





[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 %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\t%d\n", count,
1788             len(buf), totalCount)
1789         totalCount += count
1790     }
1791     return buf, nil
1792 }
1793
1794 func readStreamFromReadContentStream(r io.Reader) ([]byte, error) {
1795     log.Read.Printf("readContentStream: end\n")
1796     return buf, nil
1797 }

```

```

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 object stream content for stream dict into writable entry.
377         if err := loadContentFromStreamContentDict(ctx, sd); err != nil {
378             return errors.Errorf("referenceObjectDict: problem dereferencing stream '%d'",
379                 objNr)
380         }
381         ctx.Read.BinaryTotalSize += sd.StreamLength
382         // Read from stream
383         if s, ok := loadContentFromStreamContentDict(ctx, sd, objNr, genIn, ctx.DecomallStream);
384             !ok {
385             return err
386         }
387     }
388     n := updateBinaryTotalSize(ctx.Context, obj) {
389         switch o := n.(type) {
390             case StreamDict:
391                 ctx.Read.BinaryTotalSize += sd.StreamLength
392         }
393     }

```

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

```

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, content, streamOffset, d) (d, Content, err)
3492
3493 if ct.IsBinary() == nil {
3494     if err := decodeContent(cta, objName, content, ct.Addressing,
3495         ct.IsBinary, ct.IsBinary); err != nil {
3496         return nil, err
3497     }
3498 }
3499
3500 if content < 0 || (streamed < 0 || streamed > content) {
3501     log.Read.Println("ctd: decodeContent, objName, objName")
3502     return nil, err
3503 }
3504
3505 }

```

[illegible][illegible]

```

3457:20      (function(e,t,r,n,i){
3458:   d.err = new ReferenceError("c", {ObjectNumber: value})
3459:   if err = null
3460:     return err
3461:   }
3462:   log.Read.Print("h", d)
3463: }
3464:
3465: // do read to ensure this file is written to read it.
3466: return actualCryllLinkKey(c, d)
3467:
3468:

```

[illegible]

```

1870         return errors.Errorf("compressorTabFactory: problem dereferencing object '%s'",
1871                                objStr, entry.GetObjectStream(),
1872                                objStr, entry.GetObjectStream())
1873
1874     // Set the obj stream to be obj's objectStream.
1875     objStr = obj.GetObjectStream().TabFactory().obj(ObjectStreamFactory)
1876     if !ok {
1877         return errors.Errorf("compressorTabFactory: problem dereferencing object '%s'",
1878                                objStr, obj.GetObjectStream(),
1879                                objStr, obj.GetObjectStream())
1880     }
1881
1882     // Get indexed object from obj's objectStream.
1883     o, err := h.LookupObj(entry.GetObjectStream())
1884     if err != nil {
1885         return errors.Errorf("compressorTabFactory: problem dereferencing object stream '%s'",
1886                                entry.GetObjectStream(),
1887                                entry.GetObjectStream())
1888     }
1889
1890     // Save object to obj's TabFactory.
1891     objStr = obj
1892     entry.Object = o
1893     entry.ObjectStream = objStr
1894     entry.Compressed = false
1895
1896     log.Printf("compressorTabFactory: end, obj %d obj's %d", objStr,
1897                entry.GetObjectStream(), entry.GetObjectStream(), o)
1898
1899     return nil
1900 }
1901
1902 // Log interesting stream content.
1903 func logStream(o Object) {
1904     switch o := o.(type) {
1905     case StreamDict:
1906         if o.Content == nil {
1907             log.Printf("logStream: no stream content")
1908         }
1909         if o.IsPageContent() {
1910             log.Printf("logStream: content object's %s, streamDict's Content",
1911                        o.GetObjectStream(), streamDict.Content)
1912         }
1913     case ObjectStreamDict:
1914         if o.Content == nil {
1915             log.Printf("logStream: no object stream content")
1916         } else {
1917             log.Printf("logStream: logStream: content = %s", o.Content)
1918         }
1919     case ObjectStream:
1920         if o.Object == nil {
1921             log.Printf("logStream: no object stream obj arr")
1922         } else {
1923             log.Printf("logStream: objectStream object = %s", o.Object)
1924         }
1925     }
1926 }
1927
1928 // Defaults.

```

```

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(cxt *Context, entry *Object) {
2229 //     log.Debug.Println("dereferenceObjects", entry)
2230 //     shaTable := cxt.shaTable
2231 //     // Get source slice of object numbers.
2232 //     // from shaKey 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(cxt, 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 // }

```

```

15902  obj.contentType = generateContentType(); obj = parseObject(cta.headers(obj));
15903  if (err || null != obj)
15904      return nil, 0, 0, 0, err;
15905
15906  if obj.contentType != "application/json" {
15907      return nil, 0, 0, 0, fmt.Errorf("object has non matching obj.contentType")
15908  }
15909  // object has matching contentType, parse object
15910  obj, err = parseObject(obj)
15911
15912  return obj, err
15913 }
15914
15915 func ParseStreamObject(cta *Context, offset, objObj, streamObj, streamOffset int) (Object, error) {
15916     log.Debugf("ParseStreamObject: begin, objObj, streamObj, offset=%d", offset)
15917     obj, err := parseStreamObject(cta, offset, objObj, streamObj, streamOffset)
15918     if err != nil {
15919         return nil, err
15920     }
15921     return obj, nil
15922 }
15923
15924 func (obj *Object) isObject() bool {
15925     switch obj := obj.(type) {
15926     case Dict:
15927         // obj is dict (ct, o, objObj, streamObj, streamOffset)
15928         if err := nil || dict == nil {
15929             return false
15930         }
15931         return true
15932     // StreamObj
15933     case StreamObject:
15934         if err := nil || streamObj == nil {
15935             return false
15936         }
15937         return true
15938     case Array:
15939         if err := nil || array == nil {
15940             return false
15941         }
15942         if err := nil || arrayObj == nil {
15943             return false
15944         }
15945         return true
15946     case StringLiteral:
15947         if err := nil || stringLiteral == nil {
15948             return false
15949         }
15950         if err := nil || stringLiteralObj == nil {
15951             return false
15952         }
15953         return true
15954     case HexLiteral:
15955         if err := nil || hexLiteral == nil {
15956             return false
15957         }
15958         if err := nil || hexLiteralObj == nil {
15959             return false
15960         }
15961         return true
15962     }
15963     return false
15964 }

```

```

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(err, "objNum: %v", objNumber, err)
2668         }
2669     }
2670 }

```

[illegible]

```

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.New("Integer")
3575     }
3576     if !ok {
3577         return nil, errors.New("pdcpu: dereferenceObject: corrupt integer")
3578     }
3579 }

```

```

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     return nil
3864 }
3865
3866 // Read the linearization info from the object stream
3867 func readLinearizationInfoFromObjectStream(ctx Context, obj Object, objErr error) (
3868     linearizationInfo *LinearizationInfo, err error) {
3869     if ctx.Read.Linearized {
3870         linearizationInfo dict already processed.
3871         return nil
3872     }
3873
3874     // Read the linearization info from the object stream
3875     if err = readLinearizationInfoFromObjectStream(ctx, obj, objErr) {
3876         return nil, err
3877     }
3878     if ctx.Read.Linearized == true {
3879         linearizationInfo = &ctx.LinearizationInfo
3880     }
3881     return linearizationInfo, nil
3882 }
3883
3884 // Read the linearization info from the object stream
3885 func readLinearizationInfoFromObjectStream(ctx Context, obj Object, objErr error) (
3886     linearizationInfo *LinearizationInfo, err error) {
3887     if err = readLinearizationInfoFromObjectStream(ctx, obj, objErr) {
3888         return nil, err
3889     }
3890     if ctx.Read.Linearized == true {
3891         linearizationInfo = &ctx.LinearizationInfo
3892     }
3893     return linearizationInfo, nil
3894 }
3895
3896 // Read the linearization info from the object stream
3897 func readLinearizationInfoFromObjectStream(ctx Context, obj Object, objErr error) (
3898     linearizationInfo *LinearizationInfo, err error) {
3899     if err = readLinearizationInfoFromObjectStream(ctx, obj, objErr) {
3900         return nil, err
3901     }
3902     if ctx.Read.Linearized == true {
3903         linearizationInfo = &ctx.LinearizationInfo
3904     }
3905     return linearizationInfo, nil
3906 }

```

```

2018         return err
2019     }
2020     //Get Permission for authentication object.
2021     //Prepare decompressed object.
2022     err = decompressObject(streams[ctx])
2023     if err == nil {
2024         return err
2025     }
2026     //For each shareability assign a object either by parsing from file or passing
2027     //decompressed object.
2028     err = deferencetorefsObject(ctx)
2029     if err == nil {
2030         return err
2031     }
2032     //Identify an optional version entry in the root object/catalog.
2033     //Identify ObjectVersionInfo.ShareFile()
2034     if err == nil {
2035         return err
2036     }
2037     log.Debug.Println("deferencetorefsFile: end")
2038 }
2039 func main() {
2040     log.Read.Println("deferencetorefsFile: end")
2041 }
2042
2043 func handleOnencryptedFile(ctx *Context) error {
2044     if ctx.Cmd == DECRYPT || ctx.Cmd == SETPERMISSIONS {
2045         return errors.New("context: this file is not encrypted")
2046     }
2047     if ctx.Cmd != ENCRYPT {
2048         return nil
2049     }
2050 }
2051
2052 // Encrypt subcommand found.
2053
2054 if ctx.Command == "" {
2055     if ctx.Options.W == "" {
2056         return errors.New("password: please provide owner password and optional user password")
2057     }
2058     return nil
2059 }
2060
2061 func initObj(ctx *Context) (id []byte, err error) {
2062     if ctx.ID == "" {
2063         return nil, errors.New("id entry")
2064     }
2065     h1, ok := ctx.ID[0].DecodeHex()
2066     if ok {
2067         id = h1.Bytes()
2068     } else {
2069         return nil, err
2070     }

```