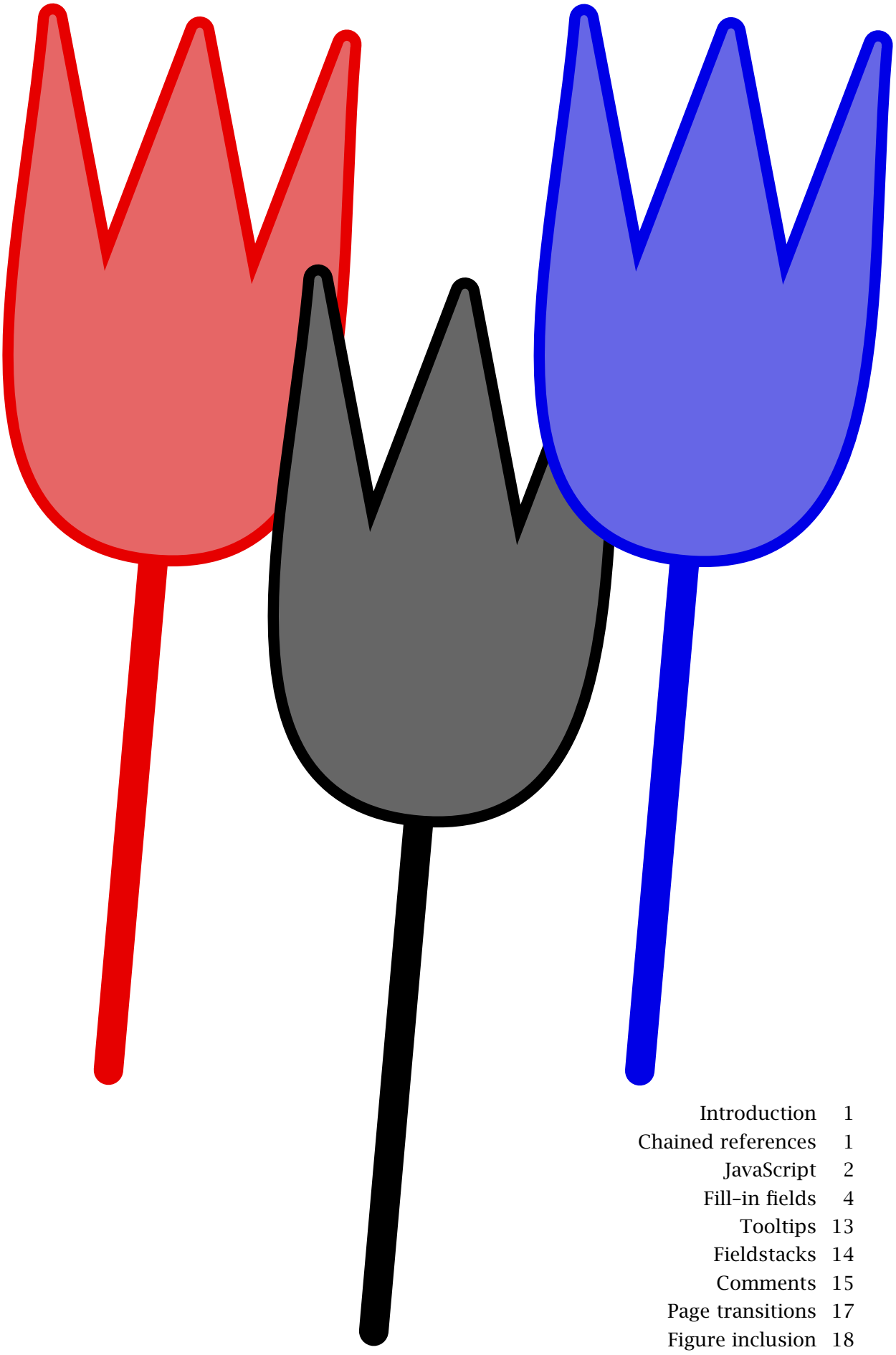
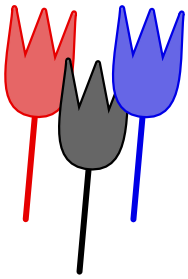
ConTeXt up-to-date 1998/1

Fields
Widgets
References



Pragma ADE
Ridderstraat 27
8061GH Hasselt NL



Introduction

This is the first issue of ConTeXt up–to–date. I will use this series of documents to introduce new features, provide suggestions, mention tricks, and write down whatever else comes up.

In this first, rather large issue, I will introduce the extended cross reference mechanism, fill–in fields and JavaScript support. Although the latter two are still sort of β, the more final release will offer at least the functionality presented here. I will also show some applications of fields combined with JavaScript.

New is the support of comment (text annotations in PDF terminology). Although already present some time, I will also discuss page transitions. Due to the fact that pdfTeX now supports PDF inclusion, I will introduce some new features of the figure inclusion mechanism, especially automatic type recognition.

Chained references

About half a year ago (end 1997/begin 1998) I 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]
```

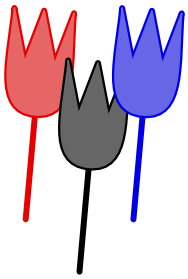
The inner reference is either a user defined one, or a system provided reference, like `previouspage` to go to the next page, `forward` to cycle, `nextcontents` for the next level table of contents in a linked list of such tables, etc. By the way, some new keywords are `backward` and `forward`, two cycling 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 everything and exit, which was entered as:

```
\goto {forget everything 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.

JavaScript

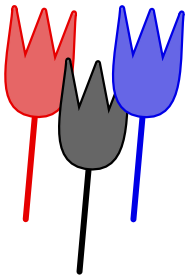
Because JavaScript support is still sort of experimental, I'll only give some simple examples. Using scripts is a multi-step process where common functions and data structures can be shared and collected in preambles:

```
\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 I'm 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:

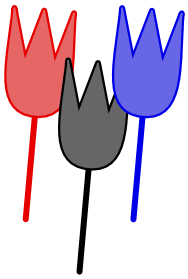
```
\startJSpreamble {something} used later
\stopJSpreamble
```

(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.



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 DVI output, but nowadays, thanks to Han The Thanh, we can also directly produce PDF! Nice eh? Actually, while the first field module was prototyped in Acrobat, the current implementation was debugged in pdfTeX. Field support in ConTeXt is rather advanced and complete and all kind of fields are supported. One can hook in appearances, and validation JavaScript's. Fields can be cloned and copied, where the latter saves some space. By using objects 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.

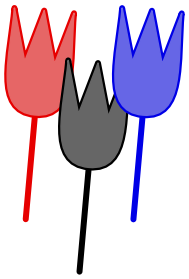
Let's start with an example of a *radio* field. In fact this is a collection of fields. Such a field is defined with:

```
\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:

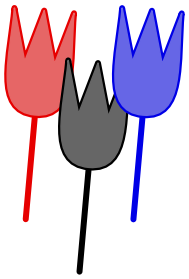
An important characteristic of field is cloning cq. copying, as demonstrated below:

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	nop-a
ex-b	radio	example-1		ex-q,ex-y	setup 1	*parent*	yes-a,nop-a	nop-a
ex-c	radio	example-1		ex-r,ex-z	setup 1	*parent*	yes-a,nop-a	yes-a
ex-p	radio		ex-a		setup 2	*clone*	yes-b,nop-b	nop-a
ex-q	radio		ex-b		setup 2	*clone*	yes-b,nop-b	nop-a
ex-r	radio		ex-c		setup 2	*clone*	yes-b,nop-b	yes-a
ex-x	radio		ex-a		setup 1	*copy*	yes-a,nop-a	nop-a
ex-y	radio		ex-b		setup 1	*copy*	yes-a,nop-a	nop-a
ex-z	radio		ex-c		setup 1	*copy*	yes-a,nop-a	yes-a

This table is generated by `\showfields` and can be used to check the relations between fields, but only when we have set `\tracefieldtrue`. 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]
```

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:

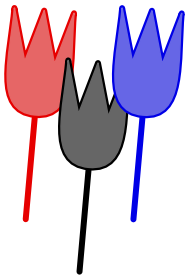
```
\definefield [check-me] [check] [setup 3] [yes,no] [no]
```

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

As setup we used:

```
\setupfield
  [setup 3]
  [width=2cm,height=2cm,linethickness=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:

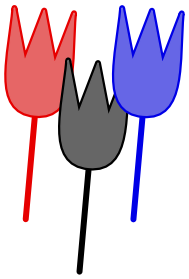
your email

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:

```
\def\MPnormalbutton#1#2#3%
  {\startreusableMPgraphic{nb:#1:#2:#3}
      input mp-tool;
      pickup pencircle scaled 3;
      fill fullcircle xscaled #1 yscaled #2 withcolor (.8,.8,.8);
      draw fullcircle xscaled #1 yscaled #2 withcolor \MPcolor{#3};
   \stopreusableMPgraphic
   \reuseMPgraphic{nb:#1:#2:#3}}

\defineoverlay
  [normalbutton]
  [\MPnormalbutton\overlaywidth\overlayheight\overlaycolor]
```

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]
```

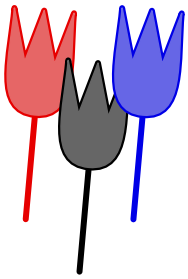
Now we can define copies, clones and even fields with the same name, also when the original already is typeset. Use `\showfields` to check the status of fields. When in this table the mode is typeset slanted, the field is not yet typeset.

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:

Just Some Text



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). 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: and combo fields permit the user to enter his or her own option: . 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.

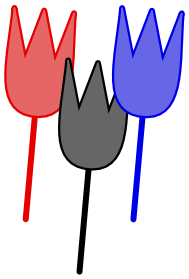
One can for instance reset the form or part of the form. This last sentence was typed in as:

```
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.

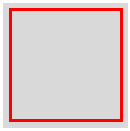
I already demonstrated how to attach scripts to events, but how about changing the appearance of the button itself? Consider the next definitions:

```
\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,backgroundoffset=2pt,rulethickness=1pt,
   frame=on,framecolor=red,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:

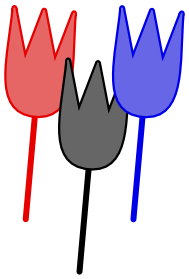
```
\definefield [Help] [check] [HelpSetup] [helpinfo] [helpinfo]
```

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.

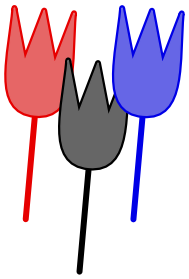
Let us now summarize the commands we have available for defining and typesetting fields. The main definition macro is:

```
\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
```

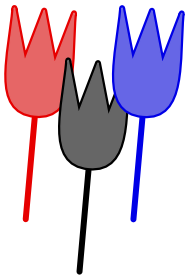
Fields can be grouped, and such a group can have its own settings. Apart from copied fields, we can define the layout of a field and set options using:

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

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

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

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

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

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
```

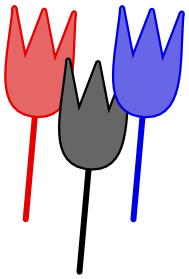
```
\logfields
```

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.}

.1.      left right middle
.2.      text
```

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.

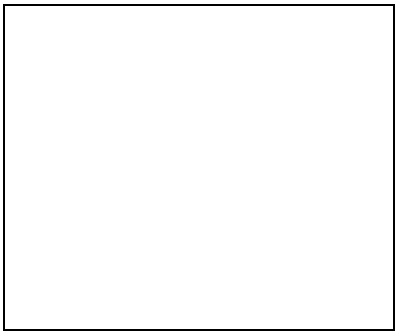


Figure 1 Wanna see what interfaces are available? Just click here a few times!

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]
  {Wanna see what interfaces are available? Just
   click \goto {here} [JS(Walk_Field{somemap})]
   a few times!}
  {\fieldstack[somemap]}
```

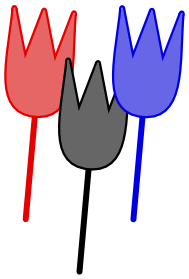
However, before we can ask for such a map, we need to define a field set, which in fact is a list of symbols to show. This list is defined using:

```
\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]}]
```

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



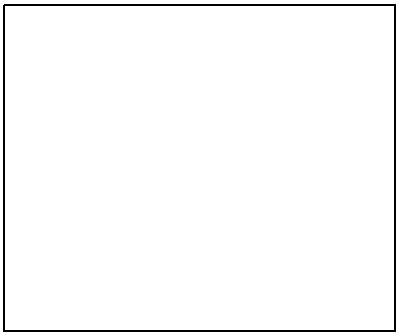


Figure 2 Choose one country, two countries, three countries or no countries at all.

```
\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.

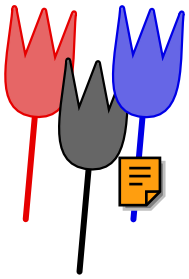
The formal definitions are:

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

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

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

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

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:

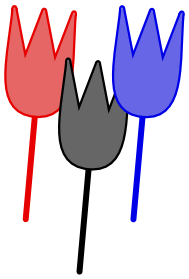
```
\inmargin {\comment{How I hate those notes spoiling the layout.}} Maybe some
day I can convince myself to add some features \comment {Think of comment
classes that can be turned on and off and get their own colors.} related to
version control.
```

Maybe some day I can convince myself to add some feat  s 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
```

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 transitions only make sense in presentations, and unfortunately the ones provided by the Acrobat viewers are just ugly. Anyhow, one automatically get 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
```

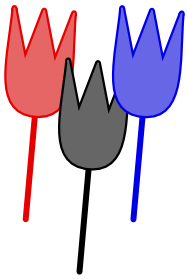


Figure inclusion

When PDFTEX started to support PDF inclusion, and thereby became a full substitute for DISTILLER, I decided to extend the figure inclusion mechanism to automatically determine what type to include. This extension is (or should be) upward compatible with the already build in features to search for dimensions in the file itself or a `texutil.tuf` file. The dimensions are needed for TEX to reserve space and for the drivers to scale in an appropriate way (some drivers only accept width and/or height while others need scale values).

In practice, this means that one no longer has to specify the filetype and the (optional) method to be used. The inclusion macros first tries to locate the file itself and searches this file for its dimensions. It traverses over the search paths defined by `\setupexternalfigures` and as long as no dimensions can be found, we search for another instance of the type supported: `eps`, `mps`, `pdf`, `tif`, `png`, `jpg` or `mov`. Thereby we prefer outlines. Also, METAPOST files are recognized and handled properly with respect to color conversion, font inclusion and, if needed, direct conversion to PDF.

When still no dimensions are determined, we search for figure definition files (`texutil.tuf`), on all paths specified, and if needed, we generate a temporary utility file (calling TEXUTIL from within TEX). When no definition can be found, the files are searched again, this time for a suitable alternative, that is: a file with the same name, but another file type.

When a file is present, but no dimensions can be found, the file is inserted anyway, but without knowing the right aspect ratio, TEX probably cannot reserve the space needed. When one explicitly specifies an extension, this extension is honored as good as possible, but it only makes sense to pass extensions for METAPOST graphics (usually these extensions are numbers and these are recognized). When possible *don't specify a type or extension*.

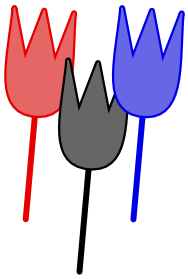
One should be aware of the way CONTEXT searches for the right files. Take for instance the next definition (beware, here we just define a figure, not include one):

```
\useexternalfigure[pragma][width=4cm,height=3cm]
```

Because no type is specified, CONTEXT scans for all the types it knows, on all the directories defined by `\setupexternalfigures[directory=...]`. Now let's imagine that we want to include the movie `pragma.mov`. We cannot determine its dimensions, and thereby one should be sure of at least the aspect ratio when specifying the width and height. Although in practice no problems are to be expected, one should realize that when there is for instance a `pragma.eps` available, CONTEXT will take the dimensions from that file and calculate the scaling factors from those. Lucky us that movies don't need those.

In fact, there are five possible results from the search:

- The file is not found at all and thereby not inserted.
- The file is found, but dimension can not be solved, therefore the file is inserted using the dimension specified.
- The file is found and its dimensions are derived from the file itself, thereby the file can be included at the scale or dimensions specified.



- The file is found, the dimensions can not be derived from the file itself but are available in `texutil.tuf`, thereby the file can be inserted the way we asked for. If needed a temporary utility file is generated.
- The file is found, the dimensions can not be derived from the file itself but can be derived from a file with the same name but a different suffix. With crossed fingers we can insert the file as requested.

Whatever method is chosen, ConTeXt reports the results, and if one really wants to know what is going on, just say:

```
\traceexternalfigurestrue
```

By saying:

```
\useexternalfigure[pragma][preset=no,width=4cm,height=3cm]
```

ConTeXt just searches the file without looking for its dimensions. In fact this is what movies need, thereby the fastest search is:

```
\useexternalfigure[pragma][type=mov,preset=no,width=4cm,height=3cm]
```

By default the `type` and/or file suffix is prepended to the default list, and therefore have the highest priority. This also makes the mechanism upward compatible, with the advantage that when generating PDF output, one does not need to adapt his or hers older sources, if only ConTeXt can find the right alternative. When possible, objects are used to share code.

When producing PostScript ready output, it makes sense to use `eps` illustrations (although when using dvipsone one can also use `tif`). When however producing PDF output, these `eps` graphics are to be converted to `pdf`. Instead of `tif` one can use `png`.

Converting `eps` into `pdf` can be done using GhostScript, for instance with Sebastian Rahtz Perl script `epstopdf`. The conversion part of this script is also present in TeXutil, and hooked into its boundingbox parsing routines (in TeXutil we also take care of high resolution bounding boxes).

As always, generating an illustration directory comes to:

```
texutil --figures <filenames>
```

When for instance one says:

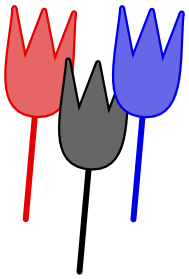
```
texutil --figures *.eps *.pdf *.tif *.png *.jpg
```

this utility parses all four file types, although in pdfTeX only `pdf`, `png`, `jpg` and `tif` make sense.

A new feature is `eps` to `pdf` conversion. The Sebastian Rahtz *page-dimension-forcing-method* is invoked by saying:

```
texutil --figures --epstopdf *.eps
```

Thereby TeXutil calls GhostScript (using the often available `gs` directive) which takes care of converting the files. The resulting files are (at least at this moment) larger in size than the ones distiller generates. If therefore one wants to use distiller instead, the next call is to be preferred:



```
texutil --figures --epspage <filenames>
```

This time TEXUTIL prepares the `eps` files by adding a suitable page size. The script is save against multiple processing of the files. When done, one can use distiller to process the files. In the process rubbish (headers and trailers) is stripped off.

Let's show one final example of including external figures now, just look careful, we don't have to specify any type. Just to force consistency, we inherit the settings.

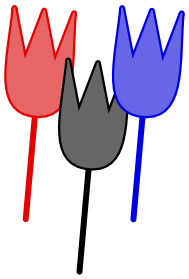
```
\useexternalfigure [europe]  [euro-10] [width=.3\hsize]
\useexternalfigure [holland] [euro-nl] [europe]
\useexternalfigure [germany] [euro-de] [europe]
\useexternalfigure [england] [euro-en] [europe]

\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

More experienced users probably already know that in CONTEXT there are several ways to specify figure dimensions: width/height, scale, factors which are related to the body font size, and of course automatic scaling to width, height or whatever suits best, or scale the figure in such a way that it fits in the available space. The latter methods are especially handy in documents with thousands of pages, where visual checks are sort of impossible. Automatic scaling could already be combined with limiting the width or height, e.g. scale to the maximum width unless heigher than 10 cm. The next release also permits settings like: scale at 100% (`scale=1000`) but never exceed a width of `.8\hsize` and/or `.4\vsize`.



task force members	Tobias Burnus Gilbert van den Dobbelsteen Hans Hagen Taco Hoekwater
dedicated mailing list	ntg-context@ntg.nl
contacting authors	pragma@wxs.nl
examples, manuals and code	www.ntg.nl/context frambach.eco.rug.nl/pragma www.pragma-ade.nl
authors	Hans Hagen Ton Otten
processing date	May 11, 1999
current version	1999.5.10