`mtxrun` scripts

Taco Hoekwater

The `mtxrun` command allows the execution of separate scripts. Most of these are written by Hans, and he occasionally creates new ones. This article will go through the `mtxrun` options, the scripts in the distribution, and show you how to write your own scripts.

1. Introduction

In ancient times, ConTEXt came with a bunch of perl (and later ruby) scripts to do various command-line tasks. There was `texexec`, `texutil`, and whole bunch of scripts with names ending in `tools`.

In more recent history, those separate scripts were all put under an umbrella script called `texmfstart`, which itself was eventually replaced with the current TEXLua-based script called `mtxrun`.

These days, `mtxrun` takes care of everything you would want to do with ConTEXt. It does most of that by running separate lua (sub)programs that implement various functionalities. In fact, even the shell command '`context`' itself is just a tiny wrapper around a call to `mtxrun`.

2. `Mtxrun` options

Calling `mtxrun` on the command-line with `--help` as single argument produces a rather long list of possible options. Some of those have sub-options, others have aliases, and then there are even some hidden options. Many options have functionality that is not immediately easy to understand, either.

At the time of writing this article, there are a total of 41 advertised top-level options.

Internally, the code actually looks at twice as many options, but quite some of those are either aliases or sub-options. Others are hidden because they really are only of interest to Hans himself.

2.1 Common flags

Much of the code inside MkIV can alter its behavior based on either 'trackers' (which add debugging information to the terminal and log output) or 'directives' or 'experiments' (for getting code to perform some alternate behavior). Since this also affects the lua code within `mtxrun` itself, it makes sense to list these options first.

Getting information on directives, trackers and experiments:

`--trackers`
: show (known) trackers

`--directives`
: show (known) directives

`--experiments`
: show (known) experiments

Enabling directives, trackers and experiments:

`--trackers=list`
: enable given trackers

`--directives=list`
: enable given directives

`--experiments=list`
: enable given experiments

The next tree (hidden) options are converted into 'directives' entries, that are then enabled. These are just syntactic sugar for the relevant directive.

`--silent[=...]`
: sets `logs.blocked={\%s}`

`--errors[=...]`
: sets `logs.errors={\%s}`

`--noconsole`
: sets `logs.target=file`

As you can see here, various directives (and even some trackers) have optional arguments, which can make specifying such directives on the command line a bit of a challenge. Explaining the details of all the directives is outside of the scope of this article, but you can look them up in the ConTEXt source by searching for `directives.register` and `trackers.register`.

In verbose mode, `mtxrun` itself gives more messages, and it enables the `resolvers.locating` tracker:

`--verbose`
: give a bit more info

The `--timedlog` (hidden) option starts the `mtxrun` output with a timestamp line:

`--timedlog`
: prepend output with a timestamp

2.2 Setup for finding files and configurations

The next block of options deals with the setup of `mtxrun` itself. You do not need to deal with these options unless you are messing with the ConTEXt distribution yourself instead of relying on a prepackaged solution, or you need to use kpathsea for some reason (typically in a MkII environment). In particular, `--progname` and `--tree` are often needed when using the `kpse` options.

`--configurations`
: show configuration order (alias `--show-configurations`)

`--resolve`
: resolve prefixed arguments (see `--prefixes`, below)

`--usekpse`
: use kpse as fallback (when no MkIV and cache installed, often slower)

`--forcekpse`
: force using kpse (handy when no MkIV and cache installed but less functionality)

`--progname=str`
: format or backend

`--tree=pathtotree`
: use given texmf tree (default file: `setuptex.tmf`)

2.3 Options for finding files and reporting configurations

Once the configuration setup is done, it makes sense to have a bunch or options to use and/or query the configuration.

`--locate`
: locate given filename in database (default) or system (uses the sub-options `--first`, `--all` and `--detail`)

`--autogenerate`
: regenerate databases if needed (handy when used to run context in an editor)

`--generate`
: generate file database

`--prefixes`
: show supported prefixes for file searches

`--variables`
: show configuration variables (uses the sub-option `--pattern`, and an alias is `--show-variables`)

`--expansions`
: show configuration variable expansion (uses the sub-options `--pattern`, alias `--show-expansions`)

`--expand-braces`
: expand complex variable

`--resolve-path`
: expand variable (completely resolve paths)

`--expand-path`
: expand variable (resolve paths)

`--expand-var`
: expand variable (resolves references inside variables, alias `--expand-variable`)

`--show-path`
: show path expansion of ... (alias `--path-value`)

`--var-value`
: report value of variable (alias `--show-value`)

`--find-file`
: report file location (uses the sub-options `--all`, `--pattern`, and `--format`)

`--find-path`
: report path of file

Hidden option:

`--format-path`
: report format path

2.4 Running code

Here we come to the core functionality of `mtxrun`: running scripts. First there are few options that trigger how the running takes place:

`--path=runpath`
: go to given path before execution

`--ifchanged=filename`
: only execute when given file has changed (this loads and saves an md5 checksum from `filename.md5`)

`--iftouched=old,new`
: only execute when given file has changed (time stamp)

`--timedrun`
: run a script or program and time its run (external)

Specifying both `--iftouched` and `--ifchanged` means both are tested, and when either one is false, nothing will happen. These options have to come before one of the next options:

`--script`
: run an mtx script (lua preferred method) (uses the sub-options `--nofiledatabase`, `--autogenerate`, `--load`, and `--save`. The latter two are currently no-ops.

`--execute`
: run a script or program externally (`texmfstart` method) (uses the sub-option `--noquotes`)

`--internal`
: run script using built in libraries (alias is `--ctxlua`)

`--direct`
: run an external program (uses the sub-option `--noquotes`)

Since scripts potentially have their own options, any options intended for `mtxrun` itself have to come *before* the option that specifies the script to run, and options for the script itself have to come *after* the option that gives the script name. This is especially true when using `--script`, so it is important to check the order of your options.

Of the four above options, `--script` is the most important one, since that is the one that finds and executes the lua `mtxrun` scripts provided by the distribution. With `--nofiledatabase`, it will not attempt to resolve any file names (which means you need either a local script or a full path name). On the opposite side, with `--autogenerate`, it will not only attempt to resolve the file name, it

will also regenerate the database if it cannot find the script on the first try. In a future version of ConTEXt, the `--load` and `--save` will allow you to save/restore the current command line in a file for reuse.

The shell return value of `mtxrun` indicates whether the script was found. When you specify something like `--script base`, `mtxrun` actually searches for `mtx-base.lua`, `mtx-bases.lua`, `mtx-t-base.lua`, `mtx-t-bases.lua`, and `base.lua`, in that order. The distribution-supplied scripts normally use `mtx-<name>.lua` as template. The template names with `mtx-t-` prefix is reserved for third-party scripts, and `<name>.lua` is just a last-ditch effort if nothing else works. Scripts are looked for in the local path, and in whatever directories the configuration variable `LUAINPUTS` points to.

The `--execute` options exists mostly for the non-lua MkII scripts that still exist in the distribution. It will try to find a matching interpreter for non-lua scripts, and it is aware of a number of distribution-supplied scripts so that if you specify `--execute texexec`, it knows that what you really want to execute is `ruby texexec.rb`. Support is present for ruby (`.rb`, lua (`.lua`), python (`.py`) and perl (`.pl`) scripts (tested in that order). File resolving uses `TEXMFSCRIPTS` from the configuration. The shell return value of `mtxrun` indicates whether the script was found and executed.

The `--internal` option uses the file search method of `--execute`, but then assumes this is a lua script and executes it internally like `--script`. This is useful if you have a lua script in an odd location.

The last of the four options, `--direct`, directly executes an external program. You need to give the full path for binaries not in the current shell `PATH`, because no searching is done at all. The shell return value of `mtxrun` in this case is a boolean based on the return value of `os.exec()`.

It is also possible to execute bare lua code directly:

`--evaluate`

run code passed on the command-line (between quotes)

2.5 Options for maintenance of `mtxrun` itself

None of these are advertised. Normally Hans should be the only one needing them, but if you made a change to one of the distributed libraries (maybe because of a beta bug), you may need to run `--selfmerge` and `--selfupdate`.

`--selfclean`

remove embedded libraries

`--selfmerge`

update embedded libraries in `mtxrun.lua`

`--selfupdate`

copy `mtxlua.lua` to the executable directory, renamed `mtxrun`

2.6 Creating stubs

Stubs are little shortcuts that live in some binaries directory. For example, the content of the unix-style `context` shell command is:

```
#!/bin/sh
mtxrun --script context "$@"
```

Apart from the `context` command itself (which is provided by the distribution), use of stubs is discouraged. Still, the `mtxrun` options are there because sometimes existing workflows depend on executable tool names like `ctxtools`.

`--makestubs`

create stubs for (context related) scripts

`--removestubs`

remove stubs (context related) scripts

`--stubpath=binpath`

paths where stubs will be written

`--windows`

create windows (mswin) stubs (alias `--mswin`)

`--unix`

create unix (linux) stubs (alias `--linux`)

2.7 Remaining options

The remaining options are hard to group into a subcategory.

2.7.1 Advertised options

`--systeminfo`

show current operating system, processor, et cetera

`--edit`

launch editor with found file. The editor is taken from the environment variable `MTXRUN_EDITOR`, or `TEXMFSTART_EDITOR`, or `EDITOR`, or as a last resort: `gvim`

`--launch`

launch files like manuals, assumes os support (uses the sub-options `--all`, `--pattern` and `--list`)

2.7.2 Hidden options

`--ansi`

colorize output to terminal using ansi escapes

`--associate`

launch files like manuals, assumes os support. This function does not do any file searching, so you have to use either a local file or a full path name

`--exporthelp`

output the `mtxrun` XML help blob (useful for creating man and html help pages)

`--fmt`

shortcut for `--script base --fmt`

`--gethelp`

attempt to look up remote ConTEXt command help (uses the sub-options `--command` and `--url`)

`--help`

print the `mtxrun` help screen

`--locale`

force setup of locale. Unless you are certain you need this option, stay away from it, because it can interfere massively with ConTEXt's lua code

`--make`

(re)create format files (aliases are `--ini` and `--compile`)

`--platform`

(alias is `--show-platform`)

`--run`

shortcut for `--script base --run`

`--version`

print `mtxrun` version

3. Known scripts

When you run `mtxrun --scripts`, it will output a list of 'known' scripts. The definition of 'known' is important here: the list comprises the scripts that are present in the same directory as `mtx-context.lua` that do not have an extra hyphen in the name (like `mtx-t-...` scripts would have). In a normal installation, this means it 'knows' almost all the scripts that are distributed with ConTEXt. Note: it skips over any files from the distribution that do have an extra hyphen, like the `mtx-server` support scripts.

Since this article is about `mtxrun`, I'll just present the list of the scripts that are 'known' in the current ConTEXt beta as output by `mtxrun` itself, and not get into detail about all of the script functionality (they all have `--help` options if you want to find out more). Where I still felt the need to explain something, there is an extra bit of text in italics.

`babel`

Babel Input To UTF Conversion

`base`

ConTeXt TDS Management Tool (aka luatools)

`bibtex`
: bibtex helpers (obsolete)

`cache`
: ConTeXt & MetaTeX Cache Management

`chars`
: MkII Character Table Generators

`check`
: Basic ConTeXt Syntax Checking
Occasionally useful on big projects, but be warned that it does not actually run any TEX engine, so it is not 100% reliable

`colors`
: ConTeXt Color Management
This displays icc color tables by name

`convert`
: ConTeXT Graphic Conversion Helpers
A wrapper around ghostscript and imagemagick that offers some extra (batch processing) functionality

`dvi`
: ConTeXt DVI Helpers

`epub`
: ConTeXt EPUB Helpers
The EPUB manual (`epub-mkiv.pdf`) explains how to use this script

`evohome`
: Evohome Fetcher
Evohome is a domotica system that controls your central heating

`fcd`
: Fast Directory Change

`flac`
: ConTeXt Flac Helpers
Extracts information from `.flac` audio files into an XML index

`fonts`
: ConTeXt Font Database Management

`grep`
: Simple Grepper

`interface`
: ConTeXt Interface Related Goodies

`metapost`
: MetaPost to PDF processor

`metatex`
: MetaTeX Process Management (obsolete)

`modules`
: ConTeXt Module Documentation Generators

`package`
: Distribution Related Goodies
This script is used to create the generic ConTEXt code used in LuaLATEX c.s.

`patterns`
: ConTeXt Pattern File Management
Hyphenation patterns, that is...

`pdf`
: ConTeXt PDF Helpers

`plain`
: Plain TeX Runner

`profile`
: ConTeXt MkIV LuaTeX Profiler

`rsync`
: Rsync Helpers

`scite`
: Scite Helper Script

`server`
: Simple Webserver For Helpers
There are some subscripts associated with this.

`synctex`
: ConTeXt SyncTeX Checker

`texworks`
: TeXworks Startup Script

`timing`
: ConTeXt Timing Tools

`tools`
: Some File Related Goodies

`unicode`
: Checker for `char-def.lua`

`unzip`
: Simple Unzipper

`update`
: ConTeXt Minimals Updater

`watch`
: ConTeXt Request Watchdog

`youless`
: YouLess Fetcher
YouLess is a domotica system that tracks your home energy use

4. Writing your own

4.1 File structure

A well-written script has some required internal structure. It should start with a module definition block. This gives some information about the module, but more importantly, it prevents double-loading.

Here is an example:

```
if not modules then modules = { } end
modules ['mtx-envtest'] = {
    version   = 0.100,
    comment   = "companion to mtxrun.lua",
    author    = "Taco Hoekwater",
    copyright = "Taco Hoekwater",
    license   = "bsd"
}
```

Next up is a variable containing the help information. The help information is actually a bit of XML stored in lua string. In the full example listing at the end of this article, you can see what the internal structure is supposed to be like.

```
local helpinfo = [[
<?xml version="1.0"?>
<application>
  ....
</application> ]]
```

And this help information is then used to create an instance of an `application` table.

```
local application = logs.application {
    name     = "envtest",
    banner   = "Mtxrun environment demo",
    helpinfo = helpinfo,
}
```

After this call, the `application` table contains (amongst some other things) three functions that are very useful:

`identify()`
: Prints out a banner identifying the current script to the user

`report(str)`
: For printing information to the terminal with the script name as prefix

`export()`
: Prints the `helpinfo` to the terminal, so it can easily be used for documentation purposes

Next up, it is good to define your scripts' functionality in functions in a private table. This prevents namespace pollution, and looks like this:

```
scripts          = scripts         or { }
scripts.envtest  = scripts.envtest or { }

function scripts.envtest.runtest()
    application.report('script name is ' ..
                       environment.ownname)
end
```

An finally, identify the current script, followed by handling the provided options (usually with an if-else statement).

```
if environment.argument("exporthelp") then
   application.export()
elseif environment.argument('test') then
   scripts.envtest.runtest()
else
    application.help()
end
```

The complete listing of this script is provided at the end of this article.

4.2 Script environment

mtxrun includes lots of the internal lua helper libraries from ConTEXt. Hans actually maintains a version of the script without all those libraries included, and before every (beta) ConTEXt release, an amalgamated version of mtxrun is added to the distribution. In the merging process, most all comments are stripped from the embedded libraries, so if you want to know details, it is better to look in the original lua source file.

Inside mtxrun, the full list of embedded libraries can be queried via the array own.libs:

```
l-lua.lua l-macro.lua l-sandbox.lua l-
package.lua l-lpeg.lua l-function.lua
l-string.lua l-table.lua l-io.lua l-
number.lua l-set.lua l-os.lua l-file.lua
l-gzip.lua l-md5.lua l-url.lua l-dir.lua
l-boolean.lua l-unicode.lua l-math.lua
util-str.lua util-tab.lua util-fil.lua
util-sac.lua util-sto.lua util-prs.lua
util-fmt.lua trac-set.lua trac-log.lua
trac-inf.lua trac-pro.lua util-lua.lua
util-deb.lua util-tpl.lua util-sbx.lua
util-mrg.lua util-env.lua luat-env.lua
lxml-tab.lua lxml-lpt.lua lxml-mis.lua
lxml-aux.lua lxml-xml.lua trac-xml.lua
data-ini.lua data-exp.lua data-env.lua
data-tmp.lua data-met.lua data-res.lua
data-pre.lua data-inp.lua data-out.lua
data-fil.lua data-con.lua data-use.lua
data-zip.lua data-tre.lua data-sch.lua
data-lua.lua data-aux.lua data-tmf.lua
data-lst.lua util-lib.lua luat-sta.lua
luat-fmt.lua
```

In fact, the lua table own contains some other useful stuff like the script's actual disk name and location (own.name and own.path) and some internal variables like a list of all the locations it searches for its embedded libraries (own.list), which is used by the --selfmerge option and also allows the non-amalgamated version to run (since otherwise --selfmerge could not be bootstrapped).

mtxrun offers a programming environment that makes it easy to write lua scripts. The most important element of that environment is a lua table that is conveniently called environment (util-env does the actual work for this).

The bulk of environment consists of functions and variables that deal with the command-line given by the user. mtxrun does quite a bit of work on the given command-line. The goal is to safely tuck all the given options into the arguments and files tables. This work is done by two functions called initializearguments() and splitarguments(). These functions are part of the environment table, but you should not need them as they have been called already once control is passed on to your script.

`arguments`

: These are the processed options to the current script. The keys are option names (without the leading dashes) and the value is either true or a string with one level of shell quotes removed.

`files`

: This array holds all the non-option arguments to the current script. Typically, those are supposed to be files, but they could be any text, really

`getargument(name,partial)`

: Queries the arguments table using a function. Its main reason for existence is the partial argument, which allows scripts to accept shortened command-line options. (alias: argument())

`setargument(name,value)`

: Sets a value in the arguments table. This can be useful in complicated scripts with default options.

In case you need access to the full command-line, there are some possibilities:

`arguments_after`

: These are the unquoted but otherwise

unprocessed arguments to your script as an array

`arguments_before`

These are the unquoted but otherwise unprocessed arguments to `mtxrun` before your scripts' name (so the last entry is usually `--script`)

`rawarguments`

This is the whole unprocessed command-line as an array

`originalarguments`

Like `rawarguments`, but with some top-level quotes removed

`reconstructcommandline(arg,noquote)`

Tries to reconstruct a command-line from its arguments. It uses `originalarguments` if no `arg` is given. Take care: due to the vagaries of shell command-line processing, this may or may not work when quoting is involved

`environment` also stores various bits of information you may find useful:

`validengines`

This table contains keys for `luatex` and `luajittex`. This is only relevant when `mtxrun` itself is called via LuaTEX's typeluaonly option.

`basicengines`

This table maps executable names to `validengines` entries

`default_texmfcnf`

This is the `texmfcnf` value from `kpathsea`, processed for use with MkIV in the unlikely event this is needed

`homedir`

The user's home directory

`ownbin`

The name of the binary used to call `mtxrun`

`ownmain`

The mapped version of `ownbin`

`ownname`

Full name of this instance of `mtxrun`

`ownpath`

The path this instance of `mtxrun` resides in

`texmfos`

Operating system root directory path

`texos`

Operating system root directory name

`texroot`

ConTEXt root directory path

As well as some functions:

`texfile(filename)`

Locates a TEX file

`luafile(filename)`

Locates a lua file

`loadluafile(filename,version)`

Locates, compiles and loads a lua file, possibly in compressed `.luc` format. In the compressed case, it uses the `version` to make sure the compressed form is up-to-date

`luafilechunk(filename,silent,macros)`

Locates and compiles a lua file, returning its contents as data

`make_format(name,arguments)`

Creates a format file and stores in in the ConTEXt cache, used by `mtxrun --make`

`relativepath(path,root)`

Returns a modified version of `root` based on the relative path in `path`

`run_format(name,data,more)`

Run a TEX format file

4.3 Shell return values

As explained earlier, the shell return value of `mtxrun` normally indicates whether the script was found. If you are running a ConTEXt release newer than September 2018 and want to modify the shell return value from within your script, you can

use `os.exitcode`. Whatever value you assign to that variable will be the shell return value of your script.

4.4 Example script

The full source code of the script used in the verbatim examples before is given below.

```
if not modules then modules = { } end modules ['mtx-envtest'] = {
    version   = 0.100,
    comment   = "companion to mtxrun.lua",
    author    = "Taco Hoekwater",
    copyright = "Taco Hoekwater",
    license   = "bsd"
}

local helpinfo = [[
<?xml version="1.0"?>
<application>
 <metadata>
  <entry name="name">mtx-envtest</entry>
  <entry name="detail">MTXRUN environment tester</entry>
  <entry name="version">0.10</entry>
 </metadata>
 <flags>
  <category name="basic">
   <subcategory>
    <flag name="test"><short>test environment table content</short></flag>
   </subcategory>
  </category>
 </flags>
 <examples>
  <category>
   <title>Examples</title>
   <subcategory>
    <example><command>mtxrun --script envtest --test</command></example>
   </subcategory>
  </category>
 </examples>
 <comments>
  <comment>initial version</comment>
 </comments>
</application>
]]

local application = logs.application {
    name     = "envtest",
    banner   = "Mtxrun environment demo",
    helpinfo = helpinfo,
}

scripts             = scripts             or { }
scripts.envtest     = scripts.envtest     or { }

function scripts.envtest.runtest()
    application.report('script name is '.. environment.ownname)
end

application.identify()

if environment.argument("exporthelp") then
   application.export()
elseif environment.argument('test') then
   scripts.envtest.runtest()
else
    application.help()
end
```