

The `glyph`^{*} package

Chris Waltrip[†]

Released 2023-08-04

1 Introduction

I had a conundrum. I wanted to use icon fonts in some of my work. I could use the `fontawesome5` package (and indeed I have!). But I wanted the ability to use new glyphs, and new styles, and `fontawesome5` is well... version 5. And really what I wanted to do was dip my feet into the world of L^AT_EX package writing, so I started working on a package I was going to call `fontawesomex`, and it would support all of the current versions of *Font Awesome*. But then I realized that I wanted to use glyphs from other icon fonts. So I changed the name to `glyph` and that's where we're at now!

The idea with this package is that it'll scan the fonts you provide in the options and create commands for every printable glyph it finds that has a printable name. The names of the commands are created based on a few basic rules; on the plus-side this means predictability no matter the font, but with the downside that some glyph names may be a bit unwieldy.

I also wanted a way to be able to print the glyphs and their glyph names as a debugging tool, but I think it can help users of this package determine how to call the glyphs they're interested in printing, and so I expanded the command and exposed it publically so that you can create a full catalog of glyphs too!

2 Credits, Thanks & Ripoffs

This needs a better title, but I could not have done this without the unwitting (and unknowing) help of Marcel Krüger. The work he's done to create the `fontawesome5` package helped give me a much deeper understanding of `expl3`, and the different typesetting engines.

2.1 Known limitations and feature requests

This list isn't exhaustive, but there are some things you should know about this package before getting too excited. I'm using this section to store my TODOs and FIXMEs. I hope to get to them *eventually*TM.

- *dual glyphs* only support the Font Awesome style of glyph names ending in `-primary` and `-secondary`.

^{*}<https://github.com/latex-glyph>

[†]<https://github.com/cawaltrip>

- Arbitrary font features (e.g., ligatures) aren't supported.
- If there's a glyph called "glyph", we'll break.
- Lua_{La}T_EX isn't handling fixed width fonts.
- options can't be changed mid-document (specifically, *dualglyphFontsSecondaryColor*).
- the command that dumps the entire glyph table does not display *dual glyph* characters together.
- some of the document commands (at least the `\<prefix>Glyph<glyph-name>` commands) should be cleaned up and placed under a single command (e.g., `\PutGlyph[<prefix>]<glyph-name>`) in order to prevent colliding with a glyph named "glyph".
- the font family naming scheme is defined at the `glyph-functions-luatex` level and should instead be defined at the `glyph-functions` to prevent mixing up names when working on support for other engines.
- rename the internal "debug" functions, to more accurately depict their use (typesetting all the glyphs of a font).

Some things that I still need to understand a bit better and get answered/fixed.

- Which of my functions can be protected/robustified?
- Is the method I use to find C0/C1 glyphs going to work?
- Do I actually need `__glyph_bool_from_str`?
- Will right-to-left languages be able to use *dual glyphs*? How is LTR/RTL done in LaTeX?

3 Typesetting Engine Support

This package currently only works with Lua_{La}T_EX. I have plans for X_{La}T_EX in the future, but I don't know that I'll ever support pdf_{La}T_EX.

4 Documentation

4.1 Configuration

This package is currently pretty inflexible when it comes to configuration. Fonts and options need to be specified at the time the package is called, and it must be in the preamble. There are no ways to add/update options on the fly. For the most part, this is no problem. But if you are using a character that requires two glyphs stacked on top of one another (e.g., Font Awesome Duotone), or what I'll refer to from here on out as *dual glyphs*, the color of the second glyph can't be changed. Calling `glyph` should look like

`\usepackage[options]{glyph} \usepackage[options]{glyph}`

The [*options*] are a key–value list which can contain the following keys:

- `fonts=` { Comma-separated list of <prefix> = }
- `dualglyphFonts=` { Comma-separated list of <prefix> = }¹
- `dualglyphFontsSecondaryColor=` {a color in a form that xcolor accepts }

Glyphs can be called with either

- `\<prefix><GlyphName>`
- `\<prefix>Glyph<glyph-name>`²³

Example

If you wanted to load some *Font Awesome* fonts, and then call a few glyphs, you might use the following invocation.

```
\documentclass{article}
\usepackage{xcolor}
\definecolor{orange}{RGB}{255, 192, 0}
\usepackage [
  fonts = {
    fa = Font Awesome 6 Pro Regular,
    fas = Font Awesome 6 Pro Solid
  },
  dualglyphFonts = {
    fad = Font Awesome 6 Duotone,
  },
  dualglyphFontsSecondaryColor = {orange}
]{glyph}
\begin{document}
Do you want an awesome face? \faFaceAwesome \\
What about a glyph with illegal control \\
sequence characters? \fasTransporterOne \\
Duotone icons work natively too: \fadWandMagicSparkles.
\end{document}
```

Do you want an awesome face? 😊
What about a glyph with illegal control
sequence characters? 🚚
Duotone icons work natively too: ✨.

²This will be changing soon to help against collisions with glyphs named “glyph”

³This is broken right now anyways.

4.2 Dependencies

This package relies on

- `fontspec` for font family creation and general font management,
- `tcolorbox` for creating the boxes that glyphs are displayed in when dumping all glyphs, and
- `multicol` for arranging all said boxes.

Index

C

collect commands:	
collect_glyphs	15
collect_glyphs()	15
create commands:	
create_font_family	12
create_font_family()	12
create_glyph_commands	11
create_glyph_commands()	11
create_icon_command	11
create_icon_command()	11
create_showcase_command	12
create_showcase_command()	12

D

debug	15
debug()	15

G

generate commands:	
generate_font_family_name	14
generate_font_family_name()	14
generate_glyph_command_name	15
generate_glyph_command_name()	13
generate_icon_command_name	14
generate_icon_command_name()	14
generate_showcase_name	12, 14
generate_showcase_name()	14
get commands:	
get_glyph_base_name	15
get_glyph_base_name()	15
get_glyph_index	15
get_glyph_index()	15
get_unicode_blocks	15
get_unicode_blocks()	15

H

has commands:	
has_only_valid_characters	14
has_only_valid_characters()	14

I

in commands:	
in_printable_range	14
in_printable_range()	14

S

sanitize commands:	
sanitize_glyph	14
sanitize_glyph()	14
senerate commands:	
senerate_glyph_command_name	13

showcase commands:

showcase_block	13
showcase_block()	13
showcase_font	12
showcase_font()	12
showcase_glyph	13
showcase_glyph()	13
showcase_sample	13
showcase_sample()	13

T

T_EX and L^AT_EX₂_ε commands:

\huge	12
\normal	12
\section	12
\subsection	12, 13

U

\usepackage[options]{glyph}	3
-----------------------------	---



Messages submodule for `glyph`^{*}

Chris Waltrip[†]

Released 2023-08-04

5 Introduction

This submodule contains messages to print for the `glyph` package.

^{*}/latex-glyph
[†]/cawaltrip

Engine-agnostic functions submodule for `glyph`^{*}

Chris Waltrip[†]

Released 2023-08-04

6 Introduction

This submodule defines functions which are engine-agnostic. The functions are dedicated to creating the data structure of font information, parsing the fonts, and placing the glyphs.

6.1 Parsing

Users provide (through the keys declared in `glyph`), a list of fonts, prefixes for how to reference those fonts, and whether or not the characters are composed of one or two glyphs. That data is turned into a sequence map as described in the documentation for the `glyph` package.

Parsing this new data structure is how the document-level commands are created.

6.2 Placing glyphs

There are a couple functions that are used to place the glyph, based on the number of glyphs required to show a character.

7 Engine-specific Implementations

There are a number of functions that each engine needs to implement (i.e., functions that the functions defined in this package call). Some of these are named functions, and others are passed by control sequence name.

Named Functions

- `__glyph_parse_font:N`
- `__glyph_font_debug_info:nn`

^{*}/latex-glyph

[†]/cawaltrip

Unnamed Functions

- `_glyph_font_family` takes the value of a command that changes the font to the one needed for placing the specified glyph.

The system-specific implementations are free to define other functions as needed to perform these requirements.



LuaTeX functions submodule for `glyph`*

Chris Waltrip†

Released 2023-08-04

8 Introduction

This package defines LuaLaTeX-specific functions to finish the `glyph` package requirements. This package defines the interface defined in `glyph-functions`, namely `\@@_parse_font:N`, `\@@_font_debug_info:nn`, and creating a command sequence for `\@@_font_family`. There are additional helper functions defined to complete this.

* /latex-glyph
† /cawaltrip

LuaTeX implementation for glyph

Chris Waltrip*

Released 2023-08-04

Contents

1	Introduction	1
2	Credits, Thanks & Ripoffs	1
2.1	Known limitations and feature requests	1
3	Typesetting Engine Support	2
4	Documentation	2
4.1	Configuration	2
4.2	Dependencies	4
	Index	5
5	Introduction	6
6	Introduction	7
6.1	Parsing	7
6.2	Placing glyphs	7
7	Engine-specific Implementations	7
8	Introduction	9
9	Introduction	11
10	Functions	11
10.1	Command-generating Functions	11
10.2	Showcase (implementation) Functions	12
10.3	Helper Functions	13
10.3.1	Name-generating Functions	13
10.3.2	Input Validation Functions	14
10.3.3	General Helper Functions	14

*/cawaltrip

9 Introduction

In the initial version of `glyph`, Lua is doing a lot of heavy lifting. Most all of the functionality has been abstracted out to it with `LuaLaTeX` being reserved for passing data structures around, and parsing user input. All of the CS generation is handled in Lua.

10 Functions

The Lua functions can be broadly categorized into three groups,

- command-generating functions,
- helper functions, and
- showcase implementation functions. In the source code, but omitted here is the namespace for every function. The only namespace used is called `u` (i.e., *userspace*).

10.1 Command-generating Functions

<code>create_glyph_commands()</code>	<code>create_glyph_commands({fontId}, {prefix}, {gType}, [<i>debug</i>])</code>
--------------------------------------	---

This function generates all of the individual glyph commands for a font. Given,

#1 : `{fontId}`, the `TeX` font number,

#2 : `{prefix}`, the prefix specified by the user,

#3 : `{gType}`, the string of either “single” or “dual” to indicate whether the glyph name needs to be massaged before being displayed (for *dual glyph* only), and

#4 : [*debug*], a boolean to print debug output to console,

the function iterates through each glyph in the font and, with few exceptions, creates a command for it based on the glyph name defined in the font. The exceptions to this are characters in the C0/C1 Unicode blocks, and glyph names that have numbers in them. For the latter, the glyph name is sanitized in a standard way. Under the hood we’re storing the unicode codepoint, so we don’t care about retaining the unmodified glyph name.

<code>create_icon_command()</code>	<code>create_icon_command({fontId}, {prefix}, {gType}, [<i>debug</i>])</code>
------------------------------------	---

This function creates the document-level generic CS that users can use to return a glyph via its internal name, instead of the sanitized version created by `create_glyph_commands`. Given,

#1 : `{fontId}`, the `TeX` font number,

#2 : `{prefix}`, the prefix specified by the user,

#3 : `{gType}`, the string of either “single” or “dual” to indicate whether the glyph name needs to be massaged before being displayed (for *dual glyph* only), and

#4 : [*debug*], a boolean to print debug output to console,

creates a command of the form `\[no-index]{prefix}Glyph`. This command stores the `{fontId}`, `{font family}`, and `{gType}`, so that a user has a simple interface to placing glyphs.

create_font_family() `create_font_family({fontId}, {prefix}, [{debug}])`

This function creates the internally-used font family name. Given,

#1 : `{fontId}`, the T_EX font number,

#2 : `{prefix}`, the prefix specified by the user,

#3 : `[{debug}]`, a boolean to print debug output to console,

the function gets the font's *PostScript* name from the `{fontId}`, and uses it as the definition of the font family. Behind the scenes, `fontspec` is being used to create the font family. The name of the font family is programmatically generated based on the `{prefix}` specified by the user.

create_showcase_command() `create_showcase_command({fontId}, {prefix}, {debugCS}, [{debug}])`

This function creates the debug command for a single font. Given,

#1 : `{fontId}`, the T_EX font number

#2 : `{prefix}`, the font prefix specified by the user

#3 : `{debugCS}`, the control sequence name that calls the function to print all glyph info

#4 : `[{debug}]` boolean to determine whether to show debug messages

create a T_EX call to create a command in the form described in the section for `generate_showcase_name`. The `{fontId}`, and `{debugCS}` values are embedded into the command so that you don't need to know anything about them.

10.2 Showcase (implementation) Functions

The showcase functions are what implement printing the glyph tables to the document.

showcase_font() `showcase_font({fontId}, {fontFamilyCommand}, [{debug}])`

#1 : `{fontId}`, the T_EX font number

#2 : `{fontFamilyCommand}`, the font family CS name for displaying the font given by `{fontId}`

#3 : `[{debug}]` boolean to determine whether to show debug messages

This is the managing function that calls all of the other functions required to print all of the glyph information about a font. Ultimately, it creates the following:

1. a new `\section` heading with the font name as the title,
2. a sample of the font (currently this is lowercase a-z),
3. for each Unicode block, if there is a glyph defined in its range, create a new `\subsection`, and display the glyphs in `\normal` size.
4. create a `\subsection` and for every printable glyph in the font, create a small box that contains,
 - the glyph name,
 - the glyph itself at `\huge` font size, and
 - the Unicode codepoint in hex.

showcase_sample() showcase_sample(*{<glyphs>}*, *{<start>}*, *{<finish>}*, *{<fontFamilyCommand>}*, [*<debug>*])

#1 : *{<glyphs>}*, the glyph table for the font
 #2 : *{<start>}*, the index to start printing the sample at
 #3 : *{<finish>}*, the index to finish printing the sample at
 #4 : *{<fontFamilyCommand>}*, the font family CS name for displaying the glyphs
 #5 : [*<debug>*] boolean to determine whether to show debug messages
 This function prints the any glyph found in *{<glyphs>}* that in the range of *{<start>}* to *{<finish>}* (inclusive).

showcase_block() showcase_block(*{<glyphs>}*, *{<block>}*, *{<fontFamilyCommand>}*, [*<debug>*])

#1 : *{<glyphs>}*, the glyph table for the font
 #2 : *{<block>}*, the Unicode block
 #3 : *{<fontFamilyCommand>}*, the font family CS name for displaying the glyphs
 #4 : [*<debug>*] boolean to determine whether to show debug messages
 This function prints glyphs found in the range specified by the Unicode block as a `\subsection` with the blocks name as the title. It specifically only prints out this information if a glyph is found anywhere in the block. This is because there's more than 300 blocks defined by Unicode currently which would create a lot of wasted space as most fonts are going to have glyphs in each block defined.

showcase_glyph() showcase_glyph(*{<glyph>}*, *{<fontFamilyCommand>}*, [*<debug>*])

#1 : *{<glyph>}*, the glyph to print
 #2 : *{<fontFamilyCommand>}*, the font family CS name for displaying the glyphs
 #3 : [*<debug>*] boolean to determine whether to show debug messages
 This functions prints a specific *{<glyph>}* inside a box with the glyph's name and Unicode codepoint in hex.

10.3 Helper Functions

Helper functions themselves can be further categorized into the following groups:

- name-generating functions,
- input-validation functions, and
- general helper functions.

10.3.1 Name-generating Functions

generate_glyph_command_name() senerate_glyph_command_name(*{<glyphName>}*, *{<prefix>}*, *{<gType>}*)

Returns the sanitized version of the document-level command name to be created. Given,
 #1 : *{<glyphName>}*, the glyph name to use
 #2 : *{<prefix>}*, the font prefix specified by the user, and
 #3 : *{<gType>}*, the string of either “single” or “dual” to indicate whether the glyph name needs to be massaged before being displayed (for *dual glyph* only),
 return a string similar to `prefixSanitizedGlyphName`. Will not return a name that has characters that can't normally be used in a command sequence after replacing numbers with their spelt out versions, and hyphens.

`generate_icon_command_name()` `generate_icon_command_name({<prefix>})`

Given,
#1 : `{<prefix>}`, the font prefix specified by the user,
return the string `GlyphFF{<prefix>}`.

`generate_font_family_name()` `generate_font_family_name({<glyphName>}, {<prefix>}, {<gType>})`

Returns the sanitized version of the document-level command name to be created. Given,
#1 : `{<prefix>}`, the font prefix specified by the user, and
return a string in the format `GlyphFF{<prefix>}`.

`generate_showcase_name()` `generate_showcase_name({<prefix>})`

This defines the name of the command you can use to print glyph information for a font.
Given,
#1 : `{<prefix>}`, the font prefix specified by the user
returns a string of the form `\GlyphFD{<prefix>}`.

10.3.2 Input Validation Functions

`sanitize_glyph()` `sanitize_glyph({<glyphName>})`

Given,
#1 : `{<glyphName>}`, a (partially edited) glyph name,
return a version that has replaced all numbers with their spelt out versions¹, with hyphens removed and the letter after a hyphen capitalized.

`in_printable_range()` `in_printable_range({<num>})`

Given,
#1 : `{<num>}`, a Unicode codepoint,
checks if it's in the *C0/C1* Unicode blocks and returns false if so.

`has_only_valid_characters()` `has_only_valid_characters({<commandString>})`

Given,
#1 : `{<commandString>}`, the sanitized glyph name,
return false if the string contains anything other than the ASCII alphabetical letters (`[A-Za-z]`).

10.3.3 General Helper Functions

Things like the format of command names are stored as helper functions so they can be called wherever needed without passing extra information around.

<hr/> <hr/>	<code>debug({\doDebug}, {\string}, [<i>indentLevel</i>])</code> #1 : <code>{\doDebug}</code> , a boolean to determine if debug message is printed #2 : <code>{\message}</code> , string to print #3 : <code>[<i>indentLevel</i>]</code> , optional integer to indent the text by (each integer adds four space). If <code>{\doDebug}</code> is <code>false</code> , then nothing is done and this function is equivalent to a no-op. If true, then the <code>{\message}</code> string is printed to the screen, indented by four spaces for each <code>[<i>indentLevel</i>]</code> .
<hr/> <hr/>	<code>get_glyph_base_name({\glyphName}, [<i>noTex</i>])</code> Useful mainly for <i>dual glyphs</i> , this is used to strip away unique parts of the glyph name. By default returns the name to the T _E X stream, but can be returned instead to a Lua function. Given, #1 : <code>{\glyphName}</code> , the glyph to strip the appendix from, and #2 : <code>[<i>noTex</i>]</code> , boolean to determine how to return the basename, strip <code>-primary</code> and <code>-secondary</code> from the glyph name and return it. Glyph names without either appendix are returned unchanged. If <code>[<i>noTex</i>]</code> is passed, then return the basename, otherwise, print it to the T _E X stream.
<hr/> <hr/>	<code>get_glyph_index({\fontId}, {\glyphName}, [<i>appendix</i>])</code> Given, #1 : <code>{\fontId}</code> , the T _E X font number, #2 : <code>{\glyphName}</code> , the glyph name as defined in the font, and #3 : <code>[<i>appendix</i>]</code> , a string to append to <code>{\glyphName}</code> , this finds the Unicode codepoint (if it exists) and returns it. The <code>[<i>appendix</i>]</code> argument is used for <i>dual glyph</i> fonts to add the part removed in <code>generate_glyph_command_name</code> in order to search for the glyph properly. Iterates through the glyphs in the font defined with <code>{\fontId}</code> and returns the index if found or <code>nil</code> otherwise.
<hr/> <hr/>	<code>collect_glyphs({\fontId})</code> Given, #1 : <code>{\fontId}</code> , the T _E X font number, iterates through all Unicode codepoints and for all printable glyphs adds them to table and returns the table.
<hr/> <hr/>	<code>get_unicode_blocks</code> Provides the a list of Unicode blocks with their name, starting index, and ending index.