low level

TeX

buffers

Contents

1 Preamble

Buffers are not that low level but it makes sense to discuss them in this perspective because it relates to tokenization, internal representation and manipulating.

In due time we can describe some more commands and details here. This is a start. Feel free to tell me what needs to be explained.

2 Encoding

Normally processing a document starts with reading from file. In the past we were talking single bytes that were then maps onto a specific input encoding that itself matches the encoding of a font. When you enter an ‘a’ its (normally ascii) number 97 becomes the index into a font. That same number is also used in the hyphenator which is why font encoding and hyphenation are strongly related. If in an eight bit TEX engine you need a precomposed ‘ä’ you have to use an encoding that has that character in some slot with again matching fonts and patterns. The actually used font can have the *shapes* in different slots and remapping is then done in the backend code using encoding and mapping files. When OpenType fonts are used the relationship between characters (input) and glyphs (rendering) also depends on the application of font features.

In eight bit environments all this brings a bit of a resource management nightmare along with complex installation of new fonts. It also puts strain on the macro package, especially when you want to mix different input encodings onto different font encodings and thereby pattern encodings in the same document. You can compare this with code pages in operating system, but imagine them potentially being mixed in one document, which can happen when you mix multiple languages where the accumulated number of different characters exceeds 256. You end up switching between encodings. One way to deal with it is making special characters active and let their meaning differ per situation. That is for instance how in MkII we handled utf8 and thereby got around distributing multiple pattern files per language as we only needed to encoding them in utf and then remap them to the required encoding when loading patterns. A mental exercise is wondering how to support cjk scripts in an eight bit MkII, something that actually can be done with some effort.

The good news is that when we moved from MkII to MkIV we went exclusively utf8 because that is what the LuaTEX engine expects. Upto four bytes are read in and translated into one Unicode character. The internal representation is a 32 bit integer (four bytes) instead of a single byte. That also means that in the transition we got rid of quite some encoding related low level font and pattern handling. We still support input encodings (called regimes in ConTEXt) but I'm pretty sure that nowadays no one uses input other than utf8. While ConTEXt is normally quite upward compatible this is one area where there were fundamental changes.

There is still some interpretation going on when reading from file: for instance, we need to normalize the Unicode input, and we feed the engine separate lines on demand. Apart from that, some characters like the backslash, dollar sign and curly braces have special meaning so for accessing them as characters we have to use commands that inject those characters. That didn't change when we went from MkII to MkIV. In practice it's never really a problem unless you find yourself in one of the following situations:

- *Example code has to be typeset as-is, so braces etc. are just that.* This means that we have to change the way characters are interpreted. Typesetting code is needed when you want to document TEX and macros which is why mechanisms for that have to be present right from the start.

- *Content is collected and used later.* A separation of content and usage later on often helps making a source look cleaner. Examples are “wrapping a table in a buffer” and “including that buffer when a table is placed” using the placement macros.

- *Embedded MetaPost and Lua code.* These languages come with different interpretation of some characters and especially MetaPost code is often stored first and used (processed) later.

- *The content comes from a different source.* Examples are xml files where angle brackets are special but for instance braces aren't. The data is interpreted as a stream or as a structured tree.

- *The content is generated.* It can for instance come from Lua, where bytes (representing utf) is just text and no special characters are to be intercepted. Or it can come from a database (using a library).

For these reasons ConTEXt always had ways to store data in ways that makes this possible. The details on how that is done might have changed over versions, been optimized, extended with additional interfaces and features but given where we come from most has been there from the start.

3 Performance

When TEX came around, the bottlenecks in running TEX were the processor, memory and disks and depending on the way one used it the speed of the console or terminal; so, basically the whole system. One could sit there and wait for the page counters (`[1] [2] ..` to show up. It was possible to run TEX on a personal computer but it was somewhat resource hungry: one needed a decent disk (a 10 MB hard disk was huge and with todays phone camera snapshots that sounds crazy). One could use memory extenders to get around the 640K limitation (keep in mind that the programs and operating systems also took space). This all meant that one could not afford to store too many tokens in memory but even using files for all kind of (multi-pass) trickery was demanding.

When processors became faster and memory plenty the disk became the bottleneck, but that changed when ssd's showed up. Combined with already present file caching that had some impact. We are now in a situation that cpu cores don't get that much faster (at least not twice as fast per iteration) and with TEX being a single core byte cruncher we're more or less in a situation where performance has to come from efficient programming. That means that, given enough memory, in some cases storing in tokens wins over storing in files, but it is no rule. In practice there is not much difference so one can even more than yesterday choose for the most convenient method. Just assume that the ConTEXt code, combined with LuaMetaTEX will give you what you need with a reasonable performance. When in doubt, test with simple test files and it that works out well compared to the real code, try to figure out where ‘mistakes’ are made. Inefficient Lua and TEX code has way more impact than storing a few more tokens or using some files.

4 Files

Nearly always files are read once per run. The content (mixed with commands) is scanned and macros are expanded and/or text is typeset as we go. Internally the LuaMetaTEX engine is in “scanning from file”, “scanning from token lists”, or “scanning from Lua output” mode. The first mode is (in principle) the slowest because utf sequences are converted to tokens (numbers) but there is no way around it. The second method is fast because we already have these numbers, but we need to take into account where the linked list of tokens comes from. If it is converted runtime from for instance file input or macro expansion we need to add the involved overhead. But scanning a stored macro body is pretty efficient especially when the macro is part of the loaded macro package (format file). The third method is comparable with reading from file but here we need to add the overhead involved with storing the Lua output into data structures suitable for TEX's input mechanism, which can involve memory allocation outside the reserved pool of tokens. On modern systems that is not really a problem. It is good to keep in mind that when TEX was written much attention was paid to optimization and in LuaMetaTEX we even went a bit further, also because we know what kind of input, processing and output we're dealing with.

When reading from file or Lua output we interpret bytes turned utf numbers and that is when catcode regimes kick in: characters are interpreted according to the catcode properties: escape character (backslash), curly braces (grouping and arguments), dollars (math), etc. While with reading from token lists these catcodes are already taken care of and we're basically interpreting meanings instead of characters. By changing the catcode regime we can for instance typeset content verbatim from files and Lua strings but when reading from token lists we're sort of frozen. There are tricks to reinterpret the token list but that comes with overhead and limitations.

5 Macros

A macro can be seen as a named token with a meaning attached. In LuaMetaTEX macros can take up to 15 arguments (six more than regular TEX) that can be separated by so called delimiters. A token has a command property (operator) and a value (operand). Because a Unicode character doesn't need all four bytes of an integer and because in the engine numbers, dimensions and pointers are limited in size we can store all of these efficiently with the command code. Here the body of `\foo` is a list of three tokens:

```
\def\foo{abc} \foo \foo \foo
```

When the engine fetches a token from a list it will interpret the command and when it fetches from file it will create tokens on the fly and then interpret those. When a file or list is exhausted the engine pops the stack and continues at the previous level. Because macros are already tokenized they are more efficient than file input. For more about macros you can consult the low level document about them.

The more you use a macro, the more it pays off compared to a file. However don't overestimate this, because in the end the typesetting and expanding all kind of other involved macros might reduce the file overhead to noise.

6 Token lists

A token list is like a macro but is part of the variable (register) system. It is just a list (so no arguments) and you can append and prepend to that list.

```
\toks123={abc}    \the\toks123
\scratchtoks{abc} \the\scratchtoks
```

Here `\scratchtoks` is defined with `\newtoks` which creates an efficient reference to a list so that, contrary to the first line, no register number has to be scanned. There are low level manuals about tokens and registers that you can read if you want to know more about this. As with macros the list in this example is three tokens long. Contrary to macros there is no macro overhead as there is no need to check for arguments.[1]

Because they use more or less the same storage method macros and token list registers perform the same. The power of registers comes from some additional manipulators in LuaTEX (and LuaMetaTEX) and the fact that one can control expansion with `\the`, although that later advantage is compensated with extensions to the macro language (like `\protected` macro definitions).

7 Buffers

Buffers are something specific for ConTEXt and they have always been part of this system. A buffer is defined as follows:

```
\startbuffer[one]
line 1
line 2
\stopbuffer
```

Among the operations on buffers the next two are used most often:

```
\typebuffer[one]
\getbuffer[one]
```

---

[1] In LuaMetaTEX a macro without arguments is also quite efficient.

Scanning a buffer at the TEX end takes a little effort because when we start reading the catcodes are ignored and for instance backslashes and curly braces are retained. Hardly any interpretation takes place. The same is true for spacing, so multiple spaces are not collapsed and newlines stay. The tokenized content of a buffer is converted back to a string and that content is then read in as a pseudo file when we need it. So, basically buffers are files! In MkII they actually were files (in the `\jobname` name space and suffix `tmp`), but in MkIV they are stored in and managed by Lua. That also means that you can set them very efficiently at the Lua end:

```
\startluacode
buffers.assign("one",[[
line 1
line 2
]])
\stopluacode
```

Always keep in mind that buffers eventually are read as files: character by character, and at that time the content gets (as with other files) tokenized. A buffer name is optional. You can nest buffers, with and without names.

Because ConTEXt is very much about re-use of content and selective processing we have an (already old) subsystem for defining named blocks of text (using `\begin...` and `\end...` tagging. These blocks are stored just like buffers but selective flushing is part of the concept. Think of coding an educational document with explanations, questions, answers and then typesetting only the explanations, or the explanation along width some questions. Other components can be typeset later so one can make for instance a special book(let) with answers that either of not repeats the questions. Here we need features like synchronization of numbers so that's why we cannot really use buffers. An alternative is to use xml and filter from that.

The `\definebuffer` command defines a new buffer environment. When you set buffers in Lua you don't need to define a buffer because likely you don't need the `\start` and `\stop` commands. Instead of `\getbuffer` you can also use `\getdefinedbuffer` with defined buffers. In that case the `before` and `after` keys of that specific instance are used.

The `\getinlinebuffer` command, which like the getters takes a list of buffer names, ignores leading and trailing spaces. When multiple buffers are flushed this way, spacing between buffers is retained.

The most important aspect of buffers is that the content is *not* interpreted and tokenized: the bytes stay as they are.

```
\definebuffer[MyBuffer]

\startMyBuffer
\bold{this is
a buffer}
\stopMyBuffer

\typeMyBuffer \getMyBuffer
```

These commands result in:

```
\bold{this is
a buffer}
```

this is a buffer

There are not that many parameters that can be set: `before`, `after` and `strip` (when set to `no` leading and trailing spacing will be kept. The `\stop...` command, in our example `\stopMyBuffer`, can be defined independent to so something after the buffer has be read and stored but by default nothing is done.

You can test if a buffer exists with `\doifelsebuffer` (expandable) and `\doifelsebufferempty` (unexpandable). A buffer is kept in memory unless it gets wiped clean with `resetbuffer`.

```
\savebuffer      [MyBuffer][temp]     % gets name: jobname-temp.tmp
\savebufferinfile[MyBuffer][temp.log] % gets name: temp.log
```

You can also stepwise fill such a buffer:

```
\definesavebuffer[slide]

\startslide
    \starttext
\stopslide
\startslide
    slide 1
\stopslide
text 1 \par
\startslide
    slide 2
\stopslide
text 2 \par
```

```
\startslide
    \stoptext
\stopslide
```

After this you will have a file `\jobname-slide.tex` that has the two lines wrapped as text. You can set up a 'save buffer' to use a different filename (with the `file` key), a different prefix using `prefix` and you can set up a `directory`. A different name is set with the `list` key.

You can assign content to a buffer with a somewhat clumsy interface where we use the delimiter `\endbuffer`. The only restriction is that this delimiter cannot be part of the content:

```
\setbuffer[name]here comes some text\endbuffer
```

For more details and obscure commands that are used in other commands you can peek into the source.

Using buffers in the cld interface is tricky because of the catcode magick that is involved but there are setters and getters:

function	**arguments**
`buffers.assign`	`name, content [,catcodes]`
`buffers.erase`	`name`
`buffers.prepend`	`name, content`
`buffers.append`	`name, content`
`buffers.exists`	`name`
`buffers.empty`	`name`
`buffers.getcontent`	`name`
`buffers.getlines`	`name`

There are a few more helpers that are used in other (low level) commands. Their functionality might adapt to their usage there. The `context.startbuffer` and `context.stopbuffer` are somewhat differently defined than regular cld commands.

8 Setups

A setup is basically a macro but is stored and accessed in a namespace separated from ordinary macros. One important characteristic is that inside setups newlines are ignored.

```
\startsetups MySetupA
```

```
    This is line 1
    and this is line 2
\stopsetups

\setup{MySetupA}
```

This is line 1and this is line 2

A simple way out is to add a comment character preceded by a space. Instead you can also use `\space`:

```
\startsetups [MySetupB]
    This is line 1 %
    and this is line 2\space
    while here we have line 3
\stopsetups

\setup[MySetupB]
```

This is line 1 and this is line 2 while here we have line 3

You can use square brackets instead of space delimited names in definitions and also in calling up a (list of) setup(s). The `\directsetup` command takes a single setup name and is therefore more efficient.

Setups are basically simple macros although there is some magic involved that comes from their usage in for instance xml where we pass an argument. That means we can do the following:

```
\startsetups MySetupC
    before#1after
\stopsetups

\setupwithargument{MySetupC}{ {\em and} }
```

before *and* after

Because a setup is a macro, the body is a linked list of tokens where each token takes 8 bytes of memory, so `MySetupC` has 12 tokens that take 96 bytes of memory (plus some overhead related to macro management).

9 xml

Discussing xml is outside the scope of this document but it is worth mentioning that once an xml tree is read is, the content is stored in strings and can be filtered into TEX, where it is interpreted as if coming from files (in this case Lua strings). If needed the content can be interpreted as TEX input.

10 Lua

As mentioned already, output from Lua is stored and when a Lua call finishes it ends up on the so called input stack. Every time the engine needs a token it will fetch from the input stack and the top of the stack can represent a file, token list or Lua output. Interpreting bytes from files or Lua strings results in tokens. As a side note: Lua output can also be already tokenized, because we can actually write tokens and nodes from Lua, but that's more an implementation detail that makes the Lua input stack entries a bit more complex. It is normally not something users will do when they use Lua in their documents.

11 Protection

When you define macros there is the danger of overloading some defined by the system. Best use CamelCase so that you stay away from clashes. You can enable some checking:

```
\enabledirectives[overloadmode=warning]
```

or when you want to quit on a clash:

```
\enabledirectives[overloadmode=error]
```

When these trackers are enabled you can get around the check with:

```
\pushoverloadmode
  ...
\popoverloadmode
```

But delay that till you're sure that redefining is okay.

11 Colofon

Author	Hans Hagen
ConTEXt	2023.04.27 17:04
LuaMetaTEX	2.1008
Support	www.pragma-ade.com contextgarden.net