

low level

TEX

lowlevel

Contents

2	Conditionals	5
2.1	Preamble	6
2.2	\TeX primitives	11
2.3	$\varepsilon\text{-}\text{\TeX}$ primitives	19
2.4	\LuaTeX primitives	21
2.5	\LuaMetaTeX primitives	25
2.6	For the brave	29
2.7	Relaxing	31
3	Boxes	35
3.1	Introduction	36
3.2	Boxes	36
3.3	\TeX primitives	37
3.4	$\varepsilon\text{-}\text{\TeX}$ primitives	39
3.5	\LuaTeX primitives	40
3.6	\LuaMetaTeX primitives	41
4	Expansion	49
4.1	Preamble	50
4.2	\TeX primitives	50
4.3	$\varepsilon\text{-}\text{\TeX}$ primitives	55
4.4	\LuaTeX primitives	57
4.5	\LuaMetaTeX primitives	58
4.6	Dirty tricks	68
5	Registers	72
5.1	Preamble	73
5.2	\TeX primitives	73
5.3	$\varepsilon\text{-}\text{\TeX}$ primitives	76
5.4	\LuaTeX primitives	76
5.5	\LuaMetaTeX primitives	77
6	Macros	78
6.1	Preamble	79
6.2	Definitions	79
6.3	Runaway arguments	89
6.4	Introspection	90
6.5	nesting	91

6.6	Prefixes	94
7	Grouping	97
7.1	Introduction	98
7.2	Pascal	98
7.3	T _E X	98
7.4	MetaPost	99
7.5	Lua	100
7.6	C	100
8	Security	102
8.1	Preamble	103
8.2	Flags	103
8.3	Complications	106
8.4	Introspection	107
9	Characters	108
9.1	Introduction	109
9.2	History	109
9.3	The heritage	110
9.4	The LMTX approach	111
10	Scope	115
10.1	Introduction	116
10.2	Registers	116
10.3	Allocation	118
10.4	Files	121
11	Paragraphs	124
11.1	Introduction	125
11.2	Paragraphs	125
11.3	Properties	129
11.4	Wrapping up	131
11.5	Hanging	131
11.6	Shapes	132
11.7	Modes	150
11.8	Normalization	150
11.9	Dirty tricks	150
12	Alignments	152
12.1	Introduction	153
12.2	Between the lines	155

12.3	Pre-, inter- and post-tab skips	157
12.4	Cell widths	160
12.5	Plugins	161
12.6	Pitfalls and tricks	164
12.7	Remark	167
13	Marks	169
13.1	Introduction	170
13.2	The basics	171
13.3	Migration	172
13.4	Tracing	174
13.5	High level commands	175
13.6	Pitfalls	178
14	Inserts	179
14.1	Introduction	180
14.2	The page builder	180
14.3	Inserts	182
14.4	Storing	184
14.5	Callbacks	184

Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

2 Conditionals

low level

TEX

conditionals

Contents

2.1	Preamble	6
2.2	T _E X primitives	11
2.3	ε -T _E X primitives	19
2.4	LuaT _E X primitives	21
2.5	LuaMetaT _E X primitives	25
2.6	For the brave	29
2.7	Relaxing	31

2.1 Preamble

2.1.1 Introduction

You seldom need the low level conditionals because there are quite some so called support macros available in ConT_EXt. For instance, when you want to compare two values (or more accurate: sequences of tokens), you can do this:

```
\doifelse {foo} {bar} {
  the same
} {
  different
}
```

But if you look in the ConT_EXt code, you will see that often we use primitives that start with `\if` in low level macros. There are good reasons for this. First of all, it looks familiar when you also code in other languages. Another reason is performance but that is only true in cases where the snippet of code is expanded very often, because T_EX is already pretty fast. Using low level T_EX can also be more verbose, which is not always nice in a document source. But, the most important reason (for me) is the layout of the code. I often let the look and feel of code determine the kind of coding. This also relates to the syntax highlighting that I am using, which is consistent for T_EX, MetaPost, Lua, etc. and evolved over decades. If code looks bad, it probably is bad. Of course this doesn't mean all my code looks good; you're warned. In general we can say that I often use `\if...` when coding core macros, and `\doifelse...` macros in (document) styles and modules.

In the sections below I will discuss the low level conditions in T_EX. For the often more convenient ConT_EXt wrappers you can consult the source of the system and support modules, the wiki and/or manuals.

Some of the primitives shown here are only available in LuaT_EX, and some only in LuaMetaT_EX. We could do without them for decades but they were added to these engines because of convenience and, more important, because then made for nicer code. Of course there's also the fun aspect. This manual is not an invitation to use these very low level primitives in your document source. The ones that probably make most sense are `\ifnum`, `\ifdim` and `\ifcase`. The others are often wrapped into support macros that are more convenient.

In due time I might add more examples and explanations. Also, maybe some more tests will show up as part of the LuaMetaT_EX project.

2.1.2 Number and dimensions

Numbers and dimensions are basic data types in T_EX. When you enter one, a number is just that but a dimension gets a unit. Compare:

```
1234
1234pt
```

If you also use MetaPost, you need to be aware of the fact that in that language there are not really dimensions. The `post` part of the name implies that eventually a number becomes a PostScript unit which represents a base point (bp) in T_EX. When in MetaPost you entry `1234pt` you actually multiply 1234 by the variable `pt`. In T_EX on the other hand, a unit like `pt` is one of the keywords that gets parsed. Internally dimensions are also numbers and the unit (keyword) tells the scanner what multiplier to use. When that multiplier is one, we're talking of scaled points, with the unit `sp`.

```
\the\dimexpr 12.34pt \relax
\the\dimexpr 12.34sp \relax
\the\dimexpr 12.99sp \relax
\the\dimexpr 1234sp  \relax
\the\numexpr 1234   \relax
```

```
12.34pt
0.00018pt
0.00018pt
0.01883pt
1234
```

When we serialize a dimension it always shows the dimension in points, unless we serialize it as number.

```
\scratchdimen1234sp
\number\scratchdimen
\the\scratchdimen
```

```
1234
0.01883pt
```

When a number is scanned, the first thing that is taken care of is the sign. In many cases, when T_EX scans for something specific it will ignore spaces. It will happily accept multiple signs:

```
\number +123
\number +++123
\number + + + 123
\number +-+-+123
\number --123
\number ---123
```

```
123
123
123
123
123
-123
```

Watch how the negation accumulates. The scanner can handle decimal, hexadecimal and octal numbers:

```
\number -123
\number -"123
\number -'123
```

```
-123
-291
-83
```

A dimension is scanned like a number but this time the scanner checks for upto three parts: an either or not signed number, a period and a fraction. Here no number means zero, so the next is valid:

```
\the\dimexpr . pt \relax
\the\dimexpr 1. pt \relax
```

```
\the\dimexpr .1pt \relax
\the\dimexpr 1.1pt \relax
```

```
0.0pt
1.0pt
0.1pt
1.1pt
```

Again we can use hexadecimal and octal numbers but when these are entered, there can be no fractional part.

```
\the\dimexpr 16 pt \relax
\the\dimexpr "10 pt \relax
\the\dimexpr '20 pt \relax
```

```
16.0pt
16.0pt
16.0pt
```

The reason for discussing numbers and dimensions here is that there are cases where when T_EX expects a number it will also accept a dimension. It is good to know that for instance a macro defined with `\chardef` or `\mathchardef` also is treated as a number. Even normal characters can be numbers, when prefixed by a ``` (backtick).

The maximum number in T_EX is 2147483647 so we can do this:

```
\scratchcounter2147483647
```

but not this

```
\scratchcounter2147483648
```

as it will trigger an error. A dimension can be positive and negative so there we can do at most:

```
\scratchdimen 1073741823sp
\scratchdimen1073741823sp
\number\scratchdimen
\the\scratchdimen
\scratchdimen16383.99998pt
\number\scratchdimen
\the\scratchdimen
```

1073741823
 16383.99998pt
 1073741823
 16383.99998pt

We can also do this:

```
\scratchdimen16383.99999pt
\number\scratchdimen
\the\scratchdimen
```

1073741823
 16383.99998pt

but the next one will fail:

```
\scratchdimen16383.9999999pt
```

Just keep in mind that T_EX scans both parts as number so the error comes from checking if those numbers combine well.

```
\ifdim 16383.99999 pt = 16383.99998 pt the same \else different \fi
\ifdim 16383.999979 pt = 16383.999980 pt the same \else different \fi
\ifdim 16383.999987 pt = 16383.999991 pt the same \else different \fi
```

Watch the difference in dividing, the / rounds, while the : truncates.

the same
 the same
 the same

You need to be aware of border cases, although in practice they never really are a problem:

```
\ifdim \dimexpr16383.99997 pt/2\relax = \dimexpr 16383.99998 pt/2\relax
  the same \else different
\fi
\ifdim \dimexpr16383.99997 pt:2\relax = \dimexpr 16383.99998 pt:2\relax
  the same \else different
\fi
```

different
 the same

```
\ifdim \dimexpr1.99997 pt/2\relax = \dimexpr 1.99998 pt/2\relax
  the same \else different
```

```
\fi
```

```
\ifdim \dimexpr1.99997 pt:2\relax = \dimexpr 1.99998 pt:2\relax
  the same \else different
```

```
\fi
```

different

the same

```
\ifdim \dimexpr1.999999 pt/2\relax = \dimexpr 1.9999995 pt/2\relax
  the same \else different
```

```
\fi
```

```
\ifdim \dimexpr1.999999 pt:2\relax = \dimexpr 1.9999995 pt:2\relax
  the same \else different
```

```
\fi
```

the same

the same

This last case demonstrates that at some point the digits get dropped (still assuming that the fraction is within the maximum permitted) so these numbers then are the same. Anyway, this is not different in other programming languages and just something you need to be aware of.

2.2 T_EX primitives

2.2.1 \if

I seldom use this one. Internally T_EX stores (and thinks) in terms of tokens. If you see for instance `\def` or `\dimen` or `\hbox` these all become tokens. But characters like `A` or `@` also become tokens. In this test primitive all non-characters are considered to be the same. In the next examples this is demonstrated.

```
[\if AB yes\else nop\fi]
```

```
[\if AA yes\else nop\fi]
```

```
[\if CDyes\else nop\fi]
```

```
[\if CCyes\else nop\fi]
```

```
[\if\dimen\font yes\else nop\fi]
```

```
[\if\dimen\font yes\else nop\fi]
```

Watch how spaces after the two characters are kept: [nop] [yes] [nop] [yes] [yes] [yes]. This primitive looks at the next two tokens but when doing so it expands. Just look at the following:

```
\def\AA{AA}%
\def\AB{AB}%
[\if\AA yes\else nop\fi]
[\if\AB yes\else nop\fi]
```

We get: [yes] [nop].

2.2.2 \ifcat

In \TeX characters (in the input) get interpreted according to their so called catcodes. The most common are letters (alphabetic) and and other (symbols) but for instance the backslash has the property that it starts a command, the dollar signs trigger math mode, while the curly braced deal with grouping. If for instance either or not the ampersand is special (for instance as column separator in tables) depends on the macro package.

```
[\ifcat AB yes\else nop\fi]
[\ifcat AA yes\else nop\fi]
[\ifcat CDyes\else nop\fi]
[\ifcat CCyes\else nop\fi]
[\ifcat Clyes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
```

This time we also compare a letter with a number: [yes] [yes] [yes] [yes] [nop] [yes] [yes]. In that case the category codes differ (letter vs other) but in this test comparing the letters result in a match. This is a test that is used only once in $\text{Con}\TeX$ t and even that occasion is dubious and will go away.

You can use `\noexpand` to prevent expansion:

```
\def\A{A}%
\let\B B%
\def\C{D}%
\let\D D%
[\ifcat\noexpand\A Ayes\else nop\fi]
[\ifcat\noexpand\B Byes\else nop\fi]
[\ifcat\noexpand\C Cyes\else nop\fi]
[\ifcat\noexpand\C Dyes\else nop\fi]
```

```
[\ifcat\noexpand\D Dyes\else nop\fi]
```

We get: [nop] [yes] [nop] [nop] [yes], so who still thinks that T_EX is easy to understand for a novice user?

2.2.3 `\ifnum`

This condition compares its argument with another one, separated by an `<`, `=` or `>` character.

```
\ifnum\scratchcounter<0  
    less than  
\else\ifnum\scratchcounter>0  
    more than  
\else  
    equal to  
\fi zero
```

This is one of these situations where a dimension can be used instead. In that case the dimension is in scaled points.

```
\ifnum\scratchdimen<0  
    less than  
\else\ifnum\scratchdimen>0  
    more than  
\else  
    equal to  
\fi zero
```

Of course this equal treatment of a dimension and number is only true when the dimension is a register or box property.

2.2.4 `\ifdim`

This condition compares one dimension with another one, separated by an `<`, `=` or `>` sign.

```
\ifdim\scratchdimen<0pt  
    less than  
\else\ifdim\scratchdimen>0pt  
    more than
```

```
\else
  equal to
\fi zero
```

While when comparing numbers a dimension is a valid quantity but here you cannot mix them: something with a unit is expected.

2.2.5 `\ifodd`

This one can come in handy, although in ConT_EXt it is only used in checking for an odd of even page number.

```
\scratchdimen 3sp
\scratchcounter4

\ifodd\scratchdimen very \else not so \fi odd
\ifodd\scratchcounter very \else not so \fi odd
```

As with the previously discussed `\ifnum` you can use a dimension variable too, which is then interpreted as representing scaled points. Here we get:

```
very odd
not so odd
```

2.2.6 `\ifvmode`

This is a rather trivial check. It takes no arguments and just is true when we're in vertical mode. Here is an example:

```
\hbox{\ifvmode\else\par\fi\ifvmode v\else h\fi mode}
```

We're always in horizontal mode and issuing a `\par` inside a horizontal box doesn't change that, so we get: `hmode`.

2.2.7 `\ifhmode`

As with `\ifvmode` this one has no argument and just tells if we're in vertical mode.

```
\vbox {
  \noindent \ifhmode h\else v\fi mode
\par
```



```

\ifhmode h\else \noindent v\fi mode
}

```

You can use it for instance to trigger injection of code, or prevent that some content (or command) is done more than once:

```

hmode
vmode

```

2.2.8 \ifmmode

Math is something very \TeX so naturally you can check if you're in math mode. here is an example of using this test:

```

\def\enforcemath#1{\ifmmode#1\else$ #1 $\fi}

```

Of course in reality macros that do such things are more advanced than this one.

2.2.9 \ifinner

```

\def\ShowMode
{\ifhmode \ifinner inner \fi hmode
\else\ifvmode \ifinner inner \fi vmode
\else\ifmmode \ifinner inner \fi mmode
\else \ifinner inner \fi unset
\fi\fi\fi}

```

```

\ShowMode \ShowMode

```

```

\ vbox{\ShowMode}

```

```

\ hbox{\ShowMode}

```

```

$\ShowMode$

```

```

$$\ShowMode$$

```

The first line has two tests, where the first one changes the mode to horizontal simply because a text has been typeset. Watch how display math is not inner.

```

vmode hmode
inner vmode

```

inner hmode
innermmode

mmode

By the way, moving the `\ifinner` test outside the branches (to the top of the macro) won't work because once the word `inner` is typeset we're no longer in vertical mode, if we were at all.

2.2.10 `\ifvoid`

A box is one of the basic concepts in \TeX . In order to understand this primitive we present four cases:

```
\setbox0\hbox{}          \ifvoid0 void \else content \fi
\setbox0\hbox{123}      \ifvoid0 void \else content \fi
\setbox0\hbox{} \box0   \ifvoid0 void \else content \fi
\setbox0\hbox to 10pt{} \ifvoid0 void \else content \fi
```

In the first case, we have a box which is empty but it's not void. It helps to know that internally an `hbox` is actually an object with a pointer to a linked list of nodes. So, the first two can be seen as:

```
hlist -> [nothing]
hlist -> 1 -> 2 -> 3 -> [nothing]
```

but in any case there is a `hlist`. The third case puts something in a `hlist` but then flushes it. Now we have not even the `hlist` any more; the box register has become void. The last case is a variant on the first. It is an empty box with a given width. The outcome of the four lines (with a box flushed in between) is:

```
content
content
```

```
void
content
```

So, when you want to test if a box is really empty, you need to test also its dimensions, which can be up to three tests, depending on your needs.

```
\setbox0\emptybox          \ifvoid0 void\else content\fi
\setbox0\emptybox \wd0=10pt \ifvoid0 void\else content\fi
```

```
\setbox0\hbox to 10pt {} \ifvoid0 void\else content\fi
\setbox0\hbox {} \wd0=10pt \ifvoid0 void\else content\fi
```

Setting a dimension of a void void (empty) box doesn't make it less void:

```
void
void
content
content
```

2.2.11 \ifhbox

This test takes a box number and gives true when it is an hbox.

2.2.12 \ifvbox

This test takes a box number and gives true when it is an vbox. Both a \vbox and \vtop are vboxes, the difference is in the height and depth and the baseline. In a \vbox the last line determines the baseline

```
vbox or vtop
```

```
vtop or vbox
```

And in a \vtop the first line takes control:

```
vbox or vtop
```

```
vtop or vbox
```

but, once wrapped, both internally are just vlists.

2.2.13 \ifx

This test is actually used a lot in ConT_EXt: it compares two token(list)s:

```
\ifx a b Y\else N\fi
\ifx ab Y\else N\fi
\def\A {a}\def\B{b}\ifx \A\B Y\else N\fi
\def\A{aa}\def\B{a}\ifx \A\B Y\else N\fi
\def\A {a}\def\B{a}\ifx \A\B Y\else N\fi
```

Here the result is: “NNNNY”. It does not expand the content, if you want that you need to use an `\edef` to create two (temporary) macros that get compared, like in:

```
\edef\TempA{...}\edef\TempB{...}\ifx\TempA\TempB ...\else ...\fi
```

2.2.14 `\ifeof`

This test checks if a the pointer in a given input channel has reached its end. It is also true when the file is not present. The argument is a number which relates to the `\openin` primitive that is used to open files for reading.

2.2.15 `\iftrue`

It does what it says: always true.

2.2.16 `\iffalse`

It does what it says: always false.

2.2.17 `\ifcase`

The general layout of an `\ifcase` tests is as follows:

```
\ifcase<number>
  when zero
\or
  when one
\or
  when two
\or
  ...
\else
  when something else
\fi
```

As in other places a number is a sequence of signs followed by one of more digits

2.3 ϵ -T_EX primitives

2.3.1 `\ifdefined`

This primitive was introduced for checking the existence of a macro (or primitive) and with good reason. Say that you want to know if `\MyMacro` is defined? One way to do that is:

```
\ifx\MyMacro\undefined
  {\bf undefined indeed}
\fi
```

This results in: **undefined indeed**, but is this macro really undefined? When T_EX scans your source and sees a the escape character (the forward slash) it will grab the next characters and construct a control sequence from it. Then it finds out that there is nothing with that name and it will create a hash entry for a macro with that name but with no meaning. Because `\undefined` is also not defined, these two macros have the same meaning and therefore the `\ifx` is true. Imagine that you do this many times, with different macro names, then your hash can fill up. Also, when a user defined `\undefined` you're suddenly get a different outcome.

In order to catch the last problem there is the option to test directly:

```
\ifdefined\MyOtherMacro \else
  {\bf also undefined}
\fi
```

This (or course) results in: **also undefined**, but the macro is still sort of defined (with no meaning). The next section shows how to get around this.

2.3.2 `\ifcsname`

A macro is often defined using a ready made name, as in:

```
\def\OhYes{yes}
```

The name is made from characters with catcode letter which means that you cannot use for instance digits or underscores unless you also give these characters that catcode, which is not that handy in a document. You can however use `\csname` to define a control sequence with any character in the name, like:

```
\expandafter\def\csname Oh Yes : 1\endcsname{yes}
```

Later on you can get this one with `\csname`:

```
\csname Oh Yes : 1\endcsname
```

However, if you say:

```
\csname Oh Yes : 2\endcsname
```

you won't get some result, nor a message about an undefined control sequence, but the name triggers a define anyway, this time not with no meaning (undefined) but as equivalent to `\relax`, which is why

```
\expandafter\ifx\csname Oh Yes : 2\endcsname\relax  
  {\bf relaxed indeed}  
\fi
```

is the way to test its existence. As with the test in the previous section, this can deplete the hash when you do lots of such tests. The way out of this is:

```
\ifcsname Oh Yes : 2\endcsname \else  
  {\bf unknown indeed}  
\fi
```

This time there is no hash entry created and therefore there is not even an undefined control sequence.

In Lua \TeX there is an option to return false in case of a messy expansion during this test, and in LuaMeta \TeX that is default. This means that tests can be made quite robust as it is pretty safe to assume that names that make sense are constructed from regular characters and not boxes, font switches, etc.

2.3.3 `\iffontchar`

This test was also part of the ε - \TeX extensions and it can be used to see if a font has a character.

```
\iffontchar\font`A  
  {\em This font has an A!}  
\fi
```

And, as expected, the outcome is: *"This font has an A!"*. The test takes two arguments,

the first being a font identifier and the second a character number, so the next checks are all valid:

```
\iffontchar\font      `A yes\else nop\fi\par
\iffontchar\nullfont `A yes\else nop\fi\par
\iffontchar\textfont0`A yes\else nop\fi\par
```

In the perspective of LuaMetaTeX I considered also supporting `\fontid` but it got a bit messy due to the fact that this primitive expands in a different way so this extension was rejected.

2.3.4 `\unless`

You can negate the results of a test by using the `\unless` prefix, so for instance you can replace:

```
\ifdim\scratchdimen=10pt
  \dosomething
\else\ifdim\scratchdimen<10pt
  \dosomething
\fi\fi
```

by:

```
\unless\ifdim\scratchdimen>10pt
  \dosomething
\fi
```

2.4 LuaTeX primitives

2.4.1 `\ifincsname`

As it had no real practical usage it might get dropped in LuaMetaTeX, so it will not be discussed here.

2.4.2 `\ifprimitive`

As it had no real practical usage due to limitations, this one is not available in LuaMetaTeX so it will not be discussed here.

2.4.3 `\ifabsnum`

This test is inherited from pdf \TeX and behaves like `\ifnum` but first turns a negative number into a positive one.

2.4.4 `\ifabsdim`

This test is inherited from pdf \TeX and behaves like `\ifdim` but first turns a negative dimension into a positive one.

2.4.5 `\ifcondition`

This is not really a test but in order to unstand that you need to know how \TeX internally deals with tests.

```
\ifdimen\scratchdimen>10pt
  \ifdim\scratchdimen<20pt
    result a
  \else
    result b
  \fi
\else
  result c
\fi
```

When we end up in the branch of “result a” we need to skip two `\else` branches after we're done. The `\if..` commands increment a level while the `\fi` decrements a level. The `\else` needs to be skipped here. In other cases the true branch needs to be skipped till we end up a the right `\else`. When doing this skipping, \TeX is not interested in what it encounters beyond these tokens and this skipping (therefore) goes real fast but it does see nested conditions and doesn't interpret grouping related tokens.

A side effect of this is that the next is not working as expected:

```
\def\ifmorethan{\ifdim\scratchdimen>}
\def\iflessthan{\ifdim\scratchdimen<}

\ifmorethan10pt
  \iflessthan20pt
    result a
  \else
```



```

        result b
    \fi
\else
    result c
\fi

```

The `\iflessthan` macro is not seen as an `\if...` so the nesting gets messed up. The solution is to fool the scanner in thinking that it is. Say we have:

```

\scratchdimen=25pt

\def\ifmorethan{\ifdim\scratchdimen>}
\def\iflessthan{\ifdim\scratchdimen<}

```

and:

```

\ifcondition\ifmorethan10pt
    \ifcondition\iflessthan20pt
        result a
    \else
        result b
    \fi
\else
    result c
\fi

```

When we expand this snippet we get: “result b” and no error concerning a failure in locating the right `\fi`'s. So, when scanning the `\ifcondition` is seen as a valid `\if...` but when the condition is really expanded it gets ignored and the `\ifmorethan` has better come up with a match or not.

In this perspective it is also worth mentioning that nesting problems can be avoided this way:

```

\def\WhenTrue {something \iftrue ...}
\def\WhenFalse{something \iffalse ...}

\ifnum\scratchcounter>123
    \let\next\WhenTrue
\else
    \let\next\WhenFalse
\fi
\next

```

This trick is mentioned in The \TeX book and can also be found in the plain \TeX format. A variant is this:

```
\ifnum\scratchcounter>123
  \expandafter\WhenTrue
\else
  \expandafter\WhenFalse
\fi
```

but using `\expandafter` can be quite intimidating especially when there are multiple in a row. It can also be confusing. Take this: an `\ifcondition` expects the code that follows to produce a test. So:

```
\def\ifwhatever#1%
  {\ifdim#1>10pt
    \expandafter\iftrue
  \else
    \expandafter\iffalse
  \fi}

\ifcondition\ifwhatever{10pt}
  result a
\else
  result b
\fi
```

This will not work! The reason is in the already mentioned fact that when we end up in the greater than 10pt case, the scanner will happily push the `\iftrue` after the `\fi`, which is okay, but when skipping over the `\else` it sees a nested condition without matching `\fi`, which makes it fail. I will spare you a solution with lots of nasty tricks, so here is the clean solution using `\ifcondition`:

```
\def\truecondition {\iftrue}
\def>falsecondition{\iffalse}

\def\ifwhatever#1%
  {\ifdim#1>10pt
    \expandafter\truecondition
  \else
    \expandafter>falsecondition
  \fi}
```

```

\ifcondition\ifwhatever{10pt}
  result a
\else
  result b
\fi

```

It will be no surprise that the two macros at the top are predefined in ConT_EXt. It might be more of a surprise that at the time of this writing the usage in ConT_EXt of this `\ifcondition` primitive is rather minimal. But that might change.

As a further teaser I'll show another simple one,

```

\def\HowOdd#1{\unless\ifnum\numexpr ((#1):2)*2\relax=\numexpr#1\relax}

\ifcondition\HowOdd{1}very \else not so \fi odd
\ifcondition\HowOdd{2}very \else not so \fi odd
\ifcondition\HowOdd{3}very \else not so \fi odd

```

This renders:

```

very odd
not so odd
very odd

```

The code demonstrates several tricks. First of all we use `\numexpr` which permits more complex arguments, like:

```

\ifcondition\HowOdd{4+1}very \else not so \fi odd
\ifcondition\HowOdd{2\scratchcounter+9}very \else not so \fi odd

```

Another trick is that we use an integer division (the `:`) which is an operator supported by LuaMetaT_EX.

2.5 LuaMetaT_EX primitives

2.5.1 `\ifcmpnum`

This one is part of a set of three tests that all are a variant of a `\ifcase` test. A simple example of the first test is this:

```

\ifcmpnum 123 345 less \or equal \else more \fi

```

The test scans for two numbers, which of course can be registers or expressions, and sets the case value to 0, 1 or 2, which means that you then use the normal `\or` and `\else` primitives for follow up on the test.

2.5.2 `\ifchknum`

This test scans a number and when it's okay sets the case value to 1, and otherwise to 2. So you can do the next:

```
\ifchknum 123\or good \else bad \fi
\ifchknum bad\or good \else bad \fi
```

An error message is suppressed and the first `\or` can be seen as a sort of recovery token, although in fact we just use the fast scanner mode that comes with the `\ifcase`: because the result is 1 or 2, we never see invalid tokens.

2.5.3 `\ifnumval`

A sort of combination of the previous two is `\ifnumval` which checks a number but also if it's less, equal or more than zero:

```
\ifnumval 123\or less \or equal \or more \else error \fi
\ifnumval bad\or less \or equal \or more \else error \fi
```

You can decide to ignore the bad number or do something that makes more sense. Often the to be checked value will be the content of a macro or an argument like `#1`.

2.5.4 `\ifcmpdim`

This test is like `\ifcmpnum` but for dimensions.

2.5.5 `\ifchkdim`

This test is like `\ifchknum` but for dimensions. The last checked value is available as `\lastchknum`.

2.5.6 `\ifdimval`

This test is like `\ifnumval` but for dimensions. The last checked value is available as `\lastchkdim`.

2.5.7 `\iftok`

Although this test is still experimental it can be used. What happens is that two to be compared ‘things’ get scanned for. For each we first gobble spaces and `\relax` tokens. Then we can have several cases:

1. When we see a left brace, a list of tokens is scanned upto the matching right brace.
2. When a reference to a token register is seen, that register is taken as value.
3. When a reference to an internal token register is seen, that register is taken as value.
4. When a macro is seen, its definition becomes the to be compared value.
5. When a number is seen, the value of the corresponding register is taken

An example of the first case is:

```
\iftok {abc} {def}%
...
\else
...
\fi
```

The second case goes like this:

```
\iftok\scratchtoksone\scratchtokstwo
...
\else
...
\fi
```

Case one and four mixed:

```
\iftok{123}\TempX
...
\else
...
\fi
```

The last case is more a catch: it will issue an error when no number is given. Eventually that might become a bit more clever (depending on our needs.)

2.5.8 `\ifcstok`

There is a subtle difference between this one and `iftok`: spaces and `\relax` tokens are

skipped but nothing gets expanded. So, when we arrive at the to be compared ‘things’ we look at what is there, as-is.

2.5.9 `\iffrozen`

This is an experimental test. Commands can be defined with the `\frozen` prefix and this test can be used to check if that has been the case.

2.5.10 `\ifprotected`

Commands can be defined with the `\protected` prefix (or in `ConTeXt`, for historic reasons, with `\unexpanded`) and this test can be used to check if that has been the case.

2.5.11 `\ifusercmd`

This is an experimental test. It can be used to see if the command is defined at the user level or is a build in one. This one might evolve.

2.5.12 `\ifarguments`

This conditional can be used to check how many arguments were matched. It only makes sense when used with macros defined with the `\tolerant` prefix and/or when the sentinel `\ignorearguments` after the arguments is used. More details can be found in the lowlevel macros manual.

2.5.13 `\orelse`

This is not really a test primitive but it does act that way. Say that we have this:

```
\ifdim\scratchdimen>10pt
  case 1
\else\ifdim\scratchdimen<20pt
  case 2
\else\ifcount\scratchcounter>10
  case 3
\else\ifcount\scratchcounter<20
  case 4
\fi\fi\fi\fi
```

A bit nicer looks this:

```
\ifdim\scratchdimen>10pt
  case 1
\orelse\ifdim\scratchdimen<20pt
  case 2
\orelse\ifcount\scratchcounter>10
  case 3
\orelse\ifcount\scratchcounter<20
  case 4
\fi
```

We stay at the same level. Sometimes a more flat test tree had advantages but if you think that it gives better performance then you will be disappointed. The fact that we stay at the same level is compensated by a bit more parsing, so unless you have millions such cases (or expansions) it might make a bit of a difference. As mentioned, I'm a bit sensitive for how code looks so that was the main motivation for introducing it. There is a companion `\orunless` continuation primitive.

A rather neat trick is the definition of `\quitcondition`:

```
\def\quitcondition{\orelse\iffalse}
```

This permits:

```
\ifdim\scratchdimen>10pt
  case 1a
  \quitcondition
  case 4b
\fi
```

where, of course, the quitting normally is the result of some intermediate extra test. But let me play safe here: beware of side effects.

2.6 For the brave

2.6.1 Full expansion

If you don't understand the following code, don't worry. There is seldom much reason to go this complex but obscure \TeX code attracts some users so ...

When you have a macro that has for instance assignments, and when you expand that macro inside an `\edef`, these assignments are not actually expanded but tokenized. In LuaMetaTeX there is a way to apply these assignments without side effects and that feature can be used to write a fully expandable user test. For instance:

```
\def\truecondition {\iftrue}
\def>falsecondition{\iffalse}

\def\fontwithidhaschar#1#2%
  {\beginlocalcontrol
   \scratchcounter\numexpr\fontid\font\relax
   \setfontid\numexpr#1\relax
   \endlocalcontrol
   \iffontchar\font\numexpr#2\relax
   \beginlocalcontrol
   \setfontid\scratchcounter
   \endlocalcontrol
   \expandafter\truecondition
  \else
   \expandafter>falsecondition
  \fi}
```

The `\iffontchar` test doesn't handle numeric font id, simply because at the time it was added to ε -TeX, there was no access to these id's. Now we can do:

```
\edef\foo{\fontwithidhaschar{1} {75}yes\else nop\fi} \meaning\foo
\edef\foo{\fontwithidhaschar{1}{999}yes\else nop\fi} \meaning\foo

[\ifcondition\fontwithidhaschar{1} {75}yes\else nop\fi]
[\ifcondition\fontwithidhaschar{1}{999}yes\else nop\fi]
```

These result in:

```
macro:yes
macro:nop
```

```
[yes]
[nop]
```

If you remove the `\immediateassignment` in the definition above then the typeset results are still the same but the meanings of `\foo` look different: they contain the assignments and the test for the character is actually done when constructing the content of

the `\edef`, but for the current font. So, basically that test is now useless.

2.6.2 User defined if's

There is a `\newif` macro that defines three other macros:

```
\newif\ifOnMyOwnTerms
```

After this, not only `\ifOnMyOwnTerms` is defined, but also:

```
\OnMyOwnTermstrue
\OnMyOwnTermsfalse
```

These two actually are macros that redefine `\ifOnMyOwnTerms` to be either equivalent to `\iftrue` and `\iffalse`. The (often derived from plain `TEX`) definition of `\newif` is a bit of a challenge as it has to deal with removing the `if` in order to create the two extra macros and also make sure that it doesn't get mixed up in a catcode jungle.

In `ConTEXt` we have a variant:

```
\newconditional\MyConditional
```

that can be used with:

```
\settrue\MyConditional
\setfalse\MyConditional
```

and tested like:

```
\ifconditional\MyConditional
...
\else
...
\fi
```

This one is cheaper on the hash and doesn't need the two extra macros per test. The price is the use of `\ifconditional`, which is *not* to be confused with `\ifcondition` (it has bitten me already a few times).

2.7 Relaxing

When `TEX` scans for a number or dimension it has to check tokens one by one. On the

case of a number, the scanning stops when there is no digit, in the case of a dimension the unit determine the end of scanning. In the case of a number, when a token is not a digit that token gets pushed back. When digits are scanned a trailing space or `\relax` is pushed back. Instead of a number of dimension made from digits, periods and units, the scanner also accepts registers, both the direct accessors like `\count` and `\dimen` and those represented by one token. Take these definitions:

```
\newdimen\MyDimenA \MyDimenA=1pt \dimen0=\MyDimenA
\newdimen\MyDimenB \MyDimenB=2pt \dimen2=\MyDimenB
```

I will use these to illustrate the side effects of scanning. Watch the spaces in the result.

First I show what effect we want to avoid. When second argument contains a number (digits) the zero will become part of it so we actually check `\dimen00` here.

```
\def\whatever#1#2%
  {\ifdim#1=#20\else1\fi}

\whatever{1pt}{2pt}           [ macro:1]
\whatever{1pt}{1pt}           [ macro:0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]
```

The solution is to add a space but watch how that one can end up in the result:

```
\def\whatever#1#2%
  {\ifdim#1=#2 0\else1\fi}

\whatever{1pt}{2pt}           [ macro:1]
\whatever{1pt}{1pt}           [ macro:0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:0]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]
```

A variant is using `\relax` and this time we get this token retained in the output.

```
\def\whatever#1#2%
  {\ifdim#1=#2\relax0\else1\fi}
```

```

\whatever{1pt}{2pt}          [ macro:1]
\whatever{1pt}{1pt}         [ macro:\relax 0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:\relax 0]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]

```

A solution that doesn't have side effects of forcing the end of a number (using a space or `\relax` is one where we use expressions. The added overhead of scanning expressions is taken for granted because the effect is what we like:

```

\def\whatever#1#2%
  {\ifdim\dimexpr#1\relax=\dimexpr#2\relax0\else1\fi}

\whatever{1pt}{2pt}          [ macro:1]
\whatever{1pt}{1pt}         [ macro:0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:0]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]

```

Just for completeness we show a more obscure trick: this one hides assignments to temporary variables. Although performance is okay, it is the least efficient one so far.

```

\def\whatever#1#2%
  {\beginlocalcontrol
   \MyDimenA#1\relax
   \MyDimenB#2\relax
  \endlocalcontrol
  \ifdim\MyDimenA=\MyDimenB0\else1\fi}

\whatever{1pt}{2pt}          [ macro:1]
\whatever{1pt}{1pt}         [ macro:0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:0]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]

```

It is kind of a game to come up with alternatives but for sure those involve dirty tricks and more tokens (and runtime). The next can be considered a dirty trick too: we use a special variant of `\relax`. When a number is scanned it acts as `\relax`, but otherwise it just is ignored and disappears.

Relaxing

```

\def\whatever#1#2%
  {\ifdim#1=#2\norelax0\else1\fi}

\whatever{1pt}{2pt}          [ macro:1]
\whatever{1pt}{1pt}         [ macro:0]
\whatever{\dimen 0}{\dimen 2} [ macro:1]
\whatever{\dimen 0}{\dimen 0} [ macro:0]
\whatever \MyDimenA \MyDimenB [ macro:1]
\whatever \MyDimenA \MyDimenB [ macro:1]

```

2.7 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

3 Boxes

low level

TEX

boxes

Contents

3.1	Introduction	36
3.2	Boxes	36
3.3	$\text{T}_{\text{E}}\text{X}$ primitives	37
3.4	$\varepsilon\text{-T}_{\text{E}}\text{X}$ primitives	39
3.5	Lua $\text{T}_{\text{E}}\text{X}$ primitives	40
3.6	LuaMeta $\text{T}_{\text{E}}\text{X}$ primitives	41

3.1 Introduction

An average Con $\text{T}_{\text{E}}\text{X}$ t user will not use the low level box primitives but a basic understanding of how $\text{T}_{\text{E}}\text{X}$ works doesn't hurt. In fact, occasionally using a box command might bring a solution not easily achieved otherwise, simply because a more high level interface can also be in the way.

The best reference is of course The $\text{T}_{\text{E}}\text{X}$ book so if you're really interested in the details you should get a copy of that book. Below I will not go into details about all kind of glues, kerns and penalties, just boxes it is.

This explanation will be extended when I feel the need (or users have questions that can be answered here).

3.2 Boxes

This paragraph of text is made from lines that contain words that themselves contain symbolic representations of characters. Each line is wrapped in a so called horizontal box and eventually those lines themselves get wrapped in what we call a vertical box.

When we expose some details of a paragraph it looks like this:

H	This is a rather narrow	LH:0.000 LS:0.000	RS:13.685	RH:0.000
H	paragraph blown up a	LH:0.000 LS:0.000	RS:19.008	RH:0.000
H	bit. Here we use a flush	LH:0.000 LS:0.000	RS:7.557	RH:0.000
H	left, aka ragged right,	LH:0.000 LS:0.000	RS:19.706	RH:0.000
H	approach.	LH:0.000 LS:0.000	RS:0.000	RH:0.000

The left only shows the boxes, the variant at the right shows (font) kerns and glue too. Because we flush left, there is rather strong right skip glue at the right boundary of the box. If font kerns show up depends on the font, not all fonts have them (or have only a few). The glyphs themselves are also kind of boxed, as their dimensions determine the area that they occupy:

This is a rather ...

But, internally they are not really boxed, as they already are a single quantity. The same is true for rules: they are just blobs with dimensions. A box on the other hand wraps a linked list of so called nodes: glyphs, kerns, glue, penalties, rules, boxes, etc. It is a container with properties like width, height, depth and shift.

3.3 T_EX primitives

The box model is reflected in T_EX's user interface but not by that many commands, most noticeably `\hbox`, `\vbox` and `\vtop`. Here is an example of the first one:

```
\hbox width 10cm{text}
\hbox width 10cm height 1cm depth 5mm{text}
text \raise5mm\hbox{text} text
```

The `\raise` and `\lower` commands behave the same but in opposite directions. One could as well have been defined in terms of the other.

```
text \raise 5mm \hbox to 2cm {text}
text \lower -5mm \hbox to 2cm {text}
text \raise -5mm \hbox to 2cm {text}
text \lower 5mm \hbox to 2cm {text}
```

```

      .text.      .text.
text.      text.      text.      text.
      .text.      .text.
```

A box can be moved to the left or right but, believe it or not, in ConT_EXt we never use that feature, probably because the consequences for the width are such that we can as well use kerns. Here are some examples:

```
text \vbox{\moveleft 5mm \hbox {left}}text !
text \vbox{\moveright 5mm \hbox{right}}text !
```


lefttext ! text righttext !

```
text \vbox{\moveleft 25mm \hbox {left}}text !
text \vbox{\moveright 25mm \hbox{right}}text !
```

left text text ! text righttext !

Code like this will produce a complaint about an underfull box but we can easily get around that:

```
text \raise 5mm \hbox to 2cm {\hss text}
text \lower -5mm \hbox to 2cm {text\hss}
text \raise -5mm \hbox to 2cm {\hss text}
text \lower 5mm \hbox to 2cm {text\hss}
```

The `\hss` primitive injects a glue that when needed will fill up the available space. So, here we force the text to the right or left.

```

┌──────────┐ ┌──text──┐
text.         text.         text.         text.
└──────────┘ └──text──┘
┌──────────┐ ┌──text──┐

```

We have three kind of boxes: `\hbox`, `\vbox` and `\vtop`:

```
\hbox{\strut height and depth\strut}
\vbox{\hsize 4cm \strut height and depth\par and width\strut}
\vtop{\hsize 4cm \strut height and depth\par and width\strut}
```

A `\vbox` aligns at the bottom and a `\vtop` at the top. I have added some so called struts to enforce a consistent height and depth. A strut is an invisible quantity (consider it a black box) that enforces consistent line dimensions: height and depth.

```

┌──────────┐
height and depth
└──────────┘
┌──────────┐ ┌──and width──┐ ┌──────────┐
height and depth and width height and depth
└──────────┘ └──────────┘ └──────────┘
┌──────────┐
and width
└──────────┘

```

You can store a box in a register but you need to be careful not to use a predefined one. If you need a lot of boxes you can reserve some for your own:

```
\newbox\MySpecialBox
```

but normally you can do with one of the scratch registers, like 0, 2, 4, 6 or 8, for local

boxes, and 1, 3, 5, 7 and 9 for global ones. Registers are used like:

```
\setbox0\hbox{here}
\global\setbox1\hbox{there}
```

In ConT_EXt you can also use

```
\setbox\scratchbox \hbox{here}
\setbox\scratchboxone\hbox{here}
\setbox\scratchboxtwo\hbox{here}
```

and some more. In fact, there are quite some predefined scratch registers (boxes, dimensions, counters, etc). Feel free to investigate further.

When a box is stored, you can consult its dimensions with `\wd`, `\ht` and `\dp`. You can of course store them for later use.

```
\scratchwidth \wd\scratchbox
\scratchheight\ht\scratchbox
\scratchdepth \dp\scratchbox
\scratchtotal \dimexpr\ht\scratchbox+\dp\scratchbox\relax
\scratchtotal \htdp\scratchbox
```

The last line is ConT_EXt specific. You can also set the dimensions

```
\wd\scratchbox 10cm
\ht\scratchbox 10mm
\dp\scratchbox 5mm
```

So you can cheat! A box is placed with `\copy`, which keeps the original intact or `\box` which just inserts the box and then wipes the register. In practice you seldom need a copy, which is more expensive in runtime anyway. Here we use copy because it serves the examples.

```
\copy\scratchbox
\box \scratchbox
```

3.4 ε -T_EX primitives

The ε -T_EX extensions don't add something relevant for boxes, apart from that you can use the expressions mechanism to mess around with their dimensions. There is a mechanism for typesetting r2l within a paragraph but that has limited capabilities and doesn't

change much as it's mostly a way to trick the backend into outputting a stretch of text in the other direction. This feature is not available in Lua \TeX because it has an alternative direction mechanism.

3.5 Lua \TeX primitives

The concept of boxes is the same in Lua \TeX as in its predecessors but there are some aspects to keep in mind. When a box is typeset this happens in Lua \TeX :

1. A list of nodes is constructed. In Lua \TeX this is a double linked list (so that it can easily be manipulated in Lua) but \TeX itself only uses the forward links.
2. That list is hyphenated, that is: so called discretionary nodes are injected. This depends on the language properties of the glyph (character) nodes.
3. Then ligatures are constructed, if the font has such combinations. When this built-in mechanism is used, in Con \TeX t we speak of base mode.
4. After that inter-character kerns are applied, if the font provides them. Again this is a base mode action.
5. Finally the box gets packaged:
 - In the case of a horizontal box, the list is packaged in a hlist node, basically one liner, and its dimensions are calculated and set.
 - In the case of a vertical box, the paragraph is broken into one or more lines, without hyphenation, with optimal hyphenation or in the worst case with so called emergency stretch applied, and the result becomes a vlist node with its dimensions set.

In traditional \TeX the first four steps are interwoven but in Lua \TeX we need them split because the step 5 can be overloaded by a callback. In that case steps 3 and 4 (and maybe 2) are probably also overloaded, especially when you bring handling of fonts under Lua control.

New in Lua \TeX are three packers: `\hpack`, `\vpack` and `\tpack`, which are companions to `\hbox`, `\vbox` and `\vtop` but without the callbacks applied. Using them is a bit tricky as you never know if a callback should be applied, which, because users can often add their own Lua code, is not something predictable.

Another box related extension is `direction`. There are four possible directions but because in LuaMeta \TeX there are only two. Because this model has been upgraded, it will

be discusses in the next section. A ConT_EXt user is supposed to use the official ConT_EXt interfaces in order to be downward compatible.

3.6 LuaMetaT_EX primitives

There are two possible directions: left to right (the default) and right to left for Hebrew and Arabic. Here is an example that shows how it'd done with low level directives:

```
\hbox direction 0 {from left to right}
\hbox direction 1 {from right to left}
```

from left to right
tfel ot thgir morf

A low level direction switch is done with:

```
\hbox direction 0
  {from left to right \textdirection 1 from right to left}
\hbox direction 1
  {from right to left \textdirection 1 from left to right}
```

from left to right tfel ot thgir morf
thgir ot tfel morf tfel ot thgir morf

but actually this is kind of *not done* in ConT_EXt, because there you are supposed to use the proper direction switches:

```
\naturalhbox {from left to right}
\reversehbox {from right to left}
\naturalhbox {from left to right \righttoleft from right to left}
\reversehbox {from right to left \lefttoright from left to right}
```

from left to right
tfel ot thgir morf
from left to right tfel ot thgir morf
from left to right tfel ot thgir morf

Often more is needed to properly support right to left typesetting so using the ConT_EXt commands is more robust.

In LuaMetaT_EX the box model has been extended a bit, this as a consequence of dropping the vertical directional typesetting, which never worked well. In previous sections

we discussed the properties width, height and depth and the shift resulting from a `\raise`, `\lower`, `\moveleft` and `\moveright`. Actually, the shift is also used in for instance positioning math elements.

The way shifting influences dimensions can be somewhat puzzling. Internally, when \TeX packages content in a box there are two cases:

- When a horizontal box is made, and height - shift is larger than the maximum height so far, that delta is taken. When depth + shift is larger than the current depth, then that depth is adapted. So, a shift up influences the height and a shift down influences the depth.
- In the case of vertical packaging, when width + shift is larger than the maximum box (line) width so far, that maximum gets bumped. So, a shift to the right can contribute, but a shift to the left cannot result in a negative width. This is also why vertical typesetting, where height and depth are swapped with width, goes wrong: we somehow need to map two properties onto one and conceptually \TeX is really set up for horizontal typesetting. (And it's why I decided to just remove it from the engine.)

This is one of these cases where \TeX behaves as expected but it also means that there is some limitation to what can be manipulated. Setting the shift using one of the four commands has a direct consequence when a box gets packaged which happens immediately because the box is an argument to the foursome.

There is in traditional \TeX , probably for good reason, no way to set the shift of a box, if only because the effect would normally be none. But in $\text{Lua}\TeX$ we can cheat, and therefore, for educational purposed $\text{Con}\TeX\text{t}$ has implements some cheats.

We use this sample box:

```
\setbox\scratchbox\hbox\bgroup
  \middlegray\vrule width 20mm depth -.5mm height 10mm
  \hskip-20mm
  \darkgray \vrule width 20mm height -.5mm depth 5mm
\egroup
```

When we mess with the shift using the $\text{Con}\TeX\text{t}$ `\shiftbox` helper, we see no immediate effect. We only get the shift applied when we use another helper, `\hpackbox`.

```
\hbox\bgroup
  \showstruts \strut
  \quad \copy\scratchbox
```

```

\quad \shiftbox\scratchbox -20mm \copy\scratchbox
\quad \hpackbox\scratchbox      \box \scratchbox
\quad \strut

```

`\egroup`



When instead we use `\vpackbox` we get a different result. This time we move left.

`\hbox\bgroup`

```

\showstruts \strut
\quad
\quad \shiftbox\scratchbox -10mm \copy\scratchbox
\quad \vpackbox\scratchbox      \copy\scratchbox
\quad \strut

```

`\egroup`



The shift is set via Lua and the repackaging is also done in Lua, using the low level `hpack` and `vpack` helpers and these just happen to look at the shift when doing their job. At the \TeX end this never happens.

This long exploration of shifting serves a purpose: it demonstrates that there is not that much direct control over boxes apart from their three dimensions. However this was never a real problem as one can just wrap a box in another one and use kerns to move the embedded box around. But nevertheless I decided to see if the engine can be a bit more helpful, if only because all that extra wrapping gives some overhead and complications when we want to manipulate boxes. And of course it is also a nice playground.

We start with changing the direction. Changing this property doesn't require repackaging because directions are not really dealt with in the frontend. When a box is converted to (for instance pdf) the reversion happens.

```

\setbox\scratchbox\hbox{whatever}
\the\boxdirection\scratchbox: \copy\scratchbox \crlf
\boxdirection\scratchbox 1
\the\boxdirection\scratchbox: \copy\scratchbox

```

0: whatever
1: revetahw

Another property that can be queried and set is an attribute. In order to get a private attribute we define one.

```

\newattribute\MyAt
\setbox\scratchbox\hbox attr \MyAt 123 {whatever}
[\the\boxattribute\scratchbox\MyAt]
\boxattribute\scratchbox\MyAt 456
[\the\boxattribute\scratchbox\MyAt]
[\ifnum\boxattribute\scratchbox\MyAt>400 okay\fi]

```

[123] [456] [okay]

The sum of the height and depth is available too. Because for practical reasons setting that property is also needed then, the choice was made to distribute the value equally over height and depth.

```

\setbox\scratchbox\hbox {height and depth}
[\the\ht\scratchbox]
[\the\dp\scratchbox]
[\the\boxtotal\scratchbox]
\boxtotal\scratchbox=20pt
[\the\ht\scratchbox]
[\the\dp\scratchbox]
[\the\boxtotal\scratchbox]

```

[8.35742pt] [2.44385pt] [10.80127pt] [10.0pt] [10.0pt] [20.0pt]

We've now arrived to a set of properties that relate to each other. They are a bit complex and given the number of possibilities one might need to revert to some trial and error: orientations and offsets. As with the dimensions, directions and attributes, they are passed as box specification. We start with the orientation.

```

\hbox \bgroup \showboxes
  \hbox orientation 0 {right}

```

```

\quad \hbox orientation 1 {up}
\quad \hbox orientation 2 {left}
\quad \hbox orientation 3 {down}

```

\egroup



When the orientation is set, you can also set an offset. Where shifting around a box can have consequences for the dimensions, an offset is virtual. It gets effective in the backend, when the contents is converted to some output format.

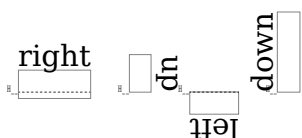
\hbox \bgroup \showboxes

```

\hbox orientation 0 yoffset 10pt {right}
\quad \hbox orientation 1 xoffset 10pt {up}
\quad \hbox orientation 2 yoffset -10pt {left}
\quad \hbox orientation 3 xoffset -10pt {down}

```

\egroup



The reason that offsets are related to orientation is that we need to know in what direction the offsets have to be applied and this binding forces the user to think about it. You can also set the offsets using commands.

```
\setbox\scratchbox\hbox{whatever}%
```

```

1 \copy\scratchbox
2 \boxorientation\scratchbox 2 \copy\scratchbox
3 \boxxoffset \scratchbox -15pt \copy\scratchbox
4 \boxyoffset \scratchbox -15pt \copy\scratchbox
5

```

```
1 whatever2 whatever3 whatever4 5
```

```
\setbox\scratchboxone\hbox{whatever}%
```

```
\setbox\scratchboxtwo\hbox{whatever}%
```

```

1 \boxxoffset \scratchboxone -15pt \copy\scratchboxone
2 \boxyoffset \scratchboxone -15pt \copy\scratchboxone
3 \boxxoffset \scratchboxone -15pt \copy\scratchboxone
4 \boxyoffset \scratchboxone -15pt \copy\scratchboxone

```



```

5 \boxxmove \scratchboxtwo -15pt \copy\scratchboxtwo
6 \boxymove \scratchboxtwo -15pt \copy\scratchboxtwo
7 \boxxmove \scratchboxtwo -15pt \copy\scratchboxtwo
8 \boxymove \scratchboxtwo -15pt \copy\scratchboxtwo

```

whatever	2	3	4	whatever6	7	8
	whatever	whatever	whatever	whatever	whatever	whatever

The move commands are provided as convenience and contrary to the offsets they do adapt the dimensions. Internally, with the box, we register the orientation and the offsets and when you apply these commands multiple times the current values get overwritten. But ... because an orientation can be more complex you might not get the effects you expect when the options we discuss next are used. The reason is that we store the original dimensions too and these come into play when these other options are used: anchoring. So, normally you will apply an orientation and offsets once only.

The orientation specifier is actually a three byte number that best can be seen hexadecimal (although we stay within the decimal domain). There are three components: x-anchoring, y-anchoring and orientation:

`0x<X><Y><O>`

or in \TeX speak:

`"<X><Y><O>`

The landscape and seascape variants both sit on top of the baseline while the flipped variant has its depth swapped with the height. Although this would be enough a bit more control is possible.

The vertical options of the horizontal variants anchor on the baseline, lower corner, upper corner or center.

```

\ruledhbox orientation "002 {\TEX} and
\ruledhbox orientation "012 {\TEX} and
\ruledhbox orientation "022 {\TEX} and
\ruledhbox orientation "032 {\TEX}

```

\TeX	and	\TeX	and	\TeX	and	\TeX
--------	-----	--------	-----	--------	-----	--------

The horizontal options of the horizontal variants anchor in the center, left, right, halfway left and halfway right.

```

\ruledhbox orientation "002 {\TEX} and
\ruledhbox orientation "102 {\TEX} and
\ruledhbox orientation "202 {\TEX} and
\ruledhbox orientation "302 {\TEX} and
\ruledhbox orientation "402 {\TEX}

```

The orientation has consequences for the dimensions so they are dealt with in the expected way in constructing lines, paragraphs and pages, but the anchoring is virtual, like the offsets. There are two extra variants for orientation zero: on top of baseline or below, with dimensions taken into account.

```

\ruledhbox orientation "000 {\TEX} and
\ruledhbox orientation "004 {\TEX} and
\ruledhbox orientation "005 {\TEX}

```

The anchoring can look somewhat confusing but you need to keep in mind that it is normally only used in very controlled circumstances and not in running text. Wrapped in macros users don't see the details. We're talking boxes here, so for instance:

```

test\quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "012 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "022 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "032 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "042 \bgroup\strut test\egroup test%

```

`\egroup`

`\quad test`

test	test	test	test	test	test	test	test
	<code>\sø</code>	<code>\sø</code>	<code>\sø</code>	<code>\sø</code>	<code>\sø</code>	<code>\sø</code>	<code>\sø</code>
	test	test	test	test	test	test	test

3.6 Colofon

Author Hans Hagen
 ConT_EXt 2021.09.06 11:47
 LuaMetaT_EX 2.0923
 Support www.pragma-ade.com
 contextgarden.net

4 Expansion

low level

TEX

expansion

Contents

4.1	Preamble	50
4.2	T _E X primitives	50
4.3	ε-T _E X primitives	55
4.4	LuaT _E X primitives	57
4.5	LuaMetaT _E X primitives	58
4.6	Dirty tricks	68

4.1 Preamble

This short manual demonstrates a couple of properties of the macro language. It is not an in-depth philosophical expose about macro languages, tokens, expansion and such that some T_EXies like. I prefer to stick to the practical aspects. Occasionally it will be technical but you can just skip those paragraphs (or later return to them) when you can't follow the explanation. It's often not that relevant. I won't talk in terms of mouth, stomach and gut the way the T_EXbook does and although there is no way to avoid the word 'token' I will do my best to not complicate matters by too much token speak. Examples show best what we mean.

4.2 T_EX primitives

The T_EX language provides quite some commands and those built in are called primitives. User defined commands are called macros. A macro is a shortcut to a list of primitives and/or macro calls. All can be mixed with characters that are to be typeset somehow.

```
\def\MyMacro{b}
```

```
a\MyMacro c
```

When T_EX reads this input the a gets turned into a glyph node with a reference to the current font set and the character a. Then the parser sees a macro call, and it will enter another input level where it expands this macro. In this case it sees just an b and it will give this the same treatment as the a. The macro ends, the input level decrements and the c gets its treatment.

Before we move on to more examples and differences between engines, it is good to stress that `\MyMacro` is not a primitive command: we made our command here. The b actually can be seen as a sort of primitive because in this macro it gets stored as so

called token with a primitive property. That primitive property can later on be used to determine what to do. More explicit examples of primitives are `\hbox`, `\advance` and `\relax`. It will be clear that ConT_EXt extends the repertoire of primitive commands with a lot of macro commands. When we typeset a source using module `m-scite` the primitives come out dark blue.

The amount of primitives differs per engine. It all starts with T_EX as written by Don Knuth. Later ε -T_EX added some more primitives and these became official extensions adopted by other variants of T_EX. The pdfT_EX engine added quite some and as follow up on that LuaT_EX added more but didn't add all of pdfT_EX. A few new primitives came from Omega (Aleph). The LuaMetaT_EX engine drops a set of primitives that comes with LuaT_EX and adds plenty new ones. The nature of this engine (no backend and less frontend) makes that we need to implement some primitives as macros. But the basic set is what good old T_EX comes with.

Internally these so called primitives are grouped in categories that relate to their nature. They can be directly expanded (a way of saying that they get immediately interpreted) or delayed (maybe stored for later usage). They can involve definitions, calculations, setting properties and values or they can result in some typesetting. This is what makes T_EX confusing to new users: it is a macro programming language, an interpreter but at the same time an executor of typesetting instructions.

A group of primitives is internally identified as a command (they have a `cmd` code) and the sub commands are flagged by their `chr` code. This sounds confusing but just thing of the fact that most of what we input are characters and therefore they make up most sub commands. For instance the 'letter `cmd`' is used for characters that are seen as letters that can be used in the name of user commands, can be typeset, are valid for hyphenation etc. The letter related `cmd` can have many `chr` codes (all of Unicode). I'd like to remark that the grouping is to a large extend functional, so sometimes primitives that you expect to be similar in nature are in different groups. This has to do with the fact that T_EX needs to be able to determine efficiently if a primitive is operating (or forbidden) in horizontal, vertical and/or math mode.

There are more than 150 internal `cmd` groups. if we forget about the mentioned character related ones, some, have only a few sub commands (`chr`) and others many more (just consider all the OpenType math spacing related parameters). A handful of these commands deal with what we call macros: user defined combinations of primitives and other macros, consider them little programs. The `\MyMacro` example above is an example. There are differences between engines. In standard T_EX there are `\outer` and `\long` commands, and most engines have these. However, in LuaMetaT_EX the later to be discussed `\protected` macros have their own specific 'call `cmd`'. Users don't need to bother about this.

So, when from now on we talk about primitives, we mean the built in, hard coded commands, and when we talk about macros we mean user commands. Although internally there are less cmd categories than primitives, from the perspective of the user they are all unique. Users won't consult the source anyway but when they do they are warned. Also, when in LuaMetaTeX you use the low level interfacing to TeX you have to figure out these subtle aspects because there this grouping does matter.

Before we continue I want to make clear that expansion (as discussed in this document) can refer to a macro being expanded (read: its meaning gets injected into the input, so the engine kind of sidetracks from what it was doing) but also to direct consequences of running into a primitive. However, users only need to consider expansion in the perspective of macros. If a user has `\advance` in the input it immediately gets done. But when it's part of a macro definition it only is executed when the macro expands. A good check in (traditional) TeX is to compare what happens in `\def` and `\edef` which is why we will use these two in the upcoming examples. You put something in a macro and then check what `\meaning` or `\show` reports.

Now back to user defined macros. A macro can contain references to macros so in practice the input can go several levels up and some applications push back a lot so this is why your TeX input stack can be configured to be huge.

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

When `\MyMacroB` is defined, its body gets three so called tokens: the character token 1 with property 'other', a token that is a reference to the macro `\MyMacroB`, and a character token 2, also with property 'other'. The meaning of `\MyMacroA` is five tokens: a reference to a space token, then three character tokens with property 'letter', and finally a space token.

```
\def \MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

In the second definition an `\edef` is used, where the e indicates expansion. This time the meaning gets expanded immediately. So we get effectively the same as in:

```
\def\MyMacroB{1 and 2}
```

Characters are easy: they just expand to themselves or trigger adding a glyph node,

but not all primitives expand to their meaning or effect.

```
\def\MyMacroA{\scratchcounter = 1 }
\def\MyMacroB{\advance\scratchcounter by 1}
\def\MyMacroC{\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 4

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter
```

Let's assume that `\scratchcounter` is zero to start with and use `\edef`'s:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 0

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:0
```

So, this time the third macro has its meaning frozen, but we can prevent this by applying a `\noexpand` when we do this:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\noexpand\the\scratchcounter}
```

```

\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC

```

```
a b c d 4
```

```

macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter

```

Of course this is a rather useless example but it serves its purpose: you'd better be aware what gets expanded immediately in an `\edef`. In most cases you only need to worry about `\the` and embedded macros (and then of course their meanings).

You can also store tokens in a so-called token register. Here we use a predefined scratch register:

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks {\MyMacroA}

```

The content of `\scratchtoks` is: “`\MyMacroA`”, so no expansion has happened here.

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroA}

```

Now the content of `\scratchtoks` is: “`and`”, so this time expansion has happened.

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}

```

Indeed the macro gets expanded but only one level: “`1\MyMacroA 2`”. Compare this with:

```

\def\MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}

```

The trick is to expand in two steps with an intermediate `\edef`: “`1 and 2`”. Later we will

see that other engines provide some more expansion tricks. The only way to get some grip on expansion is to just play with it.

The `\expandafter` primitive expands the token (which can be a macro) standing after the next next one and then injects its meaning into the stream. So:

```
\expandafter \MyMacroA \MyMacroB
```

works okay. In a normal document you will never need this kind of hackery: it only happens in a bit more complex macros. Here is an example:

```
\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\egroup
\the\scratchcounter

\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\expandafter
\egroup
\the\scratchcounter
```

The first one gives 1, while the second gives 2.

4.3 ε - $\text{T}_{\text{E}}\text{X}$ primitives

In this engine a couple of extensions were added and later on pdf $\text{T}_{\text{E}}\text{X}$ added some more. We only discuss a few that relate to expansion. There is however a pitfall here. Before ε - $\text{T}_{\text{E}}\text{X}$ showed up, Con $\text{T}_{\text{E}}\text{X}$ t already had a few mechanism that also related to expansion and it used some names for macros that clash with those in ε - $\text{T}_{\text{E}}\text{X}$. This is why we will use the `\normal` prefix here to indicate the primitive.¹

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\edef\MyMacroABC{\MyMacroA\MyMacroB\MyMacroC}
```

These macros have the following meanings:

¹ In the meantime we no longer have a low level `\protected` macro so one can use the primitive

```
macro:a
macro:b
protected macro:c
macro:ab\MyMacroC
```

In ConT_EXt you will use the `\unexpanded` prefix instead, because that one did something similar in older versions of ConT_EXt. As we were early adopters of ε -T_EX, this later became a synonym to the ε -T_EX primitive.

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded{\scratchtoks{\MyMacroA\MyMacroB\MyMacroC}}
```

Here the wrapper around the token register assignment will expand the three macros, unless they are protected, so its content becomes “ab\MyMacroC”. This saves either a lot of more complex `\expandafter` usage or the need to use an intermediate `\edef`. In ConT_EXt the `\expanded` macro does something simpler but it doesn't expand the first token as this is meant as a wrapper around a command, like:

```
\expanded{\chapter{...}} % a ConTeXt command
```

where we do want to expand the title but not the `\chapter` command (not that this would happen actually because `\chapter` is a protected command.)

The counterpart of `\normalexpanded` is `\normalunexpanded`, as in:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded {\scratchtoks
  {\MyMacroA\normalunexpanded {\MyMacroB}\MyMacroC}}
```

The register now holds “a\MyMacroB \MyMacroC”: three tokens, one character token and two macro references.

Tokens can represent characters, primitives, macros or be special entities like starting math mode, beginning a group, assigning a dimension to a register, etc. Although you can never really get back to the original input, you can come pretty close, with:

```
\detokenize{this can $ be anything \bgroup}
```

This (when typeset monospaced) is: this can \$ be anything \bgroup. The detok-

enizer is like `\string` applied to each token in its argument. Compare this to:

```
\normalexpanded {
  \normaldetokenize{10pt}
}
```

We get four tokens: 10pt.

```
\normalexpanded {
  \string 1\string 0\string p\string t
}
```

So that was the same operation: 10pt, but in both cases there is a subtle thing going on: characters have a catcode which distinguishes them. The parser needs to know what makes up a command name and normally that's only letters. The next snippet shows these catcodes:

```
\normalexpanded {
  \noexpand\the\catcode`\string 1 \noexpand\enspace
  \noexpand\the\catcode`\string 0 \noexpand\enspace
  \noexpand\the\catcode`\string p \noexpand\enspace
  \noexpand\the\catcode`\string t \noexpand
}
```

The result is “12 12 11 11”: two characters are marked as ‘letter’ and two fall in the ‘other’ category.

4.4 Lua_TE_X primitives

This engine adds a little to the expansion repertoire. First of all it offers a way to extend token lists registers:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{b}
\scratchtoks{\MyMacroA\MyMacroB}
```

The result is: “\MyMacroA \MyMacroB”.

```
\toksapp\scratchtoks{\MyMacroA\MyMacroB}
```

We're now at: “\MyMacroA \MyMacroB \MyMacroA \MyMacroB \MyMacroA \MyMacroB”.

```
\etoksapp\scratchtoks{\MyMacroA\space\MyMacroB\space\MyMacroC}
```

The register has this content: “\MyMacroA \MyMacroB \MyMacroA \MyMacroB a b \MyMacroC a b \MyMacroC”, so the additional context got expanded in the process, except of course the protected macro \MyMacroC.

There is a bunch of these combiners: \toksapp and \tokspre for local appending and prepending, with global companions: \gtoksapp and \gtokspre, as well as expanding variant: \etoksapp, \etokspre, \xtoksapp and \xtokspre.

These are not beforehand more efficient than using intermediate expanded macros or token lists, simply because in the process $\text{T}_{\text{E}}\text{X}$ has to create token lists too, but sometimes they're just more convenient to use. In $\text{ConT}_{\text{E}}\text{Xt}$ we actually do benefit from these.

4.5 LuaMeta $\text{T}_{\text{E}}\text{X}$ primitives

We already saw that macro's can be defined protected which means that

```
\def\TestA{A}
\protected \def\TestB{B}
\edef\TestC{\TestA\TestB}
```

gives this:

```
\TestC : A\TestB
```

One way to get \TestB expanded it to prefix it with \expand:

```
\def\TestA{A}
\protected \def\TestB{B}
\edef\TestC{\TestA\TestB}
\edef\TestD{\TestA\expand\TestB}
```

We now get:

```
\TestC : A\TestB
\TestD : AB
```

There are however cases where one wishes this to happen automatically, but that will also make protected macros expand which will create havoc, like switching fonts.

```
\def\TestA{A}
\protected \def\TestB{B}
```

```

\semiprotected \def\TestC{C}
      \edef\TestD{\TestA\TestB\TestC}
      \edef\TestE{\normalexpanded{\TestA\TestB\TestC}}
      \edef\TestF{\semiexpanded  {\TestA\TestB\TestC}}

```

This time `\TestC` loses its protection:

```

\TestA : A
\TestB : B
\TestC : C
\TestD : A\TestB \TestC
\TestE : A\TestB \TestC
\TestF : A\TestB C

```

Actually adding `\fullyexpanded` would be trivial but it makes not much sense to add the overhead (at least not now). This feature is experimental anyway so it might go away when I see no real advantage from it.

When you store something in a macro or token register you always need to keep an eye on category codes. A dollar in the input is normally treated as math shift, a hash indicates a macro parameter or preamble entry. Characters like ‘A’ are letters but ‘[’ and ‘]’ are tagged as ‘other’. The \TeX scanner acts according to these codes. If you ever find yourself in a situation that changing catcodes is no option or cumbersome, you can do this:

```

\edef\Test0A{\expandtoken\othercatcode `A}
\edef\TestLA{\expandtoken\lettercatcode `A}

```

In both cases the meaning is A but in the first case it's not a letter but a character flagged as ‘other’.

A whole new category of commands has to do with so called local control. When \TeX scans and interprets the input, a process takes place that is called tokenizing: (sequences of) characters get a symbolic representation and travel through the system as tokens. Often they immediately get interpreted and are then discarded. But when for instance you define a macro they end up as a linked list of tokens in the macro body. We already saw that expansion plays a role. In most cases, unless \TeX is collecting tokens, the main action is dealt with in the so-called main loop. Something gets picked up from the input but can also be pushed back, for instance because of some lookahead that didn't result in an action. Quite some time is spent in pushing and popping from the so-called input stack.

When we are in Lua, we can pipe back into the engine but all is collected till we're

back in \TeX where the collected result is pushed into the input. Because \TeX is a mix of programming and action there basically is only that main loop. There is no real way to start a sub run in Lua and do all kind of things independent of the current one. This makes sense when you consider the mix: it would get too confusing.

However, in $\text{Lua}\TeX$ and even better in $\text{LuaMeta}\TeX$, we can enter a sort of local state and this is called ‘local control’. When we are in local control a new main loop is entered and the current state is temporarily forgotten: we can for instance expand where one level up expansion was not done. It sounds complicated and indeed it is complicated so examples have to clarify it.

```
1 \setbox0\hbox to 10pt{2} \count0=3 \the\count0 \multiply\count0 by 4
```

This snippet of code is not that useful but illustrates what we're dealing with:

- The 1 gets typeset. So, characters like that are seen as text.
- The `\setbox` primitive triggers picking up a register number, then goes on scanning for a box specification and that itself will typeset a sequence of whatever until the group ends.
- The `count` primitive triggers scanning for a register number (or reference) and then scans for a number; the equal sign is optional.
- The `the` primitive injects some value into the current input stream and it does so by entering a new input level.
- The `multiply` primitive picks up a register specification and multiplies that by the next scanned number. The `by` is optional.

We now look at this snippet again but with an expansion context:

```
\def \TestA{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

These two macros have a slightly different body. Make sure you see the difference before reading on.

control sequence: TestA

500339	12	49	other char	1	U+00031	
433523	10	32	spacer			
500112	116	0	set box			setbox

502316	12	48	other char	0	U+00030	
386851	30	10	make box			hbox
499136	1	123	left brace			
386869	12	50	other char	2	U+00032	
31060	2	125	right brace			
503004	10	32	spacer			
503416	109	0	register			count
502511	12	48	other char	0	U+00030	
31100	12	61	other char	=	U+0003D	
503195	12	51	other char	3	U+00033	
502594	10	32	spacer			
503076	129	0	the			the
386994	109	0	register			count
503129	12	48	other char	0	U+00030	

control sequence: TestB

500308	12	49	other char	1	U+00031	
502250	10	32	spacer			
502462	116	0	set box			setbox
386985	12	48	other char	0	U+00030	
502805	30	10	make box			hbox
503064	1	123	left brace			
386817	12	50	other char	2	U+00032	
503180	2	125	right brace			
500346	10	32	spacer			
502411	109	0	register			count
502735	12	48	other char	0	U+00030	
500189	12	61	other char	=	U+0003D	
500138	12	51	other char	3	U+00033	
433528	10	32	spacer			
502793	12	49	other char	1	U+00031	

We now introduce a new primitive `\localcontrolled`:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

```
\edef\TestC{1 \setbox0\hbox{2} \localcontrolled{\count0=3} \the\count0}
```

Again, watch the subtle differences:

control sequence: TestB

215000	12	49	other char	1	U+00031	
502547	10	32	spacer			
502382	116	0	set box			setbox
112637	12	48	other char	0	U+00030	
499145	30	10	make box			hbox
340837	1	123	left brace			
352318	12	50	other char	2	U+00032	
502552	2	125	right brace			
502367	10	32	spacer			
502824	109	0	register			count
31041	12	48	other char	0	U+00030	
500296	12	61	other char	=	U+0003D	
502468	12	51	other char	3	U+00033	
387029	10	32	spacer			
112641	12	49	other char	1	U+00031	

control sequence: TestC

503158	12	49	other char	1	U+00031	
502598	10	32	spacer			
503164	116	0	set box			setbox
500297	12	48	other char	0	U+00030	
500064	30	10	make box			hbox
499099	1	123	left brace			
31116	12	50	other char	2	U+00032	
499108	2	125	right brace			
366852	10	32	spacer			
502334	10	32	spacer			
502706	12	51	other char	3	U+00033	

Another example:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

```
\edef\TestD{\localcontrolled{1 \setbox0\hbox{2} \count0=3 \the\count0}}
```

1 3 ← Watch how the results end up here!

control sequence: TestB

500240	12	49	other char	1	U+00031	
386765	10	32	spacer			
214990	116	0	set box			setbox
499255	12	48	other char	0	U+00030	
194183	30	10	make box			hbox
500293	1	123	left brace			
502620	12	50	other char	2	U+00032	
387017	2	125	right brace			
503449	10	32	spacer			
502646	109	0	register			count
201286	12	48	other char	0	U+00030	
502477	12	61	other char	=	U+0003D	
503187	12	51	other char	3	U+00033	
502969	10	32	spacer			
31118	12	51	other char	3	U+00033	

control sequence: TestD

<no tokens>

We can use this mechanism to define so called fully expandable macros:

```
\def\WidthOf#1%
  {\beginlocalcontrol
   \setbox0\hbox{#1}%
   \endlocalcontrol
   \wd0 }
```

```
\scratchdimen\WidthOf{The Rite Of Spring}
```

```
\the\scratchdimen
```

104.72021pt

When you want to add some grouping, it quickly can become less pretty:

```
\def\WidthOf#1%
  {\dimexpr
   \beginlocalcontrol
   \begingroup
   \setbox0\hbox{#1}%
   \expandafter
```

```

    \endgroup
    \expandafter
    \endlocalcontrol
    \the\wd0
\relax}

```

```
\scratchdimen\WidthOf{The Rite Of Spring}
```

```
\the\scratchdimen
```

```
104.72021pt
```

A single token alternative is available too and its usage is like this:

```

\def\TestA{\scratchcounter=100 }
\edef\TestB{\localcontrol\TestA \the\scratchcounter}
\edef\TestC{\localcontrolled{\TestA} \the\scratchcounter}

```

The content of `\TestB` is '100' and of course the `\TestC` macro gives ' 100'.

We now move to the Lua end. Right from the start the way to get something into \TeX from Lua has been the print functions. But we can also go local (immediate). There are several methods:

- via a set token register
- via a defined macro
- via a string

Among the things to keep in mind are catcodes, scope and expansion (especially in when the result itself ends up in macros). We start with an example where we go via a token register:

```

\toks0={\setbox0\hbox{The Rite Of Spring}}
\toks2={\setbox0\hbox{The Rite Of Spring!}}

```

```
\startluacode
```

```
tex.runlocal(0) context("[1: %p]",tex.box[0].width)
```

```
tex.runlocal(2) context("[2: %p]",tex.box[0].width)
```

```
\stopluacode
```

```
[1: 104.72021pt][2: 109.14062pt]
```

We can also use a macro:

```
\def\TestA{\setbox0\hbox{The Rite Of Spring}}
\def\TestB{\setbox0\hbox{The Rite Of Spring!}}
```

```
\startluacode
```

```
tex.runlocal("TestA") context("[3: %p]",tex.box[0].width)
```

```
tex.runlocal("TestB") context("[4: %p]",tex.box[0].width)
```

```
\stopluacode
```

```
[3: 104.72021pt][4: 109.14062pt]
```

A third variant is more direct and uses a (Lua) string:

```
\startluacode
```

```
tex.runstring([[ \setbox0\hbox{The Rite Of Spring} ]])
```

```
context("[5: %p]",tex.box[0].width)
```

```
tex.runstring([[ \setbox0\hbox{The Rite Of Spring!} ]])
```

```
context("[6: %p]",tex.box[0].width)
```

```
\stopluacode
```

```
[5: 104.72021pt][6: 109.14062pt]
```

A bit more high level:

```
context.runstring([[ \setbox0\hbox{(Here \bf 1.2345)} ]])
```

```
context.runstring([[ \setbox0\hbox{(Here \bf %.3f)} ]],1.2345)
```

Before we had `runstring` this was the way to do it when staying in Lua was needed:

```
\startluacode
```

```
token.setmacro("TestX",[[ \setbox0\hbox{The Rite Of Spring} ]])
```

```
tex.runlocal("TestX")
```

```
context("[7: %p]",tex.box[0].width)
```

```
\stopluacode
```

```
[7: 104.72021pt]
```

```
\startluacode
```

```
tex.scantoks(0,tex.ctxcatcodes,[[ \setbox0\hbox{The Rite Of Spring!} ]])
```

```
tex.runlocal(0)
```

```
context("[8: %p]",tex.box[0].width)
```

```
\stopluacode
```

[8: 109.14062pt]

The order of flushing matters because as soon as something is not stored in a token list or macro body, \TeX will typeset it. And as said, a lot of this relates to pushing stuff into the input which is stacked. Compare:

```
\startluacode
context("[HERE 1]")
context("[HERE 2]")
\stopluacode
```

[HERE 1][HERE 2]

with this:

```
\startluacode
tex.pushlocal() context("[HERE 1]") tex.poplocal()
tex.pushlocal() context("[HERE 2]") tex.poplocal()
\stopluacode
```

[HERE 1][HERE 2]

You can expand a macro at the Lua end with `token.expandmacro` which has a peculiar interface. The first argument has to be a string (the name of a macro) or a userdata (a valid macro token). This macro can be fed with parameters by passing more arguments:

string	serialized to tokens
true	wrap the next string in curly braces
table	each entry will become an argument wrapped in braces
token	inject the token directly
number	change control to the given catcode table

There are more scanner related primitives, like the ε - \TeX primitive `\detokenize`:

```
[\detokenize {test \relax}]
```

This gives: [test \relax] . In `LuaMeta \TeX` we also have complementary primitive(s):

```
[\tokenized catcodetable \vrbcatcodes {test {\bf test} test}]
[\tokenized {test {\bf test} test}]
[\retokenized \vrbcatcodes {test {\bf test} test}]
```

The `\tokenized` takes an optional keyword and the examples above give: `[test {\bf test} test]` `[test test test]` `[test {\bf test} test]` . The Lua_{TeX} primitive `\scantextokens` which is a variant of ϵ -TeX's `\scantokens` operates under the current catcode regime (the last one honors `\everyeof`). The difference with `\tokenized` is that this one first serializes the given token list (just like `\detokenize`).²

With `\retokenized` the catcode table index is mandatory (it saves a bit of scanning and is easier on intermixed `\expandafter` usage. There often are several ways to accomplish the same:

```
\def\MyTitle{test {\bf test} test}
\detokenize           \expandafter{\MyTitle}: 0.46\crlf
\meaningless         \MyTitle : 0.47\crlf
\retokenized         \notcatcodes{\MyTitle}: 0.87\crlf
\tokenized catcodetable \notcatcodes{\MyTitle}: 0.93\crlf
```

```
test {\bf test} test: 0.46
test {\bf test} test: 0.47
test {\bf test} test: 0.87
test {\bf test} test: 0.93
```

Here the numbers show the relative performance of these methods. The `\detokenize` and `\meaningless` win because they already know that a verbose serialization is needed. The last two first serialize and then reinterpret the resulting token list using the given catcode regime. The last one is slowest because it has to scan the keyword.

There is however a pitfall here:

```
\def\MyText {test}
\def\MyTitle{test \MyText\space test}
\detokenize           \expandafter{\MyTitle}\crlf
\meaningless         \MyTitle \crlf
\retokenized         \notcatcodes{\MyTitle}\crlf
\tokenized catcodetable \notcatcodes{\MyTitle}\crlf
```

The outcome is different now because we have an expandable embedded macro call. The fact that we expand in the last two primitives is also the reason why they are 'slower'.

```
test \MyText \space test
test \MyText \space test
```

² The `\scan *tokens` primitives now share the same helpers as Lua, but they should behave the same as in Lua_{TeX}.

```
test test test
test test test
```

To complete this picture, we show a variant than combines much of what has been introduced in this section:

```
\semiprotected\def\MyTextA {test}
\def\MyTextB {test}
\def\MyTitle{test \MyTextA\space \MyTextB\space test}
\detokenize          \expandafter{\MyTitle}\crlf
\meaningless        \MyTitle \crlf
\retokenized         \notcatcodes{\MyTitle}\crlf
\retokenized         \notcatcodes{\semiexpanded{\MyTitle}}\crlf
\tokenized   catcodetable \notcatcodes{\MyTitle}\crlf
\tokenized   catcodetable \notcatcodes{\semiexpanded{\MyTitle}}
```

This time compare the last four lines:

```
test \MyTextA \space \MyTextB \space test
test \MyTextA \space \MyTextB \space test
test \MyTextA test test
test test test test
test \MyTextA test test
test test test test
```

Of course the question remains to what extend we need this and eventually will apply in ConT_EXt. The `\detokenize` is used already. History shows that eventually there is a use for everything and given the way LuaMetaT_EX is structured it was not that hard to provide the alternatives without sacrificing performance or bloating the source.

4.6 Dirty tricks

When I was updating this manual Hans vd Meer and I had some discussions about expansion and tokenization related issues when combining of xml processing with T_EX macros where he did some manipulations in Lua. In these mixed cases you can run into catcode related problems because in xml you want for instance a `#` to be a hash mark (other character) and not an parameter identifier. Normally this is handled well in ConT_EXt but of course there are complex cases where you need to adapt.

Say that you want to compare two strings (officially we should say token lists) with mixed catcodes. Let's also assume that you want to use the normal `\if` construct (which was

part of the discussion). We start with defining a test set. The reason that we present this example here is that we use commands discussed in previous sections:

```

\def\abc{abc}
\semiprotected \def\xyz{xyz}
\edef\pqr{\expandtoken\notcatcodes`p%
\expandtoken\notcatcodes`q%
\expandtoken\notcatcodes`r}

1: \ifcondition\similartokens{abc} {def}YES\else NOP\fi (NOP) \quad
2: \ifcondition\similartokens{abc}{\abc}YES\else NOP\fi (YES)

3: \ifcondition\similartokens{xyz} {pqr}YES\else NOP\fi (NOP) \quad
4: \ifcondition\similartokens{xyz}{\xyz}YES\else NOP\fi (YES)

5: \ifcondition\similartokens{pqr} {pqr}YES\else NOP\fi (YES) \quad
6: \ifcondition\similartokens{pqr}{\pqr}YES\else NOP\fi (YES)

```

So, we have a mix of expandable and semi expandable macros, and also a mix of cat-codes. A naive approach would be:

```

\permanent\protected\def\similartokens#1#2%
{\iftok{#1}{#2}}

```

but that will fail on some cases:

```

1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: NOP(YES)

```

So how about:

```

\permanent\protected\def\similartokens#1#2%
{\iftok{\detokenize{#1}}{\detokenize{#2}}}

```

That one is even worse:

```

1: NOP(NOP)   2: NOP(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: NOP(YES)

```

We need to expand so we end up with this:

```

\permanent\protected\def\similartokens#1#2%

```

```
{\normalexpanded{\noexpand\iftok
  {\noexpand\detokenize{#1}}
  {\noexpand\detokenize{#2}}}}
```

Better:

```
1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: YES(YES)
```

But that will still not deal with the mildly protected macro so in the end we have:

```
\permanent\protected\def\similartokens#1#2%
  {\semiexpanded{\noexpand\iftok
    {\noexpand\detokenize{#1}}
    {\noexpand\detokenize{#2}}}}
```

Now we're good:

```
1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: YES(YES)
5: YES(YES)   6: YES(YES)
```

Finally we wrap this one in the usual `\doifelse...` macro:

```
\permanent\protected\def\doifelsesimilartokens#1#2%
  {\ifcondition\similartokens{#1}{#2}%
   \expandafter\firstoftwoarguments
   \else
   \expandafter\secondoftwoarguments
   \fi}
```

so that we can do:

```
\doifelsesimilartokens{pqr}{\pqr}{YES}{NOP}
```

A companion macro of this is `\wipetoken` but for that one you need to look into the source.

4.6 Colofon

Author Hans Hagen
ConT_EXt 2021.09.06 11:47
LuaMetaT_EX 2.0923
Support www.pragma-ade.com
 contextgarden.net

5 Registers

low level

TEX

registers

Contents

5.1	Preamble	73
5.2	$\text{T}_{\text{E}}\text{X}$ primitives	73
5.3	$\varepsilon\text{-T}_{\text{E}}\text{X}$ primitives	76
5.4	Lua $\text{T}_{\text{E}}\text{X}$ primitives	76
5.5	LuaMeta $\text{T}_{\text{E}}\text{X}$ primitives	77

5.1 Preamble

Registers are sets of variables that are accessed by index and as such resemble registers in a processing unit. You can store a quantity in a register, retrieve it, and also manipulate it.

There is hardly any need to use them in Con $\text{T}_{\text{E}}\text{X}$ t so we keep it simple.

5.2 $\text{T}_{\text{E}}\text{X}$ primitives

There are several categories:

- Integers (int): in order to be portable (at the time it surfaced) there are only integers and no floats. The only place where $\text{T}_{\text{E}}\text{X}$ uses floats internally is when glue gets effective which happens in the backend.
- Dimensions (dimen): internally these are just integers but when they are entered they are sliced into two parts so that we have a fractional part. The internal representation is called a scaled point.
- Glue (skip): these are dimensions with a few additional properties: stretch and shrink. Being a compound entity they are stored differently and thereby a bit less efficient than numbers and dimensions.
- Math glue (muskip): this is the same as glue but with a unit that adapts to the current math style properties. It's best to think about them as being relative measures.
- Token lists (toks): these contain a list of tokens coming from the input or coming from a place where they already have been converted.

The original $\text{T}_{\text{E}}\text{X}$ engine had 256 entries per set. The first ten of each set are normally reserved for scratch purposes: the even ones for local use, and the odd ones for global usage. On top of that macro packages can reserve some for its own use. It was quite

easy to reach the maximum but there were tricks around that. This limitation is no longer present in the variants in use today.

Let's set a few dimension registers:

```
\dimen 0 = 10 pt
\dimen2=10pt
\dimen4 10pt
\scratchdimen 10pt
```

We can serialize them with:

```
\the \dimen0
\number \dimen2
\meaning\dimen4
\meaning\scratchdimen
```

The results of these operations are:

```
10.0pt
655360
\dimen4
\dimen257
```

The last two is not really useful but it is what you see when tracing options are set. Here `\scratchdimen` is a shortcut for a register. This is *not* a macro but a defined register. The low level `\dimendef` is used for this but in a macro package you should not use that one but the higher level `\newdimen` macro that uses it.

```
\newdimen\MyDimenA
\def \MyDimenB{\dimen999}
\dimendef\MyDimenC998

\meaning\MyDimenA
\meaning\MyDimenB
\meaning\MyDimenC
```

Watch the difference:

```
\dimen754
macro:\dimen 999
\dimen998
```

The first definition uses a yet free register so you won't get a clash. The second one is just a shortcut using a macro and the third one too but again direct shortcut. Try to imagine how the second line gets interpreted:

```
\MyDimenA10pt \MyDimenA10.5pt
\MyDimenB10pt \MyDimenB10.5pt
\MyDimenC10pt \MyDimenC10.5pt
```

Also try to imagine what messing around with `\MyDimenC` will do when we also have defined a few hundred extra dimensions with `\newdimen`.

In the case of dimensions the `\number` primitive will make the register serialize as scaled points without unit `sp`.

Next we see some of the other registers being assigned:

```
\count 0 = 100
\skip 0 = 10pt plus 3pt minus 2pt
\skip 0 = 10pt plus 1fill
\muskip 0 = 10mu plus 3mu minus 2mu
\muskip 0 = 10mu minus 1 fil
\toks 0 = {hundred}
```

When a number is expected, you can use for instance this:

```
\scratchcounter\scratchcounterone
```

Here we use a few predefined scratch registers. You can also do this:

```
\scratchcounter\numexpr\scratchcounterone+\scratchcountertwo\relax
```

There are some quantities that also qualify as number:

```
\chardef\MyChar=123 % refers to character 123 (if present)
\scratchcounter\MyChar
```

In the past using `\chardef` was a way to get around the limited number of registers, but it still had (in traditional \TeX) a limitation: you could not go beyond 255. The `\mathchardef` could go higher as it also encodes a family number and class. This limitation has been lifted in $\text{Lua}\TeX$.

A character itself can also be interpreted as number, in which case it has to be prefixed with a reverse quote: ```, so:


```
\scratchcounter\numexpr`0+5\relax
\char\scratchcounter
```

produces “5” because the ``0` expands into the (ascii and utf8) slot 48 which represents the character zero. In this case the next makes more sense:

```
\char\numexpr`0+5\relax
```

If you want to know more about all these quantities, “*T_EX* By Topic” provides a good summary of what T_EX has to offer, and there is no need to repeat it here.

5.3 ε -T_EX primitives

Apart from the ability to use expressions, the contribution to registers that ε -T_EX brought was that suddenly we could use upto 65K of them, which is more than enough. The extra registers were not as efficient as the first 256 because they were stored in the hash table, but that was not really a problem. In Omega and later LuaT_EX regular arrays were used, at the cost of more memory which in the meantime has become cheap. As ConT_EXt moved to ε -T_EX rather early its users never had to worry about it.

5.4 LuaT_EX primitives

The LuaT_EX engine introduced attributes. These are numeric properties that are bound to the nodes that are the result of typesetting operations. They are basically like integer registers but when set their values get bound and when unset they are kind of invisible.

- Attribute (attribute): a numeric property that when set becomes part of the current attribute list that gets assigned to nodes.

Attributes can be used to communicate properties to Lua callbacks. There are several functions available for setting them and querying them.

```
\attribute999 = 123
```

Using attributes this way is dangerous (of course I can only speak for ConT_EXt) because an attribute value might trigger some action in a callback that gives unwanted side effects. For convenience ConT_EXt provides:

```
\newattribute\MyAttribute
```

Which currently defines `\MyAttribute` as integer 1026 and is meant to be used as:³

³ The low level `\attributedef` command is rather useless in the perspective of ConT_EXt.

```
\attribute\MyAttribute = 123
```

Just be aware that defining attributes can have an impact on performance. As you cannot access them at the T_EX end you seldom need them. If you do you can better use the proper more high level definers (not discussed here).

5.5 LuaMetaT_EX primitives

todo

5.5 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

6 Macros

low level

TEX

macros

Contents

6.1	Preamble	79
6.2	Definitions	79
6.3	Runaway arguments	89
6.4	Introspection	90
6.5	nesting	91
6.6	Prefixes	94

6.1 Preamble

This chapter overlaps with other chapters but brings together some extensions to the macro definition and expansion parts. As these mechanisms were stepwise extended, the other chapters describe intermediate steps in the development.

Now, in spite of the extensions discussed here the main idea is still that we have $\text{T}_{\text{E}}\text{X}$ act like before. We keep the charm of the macro language but these additions make for easier definitions, but (at least initially) none that could not be done before using more code.

6.2 Definitions

A macro definition normally looks like like this:⁴

```
\def\macro#1#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Such a macro can be used as:

```
\macro {1}{2}
\macro {1} {2} middle space gobbled
\macro 1 {2} middle space gobbled
\macro {1} 2 middle space gobbled
\macro 1 2 middle space gobbled
```

We show the result with some comments about how spaces are handled:

```
12
12 middle space gobbled
12 middle space gobbled
```

⁴ The `\dontleavehmode` command make the examples stay on one line.

```

12|         middle space gobbled
12|         middle space gobbled

```

A definition with delimited parameters looks like this:

```

\def\macro[#1]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\hss}}

```

When we use this we get:

```

\macro [1]
\macro [ 1]   leading space kept
\macro [1 ]   trailing space kept
\macro [ 1 ]  both spaces kept

```

Again, watch the handling of spaces:

```

1|
1|         leading space kept
1|         trailing space kept
1|         both spaces kept

```

Just for the record we show a combination:

```

\def\macro[#1]#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}

```

With this:

```

\macro [1]{2}
\macro [1] {2}
\macro [1] 2

```

we can again see the spaces go away:

```

12|
12|
12|

```

A definition with two separately delimited parameters is given next:

```

\def\macro[#1#2]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}

```

When used:

```
\macro [12]
\macro [ 12]    leading space gobbled
\macro [12 ]    trailing space kept
\macro [ 12 ]    leading space gobbled, trailing space kept
\macro [1 2]    middle space kept
\macro [ 1 2 ]    leading space gobbled, middle and trailing space kept
```

We get ourselves:

```
|12|
|12|    leading space gobbled
|12| |    trailing space kept
|12| |    leading space gobbled, trailing space kept
|1 2| |    middle space kept
|1 2| |    leading space gobbled, middle and trailing space kept
```

These examples demonstrate that the engine does some magic with spaces before (and therefore also between multiple) parameters.

We will now go a bit beyond what traditional T_EX engines do and enter the domain of LuaMetaT_EX specific parameter specifiers. We start with one that deals with this hard coded space behavior:

```
\def\macro[#^#^]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The #^ specifier will count the parameter, so here we expect again two arguments but the space is kept when parsing for them.

```
\macro [12]
\macro [ 12]
\macro [12 ]
\macro [ 12 ]
\macro [1 2]
\macro [ 1 2 ]
```

Now keep in mind that we could deal well with all kind of parameter handling in ConT_EXt for decades, so this is not really something we missed, but it complements the to be discussed other ones and it makes sense to have that level of control. Also, availability triggers usage. Nevertheless, some day the #^ specifier will come in handy.

```

|12|
|12|
|12|
|12|
|1 2|
|1 2|

```

We now come back to an earlier example:

```

\def\macro[#1]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}

```

When we use this we see that the braces in the second call are removed:

```

\macro [1]
\macro [{1}]

```

```

|1| |1|

```

This can be prohibited by the `#+` specifier, as in:

```

\def\macro[#+]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}

```

As we see, the braces are kept:

```

\macro [1]
\macro [{1}]

```

Again, we could easily get around that (for sure intended) side effect but it just makes nicer code when we have a feature like this.

```

|1| |{1}|

```

Sometimes you want to grab an argument but are not interested in the results. For this we have two specifiers: one that just ignores the argument, and another one that keeps counting but discards it, i.e. the related parameter is empty.

```

\def\macro[#1][#0][#3][#-][#4]%
  {\dontleavehmode\hbox spread 1em
   {\vl\type{#1}\vl\type{#2}\vl\type{#3}\vl\type{#4}\vl\hss}}

```

The second argument is empty and the fourth argument is simply ignored which is why we need `#4` for the fifth entry.


```
\macro [1][2][3][4][5]
```

Here is proof that it works:

```
|1|3|5|
```

The reasoning behind dropping arguments is that for some cases we get around the nine argument limitation, but more important is that we don't construct token lists that are not used, which is more memory (and maybe even cpu cache) friendly.

Spaces are always kind of special in T_EX, so it will be no surprise that we have another specifier that relates to spaces.

```
\def\macro[#1]#* [#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

This permits usage like the following:

```
\macro [1][2]
\macro [1] [2]
```

```
|1|2| |1|2|
```

Without the optional 'grab spaces' specifier the second line would possibly throw an error. This because T_EX then tries to match][so the] [in the input is simply added to the first argument and the next occurrence of][will be used. That one can be someplace further in your source and if not T_EX complains about a premature end of file. But, with the #* option it works out okay (unless of course you don't have that second argument [2]).

Now, you might wonder if there is a way to deal with that second delimited argument being optional and of course that can be programmed quite well in traditional macro code. In fact, ConT_EXt does that a lot because it is set up as a parameter driven system with optional arguments. That subsystem has been optimized to the max over years and it works quite well and performance wise there is very little to gain. However, as soon as you enable tracing you end up in an avalanche of expansions and that is no fun.

This time the solution is not in some special specifier but in the way a macro gets defined.

```
\tolerant\def\macro[#1]#* [#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The magic \tolerant prefix with delimited arguments and just quits when there is no match. So, this is acceptable:

```
\macro [1][2]
\macro [1] [2]
\macro [1]
\macro
```

```
|12| |12| |1| |
```

We can check how many arguments have been processed with a dedicated conditional:

```
\tolerant\def\macro[#1]#*[#2]%
  {\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

We use this test:

```
\macro [1][2] \macro [1] [2] \macro [1] \macro
```

The result is: 2: |12| 2: |12| 1: |1|0: | which is what we expect because we flush inline and there is no change of mode. When the following definition is used in display mode, the leading n= can for instance start a new paragraph and when code in `\everypar` you can loose the right number when macros get expanded before the n gets injected.

```
\tolerant\def\macro[#1]#*[#2]%
  {n=\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

In addition to the `\ifarguments` test primitive there is also a related internal counter `\lastarguments` set that you can consult, so the `\ifarguments` is actually just a shortcut for `\ifcase \lastarguments`.

We now continue with the argument specifiers and the next two relate to this optional grabbing. Consider the next definition:

```
\tolerant\def\macro#1#*#2%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this test:

```
\macro {1} {2}
\macro {1}
\macro
```

We get:

```
|12| |1|\macro|
```

This is okay because the last `\macro` is a valid (single token) argument. But, we can make the braces mandate:

```
\tolerant\def\macro#=#*#=#%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Here the `#=` forces a check for braces, so:

```
\macro {1} {2}
\macro {1}
\macro
```

gives this:

```
|1|2| |1| |
```

However, we do lose these braces and sometimes you don't want that. Of course when you pass the results downstream to another macro you can always add them, but it was cheap to add a related specifier:

```
\tolerant\def\macro#_#*#_%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Again, the magic `\tolerant` prefix works will quit scanning when there is no match. So:

```
\macro {1} {2}
\macro {1}
\macro
```

leads to:

```
|{1}|{2}| |{1}| |
```

When you're tolerant it can be that you still want to pick up some argument later on. This is why we have a continuation option.

```
\tolerant\def\foo      [#1]*[#2]#:#3{!#1!#2!#3!}
\tolerant\def\oof[#1]*[#2]#:(#3)#:#4{!#1!#2!#3!#4!}
\tolerant\def\ofo      [#1]#:(#2)#:#3{!#1!#2!#3!}
```

Hopefully the next example demonstrates how it works:

```
\foo{3} \foo[1]{3} \foo[1][2]{3}
```

```

\oof{4} \oof[1]{4} \oof[1][2]{4}
\oof[1][2](3){4} \oof[1](3){4} \oof(3){4}
\ofo{3} \ofo[1]{3}
\ofo[1](2){3} \ofo(2){3}

```

As you can see we can have multiple continuations using the #: directive:

```

!!!3! !1!!3! !1!2!3!
!!!!4! !1!!!!4! !1!2!!!4!
!1!2!3!4! !1!!3!4! !!!3!4!
!!!3! !1!!3!
!1!2!3! !!2!3!

```

The last specifier doesn't work well with the \ifarguments state because we no longer know what arguments were skipped. This is why we have another test for arguments. A zero value means that the next token is not a parameter reference, a value of one means that a parameter has been set and a value of two signals an empty parameter. So, it reports the state of the given parameter as a kind of \ifcase.

```

\def\foo#1#2{ [\ifparameter#1\or(ONE)\fi\ifparameter#2\or(TWO)\fi] }

```

Of course the test has to be followed by a valid parameter specifier:

```

\foo{1}{2} \foo{1}{} \foo{}{2} \foo{}{}

```

The previous code gives this:

```

[(ONE)(TWO)] [(ONE)] [(TWO)] []

```

A combination check \ifparameters, again a case, matches the first parameter that has a value set.

We could add plenty of specifiers but we need to keep in mind that we're not talking of an expression scanner. We need to keep performance in mind, so nesting and backtracking are no option. We also have a limited set of useable single characters, but here's one that uses a symbol that we had left:

```

\def\startfoo[#/#/#/\stopfoo{ [#1](#2) }

```

The slash directive removes leading and trailing so called spacers as well as tokens that represent a paragraph end:

```

\startfoo [x ] x \stopfoo
\startfoo [ x ] x \stopfoo

```

```
\startfoo [ x] x \stopfoo
\startfoo [ x] \par x \par \par \stopfoo
```

So we get this:

```
[x](x) [x](x) [x](x) [x](x)
```

The next directive, the quitter #;, is demonstrated with an example. When no match has occurred, scanning picks up after this signal, otherwise we just quit.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
```

```
\tolerant\def\foo[#1]#;#={/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;#2{/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1/// /2/// /3///
```

```
//1// //2// //3//
```

```
///1/ ///2/ ///3/
```

I have to admit that I don't really need it but it made some macros that I was redefining behave better, so there is some self-interest here. Anyway, I considered some other features, like picking up a detokenized argument but I don't expect that to be of much

use. In the meantime we ran out of reasonable characters, but some day #? and #! might show up, or maybe I find a use for #< and #>. A summary of all this is given here:

+	keep the braces
-	discard and don't count the argument
/	remove leading and trailing spaces and pars
=	braces are mandate
_	braces are mandate and kept
^	keep leading spaces

1-9	an argument
0	discard but count the argument

*	ignore spaces
:	pick up scanning here
;	quit scanning

.	ignore pars and spaces
,	push back space when quit

The last two have not been discussed and were added later. The period directive gobbles space and par tokens and discards them in the process. The comma directive is like * but it pushes back a space when the matching quits.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par  
\foo(1)\quad\foo(2)\quad\foo(3)\par
```

```
\tolerant\def\foo[#1]#;#=#{/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par  
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;#2{/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par  
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;(#2)#;#=#{/#1/#2/#3/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par  
\foo(1)\quad\foo(2)\quad\foo(3)\par  
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```

/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1/// /2/// /3///
//1// //2// //3//
///1/ ///2/ ///3/

```

Gobbling spaces versus pushing back is an interface design decision because it has to do with consistency.

6.3 Runaway arguments

There is a particular troublesome case left: a runaway argument. The solution is not pretty but it's the only way: we need to tell the parser that it can quit.

```

\tolerant\def\foo[#1=#2]%
  {\ifarguments 0\or 1\or 2\or 3\or 4\fi:\v\type{#1}\v\type{#2}\v\}

```

The outcome demonstrates that one still has to do some additional checking for sane results and there are alternative way to (ab)use this mechanism. It all boils down to a clever combination of delimiters and `\ignorearguments`.

```

\dontleavehmode \foo[a=1]
\dontleavehmode \foo[b=]
\dontleavehmode \foo[=]
\dontleavehmode \foo[x]\ignorearguments

```

All calls are accepted:

```

2:a|1|
2:b|
2:|
1:x]|

```

Just in case you wonder about performance: don't expect miracles here. On the one hand there is some extra overhead in the engine (when defining macros as well as when collecting arguments during a macro call) and maybe using these new features can sort of compensate that. As mentioned: the gain is mostly in cleaner macro code and less clutter in tracing. And I just want the ConT_EXt code to look nice: that way users

can look in the source to see what happens and not drown in all these show-off tricks, special characters like underscores, at signs, question marks and exclamation marks.

For the record: I normally run tests to see if there are performance side effects and as long as processing the test suite that has thousands of files of all kind doesn't take more time it's okay. Actually, there is a little gain in ConT_EXt but that is to be expected, but I bet users won't notice it, because it's easily offset by some inefficient styling. Of course another gain of loosing some indirectness is that error messages point to the macro that the user called for and not to some follow up.

6.4 Introspection

A macro has a meaning. You can serialize that meaning as follows:

```
\tolerant\protected\def\foo#1[#2]#* [#3]%
  {(1=#1) (2=#3) (3=#3)}
```

```
\meaning\foo
```

The meaning of \foo comes out as:

```
tolerant protected macro:#1[#2]#* [#3]->(1=#1) (2=#3) (3=#3)
```

When you load the module system-tokens you can also say:

```
\luatokenable\foo
```

This produces a table of tokens specifications:

```
tolerant protected macro:#1[#2]#* [#3]->(1=#1) (2=#3) (3=#3)
```

tolerant protected control sequence: foo

502432	19	49	match	argument 1
266963	12	91	other char	[U+0005B
502994	19	50	match	argument 2
503680	12	93	other char] U+0005D
500068	19	42	match	argument *
500322	12	91	other char	[U+0005B
503121	19	51	match	argument 3
31045	12	93	other char] U+0005D
46310	20	0	end match	
<hr/>				
503117	12	40	other char	(U+00028

504226	12	49	other char	1	U+00031
499176	12	61	other char	=	U+0003D
502989	21	1	parameter reference		
261002	12	41	other char)	U+00029
386857	10	32	spacer		
503692	12	40	other char	(U+00028
498426	12	50	other char	2	U+00032
499137	12	61	other char	=	U+0003D
509904	21	3	parameter reference		
386835	12	41	other char)	U+00029
502827	10	32	spacer		
498451	12	40	other char	(U+00028
500116	12	51	other char	3	U+00033
502912	12	61	other char	=	U+0003D
503758	21	3	parameter reference		
499121	12	41	other char)	U+00029

A token list is a linked list of tokens. The magic numbers in the first column are the token memory pointers. and because macros (and token lists) get recycled at some point the available tokens get scattered, which is reflected in the order of these numbers. Normally macros defined in the macro package are more sequential because they stay around from the start. The second and third row show the so called command code and the specifier. The command code groups primitives in categories, the specifier is an indicator of what specific action will follow, a register number a reference, etc. Users don't need to know these details. This macro is a special version of the online variant:

```
\showluatokens\foo
```

That one is always available and shows a similar list on the console. Again, users normally don't want to know such details.

6.5 nesting

You can nest macros, as in:

```
\def\foo#1#2{\def\oof##1{<#1>##1<#2>}}
```

At first sight the duplication of # looks strange but this is what happens. When T_EX scans the definition of `\foo` it sees two arguments. Their specification ends up in the preamble that defines the matching. When the body is scanned, the `#1` and `#2` are turned into a parameter reference. In order to make nested macros with arguments

possible a # followed by another # becomes just one #. Keep in mind that the definition of \oof is delayed till the macro \foo gets expanded. That definition is just stored and the only thing that get's replaced are the two references to a macro parameter

control sequence: foo

502384	19	49	match		argument 1
502416	19	50	match		argument 2
503312	20	0	end match		
<hr/>					
507473	115	1	def		def
504217	134	0	tolerant call		oof
499221	6	35	parameter		
201296	12	49	other char	1	U+00031
503763	1	123	left brace		
500345	12	60	other char	<	U+0003C
499172	21	1	parameter reference		
502331	12	62	other char	>	U+0003E
498449	6	35	parameter		
500258	12	49	other char	1	U+00031
502577	12	60	other char	<	U+0003C
502729	21	2	parameter reference		
503124	12	62	other char	>	U+0003E
504197	2	125	right brace		

Now, when we look at these details, it might become clear why for instance we have ‘variable’ names like #4 and not #whatever (with or without hash). Macros are essentially token lists and token lists can be seen as a sequence of numbers. This is not that different from other programming environments. When you run into buzzwords like ‘bytecode’ and ‘virtual machines’ there is actually nothing special about it: some high level programming (using whatever concept, and in the case of T_EX it's macros) eventually ends up as a sequence of instructions, say bytecodes. Then you need some machinery to run over that and act upon those numbers. It's something you arrive at naturally when you play with interpreting languages.⁵

So, internally a #4 is just one token, a operator-operand combination where the operator is “grab a parameter” and the operand tells “where to store” it. Using names is of course

⁵ I actually did when I wrote an interpreter for some computer assisted learning system, think of a kind of interpreted Pascal, but later realized that it was a a bytecode plus virtual machine thing. I'd just applied what I learned when playing with eight bit processors that took bytes, and interpreted opcodes and such. There's nothing spectacular about all this and I only realized decades later that the buzzwords describes old natural concepts.

an option but then one has to do more parsing and turn the name into a number⁶, add additional checking in the macro body, figure out some way to retain the name for the purpose of reporting (which then uses more token memory or strings). It is simply not worth the trouble, let alone the fact that we loose performance, and when T_EX showed up those things really mattered.

It is also important to realize that a # becomes either a preamble token (grab an argument) or a reference token (inject the passed tokens into a new input level). Therefore the duplication of hash tokens ## that you see in macro nested bodies also makes sense: it makes it possible for the parser to distinguish between levels. Take:

```
\def\foo#1{\def\oof##1{#1##1#1}}
```

Of course one can think of this:

```
\def\foo#fence{\def\oof#text{#fence#text#fence}}
```

But such names really have to be unique then! Actually ConT_EXt does have an input method that supports such names, but discussing it here is a bit out of scope. Now, imagine that in the above case we use this:

```
\def\foo[#1][#2]{\def\oof##1{#1##1#2}}
```

If you're a bit familiar with the fact that T_EX has a model of category codes you can imagine that a predictable “hash followed by a number” is way more robust than enforcing the user to ensure that catcodes of ‘names’ are in the right category (read: is a bracket part of the name or not). So, say that we go completely arbitrary names, we then suddenly needs some escaping, like:

```
\def\foo[#{left}][#{right}]{\def\oof#{text}{#{left}#{text}#{right}}}
```

And, if you ever looked into macro packages, you will notice that they differ in the way they assign category codes. Asking users to take that into account when defining macros makes not that much sense.

So, before one complains about T_EX being obscure (the hash thing), think twice. Your demand for simplicity for your coding demand will make coding more cumbersome for the complex cases that macro packages have to deal with. It's comparable using T_EX for input or using (say) mark down. For simple documents the later is fine, but when things become complex, you end up with similar complexity (or even worse because you lost the enforced detailed structure). So, just accept the unavoidable: any language has its peculiar properties (and for sure I do know why I dislike some languages for it). The

⁶ This is kind of what MetaPost does with parameters to macros. The side effect is that in reporting you get text0, expr2 and such reported which doesn't make things more clear.

\TeX system is not the only one where dollars, percent signs, ampersands and hashes have special meaning.

6.6 Prefixes

Traditional \TeX has three prefixes that can be used with macros: `\global`, `\outer` and `\long`. The last two are no-op's in `LuaMetaTeX` and if you want to know what they do (did) you can look it up in the \TeX book. The ε - \TeX extension gave us `\protected`.

In `LuaMetaTeX` we have `\global`, `\protected`, `\tolerant` and overload related prefixes like `\frozen`. A protected macro is one that doesn't expand in an expandable context, so for instance inside an `\edef`. You can force expansion by using the `\expand` primitive in front which is also something `LuaMetaTeX`.

Frozen macros cannot be redefined without some effort. This feature can to some extent be used to prevent a user from overloading, but it also makes it harder for the macro package itself to redefine on the fly. You can remove the lock with `\unletfrozen` and add a lock with `\letfrozen` so in the end users still have all the freedoms that \TeX normally provides.

```

                \def\foo{foo} 1: \meaning\foo
        \frozen\def\foo{foo} 2: \meaning\foo
    \unletfrozen   \foo      3: \meaning\foo
\protected\frozen\def\foo{foo} 4: \meaning\foo
    \unletfrozen   \foo      5: \meaning\foo

```

```

1: macro:foo
2: macro:foo
3: macro:foo
4: protected macro:foo
5: protected macro:foo

```

This actually only works when you have set `\overloadmode` to a value that permits redefining a frozen macro, so for the purpose of this example we set it to zero.

A `\tolerant` macro is one that will quit scanning arguments when a delimiter cannot be matched. We saw examples of that in a previous section.

These prefixes can be chained (in arbitrary order):

```
\frozen\tolerant\protected\global\def\foo[#1]#*[#2]{...}
```

There is actually an additional prefix, `\immediate` but that one is there as signal for a macro that is defined in and handled by Lua. This prefix can then perform the same function as the one in traditional `TEX`, where it is used for backend related tasks like `\write`.

Now, the question is of course, to what extent will `ConTEXt` use these new features. One important argument in favor of using `\tolerant` is that it gives (hopefully) better error messages. It also needs less code due to lack of indirectness. Using `\frozen` adds some safeguards although in some places where `ConTEXt` itself overloads commands, we need to defrost. Adapting the code is a tedious process and it can introduce errors due to mistypings, although these can easily be fixed. So, it will be used but it will take a while to adapt the code base.

One problem with frozen macros is that they don't play nice with for instance `\futurelet`. Also, there are places in `ConTEXt` where we actually do redefine some core macro that we also want to protect from redefinition by a user. One can of course `\unletfrozen` such a command first but as a bonus we have a prefix `\overloaded` that can be used as prefix. So, one can easily redefine a frozen macro but it takes a little effort. After all, this feature is mainly meant to protect a user for side effects of definitions, and not as final blocker.⁷

A frozen macro can still be overloaded, so what if we want to prevent that? For this we have the `\permanent` prefix. Internally we also create primitives but we don't have a prefix for that. But we do have one for a very special case which we demonstrate with an example:

```
\def\F00 % trickery needed to pick up an optional argument
  {\noalign{\vskip10pt}}

\noaligned\protected\tolerant\def\00F[#1]%
  {\noalign{\vskip\iftok{#1}\emptytoks10pt\else#1\fi}}

\starttabulate[|l|l|]
  \NC test \NC test \NC \NR
  \NC test \NC test \NC \NR
  \F00
  \NC test \NC test \NC \NR
  \00F[30pt]
  \NC test \NC test \NC \NR
  \00F
```

⁷ As usual adding features like this takes some experimenting and we're now at the third variant of the implementation, so we're getting there. The fact that we can apply such features in large macro package like `ConTEXt` helps figuring out the needs and best approaches.

```
\NC test \NC test \NC \NR
\stoptabulate
```

When $\text{T}_{\text{E}}\text{X}$ scans input (from a file or token list) and starts an alignment, it will pick up rows. When a row is finished it will look ahead for a `\noalign` and it expands the next token. However, when that token is protected, the scanner will not see a `\noalign` in that macro so it will likely start complaining when that next macro does get expanded and produces a `\noalign` when a cell is built. The `\noaligned` prefix flags a macro as being one that will do some `\noalign` as part of its expansion. This trick permits clean macros that pick up arguments. Of course it can be done with traditional means but this whole exercise is about making the code look nice.

The table comes out as:

```
test test
test test

test test
```

```
test test

test test
```

One can check the flags with `\ifflags` which takes a control sequence and a number, where valid numbers are:

1 frozen	2 permanent	4 immutable	8 primitive
16 mutable	32 noaligned	64 instance	

The level of checking is controlled with the `\overloadmode` but I'm still not sure about how many levels we need there. A zero value disables checking, the values 1 and 3 give warnings and the values 2 and 4 trigger an error.

6.6 Colofon

Author	Hans Hagen
Con $\text{T}_{\text{E}}\text{X}$ t	2021.09.06 11:47
LuaMeta $\text{T}_{\text{E}}\text{X}$	2.0923
Support	www.pragma-ade.com contextgarden.net

7 Grouping

low level

TEX

grouping

Contents

7.1	Introduction	98
7.2	Pascal	98
7.3	T _E X	98
7.4	MetaPost	99
7.5	Lua	100
7.6	C	100

7.1 Introduction

This is a rather short explanation. I decided to write it after presenting the other topics at the 2019 ConT_EXt meeting where there was a question about grouping.

7.2 Pascal

In a language like Pascal, the language that T_EX has been written in, or Modula, its successor, there is no concept of grouping like in T_EX. But we can find keywords that suggests this:

```
for i := 1 to 10 do begin ... end
```

This language probably inspired some of the syntax of T_EX and MetaPost. For instance an assignment in MetaPost uses := too. However, the `begin` and `end` don't really group but define a block of statements. You can have local variables in a procedure or function but the block is just a way to pack a sequence of statements.

7.3 T_EX

In T_EX macros (or source code) the following can occur:

```
\begingroup
...
\endgroup
```

as well as:

```
\bgroup
...
\egroup
```

Here we really group in the sense that assignments to variables inside a group are forgotten afterwards. All assignments are local to the group unless they are explicitly done global:

```
\scratchcounter=1
\def\foo{foo}
\begingroup
  \scratchcounter=2
  \global\globalscratchcounter=2
  \gdef\foo{F00}
\endgroup
```

Here `\scratchcounter` is still one after the group is left but its global counterpart is now two. The `\foo` macro is also changed globally.

Although you can use both sets of commands to group, you cannot mix them, so this will trigger an error:

```
\bgroup
\endgroup
```

The bottomline is: if you want a value to persist after the group, you need to explicitly change its value globally. This makes a lot of sense in the perspective of $\text{T}_{\text{E}}\text{X}$.

7.4 MetaPost

The MetaPost language also has a concept of grouping but in this case it's more like a programming language.

```
begingroup ;
  n := 123 ;
engroup ;
```

In this case the value of `n` is 123 after the group is left, unless you do this (for numerics there is actually no need to declare them):

```
begingroup ;
  save n ; numeric n ; n := 123 ;
engroup ;
```

Given the use of MetaPost (read: MetaFont) this makes a lot of sense: often you use macros to simplify code and you do want variables to change. Grouping in this language

serves other purposes, like hiding what is between these commands and let the last expression become the result. In a `vardef` grouping is implicit.

So, in MetaPost all assignments are global, unless a variable is explicitly saved inside a group.

7.5 Lua

In Lua all assignments are global unless a variable is defines local:

```
local x = 1
local y = 1
for i = 1, 10 do
    local x = 2
    y = 2
end
```

Here the value of `x` after the loop is still one but `y` is now two. As in Lua_{TeX} we mix _{TeX}, MetaPost and Lua you can mix up these concepts. Another mixup is using `:=`, `endfor`, `fi` in Lua after done some MetaPost coding or using `end` instead of `endfor` in MetaPost which can make the library wait for more without triggering an error. Proper syntax highlighting in an editor clearly helps.

7.6 C

The Lua language is a mix between Pascal (which is one reason why I like it) and C.

```
int x = 1 ;
int y = 1 ;
for (i=1; i<=10;i++) {
    int x = 2 ;
    y = 2 ;
}
```

The semicolon is also used in Pascal but there it is a separator and not a statement end, while in MetaPost it does end a statement (expression).

7.6 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

8 Security

low level

TEX

security

Contents

8.1	Preamble	103
8.2	Flags	103
8.3	Complications	106
8.4	Introspection	107

8.1 Preamble

Here I will discuss a moderate security subsystem of LuaMetaT_EX and therefore ConT_EXt LMTX. This is not about security in the sense of the typesetting machinery doing harm to your environment, but more about making sure that a user doesn't change the behavior of the macro package in ways that introduce interference and thereby unwanted side effect. It's all about protecting macros.

This is all very experimental and we need to adapt the ConT_EXt source code to this. Actually that will happen a few times because experiments trigger that. It might take a few years before the security model is finalized and all files are updated accordingly. There are lots of files and macros involved. In the process the underlying features in the engine might evolve.

8.2 Flags

Before we go into the security levels we see what flags can be set. The T_EX language has a couple of so called prefixes that can be used when setting values and defining macros. Any engine that has traditional T_EX with ε -T_EX extensions can do this:

```

                \def\foo{foo}
\global        \def\foo{foo}
\global\protected\def\foo{foo}

```

And LuaMetaT_EX adds another one:

```

                \tolerant      \def\foo{foo}
\global\tolerant      \def\foo{foo}
\global\tolerant\protected\def\foo{foo}

```

What these prefixes do is discussed elsewhere. For now it is enough to know that the two optional prefixes `\protected` and `\tolerant` make for four distinctive cases of macro calls.

But there are more prefixes:

frozen	a macro that has to be redefined in a managed way
permanent	a macro that had better not be redefined
primitive	a primitive that normally will not be adapted
immutable	a macro or quantity that cannot be changed, it is a constant
mutable	a macro that can be changed no matter how well protected it is

instance	a macro marked as (for instance) be generated by an interface
----------	---

noaligned	the macro becomes acceptable as <code>\noalign</code> alias
-----------	---

overloaded	when permitted the flags will be adapted
enforced	all is permitted (but only in zero mode or ini mode)
aliased	the macro gets the same flags as the original

These prefixed set flags to the command at hand which can be a macro but basically any control sequence.

To what extent the engine will complain when a property is changed in a way that violates the above depends on the parameter `\overloadmode`. When this parameter is set to zero no checking takes place. More interesting are values larger than zero. If that is the case, when a control sequence is flagged as mutable, it is always permitted to change. When it is set to immutable one can never change it. The other flags determine the kind of checking done. Currently the following overload values are used:

		immutable	permanent	primitive	frozen	instance
1	warning	*	*	*		
2	error	*	*	*		
3	warning	*	*	*	*	
4	error	*	*	*	*	
5	warning	*	*	*	*	*
6	error	*	*	*	*	*

The even values (except zero) will abort the run. In `ConTEXt` we plug in a callback that deals with the messages. A value of 255 will freeze this parameter. At level five and above the instance flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

So, how does it work. The following is okay:

```
\def\MacroA{A}
\def\MacroB{B}
```



```
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

The first two macros are ordinary ones, and the last two lines just create an alias. Such an alias shares the definition, but when for instance `\MacroA` is redefined, its new meaning will not be reflected in the alias.

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

This also works, because the `\let` will create an alias with the protected property but it will not take the permanent property along. For that we need to say:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\permanent\let\MyMacro\MacroA
\permanent\let\MyMacro\MacroB
```

or, when we want to copy all properties:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\aliased\let\MyMacro\MacroA
\aliased\let\MyMacro\MacroB
```

However, in `ConTEXt` we have commands that we like to protect against overloading but at the same time have a different meaning depending on the use case. An example is the `\NC` (next column) command that has a different implementation in each of the table mechanisms.

```
\permanent\protected\def\NC_in_table {...}
\permanent\protected\def\NC_in_tabulate{...}
\aliased\let\NC\NC_in_table
\aliased\let\NC\NC_in_tabulate
```

Here the second aliasing of `\NC` fails (assuming of course that we enabled overload checking). One can argue that grouping can be used but often no grouping takes place when we redefine on the fly. Because `frozen` is less restrictive than `primitive` or `permanent`, and of course `immutable`, the next variant works:

```
\frozen\protected\def\NC_in_table {...}
```

```
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\let\NC\NC_in_table
\overloaded\let\NC\NC_in_tabulate
```

However, in practice, as we want to keep the overload checking, we have to do:

```
\frozen\protected\def\NC_in_table  {...}
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\frozen\let\NC\NC_in_table
\overloaded\frozen\let\NC\NC_in_tabulate
```

or use `\aliased`, but there might be conflicting permissions. This is not that nice, so there is a kind of dirty trick possible. Consider this:

```
\frozen\protected\def\NC_in_table  {...}
\frozen\protected\def\NC_in_tabulate{...}
\def\setNCintable  {\enforced\let\frozen\let\NC\NC_in_table}
\def\setNCintabulate{\enforced\let\frozen\let\NC\NC_in_tabulate}
```

When we're in so called `initex` mode or when the overload mode is zero, the `\enforced` prefix is internalized in a way that signals that the follow up is not limited by the overload mode and permissions. This definition time binding mechanism makes it possible to use permanent macros that users cannot redefine, but existing macros can, unless of course they tweak the mode parameter.

Now keep in mind that users can always cheat but that is intentional. If you really want to avoid that you can set the overload mode to 255 after which it cannot be set any more. However, it can be useful to set the mode to zero (or some warning level) when foreign macro packages are used.

8.3 Complications

One side effect of all this is that all those prefixes can lead to more code. On the other hand we save some due to the extended macro argument handling features. When you take the size of the format file as reference, in the end we get a somewhat smaller file. Every token that you add or remove gives a 8 bytes difference. The extra overhead that got added to the engine is compensated by the fact that some macro implementations can be more efficient. In the end, in spite of these new features and the more extensive testing of flags performance is about the same.⁸

⁸ And if you wonder about memory, by compacting the used (often scattered) token memory before dumping I manages to save some 512K on the format file, so often the loss and gain are somewhere else.

8.4 Introspection

In case you want to get some details about the properties of a macro, you can check its meaning. The full variant shows all of them.

`% a macro with two optional arguments with optional spacing in between:`

```
\permanent\tolerant\protected\def\MyFoo[#1]#*[#2]{(#1)(#2)}
```

```
\meaningless\MyFoo\par
```

```
\meaning \MyFoo\par
```

```
\meaningfull\MyFoo\par
```

```
[#1]#*[#2]->(#1)(#2)
```

```
tolerant protected macro:[#1]#*[#2]->(#1)(#2)
```

```
permanent tolerant protected macro:[#1]#*[#2]->(#1)(#2)
```

8.4 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

9 Characters

low level

TEX

characters

Contents

9.1	Introduction	109
9.2	History	109
9.3	The heritage	110
9.4	The LMTX approach	111

9.1 Introduction

This explanation is part of the low level manuals because in practice users will not have to deal with these matters in MkIV and even less in LMTX. You can skip to the last section for commands.

9.2 History

If we travel back in time to when $\text{T}_{\text{E}}\text{X}$ was written we end up in eight bit character universe. In fact, the first versions assumed seven bits, but for comfortable use with languages other than English that was not sufficient. Support for eight bits permits the usage of so called code pages as supported by operating systems. Although ascii input became kind of the standard soon afterwards, the engine can be set up for different encodings. This is not only true for $\text{T}_{\text{E}}\text{X}$, but for many of its companions, like MetaFont and therefore MetaPost.⁹

Core components of a $\text{T}_{\text{E}}\text{X}$ engine are hyphenation of words, applying inter-character kerns and build ligatures. In traditional $\text{T}_{\text{E}}\text{X}$ engines those processes are interwoven into the par builder but in Lua $\text{T}_{\text{E}}\text{X}$ these are separate stages. The original approach is the reason that there is a relation between the input encoding and the font encoding: the character in the input is the slot used in a reference to a glyph. When producing the final result (e.g. pdf) there can also be a mapping to an index in a font resource.

```
input A [tex ->] font slot A [backend ->] glyph index A
```

The mapping that $\text{T}_{\text{E}}\text{X}$ does is normally one-to-one but an input character can undergo some transformation. For instance a character beyond ascii 126 can be made active and expand to some character number that then becomes the font slot. So, it is the expansion (or meaning) of a character that end up as numeric reference in the glyph node. Virtual fonts can introduce yet another remapping but that's only visible in the backend.

⁹ This remapping to an internal representation (e.g. ebcdic) is not present in Lua $\text{T}_{\text{E}}\text{X}$ where we assume utf8 to be the input encoding. The MetaPost library that comes with Lua $\text{T}_{\text{E}}\text{X}$ still has that code but in LuaMeta $\text{T}_{\text{E}}\text{X}$ it's gone. There one can set up the machinery to be utf8 aware too.

Actually, in Lua \TeX the same happens but in practice there is no need to go active because (at least in Con \TeX t) we assume a Unicode path so there the font slot is the Unicode got from the utf8 input.

In the eight bit universe macro packages (have to) provide all kind of means to deal with (in the perspective of English) special characters. For instance, `\"a` would put a diaeresis on top of the a or even better, refer to a character in the encoding that the chosen font provides. Because there are some limitations of what can go in an eight bit font, and because in different countries the used \TeX fonts evolved kind of independent, we ended up with quite some different variants of fonts. It was only with the Latin Modern project that this became better. Interesting is that when we consider the fact that such a font has often also hardly used symbols (like registered or copyright) coming up with an encoding vector that covers most (latin based) European languages (scripts) is not impossible¹⁰ Special symbols could simply go into a dedicated font, also because these are always accessed via a macro so who cares about the input. It never happened.

Keep in mind that when utf8 is used with eight bit engines, Con \TeX t will convert sequences of characters into a slot in a font (depending on the font encoding used which itself depends on the coverage needed). For this every first (possible) byte of a multi-byte utf sequence is an active character, which is no big deal because these are outside the ascii range. Normal ascii characters are single byte utf sequences and fall through without treatment.

Anyway, in Con \TeX t MkII we dealt with this by supporting mixed encodings, depending on the (local) language, referencing the relevant font. It permits users to enter the text in their preferred input encoding and also get the words properly hyphenated. But we can leave these MkII details behind.

9.3 The heritage

In MkIV we got rid of input and font encodings, although one can still load files in a specific code page.¹¹ We also kept the means to enter special characters, if only because text editors seldom support(ed) a wide range of visual editing of those. This is why we still have

```
\"u  ^a  \v{s}  \AE  \ij  \eacute  \oslash
```

¹⁰ And indeed in the Latin Modern project we came up with one but it was already too late for it to become popular.

¹¹ I'm not sure if users ever depend on an input encoding different from utf8.

and many more. The ones with one character names are rather common in the $\text{T}_{\text{E}}\text{X}$ community but it is definitely a weird mix of symbols. The next two are kind of outdated: in these days you delegate that to the font handler, where turning them into ‘single’ character references depends on what the font offers, how it is set up with respect to (for instance) ligatures, and even might depend on language or script.

The ones with the long names partly are tradition, but as we have a lot of them, in MkII they actually serve a purpose. These verbose names are used in the so called encoding vectors and are part of the utf expansion vectors. They are also used in labels so that we have a good indication if what goes in there: remember that in those times editors often didn't show characters, unless the font for display had them, or the operating system somehow provided them from another font. These verbose names are used for latin, greek and cyrillic and for some other scripts and symbols. They take up quite a bit of hash space and the format file.¹²

9.4 The LMTX approach

In the process of tagging all (public) macros in LMTX (which happened in 2020-2021) I wondered if we should keep these one character macros, the references to special characters and the verbose ones. When asked on the mailing list it became clear that users still expect the short ones to be present, often just because old $\text{bibT}_{\text{E}}\text{X}$ files are used that might need them. However, in MkIV and LMTX we load $\text{bibT}_{\text{E}}\text{X}$ files in a way that turn these special character references into proper utf8 input so it makes a weak argument. Anyway, although they could go, for now we keep them because users expect them. However, in LMTX the implementation is somewhat different now, a bit more efficient in terms of hash and memory, potentially a bit less efficient in runtime, but no one will notice that.

A new command has been introduced, the very short $\backslash\text{chr}$.

```
 $\backslash\text{chr}$  {a`}  $\backslash\text{chr}$  {a´}  $\backslash\text{chr}$  {a¨}
 $\backslash\text{chr}$  {`a}  $\backslash\text{chr}$  {'a}  $\backslash\text{chr}$  {"a}
 $\backslash\text{chr}$  {a acute}  $\backslash\text{chr}$  {a grave}  $\backslash\text{chr}$  {a umlaut}
 $\backslash\text{chr}$  {aacute}  $\backslash\text{chr}$  {agrave}  $\backslash\text{chr}$  {aumlaut}
```

In the first line the composed character using two characters, a base and a so called mark. Actually, one doesn't have to use $\backslash\text{chr}$ in that case because $\text{ConT}_{\text{E}}\text{Xt}$ does already collapse characters for you. The second line looks like the shortcuts $\backslash`$, \backslash' and $\backslash"$. The third and fourth lines could eventually replace the more symbolic long names, if we feel

¹² In MkII we have an abstract front-end with respect to encodings and also an abstract backend with respect to supported drivers but both approaches no longer make sense today.

the need. Watch out: in Unicode input the marks come *after*.

```
à á ä
à á ä
á à a~mla~t
á à a~mla~t
```

Currently the repertoire is somewhat limited but it can be easily be extended. It all depends on user needs (doing Greek and Cyrillic for instance). The reason why we actually save code deep down is that the helpers for this have always been there.¹³

The `\` commands are now just aliases to more verbose and less hackery looking macros:

```
\withgrave      à  \`  à
\withacute      á  \'  á
\withcircumflex â  \^  â
\withtilde      ã  \~  ã
\withmacron     ā  \=  ā
\withbreve      ě  \u  ě
\withdotaccent  ċ  \.  ċ
\withdiaeresis  ë  \"  ë
\withring       ũ  \r  ũ
\withhungarumlaut  ű  \H  ű
\withcaron      ě  \v  ě
\withcedilla    ħ  \c  ħ
\withogonek     ę  \k  ę
```

Not all fonts have these special characters. Most natural is to have them available as precomposed single glyphs, but it can be that they are just two shapes with the marks anchored to the base. It can even be that the font somehow overlays them, assuming (roughly) equal widths. The `compose` font feature in `ConTEXt` normally can handle most well.

An occasional ugly rendering doesn't matter that much: better have something than nothing. But when it's the main language (script) that needs them you'd better look for a font that handles them. When in doubt, in `ConTEXt` you can enable checking:

command	equivalent to
<code>\checkmissingcharacters</code>	<code>\enabletrackers[fonts.missing]</code>
<code>\removemissingcharacters</code>	<code>\enabletrackers[fonts.missing=remove]</code>

¹³ So if needed I can port this approach back to MkIV, but for now we keep it as is because we then have a reference.

```
\replacemissingcharacters \enabletrackers[fonts.missing=replace]
\handlemissingcharacters \enabletrackers[fonts.missing={decompose,replace}]
```

The decompose variant will try to turn a composed character into its components so that at least you get something. If that fails it will inject a replacement symbol that stands out so that you can check it. The console also mentions missing glyphs. You don't need to enable this by default¹⁴ but you might occasionally do it when you use a font for the first time.

In LMTX this mechanism has been upgraded so that replacements follow the shape and are actually real characters. The decomposition has not yet been ported back to MkIV.

The full list of commands can be queried when a tracing module is loaded:

```
\usemodule[s][characters-combinations]
```

```
\showcharactercombinations
```

We get this list:

acute	U+00301	´	\withacute
breve	U+00306	˘	\withbreve
caron	U+0030C	ˇ	\withcaron
caron below	U+0032C	ˇ̣	\withcaronbelow
cedilla	U+00327	¸	\withcedilla
circumflex	U+00302	ˆ	\withcircumflex
circumflex below	U+0032D	ˆ̣	\withcircumflexbelow
comma below	U+00327	¸	\withcommabelow
diaeresis	U+00308	¨	\withdiaeresis
dieresis	U+00308	¨	\withdieresis
dot	U+00307	·	\withdot
dot below	U+00323	·̣	\withdotbelow
double acute	U+0030B	˝	\withdoubleacute
double grave	U+0030F	˘˘	\withdoublegrave
double vertical line	U+0030E	="	\withdoubleverticalline
grave	U+00300	˘	\withgrave
hook	U+00309	¸	\withhook
hook below	U+1FA9D	¸̣	\withhookbelow
hungarumlaut	U+0030B	˝	\withhungarumlaut
inverted breve	U+00311	˘̂	\withinvertedbreve
line	U+00304	-	\withline

¹⁴ There is some overhead involved here.

line below	U+00331	_	\withlinebelow
macron	U+00304	˘	\withmacron
macron below	U+00331	_	\withmacronbelow
middle dot	U+000B7	·	\withmiddledot
ogonek	U+00328	˙	\withogonek
overline	U+00305	—	
ring	U+0030A	°	\withring
ring below	U+00325	◦	\withringbelow
slash	U+0002F	/	\withslash
stroke	U+0002F	/	\withstroke
tilde	U+00303	~	\withtilde
tilde below	U+00330	˜	\withtildebelow
vertical line	U+0030D	∣	\withverticalline

Some combinations are special for ConT_EXt because Unicode doesn't specify decomposition for all composed characters.

9.4 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

10 Scope

low level

TEX

scope

Contents

10.1 Introduction	116
10.2 Registers	116
10.3 Allocation	118
10.4 Files	121

10.1 Introduction

When I visited the file where register allocations are implemented I wondered to what extent it made sense to limit allocation to global instances only. This chapter deals with this phenomena.

10.2 Registers

In \TeX definitions can be local or global. Most assignments are local within a group. Registers and definitions can be assigned global by using the `\global` prefix. There are also some properties that are global by design, like `\prevdepth`. A mixed breed are boxes. When you tweak its dimensions you actually tweak the current box, which can be an outer level. Compare:

```
\scratchcounter = 1
here the counter has value 1
\begingroup
  \scratchcounter = 2
  here the counter has value 2
\endgroup
here the counter has value 1
```

with:

```
\setbox\scratchbox=\hbox{}
here the box has zero width
\begingroup
  \wd\scratchbox=10pt
  here the box is 10pt wide
\endgroup
here the box is 10pt wide
```

It all makes sense so a remark like “Assignments to box dimensions are always global”

are sort of confusing. Just look at this:

```
\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
\begingroup
  \setbox\scratchbox=\hbox{}
  here the box is \the\wd\scratchbox\ wide\par
  \begingroup
    \wd\scratchbox=15pt
    here the box is \the\wd\scratchbox\ wide\par
  \endgroup
  here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par
```

here the box is 20.0pt wide
 here the box is 0.0pt wide
 here the box is 15.0pt wide
 here the box is 15.0pt wide
 here the box is 20.0pt wide

If you don't think about it, what happens is what you expect. Now watch the next variant:

The `\global` is only effective for the current box. It is good to realize that when we talk registers, the box register behaves just like any other register but the manipulations happen to the current one.

```
\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
\begingroup
  \setbox\scratchbox=\hbox{}
  here the box is \the\wd\scratchbox\ wide\par
  \begingroup
    \global\wd\scratchbox=15pt
    here the box is \the\wd\scratchbox\ wide\par
  \endgroup
  here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par

here the box is 20.0pt wide
```

here the box is 0.0pt wide
 here the box is 15.0pt wide
 here the box is 15.0pt wide
 here the box is 20.0pt wide

```
\scratchdimen=20pt
here the dimension is \the\scratchdimen\par
\begingroup
  \scratchdimen=0pt
  here the dimension is \the\scratchdimen\par
  \begingroup
    \global\scratchdimen=15pt
    here the dimension is \the\scratchdimen\par
  \endgroup
  here the dimension is \the\scratchdimen\par
\endgroup
here the dimension is \the\scratchdimen\par

here the dimension is 20.0pt
here the dimension is 0.0pt
here the dimension is 15.0pt
here the dimension is 15.0pt
here the dimension is 15.0pt
```

10.3 Allocation

The plain T_EX format has set some standards and one of them is that registers are allocated with `\new...` commands. So we can say:

```
\newcount\mycounta
\newdimen\mydimena
```

These commands take a register from the pool and relate the given name to that entry. In ConT_EXt we have a bunch of predefined scratch registers for general use, like:

```
scratchcounter      : \meaningfull\scratchcounter
scratchcounterone  : \meaningfull\scratchcounterone
scratchcountertwo  : \meaningfull\scratchcountertwo
scratchdimen       : \meaningfull\scratchdimen
scratchdimenone    : \meaningfull\scratchdimenone
scratchdimentwo    : \meaningfull\scratchdimentwo
```


The meaning reveals what these are:

```
scratchcounter : permanent \count257
scratchcounterone : permanent \count260
scratchcountertwo : permanent \count261
scratchdimen : permanent \dimen257
scratchdimenone : permanent \dimen260
scratchdimentwo : permanent \dimen261
```

You can use the numbers directly but that is a bad idea because they can clash! In the original T_EX engine there are only 256 registers and some are used by the engine and the core of a macro package itself, so that leaves a little amount for users. The ϵ -T_EX extension lifted that limitation and bumped to 32K and LuaT_EX upped that to 64K. One could go higher but what makes sense? These registers are taking part of the fixed memory slots because that makes nested (grouped) usage efficient and access fast. The number you see above is deduced from the so called command code (here indicated by \count) and an index encoded in the same token. So, \scratchcounter takes a single token contrary to the verbose \count257 that takes four tokens where the number gets parsed every time it is needed. But those are details that a user can forget.

As mentioned, commands like \newcount \foo create a global control sequence \foo referencing a counter. You can locally redefine that control sequence unless in LuaMetaT_EX you have so called overload mode enabled. You can do local or global assignments to these registers.

```
\scratchcounter = 123
\begingroup
  \scratchcounter = 456
  \begingroup
    \global\scratchcounter = 789
  \endgroup
\endgroup
```

And in both cases count register 257 is set. When an assignment is global, all current values to that register get the same value. Normally this is all quite transparent: you get what you ask for. However the drawback is that as a user you cannot know what variables are already defined, which means that this will fail (that is: it will issue a message):

```
\newcount\scratchcounter
```

as will the second line in:

```
\newcount\myscratchcounter
\newcount\myscratchcounter
```

In ConTEXt the scratch registers are visible but there are lots of internally used ones are protected from the user by more obscure names. So what if you want to use your own register names without ConTEXt barking to you about not being able to define it. This is why in LMTX (and maybe some day in MkIV) we now have local definitions:

```
\begingroup
  \newlocaldimen\mydimena    \mydimena1\onepoint
  \newlocaldimen\mydimenb    \mydimenb2\onepoint
  (\the\mydimena,\the\mydimenb)
\begingroup
  \newlocaldimen\mydimena    \mydimena3\onepoint
  \newlocaldimen\mydimenb    \mydimenb4\onepoint
  \newlocaldimen\mydimenc    \mydimenc5\onepoint
  (\the\mydimena,\the\mydimenb,\the\mydimenc)
\begingroup
  \newlocaldimen\mydimena    \mydimena6\onepoint
  \newlocaldimen\mydimenb    \mydimenb7\onepoint
  (\the\mydimena,\the\mydimenb)
\endgroup
  \newlocaldimen\mydimend    \mydimend8\onepoint
  (\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
\endgroup
  (\the\mydimena,\the\mydimenb)
\endgroup
```

The allocated registers get zero values but you can of course set them to any value that fits their nature:

```
(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)
```

You can also use the next variant where you also pass the initial value:

```
\begingroup
```

```

\setnewlocaldimen\mydimena 1\onepoint
\setnewlocaldimen\mydimenb 2\onepoint
(\the\mydimena,\the\mydimenb)
\begingroup
  \setnewlocaldimen\mydimena 3\onepoint
  \setnewlocaldimen\mydimenb 4\onepoint
  \setnewlocaldimen\mydimenc 5\onepoint
  (\the\mydimena,\the\mydimenb,\the\mydimenc)
\begingroup
  \setnewlocaldimen\mydimena 6\onepoint
  \setnewlocaldimen\mydimenb 7\onepoint
  (\the\mydimena,\the\mydimenb)
\endgroup
  \setnewlocaldimen\mydimend 8\onepoint
  (\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
\endgroup
(\the\mydimena,\the\mydimenb)
\endgroup

```

So, again we get:

```

(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)

```

When used in the body of the macro there is of course a little overhead involved in the repetitive allocation but normally that can be neglected.

10.4 Files

When adding these new allocators I also wondered about the read and write allocators. We don't use them in ConT_EXt but maybe users like them, so let's give an example and see what more demands they have:

```

\integerdef\StartHere\numexpr\inputlineno+2\relax
\starthiding
SOME LINE 1
SOME LINE 2
SOME LINE 3

```

```

SOME LINE 4
\stophiding
\integerdef\StopHere\numexpr\inputlineno-2\relax

\begingroup
  \newlocalread\myreada
  \immediate\openin\myreada {lowlevel-scope.tex}
  \dostepwiserecurse{\StopHere}{\StartHere}{-1}{
    \readline\myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \blank
  \dostepwiserecurse{\StartHere}{\StopHere}{1}{
    \read    \myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \immediate\closein\myreada
\endgroup

```

Here, instead of hard coded line numbers we used the stored values. The optional `line` keyword is a LMTX speciality.

```

281 : SOME LINE 4
280 : SOME LINE 3
279 : SOME LINE 2
278 : SOME LINE 1

278 : SOME LINE 1
279 : SOME LINE 2
280 : SOME LINE 3
281 : SOME LINE 4

```

Actually an application can be found in a small (demonstration) module:

```
\usemodule[system-readers]
```

This provides the code for doing this:

```

\startmarkedlines[test]
SOME LINE 1
SOME LINE 2
SOME LINE 3
\stopmarkedlines

```

```

\begingroup
  \newlocalread\myreada
  \immediate\openin\myreada {\markedfilename{test}}
  \dostepwiserecurse{\lastmarkedline{test}}{\firstmarkedline{test}}{-1}{
    \readline\myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \immediate\closein\myreada
\endgroup

```

As you see in these examples, we can locally define a read channel without getting a message about it already being defined.

10.4 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

11 Paragraphs

low level

TEX

paragraphs

Contents

11.1	Introduction	125
11.2	Paragraphs	125
11.3	Properties	129
11.4	Wrapping up	131
11.5	Hanging	131
11.6	Shapes	132
11.7	Modes	150
11.8	Normalization	150
11.9	Dirty tricks	150

11.1 Introduction

This manual is mostly discussing a few low level wrappers around low level $\text{T}_{\text{E}}\text{X}$ features. Its writing is triggered by an update to the MetaFun and LuaMetaFun manuals where we mess a bit with shapes. It gave a good reason to also cover some more paragraph related topics but it might take a while to complete. Remind me if you feel that takes too much time.

Because paragraphs and their construction are rather central to $\text{T}_{\text{E}}\text{X}$, you can imagine that the engine exposes dealing with them. This happens via commands (primitives) but only when it's robust. Then there are callbacks, and some provide detailed information about what we're dealing with. However, intercepting node lists can already be hairy and we do that a lot in $\text{ConT}_{\text{E}}\text{Xt}$. Intercepting and tweaking paragraph properties is even more tricky, which is why we try to avoid that in the core. But . . . in the following sections you will see that there are actually a couple of mechanism that do so. Often new features like this are built in stepwise and enabled locally for a while and when they seem okay they get enabled by default.¹⁵

11.2 Paragraphs

Before we demonstrate some trickery, let's see what a paragraph is. Normally a document source is formatted like this:

```
some text (line 1)
some text (line 2)
```

¹⁵ For this we have `\enableexperiments` which one can use in `cont-loc.mkxl` or `cont-exp.mkxl`, files that are loaded runtime when on the system. When you use them, make sure they don't interfere; they are not part of the updates, contrary to `cont-new.mkxl`.


```
some more test (line 1)
some more test (line 2)
```

There are two blocks of text here separated by an empty line and they become two paragraphs. Unless configured otherwise an empty line is an indication that we end a paragraph. You can also explicitly do that:

```
some text (line 1)
some text (line 2)
\par
some more test (line 1)
some more test (line 2)
```

When $\text{T}_{\text{E}}\text{X}$ starts a paragraph, it actually also does something think of:

```
[\the\everypar]some text      (line 1) some text      (line 2) \par
[\the\everypar]some more test (line 1) some more test (line 2) \par
```

or more accurate:

```
[\the\everypar]some text      some text      \par
[\the\everypar]some more test some more test \par
```

because the end-of-line character has become a space. As mentioned, an empty line is actually the end of a paragraph. But in LuaMeta $\text{T}_{\text{E}}\text{X}$ we can cheat a bit. If we have this:

```
line 1
line 2
```

We can do this (watch how we need to permit overloading a primitive when we have enabled `\overloadmode`):

```
\pushoverloadmode
\def\linepar{\removeunwantedspaces !\ignorespaces}
\popoverloadmode
line 1
line 2
```

This comes out as:

```
line 1
```

line 2

I admit that since it got added (as part of some cleanup halfway the overhaul of the engine) I never saw a reason to use it, but it is a cheap feature. The `\linepar` primitive is undefined (`\undefined`) by default so no user sees it anyway. Just don't use it unless maybe for some pseudo database trickery (I considered using it for the database module but it is not needed). In a similar fashion, just don't redefine `\par`: it's asking for troubles and 'not done' in ConT_EXt anyway.

Back to reality. In LuaT_EX we get a node list that starts with a so called `localpar` node and ends with a `\parfillskip`. The first node is prepended automatically. That list travels through the system: hyphenation, applying font properties, break the effectively one line into lines, wrap them and add them to a vertical list, etc. Each stage can be intercepted via callbacks.

When the paragraph is broken into lines hanging indentation or a so called par shape can be applied, and we will see more of that later, here we talk `\par` and show another LuaMetaT_EX trick:

```
\def\foo{{\bf test:} \ignorepars}
```

```
\foo
```

line

The macro typesets some text and then skips to the next paragraph:

test: line

Think of this primitive as being a more powerful variant of `\ignorespaces`. This leaves one aspect: how do we start a paragraph. Technically we need to force T_EX into so called horizontal mode. When you look at plain T_EX documents you will notice commands like `\noindent` and `\indent`. In ConT_EXt we have more high level variants, for instance we have `\noindentation`.

A robust way to make sure that you get in horizontal mode is using `\dontleavehmode` which is a wink to `\leavevmode`, a command that you should never use in ConT_EXt, so when you come from plain or L^AT_EX, it's one of the commands you should wipe from your memory.

When T_EX starts with a paragraph the `\everypar` token list is expanded and again this is a primitive you should not mess with yourself unless in very controlled situations. If you change its content, you're on your own with respect to interferences and side effects.

Paragraphs

One of the things that \TeX does in injecting the indentation. Even when there is none, it gets added, not as skip but as an empty horizontal box of a certain width. This is easier on the engine when it constructs the paragraph from the one liner: starting with a skip demands a bit more testing in the process (a nice trick so to say). However, in $\text{Con}\TeX$ t we enable the $\text{LuaMeta}\TeX$ feature that does use a skip instead of a box. It's part of the normalization that is discussed later. Instead of checking for a box with property `indent`, we check for a skip with such property. This is often easier and cleaner.

A bit off topic is the fact that in traditional \TeX empty lines or `\par` primitives can trigger an error. This has to do with the fact that the program evolved in a time where paper terminals were used and runtime could be excessive. So, in order to catch a possible missing brace, a concept of `\long` macros, permitting `\par` or equivalents in arguments, was introduced as well as not permitting them in for instance `display math`. In $\text{Con}\TeX$ t MkII most macros that could be sensitive for this were defined as `\long` so that users never had to bother about it and probably were not even aware of it. Right from the start in $\text{Lua}\TeX$ these error-triggers could be disabled which of course we enable in $\text{Con}\TeX$ t and in $\text{LuaMeta}\TeX$ these features have been removed altogether. I don't think users will complain about this.

If you want to enforce a newline but not a new paragraph you can use the `\crlf` command. When used on its own it will produce an empty line. Don't use this to create whitespace between lines.

If you want to do something after so called par tokens are seen you can do this:

```
\def\foo{{\bf >>>> }}
\expandafterpars\foo
```

this is a new paragraph ...

```
\expandafterpars\foo
\par\par\par\par
```

this is a new paragraph ...

This not to be confused with `\everypar` which is a token list that \TeX itself injects before each paragraph (also nested ones).

```
>>>> this is a new paragraph ...
```

```
>>>> this is a new paragraph ...
```

This is typically a primitive that will only be used in macros. You can actually program it using macros: pickup a token, check and push it back when it's not a par equivalent token. The primitive is just nicer (and easier on the log when tracing is enabled).

11.3 Properties

A paragraph is just a collection of lines that result from one input line that got broken. This process of breaking into lines is influenced by quite some parameters. In traditional \TeX and also in \LuaMetaTeX by default the values that are in effect when the end of the paragraph is met are used. So, when you change them in a group and then ends the paragraph after the group, the values you've set in the group are not used.

However, in \LuaMetaTeX we can optionally store them with the paragraph. When that happens the values current at the start are frozen. You can still overload them but that has to be done explicitly then. The advantage is that grouping no longer interferes with the line break algorithm. The magic primitive is `\snapshotpar` which takes a number made from categories mentioned below:

variable	category	code
<code>\hsize</code>	hsize	0x01
<code>\leftskip</code>	skip	0x02
<code>\rightskip</code>	skip	0x02
<code>\hangindent</code>	hang	0x04
<code>\hangafter</code>	hang	0x04
<code>\parindent</code>	indent	0x08
<code>\parfillleftskip</code>	par fill	0x10
<code>\parfillrightskip</code>	par fill	0x10
<code>\adjustspacing</code>	adjust	0x20
<code>\adjustspacingstep</code>	adjust	0x20
<code>\adjustspacingshrink</code>	adjust	0x20
<code>\adjustspacingstretch</code>	adjust	0x20
<code>\protrudechars</code>	protrude	0x40
<code>\pretolerance</code>	tolerance	0x80
<code>\tolerance</code>	tolerance	0x80
<code>\emergencystretch</code>	stretch	0x100
<code>\looseness</code>	looseness	0x200
<code>\lastlinefit</code>	last line	0x400
<code>\linepenalty</code>	line penalty	0x800
<code>\interlinepenalty</code>	line penalty	0x800
<code>\interlinepenalties</code>	line penalty	0x800
<code>\clubpenalty</code>	club penalty	0x1000
<code>\clubpenalties</code>	club penalty	0x1000
<code>\widowpenalty</code>	widow penalty	0x2000
<code>\widowpenalties</code>	widow penalty	0x2000
<code>\displaywidowpenalty</code>	display penalty	0x4000

<code>\displaywidowpenalties</code>	display penalty	0x4000
<code>\brokenpenalty</code>	broken penalty	0x8000
<code>\adjdemerits</code>	demerits	0x10000
<code>\doublehyphendemerits</code>	demerits	0x10000
<code>\finalhyphendemerits</code>	demerits	0x10000
<code>\parshape</code>	shape	0x20000
<code>\baselineskip</code>	line	0x40000
<code>\lineskip</code>	line	0x40000
<code>\lineskiplimit</code>	line	0x40000
<code>\hyphenationmode</code>	hyphenation	0x80000

As you can see here, there are more paragraph related parameters than in for instance pdf \TeX and Lua \TeX and these are (to be) explained in the LuaMeta \TeX manual. You can imagine that keeping this around with the paragraph adds some extra overhead to the machinery but most users won't notice that because it is compensated by gains elsewhere.

This is pretty low level and there are a bunch of helpers that support this but these are not really user level macros. As with everything \TeX you can mess around as much as you like, and the code gives plenty of examples but when you do this, you're on your own because it can interfere with Con \TeX t core functionality.

In LMTX taking these snapshots is turned on by default and because it thereby fundamentally influences the par builder, users can run into compatibility issues but in practice there has been no complaints (and this feature has been in use quite a while before this document was written). One reason for users not noticing is that one of the big benefits is probably handled by tricks mentioned on the mailing list. Imagine that you have this:

```
{\bf watch out:} here is some text
```

In this small example the result will be as expected. But what if something magic with the start of a paragraph is done? Like this:

```
\placefigure[left]{A cow!}{\externalfigure[cow.pdf]}
```

```
{\bf watch out:} here is some text ... of course much more is needed to
  get a flow around the figure!
```

The figure will hang at the left side of the paragraph but it is put there when the text starts and that happens inside the bold group. It means that the properties we set in order to get the shape around the figure are lost as soon as we're at 'here is some text' and definitely is wrong when the paragraph ends and the par builder has to use

Properties

them to get the shape right. We get text overlapping the figure. A trick to overcome this is:

```
\dontleavehmode {\bf watch out:} here is some text ... of course much
    more is needed to get a flow around the figure!
```

where the first macro makes sure we already start a paragraph before the group is entered (using a `\strut` also works). It's not nice and I bet users have been bitten by this and by now know the tricks. But, with snapshots such fuzzy hacks are not needed any more! The same is true with this:

```
{\leftskip lem some text \par}
```

where we had to explicitly end the paragraph inside the group in order to retain the skip. I suppose that users normally use the high level environments so they never had to worry about this. It's also why users probably won't notice that this new mechanism has been active for a while. Actually, when you now change a parameter inside the paragraph its new value will not be applied (unless you prefix it with `\frozen` or snapshot it) but no one did that anyway.

11.4 Wrapping up

In ConT_EXt LMTX we have a mechanism to exercise macros (or content) before a paragraph ends. This is implemented using the `\wrapuppar` primitive. The to be wrapped up material is bound to the current paragraph which in order to get this done has to be started when this primitive is used.

Although the high level interface has been around for a while it still needs a bit more testing (read: use cases are needed). In the few cases where we already use it application can be different because again it relates to snapshots. This because in the past we had to use tricks that also influenced the user interface of some macros (which made them less natural as one would expect). So the question is: where do we apply it in old mechanisms and where not.

todo: accumulation, interference, where applied, limitations

11.5 Hanging

There are two mechanisms for getting a specific paragraph shape: rectangular hanging and arbitrary shapes. Both mechanisms work top-down. The first mechanism

uses a combination of `\hangafter` and `\hangindent`, and the second one depends on `\parshape`. In this section we discuss the rectangular one.

```
\hangafter 4 \hangindent 4cm \samplefile{tufte} \page
\hangafter -4 \hangindent 4cm \samplefile{tufte} \page
\hangafter 4 \hangindent -4cm \samplefile{tufte} \page
\hangafter -4 \hangindent -4cm \samplefile{tufte} \page
```

As you can see in figure 11.1, the four cases are driven by the sign of the values. If you want to hang into the margin you need to use different tricks, like messing with the `\leftskip`, `\rightskip` or `\parindent` parameters (which then of course can interfere with other mechanisms uses at the same time).

11.6 Shapes

In ConT_EXt we don't use `\parshape` a lot. It is used in for instance side floats but even there not in all cases. It's more meant for special applications. This means that in MkII and MkIV we don't have some high level interface. However, when MetaFun got upgraded to LuaMetaFun, and the manual also needed an update, one of the examples in that manual that used shapes also got done differently (read: nicer). And that triggered the arrival of a new low level shape mechanism.

One important property of the `\parshape` mechanism is that it works per paragraph. You define a shape in terms of a left margin and width of a line. The shape has a fixed number of such pairs and when there is more content, the last one is used for the rest of the lines. When the paragraph is finished, the shape is forgotten.¹⁶

The high level interface is a follow up on the example in the MetaFun manual and uses shapes that carry over to the next paragraph. In addition we can cycle over a shape. In this interface shapes are defined using keyword. Here are some examples:

```
\startparagraphshape[test]
  left 1mm right 1mm
  left 5mm right 5mm
\stopparagraphshape
```

This shape has only two entries so the first line will have a 1mm margin while later lines will get 5mm margins. This translates into a `\parshape` like:

```
\parshape 2_____
```

¹⁶ Not discussed here is a variant that might end up in LuaMetaT_EX that works with the progression, i.e. takes the height of the content so far into account. This is somewhat tricky because for that to work vertical skips need to be frozen, which is no real big deal but has to be done careful in the code.



Figure 11.1 Hanging indentation


```
1mm \dimexpr\hsize-1mm\relax
5mm \dimexpr\hsize-5mm\relax
```

Watch the number 2: it tells how many specification lines follow. As you see, we need to calculate the width.

```
\startparagraphshape[test]
  left 1mm right 1mm
  left 5mm right 5mm
  repeat
\stopparagraphshape
```

This variant will alternate between 1mm and 5mm margins. The repeating feature is translated as follows. Maybe at some point I will introduce a few more options.

```
\parshape 2 options 1
  1mm \dimexpr\hsize-1mm\relax
  5mm \dimexpr\hsize-5mm\relax
```

A shape can have some repetition, and we can save keystrokes by copying the last entry. The resulting `\parshape` becomes rather long.

```
\startparagraphshape[test]
  left 1mm right 1mm
  left 2mm right 2mm
  left 3mm right 3mm
  copy 8
  left 4mm right 4mm
  left 5mm right 5mm
  left 5mm hsize 10cm
\stopparagraphshape
```

Also watch the `hsize` keyword: we don't calculate the `hsize` from the left and right values but explicitly set it.

```
\startparagraphshape[test]
  left 1mm right 1mm
  right 3mm
  left 5mm right 5mm
  repeat
\stopparagraphshape
```

When a right keywords comes first the left is assumed to be zero. In the examples that follow we will use a couple of definitions:

```
\startparagraphshape[test]
  both 1mm both 2mm both 3mm both 4mm both 5mm both 6mm
  both 7mm both 6mm both 5mm both 4mm both 3mm both 2mm
\stopparagraphshape
```

```
\startparagraphshape[test-repeat]
  both 1mm both 2mm both 3mm both 4mm both 5mm both 6mm
  both 7mm both 6mm both 5mm both 4mm both 3mm both 2mm
  repeat
\stopparagraphshape
```

The last one could also be defines as:

```
\startparagraphshape[test-repeat]
  \rawparagraphshape{test} repeat
\stopparagraphshape
```

In the previous code we already introduced the repeat option. This will make the shape repeat at the engine level when the shape runs out of specified lines. In the application of a shape definition we can specify a method to be used and that determine if the next paragraph will start where we left off and discard afterwards (shift) or that we move the discarded lines up front so that we never run out of lines (cycle). It sounds complicated but just keep in mind that repeat is part of the `\parshape` and act within a paragraph while shift and cycle are applied when a new paragraph is started.

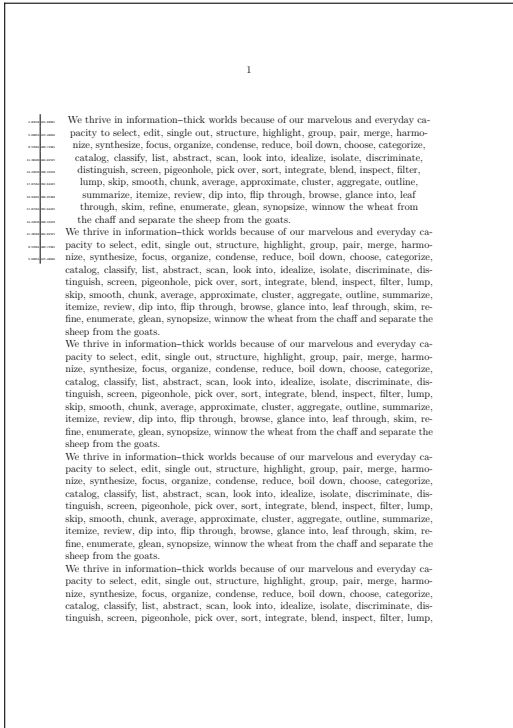
In figure 11.2 you see the following applied:

```
\startshapedparagraph[list=test]
  \dorecuse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

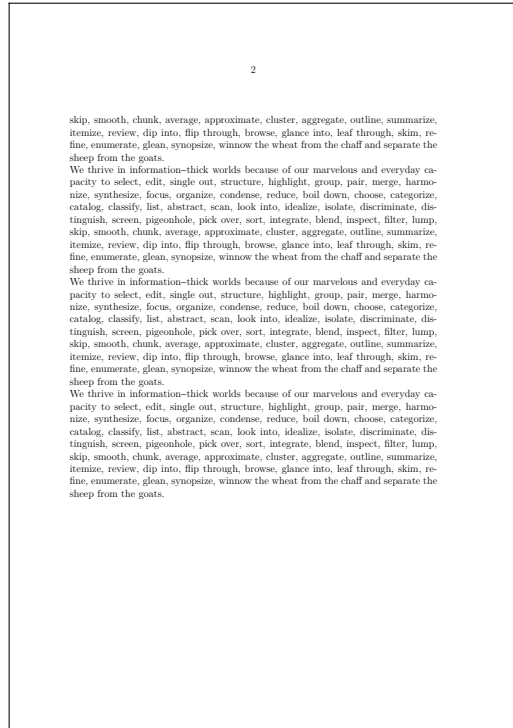
```
\startshapedparagraph[list=test-repeat]
  \dorecuse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

In figure 11.3 we use this instead:

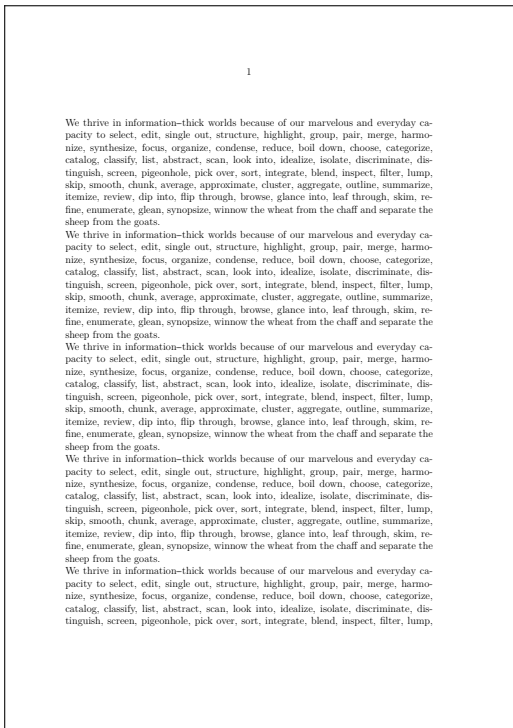
```
\startshapedparagraph[list=test,method=shift]
  \dorecuse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```



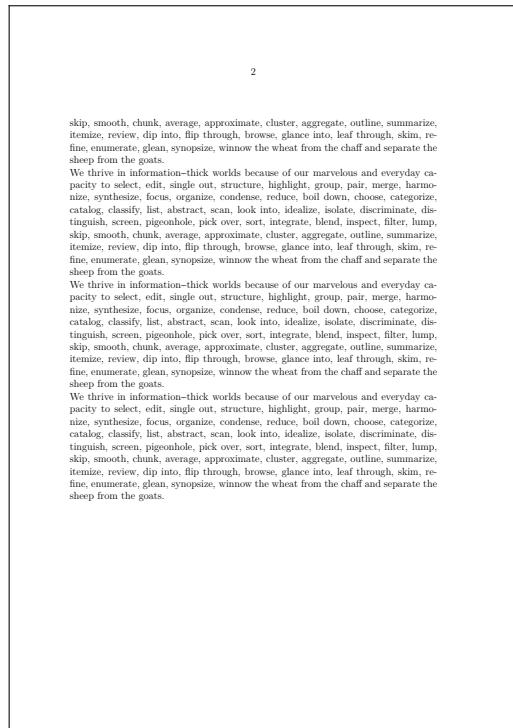
discard, finite shape, page 1



discard, finite shape, page 2



discard, repeat in shape, page 1



discard, repeat in shape, page 2

Figure 11.2 Discarded shaping

Finally, in figure 11.4 we use:

```
\startshapedparagraph[list=test,method=cycle]
  \dorecurse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

These examples are probably too small to see the details but you can run them yourself or zoom in on the details. In the margin we show the values used. Here is a simple example of (non) poetry. There are other environments that can be used instead but this makes a good example anyway.

```
\startparagraphshape[test]
  left 0em right 0em
  left 1em right 0em
  repeat
\stopparagraphshape
```

```
\startshapedparagraph[list=test,method=cycle]
  verse line 1.1\crlf verse line 2.1\crlf
  verse line 3.1\crlf verse line 4.1\par
  verse line 1.2\crlf verse line 2.2\crlf
  verse line 3.2\crlf verse line 4.2\crlf
  verse line 5.2\crlf verse line 6.2\par
\stopshapedparagraph
```

```
verse line 1.1
  verse line 2.1
verse line 3.1
  verse line 4.1

verse line 1.2
  verse line 2.2
verse line 3.2
  verse line 4.2
verse line 5.2
  verse line 6.2
```

Because the idea for this feature originates in MetaFun, we will now kick in some MetaPost. The following code creates a shape for a circle. We use a 2mm offset here:

```
\startuseMPgraphic{circle}
  path p ; p := fullcircle scaled TextWidth ;
  build_parshape(p,
```



```

    2mm, 0, 0,
    LineHeight, StrutHeight, StrutDepth, StrutHeight
  ) ;
\stopuseMPgraphic

```

We plug this into the already described macros:

```

\startshapedparagraph[mp=circle]%
  \setupalign[verytolerant,stretch,last]%
  \samplefile{tufte}
  \samplefile{tufte}
\stopshapedparagraph

```

And get ourself a circular shape. Watch out, at this moment the shape environment does not add grouping so when as in this case you change the alignment it can influence the document.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

Assuming that the shape definition above is in a buffer we can do this:

```

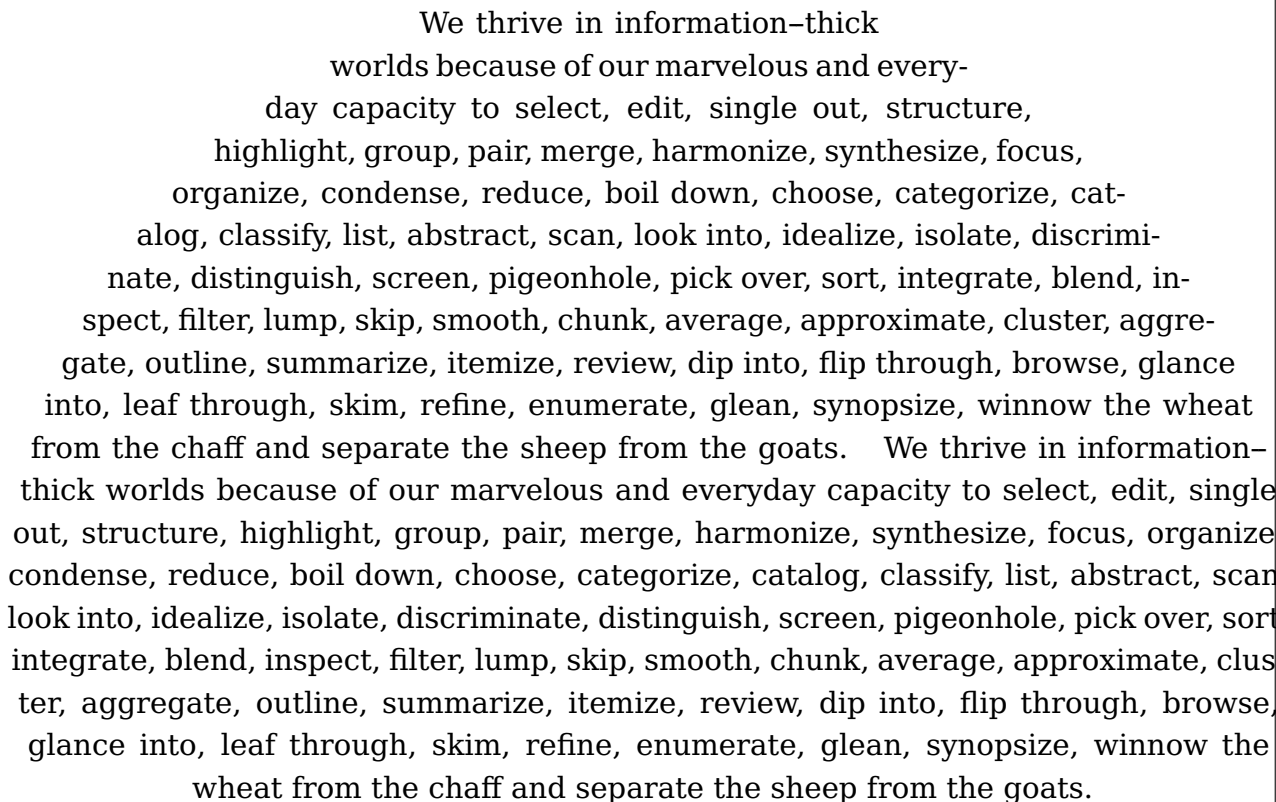
\startshapedparagraph[mp=circle]%
  \setupalign[verytolerant,stretch,last]%
  \samplefile{tufte}

```

```
\samplefile{tufte}
\stopshapedparagraph
```

The result is shown in figure 11.5. Because all action happens in the framed environment, we can also use this definition:

```
\startuseMPgraphic{circle}
  path p ; p := fullcircle scaled \the\dimexpr\framedwidth+\framedoffset
    *2\relax ;
  build_parshape(p,
    \framedoffset, 0, 0,
    LineHeight, StrutHeight, StrutDepth, StrutHeight
  ) ;
  draw p ;
\stopuseMPgraphic
```



We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

Figure 11.5 A framed circular shape

A mechanism like this is often never completely automatic in the sense that you need to keep an eye on the results. Depending on user demands more features can be added. With weird shapes you might want to set up the alignment to be tolerant and have

some stretch.

The interface described in the MetaFun manual is pretty old, the time stamp of the original code is mid 2000, but the principles didn't change. The examples in `meta-imp-txt.mkx` can now be written as:

```
\startshapetext[test 1,test 2,test 3,test 4]
  \setupalign[verytolerant,stretch,normal]%
  \samplefile{douglas} % Douglas R. Hofstadter
\stopshapetext
\startcombination[2*2]
  {\framed[offset=overlay,frame=off,background=test 1]{\getshapetext}}
  {test 1}
  {\framed[offset=overlay,frame=off,background=test 2]{\getshapetext}}
  {test 2}
  {\framed[offset=overlay,frame=off,background=test 3]{\getshapetext}}
  {test 3}
  {\framed[offset=overlay,frame=off,background=test 4]{\getshapetext}}
  {test 4}
\stopcombination
```

In figure 11.6 we see the result. Watch how for two shapes we have enabled tracing. Of course you need to tweak till all fits well but we're talking of special situations anyway.

Here is a bit more extreme example. Again we use a circle:

```
\startuseMPgraphic{circle}
  lmt_parshape [
    path      = fullcircle scaled 136mm,
    offset    = 2mm,
    bottomskip = - 1.5LineHeight,
  ] ;
\stopuseMPgraphic
```

But we output a longer text:

```
\startshapedparagraph[mp=circle,repeat=yes,method=cycle]%
  \setupalign[verytolerant,stretch,last]\dontcomplain
  {\darkred      \samplefile{tufte}}\par
  {\darkgreen    \samplefile{tufte}}\par
  {\darkblue     \samplefile{tufte}}\par
  {\darkcyan     \samplefile{tufte}}\par
```

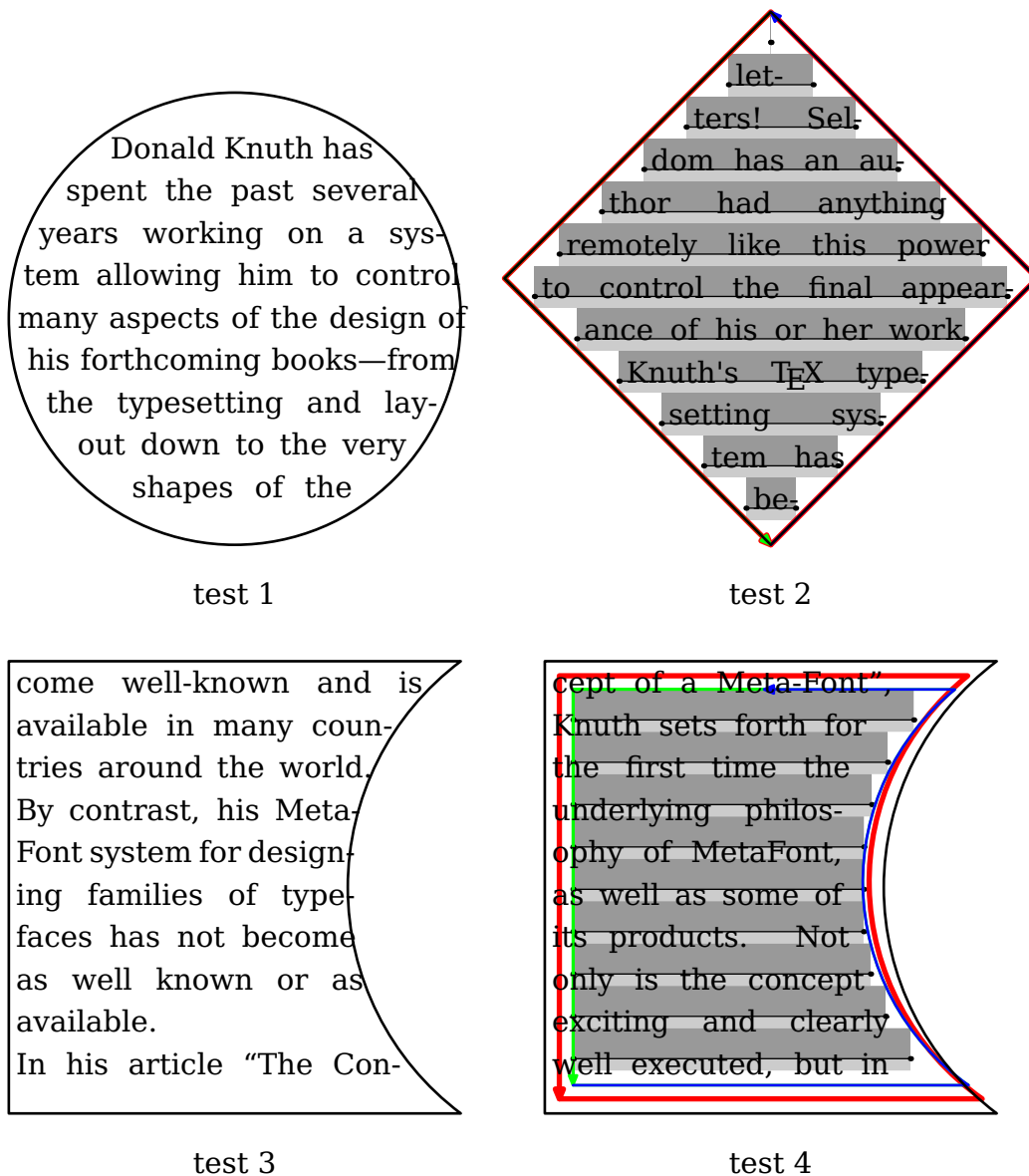


Figure 11.6

```
{\darkmagenta \samplefile{tufte}}\par
\stopshapedparagraph
```

We get a multi-page shape:

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discrimi-

nate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

size, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

Compare this with:

```
\startshapedparagraph[mp=circle,repeat=yes,method=cycle]%
  \setupalign[verytolerant,stretch,last]\dontcomplain
  {\darkred      \samplefile{tufte}}
  {\darkgreen    \samplefile{tufte}}
  {\darkblue     \samplefile{tufte}}
  {\darkcyan     \samplefile{tufte}}
  {\darkmagenta  \samplefile{tufte}}
\stopshapedparagraph
```

Which gives:

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair,

merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synthesize, winnow the wheat from the chaff and separate the sheep from the goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synthesize, winnow the wheat from the chaff and separate the sheep from the goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synthesize, winnow the wheat from the chaff and separate the sheep from the goats.

Here the `bottomskip` takes care of subtle rounding issues as well as discarding the last line in the shape so that we get nicer continuation. There is no full automated solution for all you can come up with.

Mixing a MetaPost specification into a regular one is also possible. The next example demonstrates this as well as the option to remove some lines from a specification:

```
\startparagraphshape[test]
  left 0em right 0em
  left 1em right 0em
  metapost {circle}
```

```

delete 3
metapost {circle,circle,circle}
delete 7
metapost {circle}
repeat
\stopparagraphshape

```

You can combine a shape with narrowing a paragraph. Watch the absolute keyword in the next code. The result is shown in figure 11.7.

```

\startuseMPgraphic{circle}
  lmt_parshape [
    path      = fullcircle scaled TextWidth,
    bottomskip = - 1.5LineHeight,
  ] ;
\stopuseMPgraphic

\startparagraphshape[test-1]
  metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-2]
  absolute left metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-3]
  absolute right metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-4]
  absolute both metapost {circle} repeat
\stopparagraphshape

\showframe

\startnarrower[4*left,2*right]
  \startshapedparagraph[list=test-1,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
  \stopshapedparagraph
  \page
  \startshapedparagraph[list=test-2,repeat=yes,method=repeat]%

```

```

    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
\stopshapedparagraph
\page
\startshapedparagraph[list=test-3,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
\stopshapedparagraph
\page
\startshapedparagraph[list=test-4,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
\stopshapedparagraph
\stopnarrower

```

The shape mechanism has a few more tricks but these are really meant for usage in specific situations, where one knows what one deals with. The following examples are visualized in figure 11.8.

```

\useMPlibrary[dum]
\usemodule[article-basics]

\startbuffer
    \externalfigure[dummy][width=6cm]
\stopbuffer

\startshapedparagraph[text=\getbuffer]
    \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em]
    \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em,
    hoffset=-2em]
    \dorecurse{3}{\samplefile{ward}\par}

```



```
\stopshapedparagraph
```

```
\page
```

```
\startshapedparagraph[text=\getbuffer,distance=1em,
  voffset=-2ex,hoffset=-2em]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph
```

```
\page
```

```
\startshapedparagraph[text=\getbuffer,distance=1em,
  voffset=-2ex,hoffset=-2em,lines=1]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph
```

```
\page
```

```
\startshapedparagraph[width=4cm,lines=4]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph
```

11.7 Modes

todo: some of the side effects of so called modes

11.8 Normalization

todo: users don't need to bother about this but it might be interesting anyway

11.9 Dirty tricks

todo: explain example for combining paragraphs



Figure 11.8 Flow around something

11.9 Colofon

Author Hans Hagen
 ConT_EXt 2021.09.06 11:47
 LuaMetaT_EX 2.0923
 Support www.pragma-ade.com
contextgarden.net

12 Alignments

low level

TEX

alignments

Contents

12.1	Introduction	153
12.2	Between the lines	155
12.3	Pre-, inter- and post-tab skips	157
12.4	Cell widths	160
12.5	Plugins	161
12.6	Pitfalls and tricks	164
12.7	Remark	167

12.1 Introduction

\TeX has a couple of subsystems and alignments is one of them. This mechanism is used to construct tables or alike. Because alignments use low level primitives to set up and construct a table, and because such a setup can be rather extensive, in most cases users will rely on macros that hide this.

```
\halign {
    \alignmark\hss \aligntab
  \hss\alignmark\hss \aligntab
  \hss\alignmark      \cr
  1.1      \aligntab 2,2      \aligntab 3=3      \cr
  11.11    \aligntab 22,22    \aligntab 33=33    \cr
  111.111  \aligntab 222,222  \aligntab 333=333  \cr
}
```

That one doesn't look too complex and comes out as:

```
1.1      2,2      3=3
11.11    22,22    33=33
111.111  222,222  333=333
```

This is how the previous code comes out when we use one of the $\text{Con}\text{\TeX}$ t table mechanism.

```
\starttabulate[|l|c|r|]
  \NC 1.1      \NC 2,2      \NC 3=3      \NC \NR
  \NC 11.11    \NC 22,22    \NC 33=33    \NC \NR
  \NC 111.111  \NC 222,222  \NC 333=333  \NC \NR
\stoptabulate
```

```
1.1      2,2      3=3
11.11    22,22    33=33
111.111  222,222  333=333
```

That one looks a bit different with respect to spaces, so let's go back to the low level variant:

```
\halign {
    \alignmark\hss \aligntab
    \hss\alignmark\hss \aligntab
    \hss\alignmark \cr
    1.1\aligntab    2,2\aligntab    3=3\cr
    11.11\aligntab  22,22\aligntab  33=33\cr
    111.111\aligntab 222,222\aligntab 333=333\cr
}
```

Here we don't have spaces in the content part and therefore also no spaces in the result:

```
1.1      2,2      3=3
11.11    22,22    33=33
111.111222,222333=333
```

You can automate dealing with unwanted spacing:

```
\halign {
    \ignorespaces\alignmark\unskip\hss \aligntab
    \hss\ignorespaces\alignmark\unskip\hss \aligntab
    \hss\ignorespaces\alignmark\unskip \cr
    1.1 \aligntab 2,2 \aligntab 3=3 \cr
    11.11 \aligntab 22,22 \aligntab 33=33 \cr
    111.111 \aligntab 222,222 \aligntab 333=333 \cr
}
```

We get:

```
1.1      2,2      3=3
11.11    22,22    33=33
111.111222,222333=333
```

By moving the space skipping and cleanup to the so called preamble we don't need to deal with it in the content part. We can also deal with inter-column spacing there:

```
\halign {
```

```

\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
\hss\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
\hss\ignorespaces\alignmark\unskip \tabskip 0pt \cr
1.1 \aligntab 2,2 \aligntab 3=3 \cr
11.11 \aligntab 22,22 \aligntab 33=33 \cr
111.111 \aligntab 222,222 \aligntab 333=333 \cr
}

```

```

1.1      2,2      3=3
11.11    22,22    33=33
111.111  222,222  333=333

```

If for the moment we forget about spanning columns (`\span`) and locally ignoring preamble entries (`\omit`) these basic commands are not that complex to deal with. Here we use `\alignmark` but that is just a primitive that we use instead of `#` while `\aligntab` is the same as `&`, but using the characters instead also assumes that they have the catcode that relates to a parameter and alignment tab (and in `ConTeXt` that is not the case). The `TeXbook` has plenty alignment examples so if you really want to learn about them, consult that must-have-book.

12.2 Between the lines

The individual rows of a horizontal alignment are treated as lines. This means that, as we see in the previous section, the interline spacing is okay. However, that also means that when we mix the lines with rules, the normal `TeX` habits kick in. Take this:

```

\halign {
\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
\hss\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
\hss\ignorespaces\alignmark\unskip \tabskip 0pt \cr
\noalign{\hrule}
1.1 \aligntab 2,2 \aligntab 3=3 \cr
\noalign{\hrule}
11.11 \aligntab 22,22 \aligntab 33=33 \cr
\noalign{\hrule}
111.111 \aligntab 222,222 \aligntab 333=333 \cr
\noalign{\hrule}
}

```

The result doesn't look pretty and actually, when you see documents produced by `TeX` using alignments you should not be surprised to notice rather ugly spacing. The user

(or the macropackage) should deal with that explicitly, and this is not always the case.

1.1	2,2	3=3
11.11	22,22	33=33
111.111	222,222	333=333

The solution is often easy:

```
\halign {
  \ignorespaces\strut\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\strut\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\strut\alignmark\unskip \tabskip 0pt \cr
  \noalign{\hrule}
  1.1 \aligntab 2,2 \aligntab 3=3 \cr
  \noalign{\hrule}
  11.11 \aligntab 22,22 \aligntab 33=33 \cr
  \noalign{\hrule}
  111.111 \aligntab 222,222 \aligntab 333=333 \cr
  \noalign{\hrule}
}
```

1.1	2,2	3=3
11.11	22,22	33=33
111.111	222,222	333=333

The user will not notice it but alignments put some pressure on the general $\text{T}_{\text{E}}\text{X}$ scanner. Actually, the scanner is either scanning an alignment or it expects regular text (including math). When you look at the previous example you see `\noalign`. When the preamble is read, $\text{T}_{\text{E}}\text{X}$ will pick up rows till it finds the final brace. Each row is added to a temporary list and the `\noalign` will enter a mode where other stuff gets added to that list. It all involves subtle look ahead but with minimal overhead. When the whole alignment is collected a final pass over that list will package the cells and rows (lines) in the appropriate way using information collected (like the maximum width of a cell and width of the current cell. It will also deal with spanning cells then.

So let's summarize what happens:

1. scan the preamble that defines the cells (where the last one is repeated when needed)
2. check for `\cr`, `\noalign` or a right brace; when a row is entered scan for cells in parallel the preamble so that cell specifications can be applied (then start again)
3. package the preamble based on information with regards to the cells in a column
4. apply the preamble packaging information to the columns and also deal with pending cell spans

5. flush the result to the current list

The second (repeated) step is complicated by the fact that the scanner has to look ahead for a `\noalign`, `\cr`, `\omit` or `\span` and when doing that it has to expand what comes. This can give side effects and often results in obscure error messages. When for instance an `\if` is seen and expanded, the wrong branch can be entered. And when you use protected macros embedded alignment commands are not seen at all. Also, nesting `\noalign` is not permitted.

All these side effects are to be handled in a macro package when it wraps alignments in a high level interface and `ConTeXt` does that for you. But because the code doesn't always look pretty then, in `LuaMetaTeX` the alignment mechanism has been extended a bit over time.

The first extension was to permit nested usage of `\noalign`. This has resulted of a little reorganization of the code. A next extension showed up when overload protection was introduced and extra prefixes were added. We can signal the scanner that a macro is actually a `\noalign` variant:¹⁷

```
\noaligned\protected\def\InBetween{\noalign{...}}
```

This extension resulted in a second bit of reorganization (think of internal command codes and such) but still the original processing of alignments was there.

A third overhaul of the code actually did lead to some adaptations in the way alignments are constructed so let's move on to that.

12.3 Pre-, inter- and post-tab skips

The basic structure of a preamble and row is actually not that complex: it is a mix of tab skip glue and cells (that are just boxes):

```
\tabskip 10pt
\halign {
  \strut\alignmark\tabskip 12pt\aligntab
  \strut\alignmark\tabskip 14pt\aligntab
  \strut\alignmark\tabskip 16pt\cr
  \noalign{\hrule}
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  \noalign{\hrule}
```

¹⁷ A better prefix would have been `\peekaligned` because in the meantime other alignment primitives also can use this property.

```
cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
\noalign{\hrule}
}
```

The tab skips are set in advance and apply to the next cell (or after the last one).

```
cell 1.1 cell 1.2 cell 1.3
TB:10.000 SP:3.497 TB:12.000 SP:3.497 TB:14.000 SP:3.497 TB:16.000
cell 2.1 cell 2.2 cell 2.3
TB:10.000 SP:3.497 TB:12.000 SP:3.497 TB:14.000 SP:3.497 TB:16.000
```

In the ConT_EXt table mechanisms the value of `\tabskip` is zero in most cases. As in:

```
\tabskip 0pt
\halign {
\strut\alignmark\aligntab
\strut\alignmark\aligntab
\strut\alignmark\cr
\noalign{\hrule}
cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
\noalign{\hrule}
cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
\noalign{\hrule}
}
```

When these skips are zero, they still show up in the end:

```
cell 1.1 cell 1.2 cell 1.3
TB:0.000 SP:3.497 TB:0.000 SP:3.497 TB:0.000 SP:3.497 TB:0.000
cell 2.1 cell 2.2 cell 2.3
TB:0.000 SP:3.497 TB:0.000 SP:3.497 TB:0.000 SP:3.497 TB:0.000
```

Normally, in order to achieve certain effects there will be more align entries in the preamble than cells in the table, for instance because you want vertical lines between cells. When these are not used, you can get quite a bit of empty boxes and zero skips.

Now, of course this is seldom a problem, but when you have a test document where you want to show font properties in a table and that font supports a script with some ten thousand glyphs, you can imagine that it accumulates and in LuaTeX (and LuaMetaTeX) nodes are larger so it is one of these cases where in ConTeXt we get messages on the console that node memory is bumped.

After playing a bit with stripping zero tab skips I found that the code would not really benefit from such a feature: lots of extra tests made if quite ugly. As a result a first alternative was to just strip zero skips before an alignment got flushed. At least we're then a bit leaner in the processes that come after it. This feature is now available as one of the normalizer bits.

But, as we moved on, a more natural approach was to keep the skips in the preamble, because that is where a guaranteed alternating skip/box is assumed. It also makes that the original documentation is still valid. However, in the rows construction we can be lean. This is driven by a keyword to `\halign`:

```
\tabskip 0pt
\halign noskips {
  \strut\alignmark\aligntab
  \strut\alignmark\aligntab
  \strut\alignmark\cr
  \noalign{\hrule}
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  \noalign{\hrule}
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
  \noalign{\hrule}
}
```

No zero tab skips show up here:

cell_{SP:3 497}1.1 cell_{SP:3 497}1.2 cell_{SP:3 497}1.3

cell_{SP:3 497}2.1 cell_{SP:3 497}2.2 cell_{SP:3 497}2.3

When playing with all this the LuaMetaTeX engine also got a tracing option for alignments. We already had one that showed some of the `\noalign` side effects, but showing

the preamble was not yet there. This is what `\tracingalignments = 2` results in:

```
<preamble>
\glue[ignored][...] 0.0pt
\alignrecord
..{\strut }
..<content>
..\endtemplate }
\glue[ignored][...] 0.0pt
\alignrecord
..{\strut }
..<content>
..\endtemplate }
\glue[ignored][...] 0.0pt
\alignrecord
..{\strut }
..<content>
..\endtemplate }
\glue[ignored][...] 0.0pt
```

The ignored subtype is (currently) only used for these alignment tab skips and it triggers a check later on when the rows are constructed. The `<content>` is what get injected in the cell (represented by `\alignmark`). The pseudo primitives are internal and not public.

12.4 Cell widths

Imagine this:

```
\halign {
  x\hbox to 3cm{\strut \alignmark\hss}\aligntab
  x\hbox to 3cm{\strut\hss\alignmark\hss}\aligntab
  x\hbox to 3cm{\strut\hss\alignmark } \cr
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
}
```

which renders as:

xcell 1.1	x	cell 1.2	x	cell 1.3
xcell 2.1	x	cell 2.2	x	cell 2.3

A reason to have boxes here is that it enforces a cell width but that is done at the cost of an extra wrapper. In LuaMetaTeX the `hlist` nodes are rather large because we have more options than in original TeX, for instance offsets and orientation. So, in a table with 10K rows of 4 cells yet get 40K extra `hlist` nodes allocated. Now, one can argue that we have plenty of memory but being lazy is not really a sign of proper programming.

```
\halign {
  x\tabsize 3cm\strut   \alignmark\hss\aligntab
  x\tabsize 3cm\strut\hss\alignmark\aligntab
  x\tabsize 3cm\strut\hss\alignmark\hss\cr
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
}
```

If you look carefully you will see that this time we don't have the embedded boxes:

xcell 1.1	x	cell 1.2x	cell 1.3
xcell 2.1	x	cell 2.2x	cell 2.3

So, both the sparse skip and new `\tabsize` feature help to make these extreme tables (spanning hundreds of pages) not consume irrelevant memory and also make that later on we don't have to consult useless nodes.

12.5 Plugins

Yet another LuaMetaTeX extension is a callback that kicks in between the preamble pre-roll and finalizing the alignment. Initially as test and demonstration a basic character alignment feature was written but that works so well that in some places it can replace (or compliment) the already existing features in the ConTeXt table mechanisms.

```
\starttabulate[ | lG{.} | cG{,} | rG{=} | cG{x} | ]
\NC 1.1      \NC 2,2      \NC 3=3      \NC a 0xFF   \NC \NR
\NC 11.11   \NC 22,22   \NC 33=33   \NC b 0xFFF  \NC \NR
\NC 111.111 \NC 222,222 \NC 333=333 \NC c 0xFFFF \NC \NR
\stoptabulate
```

The tabulate mechanism in ConT_EXt is rather old and stable and it is the preferred way to deal with tabular content in the text flow. However, adding the G specifier (as variant of the g one) could be done without interference or drop in performance. This new G specifier tells the tabulate mechanism that in that column the given character is used to vertically align the content that has this character.

```

1.1      2,2      3=3      a 0xFF
11.11    22,22    33=33    b 0xFFF
111.111  222,222  333=333  c 0xFFFF

```

Let's make clear that this is *not* an engine feature but a ConT_EXt one. It is however made easy by this callback mechanism. We can of course use this feature with the low level alignment primitives, assuming that you tell the machinery that the plugin is to be kicked in.

```

\halign noskips \alignmentcharactertrigger \bgroup
  \tabskip2em
  \setalignmentcharacter.\ignorespaces\alignmark\unskip\hss \aligntab
\hss\setalignmentcharacter,\ignorespaces\alignmark\unskip\hss \aligntab
\hss\setalignmentcharacter=\ignorespaces\alignmark\unskip \aligntab
\hss \ignorespaces\alignmark\unskip\hss \cr
1.1 \aligntab 2,2 \aligntab 3=3 \aligntab \setalignmentcharacter{.}\relax 4.4\cr
11.11 \aligntab 22,22 \aligntab 33=33 \aligntab \setalignmentcharacter{,}\relax 44,44\cr
111.111 \aligntab 222,222 \aligntab 333=333 \aligntab \setalignmentcharacter{!}\relax 444!444\cr
x \aligntab x \aligntab x \aligntab \setalignmentcharacter{/}\relax /\cr
.1 \aligntab ,2 \aligntab =3 \aligntab \setalignmentcharacter{?}\relax ?4\cr
.111 \aligntab ,222 \aligntab =333 \aligntab \setalignmentcharacter{=}\relax 44=444\cr
\egroup

```

This rather verbose setup renders as:

```

1.1      2,2      3=3      4.4
11.11    22,22    33=33    44,44
111.111  222,222  333=333  444!444
x        x        x        /
.1       ,2       =3       ?4
.111    ,222     =333     44=444

```

Using a high level interface makes sense but local control over such alignment too, so here follow some more examples. Here we use different alignment characters:

```

\starttabulate[|lG{.}|cG{,}|rG{=}|cG{x}||]
\NC 1.1 \NC 2,2 \NC 3=3 \NC a 0xFF \NC \NR
\NC 11.11 \NC 22,22 \NC 33=33 \NC b 0xFFF \NC \NR
\NC 111.111 \NC 222,222 \NC 333=333 \NC c 0xFFFF \NC \NR
\stoptabulate

```

```

1.1      2,2      3=3      a 0xFF
11.11    22,22    33=33    b 0xFFF
111.111  222,222  333=333  c 0xFFFF

```

In this example we specify the characters in the cells. We still need to add a specifier in the preamble definition because that will trigger the plugin.

```

\starttabulate[|lG{}|lG{}|]
\NC \showglyphs \setalignmentcharacter{.}1.1 \NC \setalignmentcharacter{.}1.1 \NC\NR
\NC \showglyphs \setalignmentcharacter{,}11,11 \NC \setalignmentcharacter{,}11,11 \NC\NR
\NC \showglyphs \setalignmentcharacter{=}111=111 \NC \setalignmentcharacter{=}111=111 \NC\NR
\stoptabulate

```

```

1.1      1 . 1
11,11    11 , 11
111=111  111=111

```

You can mix these approaches:

```

\starttabulate[|lG{.}|lG{}|]
\NC 1.1 \NC \setalignmentcharacter{.}1.1 \NC\NR
\NC 11.11 \NC \setalignmentcharacter{.}11.11 \NC\NR
\NC 111.111 \NC \setalignmentcharacter{.}111.111 \NC\NR
\stoptabulate

```

```

1.1      1.1
11.11    11.11
111.111  111.111

```

Here the already present alignment feature, that at some point in tabulate might use this new feature, is meant for numbers, but here we can go wild with words, although of course you need to keep in mind that we deal with typeset text, so there may be no match.

```

\starttabulate[|lG{.}|rG{.}|]
\NC foo.bar \NC foo.bar \NC \NR
\NC oo.ba \NC oo.ba \NC \NR
\NC o.b \NC o.b \NC \NR
\stoptabulate

```

```

foo.bar  foo.bar
oo.ba    oo.ba
o.b      o.b

```

This feature will only be used in know situations and those seldom involve advanced typesetting. However, the following does work:¹⁸

```
\starttabulate[|cG{d}|]
\NC \smallcaps abcdefgh \NC \NR
\NC          xdy       \NC \NR
\NC \sl          xdy       \NC \NR
\NC \tttf        xdy       \NC \NR
\NC \tfd          d        \NC \NR
```

\stoptabulate

```
abc d efg
  x d y
  x d y
  x d y
  d
```

As always with such mechanisms, the question is “Where to stop?” But it makes for nice demos and as long as little code is needed it doesn't hurt.

12.6 Pitfalls and tricks

The next example mixes bidirectional typesetting. It might look weird at first sight but the result conforms to what we discussed in previous paragraphs.

```
\starttabulate[|lG{.}|lG{}|]
\NC \righttoleft 1.1 \NC \righttoleft \setalignmentcharacter{.}1.1 \NC\NR
\NC          1.1 \NC          \setalignmentcharacter{.}1.1 \NC\NR
\NC \righttoleft 1.11 \NC \righttoleft \setalignmentcharacter{.}1.11 \NC\NR
\NC          1.11 \NC          \setalignmentcharacter{.}1.11 \NC\NR
\NC \righttoleft 1.111 \NC \righttoleft \setalignmentcharacter{.}1.111 \NC\NR
\NC          1.111 \NC          \setalignmentcharacter{.}1.111 \NC\NR
\stoptabulate
```

```
  1.1    1.1
1.1    1.1
  11.1   11.1
1.11   1.11
111.1  111.1
1.111  1.111
```

In case of doubt, look at this:

¹⁸ Should this be an option instead?


```

\starttabulate[|lG{.}|lG{}|lG{.}|lG{}|]
\NC \righttoleft 1.1 \NC \righttoleft \setalignmentcharacter{.}1.1 \NC
1.1 \NC \setalignmentcharacter{.}1.1 \NC\NR
\NC \righttoleft 1.11 \NC \righttoleft \setalignmentcharacter{.}1.11 \NC
1.11 \NC \setalignmentcharacter{.}1.11 \NC\NR
\NC \righttoleft 1.111 \NC \righttoleft \setalignmentcharacter{.}1.111 \NC
1.111 \NC \setalignmentcharacter{.}1.111 \NC\NR
\stoptabulate

```

```

1.1 1.1 1.1 1.1
11.1 11.1 1.11 1.11
111.1 111.1 1.111 1.111

```

The next example shows the effect of `\omit` and `\span`. The first one makes that in this cell the preamble template is ignored.

`\halign \bgroup`

```

\tabsize 2cm\relax [\alignmark]\hss \aligntab
\tabsize 2cm\relax \hss[\alignmark]\hss \aligntab
\tabsize 2cm\relax \hss[\alignmark]\cr
1\aligntab 2\aligntab 3\cr
\omit 1\aligntab \omit 2\aligntab \omit 3\cr
1\aligntab 2\span 3\cr
1\span 2\aligntab 3\cr
1\span 2\span 3\cr
1\span \omit 2\span \omit 3\cr
\omit 1\span \omit 2\span \omit 3\cr

```

`\egroup`

Spans are applied at the end so you see a mix of templates applied.

[1]	[2]	[3]
1	2	3
[1]	[2]	[3]
[1]	[2]	[3]
[1]	[2]	[3]
[1]	23	
123		

When you define an alignment inside a macro, you need to duplicate the `\alignmark` signals. This is similar to embedded macro definitions. But in LuaMetaTeX we can get around that by using `\aligncontent`. Keep in mind that when the preamble is scanned there is no doesn't expand with the exception of the token after `\span`.

`\halign \bgroup`

```
\tabsize 2cm\relax \aligncontent\hss \aligntab
\tabsize 2cm\relax \hss\aligncontent\hss \aligntab
\tabsize 2cm\relax \hss\aligncontent\cr
1\aligntab 2\aligntab 3\cr
A\aligntab B\aligntab C\cr
```

`\egroup`

1	2	3
A	B	C

In this example we still have to be verbose in the way we align but we can do this:

`\halign \bgroup`

```
\tabsize 2cm\relax \aligncontentleft \aligntab
\tabsize 2cm\relax \aligncontentmiddle\aligntab
\tabsize 2cm\relax \aligncontentright \cr
```

```

1\aligntab 2\aligntab 3\cr
A\aligntab B\aligntab C\cr
\egroup

```

Where the helpers are defined as:

```

\noaligned\protected\def\aligncontentleft
{\ignorespaces\aligncontent\unskip\hss}

\noaligned\protected\def\aligncontentmiddle
{\hss\ignorespaces\aligncontent\unskip\hss}

\noaligned\protected\def\aligncontentright
{\hss\ignorespaces\aligncontent\unskip}

```

The preamble scanner see such macros as candidates for a single level expansion so it will inject the meaning and see the `\aligncontent` eventually.

```

1          2          3
A          B          C

```

The same effect could be achieved by using the `\span` prefix:

```

\def\aligncontentleft{\ignorespaces\aligncontent\unskip\hss}

\halign { ... \span\aligncontentleft ...}

```

One of the reasons for not directly using the low level `\halign` command is that it's a lot of work but by providing a set of helpers like here might change that a bit. Keep in mind that much of the above is not new in the sense that we could not achieve the same already, it's just a bit programmer friendly.

12.7 Remark

It can be that the way alignments are interfaced with respect to attributes is a bit different between `LuaTeX` and `LuaMetaTeX` but because the former is frozen (in order not to interfere with current usage patterns) this is something that we will deal with deep down in `ConTeXt LMTX`.

In principle we can have hooks into the rows for pre and post material but it doesn't really pay of as grouping will still interfere. So for now I decided not to add these.

12.7 Colofon

Author Hans Hagen
ConT_EXt 2021.09.06 11:47
LuaMetaT_EX 2.0923
Support www.pragma-ade.com
 contextgarden.net

13 Marks

low level

TEX

marks

Contents

13.1	Introduction	170
13.2	The basics	171
13.3	Migration	172
13.4	Tracing	174
13.5	High level commands	175
13.6	Pitfalls	178

13.1 Introduction

Marks are one of the subsystems of $\text{T}_{\text{E}}\text{X}$, as are for instance alignments and math as well as inserts which they share some properties with. Both inserts and marks put signals in the list that later on get intercepted and can be used to access stored information. In the case of inserts this is typeset materials, like footnotes, and in the case of marks it's token lists. Inserts are taken into account when breaking pages, and marks show up when a page has been broken and is presented to the output routine. Marks are used for running headers but other applications are possible.

In MkII marks are used to keep track of colors, transparencies and more properties that work across page boundaries. It permits picking up at the top of a page from where one left at the bottom of the preceding one. When MkII was written there was only one mark so on top of that a multiple mark mechanism was implemented that filtered specific marks from a collection. Later, $\varepsilon\text{-T}_{\text{E}}\text{X}$ provided mark classes so that mechanism could be simplified. Although it is not that hard to do, this extension to $\text{T}_{\text{E}}\text{X}$ didn't add any further features, so we can assume that there was no real demand for that.¹⁹

But, marks have some nasty limitations, so from the $\text{ConT}_{\text{E}}\text{Xt}$ perspective there always was something to wish for. When you hide marks in boxes they will not be seen (the same is true for inserts). You cannot really reset them either. Okay, you can set them to nothing, but even then already present marks are still there. The $\text{LuaT}_{\text{E}}\text{X}$ engine has a `\clearmarks` primitive but that works global. In $\text{LuaMetaT}_{\text{E}}\text{X}$ a proper mark flusher is available. That engine also can work around the deeply nested disappearing marks. In addition, the current state of a mark can be queried and we have some tracing facilities.

In MkIV the engine's marks were not used at all and an alternative mechanism was written using Lua. It actually is one of the older MkIV features. It doesn't have the side

¹⁹ This is probably true for most $\text{LuaT}_{\text{E}}\text{X}$ and $\text{LuaMetaT}_{\text{E}}\text{X}$ extensions, maybe example usage create retrospective demand. But one reason for picking up on engine development is that in the $\text{ConT}_{\text{E}}\text{Xt}$ perspective we actually had some demands.

effects that native marks have but it comes at the price of more overhead, although that is bearable.

In this document we discuss marks but assume that LuaMetaT_EX is used with ConT_EXt LMTX. There we experiment with using the native marks, complemented by a few Lua mechanisms, but it is to be seen if that will be either a replacement or an alternative.

13.2 The basics

Although the original T_EX primitives are there, the plural ε -T_EX mark commands are to be used. Marks, signals with token lists, are set with:

```
\marks0{This is mark 0} % equivalent to: \mark{This is mark 0}
\marks4{This is mark 4}
```

When a page has been split off, you can (normally this only makes sense in the output routine) access marks with:

```
\topmarks 4
\firstmarks4
\botmarks 4
```

A ‘top’ mark is the last one on the previous page(s), the ‘first’ and ‘bottom’ refer to the current page. A mark is a so called node, something that ends up in the current list and the token list is stored with it. The accessors are just commands and they fetch the token list from a separately managed storage. When you set or access a mark that has not yet been used, the storage is bumped to the right size, so it doesn't make sense to use e.g. `\marks 999` when there are no 998 ones too: it not only takes memory, it also makes T_EX run over all these mark stores when synchronization happens. The best way to make sure that you are sparse is:

```
\newmarks\MyMark
```

Currently the first 16 marks are skipped so this makes `\MyMark` become mark 17. The reason is that we want to make sure that users who experiment with marks have some scratch marks available and don't overload system defined ones. Future versions of ConT_EXt might become more restrictive.

Marks can be cleared with:

```
\clearmarks 4
```


which clears the storage that keeps the top, first and bot marks. This happens immediately. You can delay this by putting a signal in the list:

`\flushmarks 4`

This (LuaMetaT_EX) feature makes it for instance easy to reset marks that keep track of section (and lower) titles when a new chapter starts. Of course it still means that one has to implement some mechanism that deals with this but ConT_EXt always had that.

The current, latest assigned, value of a mark is available too:

`\currentmarks 4`

Using this value in for instance headers and footers makes no sense because the last node set can be on a following page.

13.3 Migration

In the introduction we mentioned that LuaMetaT_EX has migration built in. In MkIV we have this as option too, but there it is delegated to Lua. It permits deeply nested inserts (notes) and marks (but we don't use native marks in MkIV).

Migrated marks end up in the postmigrated sublist of a box. In other lowlevel manuals we discuss these pre- and postmigrated sublists. As example we use this definition:

```
\setbox0\vbox\bgroup
test \marks 4 {mark 4.1}\par
test \marks 4 {mark 4.1}\par
test \marks 4 {mark 4.1}\par
\egroup
```

When we turn migration on (officially the second bit):

`\automigrationmode"FF \showbox0`

we get this:

```
> \box0=
2:4: \vbox[normal][...], width 483.69687, height 63.43475, depth 0.15576, direction l2r
2:4: .\list
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillskip][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
```

```

2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ....\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ....\glue[baseline][...] 8.34883pt
2:4: ....\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...List
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ....\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ....\glue[baseline][...] 8.34883pt
2:4: ....\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...List
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: \postmigrated
2:4: \mark[4][...]
2:4: ..{mark 4.1}
2:4: \mark[4][...]
2:4: ..{mark 4.1}
2:4: \mark[4][...]
2:4: ..{mark 4.1}

```

When we don't migrate, enforced with:

`\automigrationmode"00 \showbox0`

the result is:

```

> \box0=
2:4: \vbox[normal][...], width 483.69687, height 63.43475, depth 0.15576, direction l2r
2:4: \list
2:4: \hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: \list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: \mark[4][...]
2:4: ..{mark 4.1}
2:4: \glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: \glue[baseline][...] 8.34883pt
2:4: \hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r

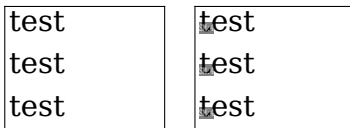
```

```

2:4: ...\list
2:4: ...\glue[left hang][...] 0.0pt
2:4: ...\glue[left][...] 0.0pt
2:4: ...\glue[parfillleft][...] 0.0pt
2:4: ...\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ...\glue[indent][...] 0.0pt
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ...\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ...\penalty[line][...] 10000
2:4: ...\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ...\glue[right][...] 0.0pt
2:4: ...\glue[right hang][...] 0.0pt
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}
2:4: ...\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ...\glue[baseline][...] 8.34883pt
2:4: ...\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...\list
2:4: ...\glue[left hang][...] 0.0pt
2:4: ...\glue[left][...] 0.0pt
2:4: ...\glue[parfillleft][...] 0.0pt
2:4: ...\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
  finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ...\glue[indent][...] 0.0pt
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ...\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ...\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ...\penalty[line][...] 10000
2:4: ...\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ...\glue[right][...] 0.0pt
2:4: ...\glue[right hang][...] 0.0pt
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}

```

When you say `\showmakeup` or in this case `\showmakeup [mark]` the marks are visualized:



enabled

disabled

Here `sm` means ‘set mark’ while `rm` would indicate a ‘reset mark’. Of course migrated marks don't show up because these are bound to the box and thereby have become a specific box property as can be seen in the above trace.

13.4 Tracing

The LuaMetaTeX engine has a dedicated tracing option for marks. The fact that the traditional engine doesn't have this can be seen as indication that this is seldom needed.

`\tracingmarks1`

`\tracingonline2`

When tracing is set to 1 we get a list of marks for the just split of page:

```

2:7: <mark class 51, top := bot>
2:7: ..{sample 9.1}
2:7: <mark class 51: first := mark>
2:7: ..{sample 10.1}
2:7: <mark class 51: bot := mark>
2:7: ..{sample 10.1}
2:7: <mark class 51, page state>
2:7: ..top {sample 9.1}
2:7: ..first {sample 10.1}
2:7: ..bot {sample 10.1}

```

When tracing is set to 2 you also get details we get a list of marks of the analysis:

```

1:9: <mark class 51, top := bot>
1:9: ..{sample 5.1}
1:9: <mark class 51: first := mark>
1:9: ..{sample 6.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 6.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 7.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 8.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 9.1}
1:9: <mark class 51, page state>
1:9: ..top {sample 5.1}
1:9: ..first {sample 6.1}
1:9: ..bot {sample 9.1}

```

13.5 High level commands

I think that not that many users define their own marks. They are useful for showing section related titles in headers and footers but the implementation of that is hidden. The native mark references are `top`, `first` and `bottom` but in the `ConTeXt` interface we use different keywords.

ConTeXt	TeX	column	page
previous	top	last before sync	last on previous page
top	first	first in sync	first on page

bottom	bot	last in sync	last on page
first	top	first not top in sync	first on page
last	bot	last not bottom in sync	last on page
<hr/>			
default		the same as first	
current		the last set value	
<hr/>			

In order to separate marks in ConT_EXt from those in T_EX, the term ‘marking’ is used. In MkIV the regular marks mechanism is of course there but, as mentioned, not used. By using a different namespace we could make the transition from MkII to MkIV (the same is true for some more mechanisms).

A marking is defined with

```
\definemarking[MyMark]
```

A defined marking can be set with two equivalent commands:

```
\setmarking[MyMark]{content}
\marking [MyMark]{content}
```

The content is not typeset but stored as token list. In the sectioning mechanism that uses markings we don't even store titles, we store a reference to a title. In order to use that (deep down) we hook in a filter command. By default that command does nothing:

```
\setupmarking[MyMark][filtercommand=\firstofoneargument]
```

The token list does *not* get expanded by default, unless you set it up:

```
\setupmarking[MyMark][expansion=yes]
```

The current state of a marking can be cleared with:

```
\clearmarking[MyMark]
```

but because that en is not synchronized the real deal is:

```
\resetmarking[MyMark]
```

Be aware that it introduces a node in the list. You can test if a marking is defined with (as usual) a test macro. Contrary to (most) other test macros this one is fully expandable:

```
\doifelsemarking {MyMark} {
  defined
} {
```

```

    undefined
}

```

Because there can be a chain involved, we can relate markings. Think of sections below chapters and subsections below sections:

```
\relatemarking[MyMark][YourMark]
```

When a marking is set its relatives are also reset, so setting YourMark will reset MyMark. It is this kind of features that made for marks being wrapped into high level commands very early in the ConT_EXt development (and one can even argue that this is why a package like ConT_EXt exists in the first place).

The rest of the (relatively small) repertoire of commands has to do with fetching markings. The general command is `\getmarking` that takes two or three arguments:

```

\getmarking[MyMarking][first]
\getmarking[MyMarking][page][first]
\getmarking[MyMarking][page][first]
\getmarking[MyMarking][column:1][first]

```

There are (normally) three marks that can be fetched so we have three commands that do just that:

```

\fetchonemark [MyMarking][which one]
\fetchtwomarks[MyMarking]
\fetchallmarks[MyMarking]

```

You can setup a separator key which by default is:

```
\setupmarking[MyMarking][separator=\space\emdash\space]
```

Injection is enabled by default due to this default:

```
\setupmarking[MyMarking][state=start]
```

The following three variants are (what is called) fully expandable:

```

\fetchonemarking [MyMarking][which one]
\fetchtwomarkings[MyMarking]
\fetchallmarkings[MyMarking]

```

13.6 Pitfalls

The main pitfall is that a (re)setting a mark will inject a node which in vertical mode can interfere with spacing. In for instance section commands we wrap them with the title so there it should work out okay.

13.6 Colofon

Author	Hans Hagen
ConT _E Xt	2021.09.06 11:47
LuaMetaT _E X	2.0923
Support	www.pragma-ade.com contextgarden.net

14 Inserts

low level

TEX

inserts

Contents

14.1	Introduction	180
14.2	The page builder	180
14.3	Inserts	182
14.4	Storing	184
14.5	Callbacks	184

14.1 Introduction

This document is a mixed bag. We do discuss inserts but also touch elements of the page builder because inserts and regular page content are handled there. Examples of mechanisms that use inserts are footnotes. These have an anchor in the running text and some content that ends up (normally) at the bottom of the page. When considering a page break the engine tries to make sure that the anchor (reference) and the content end up on the same page. When there is too much, it will distribute (split) the content over pages.

We can discuss page breaks in a (pseudo) scientific way and explore how to optimize this process, taking into accounts also inserts that contain images but it doesn't make much sense to do that because in practice we can encounter all kind of interferences. Theory and practice are too different because a document can contain a wild mix of text, figures, formulas, notes, have backgrounds and location dependent processing. It get seven more complex when we are dealing with columns because $\text{T}_{\text{E}}\text{X}$ doesn't really know that concept.

I will therefore stick to some practical aspects and the main reason for this document is that I sort of document engine features and at the same time give an impression of what we deal with. I will do that in the perspective of $\text{LuaMetaT}_{\text{E}}\text{X}$, which has a few more options and tracing than other engines.

Currently this document is mostly for myself to keep track of the state of inserts and the page builder in $\text{LuaMetaT}_{\text{E}}\text{X}$ and $\text{ConT}_{\text{E}}\text{Xt LMTX}$. The text is not yet corrected and can have errors.

14.2 The page builder

When your document is processed content eventually gets added to the so called main vertical list (mvl). Content first get appended to the list of contributions and at specific moments it will be handed over to the mvl. This process is called page building. There we can encounter the following elements (nodes):

<code>glue</code>	a vertical skip
<code>penalty</code>	a vertical penalty
<code>kern</code>	a vertical kern
<code>vlist</code>	a a vertical box
<code>hlist</code>	a horizontal box (often a line)
<code>rule</code>	a horizontal rule
<code>boundary</code>	a boundary node
<code>whatsit</code>	a node that is used by user code (often some extension)
<code>mark</code>	a token list (as used for running headers)
<code>insert</code>	a node list (as used for notes)

The engine itself will not insert anything other than this but Lua code can mess up the contribution list and the mvl and that can trigger an error. Handing over the contributions is done by the page builder and that one kicks in in several places:

- When a penalty gets inserted it is part of evaluating if the output routine should be triggered. This triggering can be enforced by values equal or below 10.000 that then can be checked in the set routine.
- The builder is *not* exercised when a glue or kern is injected so there can be multiple of them before another element triggers the builder.
- Adding a box triggers the builder as does the result of an alignment which can be a list of boxes.
- When the output routine is finished the builder is executed because the routine can have pushed back content.
- When a new paragraph is triggered by the `\par` command the builder kicks in but only when the engine was able to enter vertical mode.
- When the job is finished the builder will make sure that pending content is handled.
- An insert and `vadjust` *can* trigger the builder but only at the nesting level zero which normally is not the case (I need an example).
- At the beginning of a paragraph (like text), before display math is entered, and when display math ends the builder is also activated.

At the $\text{T}_{\text{E}}\text{X}$ the builder is triggered automatically in the mentioned cases but at the Lua end you can use `tex.triggerbuildpage()` to flush the pending contributions.

The properties that relate to the page look like counter and dimension registers but they are not. These variables are global and managed differently.

<code>\pagegoal</code>	the available space
<code>\pagetotal</code>	the accumulated space
<code>\pagestretch</code>	the possible zero order stretch
<code>\pagefilstretch</code>	the possible one order stretch
<code>\pagefillstretch</code>	the possible second order stretch
<code>\pagefilllstretch</code>	the possible third order stretch
<code>\pageshrink</code>	the possible shrink
<code>\pagedepth</code>	the current page depth
<code>\pagevsize</code>	the initial page goal

When the first content is added to an empty page the `\pagegoal` gets the value of `\vsize` and gets frozen but the value is diminished by the space needed by left over inserts. These inserts are managed via a separate list so they don't interfere with the page that itself of course can have additional inserts. The `\pagevsize` is just a (LuaMeta $\text{T}_{\text{E}}\text{X}$) status variable that hold the initial `\pagegoal` but it might play a role in future extensions.

Another variable is `\deadcycles` that registers the number of times the output routine is called without returning result.

14.3 Inserts

We now come to inserts. In traditional $\text{T}_{\text{E}}\text{X}$ an insert is a data structure that runs on top of registers: a box, count, dimension and skip. An insert is accessed by a number so for instance `insert 123` will use the four registers of that number. Because $\text{T}_{\text{E}}\text{X}$ only offers a command alias mechanism for registers (like `\countdef`) a macro package will implement some allocator management subsystem (like `\newcount`). A `\newinsert` has

to be defined in a way that the four registers are not clashing with other allocators. When you start with $\text{T}_{\text{E}}\text{X}$ seeing code that deals with in (in plain $\text{T}_{\text{E}}\text{X}$) can be puzzling but it follows from the way $\text{T}_{\text{E}}\text{X}$ is set up. But inserts are probably not what you start exploring right away away.

In $\text{LuaMetaT}_{\text{E}}\text{X}$ you can set `\insertmode` to 1 and that is what we do in $\text{ConT}_{\text{E}}\text{Xt}$. In that mode inserts are taken from a pool instead of registers. A side effect is that like the page properties the insert properties are global too but that is normally no problem and can be managed well by a macro package (that probably would assign register the values globally too). The insert pool will grow dynamically on demand so one can just start at 1; in $\text{ConT}_{\text{E}}\text{Xt MkIV}$ we use the range 127 upto 255 in order to avoid a clash with registers. In LMTX we start at 1 because there are no clashes.

A consequence of this approach is that we use dedicated commands to set the insert properties:

<code>\insertdistance</code>	glue	the space before the first instance (on a page)
<code>\insertmultiplier</code>	count	a factor that is used to calculate the height used
<code>\insertlimit</code>	dimen	the maximum amount of space on a page to be taken
<code>\insertpenalty</code>	count	the floating penalty (used when set)
<code>\insertmaxdepth</code>	dimen	the maximum split depth (used when set)
<code>\insertstorage</code>	count	signals that the insert has to be stored for later
<code>\insertheight</code>	dimen	the accumulated height of the inserts so far
<code>\insertdepth</code>	dimen	the current depth of the inserts so far
<code>\insertwidth</code>	dimen	the width of the inserts

These commands take a number and an integer, dimension or glue specification. They can be set and queried but setting the dimensions can have side effects. The accumulated height of the inserts is available in `\insertheights` (which can be set too). The `\floatingpenalty` variable determines the penalty applied when a split is needed.

In the output routine the original $\text{T}_{\text{E}}\text{X}$ variable `\insertpenalties` is a counter that keeps the number of insertions that didn't fit on the page while otherwise it has the accumu-

lated penalties of the split insertions. When `\holdinginserts` is non zero the inserts in the list are not collected for output, which permits the list to be fed back for reprocessing.

The LuaMetaTeX specific storage mode `\insertstoring` variable is explained in the next section.

14.4 Storing

This feature is kind of special and still experimental. When `\insertstoring` is set 1, all inserts that have their storage flag set will be saved. Think of a multi column setup where inserts have to end up in the last column. If there are three columns, the first two will store inserts. Then when the last column is dealt with `\insertstoring` can be set to 2 and that will signal the builder that we will inject the inserts. In both cases, the value of this register will be set to zero so that it doesn't influence further processing.

14.5 Callbacks

Todo, nothing new there, so no hurry.

14.5 Colofon

Author	Hans Hagen
ConTeXt	2021.09.06 11:47
LuaMetaTeX	2.0923
Support	www.pragma-ade.com contextgarden.net