



Context

Hans Hagen

Contents

Introduction	4
1 Annotations	6
2 Enabling	8
3 Actions	12
4 Hyperlinks	18
5 Structure	20
6 Comments	22
7 Attachments	26
8 Bookmarks	30
9 JavaScript	32
10 Buttons	34
11 Menus	36
12 Progress	38
13 Widgets	40
14 Page transitions	42
15 Importing	44
16 Tagging	46

Introduction

This document introduces the cross reference mechanism, viewer control, fill-in fields, JavaScript support, comments, attachments and more. It is a rewrite of the MkII widgets manual. There is (always) more than discussed in manuals so if you miss something, take a look at test suite or when you're brave, peek into the source code as there can be examples there.

Interactivity has always been available in ConT_EXt and in fact it was one of the reasons for writing it. In for instance the YandY Windows previewer, one could have hyperlinks and we used that for a while when checking documents. Later Acrobat showed up and pdf stepwise added interactive features that we always supported right from the start. Unfortunately there is a viewer dependency and the documentation of pdf lagged behind so solutions based on trial and error could not work well in a follow up on pdf. Some features disappeared or became so limited that they effectively became useless. Especially multi-media have a reputation of unreliability. Because open source viewers never really caught up (at least not in this area) the momentum was lost to make sure that documents could have audio and video embedded in reliable ways. Even forms and basic JavaScript control of for instance layers is often missing.

That said, we do support a lot and can support more when it makes sense. Deep down in ConT_EXt we always had the mechanisms to deal with this, so extensions are not that hard to program. We always thought that publishers would like these features but that never really was the cases, so there was no pressure from that end. Most features are user driven or just there because at some point we wanted to make some fancy presentation.

The original pdf was a couple of hundred pages and looked quite nice. A later print has manu more pages and still looks ok, but nowadays we have to do with a pdf document. If you want to see what pdf supports you can study this (now about) 750 pdf standard. It is, being an iso standard, not public, but you can probably find a (maybe older) copy someplace on the web.

When reading this manual you need to keep in mind that we assume that you design a decent layout and when you make something for an electronic medium, we assume that you pay attention to the way you can enhance accessibility.

If you miss something here, don't hesitate to ask for clarification, or even better, provide an example that we then can use to discuss (an aspect of) some mechanism.

1 Annotations

Before we discuss interactive features (in following chapter) a few words will be spent on so called annotations. This term is used in Acrobat and is somewhat confusing as hyperlinks conceptually are not really annotations while comments are. The term relates to the way pdf files can have added functionality. It might help understand the following chapters better when you know what model is used inside a pdf.

If you open a pdf file in an editor you will find lots of objects. Numbers are an object, as are strings and booleans. Symbols (represented as strings with a leading slash) are also objects. Objects can be collected in indexed tables (arrays) and hash tables (dictionaries). Serialized arrays are bounded by square brackets:

```
[ (value1) (value2) ]  
[ 1 2 3 ]  
[ 1 2 (value1) (value3) true /foo ]
```

and hashes by double angle brackets:

```
<<  
  /Key1 (value1)  
  /Key2 (value2)  
  /Key3 123  
  /Key4 true  
  /Key5 [ 1 2 3 4 ]  
>>
```

A pdf file is a collection of objects:

```
1 0 obj
```

```
endobj
```

Instead of a number, string, boolean, array or dictionary an object can also be a stream of bytes:

```
1 0 obj << /Length 123 >>  
stream  
... 123 bytes ...  
endstream
```

endobj

Objects can refer to each other and can be looked up via a so called xref table. We refer to object with number one and revision zero as follows:

```
/foo 1 0 R
```

When an object is updated it can be added to the end of the file and the version number can get bumped. A program that does something with the pdf is supposed to do something clever with these numbers. More often the revision stays zero.

A document is normally a sequence of pages. When a file is opened the cross reference table is loaded and the so called catalog is looked up. From there pages can be found. Pages have a content stream and can refer to resources, like fonts, special color spaces, complex objects (xforms) and among other things annotations.

The page is rendered from the content stream but that stream has no information about hyperlinks and such. The /Link annotation objects that implement interactivity are independent and kind of layered over the rendered page. They describe rectangular areas that a viewer can use to intercept mouse clicks. If you want to see where the actions happens, search for /Dest and /Annot in an (uncompressed) pdf file.

The main reason for mentioning these details is that when you are checking a file for problems you need to look for annotation objects instead of the page stream. You also need to realize that these annotations in no way interfere with the page stream. They only exist because a viewer can use that information to add functionality. Their reference point is the lower lefty corner of the page. This is a conceptual difference with html where hyperlinks are often in-line and part of the content stream. Both approaches have their advantages and disadvantages. From the perspective of quality typesetting it makes much sense to have them overlayed and explicitly defined (in terms of dimensions) but users will of course in most cases define them inline. This means that in order to make the pdf some analyzing and juggling has to take place. In ConT_EXt we always have done as much as possible at the T_EX (therefore bypassing some limitations in the engine) end in MkIV we don't use the engine's features at all.

2 Enabling

Interaction is turned off by default. Of course cross referencing work without interaction but there are no hyperlinks. You turn on interaction with the `\setupinteraction` command:

```
\setupinteraction [...,1...] [...,2...]
                        OPT
1  NAME
2  state          = start stop
   style          = STYLE COMMAND
   color          = COLOR
   contrastcolor  = COLOR
   title          = TEXT
   subtitle       = COLOR
   author         = TEXT
   date           = TEXT
   keyword        = TEXT
   focus          = standard frame width minwidth height minheight fit
                  tight
   menu           = on off
   fieldlayer     = auto NAME
   calculate      = REFERENCE
   click          = yes no
   display        = normal new
   page           = yes no page name auto
   openaction     = REFERENCE
   closeaction    = REFERENCE
   openpageaction = REFERENCE
   closepageaction = REFERENCE
   symbolset      = NAME
   height         = DIMENSION
   depth          = DIMENSION
   focusoffset    = DIMENSION
```

The state key is the switch you need to use. In addition you might want to setup the style and color.

```
\setupinteraction
  [state=start,
   style=,
   color=,
   contrastcolor=]
```

This is the least intrusive way to get interaction in your document. By default the style is bold and the color defaults to green. The `contrastcolor` is used when a hyperlink refers to the same page and defaults to red. A neutral setup makes sense because nowadays the reader kind of knows what can be clicked on.

The `title`, `subtitle`, `author`, `date` and `keyword` parameters are passed to the document and will show up when you request document information.

The `openaction` parameter can for instance be used to start at a specific page, while the `closeaction` can for instance trigger a JavaScript cleanup script. The `openpageaction` and `closepageaction` can for instance initialize and reset states, something we do in some presentation styles.

The `click` parameter controls how a viewer responds to pressing a mouse button on an annotation: highlight or not. The `display` parameter determines if a cross document link opens in the current window.

The `menu` parameter is a quick way to disable menus, of which there can be many: at each side of the page, stacked or not, etc. The `symbolset` determines the look and feel of symbols used in for instance navigational buttons and status bars.

The `page` parameters is a bit special, and its function is an inheritance from the early days. Some dvi and pdf viewers supported named destinations, others only page references. This parameter can be used to force one or the other. There was a time that there was a limit on the number of named references, so going page was the only option¹

Personally I consider an electronic document an entity to be seen full screen on a dedicated device. However some users prefer the target of a link to fit the width of the screen and alike. The `focus` parameter can (within) reasonable bounds provide this. The `focusoffset` is then used to keep things a bit visual convenient.

The `height` and `depth` parameters are sort of special and probably never used. When we go back in time, to when we started adding interactivity, there were a few issues that needed to be dealt with:

¹ We're talking of 1995 when we made documents of many thousands of pages with tens of thousands of hyperlinks, cross linked tables of contents, registers, active graphics, etc. Think of dictionaries used in very specific projects, or quality assurance manuals.

- We need to make sure that we have something to click on, so we need to add some offset if needed.
- We need to handle nested hyperlinks, which is why ConT_EXt didn't use the link features of for instance pdfT_EX but built its own.
- Hyperlinks should break properly across lines without side effects, again a reason for bypassing some of the T_EX engine's behaviour.
- We have to make sure that there is at least a consistent height and depth of hyperlinks. These tight links with viewer supplied bounding boxes to click on just look real bad! So, we had to do better.

Normally the two mentioned parameters are not used. However, their value will kick in when we say `\setfalse \locationstrut`, in which case the given height and depth will be used. Some advice: don't mess with this. We only have this because it permits special effects.

If you want to see what the target (destinations) and sources (references) of links are, you can say:

```
\enabletrackers[nodes.references,nodes.destinations]
```

The `fieldlayer` parameter can be used to set a so called viewer layer, so that you can hide them (given that a viewer supports that). The `calculate` parameter can associate a calculator (initializer) with the fields.

You can create an interaction environment with:

```
\defineinteraction [1...] [2...] [3...OPT...OPT...]
1 NAME
2 NAME
3 inherits: \setupinteraction
```

which then can be used with:

```
\startinteraction [...]* ... \stopinteraction
* hidden NAME
```


3 Actions

The reference mechanism not only deals with the more traditional cross references, but also takes care of navigational, launching applications (although that is often limited by viewers), running JavaScript, etc. By integrating these features in one mechanism, we limit the number of commands needed for hyper-references, menus and buttons. Normally such actions are driven by the `\goto` command, but you can also use buttons:

```
\goto[inner reference]
\goto[outer reference::]
\goto[outer reference::inner reference]
```

The inner reference is normally a user defined one, for instance a reference to a named location like a chapter or figure. The outer reference refers to a file or url and is normally defined at the document level and is accessed by the `::`. By using symbolic names updating them becomes easier.

There are also predefined references, like `previouspage` to go to the next page or `forward` to cycle, `nextcontents` for the next level table of contents in a linked list of such tables, etc. Some keywords are actually shortcuts to actions that are delegated to the viewer. Here you need to keep in mind that nowadays we're talking of pdf viewers, but originally (MkII) we also supported dvi viewers. A special class of references are the viewer control ones, like `CloseDocument` or `PreviousJump`. They can be recognized by their capitals.

When we speak of a reference, we actually refer to a whole bunch of possible references. We already mentioned inner and outer references, but special actions are also possible. These are actually plugins. Examples are the JavaScript and url plugins. The interface evolved over a few decades but most has been there right from the start, which is why we keep it as is. Actually, there is not that much new functionality added in MkIV, although the implementation was mostly rewritten. Here is an overview of the syntax, just to give you an idea.

```
\goto[inner]
\goto[inner(foo,bar)]
\goto[inner{argument,argument}]
\goto[inner{argument}]
\goto[outer::]
\goto[outer::inner]
\goto[outer::special(operation{argument,argument})]
```

```

\goto[outer::special(operation)]
\goto[outer::special()]
\goto[outer::inner{argument}]
\goto[special(operation{argument})]
\goto[special(operation{argument,argument})]
\goto[special(operation)]
\goto[special(outer::operation)]
\goto[special(operation)]
\goto[special(operation(whatever)))]
\goto[special(operation{argument,argument{whatever}})]
\goto[special(operation{argument{whatever}})]
\goto[special(operation{"argument,argument{whatever}"})]
\goto[special(operation{"argument{whatever}"})]

```

There can be multiple actions, separated by a comma, think of: go to the page with label ‘foo’ and start video ‘bar’.

```

\goto {1...} [2...]

```

1 CONTENT

2 REFERENCE

Examples of operations are page, program, action, url and JS. The page operation accepts a pagenumber as well as relevant keywords. One can prefix a pagenumber by a file or url tag. The program operation starts up a program. It is an example of an old feature that has proven to be unstable, simply because viewers change behaviour over time.

```

\definereference [1...] [2...,...]

```

1 NAME

2 REFERENCE

The built-in actions are interfaces via shortcuts with camelcase names. In most cases the name is a good indication of what to expect:

```

\definereference [CloseDocument]      [action(close)]
\definereference [ExitViewer]         [action(exit)]

```

\definereference [FirstPage]	[action(first)]
\definereference [LastPage]	[action(last)]
\definereference [NextJump]	[action(forward)]
\definereference [NextPage]	[action(next)]
\definereference [PauseMovie]	[action(pausemovie)]
\definereference [PauseSound]	[action(pausesound)]
\definereference [PauseRendering]	[action(pauserendering)]
\definereference [PreviousJump]	[action(backward)]
\definereference [PreviousPage]	[action(previous)]
\definereference [PrintDocument]	[action(print)]
\definereference [SaveForm]	[action(exportform)]
\definereference [LoadForm]	[action(importform)]
\definereference [ResetForm]	[action(resetform)]
\definereference [ResumeMovie]	[action(resumemovie)]
\definereference [ResumeSound]	[action(resumesound)]
\definereference [ResumeRendering]	[action(resumerendering)]
\definereference [SaveDocument]	[action(save)]
\definereference [SaveNamedDocument]	[action(savenamed)]
\definereference [OpenNamedDocument]	[action(opennamed)]
\definereference [SearchDocument]	[action(search)]
\definereference [SearchAgain]	[action(searchagain)]
\definereference [StartMovie]	[action(startmovie)]
\definereference [StartSound]	[action(startsound)]
\definereference [StartRendering]	[action(startrendering)]
\definereference [StopMovie]	[action(stopmovie)]
\definereference [StopSound]	[action(stopsound)]
\definereference [StopRendering]	[action(stoprendering)]
\definereference [SubmitForm]	[action(submitform)]
\definereference [ToggleViewer]	[action(toggle)]
\definereference [ViewerHelp]	[action(help)]
\definereference [HideField]	[action(hide)]
\definereference [ShowField]	[action(show)]
\definereference [GotoPage]	[action(gotopage)]
\definereference [Query]	[action(query)]
\definereference [QueryAgain]	[action(queryagain)]
\definereference [FitWidth]	[action(fitwidth)]
\definereference [FitHeight]	[action(fitheight)]

```

\definereference [ShowThumbs]      [action(thumbnails)]
\definereference [ShowBookmarks]   [action(bookmarks)]
\definereference [HideLayer]       [action(hidelayer)]
\definereference [VideLayer]       [action(videlayer)]
\definereference [ToggleLayer]     [action(togglelayer)]

```

In the java-imp-*.mkiv files you will find examples of similar shortcuts, for instance:

```

\definereference [SetupStepper]    [JS(SetupStepper{step,50})]
\definereference [ResetStepper]    [JS(ResetStepper)]
\definereference [CheckStepper]    [JS(CheckStepper{\StepCounter})]
\definereference [InvokeStepper]   [JS(InvokeStepper)]

```

Other examples of redefined references are:

```

\definereference [firstpage]       [page(firstpage)]
\definereference [previouspage]    [page(previouspage)]
\definereference [nextpage]       [page(nextpage)]
\definereference [lastpage]       [page(lastpage)]
\definereference [forward]        [page(forward)]
\definereference [backward]       [page(backward)]
\definereference [firstsubpage]    [page(firstsubpage)]
\definereference [previoussubpage] [page(previoussubpage)]
\definereference [nextsubpage]     [page(nextsubpage)]
\definereference [lastsubpage]     [page(lastsubpage)]

```

Some of these actions expect arguments, for instance:

```

\goto{start}[StartMovie{mymovie}]

```

You can load the module references-identify which gives you a command:

```

\showreference[page(123),StartMovie{mymovie}]
\showreference[JS(Forget_Changes),CloseDocument]
\showreference[manual::contents]

```

1 reference page(123)
 kind special operation
 operation 123

	arguments	
	special	page
<hr/>		
2	reference	action(startmovie)
	kind	special operation with arguments
	operation	startmovie
	arguments	mymovie
	special	action
<hr/>		
1	reference	JS(Forget_Changes)
	kind	special operation
	operation	Forget_Changes
	arguments	
	special	JS
<hr/>		
2	reference	action(close)
	kind	special operation
	operation	close
	arguments	
	special	action
<hr/>		
1	reference	manual::contents
	kind	outer with inner
	operation	
	arguments	
	special	
<hr/>		

You should be careful with colons in references. This gives you an idea how Con-TeXt interprets what you requested.

prefix:whatever	The prefix creates a namespace. When references are resolved and there is no hit a lookup without prefix takes place.
document::whatever	The document is a symbolic reference to an external resource. This is explained elsewhere.
component:::whatever	The component is a symbolic reference to a component in a product. References defined in such a component are loaded on demand.

4 Hyperlinks

5 Structure

6 Comments

Many pdf viewers support text annotations. These are small notes that can be popped up. In ConT_EXt we call them comments, because often that's what they are used for. Comments evolved from simple ones using a limited encoding into more advanced ones with representations. A comment looks like:

```
\startcomment
  Hello beautiful world!
\stopcomment
```



When you open a document with comment you will likely see some symbol depicting it. But, it's one of those features that is viewer dependent so when it looks odd or unexpected, check in Acrobat first. The position and size can differ per viewer and when you zoom in the size can either stay the same or scale. The viewer can show the pop up text at the same location or someplace else. Although in principle there is control over this, my experience is that viewers (also Acrobat) keep changing this (not always for the best). Just assume the worst: it will never look good and although for a while we kept up with viewers, the inconsistency (and accumulated waste of time) led us to the current minimalistic approach.

By default, in ConT_EXt comments are placed at the spot a bit raised. In this document we put them in the margin, by saying:

```
\setupcomment
  [location=inmargin]
```

Comments can have titles and properties but not all viewers support properties. Contrary to other environments, the first argument is not a category but a title. This because we are compatible with MkII.

```
\startcomment[french]
  In France they use «these angle bracket glyphs» in subsentences.
\stopcomment
```

```
\startcomment[accents][color=darkgreen]
  You can use an àçéñţêő character too.
\stopcomment
```

And normally empty lines are also supported (again this can differ per viewer):

```
\startcomment[lines][color=darkblue]
```

How about an

empty line?

```
\stopcomment
```



And normally empty lines are also supported (again this can differ per viewer):



As we can see here, comments are sort of stacked. These examples also show that we can pass an optional title and set up some characteristics. An inline comment is defined with `\comment`:

```
\comment {How I hate those notes spoiling the layout.} Maybe  
some day
```

```
I can convince myself to add some features \comment {Think of  
comment classes
```

```
that can be turned on and off and get their own colors.} related  
to version  
control.
```



Maybe some day I can convince myself to add some features related to version control. Comments hide part of the text and thereby are to be used with care. Until now I never used them. Anyhow, from now on, one can happily use:

You can use other symbols than the default, and a couple are predefined in the standard: `Comment`, `Help`, `Insert`, `Key`, `Newparagraph`, `Note`, `Paragraph`.

You can also use your own symbols:

```
\startuniqueMPgraphic{cow}{height,s:color}  
  loadfigure "cow.mp" number 1 ;  
  refill currentpicture withcolor "\MPvar{color}" ;  
  currentpicture := currentpicture yscaled \MPvar{height} ;  
\stopuniqueMPgraphic
```

```
\definesymbol
```

```
[comment-normal]
```

```
[\uniqueMPgraphic{cow}{height=4ex,color=darkred}]
```



```

\definesymbol
  [comment-down]
  [\uniqueMPgraphic{cow}{height=4ex,color=darkgreen}]

\startcomment[hello][symbol={comment-normal,comment-down}]
  oeps
\stopcomment

```



Again the way this shows up depends on the viewer capabilities so there might be a fallback on the normal comment symbol. You can influence the size of the image (icon):

```

\startcomment[hello]
  [symbol={comment-normal,comment-down},width=\marginwidth]
  oeps
\stopcomment

```



There are some options that you can use for finetuning the comments.

```

\setupcomment [...1,...2] [...2,...1]
               OPT
1  NAME
2  state      = start stop none
   method    = normal hidden
   symbol    = Comment Help Insert Key Newparagraph Note Paragraph
               Default
   width     = fit DIMENSION
   height    = fit DIMENSION
   depth     = fit DIMENSION
   title     = TEXT
   subtitle  = TEXT
   author    = TEXT
   nx        = NUMBER
   ny        = NUMBER
   color     = COLOR
   option    = xml max
   textlayer = NAME
   location  = leftedge rightedge inmargin leftmargin rightmargin text
               high none
   distance  = DIMENSION
   space     = yes
   buffer    = BUFFER

```

A new instance is defined with:

```
\definecomment [1...] [2...] [3...OPT...OPT]
1  NAME
2  NAME
3  inherits: \setupcomment
```

The default instance is predefined by

```
\definecomment[comment]
```

You can define your own instances:

```
\definecomment[mycomment]
```

The generated commands have a syntax like:

```
\startCOMMENT [1...] [2...OPT...OPT] ... \stopCOMMENT
1  TEXT
2  inherits: \setupcomment
instances: comment
```

and:

```
\COMMENT [1...] [2...OPT...OPT] {3...}
1  TEXT
2  inherits: \setupcomment
3  TEXT
instances: comment
```

Most fields explain themselves. With state you can disable this feature. Comments can be hidden in which there is no icon shown. The nx and ny fields determine the size of the popup.

In case you wonder where the yellow backgrounds come from, here is the trick:


```
\enabletrackers[comments.anchors]
```

7 Attachments

Attachments are (normally) embedded files that the reader can extract. A viewer can decide to just show the content or call an associated program (which one depends on the operating system) to deal with the file. As with other annotations they started out depicted by symbols but then browsers started showing them in lists next to the displayed page.

```
\attachment
[attachment 1]
[file=interaction-attached-001.txt,
 title=Just some text,
 width=2em,
 height=2em,
 author=Hans,
 subtitle=Plain text]
```

```
\attachment
[attachment 2]
[file=cow.mp,
 title=Just a graphic,
 author=Hans,
 subtitle=Some MetaPost,
 method=hidden]
```



These two attachments differ in one aspect: the second one is hidden, i.e. it has no icon in the text. However, the attachment is in the file and is (often) visible in the side bar. The symbol for the visible one is in the margin, which is achieved with:

```
\setupattachment
[location=inmargin]
```

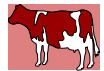
You can use your own icon, for instance:

```
\startuniqueMPgraphic{cow}{height,s:color}
loadfigure "cow.mp" number 1 ;
refill currentpicture withcolor "\MPvar{color}" ;
currentpicture := currentpicture ysize \MPvar{height} ;
\stopuniqueMPgraphic
```

```

\definesymbol
[attachment-normal]
[\uniqueMPgraphic{cow}{height=4ex,color=darkblue}]
\definesymbol
[attachment-down]
[\uniqueMPgraphic{cow}{height=4ex,color=darkyellow}]

```



This time we get a cow as icon and the cow is also embedded as image. When writing this manual a click in Sumatra just opens the pdf file, but when I embed an mp3 file, a save-as window pops up.

The previous examples directly injected the attachment using the `\attachment` commands with the appropriate arguments. You can add titles, define a name that will be used when the attachment is saved:

```

\attachment[sometag][extra specs]
\attachment[test.tex]
\attachment[file=test.tex]
\attachment[file=test.tex,method=hidden]
\attachment[name=newname,file=test.tex]
\attachment[title=mytitle,name=newname,file=test.tex]

```

but there's also a more indirect way, for instance here we define some attachments:

```

\defineattachment[whatever-1][file=test.tex]
\defineattachment[whatever-2][file=test.tex,method=hidden]

```

that later can be called up with:

```

\attachment[whatever-1][method=hidden]
\attachment[whatever-2]

```

where of course hidden is to be omitted when you want an icon. The commands that are used to define and tune an instance are:

```
\defineattachment [...]1 [...]2 [...,...3...]
                        OPT      OPT
```

- 1 NAME
- 2 NAME
- 3 inherits: \setupattachment

```
\setupattachment [...]1 [...,...2...]
                        OPT
```

- 1 NAME
- 2 title = TEXT
 subtitle = TEXT
 author = TEXT
 file = FILE
 name = NAME
 buffer = BUFFER
 state = start stop
 method = normal hidden
 symbol = Graph Paperclip Pushpin Default
 width = fit DIMENSION
 height = fit DIMENSION
 depth = fit DIMENSION
 color = COLOR
 textlayer = NAME
 location = leftedge rightedge inmargin leftmargin rightmargin text
 high none
 distance = DIMENSION

There is one predefined instance:

```
\defineattachment[attachment]
```

So we have:

```
\startATTACHMENT [...]1 [...,...2...] ... \stopATTACHMENT
                        OPT      OPT
```

- 1 NAME
 - 2 inherits: \setupattachment
- instances: attachment**

```

\ATTACHMENT [1...] [2.........]
              OPT      OPT
1  NAME
2  inherits: \setupattachment
instances: attachment

```

Yet another level of abstraction can be achieved with:

```

\registerattachment [1...] [2.........]
1  NAME
2  inherits: \setupattachment

```

For example:

```

\registerattachment
[sometag]
[name=fool.txt,
 file=foo.txt,
 title=Fool me,
 subtitle=Not you,
 author= Joker]

```

This is the MkIV replacement for the MkII method:

```

\useattachment[test.tex]
\useattachment[whatever][test.tex]
\useattachment[whatever][newname][test.tex]
\useattachment[whatever][title][newname][test.tex]

```

or with all options:

```

\useattachment[name][file][author][title][subtitle]

```

This variant stays around for old times sake and just maps onto `\registerattachment`, you can better use that one because it frees you from remembering which arguments is what for.

8 Bookmarks

9 JavaScript

Annotations can be controlled with JavaScript but it really depends on the viewer if it works out well. Using these scripts is a multi-step process where common functions and data structures can be shared and collected in preambles:

```
\startJSpreamble {name}  
  MyCounter = 0 ;  
\stopJSpreamble
```

The more action oriented scripts are defined as:

```
\startJScode {increment}  
  MyCounter = MyCounter + 1 ; // or: ++MyCounter ;  
\stopJScode
```

This script is executed with:

```
\goto {advance by one} [JS(increment)]
```

Nicer is to define a function:

```
\startJSpreamble {helpers} used now  
  function Increment(n) {  
    MyCounter = MyCounter + n ;  
  }  
\stopJSpreamble
```

and then say:

```
\goto {advance by one} [JS(Increment{5})]
```

The distribution contains a collection of scripts that can be preloaded and used when needed. You can recognize the files by the `java-imp-` prefix. To prevent all preambles ending up in the pdf file, we can say:

```
\startJSpreamble {something} used later  
\stopJSpreamble
```

We already saw that one can also say `used now` and there's also a way to filter specific preambles on usage:

```
\startJScode {mything} uses {something}
```

`\stopJScript`

One should be aware of the fact that there is no decent way to check if every script is all right! Even worse, the JavaScript interpreter currently used in the Acrobat tools is not reentrant, and breaks down on typos

The full repertoire of commands is:

`\startJScript` ¹... ²... ³... `\stopJScript`

1 NAME

2 uses

3 NAME

`\startJSpreable` ¹... ²... ³... `\stopJSpreable`

1 NAME

2 used

3 now later

`\addtoJSpreable` {¹...} {²...}

1 NAME

2 CONTENT

`\setJSpreable` {¹...} {²...}

1 NAME

2 CONTENT

10 Buttons

11 Menus

12 Progress

13 Widgets

14 Page transitions

I'm not sure if this feature is still used but in the early days of pdf support in T_EX, users loved it. I never really used it myself. Page transitions only make sense in presentations, and unfortunately the ones provided by the Acrobat viewers are just ugly. Anyhow, one automatically gets them by saying:

```
\setuppagetransitions[random]
```

This way one gets random transitions. Resetting transitions is done by:

```
\setuppagetransitions[reset]
```

If needed one can specify transitions but I strongly advice against this, because these commands are very viewer dependant, therefore: if in despair, use numbers! By default, the next set is used, and one can access them by number,

number		transition effects
1	2	{split,in,vertical} {split,in,horizontal}
3	4	{split,out,vertical} {split,out,horizontal}
5	6	{blinds,horizontal} {blinds,vertical}
7	8	{box,in} {box,out}
9	10 11 12	{wipe,east} {wipe,west} {wipe,north} {wipe,south}
	13	dissolve
	14 15	{glitter,east} {glitter,south}

The next settings are all valid:

```
\setuppagetransitions
\setuppagetransitions[1]
\setuppagetransitions[3,5,8,random]
```

Valid setups are:

```
\setuppagetransitions [...]  
* reset auto start random NUMBER
```


15 Importing

16 Tagging

