
WIDGETS
uncovered

Introduction

This document introduces the extended cross reference mechanism, fill–in fields, JAVASCRIPT support, and comment (text annotations in PDF terminology). We will also discuss page transitions. Although not all interactive features are covered — for instance, we don't discuss menus and roll over buttons— this manual is a good introduction to the way CONTEXT deals with interactivity.

This manual will be updated. A couple of things can be done in a more convenient way and there are some more features.

1 Chained references

About half a year ago (end 1997/begin 1998) we reimplemented part of the reference macros. The reference mechanism not only deals with the more traditional cross references, but also takes care of hyperefences, navigational means, launching applications, running JAVASCRIPT, etc. By integrating these features in one mechanism, we limit the number of commands needed for hyperreferences, menus and buttons. Like before, we have the normal cross references (here I only demonstrate `\goto`):

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

The inner reference is either a user defined one, or a system provided reference, like `previouspage` to go to the next page, `forward` to cycle, `nextcontents` for the next level table of contents in a linked list of such tables, etc. By the way, some new keywords are `backward` and `forward`, two cycling alternatives for `previous` and `next`, that jump from last page to the first one and vice versa.

The outer reference, being a file or URL, is defined at the document level and is accessed by the `::`. When possible one should use logical names and define such files and URL's at the outer document level using the appropriate definition commands.

A special class of references are the viewer control ones, like `CloseDocument` or `PreviousJump`. They can be recognized by their capitals.

So far, nothing is new but after half a year of experimenting, I decided to make some of these extensions permanent:

```
\goto[action{arguments}]
\goto[operation(arguments)]
\goto[operation(action{arguments})]
```

Actions are for instance the already mentioned viewer controls as well as specific commands dealing with viewer data, like `ResetForm`, which optionally takes a comma seperated list of fieldnames. Later I will show an example if this action.

Valid operations are `page`, `program`, `action`, or `JS`. The `page` operation replaces `\gotopage`, and accepts a pagenumber as well as relevant keywords. One can prefix a pagenumber by a file or URL tag. The `program` operation replaces `\gotoprogram` and the `action` operation is compatible with the viewer specific commands.

```
\goto {Colofon} [colofon]
```

Also new in the next release is the possibility to chain references. When one passes a comma separated list, the references are executed one after another, which means that we can say things like *goto the chapter on installation and start the movie showing how to calibrate*, or:

```
\goto {calibration} [installation,StartMovie{calibrate}]
```

It's no news that ConTeXt is able to handle multi-word references and permits them to be split across lines. One can imagine that when saying things like: forget and exit, which was entered as:

```
\goto {forget and exit} [JS(Forget_Changes),CloseDocument]
```

one actually ends up with four times two references. Depending on the driver used, ConTeXt tries to limit the resources, using shared objects.

An operation has an argument which itself can have one or more arguments. These are passed as comma separated list between `{}`, like in:

```
\goto {do something nice} [JS(something{S{alpha},V{2},V{true}})]
```

Validation of these arguments is upto the specific action handler.

2 JavaScript

Because JavaScript support is still sort of experimental, I'll only give some simple examples. Using 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 five} [JS(increment)]
```

It is possible to pass arguments to the scripts. Consider for instance:

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

combined with:

```
\startJScode {increment}
  MyCounter = MyCounter + JS_V_1 ;
\stopJScode
```

Here the `V{...}` means verbose. By default arguments are passed as strings. Other prefixes are `R{...}` for references or the optional `S{...}` for strings, all shown in:

```
\goto
 {calculate total}
 [JS(Sum{V{1.5},V{2.3},S{Problems!},R{overflow}})]
```

These arguments end up in the script as:

```
JS_V_1 = 1.5 ;
JS_V_2 = 2.3 ;
JS_S_3 = "problems!" ;
JS_R_4 = "overflow" ;
JS_P_4 = 3 ;
```

As one can see, the reference is translated in a named destination as well as pagenumber. We also have a counter that tells JAVASCRIPT how many arguments were passed: `JS_N`. Some day symbolic arguments will be handled too.

Currently we're writing a collection of scripts that can be preloaded and used when needed. To prevent all preambles ending up in the PDF file, we can say:

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

(one can also say `used now`) and:

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

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. Due to rather unsafe line breaking, DISTILLER output is more error prone than PDFTEX's output. Technically at this moment (mid 1998) only PDFTEX supports proper embedding of document scripts (the preambles), while for DISTILLER we have to use a workaround. But most users will probably never notice.

The verbatim pretty printing mechanism supports JAVASCRIPT, which means that keywords as well as special tokens are recognized. One can use the prefix `TEX` to mark words to be typeset using the TEX filter. This prefix is of course not passed to the PDF file.

3 Fill-in fields

Fields come in many disguises. Currently CONTEXT supports the field types provided by PDF, which in turn are derived from HTML. Being a static format and not a programming language, PDF only provides the interface. Entering data is up to the viewer and validation to the built in JAVASCRIPT interpreter. The next paragraph shows an application.

A few years back, TEX could only produce [ ] output, but nowadays, thanks to [ ], we can also directly produce [ ]! Nice eh? Actually, while the first field module was prototyped in ACROBAT, the current implementation was debugged in [ ] Field support in [ ] is rather advanced and complete and all kind of fields are supported. One can hook in appearances, and validation [ ]'s. Fields can be cloned and copied, where the latter saves some space. By using [ ] when suited, this module saves space anyway.

This paragraph is entered in the source file as:

```
A few years back, \TEX\ could only produce \fillinfield [dvi]
{\DVI} output, but nowadays, thanks to \fillinfield {Han The
Thanh}, we can also directly produce \fillinfield [pdf] {\PDF}!
Nice eh? Actually, while the first field module was prototyped
in \ACROBAT, the current implementation was debugged in
\fillinfield [pdfTeX] {\PDFTEX}. Field support in \fillinfield
[ConTeXt] {\CONTEXT} is rather advanced and complete and all
kind of fields are supported. One can hook in appearances, and
validation \fillinfield [JavaScripts] {\JAVASCRIPT}'s. Fields
can be cloned and copied, where the latter saves some space. By
using \fillinfield {objects} when suited, this module saves
space anyway.
```

I leave it to the imagination of the user how `\fillinfield` is implemented, but trust me, the definition is rather simple and is based on the macros mentioned below.

Because I envision documents with many thousands of fields, think for instance of tutorials, I rather early decided to split the definition from the setup. Due to the fact that while typesetting a field upto three independant instances of `\framed` are called, we would end up with about 150 hash entries per field, while in the current implementation we only need a few. Each field can inherit its specific settings from the setup group it belongs to.

Let's start with an example of a *radio* field. In fact this is a collection of fields. Such a field is defined with:

```
\definefield
  [Logos] [radio] [LogoSetup]
  [ConTeXt,PPCHTEX,TeXUtil] [PPCHTEX]
```

Here the fourth argument specifies the subfields and the last argument tells which one to use as default. We have to define the subfields separately:

```
\definesubfield [ConTeXt] [] [ConTeXtLogo]
\definesubfield [PPCHTEX] [] [PPCHTEXLogo]
\definesubfield [TeXUtil] [] [TeXUtilLogo]
```

The second argument specifies the setup. In this example the setup (`LogoSetup`) is inherited from the main field. The third arguments tells ConTeXt how the fields look like when turned on. These appearances are to be defined as symbols:

```
\definesymbol [ConTeXtLogo] [{\externalfigure[mp-cont.502]}]
\definesymbol [PPCHTEXLogo] [{\externalfigure[mp-cont.503]}]
\definesymbol [TeXUtilLogo] [{\externalfigure[mp-cont.504]}]
```

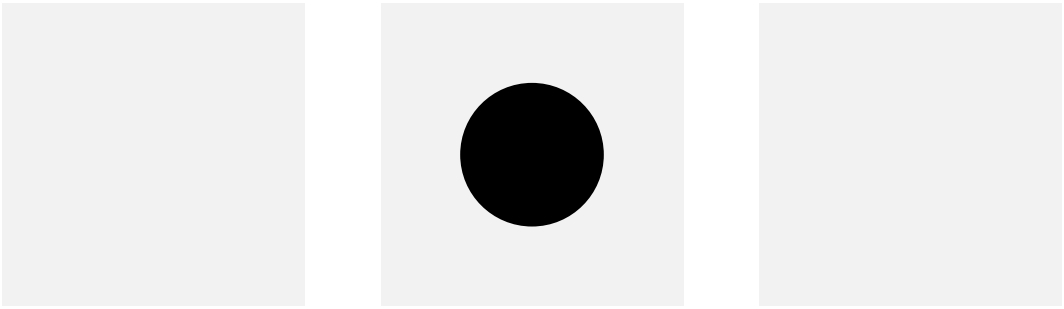
Before we typeset the fields, we specify some settings to use:

```
\setupfield [LogoSetup]
  [width=4cm,
   height=4cm,
   frame=off,
   background=screen]
```

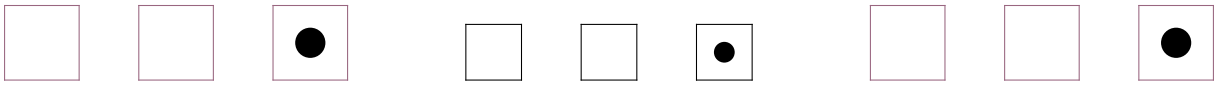
Finally we can typeset the fields:

```
\hbox to \hsize
  {\hss\field[ConTeXt]\hss\field[PPCHTEX]\hss\field[TeXUtil]\hss}
```

This shows up as:



An important characteristic of field is cloning cq. copying, as demonstrated below:



The next table shows the relations between these fields of type radio:

NAME	TYPE	ROOT	PARENT	KIDS	GROUP	MODE	VALUES	DEFAULT
example-1	radio			ex-a,ex-b,ex-c	setup 1	*loner*		ex-c
ex-a	radio	example-1		ex-p,ex-x	setup 1	*parent*	yes-a,nop-a	
ex-b	radio	example-1		ex-q,ex-y	setup 1	*parent*	yes-a,nop-a	
ex-c	radio	example-1		ex-r,ex-z	setup 1	*parent*	yes-a,nop-a	ex-c
ex-p	radio		ex-a		setup 2	*clone*	yes-b,nop-b	
ex-q	radio		ex-b		setup 2	*clone*	yes-b,nop-b	
ex-r	radio		ex-c		setup 2	*clone*	yes-b,nop-b	ex-c
ex-x	radio		ex-a		setup 1	*copy*	yes-a,nop-a	
ex-y	radio		ex-b		setup 1	*copy*	yes-a,nop-a	
ex-z	radio		ex-c		setup 1	*copy*	yes-a,nop-a	ex-c

This table is generated by \showfields and can be used to check the relations between fields, but only when we have set \tracefieldstrue. Radio fields have the most complicated relationships of fields, due to the fact that only one of them can be activated (on). By saying \logfields one can write the current field descriptions to the file fields.log.

Here we used some TEX mathematical symbols. These are functional but sort of dull, so later we will define a more suitable visualization.

```
\definesymbol [yes-a] [$\times$]
\definesymbol [yes-b] [$\star$]
\definesymbol [nop-a] [$\bullet$]
\definesymbol [nop-b] [$-$]
```

The parent fields were defined by:

```
\definefield [example-1] [radio] [setup 1] [ex-a,ex-b,ex-c] [ex-c]
\definesubfield [ex-a,ex-b,ex-c] [setup 1] [yes-a,nop-a]
```

and the clones, which can have their own appearance, by:

```
\clonefield [ex-a] [ex-p] [setup 2] [yes-b,nop-b]
\clonefield [ex-b] [ex-q] [setup 2] [yes-b,nop-b]
\clonefield [ex-c] [ex-r] [setup 2] [yes-b,nop-b]
```

The copies are defined using:

```
\copyfield [ex-a] [ex-x]
\copyfield [ex-b] [ex-y]
\copyfield [ex-c] [ex-z]
```

using the setups

```
\setupfield [setup 1] [width=1cm,height=1cm,framecolor=red]
\setupfield [setup 2] [width=.75cm,height=.75cm]
```

Finally all these fields are called using \field:

```
\hbox to \hsize
  {\field[ex-a]\hfil\field[ex-b]\hfil\field[ex-c]\hfil\hfil
   \field[ex-p]\hfil\field[ex-q]\hfil\field[ex-r]\hfil\hfil
   \field[ex-x]\hfil\field[ex-y]\hfil\field[ex-z]}
```

Now we will define a so called *check* field. This field looks like a radio field but is independant of others. First we define some suitable symbols:

```
\definesymbol [yes] [{\externalfigure[mp-cont.502]}]
\definesymbol [no]  []
```

A check field is defined as:

```
\definefield [check-me] [check] [setup 3] [yes,no] [no]
```

This time we say `\field[check-me]` and get:

As setup we used:

```
\setupfield
  [setup 3]
  [width=2cm, height=2cm,
   rulethickness=3pt, corner=round, framecolor=red]
```

We already saw an example of a *line* field. By default such a line field looks like:

your email	

We defined this field as:

```
\definefield [Email] [line] [ShortLine] [] [pragma@wxs.nl]
```

and called it using a second, optional, argument:

```
\field [Email] [your email]
```

As shown, we can influence the way such a field is typeset. It makes for instance sense to use a monospaced typeface and limit the height. When we set up a field, apart from the setup class we pass some general characteristics, and three more detailed definitions, concerning the surrounding, the label and the field itself.

```
\setupfield
  [ShortLine]
  [label,frame,horizontal]
  [offset=4pt,height=fit,framecolor=green,
   background=screen,backgroundscreen=.80]
  [height=18pt,width=80pt,align=middle,
   background=screen,backgroundscreen=.90,frame=off]
  [height=18pt,width=80pt,color=red,align=right,style=type,
   background=screen,backgroundscreen=.90,frame=off]
```

So now we get:

your email

Such rather long definitions can be more sparse when we set up all fields at once, like:

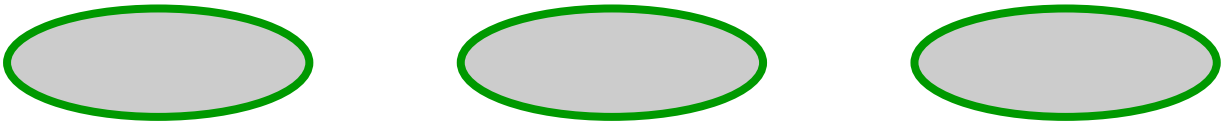
```
\setupfields
  [label,frame,horizontal]
  [offset=4pt,height=fit,framecolor=green,
   background=screen,backgroundscreen=.80]
  [height=18pt,width=80pt,
   background=screen,backgroundscreen=.90,frame=off]
  [height=18pt,width=80pt,color=red,align=middle,
   background=screen,backgroundscreen=.90,frame=off]
```

So given that we have defined field `MainMail` we can say:

```
\setupfield [LeftLine]
  [background=normalbutton, backgroundcolor=darkgreen,
   offset=2ex, height=7ex, width=.25\hsize,
   style=type, frame=off, align=left]
\setupfield [MiddleLine]
  [background=normalbutton, backgroundcolor=darkgreen,
   offset=2ex, height=7ex, width=.25\hsize,
   style=type, frame=off, align=middle]
\setupfield [RightLine]
  [background=normalbutton, backgroundcolor=darkgreen,
   offset=2ex, height=7ex, width=.25\hsize,
   style=type, frame=off, align=right]
```

```
\clonefield [MainMail] [LeftMail]   [LeftLine]
\clonefield [MainMail] [MiddleMail] [MiddleLine]
\clonefield [MainMail] [RightMail]  [RightLine]
```

We get get three connected fields:



(Keep in mind that in ConTeXt left aligned comes down to using `\raggedleft`, which can be confusing, but history cannot be replayed.)

By the way, this shape was generated by METAPOST using the overlay mechanism:

```
\startuniqueMPgraphic{button}
  path p ; p := fullcircle xyscaled (OverlayWidth,OverlayHeight) ;
  fill p withcolor (.8,.8,.8) ;
  draw p withcolor OverlayColor withpen pencircle scaled 3 ;
\stopuniqueMPgraphic

\defineoverlay [normalbutton] [\uniqueMPgraphic{button}]
```

Due to the fact that a field can have several modes (loner, parent, clone or copy), one cannot define a clone or copy when the parent field is already typeset. When one knows in advance that there will be clones or copies, one should use:

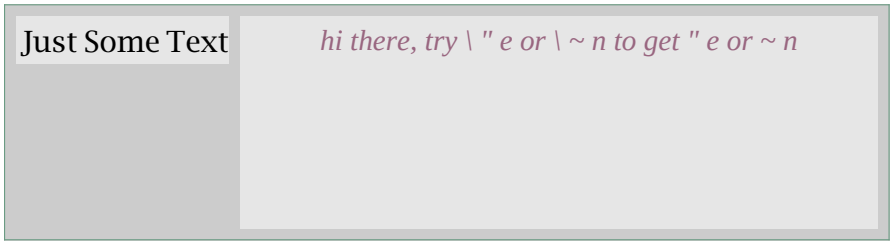
```
\definemainfield [MainMail] [line] [ShortLine] [] [pragma@wxs.nl]
```

Now we can define copies, clones and even fields with the same name, also when the original already is typeset. Use `\showfields` to check the status of fields. When in this table the mode is typeset slanted, the field is not yet typeset.

The values set up with `\setupfield` are inherited by all following setup commands. One can reset these default values by:

```
\setupfields[reset]
```

When we want more than one line, we use a *text* field. Like the previous fields, text must be entered in the viewer specific encoding, in our case, PDF document encoding. To free users from thinking of encoding, ConTeXt provides a way to force at least the accented glyphs into a text field in a for TeX users familiar way:



Now, how is this done? Defining the field is not that hard:

```
\definefield [SomeField] [text] [TextSetup] [default text]
```

The conversion is taken care of by a JAVASCRIPT's. We can assign such scripts to mouse and keyboard events, like in:

```
\setupfield
  [TextSetup][...][...][...]
  [....,
   regionin=JS(Initialize_TeX_Key),
   afterkey=JS(Convert_TeX_Key),
   validate=JS(Convert_TeX_String)]
```

The main reason for using the `JS(...)` method here is that this permits future extensions and looks familiar at the same time. Depending on the assignments, one can convert after each keypress and/or after all text is entered.

We've arrived at another class of fields: *choice*, *pop-up* and *combo* fields. All those are menu based (and ugly). This in-line menu was defined as:

```
\definefield
  [Ugly] [choice] [UglySetup]
  [ugly,awful,bad] [ugly]
\setupfield
  [UglySetup]
  [width=6em,
   height=1.2\lineheight,
   location=low]
```

Pop-up fields look like: ugly and combo fields permit the user to enter his or her own option: ugly . The amount of typographic control over these three type of fields is minimal, but one can specify what string to show and what string results:

```
\definefield
  [Ugly2] [popup] [UglySetup]
  [ugly,awful,bad] [ugly]
\definefield
  [Ugly3] [combo] [UglySetup]
  [ugly,{AWFUL=>awful},bad] [ugly]
```

Here AWFUL is shown and when selected becomes the choice awful. Just in case one wonders why we use =>, well, it just looks better and the direction shows what value will be output.

A special case of the check type field is a pure *push* field. Such a field has no export value and has only use as a pure interactive element. For the moment, let's forget about that one.

Before we demonstrate our last type of fields and show some more tricky things, we need to discuss what to do with the information provided by filling in the fields. There are several actions available related to fields.

One can for instance reset the form or part of the form. This last sentence was typed in as:

```
One can for instance \goto {reset the form} [ResetForm] or
\goto {part of the form} [ResetForm{AllUglies}]. This last
sentence was typed in as:
```

Hereby AllUglies is a set of fields to be defined on forehand, using

```
\definefieldset [AllUglies] [Ugly, Ugly2, Ugly3]
```

In a similar way one can submit some or all fields using the SubmitForm directive. This action optionally can take two arguments, the first being the destination, the second a list of fields to submit, for instance:

```
\button{submit}[SubmitForm{mailto::pragma@wxs.nl,AllUglies}]
```

Once the fields are submitted (or saved in a file), we can convert the resulting FDF file into something TEX with the perl program fdf2tex. One can use \ShowFDFFields{filename} to typeset the values. If you do not want to run the PERL converter from within TEX, say \runFDFconverterfalse. In that case, the (stil) less robust TEX based converter will be used.

I already demonstrated how to attach scripts to events, but how about changing the appearance of the button itself? Consider the next definitions:

```
\definesymbol [my-y] [$\times$]
\definesymbol [my-r] [?]
\definesymbol [my-d] [!]

\definefield
  [my-check] [check] [my-setup]
  [{my-y,my-r,my-d},{,my-r,my-d}]
```

Here we omitted the default value, which always is *no* by default. The setup can look like this:

```
\setupfield
  [my-setup]
  [width=1.5cm, height=1.5cm,
   frame=on, framecolor=red, rulethickness=1pt,
   backgroundoffset=2pt, background=screen, backgroundscreen=.85]
```

Now when this field shows up, watch what happens when the mouse enters the region and what when we click.

So, when instead of something `[yes,no]` we give triplets, the second element of such a triplet declares the roll-over appearance and the third one the push-down appearance. The braces are needed!

One application of appearances is to provide help or additional information. Consider the next definition:

```
\definefield [Help] [check] [HelpSetup] [helpinfo] [helpinfo]
```

This means as much as: define a check field, typeset this field using the help specific setup and let `helpinfo` be the on-value as well as the default. Here we use the next setup:

```
\setupfields
  [reset]
\setupfield
  [HelpSetup]
  [width=fit,height=fit,frame=off,option={readonly,hidden}]
```

We didn't use options before, but here we have to make sure that users don't change the content of the field and by default we don't want to show this field at all. The actual text is defined as a symbol:

```
\definesymbol [helpinfo] [\SomeHelpText]

\def\SomeHelpText%
  {\framed
     [width=\leftmarginwidth,height=fit,align=middle,style=small,
      frame=on,background=color,backgroundcolor=white,framecolor=red]
     {Click on the hide button to remove this screen}}
```

Now we can put the button somewhere and turn the help on or off by saying Hide Help or Show Help. Although it's better to put these commands in a dedicated part of the screen. And try Help.

We can place a field anywhere on the page, for instance by using the `\setup...texts` commands. Here we simply said:

```
\inmargin {\fitfield[Help]} Now we can put the button somewhere and
turn the help on or off by saying \goto {Hide Help} [HideField{Help}]
or \goto {Show Help} [ShowField{Help}]. Although it's better to put
these commands in a dedicated part of the screen. And try \goto
{Help} [JS(Toggle_Hide{Help})].
```

When one uses for instance `\setup...texts`, one often wants the help text to show up on every next page. This can be accomplished by saying:

```
\definemainfield [Help] [check] [HelpSetup] [helpinfo] [helpinfo]
```

Every time such a field is called again, a new copy is generated automatically. Because fields use the objectreference mechanism and because such copies need to be known to their parent, field inclusion is a multi–pass typesetting job (upto 4 passes can be needed!).

When possible, appearances are shared between fields, mainly because this saves space, but at the cost of extra object references. This feature is not that important for straight forward forms, but has some advantages when composing more complicated (educational) documents.

Let us now summarize the commands we have available for defining and typesetting fields. The main definition macro is:

```
\definefield[.1.][.2.][.3.][..,.4.,..][.5.]

.1.     name
.2.     name
.3.     name
.4.     name
.5.     name
```

and for radiofields we need to define the components by:

```
\definesubfield[.1.][.2.][..,.3.,..]

.1.     name
.2.     name
.3.     name
```

Fields can be cloned and copied, where the latter can not be set up independently.

```
\clonefield[.1.][..,.2.,..][.3.][..,.4.,..]

.1.     name
.2.     name
.3.     name
.4.     name
```

```
\copyfield[.1.][..,.2.,..]

.1.     name
.2.     name
```

Fields can be grouped, and such a group can have its own settings. Apart from copied fields, we can define the layout of a field and set options using:

```
\setupfield[.1.][..,.2.,..][..,..=..,..][..,..=..,..][..,..=..,..]

.1.     name
.2.     label horizontal vertical frame
..=..   see \setupfields
```

Such a group inherits its settings from the general setup command:

```
\setupfields[..,.1.,..][.2.][..,..=..,..][..,..=..,..][..,..=..,..]
```

.1.	*name*
.2.	reset label horizontal vertical frame
n	*number*
distance	*dimension*
before	*command*
after	*command*
inbetween	*command*
color	*name*
style	normal bold slanted boldslanted type
align	left middle right
option	readonly required protected sorted unavailable hidden printable
clickin	*reference*
clickout	*reference*
regionin	*reference*
regionout	*reference*
afterkey	*reference*
format	*reference*
validate	*reference*
calculate	*reference*
fieldoffset	*dimension*
fieldframecolor	*name*
fieldbackgroundcolor	*name*
..=..	see \framed

Fields are placed using one of:

```
\field[...]
```

... *name*

or

```
\fitfield[...]
```

... *name*

Some pages back I showed an example of:

```
\fillinfield[.1.]{.2.}

.1.    text
.2.    text
```

Finally there are two commands to trace fields. These commands only make sense when one already has said: `\tracefieldstrue`.

```
\showfields[..,...,..]

...    name
```

```
\logfields
```

4 Tooltips

Chinese people seem to have no problems in recognizing their many different pictorial glyphs. Western people however seem to have problems in understanding what all those icons on their computer screens represent. But, instead of standardizing on a set of icons, computer programmers tend to fill the screen with so called tooltips. Well, ConTeXt can do tooltips too, and although a good design can do without them, TeX at least can typeset them correctly.

The previous paragraph has three of such tooltips under *western*, *icons* and ConTeXt, each aligned differently. We just typed:

```
Chinese people seem to have no problems in recognizing their many
different pictorial glyphs. \tooltip [left] {Western} {European
and American} people however seem to have problems in understanding
what all those \tooltip [middle] {icons} {small graphics} on their
computer screens represent. But, instead of standardizing on a set
of icons, computer programmers tend to fill the screen with so
called tooltips. Well, \tooltip {\CONTEXT} {a \TEX\ macro package}
can do tooltips too, and although a good design can do without them,
\TEX\ at least can typeset them correctly.
```

This is an official command, and thereby we can show its definition:

\tooltip*[.1.]*{.2.}{.3.}

.1. left right middle
.2. *text*

5 Fieldstacks

In due time I will provide more dedicated field commands. Currently apart from \fillinfield and \tooltip we have \fieldstack. Let's spend a few words on those now.

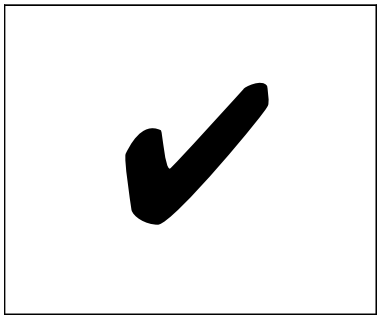


Figure 1 Do you want to see what interfaces are available? Just click here a few times!

One can abuse field for educational purposes. Take for instance figure 1. In this figure we can sort of walk over different alternatives of the same graphic. This illustration was typeset by saying:

```
\placefigure
  [left][fig:somemap]
  {Do you want to see what interfaces
   are available? Just click \goto
   {here} [JS(Walk_Field{somemap})]
   a few times!}
  {\fieldstack[somemap]}
```

However, before we can ask for such a map, we need to define a field set, which in fact is a list of symbols to show. This list is defined using:

```
\definefieldstack
  [somemap]
  [map -- -- --, map nl -- --, map nl de --, map nl de en]
  [frame=on]
```

which in turn is preceded by:

```
\useexternalfigure [map -- -- --] [euro-10] [width=.3\hsize]
\useexternalfigure [map nl -- --] [euro-11] [map -- -- --]
\useexternalfigure [map nl de --] [euro-12] [map -- -- --]
\useexternalfigure [map nl de en] [euro-13] [map -- -- --]

\definesymbol [map -- -- --] [{\externalfigure[map -- -- --]}]
\definesymbol [map nl -- --] [{\externalfigure[map nl -- --]}]
```

```
\definesymbol [map nl de --] [{\externalfigure[map nl de --]}]
\definesymbol [map nl de en] [{\externalfigure[map nl de en]}]
```

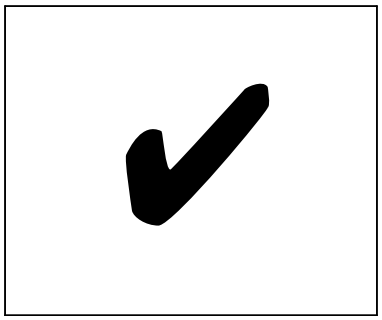


Figure 2 Choose one country, two countries, three countries or no countries at all.

A slightly different illustration is shown in figure 2. Here we use the same symbols but instead say:

```
\placefigure
  [here][fig:anothermap]
  {Choose \goto {one} [JS(Set_Field{anothermap,2})] country,
   \naar {two} [JS(Set_Field{anothermap,3})] countries,
   \naar {three} [JS(Set_Field{anothermap,4})] countries or
   \naar {no} [JS(Set_Field{anothermap,1})] countries at all.}
  {\fieldstack
     [anothermap]
     [map -- -- --, map nl -- --, map nl de --, map nl de en]
     [frame=on]}
```

As one can see, we can can skip the definition and pass it directly, but I wouldn't call that beautiful.

The formal definitions are:

```
\definefieldstack[.1.][..,.2.,..][..,..=..,..]

.1.     name
.2.     name
..=..   see \setupfields
```

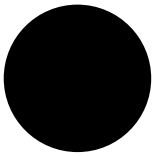
```
\fieldstack[.1.][..,.2.,..][..,..=..,..]

.1.     name
.2.     name
..=..   see \setupfields
```

Instead if stacking fields, you can of course also put them alongside. This makes sense when you want to use dedicated (visible) captions for each image.

```
\useexternalfigure [europe]  [euro-10] [width=.3\hsize]
\useexternalfigure [holland] [euro-nl] [europe]
\useexternalfigure [germany] [euro-de] [europe]
\useexternalfigure [england] [euro-en] [europe]

\definesymbol [europe]  [{\externalfigure[europe]}]
\definesymbol [holland] [{\externalfigure[holland]}]
\definesymbol [germany] [{\externalfigure[germany]}]
\definesymbol [england] [{\externalfigure[england]}]

\definefield
  [interface] [radio] [map]
  [england,germany,holland] [holland]

\definesubfield [holland] [] [holland,europe]
\definesubfield [germany] [] [germany,europe]
\definesubfield [england] [] [england,europe]

\setupfield[map][frame=off]
```

We can for instance typeset the fields by saying:

```
\startcombination[3]
  {\fitfield[holland]} {Dutch Interface}
  {\fitfield[germany]} {German Interface}
  {\fitfield[england]} {English Interface}
\stopcombination
```



Dutch Interface German Interface English Interface

6 Comments

The ACROBAT viewers support so called text annotations. These are small notes that can be popped up. In CONTEXT we will name them comment, because often that's what they represent. Comment uses a restricted encoding, but fortunately we can map most common accented characters onto it. A comment looks like:

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

Because comments are automatically placed in the margin, one can wonder what happens when we have more of them, like:

```
\startcomment[french]
  In France they use
  \leftguillemot these glyphs\rightguillemot
  in subsentences.
\stopcomment

\startcomment[accents][color=green,width=4cm,height=3cm]
  We love \'a\cc\cc\`e\~nt\SS
\stopcomment

\startcomment[lines][color=green,width=4cm,height=3cm]
  How about an

  empty line?
\stopcomment
```

Well, 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. Special TEX token sequences are converted and empty lines are honored or can be forced by \\. Just in case one wants to include a note inline, we offer \comment:

```
\inmargin {\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:

```
\startcomment[...][..,..=..,..] ... \stopcomment

...     name
..=..   see \setupcomment
```

```
\comment[.1.][..,..=..,..]{.2.}

.1.     name
..=..   see \setupcomment
```

Both can be set up using:

```
\setupcomment[..,..=..,..]

width     dimension
height    dimension
color     name
title     text
space     yes no
symbol    normal New Balloon Addition Help Paragraph Key
option    max
```

7 Page transitions

Some time ago Tobias asked me if ConTEXt could support page transitions, and the fact they could be implemented rather easy made me write these macros. Page tran-

sitions 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]
```

That way one gets random transitions. (We use Donald Arseneau's generic random number generator.) Resetting transitions is done by:

```
\setuppagetransitions[reset]
```

If needed one can specify transitions, but only in english. However, 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,

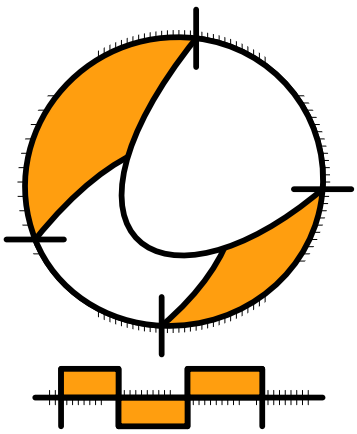
number	transition effects
1 2	split,in,vertical split,in,horizontal
3 4	split,out,vertical split,out,horizontal
5 6	blinds,horizontal blinds,vertical
7 8	box,in box,out
9 10 11 12	wipe,east wipe,west wipe,north wipe,south
13	dissolve
14 15	glitter,east glitter,south

The next settings are all valid:

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

To summarize this command we show its formal definition:

```
\setuppagetransitions[..,...,..]

...     reset number
```



PRAGMA

Advanced Document Engineering | Ridderstraat 27 | 8061GH Hasselt NL
tel: +31 (0)38 477 53 69 | email: pragma@wxs.nl | internet: www.pragma-ade.com