How to write a module

ConTEXt meeting 2024

Wolfgang Schuster

What is a module?

Reusable code which is used by module document unlike environment files which are tied to single documents.

- Magazine style, e.g. MAPS.
- Language specific settings, e.g. french or uppercase ß.
- Abbreviations, e.g. PDF.
- Missing functionality in ConTEXt, e.g. letters.

Modules are loaded with the `\usemodule` command:

```
\usemodule [...] [...,...] [..,..=..,..]
           OPT             OPT
1  m p s x t
2  FILE
3  KEY = VALUE
```

Example:

```
\usemodule[messenger]
```

Naming conventions

While modules can have simple names ending with `tex` it is recommended to add a prefix, it is also possible to use other file extensions.

File prefix

- `m` (core module)
- `p` (private code)
- `s` (style file)
- `x` (XML code)
- `t` (thid party)

File extension

- `mklx`
- `mkxl`
- `mkvi`
- `mkiv`
- `tex`
- `cld`
- `lua`

Example: `t-messenger.mkxl`.

Documentation

ConTEXt provides a way to include documentation to code of a module which can be used to create a formatted PDF output. To assist the process various comment variations are provided which create different results in the converted documentation file.

```
%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 commented in the output, e.g. license information
%C (when you include the GNU GPL header).

%S B
%S This comment type is used when you add text which
%S should ignored/skipped. It is necessary to use "B"
%S at the first and "E" at the last line.
%S E
```

Preamble

Each file begins with a preamble which lists information about the file.

```
%D \module
%D   [     file=t-drofnats,
%D         title=\CONTEXT\ user module,
%D      subtitle=How to write a module,
%D       version=2024.08.20,
%D        author=R. J. Drofnats,
%D     copyright=R. J. Drofnats,
%D       license=Public Domain,
%D         email=drofnats@sanserriffe]
```

Mandatory entries

- title
- subtitle
- author

Optional entries

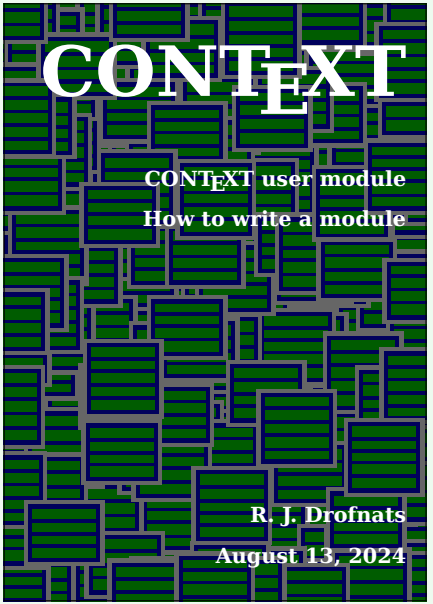
- file
- version
- copyright
- license
- email

Preamble

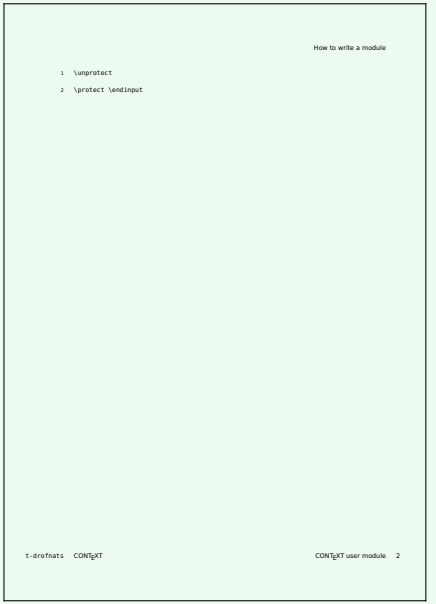
Preamble – Output

To create a PDF from the module with formatted output of the documentation use *module* extra. It is recommended to provide a filename for the resulting PDF file because the default is `context-extra.pdf`.

```
context --extra=module --result=... <file>
```



Cover



Normal page

Info block

In addition to the preamble it is possible to an information block which gives a short description about the module and its status.

```
% begin info
%
% title   : <a short title to explain the module>
% comment : <a longer description which explains the purpose of the module>
% status  : <current status of the module>
%
% end info
```

To see the result of these information blocks of all module use `showmodules` argument for the `context` runner.

```
context --showmodules
```

Adding examples at the end

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

With

```
\continueifinputfile {...*}
*   FILE
```

where the argument takes the filename of the module you can put content after the barrier which is processed when you pass the module itself as argument to the `context` process, e.g.

```
context t-messenger.mkxl
```

Using special character

To ensure module can't be changed by users and to allow the multilingual interface ConTEXt permits special characters for commands between `\unprotect` and `\protect`.

```
\unprotect
...
\protect
```

Between both commands you can use `?`, `!`, `@` and `_` as part of command names, e.g.

```
\messenger_start
\????messenger
\c!location
```

The command handler

In the old MkII and early MkIV days all \define and \setup commands had to be written by hand.

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

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

\def\setupmessenger
  {\dodoubleempty\dosetupmessenger}

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

To make the process to create these \define and \setup command easier ConTEXt added a mechanism called the command handler.

Namespaces

When you change the values of a command or environment with a `\setup` command the value has to be stored in a macro to be recalled later on.

The setting

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

could in the old times achieved with

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

which result in the following internal representation:

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

To ensure these internal macros are protected from user changes and to ensure names are unique ConTEXt added a namespace mechanism.

Namespace setting

To create a namespace for a command/environment there are

```
\installcorenamespace {...*}
*   NAME
```

for command of the ConTEXt core and

```
\installnamespace {...*}
*   NAME
```

for third party modules. When you use the mechanism to create a new *messenger* namespace with

```
\installnamespace {messenger}
```

you get the following command for usage of the later described mechanism

```
\????messenger
```

Namespace internals

As seen before a namespace uses 4 (or 2 for the core version) ? in front of the name but when you expand the command results in something like

```
\2DD>
```

which can't be used in normal documents and ensures no users can modify the values.

When we replace the previous assignment with

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

we get now the following internal representation:

```
\def\2DD>width{3cm}
```

\installdefinehandler

The first step to create a new command/environment is a \define command.

```
\installdefinehandler \...[1] {...[2]} \...[3]
1   CSNAME
2   NAME
3   CSNAME
```

This handler creates

defineCOMMAND [...] [...] [..=..]

which has two optional commands which can be used to

1. create a new instance and change the default values or
2. make a copy of an instance and change the default values.

The names of the instance or copy can be accessed with

currentCOMMAND
currentCOMMANDparent

and additional settings can be applied with

everypresetCOMMAND
everydefineCOMMAND

\define... example

We create a *messenger* environment and want different instances for different messengers.

```
\installdefinehandler \????messenger {messenger} \????messenger
```

Now we get a new \definemessenger command to create these instances.

```
                      1     2          3
\definemessenger [...] [...] [..,..=..,..]
                           OPT        OPT
```

```
1  NAME
2  NAME
3  inherits: \setupmessenger
```

With the help of the \everydefinemessenger hook a custom environment is created when you call \definemessenger.

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

\installsetuphandler

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

```
\installsetuphandler \...[1] {...[2]}
1   CSNAME
2   NAME
```

This handler creates

setupCOMMAND [...] [..=..]

with one optional arguments which allows to change instance values.

In addition you get

setupcurrentCOMMAND [..=..]

which is used to change value within a command/environment.

The name of the current instance can be assigned to

currentCOMMAND

Additional settings can be applied with

everysetupCOMMAND
everysetupCOMMANDroot

\setup... example

We want to assign default values or change them you the *messenger* environment.

```
\installsetuphandler \????messenger {messenger}
```

Now we get a dedicated \setupmessenger command.

```
                    1              2
\setupmessenger [...,...] [..,..=..,..]
                  OPT
```

```
1  NAME
2  KEY = VALUE
```

with a local \setupcurrentmessenger command

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

```
*  KEY = VALUE
```

\installparameterhandler

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

```
\installparameterhandler \...[1] {...[2]}

1   CSNAME
2   NAME
```

This handler creates

currentCOMMAND
COMMANDparameter {...}
namedCOMMANDparameter {...} {...}
detokenizedCOMMANDparameter {...}
directCOMMANDparameter {...}
letfromCOMMANDparameter \... {...}

\...parameter example

To finish it of we want to access the values for the *messenger* environment.

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

We can now access the values with

```
\messengerparameter {...}
```

```
*  KEY
```

or

```
\namedmessengerparameter {...} {...}
```

```
1  NAME
2  KEY
```

or

```
\directmessengerparameter {...}
```

```
*  KEY
```

\installrootparameterhandler

In case you want the access the root values of a command we can create additional handles.

```
\installrootparameterhandler \...[1] {...[2]}

1   CSNAME
2   NAME
```

This handler creates

\detokenizedrootCOMMANDparameter {...}
\rootCOMMANDparameter {...}

\root...parameter example

We can also access only the root values for *messenger*.

```
\installrootparameterhandler \????messenger {messenger}
```

This creates an additional parameter command.

```
                          *
\rootmessengerparameter {...}
```

`*` KEY

\installstyleandcolorhandler

After we established a way to create new command and environments, set their values and access we still lack a way to apply different styles and colors.

```
\installstyleandcolorhandler \...[1] {...[2]}

1   CSNAME

2   NAME
```

This handler creates

COMMANDparameter {...}
useCOMMANDstyleandcolor {...} {...}
useCOMMANDstyleparameter {...}
useCOMMANDcolorparameter {...}

\use...styleandcolor example

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

```
\installstyleandcolorhandler \????messenger {messenger}
```

The module can now apply the values with

```
\usemessengerstyleandcolor {...}[1] {...}[2]
```

1 KEY
2 KEY

or

```
\usemessengerstyleparameter {...}[*]
```

* KEY

or

```
\usemessengercolorparameter {...}[*]
```

* KEY

\installparametersethandler

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

```
\installparametersethandler \...[1] {...[2]}
1   CSNAME
2   NAME
```

This handler creates

currentCOMMAND
setCOMMANDparameter {...} {...}
setexpandedCOMMANDparameter {...} {...}
letCOMMANDparameter {...} \...
resetCOMMANDparameter {...}

\set...parameter example

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

```
\installparametersethandler \????messenger {messenger}
```

The module can now use

```
\setmessengerparameter {...}[1] {...}[2]
```

1 KEY
2 CONTENT

or

```
\letmessengerparameter {...}[1] \...[2]
```

1 KEY
2 CSNAME

or

```
\resetmessengerparameter {...}[*]
```

* KEY

\installinheritedframed

Because \framed is a ConTEXt mechanism which used in many cases we want to way to use it in our own commands.

```
\installinheritedframed {.*.}
*   NAME
```

This handler creates

```
\currentCOMMAND
\COMMANDparameter {...}
\COMMANDparameterhash {...}
\setupcurrentCOMMAND [..=..]
\inheritedCOMMANDframed {...}
\inheritedCOMMANDframedbox {...} ...
```

\inherited...framed example

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

```
\installinheritedframed \????messenger {messenger}
```

The module can now use

```
\inheritedmessengerframed {*...}
```

```
*  CONTENT
```

which is \framed with custom settings. To avoid clashes with other *messenger* settings the namespace and command name should be different from the default one.

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

\installbasicparameterhandler

To make the creation of a new command or environment easier ConTEXt combines multiple handlers in a single `\install..handler` command.

The first one is

```
\installbasicparameterhandler \...[1] {...[2]}
1   CSNAME
2   NAME
```

which combines all commands to access the values.

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

\installcommandhandler

The main handler used by most commands is

```
\installcommandhandler \...[1] {...[2]} \...[3]

1   CSNAME
2   NAME
3   CSNAME
```

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

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

\installsimplecommandhandler

For simpler commands without dedicated instaces one can use

```
\installsimplecommandhandler \...[1] {...[2]} \...[3]
1   CSNAME
2   NAME
3   CSNAME
```

which lacks the \define handler.

- \installbasicparameterhandler
- \installsetuphandler
- \installstyleandcolorhandler

\installframedcommandhandler

The last major handler is

```
\installframedcommandhandler \...[1] {...[2]} \...[3]
1   CSNAME
2   NAME
3   CSNAME
```

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

- `\installcommandhandler`
- `\installinheritedframed`

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 `\getdummyparameters`.

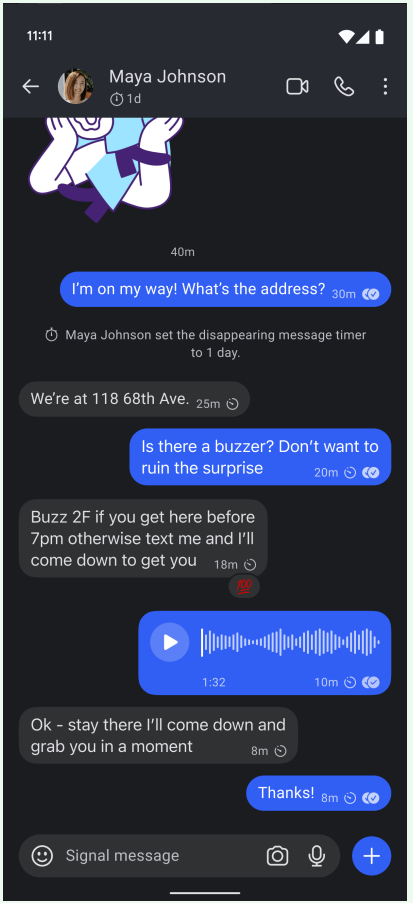
```
\getdummyparameters [..,..*=..,..]
*   KEY  =  VALUE
```

The values can be retrieved with

```
\dummyparameter {.*.}
*   KEY
```

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

Example – Signal Messenger

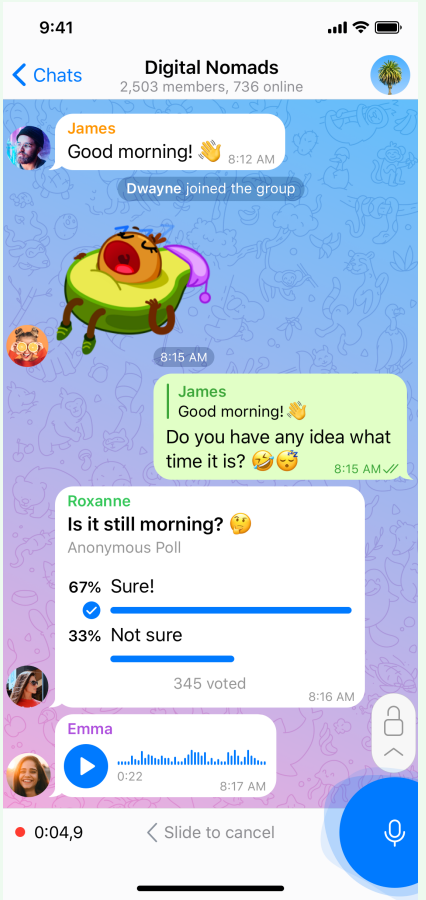


Chat

Group Chat

Example – Telegram Messenger

Chat



Group Chat