

How to Write a Module

Hraban Ramm & Wolfgang Schuster

Modules are reusable code and should adhere to the same conventions as ConT_EXt. How can you achieve all that fancy `\setup` and key-value parameter stuff?

1. Introduction

Hraban found his code too ugly and asked for a workshop to learn how to make it better. Wolfgang took pity and delivered.

While nothing stops you from writing modules for your own needs, this article is aimed at public modules. We'll start with the 'philosophy':

1.1 When should you write a public module?

A module is a collection of reusable code that is general enough that it may be of use for others, but complex enough that it's more than a few settings.

1.2 What to aim for

Your module should follow common ConT_EXt practices, as far as it makes sense: `\define` an instance, `\setup` key-value parameters, provide a `\start ... \stop` environment as well as a short `\command`.

Your code is only useful for others if it's well documented and if you support it for at least a while – answer to questions, fix bugs, adapt it to new ConT_EXt versions and maybe enhance it.

1.3 What to avoid

Your module must not interfere with core features or commands. If there's something in the distribution that you would like to fix, provide a fix and not a workaround module.

Usually it makes no sense to offer options for settings that are better handled by default setup commands. E. g. don't offer a range of font presets instead of leaning on `\setupbodyfont`. Of course you're welcome to suggest settings that make sense for your module and to document limitations.

Don't create modules for just a few settings – that might make sense for private code (environment or module), but we don't want to follow the L^AT_EX fashion of micropackages that just exist because the settings are too obscure.

Avoid undocumented low level code, because it's prone to change, instead of high level commands that are usually stable.

Don't publish modules if you don't want to support them (or find someone else who's interested in it and takes over).

Font support modules are usually unnecessary. Better publish a typescript on the wiki. Support for symbols or unusual scripts might make sense though.

2. Our example

Hraban published a preliminary module named “chat” for depicting messenger conversations in a youth novel. Wolfgang extended this idea – we're pretending to write a module for typesetting chats in the style of specific messengers. We won't cover the actual design here and just assume that we need frames for speech bubbles.

Besides `messenger` we'll use `CMD` as a placeholder, because it's shorter and easily distinguishable from surrounding code.

3. What's in a module?

3.1 File naming conventions

For compatibility (also with CTAN), use 7 bit ASCII file names with only one dot and no other interpunction than hyphens (don't use underscores).

You need a “code word” as module ID. In our examples, this will be “messenger”.

Choose a short and unique name that hints at the function or is at least original. Avoid naming conflicts with core files and other modules.

Usually, a module has one “entry point” or “main file” that is loaded with `\usemodule`. This file starts with a code letter:

- `m`: core module – not for you
- `p`: private code – unpublished, use at will
- `s`: style – an environment that is to be used unchanged
- `x`: xml – support for one flavour of XML
- `t`: third party – usually what you would publish

File extensions for the module file can be:

- `.tex`: generally usable
- `.cld`: ConTEXt Lua document, if it's ConTEXt-style Lua code
- `.lua`: general Lua code
- `.mkiv`, `.mkx1`: if you want to distinguish code for MkIV and LMTX
- `.mkvi`, `.mk1x`: MkIV / LMTX with named macro parameters

Of course you can use other file extension for additional files.

Our example module file is named `t-messenger.tex`.

Independent of code letter and file extension, we can use `\usemodule[messenger]`.

contextgroup > context meeting 2024

3.2 Directory structure

It makes sense to keep your module in a version control system such as Git, Subversion or Mercurial. We'll refer to Git, since it's the most common one.

The same rules apply to a Git repository or a ZIP file, their folder structure is defined by TDS (T_EX directory structure).

The following **top level files** are required for publication and get deleted or moved to doc on installation:

- README(|.txt|.md): a simple text or Markdown file, “homepage” text of a public git repository (see below) and on CTAN¹; see section 7.1
- VERSION: contains only the version date, preferably as a date like 2024.04.01 (CTAN prefers hyphens)
- LICENSE: license text (plain text); see section 3.3

There can be more top level files that might be useful for publication on CTAN, e. g. preview images².

- Your **code files**, i. e. everything that does the actual work of your module, belong in `tex/context/third/modulename`³. Often this is your only module file.
- **Documentation** goes into `doc/context/third/modulename`.
- An **interface** XML (see section 7.4) file would be nice, it belongs in `tex/context/interface/third`.
- In case you provide **scripts** that extend `mtxrun`, they belong in `scripts/context/luathird/modulename`.
- If your module needs special **fonts**, put them into `fonts/data/modulename`.

Don't publish unnecessary files! Leave out everything that is created automatically (e. g. `.log`, `.tuc`, `m_k_i_v_*`), backups (`.bak`, `.sav` ...) or anything that is only relevant to other programs or the operating system (`.DS_Store`, `.project` ...). In Git, list these in a `.gitignore` file.

3.3 Licensing

Even if you don't care about what happens with your code, choose appropriate licenses for your code. (If you don't define a license, everyone must assume standard copyright.)

The ConT_EXt sources are licensed under GPLv2 or the LPPL, so you might want to stick to these. ConT_EXt documentation is double-licensed under GNU FDL 1.3 and CC-BY-SA 3.0.

¹ “Comprehensive T_EX Archive Network”, ctan.org

² ctan.org/help/teaser-images

³ The additional level `tex/context/modules/third/modulename` would make sense, but since modules are usually installed in `texmf-modules` and most existing modules omit the `modules` folder, we'll continue to omit it.

If you include files by others, check their license! Don't include copyrighted material!

4. Module template

The following can be used as a template for the header and structure of a module (main file):

```
1 %D \module
2 %D   [   file=<filename>, % e.g. t-messenger
3 %D     version=<module version>, % e.g. 2024.04.01
4 %D     title=<module title>, % e.g. \CONTEXT\ User Module
5 %D     subtitle=<module subtitle>, % short description
6 %D     author=<author>, % name(s)
7 %D     email=<author email>,
8 %D     date=<date>, % e.g. \currentdate
9 %D     copyright=<copyright>, % e.g. <author>
10 %D     license=<license>] % e.g. 'GNU GPL 2.0'
11
12 %C copyright/license information
13
14 % begin info
15 %
16 % title   : <a short title to explain the module>
17 % comment : <a longer description of the purpose of the module>
18 % status  : <current status of the module>
19 %
20 % end info
21
22 \unprotect
23
24 % module code
25
26 \protect
27
28 \continueifinputfile {<filename>}
29
30 % usage examples
31
32 \endinput
```

4.1 Preamble

The entries type, subtitle, and author are mandatory and get used for the title page of the code documentation (??).

The entries file, version, copyright, license, and email are optional.

contextgroup > context meeting 2024

4.2 Info block

In addition to the preamble it's suggested to add an information block which gives a short description about the module and its status. It is used with `context --showmodules`.

4.3 Examples

Besides using the documentation mechanism and creating a separate document to explain the module, it is possible to add code at the end of the file after `\continueifinputfile` which is ignored when the module is loaded with `\usemodule`.

The argument of `\continueifinputfile` takes the filename of the module. Content after that is processed when you process the module itself (??).

4.4 Private area

To ensure that the module's commands can't be changed by users and to enable the multilingual interface, ConTeXt permits special characters in command names between `\unprotect` and `\protect` (see 5.).

In this 'unprotected' area, you can use `?`, `!`, `@` and `_` as part of command names, e. g.

```
\example_start
\????example
\c!location
```

5. Namespaces and setups

Namespaces group commands that belong together, e. g. for `\color`, `\startcolor`, `\definecolor`, and `\setupcolors`⁴, the namespace is 'color'. How do we get such a command set?

5.1 Rise of the command handler (history)

In ye olden MkII⁵ tymes and early MkIV⁶ days⁷, all `\define` and `\setup` commands had to be written manually:

```
\def\definemessenger
  {\dotripleempty\dodefinemessenger}

\def\dodefinemessenger[#1] [#2] [#3]%
  {...}
```

⁴ In this case, it's the plural form, you'll condone that for the sake of simplicity.

⁵ MkII was the ConTeXt version from more than 20 years ago, still based on 8 bit pdfTeX and without Lua.

⁶ MkIV is the ConTeXt version that's based on LuaTeX; it's frozen and only seldomly gets bugfixes.

⁷ ... when small furry creatures from Alpha Centauri were real small furry creatures from Alpha Centauri ...

```
\def\setupmessenger
  {\dodoubleempty\dosetupmessenger}

\def\dosetupmessenger[#1][#2]%
  {...}
```

That's quite tedious, and the frequent indirections with `\do` and `\dodo` macros led to the adoption of the dodo as a mascot for ConT_EXt.

Now, if you change the parameters of a command or environment with a `\setup` command, the value has to be stored in a macro.

The setting

```
\setupmessenger[width=10cm]
```

was achieved with

```
\getparameters[messenger][width=10cm]
```

which resulted in the internal representation

```
\def\messengerwidth{10cm}
```

(Yes, `\getparameters` actually *sets* parameters.)

But MkII is long gone, and the dodo is mostly extinct even in ConT_EXt. To make it easier to create these commands, to keep names unique and to ensure that macros are protected from user changes, namespaces and a mechanism called ‘command handler’ were added to ConT_EXt⁸.

5.2 Namespaces

In current ConT_EXt versions (MkIV and LMTX), you can use `\installnamespace{...}` to reserve one or more namespaces for a module.

With

```
\installnamespace {messenger}
```

you get a protected macro `\???messenger` to use in the upcoming definitions.

If you would expand this macro, you'd get something like `\2DD>`, which cannot be used in normal documents.

When we replace the assignment from section 5.1 with

```
\getparameters[\???messenger][width=10cm]
```

we now get the internal representation

```
\def\2DD>width{10cm}
```

⁸ ... by the Great Old Ones, when the stars were right, of course.

contextgroup > context meeting 2024

The functionality is mostly the same, but nobody can accidentally interfere with our parameters.

5.3 \define handler and inheritance

When we have a namespace, we'll first create a \define command to be able to create custom instances.

This doesn't make sense for every kind of command or environment, just if you need the *inheritance* from a basic construct. But if we didn't introduce this concept here, you wouldn't understand some of the following definitions.

```
\installdefinehandler \????CMD {CMD} \????CMD
```

creates \defineCMD[instance][parent][key=value,] which has two optional arguments that can be used to create a new instance or make a copy of an existing instance and set the default values for that new instance.

The names of the instance and its parent can be accessed with \currentCMD and \currentCMDparent:

Example

We create a *messenger* environment and want single instances for different messengers. The instance telegram inherits settings from its 'parent' signal that was defined first:

```
\installnamespace {messenger}
\installdefinehandler \????messenger {messenger} \????messenger
\definemessenger[signal][width=17dk]
\definemessenger[telegram][signal][color=blue]

\startsignal % within the signal instance
\currentmessenger      % = signal
\currentmessengerparent % = messenger
\stopsignal

\starttelegram % within the telegram instance
\currentmessenger      % = telegram
\currentmessengerparent % = signal
\stoptelegram
```

Define hooks

You can set parameters on definition of a new instance using \everypresetCMD and \everydefineCMD.

Before ConT_EXt reads the list of instance specific settings (\defineCMD[...][key=value]) on instantiation, it executes \everypresetCMD. This is useful e. g. to reset a counter.

\everydefineCMD is executed after \defineCMD. You can now check the values

and decide what to do with them. This is also used to create new commands (e. g. `\footnote`) for the document.

Example

From the previous example, we have the command `\definemessenger[...][...][...=...,]`, and we have defined *signal* as an instance of *messenger*, but that doesn't work automatically. You must employ the `\everydefinemessenger` hook, so that a custom environment is created when you call `\definemessenger`:

```
\appendtoks
  \setevalue{\e!start\currentmessenger}{}%
    \messenger_start{\currentmessenger}{}%
  \setevalue{\e!stop \currentmessenger}{\messenger_stop }%
\to \everydefinemessenger
```

To reset a counter before instantiation, as mentioned above:

```
\appendtoks
  \resetmessengerparameter\s!counter
\to \everypresetmessenger
```

5.4 What's this `\every...` and `\appendtoks` stuff?

I read `\appendtoks` as 'append to ks' (whatever *ks* would be), but it actually means 'append tokens'.

Token lists are the one and only data structure of T_EX. Such a list consists of characters (character tokens) and macro references (control sequence tokens).

An **a** is a token as well as `\emph`. Macros get expanded into their tokens step-by-step.

Some token lists are available for macro programming. Especially interesting are those that get inserted automatically in certain places, their names start with `\every`; e. g. `\everypar` gets inserted each time T_EX switches from a vertical mode into unrestricted horizontal mode, that means: at the beginning of a paragraph.

`\every...` commands are also called *hooks* because you can 'hook into' them to insert your own commands. ConT_EXt alleviates this with `\appendtoks ... \to\every...` ("add the following tokens to the token list that is inserted at every ...").

5.5 `\setup handler`

The next step is a `\setup` command to set and change values.

`\installsetuphandler\????CMD{CMD}` creates `\setupCMD[...][...=...,]` with one optional argument to indicate the instance.

Additionally, you get `\setupcurrentCMD[...=...,]` to change a value within a command or environment.

The name of the current instance is again accessible as `\currentCMD`. Additional

contextgroup > context meeting 2024

settings can be applied with `\everysetupCMD` and `\everysetupCMDroot`.

‘Root’ means the base instance, without using `\define`; in our example it’s *messenger*. If you use `\setup` without mentioning an instance name, the settings are applied to all instances.

Example

We want to assign default values to the *messenger* environment and let users change them.

```
\installsetuphandler \????messenger {messenger}
\setupmessenger[width=99ts,color=blue]
% We can still override the settings:
\setupmessenger[signal][width=17dk]
```

Now we have our `\setupmessenger[...][...=...,]` command and also a local `\setupcurrentmessenger[...=...,]` command.

5.6 Parameter handlers

The last step is to provide a way to access the values of the `\setup` command.

```
\installparameterhandler \????CMD {CMD}
```

creates:

- `\currentCMD` (as before)
- `\CMDparameter{key}`
- `\namedCMDparameter{instance}{key}`
- `\detokenizedCMDparameter{key}`
- `\directCMDparameter{key}`
- `\letfromCMDparameter\...{...}`

Example

Finally we want to access the values of the *messenger* environment:

```
\installparameterhandler \????messenger {messenger}

\messengerparameter{width}
\namedmessengerparameter{signal}{width}

% e.g. within the styling code:
\framed[width=\messengerparameter{width}]{...}
```

Root parameter handler

In case we want to access the root values of a command, we can create additional handlers:

```
\installrootparameterhandler \????CMD {CMD}
```

creates `\detokenizedrootCMDparameter{...}` and `\rootCMDparameter{...}`.

If you use `\CMDparameter`, it checks if there is an instance specific value, if not, you'll get the root value. With `\rootCMDparameter`, you'll directly get the root value, while with `\directCMDparameter` you'll get instance values without fallback.

If you set `\currentCMD` to empty, all three parameter commands will give you root values.

5.7 Style and color handler

For many commands and environments, styling by the usual `style` and `color` keys makes sense.

After we have established a way to create new commands and environments as well as set and access their parameters, we still lack a way to apply these settings.

```
\installstyleandcolorhandler \????CMD {CMD}
```

creates

- `\CMDparameter{...}`
- `\useCMDstyleandcolor{...}{...}`
- `\useCMDstyleparameter{...}`
- `\useCMDcolorparameter{...}`

Example

We can now use the style and color mechanism with *messenger* setups.

```
\installstyleandcolorhandler \????messenger {messenger}
\usemessengerstyleandcolor {style} {color}
\usemessengerstyleparameter {style}
\usemessengercolorparameter {color}
```

Parameter names

Usual behavior for the `style` and `color` parameters is to affect the (main) text in your command or environment, but you're not limited to those. Every key that you use with one of the `\use` commands above will understand the usual styling shortcuts.

So it's up to you which parameters you use for styling, but try to stay close to Con-TeXt's conventions.

```
\setupmessenger[
  style={\ss\tfx},
  titlestyle=bold,
  titlecolor=orange,
  framecolor=white,
]
{\usemessengerstyleandcolor{style}{color}
  styled text}
```

contextgroup > context meeting 2024

```
\framed[
  style=\usemessengerstyleparameter{style},
  color=\usemessengercolorparameter{color},
  framecolor=\usemessengercolorparameter{framecolor},
]{
\usemessengerstyleandcolor{titlestyle}{titlecolor}my styled title
}\par

other styled text
}
```

5.8 Parameter set handler

In some cases you want to change the values of a single key.

```
\installparametersethandler \????CMD {CMD}
```

creates:

- \currentCMD (as before)
- \setCMDparameter{key}
- \setexpandedCMDparameter{instance}{key}
- \letCMDparameter{key}\...
- \resetCMDparameter{key}

Example

We want a direct way to change *messenger* values.

```
\installparametersethandler \????messenger {messenger}
\setmessengerparameter {key} {content}
\letmessengerparameter {key} \...
\resetmessengerparameter {key}
```

5.9 Inherit from \framed

Like many features of ConT_EXt are based on \framed, we often want to use it for our own commands and environments.

```
\installinheritedframed {CMD}
```

creates:

- \currentCMD (as before)
- \CMDparameter{key}
- \CMDparameterhash{key}
- \setupcurrentCMD[key=value,]
- \inheritedCMDframed{key}
- \inheritedCMDframedbox{key} content

Example

We want a new `\framed` command which takes its values from `\setupmessenger`.

To avoid clashes with other *messenger* settings, the namespace and command name should be different from the default one.

```
\installnamespace {messengerframe}
\installinheritedframed \???messengerframe {messengerframe}
```

This creates `\inheritedmessengerframeframed{content}`, which is `\framed` with custom settings.

5.10 Basic parameter handler

To ease the creation of a new command or environment, `ConTEXt` combines multiple handlers into a single `\install...handler` command.

The first one is

```
\installbasicparameterhandler \???CMD {CMD}
```

which combines all commands for parameter access:

- `\installparameterhandler`
- `\installparameterhashhandler`
- `\installparametersethandler`
- `\installrootparameterhandler`

5.11 Command handler

The main handler used by most commands is

```
\installcommandhandler \???CMD {CMD} \???CMD
```

which allows the creation of new commands, changing values and access to all values. It includes:

- `\installbasicparameterhandler`
- `\installdefinehandler`
- `\installsetuphandler`
- `\installstyleandcolorhandler`

Namespace inheritance

In most cases, you'll use the same namespace for the first and third argument of `\installcommandhandler`, but you can also create an inheritance relation.

In the example below, 'TWO' inherits from 'ONE', i. e. `\setupTWO` affects only 'TWO', while it gets root settings from `\setupONE`.

```
\installnamespace {one}
\installnamespace {two}
```

contextgroup > context meeting 2024

```
\installcommandhandler \????one {ONE} \????one  
\installcommandhandler \????two {TWO} \????one
```

5.12 Simple command handler

For simpler commands without dedicated instances you can use

```
\installsimplecommandhandler \????CMD {CMD} \????CMD
```

which lacks the `\define handler`, but still includes:

- `\installbasicparameterhandler`
- `\installsetuphandler`
- `\installstyleandcolorhandler`

5.13 Framed command handler

The last major handler is

```
\installframedcommandhandler \????CMD {CMD} \????CMD
```

which combines `\installcommandhandler` with the `\framed handler`:

- `\installcommandhandler`
- `\installinheritedframed`

5.14 Local variables

In some cases you want a mechanism to accept values as assignments which are local to the command or environment. This can be done with the predefined `\get-dummyparameters`.

```
\getdummyparameters[key=value]  
\dummyparameter{key}
```

Example

```
\def\messagetext[#1]{#2}%  
{\begingroup  
  \getdummyparameters[sender=,#1]%  
  \dummyparameter{sender}: #2%  
  \endgroup}  
  
\messagetext[Hraban]{Hello Wolfgang!}
```

6. Complete example

```
1 %D \module  
2 %D [ file=t-messenger,  
3 %D title=\CONTEXT\ user module,
```

```

4 %D subtitle=Messenger dummy,
5 %D version=2024.08.20,
6 %D author=Wolfgang Schuster,
7 %D copyright=Wolfgang Schuster,
8 %D license=Public Domain,
9 %D email=wolfgang.schuster.lists@gmail.com]
10 \unprotect
11 \installnamespace {messenger}
12 \installnamespace {messengerchat}
13 \installnamespace {messengerframe}
14 \installcommandhandler \????messenger {messenger}
15 \????messenger
16 \installcommandhandler \????messengerchat {messengerchat}
17 \????messengerchat
18 \installframedcommandhandler \????messengerframe {messengerframe}
19 \????messengerframe
20 \appendtoks
21 \edefcsname\!start\currentmessenger\endcsname{\startmessenger
22 [\currentmessenger]}%
23 \edefcsname\!stop \currentmessenger\endcsname{\stopmessenger
24 }%
25 \to \everydefinemessenger
26 \appendtoks
27 \normalexpanded{\definemessengerframe[\currentmessengerchat]}%
28 \to \everydefinemessengerchat
29 \newtoks\t_messenger
30 \tolerant\protected\def\startmessenger[#1]#*[#2]%
31 {\begingroup
32 \cdef\currentmessenger{#1}%
33 \ifparameter#2\or
34 \setupcurrentmessenger[#2]%
35 \fi
36 \expand\t_messenger
37 \usemessengerstyleandcolor\c!style\c!color}
38 \protected\def\stopmessenger
39 {\endgroup}
40 \letnothing\m_messengerchat_previous
41 \tolerant\protected\def\messenger_chat[#1]#:#2%

```

contextgroup > context meeting 2024

```
42  {\par
43  \begingroup
44  \cdef\currentmessengerchat {#1}%
45  \cdef\currentmessengerframe{#1}%
46  \ifx\m_messengerchat_previous\currentmessengerchat \else
47  % person has changed!
48  \fi
49  \ifcstok{\messengerchatparameter\c!location}\v!left
50  \leftaligned {\inheritedmessengerframeframed{#2}}%
51  \else
52  \rightaligned{\inheritedmessengerframeframed{#2}}%
53  \fi
54  \globallet\m_messengerchat_previous\currentmessengerchat
55  \endgroup}

56  \appendtoks
57  \let\chat\messenger_chat
58  \to \t_messenger

59  \protect

60  \continueifinputfile{t-messenger.mkx1}

61  \starttext

62  \usemodule[visual]

63  \definemessenger[signal]

64  \definemessengerchat [a] [location=left]
65  \definemessengerchat [b] [location=right]

66  \setupmessengerframe [width=.7tw,autowidth=force,align=yes]

67  \setupmessengerframe [a] [framecolor=red]
68  \setupmessengerframe [b] [framecolor=green]

69  \startsignal
70  \chat[a]{\fakewords{3}{8}}
71  \chat[b]{\fakewords{3}{8}}
72  \chat[a]{\fakewords{3}{8}}
73  \chat[b]{\fakewords{3}{8}}
74  \chat[b]{\fakewords{3}{8}}
75  \stopsignal

76  \stoptext
```

7. Documentation

7.1 README

A README file (usually plain text or Markdown) makes sense in a public Git repository and for publication on CTAN. It should contain:

- title of the module
- a short explanation what it does
- dependencies (ConT_EXt LMTX, fonts, other programs)
- a hint on installing (e. g. via `mtxrun`)
- an example of its use
- references to further documentation
- the list of authors/contributors
- license (name and link to the LICENSE file)

7.2 Manual

If a simple example and the self-documenting source code isn't enough to understand usage and options of your module, please include a manual (source and PDF). You can use the MyWay style.

7.3 Self-documenting sourcecode

T_EX files

In .tex files (and other files containing primarily T_EX code, e. g. .mkiv), any line beginning with a comment leader will be treated as a 'docstring'. Formatting is done via ordinary ConT_EXt commands.

```
%D Use this comment type for examples and explanations
%D of the module. Keep in mind that each comment block
%D creates a local group when you change settings.
```

```
%M Use this comment type to load additional modules
%M (even the one you're documenting at the moment)
%M or make layout changes, because unlike the previous,
%M type settings are global.
```

```
%C Use this comment type for text which should remain
%C invisible in the output, e.g. license information.
```

Docstrings, though they appear to the T_EX interpreter as ordinary comments, allow for pretty printing source code.

In order to generate a PDF with source code as a colorful listing and the docstrings as plain serif text, run `context` on your main file like this:

```
context --extra=module --result=messenger t-messenger
```

It is recommended to specify a filename for the resulting PDF file because the default

contextgroup > context meeting 2024

is context-extra.pdf.

Lua files

In Lua files (e.g. .cld) docstrings start with `-- [[ltx--` and end with `--ltx]]--`. Text inside those delimiters can be formatted using basic HTML tags. Ordinary comments are still treated as part of the source and therefore will be typeset inside the listing.

There's an option to create a PDF from Lua code, too:

```
context --ctx=x-ltx somename.lua
```

7.4 XML interface file

It would be nice if each module had an associated XML specification file (as in `/tex/context/interface/mkiv/i-*.xml`). Its purpose is a comprehensive listing of the optional and non-optional arguments accepted by macros defined in the module.

From the interface a good deal of documentation can be auto-generated, as are for instance the ConT_EXt Quick Reference and the initial input of the Command Reference, which itself started as a wikification of the now obsolete TeXshow.

When documenting your module, you can use

```
\usemodule[setups-basics]
\loadsetups[t-messenger.xml] % to load the file with definitions
\setup{messenger}
```

An example:

```
\type [...1...2...] {2...}
1 inherits: \setuptype
2 CONTENT
```

By default, this places a frame around the setup. If you want to get a gray background, like in the ConT_EXt documentation, add a setup like:

```
\setupframedtexts[setuptext][
  background=color,
  backgroundcolor=lightgray,
  frame=off,]
```

Apart from the existing XML files in the ConT_EXt tree, there is no documentation, so feel free to relay your questions to the mailing list.

At the moment, interface files of modules aren't used by any mechanism. It would make sense to integrate them into the command reference in the wiki and syntax highlighting for editors.

8. Publication and maintenance

Beware, since Taco is planning to re-write the module website, this information might be slightly outdated.

Please upload your module(s) to *modules.contextgarden.net*! Our server scripts handle distribution for ConT_EXt and CTAN.

1. Register an account, then you can login to the “member section”.
2. If you lost your password, please ask Taco.
3. Please read the help page (it’s mostly the same as this section).
4. Create a **new module** entry with a distinct name (e.g. “messenger”; this will become the internal ID) and fill in the metadata:
 - Title (misleading, best use the same as before) “messenger” (This will get used in filenames!)
 - Short and longer description (the short desc. is meant as the name and gets published e.g. in CTAN updates).
 - Home URL, if the module has a homepage, e.g. a wiki page or git repository.
 - Keywords (for CTAN search)
 - Type: Macro or font (Please don’t publish font modules any more! Style, Lua and XML modules will get added soon.)
 - Works with Mk... (please check)
 - License (there are licenses missing, we need to fix that)
 - Check “Put in download section” (yes please, allows installation by script)
 - Check “Put in ConT_EXt distribution” (ATM just a hint for the admins, no automation)
 - Check “Synch with CTAN” (yes please, makes it visible; ATM just a hint for the admins)
 - CTAN location: e.g. /macros/context/contrib/context-messenger
 - Comment: only for you and the server admins
5. Create a **new module version** from a ZIP upload/download or checkout from SVN or git
 - Log message: short information about changes in this version (if empty, latest commit message is used)
 - Version: usually YYYY.MM.DD (hyphens are also ok; please avoid other versioning schemas)
 - File upload / HTTP download URL: release file as ZIP, as outlined above
 - SVN/GIT URL: repository checkout, structured like the ZIP, as outlined above

As a module author, it makes sense to subscribe to the developers mailing list⁹.

⁹ lists.contextgarden.net/mailman3/lists/dev-context.ntg.nl/

9. Conclusion

ConT_EXt offers a set of helpful commands to create advanced custom environments. We didn't cover the possibilities to do this in Lua (`interface.implement`).

ConT_EXt's ecosystem also offers means to publish and install custom modules.

Carlo

Mikael

Ryszard

Riviera

Enjoying the evening sun

photos: Hraban