err = nil
6029     // If we find a %EOF, err = nil
6030     // If we find a %EOF, err = nil
6031     // If we find a %EOF, err = nil
6032     // If we find a %EOF, err = nil
6033     // If we find a %EOF, err = nil
6034     // If we find a %EOF, err = nil
6035     // If we find a %EOF, err = nil
6036     // If we find a %EOF, err = nil
6037     // If we find a %EOF, err = nil
6038     // If we find a %EOF, err = nil
6039     // If we find a %EOF, err = nil
6040     // If we find a %EOF, err = nil
6041     // If we find a %EOF, err = nil
6042     // If we find a %EOF, err = nil
6043     // If we find a %EOF, err = nil
6044     // If we find a %EOF, err = nil
6045     // If we find a %EOF, err = nil
6046     // If we find a %EOF, err = nil
6047     // If we find a %EOF, err = nil
6048     // If we find a %EOF, err = nil
6049     // If we find a %EOF, err = nil
6050     // If we find a %EOF, err = nil
6051     // If we find a %EOF, err = nil
6052     // If we find a %EOF, err = nil
6053     // If we find a %EOF, err = nil
6054     // If we find a %EOF, err = nil
6055     // If we find a %EOF, err = nil
6056     // If we find a %EOF, err = nil
6057     // If we find a %EOF, err = nil
6058     // If we find a %EOF, err = nil
6059     // If we find a %EOF, err = nil
6060     // If we find a %EOF, err = nil
6061     // If we find a %EOF, err = nil
6062     // If we find a %EOF, err = nil
6063     // If we find a %EOF, err = nil
6064     // If we find a %EOF, err = nil
6065     // If
```

```

1197 log.Read.Println("readmeFile: begin")
1198
1199 // Read the file
1200 offset, err := offsetList.WhereSection(cta)
1201 if err != nil {
1202     return
1203 }
1204
1205 err = buildOffsetTableStartingAt(cta, offset)
1206 if err != doErr {
1207     return errors.Wrap(err, "readmeFile: unexpected eof")
1208 }
1209
1210 // If err == nil
1211 if err == nil {
1212     return
1213 }
1214
1215 // Log list of free objects (not the "free list").
1216 // /log.Read.Println("freeList: begin", cta.FreeObjects)
1217
1218 // Enumerate valid freeList of objects.
1219 err = enumerateValidFreeList()
1220 if err != nil {
1221     return
1222 }
1223
1224 log.Read.Println("readmeFile: end")
1225
1226 return
1227
1228 func growbody(buf []byte, size int, rd io.Reader) ([]byte, error) {
1229
1230     b := make([]byte, size)
1231
1232     // err := rd.Read(b)
1233     if err != nil {
1234         return nil, err
1235     }
1236     // log.Read.Println("growbody: Read %d bytes", n)
1237
1238     return append(buf, b...), nil
1239 }
1240
1241 func nextStreamOffset(list []int, streamid int) (off int) {
1242     off = streamid + len(stream)
1243 }
1244
1245 // Skip optional whitespaces
1246 // 7000 Should be skip optional whitespace instead
1247 for j := lineOff + 8240; off++ < 1
1248 }
1249
1250 // Skip 80 cols
1251 if lineOff == '\n' {
1252     off = 80
1253     return
1254 }
1255 }
1256 }
1257 // Skip 80 cols

```

```

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

```

```

3432 //
3433 // Return null if object is not an object stream and vice versa. Since obj is a java.io.ObjectStreamClass,
3434 // this method can only be called on object streams.
3435 func parseObjectStreamClass() error {
3436     log.Read.Print("parseObjectStreamClass begin: decoding %s\n", obj.ObjName)
3437
3438     decodedContent := obj.Content
3439     posLog := decodedContent.Content.FirstObjOffset
3440
3441     //
3442     // obj is strings.Fields(decodedContent)
3443     if len(obj) < 3 & & {
3444         return errors.New("parsing parseObjectStreamClass corrupt: object stream dict'")
3445     }
3446     // e.g., 10 0 11 25 + 2 Objects: 110 & offset 0, 112 & offset 25
3447
3448     var objArray Array
3449
3450     var offsetOld int
3451
3452     for i := 0; i < len(obj); i += 2 {
3453         offset, err := stream.Atol(obj[i+1])
3454         if err != nil {
3455             return err
3456         }
3457
3458         offset += obj.FirstObjOffset
3459
3460         if i % 2 == 1 {
3461             dict := string(decodedContent[offset:offset+obj[i]])
3462             log.Read.Print("parseObjectStreamClass: objStream = %s\n", dict)
3463             o, err := compressedDict.Dict(dict)
3464             if err != nil {
3465                 return err
3466             }
3467
3468             objArray = append(objArray, o)
3469         }
3470
3471         log.Read.Print("parseObjectStreamClass: %s = obj %s\n%s\n", i/2-1, obj[i], obj[i+1])
3472         objArray = append(objArray, o)
3473     }
3474
3475     if i % 2 == 1 {
3476         dict := string(decodedContent[offset:offset+obj[i]])
3477         log.Read.Print("parseObjectStreamClass: objStream = %s\n", dict)
3478         o, err := compressedDict.Dict(dict)
3479         if err != nil {
3480             return err
3481         }
3482
3483         log.Read.Print("parseObjectStreamClass: %s = obj %s\n%s\n", i/2, obj[i], obj[i+1])
3484         objArray = append(objArray, o)
3485     }
3486
3487     if offsetOld < offset {
3488         offset = offsetOld
3489     }
3490 }

```

```

545         return nil, err
546     }
547
548     log.Read_Parallel("parseStream: endOfNil=%d[%d1] streamEnd=%d[%d42] %v",
549         endOfNil, streamEnd)
550
551     // Line = string(buf)
552
553     // If object is a stream and therefore "stream" before "endOfNil" if "endOfNil" within
554     // there is no guarantee that "endOfNil" is contained in this buffer for large
555     // streams
556     if streamEnd < 0 || endOfNil < 0 || endOfNil < streamEnd {
557         return nil, errors.New("offset: parseStream: corrupt pdf file")
558     }
559
560     // nil object return buf
561     // nil object streamEnd
562     // object stream, generationNumber, err = parseObjectSubStreams(a)
563     if err == nil {
564         return nil, err
565     }
566
567     // parse nil object
568     log.Read_Parallel("parseStream: xrefObj objId genNum", objectNumber,
569         generationNumber)
570     // err = parseObject(a)
571     if err == nil {
572         return nil, err
573     }
574
575     // parse nil object
576     log.Read_Parallel("parseStream: we have an object: %d", a)
577
578     streamOffset = -offset
579     id, err = objRefStreamObj(a, v, objectNumber, streamOffset)
580     if err == nil {
581         return nil, err
582     }
583
584     // No have an xref stream object
585
586     err = parallelFindObjIdAnd, createObjTable(a)
587     if err == nil {
588         return nil, err
589     }
590
591     // parse objectStream and create objTable entries for embedded objects
592     err = extractObjTableEntriesFromStream(a.Content, id, obj)
593     if err == nil {
594         return nil, err
595     }
596
597     // Create objTableEntry for objRefStreamId
598     entry = &objRefStream{
599         objRefStream:
600             {
601                 offset:         offset,
602                 generationNumber: generationNumber,
603                 object:         obj
604             }
605     }

```

```

787 //
788 return nil, nil
789 }
790
791 // Indirectly string (book, error) {
792 func indirectString(book, error) {
793     & err = paraDict(jdict)
794     if err == nil {
795         return false, err
796     }
797     // do a go.Dict()
798     return &w, nil
799 }
800
801 // scanTracker <- bufio.Scanner, line string() Tracker, error() {
802 func scanTracker(s *bufio.Scanner, line string) Tracker, error() {
803     //
804     var buf bytes.Buffer
805     var err error
806     var i, j, k int
807
808     log.Debug.Printf("line: %s\n", line)
809
810     // Scan for dict start tag "o"
811     for {
812         i = strings.Index(line, "o")
813         if i >= 0 {
814             break
815         }
816         line = scanTrimLine(line, i)
817         log.Debug.Printf("line: %s\n", line)
818         if err == nil {
819             return "err"
820         }
821     }
822
823     line = scanTrimLine(line, i)
824     buf.WriteString(line)
825     buf.WriteString("\n")
826     log.Debug.Printf("scanTracker dictish after start tag: %s\n", line)
827
828     // Scan for dict end tag "}" but account for inner dicts.
829     line = scanTrimLine(line, i)
830     log.Debug.Printf("line: %s\n", line)
831     for {
832         if len(line) == 0 {
833             line, err = scanTrimLine(s)
834             if err == nil {
835                 return "err"
836             }
837             buf.WriteString(line)
838             buf.WriteString("\n")
839             log.Debug.Printf("scanTracker dictish next line: %s\n", line)
840         }
841         i = strings.Index(line, "}")
842         if i < 0 {
843             i = strings.Index(line, "o")
844             if i >= 0 {

```

```

3620 if (fS) = read {
3621     eCount := 1
3622 }
3623 } else if (fS) = 0x0 {
3624     eCount := 1
3625     if (fS) = read {
3626         eCount := 2
3627     }
3628 }
3629 } else {
3630     return nil, &, errorCorruptHeader
3631 }
3632
3633 readLog.Printf("headerVersion: end, found header version: %d\n", pdfVersion)
3634
3635 readPdfHeader(eCount, nil)
3636
3637 // =====
3638 // byteStream is a hack for displaying corrupt sct sections.
3639 // It populates the streamable by reading in all indirect objects line by line
3640 // and then concatenates a streamable of a single sct section - meaning no incremental
3641 // updates have been used.
3642 // =====
3643 func byteStream(sct *Context) error {
3644     var s1 index
3645     s := PdfStreamDecoder(
3646         ctx := sct,
3647         f := sct,
3648         f := sct,
3649         f := sct,
3650         f := sct,
3651         f := sct,
3652         f := sct,
3653         f := sct,
3654         f := sct,
3655         f := sct,
3656         f := sct,
3657         f := sct,
3658         f := sct,
3659         f := sct,
3660         f := sct,
3661         f := sct,
3662         f := sct,
3663         f := sct,
3664         f := sct,
3665         f := sct,
3666         f := sct,
3667         f := sct,
3668         f := sct,
3669         f := sct,
3670         f := sct,
3671         f := sct,
3672         f := sct,
3673         f := sct,
3674         f := sct,
3675         f := sct,
3676         f := sct,
3677         f := sct,
3678         f := sct,
3679         f := sct,
3680         f := sct,
3681         f := sct,
3682         f := sct,
3683         f := sct,
3684         f := sct,
3685         f := sct,
3686         f := sct,
3687         f := sct,
3688         f := sct,
3689         f := sct,
3690         f := sct,
3691         f := sct,
3692         f := sct,
3693         f := sct,
3694         f := sct,
3695         f := sct,
3696         f := sct,
3697         f := sct,
3698         f := sct,
3699         f := sct,
3700         f := sct,
3701         f := sct,
3702         f := sct,
3703         f := sct,
3704         f := sct,
3705         f := sct,
3706         f := sct,
3707         f := sct,
3708         f := sct,
3709         f := sct,
3710         f := sct,
3711         f := sct,
3712         f := sct,
3713         f := sct,
3714         f := sct,
3715         f := sct,
3716         f := sct,
3717         f := sct,
3718         f := sct,
3719         f := sct,
3720         f := sct,
3721         f := sct,
3722         f := sct,
3723         f := sct,
3724         f := sct,
3725         f := sct,
3726         f := sct,
3727         f := sct,
3728         f := sct,
3729         f := sct,
3730         f := sct,
3731         f := sct,
3732         f := sct,
3733         f := sct,
3734         f := sct,
3735         f := sct,
3736         f := sct,
3737         f := sct,
3738         f := sct,
3739         f := sct,
3740         f := sct,
3741         f := sct,
3742         f := sct,
3743         f := sct,
3744         f := sct,
3745         f := sct,
3746         f := sct,
3747         f := sct,
3748         f := sct,
3749         f := sct,
3750         f := sct,
3751         f := sct,
3752         f := sct,
3753         f := sct,
3754         f := sct,
3755         f := sct,
3756         f := sct,
3757         f := sct,
3758         f := sct,
3759         f := sct,
3760         f := sct,
3761         f := sct,
3762         f := sct,
3763         f := sct,
3764         f := sct,
3765         f := sct,
3766         f := sct,
3767         f := sct,
3768         f := sct,
3769         f := sct,
3770         f := sct,
3771         f := sct,
3772         f := sct,
3773         f := sct,
3774         f := sct,
3775         f := sct,
3776         f := sct,
3777         f := sct,
3778         f := sct,
3779         f := sct,
3780         f := sct,
3781         f := sct,
3782         f := sct,
3783         f := sct,
3784         f := sct,
3785         f := sct,
3786         f := sct,
3787         f := sct,
3788         f := sct,
3789         f := sct,
3790         f := sct,
3791         f := sct,
3792         f := sct,
3793         f := sct,
3794         f := sct,
3795         f := sct,
3796         f := sct,
3797         f := sct,
3798         f := sct,
3799         f := sct,
3800         f := sct,
3801         f := sct,
3802         f := sct,
3803         f := sct,
3804         f := sct,
3805         f := sct,
3806         f := sct,
3807         f := sct,
3808         f := sct,
3809         f := sct,
3810         f := sct,
3811         f := sct,
3812         f := sct,
3813         f := sct,
3814         f := sct,
3815         f := sct,
3816         f := sct,
3817         f := sct,
3818         f := sct,
3819         f := sct,
3820         f := sct,
3821         f := sct,
3822         f := sct,
3823         f := sct,
3824         f := sct,
3825         f := sct,
3826         f := sct,
3827         f := sct,
3828         f := sct,
3829         f := sct,
3830         f := sct,
3831         f := sct,
3832         f := sct,
3833         f := sct,
3834         f := sct,
3835         f := sct,
3836         f := sct,
3837         f := sct,
3838         f := sct,
3839         f := sct,
3840         f := sct,
3841         f := sct,
3842         f := sct,
3843         f := sct,
3844         f := sct,
3845         f := sct,
3846         f := sct,
3847         f := sct,
3848         f := sct,
3849         f := sct,
3850         f := sct,
3851         f := sct,
3852         f := sct,
3853         f := sct,
3854         f := sct,
3855         f := sct,
3856         f := sct,
3857         f := sct,
3858         f := sct,
3859         f := sct,
3860         f := sct,
3861         f := sct,
3862         f := sct,
3863         f := sct,
3864         f := sct,
3865         f := sct,
3866         f := sct,
3867         f := sct,
3868         f := sct,
3869         f := sct,
3870         f := sct,
3871         f := sct,
3872         f := sct,
3873         f := sct,
3874         f := sct,
3875         f := sct,
3876         f := sct,
3877         f := sct,
3878         f := sct,
3879         f := sct,
3880         f := sct,
3881         f := sct,
3882         f := sct,
3883         f := sct,
3884         f := sct,
3885         f := sct,
3886         f := sct,
3887         f := sct,
3888         f := sct,
3889         f := sct,
3890         f := sct,
3891         f := sct,
3892         f := sct,
3893         f := sct,
3894         f := sct,
3895         f := sct,
3896         f := sct,
3897         f := sct,
3898         f := sct,
3899         f := sct,
3900         f := sct,
3901         f := sct,
3902         f := sct,
3903         f := sct,
3904         f := sct,
3905         f := sct,
3906         f := sct,
3907         f := sct,
3908         f := sct,
3909         f := sct,
3910         f := sct,
3911         f := sct,
3912         f := sct,
3913         f := sct,
3914         f := sct,
3915         f := sct,
3916         f := sct,
3917         f := sct,
3918         f := sct,
3919         f := sct,
3920         f := sct,
3921         f := sct,
3922         f := sct,
3923         f := sct,
3924         f := sct,
3925         f := sct,
3926         f := sct,
3927         f := sct,
3928         f := sct,
3929         f := sct,
3930         f := sct,
3931         f := sct,
3932         f := sct,
3933         f := sct,
3934         f := sct,
3935         f := sct,
3936         f := sct,
3937         f := sct,
3938         f := sct,
3939         f := sct,
3940         f := sct,
3941         f := sct,
3942         f := sct,
3943         f := sct,
3944         f := sct,
3945         f := sct,
3946         f := sct,
3947         f := sct,
3948         f := sct,
3949         f := sct,
3950         f := sct,
3951         f := sct,
3952         f := sct,
3953         f := sct,
3954         f := sct,
3955         f := sct,
3956         f := sct,
3957         f := sct,
3958         f := sct,
3959         f := sct,
3960         f := sct,
3961         f := sct,
3962         f := sct,
3963         f := sct,
3964         f := sct,
3965         f := sct,
3966         f := sct,
3967         f := sct,
3968         f := sct,
3969         f := sct,
3970         f := sct,
3971         f := sct,
3972         f := sct,
3973         f := sct,
3974         f := sct,
3975         f := sct,
3976         f := sct,
3977         f := sct,
3978         f := sct,
3979         f := sct,
3980         f := sct,
3981         f := sct,
3982         f := sct,
3983         f := sct,
3984         f := sct,
3985         f := sct,
3986         f := sct,
3987         f := sct,
3988        
```

```

618 if lim(offset) == "if" {
619     off++
620 } else if lim(offset) == "not" {
621     if lim(off) == "\n" {
622         off++
623     }
624 }
625 return
626 }
627
628 func lastStreamMarker(streamEnd int, ended int, line string) {
629     // We found the last stream marker.
630     if streamEnd > len(line)-len("stream") {
631         // No guess for another stream start.
632         streamEnd = -1
633         return
634     }
635     // We start searching after this stream marker.
636     bufpos := streamEnd + len("stream")
637     // We found the next stream marker.
638     if streamEnd == len("stream") + 1 {
639         // No guess for stream start or if another object
640         streamEnd = -1
641     }
642     // We found the next stream marker.
643     streamEnd += len("stream") + 1
644 }
645
646 if ended > 0 || streamEnd > 0 {
647     // We found a stream start or if another object
648     streamEnd = -1
649 }
650
651 // Finally we got file buffer of sufficient size for parsing an object via stream
652 // buffer(s) to reader() but [byte, ended int, streaming int, streamoffset int] error()
653 err error()
654
655 // process: p gen obj... obj dict ... [stream ... data ... endstream] ... endobj
656 // above ... .. if above ... .. if if
657
658 //log.Based.ResetIn(buf, begin)
659
660 ended, streamEnd = -1, -1
661
662 for ended <= 0 || streamEnd < 0 {
663     buf, err = growBufBy(buf, defaultBufSize, rd)
664     if err != nil {
665         return nil, 0, 0, 0, err
666     }

```

```

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

```

[illegible]

```

600 // Insert new key-value pair into the map
601 log.Debug.Print("parseInfoStream: Insert new infoTable entry for Object %d\n",
602     subjectNumber)
603
604 ctx.Table.Add(newKeyValuePair) // Entry
605 log.Debug.Print("parseInfoStream:obj(subjectNumber) = true
606     preoffset = cctx.PreviousOffset
607
608
609
610 log.Debug.Print("parseInfoStream: end")
611
612
613 return preoffset, nil
614
615
616
617 // Parse the infoTable stream and return the file
618 func parseInfoTableStream(offset int64, cctx *Context) error {
619
620     log.Debug.Print("parseInfoTableStream: begin")
621
622     rd, err = newMultipartReader(cctx.NewReader(), cctx)
623     if err != nil {
624         return err
625     }
626
627     // err = parseInfoStreamAndOffset, offset, etc)
628     if err != nil {
629         return err
630     }
631
632
633 log.Debug.Print("parseInfoTableStream: end")
634
635 return nil
636
637
638 // Parse the infoTable stream and return the offset of the previous seed section.
639 func parseInfoTableAndFindSeedOffset(infoTable *infoTable) error {
640
641     log.Debug.Print("parseInfoTableAndFindSeedOffset: begin")
642
643     // found = findSeedOffset("encrypt"): found =
644     // encryptOffset = nil
645     // infoTable.encrypt = encryptOffset
646     log.Debug.Print("parseInfoTableAndFindSeedOffset: Encrypt object %d\n",
647         infoTable.encrypt)
648
649
650
651
652
653
654
655
656
657
658
659
660 if infoTable.Size == nil {
661     // size = 512 * 1024
662     if size == nil {
663         return errors.New("offset: parseInfoTableAndFindSeedOffset: missing entry \"Size\"")
664     }
665     // Not reliable:
666     // Assume fixed offset read size.
667     // offsetTable.Size = 512 * 1024
668 }
669
670 if infoTable.Root == nil {
671     rootObjName = cctx.InfoTableEntryName("Root")
672 }
673

```

```

800         }
801         if s == 0 {
802             // Check for data
803             ok, err := bufio.ReadString()
804             if err == nil || ok {
805                 return buf.String(), nil
806             }
807         } else {
808             k++
809             line = line[:j-1]
810         }
811         continue
812     }
813     // Check for line
814     line, err = scanline(s)
815     if err == nil {
816         return "err"
817     }
818     bufio.ReadString(line)
819     buf.WriteString("\n")
820     // ReadPrint("controller dicted next line: %s\n", line)
821 } else {
822     // ReadPrint("Index line: %s")
823     if j < 0 {
824         k++
825         line = line[:j-1]
826     } else {
827         if j < j {
828             // Check for
829             k++
830             line = line[:j-1]
831         } else {
832             // Check for
833             if k == 0 {
834                 // Check for dict
835                 ok, err := bufio.ReadString()
836                 if err == nil || ok {
837                     return buf.String(), nil
838                 }
839             } else {
840                 k++
841                 line = line[:j-1]
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 }
14
```

```

5830         //err = processTrailer(ctx, s, string(bb))
5831         return err
5832     }
5833     continue
5834 } // done all until "trailer"
5835 if s == string(line, "trailer")
5836 if s > 0
5837     bb = append(bb, line...)
5838     withinTrailer = true
5839 }
5840 continue
5841 }
5842 s = string(line, "zero")
5843 if s > 0
5844     offset = int64(mc(line) + eolCount)
5845     withinTrailer = true
5846     withinObj = 1
5847     if withinObj > 1
5848         s = string(line, "obj")
5849         if s > 0
5850             withinObj = true
5851             off = offset
5852             bb = append(bb, line[1:off-1]...)
5853             offset = int64(mc(line) + eolCount)
5854         }
5855     }
5856 }
5857 //various obj
5858 offset = int64(mc(line) + eolCount)
5859 bb = append(bb, ")")
5860 bb = append(bb, line...)
5861 if s == string(line, "endobj")
5862 if s > 0
5863     s = string(bb)
5864     objId = generation, err = parseObjectAttributes(s)
5865     if err == nil
5866         return err
5867     }
5868     off = off + 1
5869     ctx.fullObjId = objId
5870     free, false,
5871     offset, off,
5872     generation, generation)
5873     bb = nil
5874     withinObj = false
5875 }
5876 }
5877 return nil
5878 }
5879 // Build a readable log from stream s and section.
5880 func buildReadableFromStartingIndex(ctx *Context, offset int64) error {
5881     log, err := Log.Read_Primitive(buildReadableFromStartingIndex, begin)
5882 }

```

[illegible]


```

1517         }
1518         return false
1519     }
1520 }
1521 // We found the last zero (end1) just after end of dict only whitespace,
1522 ok = strings.TrimSpace(end1) == ">"
1523 // Log, Read, Printlnf "keywords:dictname:lighter:dict: end: %s\n", ok)
1524 // Log, Read, Printlnf "keywords:dictname:lighter:dict: end: %s\n", ok)
1525 return ok
1526 }
1527
1528 func buildFilterPipeline(ctx *Context, filterArray, decodeParseArr Array, decodeParse
1529 *dict) (*FilterPipeline, error) {
1530     var filterPipeline []Filter
1531     for i, f := range filterArray {
1532         filterName, ok := f.(Name)
1533         if !ok {
1534             return nil, errors.New("pdpcc: buildFilterPipeline: filterArray elements
1535 corrupt")
1536         }
1537         if decodeParse == nil || decodeParseArr[i] == nil {
1538             filterPipeline = append(filterPipeline, POFfilter{Name:
1539 filterName, decodeParse: nil})
1540             continue
1541         }
1542         dict, ok := decodeParseArr[i].(Dict)
1543         if !ok {
1544             return nil, errors.New("pdpcc: buildFilterPipeline: i:IndexFilter)
1545 corrupt")
1546         }
1547         return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s\n",
1548 dict)
1549     }
1550     if err := deferenceDicts(dict, indexDef.ObjectName.Value());
1551     if err != nil
1552         return nil, err
1553     }
1554     dict = d
1555 }
1556
1557 filterPipeline = append(filterPipeline, POFfilter{Name: filterName.String(),
1558 decodeParse: dict})
1559 }
1560
1561 return filterPipeline, nil
1562 }
1563
1564 // Begin the after pipeline associated with this source dict:
1565 func buildFilterPipelines(*Context, dict Dict) (*FilterPipeline, error) {
1566     log.Read.Printlnf("pdpfilterPipeline: begin")
1567     var err error
1568     if found := dict.Find("Filter")
1569     if found {
1570         // stream is not compressed
1571     }
1572 }

```

```

3350 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3351 // @return {nil}
3352 // @type {nil, 0, 0, 0, 0, err}
3353
3354 if objErr == nil {
3355     return nil, 0, 0, 0, 0, err
3356 }
3357
3358 if objErr != nil {
3359     if objErr == nil {
3360         if objErr == nil {
3361             return nil, 0, 0, 0, 0, err
3362         }
3363         return nil, 0, 0, 0, 0, err
3364     }
3365     return nil, 0, 0, 0, 0, err
3366 }
3367
3368 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3369 // @return {nil}
3370 // @type {nil, 0, 0, 0, 0, err}
3371
3372 if objErr == nil {
3373     return nil, 0, 0, 0, 0, err
3374 }
3375
3376 if objErr != nil {
3377     if objErr == nil {
3378         if objErr == nil {
3379             return nil, 0, 0, 0, 0, err
3380         }
3381         return nil, 0, 0, 0, 0, err
3382     }
3383     return nil, 0, 0, 0, 0, err
3384 }
3385
3386 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3387 // @return {nil}
3388 // @type {nil, 0, 0, 0, 0, err}
3389
3390 if objErr == nil {
3391     return nil, 0, 0, 0, 0, err
3392 }
3393
3394 if objErr != nil {
3395     if objErr == nil {
3396         if objErr == nil {
3397             return nil, 0, 0, 0, 0, err
3398         }
3399         return nil, 0, 0, 0, 0, err
3400     }
3401     return nil, 0, 0, 0, 0, err
3402 }
3403
3404 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3405 // @return {nil}
3406 // @type {nil, 0, 0, 0, 0, err}
3407
3408 if objErr == nil {
3409     return nil, 0, 0, 0, 0, err
3410 }
3411
3412 if objErr != nil {
3413     if objErr == nil {
3414         if objErr == nil {
3415             return nil, 0, 0, 0, 0, err
3416         }
3417         return nil, 0, 0, 0, 0, err
3418     }
3419     return nil, 0, 0, 0, 0, err
3420 }
3421
3422 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3423 // @return {nil}
3424 // @type {nil, 0, 0, 0, 0, err}
3425
3426 if objErr == nil {
3427     return nil, 0, 0, 0, 0, err
3428 }
3429
3430 if objErr != nil {
3431     if objErr == nil {
3432         if objErr == nil {
3433             return nil, 0, 0, 0, 0, err
3434         }
3435         return nil, 0, 0, 0, 0, err
3436     }
3437     return nil, 0, 0, 0, 0, err
3438 }
3439
3440 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3441 // @return {nil}
3442 // @type {nil, 0, 0, 0, 0, err}
3443
3444 if objErr == nil {
3445     return nil, 0, 0, 0, 0, err
3446 }
3447
3448 if objErr != nil {
3449     if objErr == nil {
3450         if objErr == nil {
3451             return nil, 0, 0, 0, 0, err
3452         }
3453         return nil, 0, 0, 0, 0, err
3454     }
3455     return nil, 0, 0, 0, 0, err
3456 }
3457
3458 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3459 // @return {nil}
3460 // @type {nil, 0, 0, 0, 0, err}
3461
3462 if objErr == nil {
3463     return nil, 0, 0, 0, 0, err
3464 }
3465
3466 if objErr != nil {
3467     if objErr == nil {
3468         if objErr == nil {
3469             return nil, 0, 0, 0, 0, err
3470         }
3471         return nil, 0, 0, 0, 0, err
3472     }
3473     return nil, 0, 0, 0, 0, err
3474 }
3475
3476 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3477 // @return {nil}
3478 // @type {nil, 0, 0, 0, 0, err}
3479
3480 if objErr == nil {
3481     return nil, 0, 0, 0, 0, err
3482 }
3483
3484 if objErr != nil {
3485     if objErr == nil {
3486         if objErr == nil {
3487             return nil, 0, 0, 0, 0, err
3488         }
3489         return nil, 0, 0, 0, 0, err
3490     }
3491     return nil, 0, 0, 0, 0, err
3492 }
3493
3494 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3495 // @return {nil}
3496 // @type {nil, 0, 0, 0, 0, err}
3497
3498 if objErr == nil {
3499     return nil, 0, 0, 0, 0, err
3500 }
3501
3502 if objErr != nil {
3503     if objErr == nil {
3504         if objErr == nil {
3505             return nil, 0, 0, 0, 0, err
3506         }
3507         return nil, 0, 0, 0, 0, err
3508     }
3509     return nil, 0, 0, 0, 0, err
3510 }
3511
3512 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3513 // @return {nil}
3514 // @type {nil, 0, 0, 0, 0, err}
3515
3516 if objErr == nil {
3517     return nil, 0, 0, 0, 0, err
3518 }
3519
3520 if objErr != nil {
3521     if objErr == nil {
3522         if objErr == nil {
3523             return nil, 0, 0, 0, 0, err
3524         }
3525         return nil, 0, 0, 0, 0, err
3526     }
3527     return nil, 0, 0, 0, 0, err
3528 }
3529
3530 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3531 // @return {nil}
3532 // @type {nil, 0, 0, 0, 0, err}
3533
3534 if objErr == nil {
3535     return nil, 0, 0, 0, 0, err
3536 }
3537
3538 if objErr != nil {
3539     if objErr == nil {
3540         if objErr == nil {
3541             return nil, 0, 0, 0, 0, err
3542         }
3543         return nil, 0, 0, 0, 0, err
3544     }
3545     return nil, 0, 0, 0, 0, err
3546 }
3547
3548 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3549 // @return {nil}
3550 // @type {nil, 0, 0, 0, 0, err}
3551
3552 if objErr == nil {
3553     return nil, 0, 0, 0, 0, err
3554 }
3555
3556 if objErr != nil {
3557     if objErr == nil {
3558         if objErr == nil {
3559             return nil, 0, 0, 0, 0, err
3560         }
3561         return nil, 0, 0, 0, 0, err
3562     }
3563     return nil, 0, 0, 0, 0, err
3564 }
3565
3566 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3567 // @return {nil}
3568 // @type {nil, 0, 0, 0, 0, err}
3569
3570 if objErr == nil {
3571     return nil, 0, 0, 0, 0, err
3572 }
3573
3574 if objErr != nil {
3575     if objErr == nil {
3576         if objErr == nil {
3577             return nil, 0, 0, 0, 0, err
3578         }
3579         return nil, 0, 0, 0, 0, err
3580     }
3581     return nil, 0, 0, 0, 0, err
3582 }
3583
3584 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3585 // @return {nil}
3586 // @type {nil, 0, 0, 0, 0, err}
3587
3588 if objErr == nil {
3589     return nil, 0, 0, 0, 0, err
3590 }
3591
3592 if objErr != nil {
3593     if objErr == nil {
3594         if objErr == nil {
3595             return nil, 0, 0, 0, 0, err
3596         }
3597         return nil, 0, 0, 0, 0, err
3598     }
3599     return nil, 0, 0, 0, 0, err
3600 }
3601
3602 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3603 // @return {nil}
3604 // @type {nil, 0, 0, 0, 0, err}
3605
3606 if objErr == nil {
3607     return nil, 0, 0, 0, 0, err
3608 }
3609
3610 if objErr != nil {
3611     if objErr == nil {
3612         if objErr == nil {
3613             return nil, 0, 0, 0, 0, err
3614         }
3615         return nil, 0, 0, 0, 0, err
3616     }
3617     return nil, 0, 0, 0, 0, err
3618 }
3619
3620 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3621 // @return {nil}
3622 // @type {nil, 0, 0, 0, 0, err}
3623
3624 if objErr == nil {
3625     return nil, 0, 0, 0, 0, err
3626 }
3627
3628 if objErr != nil {
3629     if objErr == nil {
3630         if objErr == nil {
3631             return nil, 0, 0, 0, 0, err
3632         }
3633         return nil, 0, 0, 0, 0, err
3634     }
3635     return nil, 0, 0, 0, 0, err
3636 }
3637
3638 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3639 // @return {nil}
3640 // @type {nil, 0, 0, 0, 0, err}
3641
3642 if objErr == nil {
3643     return nil, 0, 0, 0, 0, err
3644 }
3645
3646 if objErr != nil {
3647     if objErr == nil {
3648         if objErr == nil {
3649             return nil, 0, 0, 0, 0, err
3650         }
3651         return nil, 0, 0, 0, 0, err
3652     }
3653     return nil, 0, 0, 0, 0, err
3654 }
3655
3656 // @param {Object} generatorOrErr - parseObjectGeneratorOrErr
3657 // @return {nil}
3658 // @type {nil, 0, 0, 0, 0, err}
3659
3660 if objErr == nil {
3661     return nil, 0, 0, 0, 0, err
3662 }
3663
3664 if objErr != nil {
3665     if objErr == nil {
3666         if objErr == nil {
3667             return nil, 0, 0, 0, 0, err
3668
```

```

1130 // Save the saveDecodedContentContent to ctx.content, id, saveStreams, objKey, govt, int,
1131 // err (error)
1132
1133 // Log.Read.Print("saveDecodedContentContent: begin decode\n"), decode)
1134
1135 // If the "identity" crypt filter is used we do not need to decode.
1136 if cty == nil || cty.IsKey == nil {
1137     // If id is FilterPipeline[id] = 1 do so, FilterPipeline[id].Name = "Crypt"
1138     idContent = so.Raw
1139     return nil
1140 }
1141
1142 // Log.Read.Print("saveDecodedContentContent: end decode\n")
1143
1144 // Special case: If the length of the encoded data is 0, we do not need to decode
1145 // anything.
1146 if len(idRaw) == 0 {
1147     idContent = id.Raw
1148     return nil
1149 }
1150
1151 // cty gets created after theStream parsing.
1152 // theStream is not encrypted.
1153 if cty == nil || cty.IsKey == nil {
1154     idRaw, err = decryptRaw(idRaw, objKey, govt, cty.IsKey, cty.AESStreams,
1155 cty.Ex)
1156     if err == nil {
1157         return err
1158     }
1159     // If id is not nil
1160     id = len(idRaw)
1161     idContent.Length = id
1162 }
1163
1164 // If decode
1165 return nil
1166
1167 // Actual decoding of content stream.
1168 err = decodeStream(id,
1169 // If err = filter.StreamSupportFilter {
1170 // err = nil
1171 // If err == nil {
1172     return err
1173 }
1174
1175 // Log.Read.Print("saveDecodedContentContent: end")
1176
1177 return nil
1178
1179 // Decode compressed objTableEntry
1180 func DecodeCompressedObjTableEntry (objTable *ObjTable, objStream int, entry
1181 // objTableEntry *ObjTableEntry) error {
1182     // Log.Read.Print("DecodeCompressedObjTableEntry: compressed object id at %d\n"),
1183     // objTableEntry, entry.ObjStream, entry.ObjStreamLen)
1184
1185     // Missing stream entry in reference object stream.
1186     objStreamLen, objTable, objEntry, objStreamLen)
1187     if objTableEntry.ObjTableEntry, objTableEntry.ObjStreamLen)
1188     if objTableEntry.ObjTableEntry, objTableEntry.ObjStreamLen)

```

```

2037         if m == nil {
2038             return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = missing array entry m, objIdR")
2039         }
2040         if len(m) == 2 {
2041             if len(a) == 4 {
2042                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs length 2 obj+id, objIdR")
2043             }
2044             offset, ok = a[0].(Integer)
2045             if !ok {
2046                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objIdR)
2047             }
2048             offset64 := Int64(offset.Value())
2049             ctx.OffsetPrincipalsTable = offset64
2050         }
2051         if len(a) == 4 {
2052             if !
2053                 return errors.Errorf("handleLinearizationPanicbit: corrupt linearization dict at objId = corrupt array entry m, needs Integer values", objIdR)
2054             }
2055             offset64 := Int64(offset.Value())
2056             ctx.OffsetOverLimitInitial = offset64
2057         }
2058     }
2059     return nil
2060 }
2061
2062 func LoadBinaryStream(ctx *Context, s *StreamReader, objIdR, genR int) error {
2063     var err error
2064     if
2065         // Load stream's content into stream data into offsetable entry
2066         err = LoadOffsetableStream(ctx, s, objIdR, genR, mId) {
2067             return errors.Newf("dereferencingContext: problem dereferencing stream %d",
2068                 objIdR)
2069         }
2070     }
2071     ctx.Read.BinarySize += s.GetSizeLength()
2072     // Decode stream's content
2073     err = saveDecodedStreamContent(ctx, s, objIdR, genR, ctx.DecodedAllStreams)
2074     return err
2075 }
2076
2077 func updateLinearizationDict(ctx *Context, o Object) {
2078     switch o := o.(type) {
2079     case StreamDict:
2080         ctx.AddBinarySize(s + s.GetSizeLength()

```

```

290 }
291 // Create a mutable version entry (since it's in the catalog
292 // and record this as rootVersion (as opposed to headerVersion).
293 xHeaderTable.rootVersion = xHeaderTable.createRootVersion();
294 log.Read.Println("IdentifyRootVersion: begin");
295 // Copy to get version from xHeaderTable
296 RootVersionStr = xHeaderTable.RootVersion();
297 if err == nil {
298     return err
299 }
300 if rootVersionStr == nil {
301     return nil
302 }
303 // Validate version and save corresponding constant to xHeaderTable.
304 rootVersion, err := PDPVersion(rootVersionStr)
305 if err != nil {
306     return errors.Wrap(err, "IdentifyRootVersion: unknown PDP Root version: %v", rootVersionStr)
307 }
308 xHeaderTable.RootVersion = rootVersion
309 // Since v1.1, the header version may be overridden by a version entry in the catalog.
310 if xHeaderTable.HeaderVersion < v1.1 {
311     log.Info.Println("IdentifyRootVersion: PDP version is %s - will ignore root version %s",
312         xHeaderTable.HeaderVersion, rootVersionStr)
313     log.Read.Println("IdentifyRootVersion: end")
314     return nil
315 }
316 // Parse all Objects including stream content from file and save to the corresponding
317 // metadataTables.
318 // Note: This process of object stream and linearization dicts.
319 xMeta = new(xObjectMetaContext, conf.Configuration) error {
320     log.Read.Println("xMetaCreateMetaFile: begin")
321     xMetaFile = ctx.XMetaFile
322     // Note for unencrypted files.
323     // Metadata provide users to open & display file.
324     // Access may be restricted (upon access strategies).
325     // Optionally provide contexts in order to gain unrestricted access.
326     if err := xMeta.CreateMetaFile();
327 
```

```

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

```

```

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

```

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

```

```

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

```

```

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

```

2120         return err
2121     }
2122     //fmt.Println("pw authenticated")
2123
2124     // Prepare decrypted entry object.
2125     err = decodeObject(object)
2126     if err != nil {
2127         return err
2128     }
2129
2130     // For each shellEntry entry assign a object either by parsing from file or pass
2131     // a decrypted object.
2132     err = decodeObject(object)
2133     if err != nil {
2134         return err
2135     }
2136
2137     // Identify an optional Version entry in the root object/catalog.
2138     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 := CRYPT_DECRYPT(cts.Cmd == SETPASSWORDS)
2150     return errors.New("pfcpu: this file is not encrypted")
2151 }
2152
2153 if cts.Cmd == DECRYPT {
2154     err := CRYPT_DECRYPT()
2155     return nil
2156 }
2157 // Encrypt subcommand found.
2158
2159 if cts.SubCmd == "i" {
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     N1, ok := cts.ID[0].(shellEntry)
2173     if ok {
2174         id, err = N1.Bytes()
2175         if err != nil {
2176             return nil, err
2177         }
2178     }
2179 }

```

[illegible]

```

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

```

```

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

```

```

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

```

```

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

```

[illegible]

```

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

```

```

1968 // Use the object stream directly for object stream reads.
1969 // If !sd, !isObject()
1970     return errors.New("pdcps: decodeObjectStream: corrupt object stream")
1971 }
1972
1973 // We have an object stream.
1974 // If !sd, err = objectStreamDict(svd)
1975 // If err == nil
1976     return errors.Wrap(err, "decodeObjectStream: problem dereferencing
1977 object stream svd", objectStream)
1978
1979 // Log Read, Print("decodeObjectStream: decoding object stream %d\n",
1980 // objectStream)
1981
1982 // Have all objects of this object stream and save them to
1983 // ObjectStreamDict objectStream.
1984 // If err = readObjectStreamDict(svd) err == nil {
1985     return errors.Wrap(err, "decodeObjectStream: problem decoding
1986 object stream svd", objectStream)
1987 }
1988
1989 // If sd objArray == nil {
1990     return errors.Wrap(err, "decodeObjectStream: objArray should be set")
1991 }
1992
1993 // Log Read, Print("decodeObjectStream: decoded object stream %d\n",
1994 // objectStream)
1995
1996 // Save object stream dict to sHeaderEntry.
1997 entry.Object = svd
1998
1999 // Log Read, Print("decodeObjectStream: end")
2000
2001 return nil
2002
2003 func handleLinearizationPanicDict(cxt *Context, obj Object, objIn int) error {
2004     // Log Read, linearized {
2005     // commentation dict already processed.
2006     return nil
2007 }
2008
2009 // handle Linearization panic dict.
2010 // If d == c == obj (dict) obj == d, itLinearizationPanicDict {
2011     handleLinearization := true
2012     cxt.LinearizationPanicDict = true
2013     // Log Read, Print("handleLinearizationPanicDict: identified LinearizationObj
2014 // %d\n", obj)
2015
2016     a := d.ArrayEntry("pr")
2017
2018 }

```

```

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

```

```

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

```