What's a module?

A module is a collection of code for

- a function which is missing in CONTEXT,
- a style for a document or presentation which is used multiple times or
- a collection of fonts.

The following code is of the first category and provides the new command `\fancybreak` to insert thought breaks in your document.

```
\unprotect

\def\????fb{@@@@fb} % Namespace: FancyBreak

\def\fancybreakparameter#1%
  {\csname\????fb#1\endcsname}

\def\setupfancybreak
  {\dodoubleargument\getparameters[\????fb]}

\def\fancybreak
  {\dowithnextboxcontent
     {\setupalign[\fancybreakparameter\c!align]}
     {\blank[\fancybreakparameter\c!spavebefore]%
      \flushnextbox
      \blank[\fancybreakparameter\c!spaceafter ]}
     \vbox}

\setupfancybreak
  [\c!spacebefore=,
   \c!spaceafter=\v!nowhite,
   \c!align=\v!middle]

\protect \endinput
```

META-Information

The first part to document a module to add META-Information about the author and title to the file with the `\module[...]` block. Part of these information are used when you a PDF of the modules source.

The fields

- title,
- subtitle,
- author and
- date (default: `\currentdate`)

are mandatory, additional fields like

- copyright,
- license (suggested),
- email and
- version

are optional but it's a good style to provide more information.

```
%D \module
%D   [     file=t-fancybreak,
%D      version=20xx.xx.xx,
%D        title=\CONTEXT\ User Module,
%D     subtitle=Fancybreak,
%D       author=Ben Lee User,
%D         date=\currentdate,
%D    copyright=Ben Lee User,
%D        email=ben.lee.user@tex.org,
%D      license=Public Domain]
```

Comments

To document a module you can write a separate manual for it but it's also possible to include the documentation in the module itself, CONTEXT provides three different different types of comment environments for this. All of them start with a normal comment character "%" followed by second letter which indicates the type of the envionment.

The first type is used load modules are define additional commands, you can use it with "%M" at the begin of the line.

The second type is for normal documentation like text, tables, headers and all other kind of markup. To use this type write "%D" at the begin of each line.

The last type is used to skip certain parts of the documentation or code and exlude it from the produced PDF file.

```
%M \usemodule [fancybreak]
%M \loadsetups[fancybreak.xml]

%D This is modul is intended for ...

%D The following optiona are availabe:
%D
%D \startitemize
%D \item spacebefore,
%D \item spaceafter and
%D \item align.
%D \stopitemize

%S B
%S Everything onwards is
%S skipped and does not
%S appear in the PDF.
%S E
```

Document Structure

You can use CONTEXT's normal heading commands like `\chapter` or `\section` without any changes but one should know that `\section` starts a new page in the PDF.

Global to these settings can be done at the begin of the file with the "`%M`" comment type, local changes to the current can be done with "`%D`" at the desired place.

```
%D \chapter{Documentation}

%D \section{Examples}

%M \setuphead
%M   [section]
%M   [page=no]
```

Markup

Besides CONTEXT's core functions there a few extra commands for modules.

There are three commands which show there argument in the margin at the current position where you include them in the source and also in a auto generated index at the end of the documentation, these three commands are

- `\macros{...}`,
- `\extras{...}` and
- `\elements{...}`.

With the "example" environment you typeset text which is indented at the left and right margins.

The functions from the units and the abbreviation modules are also supported because both of them are loaded when you generate a PDF from the source.

```
%D \macros   {macro}    -> \macro
%D \extras   {extra}    ->  extra
%D \elements{element} -> <element>

%D \macros   {one,two} -> \one \two
%D \extras   {a,b}     ->  a b
%D \elements{em,bf}    -> <em> <bf>

%D \startexample
%D ...
%D \stopexample
```

PDF generation

The main reason to include documentation to the source itself is to produce a PDF from the source instead of a separate manual.

The PDF is generated with the following command:

```
context --ctx=s-mod.ctx <modulename>
```

The resulting file has the same name as the input file, when the input is a MKII or MKIV file the name of the result is `<filename>-mkix.pdf`.

To be aware of the file type one has to give the complete name including the extension as argument to the context command.

ConTEXt

title : ConTEXt User Module
subtitle : Fancybreak
author : Ben Lee User
date : September 12, 2010

The source formatter is not only limited to TEX files, because of METAPOST's origin with % as comment sign you add at to them documentation in the same form as you can do with TEX files.

A last option for source documentation is to exclude the real code from the PDF and to output the only what's written as documentation itseld, you can do this with the `nocode` mode at the commandline.

```
context --ctx=s-mod.ctx --nomode <module>
```



```
1  \unprotect

2  \def\????fb{@@@@fb} % Namespace: FancyBreak

3  \def\fancybreakparameter#1%
     {\csname\????fb#1\endcsname}

4  \def\setupfancybreak
     {\dodoubleargument\getparameters[\????fb]}

5  \def\fancybreak
     {\dowithnextboxcontent
        {\setupalign[\fancybreakparameter\c!align]}
        {\blank[\fancybreakparameter\c!spacebefore]%
         \flushnextbox
         \blank[\fancybreakparameter\c!spaceafter]}
        \vbox}

6  \setupfancybreak
     [\c!spacebefore=,
      \c!spaceafter=\v!nowhite,
      \c!align=\v!middle]

7  \protect \endinput
```

LUA files

CONTEXT does not only provide the option to include documentation in TEX files but also in LUA files.

Again you can start with with a pure LUA file which contains functions only and you add the documentation afterwards or at the time of writing the code.

```
if not math.round then
    function math.round(x)
        return floor(x + 0.5)
    end
end

if not math.div then
    function math.div(n,m)
        return floor(n/m)
    end
end

if not math.mod then
    function math.mod(n,m)
        return n % m
    end
end
```

META-Information

When you write a TEX module you add meta information at the begin of document, the same is done with a LUA file too but now the information are written as a normal LUA table, a example from the core can be see in the other frame.

Currently none of these information are used when you generate a PDF from the source but you should use the same fields for your own file.

```
if not modules then modules = { } end modules ['l-math'] = {
    version   = 1.001,
    comment   = "companion to luat-lib.mkiv",
    author    = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
    copyright = "PRAGMA ADE / ConTeXt Development Team",
    license   = "see context related readme files"
}
```

Markup

After we added the meta information we can now start with the documentation itself, the difference compared with TEX files is that in the LUA files XML is used for the markup and each part to be printed is written as a block comment.

It's important to add "ldx" to the comments because they indicate what is supposed to be a normal LUA and what is documentation.

```
--[[ldx
...
--ldx]]
```

Because we use XML markup each paragraph has now to be included in a pair of p-tags. The complete list of elements is:

- p,
- source,
- key,
- lines,
- line,
- logo or l,
- typing and
- type or t.

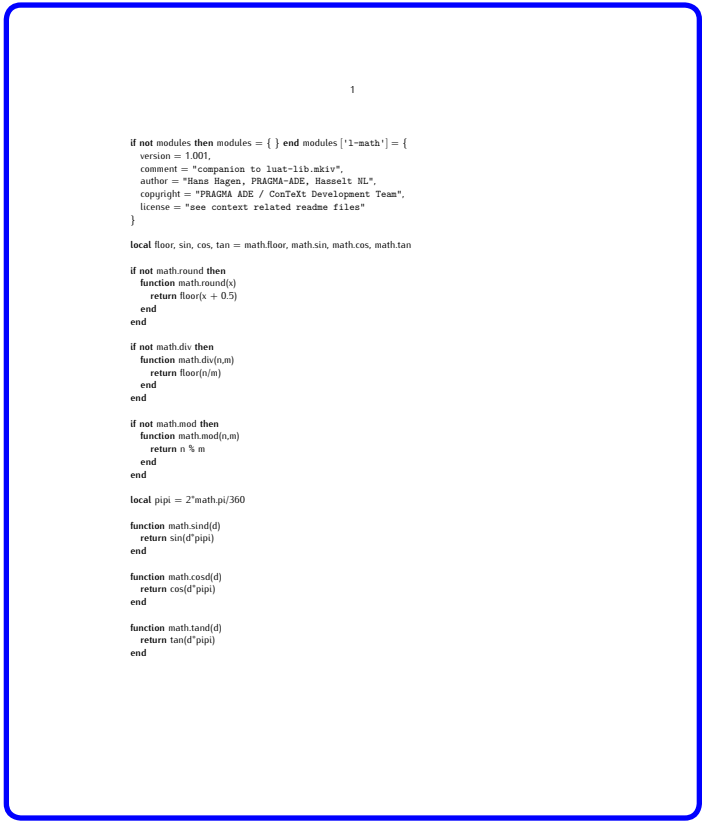
```
<p>This is a paragraph.</p>

<type>this</type> and <t>also</t> are written
verbatim in the output

<typing>
this a longer code
block similar to \TEX's
\starttyping/\stoptyping
environment
</typing>

<logo n="tex"/>, <logo n="lua"/>, <l l="mkii"/>
and <l n="mkiv"/> are the xml way for \TEX,
\LUA, \MKII and \MKIV

<key>function</key> embolden it's content

<lines>
<line>this text is written</line>
<line>like in ConTeXt's line</line>
<line>environment but you tags</line>
<line>where a line starts and ends</line>
</lines>

<source>Examples</source> produces a \subject head
```

PDF

A formatted version of the source is produce with the line

```
context --ctx=x-ldx.ctx <filename>
```

on the command line, the other from shows the PDF file of the core file `l-math.lua`.



1

```
if not modules then modules = { } end modules ['l-math'] = {
  version = 1.001,
  comment = "companion to luat-lib.mkiv",
  author = "Hans Hagen, PRAGMA-ADE, Hasselt NL",
  copyright = "PRAGMA ADE / ConTeXt Development Team",
  license = "see context related readme files"
}

local floor, sin, cos, tan = math.floor, math.sin, math.cos, math.tan

if not math.round then
  function math.round(x)
    return floor(x + 0.5)
  end
end

if not math.div then
  function math.div(n,m)
    return floor(n/m)
  end
end

if not math.mod then
  function math.mod(n,m)
    return n % m
  end
end

local pipi = 2*math.pi/360

function math.sind(d)
  return sin(d*pipi)
end

function math.cosd(d)
  return cos(d*pipi)
end

function math.tand(d)
  return tan(d*pipi)
end
```

Interface files

Interface files are files in XML format where the syntax of commands whith it's possible parametrs are described. These files gives only a overview of a command syntax but not what's the result of a parameter or in which way they have to be combined.

This feature is not limited to source file and you can use it in any document when you load the corresponding module with

```
\usemodule[int-load]
```

The file with the definition can then be loaded with

```
\loadsetups[<filename>]
```

```
\usemodule[int-load]

\loadsetups[t-fancybreak.xml]

\showsetup{fancybreak}


  \fancybreak {...*...}

  *   CONTENT
```

The XML file

The XML does always follow the same structure, at the comes is the XML declaration.

The definitions for the command defintion are enclosed in a pair if “interface” tags:

```
<?xml version="1.0" encoding="UTF-8"?>

<cd:interface
xmlns:cd="http://www.pragma-ade.com/commands"
name="context" language="en" version="2010.08.30">

...

</cd:interface>
```

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

1   IDENTIFIER

2   width         =  DIMENSION
    height        =  DIMENSION
    voffset       =  DIMENSION
    hoffset       =  DIMENSION
    page          =  left yes right
    commands      =  COMMAND
    doublesided   =  yes no empty
    headerstate   =  normal stop start empty none
                     nomarking
    footerstate   =  normal stop start empty none
                     nomarking
    textstate     =  normal stop start empty none
                     nomarking
    topstate      =  stop start
    bottomstate   =  stop start
    pagestate     =  stop start
    color         =  IDENTIFIER
```

First Example

This is a simple example with a command without any parameter.

```
<cd:command name="donothing" file="syst-aux.mkiv">
    <cd:sequence>
        <cd:string value="donothing"/>
    </cd:sequence>
</cd:command>
```

To produce the formatted output of this code write

```
\showsetup{donothing}
```

in your document.

```
\donothing
```

Environments

Even environments are possible:

```
<cd:command name="text" type="environment"
file="core-job.mkiv">
    <cd:sequence>
        <cd:string value="text"/>
    </cd:sequence>
</cd:command>
```

Although the name states “text” because of the environment type a “start” is added and you need “starttext” to show the output:

```
\showsetup{starttext}
```

```
\starttext ... \stoptext
```

Generated commands

When you generate a new float with `\definefloat` a `\place...` command is generated. To make clear the the command applies to all `\place...` commands the variable part is printed uppercase.

```
<cd:command name="placelistoffloats" generated="yes"
file="strc-flt.mkiv">
    <cd:sequence>
        <cd:string value="placelistof"/>
        <cd:variable value="floats"/>
    </cd:sequence>
</cd:command>
```

To produce the output in the document a “*” is added after the name of the defintion:

```
\showsetup{placelistoffloats*}
```

\placelistofFLOATS

Arguments

When a command takes a argument additional elements are written after the sequence block between the arguments elements.

```
<cd:command name="high" file="core-fnt.mkiv">
    <cd:sequence>
        <cd:string value="high"/>
    </cd:sequence>
    <cd:arguments>
        <cd:content n="1"/>
    </cd:arguments>
</cd:command>
```

The name for the definition depends here as well and the attributes environment and generated.

```
\showsetup{high}
```

```
\high {.*.}

*   CONTENT
```

Keywords

Besides arguments between braces also brackets are possible:

```
<cd:command name="defineblock" file="strc-blk.mkiv">
    <cd:sequence>
        <cd:string value="defineblock"/>
    </cd:sequence>
    <cd:arguments>
        <cd:keywords n="1" list="yes">
            <cd:constant type="cd:name"/>
        </cd:keywords>
    </cd:arguments>
</cd:command>
```

The command for the other frame is:

```
\showsetup{defineblock}
```

```
\defineblock [...,*,...]

*   IDENTIFIER
```

Assignments

Besides keywords assignments are also possible.

```
<cd:command name="setupfancybreak"
file="t-fancybreak.tex">
    <cd:sequence>
        <cd:string value="setupfancybreak"/>
    </cd:sequence>
    <cd:arguments>
        <cd:assignments n="1" list="yes">
            <cd:parameter name="spacebefore">
                <cd:inherit name="blank" n="1"/>
            </cd:parameter>
            <cd:parameter name="spaceafter">
                <cd:inherit name="blank" n="1"/>
            </cd:parameter>
            <cd:parameter name="align">
                <cd:inherit name="setupalign"
n="1"/>
            </cd:parameter>
        </cd:assignments>
    </cd:arguments>
</cd:command>
```

```
\showsetup{setupfancybreak}
```

```
\setupfancybreak [..,.*=.,..]

*   spacebefore  =  inherits from \blank
    spaceafter   =  inherits from \blank
    align        =  inherits from \setupalign
```

Defintions

Often used values for assignments and keywords can be saved in a defintion and be resued later in other code defintion with

```
<cd:resolve name="..."/>
```

The values itself are normal constants which are written between “define” elements which have the form:

```
<cd:define name="..."/>
    <cd:constant type="..."/>
    ...
</cd:define>
```

```
<cd:define name="align">
    <cd:constant type="inner"/>
    <cd:constant type="outer"/>
    <cd:constant type="left"/>
    <cd:constant type="right"/>
    <cd:constant type="flushleft"/>
    <cd:constant type="flushright"/>
    <cd:constant type="middle"/>
    <cd:constant type="center"/>
    <cd:constant type="normal"/>
    <cd:constant type="no"/>
    <cd:constant type="yes"/>
</cd:define>
```

Elements

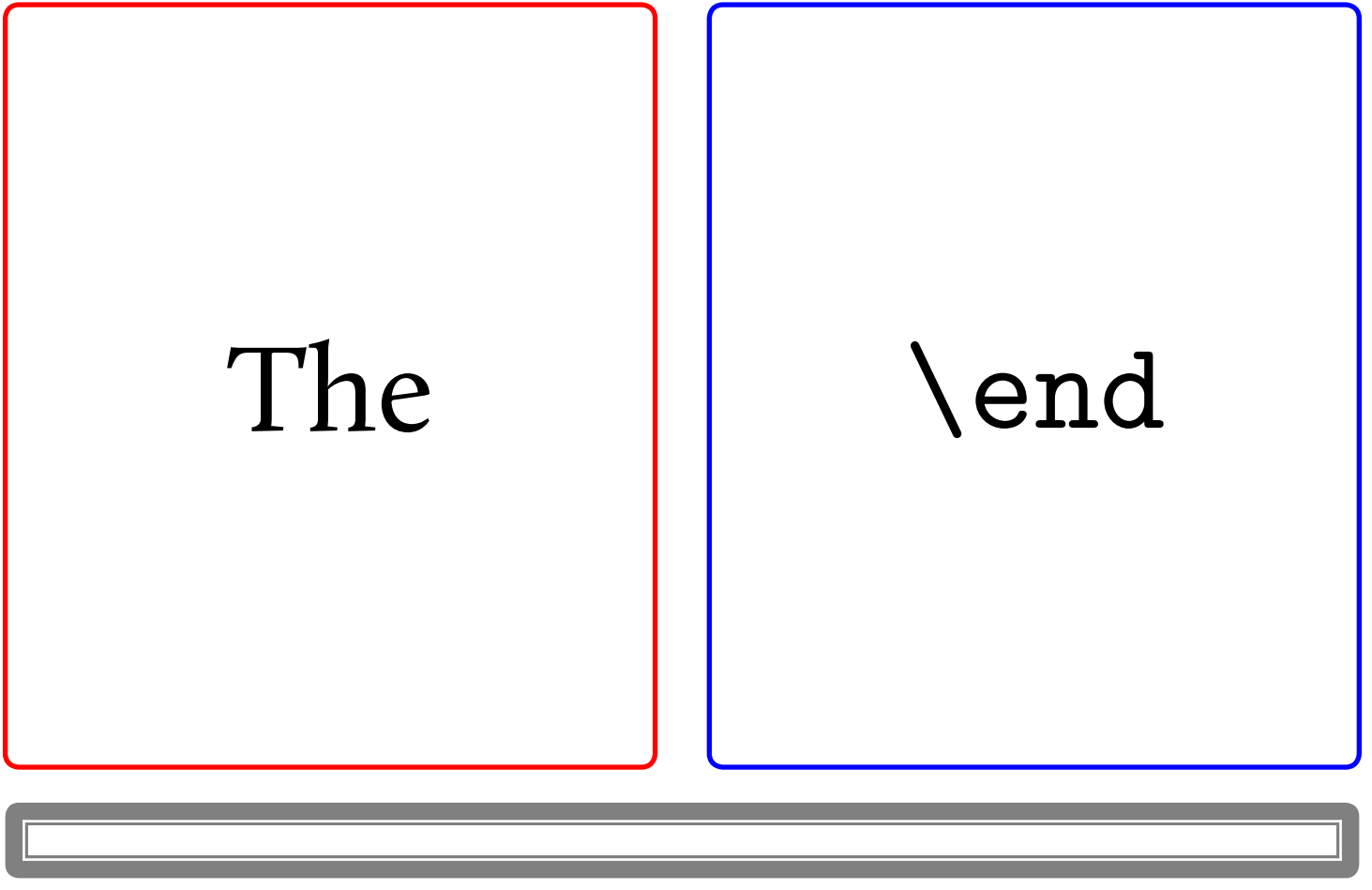
```
cd:assignments
cd:keywords
cd:displaymath
cd:index
cd:math
cd:nothing
cd:file
cd:position
cd:reference
cd:csname
cd:destination
cd:triplet
cd:word
cd:content
```

```
[.=.]           [..,.=.,..]
[...]           [...,...]
$$...$$         $$...$$
{...}           {..+...+..}
$...$           $...$
...             -
~...~           -
(...)           (...,...)
[...]           [...,...]
\...            -
[{..[ref]}]     [..,{..[ref]},..]
[x:y:z=]        [x:y:z=,..]
{...}           {... ... ...}
{...}           {... ... ...}
```

Attributes

```
cd:command
cd:dimension
cd:file
cd:name
cd:character
cd:mark
cd:number
cd:reference
cd:plural
cd:singular
cd:text
cd:formula
cd:matrix
cd:list
cd:section
cd:noargument
cd:oneargument
cd:twoarguments
cd:threearguments
```

```
COMMAND
DIMENSION
FILE
IDENTIFIER
-
MARK
NUMBER
REFERENCE
PLURAL NAME
SINGULAR NAME
TEXT
FORMULA
N*M
LIST
SECTION
\...
\...#1
\...#1#2
\...#1#2#3
```


The
\end