

```

3 | }
4 |
5 | Copyright 2018 The pdfcpu Authors.
6 |
7 | Licensed under the Apache License, Version 2.0 (the "License");
8 | you may not use this file except in compliance with the License.
9 | You may obtain a copy of the License at
10 |
11 | https://www.apache.org/licenses/LICENSE-2.0
12 |
13 | Unless required by applicable law or agreed to in writing, software
14 | distributed under the License is distributed on an "AS IS" BASIS,
15 | WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
16 | See the License for the specific language governing permissions and
17 | limitations under the License.
18 |
19 |
20 | package pdfcpu
21 |
22 | import (
23 |     "fmt"
24 |     "bytes"
25 |     "io"
26 |     "net"
27 |     "github.com/pdfcpu/pdfcpu/pkg/ctx"
28 |     "github.com/pdfcpu/pdfcpu/pkg/objref"
29 |     "github.com/pdfcpu/errors"
30 | )
31 |
32 | const (
33 |     defaultHost = "http"
34 |     // @formatter:off
35 |     // @formatter:on
36 | )
37 |
38 | // Readfile reads a file from a file and builds an internal structure holding its cross
39 | // reference table.
40 | func Readfile(host string, conf *Configuration) (*Context, error) {
41 |     log.Infof("Print('reading %s...'", host)
42 |     f, err := os.Open(host)
43 |     if err != nil {
44 |         return nil, errors.Wrap(err, "can't open %s", host)
45 |     }
46 |
47 |     defer func() {
48 |         f.Close()
49 |     }()
50 |
51 |     return Readf(f, conf)
52 | }
53 |
54 | // Host takes a readwriter and generates a Context.
55 | // An arbitrary representation containing a cross reference table.
56 | func Readf(rw io.ReadWriter, conf *Configuration) (*Context, error) {

```

```

543 // Parse all contents of obj object stream and save them into objStreamAndOffsetArray
544 var objStreamAndOffsetArray: Array<IObjStreamAndOffset> = new Array<IObjStreamAndOffset>()
545
546 log.debug.Printf("parseObjStreamAndOffset: begin decoding Obj objects (%d), endOfContent:
547 %d", objCount, endOfContent)
548
549 // Decode content of obj object stream and offset
550
551 // e.g., 28 0 1 25 - 2 Objects: #1 offset 0, #2 offset 25
552
553 // objCount = 2
554
555 // objCount = 2
556
557 // objCount = 2
558
559 // objCount = 2
560
561 // objCount = 2
562
563 // objCount = 2
564
565 // objCount = 2
566
567 // objCount = 2
568
569 // objCount = 2
570
571 // objCount = 2
572
573 // objCount = 2
574
575 // objCount = 2
576
577 // objCount = 2
578
579 // objCount = 2
580
581 // objCount = 2
582
583 // objCount = 2
584
585 // objCount = 2
586
587 // objCount = 2
588
589 // objCount = 2
590
591 // objCount = 2
592
593 // objCount = 2
594
595 // objCount = 2
596
597 // objCount = 2
598
599 // objCount = 2
600
601 // objCount = 2
602
603 // objCount = 2
604
605 // objCount = 2
606
607 // objCount = 2
608
609 // objCount = 2
610
611 // objCount = 2
612
613 // objCount = 2
614
615 // objCount = 2
616
617 // objCount = 2
618
619 // objCount = 2
620
621 // objCount = 2
622
623 // objCount = 2
624
625 // objCount = 2
626
627 // objCount = 2
628
629 // objCount = 2
630
631 // objCount = 2
632
633 // objCount = 2
634
635 // objCount = 2
636
637 // objCount = 2
638
639 // objCount = 2
640
641 // objCount = 2
642
643 // objCount = 2
644
645 // objCount = 2
646
647 // objCount = 2
648
649 // objCount = 2
650
651 // objCount = 2
652
653 // objCount = 2
654
655 // objCount = 2
656
657 // objCount = 2
658
659 // objCount = 2
660
661 // objCount = 2
662
663 // objCount = 2
664
665 // objCount = 2
666
667 // objCount = 2
668
669 // objCount = 2
670
671 // objCount = 2
672
673 // objCount = 2
674
675 // objCount = 2
676
677 // objCount = 2
678
679 // objCount = 2
680
681 // objCount = 2
682
683 // objCount = 2
684
685 // objCount = 2
686
687 // objCount = 2
688
689 // objCount = 2
690
691 // objCount = 2
692
693 // objCount = 2
694
695 // objCount = 2
696
697 // objCount = 2
698
699 // objCount = 2
700
701 // objCount = 2
702
703 // objCount = 2
704
705 // objCount = 2
706
707 // objCount = 2
708
709 // objCount = 2
710
711 // objCount = 2
712
713 // objCount = 2
714
715 // objCount = 2
716
717 // objCount = 2
718
719 // objCount = 2
720
721 // objCount = 2
722
723 // objCount = 2
724
725 // objCount = 2
726
727 // objCount = 2
728
729 // objCount = 2
730
731 // objCount = 2
732
733 // objCount = 2
734
735 // objCount = 2
736
737 // objCount = 2
738
739 // objCount = 2
740
741 // objCount = 2
742
743 // objCount = 2
744
745 // objCount = 2
746
747 // objCount = 2
748
749 // objCount = 2
750
751 // objCount = 2
752
753 // objCount = 2
754
755 // objCount = 2
756
757 // objCount = 2
758
759 // objCount = 2
760
761 // objCount = 2
762
763 // objCount = 2
764
765 // objCount = 2
766
767 // objCount = 2
768
769 // objCount = 2
770
771 // objCount = 2
772
773 // objCount = 2
774
775 // objCount = 2
776
777 // objCount = 2
778
779 // objCount = 2
780
781 // objCount = 2
782
783 // objCount = 2
784
785 // objCount = 2
786
787 // objCount = 2
788
789 // objCount = 2
790
791 // objCount = 2
792
793 // objCount = 2
794
795 // objCount = 2
796
797 // objCount = 2
798
799 // objCount = 2
800
801 // objCount = 2
802
803 // objCount = 2
804
805 // objCount = 2
806
807 // objCount = 2
808
809 // objCount = 2
810
811 // objCount = 2
812
813 // objCount = 2
814
815 // objCount = 2
816
817 // objCount = 2
818
819 // objCount = 2
820
821 // objCount = 2
822
823 // objCount = 2
824
825 // objCount = 2
826
827 // objCount = 2
828
829 // objCount = 2
830
831 // objCount = 2
832
833 // objCount = 2
834
835 // objCount = 2
836
837 // objCount = 2
838
839 // objCount = 2
840
841 // objCount = 2
842
843 // objCount = 2
844
845 // objCount = 2
846
847 // objCount = 2
848
849 // objCount = 2
850
851 // objCount = 2
852
853 // objCount = 2
854
855 // objCount = 2
856
857 // objCount = 2
858
859 // objCount = 2
860
861 // objCount = 2
862
863 // objCount = 2
864
865 // objCount = 2
866
867 // objCount = 2
868
869 // objCount = 2
870
871 // objCount = 2
872
873 // objCount = 2
874
875 // objCount = 2
876
877 // objCount = 2
878
879 // objCount = 2
880
881 // objCount = 2
882
883 // objCount = 2
884
885 // objCount = 2
886
887 // objCount = 2
888
889 // objCount = 2
890
891 // objCount = 2
892
893 // objCount = 2
894
895 // objCount = 2
896
897 // objCount = 2
898
899 // objCount = 2
900
901 // objCount = 2
902
903 // objCount = 2
904
905 // objCount = 2
906
907 // objCount = 2
908
909 // objCount = 2
910
911 // objCount = 2
912
913 // objCount = 2
914
915 // objCount = 2
916
917 // objCount = 2
918
919 // objCount = 2
920
921 // objCount = 2
922
923 // objCount = 2
924
925 // objCount = 2
926
927 // objCount = 2
928
929 // objCount = 2
930
931 // objCount = 2
932
933 // objCount = 2
934
935 // objCount = 2
936
937 // objCount = 2
938
939 // objCount = 2
940
941 // objCount = 2
942
943 // objCount = 2
944
945 // objCount = 2
946
947 // objCount = 2
948
949 // objCount = 2
950
951 // objCount = 2
952
953 // objCount = 2
954
955 // objCount = 2
956
957 // objCount = 2
958
959 // objCount = 2
960
961 // objCount = 2
962
963 // objCount = 2
964
965 // objCount = 2
966
967 // objCount = 2
968
969 // objCount = 2
970
971 // objCount = 2
972
973 // objCount = 2
974
975 // objCount = 2
976
977 // objCount = 2
978
979 // objCount = 2
980
981 // objCount = 2
982
983 // objCount = 2
984
985 // objCount = 2
986
987 // objCount = 2
988
989 // objCount = 2
990
991 // objCount = 2
992
993 // objCount = 2
994
995 // objCount = 2
996
997 // objCount = 2
998
999 // objCount = 2
1000

```

```

642         if xobjTableInfo == nil {
643             log.Errorf("xobjTableInfo: parentTableInfo: missing entry '%s'", x)
644         }
645         xobjTableInfo = parentTableInfo
646         log.Debug.Printf("parentTableInfo: Root object: %s", xobjTableInfo.Root)
647     }
648     if xobjTableInfo != nil {
649         if objInfoTable == nil {
650             objInfoTable = m.NewIndirectObjectTable("Info")
651         }
652         if infoObjRef == nil {
653             xobjTableInfo = infoObjRef
654             log.Debug.Printf("parentTableInfo: Info object: %s", xobjTableInfo)
655         }
656     }
657     if xobjTableInfo == nil {
658         idObjRef = objInfoTable.GetID("ID")
659         if idObjRef == nil {
660             objInfoTable = idObjRef
661             xobjTableInfo = idObjRef
662             log.Debug.Printf("ID object: %s", xobjTableInfo.ID)
663         } else if xobjTableInfo.Object == nil {
664             return errors.New("xobjTableInfo: missing entry 'ID'")
665         }
666     }
667     log.Debug.Printf("parentTableInfo: Root")
668     return nil
669 }
670
671 func parentTableIndirect(trailerDict Dict, ctx *Context) (uint64, error) {
672     log.Debug.Printf("parentTableIndirect begin")
673     xobjTable := ctx.XobjTable
674     err := parentTableInfoForIndirect(xobjTable)
675     if err == nil {
676         return nil, err
677     }
678     if err == TrailerAlreadyExists("AdditionalStreams"); err == nil {
679         err = TrailerAlreadyExists("AdditionalStreams: Root object AdditionalStreams", err)
680     }
681     for i, value := range err {
682         if !isObjRef, ok := value.(IndirectObject); ok {
683             a := append(a, indObjRef)
684         }
685     }
686     xobjTable.AdditionalStreams = a
687     return trailerDict.GetRef()
688 }
689
690 if offset == nil {
691     log.Errorf("parentTableIndirect: previous ref table section offset '%s'",
692         refName)
693     offsetFromStream = trailerDict.GetIndirect("RefFromS")
694 }

```

```

6020 if s[i] == 0x0a {
6021     eofError = 0;
6022 } else if s[i] == 0x0d {
6023     eofCount = 1;
6024     if s[i] == 0x0a {
6025         eofCount = 2;
6026     } else {
6027         return nil, 0, errCopyrightHeader
6028     }
6029 }
6030
6031 log.debug.Printf("headerVersion: endf, found header version: %d\n", pdfVersion)
6032
6033 return buf, 0, eofError, eofCount, nil
6034 }
6035
6036 // pdfHeaderVersion is a hash for digesting corrupt pdf sections.
6037 // It guarantees the shArtable by reading in all indirect objects line by line
6038 // and then concatenating all of a single pdf section - making no incremental
6039 // hash updates.
6040 func (p *PDF) hashSection(s *Content) error {
6041     var z []byte
6042     z = append(z, s.Hash())
6043     for _, table := range s.Tables() {
6044         z = append(z, table.Hash())
6045     }
6046     return z
6047 }
6048
6049 func (p *PDF) GetPage() (Page, error) {
6050     Offset := 0
6051     Generation := 0
6052
6053     rx := cstr.NewReader()
6054     eofCount := cstr.NewReader()
6055     var off, offEnd int64
6056
6057     rx, off, err := pdfutils.NewReaderHeader(rx, Offset)
6058     if err == nil {
6059         return err
6060     }
6061
6062     rx = bufio.NewReaderScanner(rx)
6063     s.SplitScanLines()
6064
6065     for {
6066         rx, err := rx.ScanLine()
6067         if err == nil {
6068             break
6069         }
6070         if rx == nil {
6071             return err
6072         }
6073         if rx == nil {
6074             return err
6075         }
6076         if rx == nil {
6077             return err
6078         }
6079         if rx == nil {
6080             return err
6081         }
6082         if rx == nil {
6083             return err
6084         }
6085         if rx == nil {
6086             return err
6087         }
6088         if rx == nil {
6089             return err
6090         }
6091         if rx == nil {
6092             return err
6093         }
6094         if rx == nil {
6095             return err
6096         }
6097         if rx == nil {
6098             return err
6099         }
6100         if rx == nil {
6101             return err
6102         }
6103         if rx == nil {
6104             return err
6105         }
6106         if rx == nil {
6107             return err
6108         }
6109         if rx == nil {
6110             return err
6111         }
6112         if rx == nil {
6113             return err
6114         }
6115         if rx == nil {
6116             return err
6117         }
6118         if rx == nil {
6119             return err
6120         }
6121         if rx == nil {
6122             return err
6123         }
6124         if rx == nil {
6125             return err
6126         }
6127         if rx == nil {
6128             return err
6129         }
6130         if rx == nil {
6131             return err
6132         }
6133         if rx == nil {
6134             return err
6135         }
6136         if rx == nil {
6137             return err
6138         }
6139         if rx == nil {
6140             return err
6141         }
6142         if rx == nil {
6143             return err
6144         }
6145         if rx == nil {
6146             return err
6147         }
6148         if rx == nil {
6149             return err
6150         }
6151         if rx == nil {
6152             return err
6153         }
6154         if rx == nil {
6155             return err
6156         }
6157         if rx == nil {
6158             return err
6159         }
6160         if rx == nil {
6161             return err
6162         }
6163         if rx == nil {
6164             return err
6165         }
6166         if rx == nil {
6167             return err
6168         }
6169         if rx == nil {
6170             return err
6171         }
6172         if rx == nil {
6173             return err
6174         }
6175         if rx == nil {
6176             return err
6177         }
6178         if rx == nil {
6179             return err
6180         }
6181         if rx == nil {
6182             return err
6183         }
6184         if rx == nil {
6185             return err
6186         }
6187         if rx == nil {
6188             return err
6189         }
6190         if rx == nil {
6191             return err
6192         }
6193         if rx == nil {
6194             return err
6195         }
6196         if rx == nil {
6197             return err
6198         }
6199         if rx == nil {
6200             return err
6201         }
6202         if rx == nil {
6203             return err
6204         }
6205         if rx == nil {
6206             return err
6207         }
6208         if rx == nil {
6209             return err
6210         }
6211         if rx == nil {
6212             return err
6213         }
6214         if rx == nil {
6215             return err
6216         }
6217         if rx == nil {
6218             return err
6219         }
6220         if rx == nil {
6221             return err
6222         }
6223         if rx == nil {
6224             return err
6225         }
6226         if rx == nil {
6227             return err
6228         }
6229         if rx == nil {
6230             return err
6231         }
6232         if rx == nil {
6233             return err
6234         }
6235         if rx == nil {
6236             return err
6237         }
6238         if rx == nil {
6239             return err
6240         }
6241         if rx == nil {
6242             return err
6243         }
6244         if rx == nil {
6245             return err
6246         }
6247         if rx == nil {
6248             return err
6249         }
6250         if rx == nil {
6251             return err
6252         }
6253         if rx == nil {
6254             return err
6255         }
6256         if rx == nil {
6257             return err
6258         }
6259         if rx == nil {
6260             return err
6261         }
6262         if rx == nil {
6263             return err
6264         }
6265         if rx == nil {
6266             return err
6267         }
6268         if rx == nil {
6269             return err
6270         }
6271         if rx == nil {
6272             return err
6273         }
6274         if rx == nil {
6275             return err
6276         }
6277         if rx == nil {
6278             return err
6279         }
6280         if rx == nil {
6281             return err
6282         }
6283         if rx == nil {
6284             return err
6285         }
6286         if rx == nil {
6287             return err
6288         }
6289         if rx == nil {
6290             return err
6291         }
6292         if rx == nil {
6293             return err
6294         }
6295         if rx == nil {
6296             return err
6297         }
6298         if rx == nil {
6299             return err
6300         }
6301         if rx == nil {
6302             return err
6303         }
6304         if rx == nil {
6305             return err
6306         }
6307         if rx == nil {
6308             return err
6309         }
6310         if rx == nil {
6311             return err
6312         }
6313         if rx == nil {
6314             return err
6315         }
6316         if rx == nil {
6317             return err
6318         }
6319         if rx == nil {
6320             return err
6321         }
6322         if rx == nil {
6323             return err
6324         }
6325         if rx == nil {
6326             return err
6327         }
6328         if rx == nil {
6329             return err
6330         }
6331         if rx == nil {
6332             return err
6333         }
6334         if rx == nil {
6335             return err
6336         }
6337         if rx == nil {
6338             return err
6339         }
6340         if rx == nil {
6341             return err
6342         }
6343         if rx == nil {
6344             return err
6345         }
6346         if rx == nil {
6347             return err
6348         }
6349         if rx == nil {
6350             return err
6351         }
6352         if rx == nil {
6353             return err
6354         }
6355         if rx == nil {
6356             return err
6357         }
6358         if rx == nil {
6359             return err
6360         }
6361         if rx == nil {
6362             return err
6363         }
6364         if rx == nil {
6365             return err
6366         }
6367         if rx == nil {
6368             return err
6369         }
6370         if rx == nil {
6371             return err
6372         }
6373         if rx == nil {
6374             return err
6375         }
6376         if rx == nil {
6377             return err
6378         }
6379         if rx == nil {
6380             return err
6381         }
6382         if rx == nil {
6383             return err
6384         }
6385         if rx == nil {
6386             return err
6387         }
6388         if rx == nil {
6389             return err
6390         }
6391         if rx == nil {
6392             return err
6393         }
6394         if rx == nil {
6395             return err
6396         }
6397         if rx == nil {
6398             return err
6399         }
6400         if rx == nil {
6401             return err
6402         }
6403         if rx == nil {
6404             return err
6405         }
6406         if rx == nil {
6407             return err
6408         }
6409         if rx == nil {
6410             return err
6411         }
6412         if rx == nil {
6413             return err
6414         }
6415         if rx == nil {
6416             return err
6417         }
6418         if rx == nil {
6419             return err
6420         }
6421         if rx == nil {
6422             return err
6423         }
6424         if rx == nil {
6425             return err
6426         }
6427         if rx == nil {
6428             return err
6429         }
6430         if rx == nil {
6431             return err
6432         }
6433         if rx == nil {
6434             return err
6435         }
6436         if rx == nil {
6437             return err
6438         }
6439         if rx == nil {
6440             return err
6441         }
6442         if rx == nil {
6443             return err
6444         }
6445         if rx == nil {
6446             return err
6447         }
6448         if rx == nil {
6449             return err
6450         }
6451         if rx == nil {
6452             return err
6453         }
6454         if rx == nil {
6455             return err
6456         }
6457         if rx == nil {
6458             return err
6459         }
6460         if rx == nil {
6461             return err
6462         }
6463         if rx == nil {
6464             return err
6465         }
6466         if rx == nil {
6467             return err
6468         }
6469         if rx == nil {
6470             return err
6471         }
6472         if rx == nil {
6473             return err
6474         }
6475         if rx == nil {

```

```

350 // https://en.cppreference.com/w/cpp/string/basic_string_view
360 Log_Read_Printf("Read: begin")
370
380 ctx_err = ReadContext(ctx, &err)
390 if err == nil {
400     return nil, err
410 }
420
430 if ctx.ReadOnly {
440     Log_Info_Printf("PDF Version 1.5 conforming reader")
450 } else {
460     Log_Info_Printf("PDF Version 1.6 conforming reader - no object streams")
470 }
480
490 // Private methods
500
510 func (c *reader) objInfo(ctx *Context, err *err) nil {
520     return nil, errors.Wrapf(err, "ctx: %v, Read: %v", ctx, err)
530 }
540
550 // Max obj objects explicitly available (load into memory) in corresponding
560 // stream
570 func (c *reader) copyObjStream(stream *Stream) {
580     if err := derefContextInfo(&ctx, ctx, err); err != nil {
590         return nil, err
600     }
610 }
620
630 // Some buffers were an invariant size size trailer.
640 func (c *reader) InfoTableSize() (ctx *Context, InfoTable *Table) {
650     ctx, InfoTable, Size = (ctx, InfoTable, 0)
660 }
670
680 Log_Read_Printf("Read: end")
690
700 return ctx, nil
710
720
730 // Scanlines is a utility function for a Scanner that returns each line of
740 // text, stripped of any trailing end-of-line marker. The returned line may
750 // contain the end-of-line marker as a carriage return followed
760 // by one newline or one carriage return or one newline.
770 // The end-of-line marker will be returned even if it has no newline.
780 func (c *reader) scanLine(data []byte, offset, token []byte, err error) (
790     scanner int, data []byte, offset int) {
800     if atEOF := data == nil {
810         return 0, nil, nil
820     }
830     index := bytes.Index(data, "\r")
840     index = bytes.Index(data, "\n")
850
860 switch {
870 case index > 0 && offset < 0:
880     if index < index1 {
890         if index1 < index2 {
900             // skip
910             return index1 + 1, data[index1:index2], nil
920         }
930     }
940 }

```

[illegible][illegible][illegible]

```

310         return (lenOfData == 1, data[0], lenOfData, nil)
311     }
312     // return nil if we have a full, non-terminated line.
313     return (lenOfData + 1, data[0:lenOfData], nil)
314 }
315
316 case lenOfData > 0:
317     // return nil if full string, error terminated line.
318     return (lenOfData + 1, data[0:lenOfData], nil)
319
320 case lenOfData > 0:
321     // return lenOfData + full and non-terminated line.
322     return (lenOfData + 1, data[0:lenOfData], nil)
323 }
324 }
325
326 // If we're at EOF, we have a final, non-terminated line. Return it.
327 if !err {
328     return (lenOfData), data, nil
329 }
330 // Request more data.
331 return nil, nil
332 }
333
334 func newUninitializedReader(r io.Reader, offset int64) (*bufio.Reader, error) {
335     if err := r.Seek(offset, io.SeekStart); err != nil {
336         return nil, err
337     }
338     log.RawPrint("newUninitializedReader: positioned to offset '%d'", offset)
339     return bufio.NewReader(r), nil
340 }
341
342 // Get the file offset of the last Modification.
343 // Get the file offset and return successfully for the first occurrence of startup
344 // error and fail on subsequent errors.
345 func offsetLastModification(ct *Moment) (*int64, error) {
346     rs := ct.Raw.NewReader()
347     var (
348         prevBuf, curBuf []byte
349         bufSize         int64 = 512
350         offset          int64
351     )
352     for i := 0; offset == 0; i++ {
353         if err := rs.Seek(-len(bufSize), io.SeekEnd);
354         if err != nil {
355             return nil, errors.New("previous can't find last read section")
356         }
357         log.RawPrint("checking for offsetLastModification starting at %d", offset)
358         prevBuf = curBuf
359         curBuf = make([]byte, bufSize)
360     }
361 }

```

[illegible]

```

767 //
768 return nil, nil
769 }
770
771 func Index(s string, needle, start, end int) (int, error) {
772     s, err := Normalize(s)
773     if err != nil {
774         return -1, err
775     }
776     if ok := IsIndex(s, needle); ok {
777         // ok we can find it
778         return start, nil
779     }
780
781     return scan(s, needle, start, end, 1)
782 }
783
784 func scan(s string, needle, start, end int) (string, error) {
785     var buf bytes.Buffer
786     buf.WriteString(s[start:end])
787     needle = needle[:len(needle)-1]
788     // log.Badfprintf("line: %s\n", line)
789     // if !len(needle) {
790     //     return "", nil
791     // }
792     // if !len(needle) {
793     //     return "", nil
794     // }
795     // if !len(needle) {
796     //     return "", nil
797     // }
798     // if !len(needle) {
799     //     return "", nil
800     // }
801     // if !len(needle) {
802     //     return "", nil
803     // }
804     // if !len(needle) {
805     //     return "", nil
806     // }
807     // if !len(needle) {
808     //     return "", nil
809     // }
810     // if !len(needle) {
811     //     return "", nil
812     // }
813     // if !len(needle) {
814     //     return "", nil
815     // }
816     // if !len(needle) {
817     //     return "", nil
818     // }
819     // if !len(needle) {
820     //     return "", nil
821     // }
822     // if !len(needle) {
823     //     return "", nil
824     // }
825     // if !len(needle) {
826     //     return "", nil
827     // }
828     // if !len(needle) {
829     //     return "", nil
830     // }
831     // if !len(needle) {
832     //     return "", nil
833     // }
834     // if !len(needle) {
835     //     return "", nil
836     // }
837     // if !len(needle) {
838     //     return "", nil
839     // }
840     // if !len(needle) {
841     //     return "", nil
842     // }
843     // if !len(needle) {
844     //     return "", nil
845     // }
846     // if !len(needle) {
847     //     return "", nil
848     // }
849     // if !len(needle) {
850     //     return "", nil
851     // }
852     // if !len(needle) {
853     //     return "", nil
854     // }
855     // if !len(needle) {
856     //     return "", nil
857     // }
858     // if !len(needle) {
859     //     return "", nil
860     // }
861     // if !len(needle) {
862     //     return "", nil
863     // }
864     // if !len(needle) {
865     //     return "", nil
866     // }
867     // if !len(needle) {
868     //     return "", nil
869     // }
870     // if !len(needle) {
871     //     return "", nil
872     // }
873     // if !len(needle) {
874     //     return "", nil
875     // }
876     // if !len(needle) {
877     //     return "", nil
878     // }
879     // if !len(needle) {
880     //     return "", nil
881     // }
882     // if !len(needle) {
883     //     return "", nil
884     // }
885     // if !len(needle) {
886     //     return "", nil
887     // }
888     // if !len(needle) {
889     //     return "", nil
890     // }
891     // if !len(needle) {
892     //     return "", nil
893     // }
894     // if !len(needle) {
895     //     return "", nil
896     // }
897     // if !len(needle) {
898     //     return "", nil
899     // }
900     // if !len(needle) {
901     //     return "", nil
902     // }
903     // if !len(needle) {
904     //     return "", nil
905     // }
906     // if !len(needle) {
907     //     return "", nil
908     // }
909     // if !len(needle) {
910     //     return "", nil
911     // }
912     // if !len(needle) {
913     //     return "", nil
914     // }
915     // if !len(needle) {
916     //     return "", nil
917     // }
918     // if !len(needle) {
919     //     return "", nil
920     // }
921     // if !len(needle) {
922     //     return "", nil
923     // }
924     // if !len(needle) {
925     //     return "", nil
926     // }
927     // if !len(needle) {
928     //     return "", nil
929     // }
930     // if !len(needle) {
931     //     return "", nil
932     // }
933     // if !len(needle) {
934     //     return "", nil
935     // }
936     // if !len(needle) {
937     //     return "", nil
938     // }
939     // if !len(needle) {
940     //     return "", nil
941     // }
942     // if !len(needle) {
943     //     return "", nil
944     // }
945     // if !len(needle) {
946     //     return "", nil
947     // }
948     // if !len(needle) {
949     //     return "", nil
950     // }
951     // if !len(needle) {
952     //     return "", nil
953     // }
954     // if !len(needle) {
955     //     return "", nil
956     // }
957     // if !len(needle) {
958     //     return "", nil
959     // }
960     // if !len(needle) {
961     //     return "", nil
962     // }
963     // if !len(needle) {
964     //     return "", nil
965     // }
966     // if !len(needle) {
967     //     return "", nil
968     // }
969     // if !len(needle) {
970     //     return "", nil
971     // }
972     // if !len(needle) {
973     //     return "", nil
974     // }
975     // if !len(needle) {
976     //     return "", nil
977     // }
978     // if !len(needle) {
979     //     return "", nil
980     // }
981     // if !len(needle) {
982     //     return "", nil
983     // }
984     // if !len(needle) {
985     //     return "", nil
986     // }
987     // if !len(needle) {
988     //     return "", nil
989     // }
990     // if !len(needle) {
991     //     return "", nil
992     // }
993     // if !len(needle) {
994     //     return "", nil
995     // }
996     // if !len(needle) {
997     //     return "", nil
998     // }
999     // if !len(needle) {
1000     //     return "", nil
1001     // }
1002     // if !len(needle) {
1003     //     return "", nil
1004     // }
1005     // if !len(needle) {
1006     //     return "", nil
1007     // }
1008     // if !len(needle) {
1009     //     return "", nil
1010     // }
1011     // if !len(needle) {
1012     //     return "", nil
1013     // }
1014     // if !len(needle) {
1015     //     return "", nil
1016     // }
1017     // if !len(needle) {
1018     //     return "", nil
1019     // }
1020     // if !len(needle) {
1021     //     return "", nil
1022     // }
1023     // if !len(needle) {
1024     //     return "", nil
1025     // }
1026     // if !len(needle) {
1027     //     return "", nil
1028     // }
1029     // if !len(needle) {
1030     //     return "", nil
1031     // }
1032     // if !len(needle) {
1033     //     return "", nil
1034     // }
1035     // if !len(needle) {
1036     //     return "", nil
1037     // }
1038     // if !len(needle) {
1039     //     return "", nil
1040     // }
1041     // if !len(needle) {
1042     //     return "", nil
1043     // }
1044     // if !len(needle) {
1045     //     return "", nil
1046     // }
1047     // if !len(needle) {
1048     //     return "", nil
1049     // }
1050     // if !len(needle) {
1051     //     return "", nil
1052     // }
1053     // if !len(needle) {
1054     //     return "", nil
1055     // }
1056     // if !len(needle) {
1057     //     return "", nil
1058     // }
1059     // if !len(needle) {
1060     //     return "", nil
1061     // }
1062     // if !len(needle) {
1063     //     return "", nil
1064     // }
1065     // if !len(needle) {
1066     //     return "", nil
1067     // }
1068     // if !len(needle) {
1069     //     return "", nil
1070     // }
1071     // if !len(needle) {
1072     //     return "", nil
1073     // }
1074     // if !len(needle) {
1075     //     return "", nil
1076     // }
1077     // if !len(needle) {
1078     //     return "", nil
1079     // }
1080     // if !len(needle) {
1081     //     return "", nil
1082     // }
1083     // if !len(needle) {
1084     //     return "", nil
1085     // }
1086     // if !len(needle) {
1087     //     return "", nil
1088     // }
1089     // if !len(needle) {
1090     //     return "", nil
1091     // }
1092     // if !len(needle) {
1093     //     return "", nil
1094     // }
1095     // if !len(needle) {
1096     //     return "", nil
1097     // }
1098     // if !len(needle) {
1099     //     return "", nil
1100     // }
1101     // if !len(needle) {
1102     //     return "", nil
1103     // }
1104     // if !len(needle) {
1105     //     return "", nil
1106     // }
1107     // if !len(needle) {
1108     //     return "", nil
1109     // }
1110     // if !len(needle) {
1111     //     return "", nil
1112     // }
1113     // if !len(needle) {
1114     //     return "", nil
1115     // }
1116     // if !len(needle) {
1117     //     return "", nil
1118     // }
1119     // if !len(needle) {
1120     //     return "", nil
1121     // }
1122     // if !len(needle) {
1123     //     return "", nil
1124     // }
1125     // if !len(needle) {
1126     //     return "", nil
1127     // }
1128     // if !len(needle) {
1129     //     return "", nil
1130     // }
1131     // if !len(needle) {
1132     //     return "", nil
1133     // }
1134     // if !len(needle) {
1135     //     return "", nil
1136     // }
1137     // if !len(needle) {
1138     //     return "", nil
1139     // }
1140     // if !len(needle) {
1141     //     return "", nil
1142     // }
1143     // if !len(needle) {
1144     //     return "", nil
1145     // }
1146     // if !len(needle) {
1147     //     return "", nil
1148     // }
1149     // if !len(needle) {
1150     //     return "", nil
1151     // }
1152     // if !len(needle) {
1153     //     return "", nil
1154     // }
1155     // if !len(needle) {
1156     //     return "", nil
1157     // }
1158     // if !len(needle) {

```

[illegible]

```

165         //err = ferror(curbuf);
166         if err == nil {
167             return nil, err
168         }
169     }
170
171     workbuf = curbuf
172     if predef == nil {
173         workbuf = append(curbuf, predef...)
174     }
175
176     // strings.LastIndex(string(workbuf), "startstr")
177     if i == -1 {
178         return nil, errors.New("no matching NDEF for startstr")
179     }
180
181     p = workbuf[i:]
182     pos := strings.Index(string(p), "startstr")
183     if pos == -1 {
184         return nil, errors.New("padding: no matching NDEF for startstr")
185     }
186
187     p = p[pos:]
188     offset, err := strconv.ParseInt(strings.TrimPrefix(string(p), "0x"), 16, 64)
189     if err != nil {
190         return nil, errors.New("padding: corrupted last sec' section")
191     }
192 }
193
194 //
195 //
196 //
197 //
198 //
199 //
200 //
201 //
202 //
203 //
204 //
205 //
206 //
207 //
208 //
209 //
210 //
211 //
212 //
213 //
214 //
215 //
216 //
217 //
218 //
219 //
220 //
221 //
222 //
223 //
224 //
225 //
226 //
227 //
228 //
229 //
230 //
231 //
232 //
233 //
234 //
235 //
236 //
237 //
238 //
239 //
240 //
241 //
242 //
243 //
244 //
245 //
246 //
247 //
248 //
249 //
250 //
251 //
252 //
253 //
254 //
255 //
256 //
257 //
258 //
259 //
260 //
261 //
262 //
263 //
264 //
265 //
266 //
267 //
268 //
269 //
270 //
271 //
272 //
273 //
274 //
275 //
276 //
277 //
278 //
279 //
280 //
281 //
282 //
283 //
284 //
285 //
286 //
287 //
288 //
289 //
290 //
291 //
292 //
293 //
294 //
295 //
296 //
297 //
298 //
299 //
300 //
301 //
302 //
303 //
304 //
305 //
306 //
307 //
308 //
309 //
310 //
311 //
312 //
313 //
314 //
315 //
316 //
317 //
318 //
319 //
320 //
321 //
322 //
323 //
324 //
325 //
326 //
327 //
328 //
329 //
330 //
331 //
332 //
333 //
334 //
335 //
336 //
337 //
338 //
339 //
340 //
341 //
342 //
343 //
344 //
345 //
346 //
347 //
348 //
349 //
350 //
351 //
352 //
353 //
354 //
355 //
356 //
357 //
358 //
359 //
360 //
361 //
362 //
363 //
364 //
365 //
366 //
367 //
368 //
369 //
370 //
371 //
372 //
373 //
374 //
375 //
376 //
377 //
378 //
379 //
380 //
381 //
382 //
383 //
384 //
385 //
386 //
387 //
388 //
389 //
390 //
391 //
392 //
393 //
394 //
395 //
396 //
397 //
398 //
399 //
400 //
401 //
402 //
403 //
404 //
405 //
406 //
407 //
408 //
409 //
410 //
411 //
412 //
413 //
414 //
415 //
416 //
417 //
418 //
419 //
420 //
421 //
422 //
423 //
424 //
425 //
426 //
427 //
428 //
429 //
430 //
431 //
432 //
433 //
434 //
435 //
436 //
437 //
438 //
439 //
440 //
441 //
442 //
443 //
444 //
445 //
446 //
447 //
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825 //
826 //
827 //
828 //
829 //
830 //
831 //
832 //
833 //
834 //
835 //
836 //
837 //
838 //
839 //
840 //
841 //
842 //
843 //
844 //
845 //
846 //
847 //
848 //
849 //
850 //
851 //
852 //
853 //
854 //
855 //
856 //
857 //
858 //
859 //
860 //
861 //
862 //
863 //
864 //
865 //
866 //
867 //
868 //
869 //
870 //
871 //
872 //
873 //
874 //
875 //
876 //
877 //
878 //
879 //
880 //
881 //
882 //
883 //
884 //
885 //
886 //
887 //
888 //
889 //
890 //
891 //
892 //
893 //
894 //
895 //
896 //
897 //
898 //
899 //
900 //
901 //
902 //
903 //
904 //
905 //
906 //
907 //
908 //
909 //
910 //
911 //
912 //
913 //
914 //
915 //
916 //
917 //
918 //
919 //
920 //
921 //
922 //
923 //
924 //
925 //
926 //
927 //
928 //
929 //
930 //
931 //
932 //
933 //
934 //
935 //
936 //
937 //
938 //
939 //
940 //
941 //
942 //
943 //
944 //
945 //
946 //
947 //
948 //
949 //
950 //
951 //
```

```

345 // Remove dictionary, object, array, and string
346 if (obj instanceof Dictionary) {
347     // obj instanceof Dictionary
348     ctx.set(objectNumber) = obj instanceof Dictionary
349 }
350 }
351 }
352
353 log.Warn.Printf("UntrackableTabIntrinsicFromStack: end")
354
355 return nil
356 }
357
358 func (stream *Stream) ctxContent(s, object, objNr int, streamOffset int64)
359 func (stream *Stream) error() {
360     // stream
361     // objNr = 0 (Dict)
362     if !ok {
363         return nil, errors.New("objNr: objNrStreamIntrinsic not dict")
364     }
365 }
366
367 // Stream string for stream string
368 streamLength, streamOffsetObj = s.Length()
369 if streamLength == nil || streamOffsetObj == nil {
370     return nil, errors.New("objNr: objNrStreamIntrinsic not 'Length' string")
371 }
372
373 filterPipeline, err = perfFilterPipeline(s, ctx)
374 if err != nil {
375     return nil, err
376 }
377
378 // objNr = stream object
379 log.Warn.Printf("UntrackableTabIntrinsic: streamObj objNr, objNr")
380 streamLength, streamOffset, streamLengthObj, streamOffsetObj,
381 filterPipeline)
382
383 if _, err = loadContentStreamContent(s, ctx); err == nil {
384     return nil, err
385 }
386
387 // Decode streamContent content
388 if err = s.decodeStreamContent(s, objNr, s, true); err == nil {
389     if err = s.decodeStreamContent(s, objNr, s, true); err == nil {
390         return nil, errors.New("UntrackableTabIntrinsic: cannot decode stream for
391         objNr, objNr")
392     }
393 }
394
395 return streamObjStreamIntrinsic(s)
396 }
397
398 // Stream obj stream and return streamable contents for all embedded objects and the stream
399 func (stream *Stream) inHeader, offset, object, ctxContent(s, objNr int, streamOffset int64,
400 streamLength, streamOffsetObj, streamLengthObj, streamOffsetObj,
401 filterPipeline)
402
403 buf, err := stream, streamOffset, streamOffset, err = buffer()
404 if err == nil {

```

```

450 // If the string is not empty
451 if k < 0 {
452     // If the string is not empty
453     ok, err = strconv.ParseInt(buf.String())
454     if err == nil {
455         // If the string is not empty
456         return buf.String(), nil
457     }
458 } else {
459     // If the string is not empty
460     line = line[:j+1]
461     continue
462 }
463 // If the string is not empty
464 line, err = strconv.ParseInt(buf.String())
465 if err == nil {
466     // If the string is not empty
467     return "", err
468 }
469 log.Printf("strconv.ParseInt: %s", buf.String())
470 } else {
471     // If the string is not empty
472     log.Printf("strconv.ParseInt: %s", buf.String())
473 }
474 // If the string is not empty
475 if j < 0 {
476     // If the string is not empty
477     k++
478     line = line[:j+1]
479 } else {
480     // If the string is not empty
481     if j < 0 {
482         // If the string is not empty
483         k++
484         line = line[:j+1]
485     } else {
486         // If the string is not empty
487         k++
488         line = line[:j+1]
489     }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

```

1890 log.Read(Println("readableFile: begin"))
1891
1892 // Log list of free objects on the "free list".
1893 // Log Read.FreeObjects(). Write, etc. FreeObjects()
1894 // Ensure valid freelist of objects.
1895 err = c.UnsafeWriteFreeObjects()
1896 if err != nil {
1897     return errors.Wrap(err, "readableFile: unexpected err")
1898 }
1899
1900 // Log list of free objects on the "free list".
1901 // Log Read.FreeObjects(). Write, etc. FreeObjects()
1902 // Ensure valid freelist of objects.
1903 err = c.UnsafeWriteFreeObjects()
1904 if err != nil {
1905     return errors.Wrap(err, "readableFile: unexpected err")
1906 }
1907
1908 log.Read(Println("readableFile: end"))
1909
1910 return
1911 }
1912
1913 func readable(buf []byte, size int, rd io.Reader) ([]byte, error) {
1914     b := make([]byte, size)
1915     _, err := rd.Read(b)
1916     if err != nil {
1917         return nil, err
1918     }
1919     // Log Read.FreeObjects(). Write, etc. FreeObjects()
1920     // Ensure valid freelist of objects.
1921     err = c.UnsafeWriteFreeObjects()
1922     if err != nil {
1923         return append(buf, b...), nil
1924     }
1925     return append(buf, b...), nil
1926 }
1927
1928 func nextStreamReader(line string, streamid int) (io.Reader) {
1929     off := streamid * len("stream")
1930     // Read streamid * len("stream")
1931     // Read streamid * len("stream")
1932     for i := len(off) - 1; i >= 0; i-- {
1933         if line[i] == '\n' {
1934             return
1935         }
1936     }
1937     // Read streamid * len("stream")
1938     for i := len(off) - 1; i >= 0; i-- {
1939         if line[i] == '\n' {
1940             return
1941         }
1942     }
1943     // Read streamid * len("stream")
1944     for i := len(off) - 1; i >= 0; i-- {
1945         if line[i] == '\n' {
1946             return
1947         }
1948     }
1949     // Read streamid * len("stream")
1950     for i := len(off) - 1; i >= 0; i-- {
1951         if line[i] == '\n' {
1952             return
1953         }
1954     }
1955     // Read streamid * len("stream")
1956     for i := len(off) - 1; i >= 0; i-- {
1957         if line[i] == '\n' {
1958             return
1959         }
1960     }
1961     // Read streamid * len("stream")
1962     for i := len(off) - 1; i >= 0; i-- {
1963         if line[i] == '\n' {
1964             return
1965         }
1966     }
1967     // Read streamid * len("stream")
1968     for i := len(off) - 1; i >= 0; i-- {
1969         if line[i] == '\n' {
1970             return
1971         }
1972     }
1973     // Read streamid * len("stream")
1974     for i := len(off) - 1; i >= 0; i-- {
1975         if line[i] == '\n' {
1976             return
1977         }
1978     }
1979     // Read streamid * len("stream")
1980     for i := len(off) - 1; i >= 0; i-- {
1981         if line[i] == '\n' {
1982             return
1983         }
1984     }
1985     // Read streamid * len("stream")
1986     for i := len(off) - 1; i >= 0; i-- {
1987         if line[i] == '\n' {
1988             return
1989         }
1990     }
1991     // Read streamid * len("stream")
1992     for i := len(off) - 1; i >= 0; i-- {
1993         if line[i] == '\n' {
1994             return
1995         }
1996     }
1997     // Read streamid * len("stream")
1998     for i := len(off) - 1; i >= 0; i-- {
1999         if line[i] == '\n' {
2000             return
2001         }
2002     }
2003     // Read streamid * len("stream")
2004     for i := len(off) - 1; i >= 0; i-- {
2005         if line[i] == '\n' {
2006             return
2007         }
2008     }
2009     // Read streamid * len("stream")
2010     for i := len(off) - 1; i >= 0; i-- {
2011         if line[i] == '\n' {
2012             return
2013         }
2014     }
2015     // Read streamid * len("stream")
2016     for i := len(off) - 1; i >= 0; i-- {
2017         if line[i] == '\n' {
2018             return
2019         }
2020     }
2021     // Read streamid * len("stream")
2022     for i := len(off) - 1; i >= 0; i-- {
2023         if line[i] == '\n' {
2024             return
2025         }
2026     }
2027     // Read streamid * len("stream")
2028     for i := len(off) - 1; i >= 0; i-- {
2029         if line[i] == '\n' {
2030             return
2031         }
2032     }
2033     // Read streamid * len("stream")
2034     for i := len(off) - 1; i >= 0; i-- {
2035         if line[i] == '\n' {
2036             return
2037         }
2038     }
2039     // Read streamid * len("stream")
2040     for i := len(off) - 1; i >= 0; i-- {
2041         if line[i] == '\n' {
2042             return
2043         }
2044     }
2045     // Read streamid * len("stream")
2046     for i := len(off) - 1; i >= 0; i-- {
2047         if line[i] == '\n' {
2048             return
2049         }
2050     }
2051     // Read streamid * len("stream")
2052     for i := len(off) - 1; i >= 0; i-- {
2053         if line[i] == '\n' {
2054             return
2055         }
2056     }
2057     // Read streamid * len("stream")
2058     for i := len(off) - 1; i >= 0; i-- {
2059         if line[i] == '\n' {
2060             return
2061         }
2062     }
2063     // Read streamid * len("stream")
2064     for i := len(off) - 1; i >= 0; i-- {
2065         if line[i] == '\n' {
2066             return
2067         }
2068     }
2069     // Read streamid * len("stream")
2070     for i := len(off) - 1; i >= 0; i-- {
2071         if line[i] == '\n' {
2072             return
2073         }
2074     }
2075     // Read streamid * len("stream")
2076     for i := len(off) - 1; i >= 0; i-- {
2077         if line[i] == '\n' {
2078             return
2079         }
2080     }
2081     // Read streamid * len("stream")
2082     for i := len(off) - 1; i >= 0; i-- {
2083         if line[i] == '\n' {
2084             return
2085         }
2086     }
2087     // Read streamid * len("stream")
2088     for i := len(off) - 1; i >= 0; i-- {
2089         if line[i] == '\n' {
2090             return
2091         }
2092     }
2093     // Read streamid * len("stream")
2094     for i := len(off) - 1; i >= 0; i-- {
2095         if line[i] == '\n' {
2096             return
2097         }
2098     }
2099     // Read streamid * len("stream")
2100     for i := len(off) - 1; i >= 0; i-- {
2101         if line[i] == '\n' {
2102             return
2103         }
2104     }
2105     // Read streamid * len("stream")
2106     for i := len(off) - 1; i >= 0; i-- {
2107         if line[i] == '\n' {
2108             return
2109         }
2110     }
2111     // Read streamid * len("stream")
2112     for i := len(off) - 1; i >= 0; i-- {
2113         if line[i] == '\n' {
2114             return
2115         }
2116     }
2117     // Read streamid * len("stream")
2118     for i := len(off) - 1; i >= 0; i-- {
2119         if line[i] == '\n' {
2120             return
2121         }
2122     }
2123     // Read streamid * len("stream")
2124     for i := len(off) - 1; i >= 0; i-- {
2125         if line[i] == '\n' {
2126             return
2127         }
2128     }
2129     // Read streamid * len("stream")
2130     for i := len(off) - 1; i >= 0; i-- {
2131         if line[i] == '\n' {
2132             return
2133         }
2134     }
2135     // Read streamid * len("stream")
2136     for i := len(off) - 1; i >= 0; i-- {
2137         if line[i] == '\n' {
2138             return
2139         }
2140     }
2141     // Read streamid * len("stream")
2142     for i := len(off) - 1; i >= 0; i-- {
2143         if line[i] == '\n' {
2144             return
2145         }
2146     }
2147     // Read streamid * len("stream")
2148     for i := len(off) - 1; i >= 0; i-- {
2149         if line[i] == '\n' {
2150             return
2151         }
2152     }
2153     // Read streamid * len("stream")
2154     for i := len(off) - 1; i >= 0; i-- {
2155         if line[i] == '\n' {
2156             return
2157         }
2158     }
2159     // Read streamid * len("stream")
2160     for i := len(off) - 1; i >= 0; i-- {
2161         if line[i] == '\n' {
2162             return
2163         }
2164     }
2165     // Read streamid * len("stream")
2166     for i := len(off) - 1; i >= 0; i-- {
2167         if line[i] == '\n' {
2168             return
2169         }
2170     }
2171     // Read streamid * len("stream")
2172     for i := len(off) - 1; i >= 0; i-- {
2173         if line[i] == '\n' {
2174             return
2175         }
2176     }
2177     // Read streamid * len("stream")
2178     for i := len(off) - 1; i >= 0; i-- {
2179         if line[i] == '\n' {
2180             return
2181         }
2182     }
2183     // Read streamid * len("stream")
2184     for i := len(off) - 1; i >= 0; i-- {
2185         if line[i] == '\n' {
2186             return
2187         }
2188     }
2189     // Read streamid * len("stream")
2190     for i := len(off) - 1; i >= 0; i-- {
2191         if line[i] == '\n' {
2192             return
2193         }
2194     }
2195     // Read streamid * len("stream")
2196     for i := len(off) - 1; i >= 0; i-- {
2197         if line[i] == '\n' {
2198             return
2199         }
2200     }
2201     // Read streamid * len("stream")
2202     for i := len(off) - 1; i >= 0; i-- {
2203         if line[i] == '\n' {
2204             return
2205         }
2206     }
2207     // Read stream
```

```

230 offset, err := stream.Seek(field(1), 0)
231 if err != nil {
232     return err
233 }
234
235 generation, err := stream.Seek(field(1))
236 if err != nil {
237     return err
238 }
239
240 entryType := field(2)
241 if entryType == "in" || entryType == "ai" {
242     return errors.New("pduip: parseMetaTableEntry: corrupt xref subsection
243 entry")
244 }
245
246 var xrefMetaTableEntry xrefMetaEntry
247
248 // use object
249
250 log.Read.Printf("parseMetaTableEntry: Object %d is in use at offset%v,
251 generation%v", objectNumber, offset, generation)
252
253 if offset == 0 {
254     log.Info.Printf("parseMetaTableEntry: Skip entry in use object %d
255 with offset 0", objectNumber)
256     return nil
257 }
258
259 xrefMetaTableEntry =
260     xrefMetaTableEntry{
261         Free:      true,
262         Offset:    offset,
263         Generation: generation)
264 } else {
265
266     // free object
267
268     log.Read.Printf("parseMetaTableEntry: Object %d is unused, next free is
269 object%v, generation%v", objectNumber, offset, generation)
270 }
271 xrefMetaTableEntry.Free = true
272
273 // free, true,
274 // Offset:    offset,
275 // Generation: generation)
276 }
277
278 log.Read.Printf("parseMetaTableEntry: Insert new xrefable entry for Object %d",
279 objectNumber)
280 xrefMetaTableEntry.objectNumber = xrefMetaEntry
281
282 log.Read.Printf("parseMetaTableEntry: end")
283
284 return nil

```

```

350         return err, err
351     }
352     log.Read_Printf("parameterStream: readin[%d][%d]s", streamIndex, streamIndex+1,
353         parameterStream, readin[streamIndex][streamIndex+1])
354     err = string(buf)
355     // Do check a parameter and therefore "stream before 'endof' if 'endof' exists.
356     // If there is no parameter then 'endof' is considered to be the end for large
357     // stream.
358     if streamIndex < 0 || (endIndex > 0 && !outEnd && streamIndex) {
359         return nil, errors.New("pocp: parameterStream: corrupt pdf file")
360     }
361     // Do it object name buf.
362     // 1. line[streamIndex]
363     objectNumber, generationNumber, err = parseObjectAttributes(s)
364     if err != nil {
365         return nil, err
366     }
367     // Do it object object
368     log.Read_Printf("parameterStream: xrefits object %d\n", objectNumber,
369         generationNumber)
370     err = parseObject(s, dereferencing object %d\n", objectNumber)
371     if err != nil {
372         return nil, errors.New("pocp: parameterStream: no object")
373     }
374     // Do it object object
375     log.Read_Printf("parameterStream: we have an object %d\n", o)
376     streamIndex++
377     // Do it stream
378     err = parseStream(s, object, o, objectNumber, streamIndex)
379     if err != nil {
380         return nil, err
381     }
382     // Do it xref stream object
383     err = parseTable(s, object, o, objectNumber)
384     if err != nil {
385         return nil, err
386     }
387     // Do it stream and create a table pointer for embedded object.
388     err = extractObjectTableInfosForStreamRead(s, content, sd, o)
389     if err != nil {
390         return nil, err
391     }
392     // Create object table for the streamRead.
393     entry =
394         struct {
395             Object      bool
396             Offset      int64
397             Generation Generation
398             Object      bool
399         }{
400             Object:      false,
401             Offset:      0,
402             Generation: 0,
403             Object:      false,
404         }
405     }

```

```

740         } else {
741             log.Redis.Print("line (%s) %s", line(line), line)
742         }
743     }
744     trailerString := s.ScanValues(&trailerString)
745     if err == nil {
746         return nil, err
747     }
748     log.Redis.Print("processTrailer: trailerString: (%s)", trailerString)
749     trailerObject := trailerString
750     if err == nil {
751         return nil, err
752     }
753     trailerObject, ok = o.Object()
754     if !ok {
755         return nil, errors.New("pofpm: processTrailer: corrupt trailer object")
756     }
757     log.Redis.Print("processTrailer: trailerObject (%s)", trailerObject)
758     return parseTrailerObject(trailerObject, ctx)
759 }
760
761 // parse the section into a corresponding name of table entries
762 // parse the section's header, scanner, ctx, Context, <table>, <table>, error {}
763 func (p *Parser) parseSectionIntoName(scannerName string) error {
764     log.Redis.Print("parseSectionIntoName begin")
765     line, err := scannerName()
766     if err == nil {
767         return nil, err
768     }
769     log.Redis.Print("parseSectionIntoName: %s", line)
770     fields := strings.Fields(line)
771     // parse the section's header into table entries
772     for i := 0; i < len(fields); i++ {
773         if err := parseTableFields(scannerName, ctx, fields[i], fields[i]); err == nil {
774             return nil, err
775         }
776     }
777     // trailer or another table subsection?
778     if line, err := scannerName(); err == nil {
779         return nil, err
780     }
781     // only line type next line for trailer
782     if len(line) == 1 {
783         if line, err := scannerName(); err == nil {
784             return nil, err
785         }
786     }
787 }

```

[illegible][illegible][illegible][illegible]

```

101 line = string(buf)
102   defined = string(line.find, "endMsg")
103   streamid = string(line.find, "stream")
104
105   if endMsg > 0 && (streamid < 0 || streamid > endMsg) {
106     break
107   }
108
109   // For every new stream there are "headers" also occur within this msg
110   // we need to find the last "stream" header before a possible end marker.
111   // streamid > 0 && !defined means we have a "stream" header (line, streamid)
112   // lastStreamHeader(streamid, endMsg, line)
113
114   log.Debug.Printf("header: endMsg=%d streamid=%d", endMsg, streamid)
115
116   if streamid < 0 {
117
118     // streamid == 0 - the offset where the actual stream data begins.
119     // This is the offset where the msg after "stream"
120
121     slack = 18 // if an optional whitespace & one (max 2 chars)
122     // before the "streamid" - slack
123
124     if line[line] < end {
125
126       // to stream buf after overflow
127       buf, err = readFromBuf, needs(line[line], rd)
128       if err != nil {
129         return nil, 0, 0, err
130       }
131
132       line = string(buf)
133
134       streamOffset = lineIdx(max(streamOffset, line), streamid)
135     }
136
137     //log.Debug.Printf("header end, returned buf=%d streamOffset=%d", lineIdx(buf), lineIdx(buf))
138
139     return buf, endMsg, streamOffset, nil
140   }
141 }
142
143 // Read a stream of data.
144 func NewStreamFromStreamID(streamID int, streamid int) bool {
145
146   //log.Debug.Printf("NewStreamFromStreamID(%d,%d) begin",
147
148   // Get a slice of the chunk right in front of "streamid"
149   b := buf[line:streamid]
150
151   // Look for the end of next marker.
152   end = string(line.find, ">")
153   if end < 0 {
154     // no end of data in buf
155   }

```


[illegible]

```

1750         return &v, nil
1751     }
1752 }
1753
1754 func dereferenceDict(ctx *Context, objectNumber int) (Dict, error) {
1755     obj := dereferenceObject(ctx, objectNumber)
1756     if err := nil {
1757         return nil, err
1758     }
1759     d, ok := v.(Dict)
1760     if !ok {
1761         return nil, errors.New("pdf/gof: dereferenceDict: corrupt dict")
1762     }
1763     return d, nil
1764 }
1765
1766 // dereference a float object, returning nil on error value.
1767 func dereferenceFloat(ctx *Context, objectNumber int) (float64, error) {
1768     log.Read.Printf("float object begin: %d\n", objectNumber)
1769     f, err := dereferenceInteger(ctx, objectNumber)
1770     if err != nil {
1771         return nil, err
1772     }
1773     f64 := float64(f.Value())
1774     log.Read.Printf("float object end: %d\n", objectNumber)
1775     return f64, nil
1776 }
1777
1778 // Reads and returns a file buffer with length = stream length using provided reader.
1779 func readStreamFromReader(r io.Reader, streamLength int) ([]byte, error) {
1780     log.Read.Printf("readStreamFromReader: begin stream length %d\n", streamLength)
1781     buf := make([]byte, streamLength)
1782     for totalCount := 0; totalCount < streamLength; {
1783         count, err := r.Read(buf[totalCount:])
1784         if err != nil {
1785             return nil, err
1786         }
1787         log.Read.Printf("readStreamFromReader: count: %d, bufLen: %d\n", count,
1788             totalCount + count)
1789         totalCount += count
1790     }
1791     return buf, nil
1792 }
1793
1794 func readStreamFromReaderAndWrite(w io.Writer, streamLength int) {
1795     log.Read.Printf("readStreamFromReaderAndWrite: end\n")
1796 }
1797
1798 // Reads and returns a file buffer with length = stream length using provided reader.
1799 func readStreamFromReaderAndWriteAndReturn(w io.Writer, streamLength int) ([]byte, error) {
1800     log.Read.Printf("readStreamFromReaderAndWriteAndReturn: begin stream length %d\n", streamLength)
1801     buf := make([]byte, streamLength)
1802     for totalCount := 0; totalCount < streamLength; {
1803         count, err := r.Read(buf[totalCount:])
1804         if err != nil {
1805             return nil, err
1806         }
1807         log.Read.Printf("readStreamFromReaderAndWriteAndReturn: count: %d, bufLen: %d\n", count,
1808             totalCount + count)
1809         totalCount += count
1810     }
1811     return buf, nil
1812 }

```

```

352         if a == null {
353             return errors.Errorf("hashInLinearizationParamDict: corrupt Linearization dict at objId - missing array entry 'A', objNr")
354         }
355         if len(a) == 2 && len(a) < 4 {
356             return errors.Errorf("hashInLinearizationParamDict: corrupt Linearization dict at objId - corrupt array entry 'A', needs length 2 or 4", objNr)
357         }
358         offset, ok = a[0].(Integer)
359         if !ok {
360             return errors.Errorf("hashInLinearizationParamDict: corrupt Linearization dict at objId - corrupt array entry 'A', needs Integer values", objNr)
361         }
362         offset = Int64Offset.ValueOf(
363             ctx.offsetFromLinearization + *offsetA)
364         if len(a) == 4 {
365             offset, ok = a[2].(Integer)
366             if !ok {
367                 return errors.Errorf("hashInLinearizationParamDict: corrupt Linearization dict at objId - corrupt array entry 'A', needs Integer values", objNr)
368             }
369             offset = Int64Offset.ValueOf(
370                 ctx.offsetFromLinearization + *offsetA + *offsetB)
371         }
372         return nil
373     }
374     return loadFromStreamDict(ctx.Context, sd.StreamDict{objNr, genIn nil} error {
375         err := error {
376             // load from stream content for stream dict into writable entry.
377             if err := loadFromStreamContent(ctx, sd); err != nil {
378                 return errors.Errorf("referenceObject: problem dereferencing stream '%d'",
379                     objNr)
380             }
381             ctx.Read.BinaryTotalSize += sd.StreamLength
382             // Remove stream content
383             sd.RemoveStreamContent()
384             // Remove stream content, sd, objNr, genNr, ctx.DoneallStream()
385             return err
386         }
387     }
388     func updateBinaryTotalSize(ctx.Context, o Object) {
389         switch o := o.(type) {
390             case StreamDict:
391                 ctx.Read.BinaryTotalSize += o.StreamLength
392             }
393     }

```

```

2757         }
2758         val ok = ctx.ID().StringLiteral()
2759         return ctx.nil(), errors.New("%pfcsp: ID must contain hex literals or string
2760         literals")
2761     }
2762     id_err = uunescape(s.Value())
2763     if err == nil {
2764         return nil, err
2765     }
2766     return ID, nil
2767 }
2768
2769 func needsHashedPermissionsAndCmd (cmdMode bool)
2770
2771 return cmd == CHANGEPW || cmd == CHANGELD || cmd == SETPERMISSIONS
2772
2773 func runtimePermissions(ctx *Context) error {
2774
2775     // ACESM Validate permissions
2776     ok_err = validatePermissions(ctx)
2777     if err == nil {
2778         return err
2779     }
2780     if ok {
2781         return errors.New("%pfcsp: corrupted permissions after upw ok")
2782     }
2783
2784     // Double check runtime permissions for pfcsp processing.
2785     if hasRuntimePermissions(ctx.cmd, ctx.id) {
2786         return errors.New("%pfcsp: insufficient access permissions")
2787     }
2788     return nil
2789 }
2790
2791 func setupContextKeyPw(*Context, d.Ctx) (err error) {
2792     ctx.i, err = supportedContextKey(d.Ctx)
2793     if err == nil {
2794         return err
2795     }
2796     ctx.i.ID, err = idByte(ctx)
2797     if err == nil {
2798         return err
2799     }
2800     var ok bool
2801     // /net/Root/"/pfcsp okpw: rds: lru", ctx.Owner(), ctx.User()
2802     // Validate the owner password via permissions/master password.
2803     ok_err = validateOwnerPassword(ctx)
2804 }

```

```

547         return nil, err
548     }
549 // compressed stream.
550 var filterPipeline [PPOffFilter]
551 if isDefect, ok := o.(IndirectRef); ok {
552     o, err = deferencementObject{city, indirectObjectNumber.Value()}.
553     if err == nil {
554         return nil, err
555     }
556 }
557 //Put.Print("deferencement filter obj: %v\n", obj)
558 if name, ok := o.(Name); ok {
559 // single filter.
560 filterName := name.String()
561 filterName = name.String()
562 o, found = dict.Find("decodeParam")
563 if found !=
564     o, err := decodeParameter(
565         lookupObject(filterPipeline, &filterName).decodeParam())
566         return append(filterPipeline, PPOffFilter{Name: filterName, DecodeParams:
567             []int, nil}...)
568     }
569     if ok := o.(Dict)
570     if ok {
571         if u, ok := o.(IndirectRef)
572         if !ok {
573             return nil, errors.Errorf("pdfOffFilterPipeline: corrupt Dict: %v\n", o)
574         }
575         o, err = deferencementObject{city, indirectReference.Value()}.
576         if err == nil {
577             return nil, err
578         }
579     }
580 // with decode parameters.
581 log.Printf("pdfOffFilterPipeline: and with decode param")
582 return append(filterPipeline, PPOffFilter{Name: filterName, DecodeParam: d}),
583 nil
584 }
585 // filter pipeline.
586 // array of filter names
587 filtersArray, ok := s.(Array)
588 if !ok {
589     return nil, errors.Errorf("pdfOffFilterPipeline: Expected filtersArray corrupt, %v",
590         s, 0, 0)
591 }
592 // Optional array of decode parameter dicts.
593 var decodeParameter Array

```

[illegible]

```

230 case (obj instanceof File) {
231     case XFileReadBinaryFile : w => StreamLength
232 }
233
234 case XFileReadMetadata :
235     case XFileReadMetadataSize : w => StreamLength
236     case XFileReadMetadataSize : w => StreamLength
237 }
238
239 // Dereference object
240 case dereferenceObject(obj, ctx => Context, objNr, last) error {
241     // Dereference object
242     xFileMeta = ctx.xFileMeta
243     xFileTableSize = load(xFileMeta.Table)
244
245     // Dereference object
246     case XFileReadMetadata : begin, dereferenceObject objNr, objNr
247     entry = xFileMeta.Table[objNr]
248
249     if entry.free {
250         // Free object
251         log-Mem-Print("Free object %d\n", objNr)
252         return nil
253     }
254
255     if entry.Compressed {
256         err = decompressFileTableEntry(xFileMeta, objNr, entry)
257         if err != nil {
258             return err
259         }
260     }
261     // Log-Mem-Print("Dereference object: decompressed entry,
262     // improving usage = entry.compressed, entry.objNr\n",
263     // objNr)
264
265     // entry is in use.
266     log-Mem-Print("In use object %d\n", objNr)
267
268     if entry.offset == nil || entry.offset == 0 {
269         log-Mem-Print("Object %d is already decompressed or used object %d\n",
270             objNr, objNr)
271         return nil
272     }
273
274     o = entry.Object
275
276     // Already dereferenced stream data.
277     if o == nil {
278         log(MemEntryObj, objNr)
279         log(MemEntryObj, objNr)
280         log(MemEntryObj, objNr)
281         // Dereference object
282         case XFileReadMetadata : using cached object %d of
283             %d\n", objNr, objNr, objNr, objNr)
284         return nil
285     }
286
287     // Dereference (load from disk into memory).
288     log-Mem-Print("Dereference object: dereferencing object %d\n", objNr)
289
290     // Free object from File writing good disk entry, stronger, free.

```

```

3444 if err == nil {
3445     return err
3446 }
3447
3448 // If the master password does not match we probably never in the user password
3449 // is correct
3450 // We have to be strict on this match, once master password has the specific
3451 // command in signature
3452 // If we look the master password is valid (its Cmd) {
3453     return errors.New("password: please provide the user password with '-ow'")
3454 }
3455
3456 // Generally the user password, which is also regarded as the Master Password or
3457 // an Permissions Password
3458 // We can't use the existing user password change is an exception since it
3459 // requires both current passwords
3460 // If we look the user password is valid (its Cmd) {
3461     // delete (deletePermissions)
3462     if err := deletePermissions(ctx)
3463     if err == nil {
3464         return err
3465     }
3466     // if we look the user password is corrupted
3467     return errors.New("password: corrupted permissions after owp ok")
3468 }
3469
3470 return nil
3471 }
3472
3473 // validate the user password (ex. document open password)
3474 // If err == validatePermissions(ctx)
3475 if err == nil {
3476     return err
3477 }
3478 // if we look {
3479     // if we look the user password is valid (its Cmd) {
3480     return errors.New("password: please provide the correct password")
3481 }
3482
3483 //fmt.Printf("how ok: %s\n", ok)
3484
3485 return handleEmissions(ctx)
3486 }
3487
3488 // If we look the user password is valid (its Cmd) {
3489 if err := validatePermissions(ctx) {
3490     return err
3491 }
3492
3493 // If this file is not corrupted.
3494 if err == nil {
3495     return handleEmissions(ctx)
3496 }
3497
3498 // If this file is corrupted.
3499 // If we look the user password is valid (its Cmd) {
3500 if err := validatePermissions(ctx) {
3501     return err
3502 }
3503
3504 // If we want to encrypt this file.
3505 return errors.New("password: this file is already encrypted")
3506 }
3507
3508 // Permissions (permissions)

```

```

3450 // Use the stream object to get the expectedDecompress
3451 // if found {
3452     decompressor = a < decompressors[Array]
3453     if (a < 0) {
3454         return nil, errors.New("pdpic: parserPipeline: expected decompress
3455             array corrupt")
3456     }
3457 }
3458
3459 // Parse the "decompressors" list, decompressors
3460
3461 filterPipeline, err = buildFilterPipeline(cta, filterArray, decompressor,
3462     decompressors)
3463
3464 log.Read.Println("pdpicFilterPipeline: end")
3465
3466 return filterPipeline, err
3467
3468 // streamOfStreamObject(cta, content, d, objName, streamLen, streamOffset,
3469 // offsetStart, offsetStream, d, content, objName)
3470 streamLength, streamOffset = d.Length()
3471
3472 if streamLength < 0 {
3473     return nil, errors.New("pdpic: streamOfStreamObject: stream object without
3474         streamLength")
3475 }
3476
3477 filterPipeline, err = pdpicFilterPipeline(cta, d)
3478 if err == nil {
3479     return nil, err
3480 }
3481
3482 streamOffset += offset
3483
3484 // We have a stream object.
3485 // We need to find the stream object, streamLength, streamOffset, filterPipeline
3486 // and streamOfStreamObject(cta, content, d, objName, streamLen, streamOffset,
3487 // offsetStart, offsetStream, d, content, objName)
3488
3489 return d, nil
3490
3491 // decodeContent(cta, d, objName, objName, streamLen, streamOffset, d) (d,
3492 // error)
3493
3494 if ct.IsEncrypted() == nil {
3495     // If it is not encrypted, then we can return the content as is.
3496     ct, err := d.DecodeContent(cta, objName, streamLen, ct.Addressing,
3497         ct.IsEncrypted())
3498     if err == nil {
3499         return nil, err
3500     }
3501 }
3502
3503 if streamLen > 0 && (streamLen < 0 || streamLen > streamLen) {
3504     log.Read.Println("decodeContent: end, objName, objName")
3505     return nil, err
3506 }
3507
3508 }

```

```

1800 func decompressStream(s *Stream, dst []byte, dstLen int, decode bool) (err error) {
1801     log.Debug.Printf("saveDecompressedStream: begin decode %v", decode)
1802
1803     if !(*IsIdentityCryptoFilter is used we do not need to decode) {
1804         if ctx := s.ctx; ctxFilter == nil {
1805             if !isDecryptedStream(s) == nil {
1806                 s.ctxFilter = filterFactory.Pipeline(s).Name = "Crypto"
1807                 s.ctxFilter = sd.New
1808                 return nil
1809             }
1810         }
1811     }
1812
1813     // Special case: if the length of the second data is 0, we do not need to decode
1814     // the stream.
1815     if len(s.data) == 0 {
1816         sd.Content = sd.Raw
1817         return nil
1818     }
1819
1820     // If ctx is created after the stream parsing.
1821     if ctx := s.ctx; ctxFilter == nil {
1822         if !isDecryptedStream(s, objKey, genKey, ctx.EncKey, ctx.HdrFields,
1823             ctx.Ex) {
1824             return err
1825         }
1826         l := int64(s.ctx.Len)
1827         sd.ContentLength = 0
1828     }
1829
1830     if decode {
1831         return nil
1832     }
1833
1834     // Actual decoding of content stream.
1835     err = decompressStream(s)
1836     if err == filter.ErrEmptyContentFilter {
1837         err = nil
1838     }
1839     if err == nil {
1840         return err
1841     }
1842
1843     log.Debug.Printf("saveDecompressedStream: end")
1844
1845     return nil
1846 }
1847
1848 // DecompressStream helper function
1849 func decompressStream(s *Stream, objKey string, genKey string, ctx *Context,
1850     objFields []string, entryObjFields []string) (err error) {
1851     log.Debug.Printf("decompressStream: decompress object %v to %v", objKey,
1852         entryObjFields)
1853
1854     // Assume the filter is referenced object stream.
1855     objFields = objFields[:len(objFields)-1]
1856     return decompressStream(s, objKey, genKey, ctx, objFields, entryObjFields)
1857 }

```

[illegible]

```

3457:20      (function(e,r,g){var t=document.getElementsByTagName("script");
3458:20      d.err = new ReferenceError("c", t[t.length-1].value)}
3459:20      if(err === null)
3460:20          return err
3461:20      }
3462:20      log.read.Print("haha", d)
3463:20      }
3464:20      // do read it except this file is under to read it.
3465:20      return setTimeout(function(){}, d)
3466:20      }
3467:20      }
3468:20      }

```

[illegible][illegible]

```

2200 //
2201 // processArrayRefs(shaTable, xrefTable, n Array) {
2202 //     for i = 0; i < n; i++
2203 //         switch (a = n.Type) {
2204 //             case IndirectRef:
2205 //                 obj := shaTable.FindAddressForIndirectRef(a)
2206 //                 if obj != nil
2207 //                     entry.RefCount++
2208 //             case Blob:
2209 //                 processRefsCounts(shaTable, obj)
2210 //             case Array:
2211 //                 processRefsCounts(shaTable, obj)
2212 //         }
2213 //     }
2214 // }
2215 //
2216 // processArrayRefs(shaTable, xrefTable, o Object) {
2217 //     switch o := o.Type {
2218 //     case Slice:
2219 //         processRefsCounts(shaTable, o)
2220 //     case StreamInit:
2221 //         processRefsCounts(shaTable, o.Object)
2222 //     case Array:
2223 //         processArrayRefs(shaTable, o)
2224 //     }
2225 // }
2226 //
2227 // //References all objects including compressed objects from object streams.
2228 // func dereferenceObjects(cxs <Context> error) {
2229 //     log.Debug.Println("dereference objects begin")
2230 //     shaTable := cxs.shaTable
2231 //     // Get sorted slice of object numbers.
2232 //     // From this sorting for performance gain.
2233 //     var keys []int
2234 //     for k := range shaTable.Table {
2235 //         keys = append(keys, k)
2236 //     }
2237 //     sort.Ints(keys)
2238 //     for i, objNr := range keys {
2239 //         err := dereferenceObject(cxs, objNr)
2240 //         if err != nil {
2241 //             return err
2242 //         }
2243 //     }
2244 //     for i, objNr := range keys {
2245 //         entry := shaTable.Table[objNr]
2246 //         if entry.Ref == entry.Compressed {
2247 //             continue
2248 //         }
2249 //         processRefsCounts(shaTable, entry.Object)
2250 //     }
2251 // }

```

[illegible]

```

2620         log.ReadPrintln("logstream: no object's readout!")
2621     }
2622 }
2623
2624 // Decode all object streams no contained objects are ready to be used.
2625 func decodeObjStreams(ctx context.Context) error {
2626     // Note:
2627     // dirty "streams" intentionally left out.
2628     // No object stream collision validation necessary.
2629     log.ReadPrintln("decoding objStreams: begin!")
2630
2631     // Get sorted slice of object numbers.
2632     objNums := ObjList.List
2633     for k, v := range ctx.ReadObjStreams() {
2634         log.Debugf("objStreams: %v", v)
2635     }
2636     sort.Ints(objNums)
2637
2638     for _, objNumber := range objNums {
2639         // Get object's stream.
2640         entry := ctx.GetTableEntry("table[objNumber]")
2641         if entry == nil {
2642             return errors.New("objNumber: missing entry for objNum")
2643         }
2644         objNumber := entry.Number
2645
2646         log.ReadPrintln("decoding objStreams: parsing object stream for objNum",
2647             objNumber)
2648
2649         // Decode object stream from file.
2650         objStream, err := ParseObjStream(entry.Offset, objNumber, entry.Generation)
2651         if err == nil {
2652             return errors.New("objNumber: decoding objStreams: corrupt object stream")
2653         }
2654
2655         // Return StreamError
2656         if ok := objStream.IsErr(); ok {
2657             return errors.New("objNumber: decoding objStreams: corrupt object stream")
2658         }
2659
2660         // Load decoded stream content to memTable.
2661         if err := objStream.LoadToMemTable(ctx, objNumber, entry.Generation,
2662             return errors.Wrapf(err, "decoding objStreams: problem dereferencing
2663             object stream %v, objNumber")
2664
2665         // Save decoded stream content to memTable.
2666         if err := objStream.SaveToMemTable(ctx, objNumber, entry.Generation,
2667             return errors.Wrapf("objNum: %v", objNumber, err)
2668         }
2669     }
2670 }

```

```

2620         }
2621         log.Debug.Print("dereferenceObjects: end")
2622     }
2623     return nil
2624 }
2625
2626 // Locate a possible version string, (since v1.0 in the calling
2627 // and record this as a reservation (as opposed to headerVersion),
2628 // and identify if the version is different to headerVersion() error {
2629 func IdentifyVersion(sha1able *SHA1able) error {
2630     log.Debug.Print("IdentifyVersion: begin")
2631
2632     // try to get version from root
2633     rootVersion := sha1able.RootVersion()
2634     if err := nil
2635     {
2636         return err
2637     }
2638     if rootVersion != nil
2639     {
2640         if rootVersionStr := nil {
2641             return nil
2642         }
2643
2644         // validate version and return corresponding content to sha1able,
2645         // otherwise, err = nil
2646         if err := nil
2647         {
2648             sha1able.RootVersionStr = sha1able.RootVersion()
2649             sha1able.RootVersionStr = sha1able.RootVersion()
2650         }
2651     }
2652     sha1able.RootVersion = sha1able.RootVersion
2653
2654     // validate the sha1able version may be overridden by a version entry in the
2655     // sha1able
2656     if sha1able.HeaderVersion() != nil
2657     {
2658         log.Info.Printf("IdentifyVersion: PDR version is %s - will ignore root
2659         version: %s",
2660             sha1able.HeaderVersionStr, rootVersionStr)
2661     }
2662     log.Debug.Print("IdentifyVersion: end")
2663 }
2664
2665     return nil
2666 }
2667
2668 // Parse all objects including object stream from file and save to the corresponding
2669 // HeaderFile
2670 // Parse all objects processing object streams and linearization dicts.
2671 func dereferenceObjects(ctx *Context, conf *Configuration) error {
2672     log.Debug.Print("dereferenceObjects: begin")
2673
2674     sha1able := ctx.sha1able
2675
2676     // Note for unencrypted files
2677     // Requires permission access to open & display file.
2678     // Access may be restricted (insecure access privileges).
2679     // Access may be restricted to the file's public access privileges.
2680     err := checkDecryptionStatus()
2681 }

```

```

3531 public static void main(String[] args) {
3532     // If err is null {
3533         return nil, err
3534     }
3535     return StringLiteral(string(bb)), nil
3536 }
3537
3538 default:
3539     return o, nil
3540 }
3541
3542 func dereferenceObject(cx *Context, objObjectVar int) (Object, error) {
3543     entry, ok := cx.Find(objObjectVar)
3544     if !ok {
3545         return nil, errors.New("pdcpu: dereferenceObject: unregistered object")
3546     }
3547     if entry.Contained {
3548         err := dereferenceTable(entry.cx.XrefTable, objObjectVar, entry)
3549         if err != nil {
3550             return nil, err
3551         }
3552     }
3553     if entry.Object == nil {
3554         log.BadPrintf("dereferenceObject: dereferencing object %d\n", objObjectVar)
3555         o, err := ParseObject(entry, entry.Offset, dereferencing, entry.Generation)
3556         if err != nil {
3557             return nil, errors.Newf("pdcpu: dereferenceObject: problem dereferencing object %d", objObjectVar)
3558         }
3559         return o, objObjectVar
3560     }
3561     if o == nil {
3562         return nil, errors.Newf("pdcpu: dereferenceObject: object is nil")
3563     }
3564     entry.Object = o
3565     return entry.Object, nil
3566 }
3567
3568 func dereferenceInteger(cx *Context, objObjectVar int) (Integer, error) {
3569     o, err := dereferenceObject(cx, objObjectVar)
3570     if err != nil {
3571         return nil, err
3572     }
3573     if o == nil {
3574         return nil, errors.Newf("pdcpu: dereferenceObject: corrupt integer")
3575     }
3576     i, ok := o.Integer()
3577     if !ok {
3578         return nil, errors.Newf("pdcpu: dereferenceObject: corrupt integer")
3579     }
3580 }

```

```

3830         //Decompress (in Object stream class)
3831         if (isDecompressing) {
3832             return error.Wrapped("decompressObjectStream: corrupt object stream")
3833         }
3834
3835         // Use the object stream
3836         log.Read.Printf("decompressObjectStream: object stream %d\n", objStreamNum)
3837         ctx.readUsingObjectStream = true
3838
3839         // Create new object stream dict
3840         obj_stream = ObjectStreamDict()
3841         obj_err = error.Wrapped(err, "decompressObjectStream: problem dereferencing object stream %d, objStreamNum")
3842         if err == nil {
3843             return error.Wrapped(err, "decompressObjectStream: problem dereferencing object stream %d, objStreamNum")
3844         }
3845
3846         log.Read.Printf("decompressObjectStream: decoding object stream %d\n", objStreamNum)
3847
3848         // Parse all objects of this object stream and save them to ObjectStreamDict
3849         if err = parseObjectStream(objStreamNum) {
3850             return error.Wrapped(err, "decompressObjectStream: error parsing object stream %d", objStreamNum)
3851         }
3852         return error.Wrapped(err, "decompressObjectStream: problem decoding object stream %d", objStreamNum)
3853     }
3854
3855     if objStreamNum == nil {
3856         return error.Wrapped(err, "decompressObjectStream: objStream should be set")
3857     }
3858     log.Read.Printf("decompressObjectStream: decoded object stream %d\n", objStreamNum)
3859
3860     // Save object stream dict to the file/fatality.
3861     entry.Object = *objStreamDict
3862     log.Read.Printf("decompressObjectStream: end")
3863
3864     return nil
3865 }
3866
3867 // Save linearization information to the context, obj, object, objErr error
3868 func handleLinearizationInfo(context *Context, obj Object, objErr error) {
3869     if objErr != nil {
3870         log.LinearizationDictAlreadyProcessed()
3871         return nil
3872     }
3873
3874     // Handle Linearization from root dict
3875     if obj.Root == obj.Dict {
3876         obj.LinearizationInfoFromRootDict()
3877     }
3878
3879     if obj.LinearizationDict == true {
3880         ctx.linearizationInfoFromRootDict = true
3881     }
3882     log.Read.Printf("handleLinearizationInfoFromRootDict identified linearizationObj %d\n", obj.Root)
3883
3884     a := A.NewArrayPrint("H")
3885 }

```

[illegible]