PyDvi Documentation

Release 0.1.0

Fabrice Salvaire

Mar 04, 2017

Contents

Welcome to the PyDvi documentation!

Note: The official Home Page of PyDvi is located at http://fabricesalvaire.github.io/PyDVI

If you are at http://readthedocs.org then you are reading the so called *latest documentation*.

The *latest documentation* is automatically built from the git repository after each commit.

As opposite the PyDvi Home Page is built manually and is thus less prone to errors.

PyDvi is a Python library to read and process DVI (DeVice Independent) files, the native output of the famous TeX typesetting program implemented by Donald E. Knuth.

PyDvi is also able to read most of the file formats associated to the TeX world like packed font, virtual font, TeX font metric, font map and font encoding. It can also read Adobe Font Metrics files.

Basically a DVI file describes the layout of a page by a list of opcodes that interact with a register machine to update the position on the page, to load fonts and to paint glyphs and rules. In short it contains the glyphs and their positions on the page. Since TeX was designed to layout the series of books *The Art of Computer Programming* at the beginning of the eighties, it focuses on texts and mathematical expressions. Thus DVI is much simpler than Postscript or its successor PDF which are designed for graphics. However we can extend the capabilities of DVI using the *special* opcode which can contain any text like Postscript snippets.

A DVI stream can come from a file or a TeX daemon in order to render TeX inputs on-the-fly.

The DVI parser of PyDvi builds a program from a DVI stream that could be later processed by the provided DVI machine which is designed to be subclassed by the user.

The source code includes an experimental DVI viewer as exemple which uses the OpenGL API for the rendering and thus feature an hardware accelerated rendering. PyDvi and the viewer can be used as a platform to experiment complex text rendering on GPU.

The source code includes also a clone of the dvipng tool to render DVI to PNG image. This tool is mainly intended to check the compliance of PyDvi by image comparison.

PyDvi can be used for several purpose, we will review them in the followings:

TeX is a major and historical typesetting program. PyDvi can serve to read and process its output using Python. The user can be a curious pearson who want to lean TeX or somebody interested by TeX postprocessing.

TeX is one of the oldest computer program still in activity. The reason is quite simple, these algorithms do the right job, its ecosystem is rich and its code is not so simple. Thus nobody succeeds to reimplement it up to now, excepted its mathematical layout algorithms by the MathJax Javascript library which is intended to bring Mathematical layout to web browser. Before the delivery of Mathjax, the only solution to render properly mathematical content was to generate an image using a program like dvipng. It is what does the engine of Wikipedia behind the scene. Usually these programs like Asymptote or Circuit_macros generate the graphics as a PDF document and then include this document in a LaTeX document which contains the labels placed at absolute positions in the page. With PyDvi we can try another approach which consists to send TeX content to a daemon and get back the glyphs and their positions. Matplotlib uses this approach to render LaTeX labels.

CHAPTER 1

Installation

The procedure to install PyDvi is described in the *Installation Manual*.

CHAPTER 2

Testing

Some examples are provided with PyDvi, see the *example section*.

CHAPTER 3

Documentation

- *PyDvi Reference Manual*
- *Bibliography*

CHAPTER 4

Overview

Installation

The installation of PyDvi by itself is quite simple. However it will be easier to get the dependencies on a Linux desktop.

Dependencies

PyDvi requires the following dependencies:

- Python
- Numpy
- freetype-py for Type1 font rendering

The OpenGL DVI viewer requires these additional dependencies:

- pyqt
- PyOpenGLng

The DVI to PNG tool requires these additional dependencies:

- pillow

Also it is recommanded to have these Python modules:

- pip
- virtualenv

For development, you will need in addition:

- Sphinx

Installation from PyPi Repository

PyDvi is made available on the PyPI repository at https://pypi.python.org/pypi/PyDVI

Run this command to install the last release:

```
pip install PyDvi
```

Installation from Source

The PyDvi source code is hosted at https://github.com/FabriceSalvaire/PyDVI

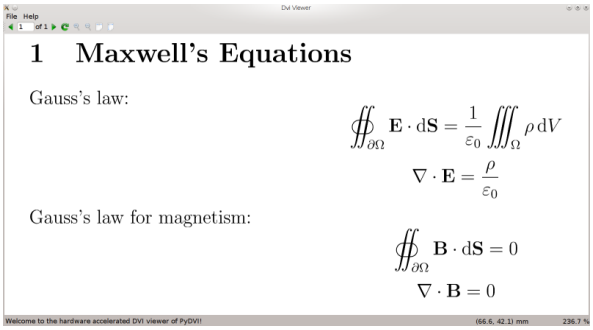
To clone the Git repository, run this command in a terminal:

```
git clone git@github.com:FabriceSalvaire/PyDvi.git
```

Then to build and install PyDvi run these commands:

```
python setup.py build
python setup.py install
```

Examples



The program *gui/dvi-gl-viewer* is a DVI viewer demonstrator with a rendering accelerated by GPU using the OpenGL API.

The program *gui/font-viewer* is a tool to display font glyph.

To run the unit tests use this shell command:

```
for i in unit_test/*.py; do python $i; done
```

To run the test programs do:

```
python test/test-dvi-machine tex-samples/text.cmr.latin1.dvi
python test/test_encoding.py ec.enc
python test/test_font_manager.py
python test/test_font_map.py pdftex.map
python test/test_pkfont.py cmr10
python test/test_tex_daemon.py
python test/test_tfm.py cmr10
```

API Documentation

This is the auto-generated API documentation for the PyDvi library.

Note: The docstings and the code need a review. Most of the code was written a long time ago (for Python 2.4).

Warning: The API documentation is automatically generated from the docstrings in the source using the Sphinx tool. This way to produce the documentation is know to be perfectible actually, but not too bad.

Contents:

PyDvi

Config

ConfigInstall

class PyDvi.Config.ConfigInstall.**Logging**
Bases: object

default_config_file = 'logging.yml'

directories = ('/home/docs/checkouts/readthedocs.org/user_builds/pydvi/envs/latest/local/lib/python2.7/site-packa

static find(*config_file*)

class PyDvi.Config.ConfigInstall.**Path**
Bases: object

config_directory = '/home/docs/checkouts/readthedocs.org/user_builds/pydvi/envs/latest/local/lib/python2.7/site-

pydvi_module_directory = '/home/docs/checkouts/readthedocs.org/user_builds/pydvi/envs/latest/local/lib/pytho

Dvi

DviMachine

class PyDvi.Dvi.DviMachine.**Opcode_set_char**(*char_code*)
Bases: PyDvi.Dvi.DviMachine.Opcode_putset_char

This class implements the set_char opcode.

class PyDvi.Dvi.DviMachine.**Opcode_put_char**(*char_code*)
Bases: PyDvi.Dvi.DviMachine.Opcode_putset_char

This class implements the put_char opcode.

class PyDvi.Dvi.DviMachine.**Opcode_set_rule**(*height*, *width*)
Bases: PyDvi.Dvi.DviMachine.Opcode_putset_rule

This class implements the set_rule opcode.

class PyDvi.Dvi.DviMachine.**Opcode_put_rule**(*height*, *width*)
Bases: PyDvi.Dvi.DviMachine.Opcode_putset_rule

This class implements the put_rule opcode.

class PyDvi.Dvi.DviMachine.**Opcode_push**
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the push opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_pop**(*n=1*)
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the pop opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_push_colour**(*colour*)
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the push_colour opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_pop_colour**(*n=1*)
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the pop_colour opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_right**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the right opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_w0**
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the w0 opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_w**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the w opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_x0**
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the x0 opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_x**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the x opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_down**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the down opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_y0**
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the y0 opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_y**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the y opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_z0**
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the z0 opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_z**(*x*)
Bases: PyDvi.Dvi.DviMachine.OpcodeX

This class implements the z opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_font**(*font_id*)
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the font opcode.

run(*dvi_machine*, *compute_bounding_box=False*)

class PyDvi.Dvi.DviMachine.**Opcode_xxx**(*code*)
Bases: PyDvi.Dvi.DviMachine.Opcode

This class implements the xxx opcode.

class PyDvi.Dvi.DviMachine.**DviFont**(*font_id*, *name*, *checksum*, *scale_factor*, *design_size*)
Bases: object

This class implements a DVI Font.

char_scaled_depth(*tfm_char*)
Return the scale depth for the PyDvi.TfmChar instance.

char_scaled_height(*tfm_char*)
Return the scale height for the PyDvi.TfmChar instance.

char_scaled_width(*tfm_char*)
Return the scale width for the PyDvi.TfmChar instance.

class PyDvi.Dvi.DviMachine.**DviColourBlack**
Bases: PyDvi.Dvi.DviMachine.DviColour

This class implements the black colour.

class PyDvi.Dvi.DviMachine.**DviColourGray**(*gray_level*)
Bases: PyDvi.Dvi.DviMachine.DviColour

This class implements gray colour.

class PyDvi.Dvi.DviMachine.**DviColourRGB**(*red*, *green*, *blue*, *alpha=1*)
Bases: PyDvi.Dvi.DviMachine.DviColour

This class implements RGB colour.

class PyDvi.Dvi.DviMachine.**DviColourCMYK**(*cyan*, *magenta*, *yellow*, *dark*)
Bases: PyDvi.Dvi.DviMachine.DviColour

This class implements CMYK colour.

class PyDvi.Dvi.DviMachine.**DviProgam**
Bases: object

This class implements a DVI program.

append_page(*i*)

dvi_font_iterator()

get_font(*i*)

print_summary()

register_font(*font*)
Register a *DviFont* instance.

set_postambule_data(*max_height*, *max_width*, *stack_depth*, *number_of_pages*)
Set the postamble data.

set_preambule_data(*comment*, *dvi_format*, *numerator*, *denominator*, *magnification*)
Set the preambule data.

class PyDvi.Dvi.DviMachine.**DviProgramPage**(*page_number*, *height=0*, *width=0*, *paper_orientation=0*)
Bases: list

This class defines a page.

print_program()
Print the program.

set_paper_size(*height*, *width*)
Set the paper size in mm.

class PyDvi.Dvi.DviMachine.**DviSubroutine**
Bases: list

class PyDvi.Dvi.DviMachine.**DviMachine**(*font_manager*)
Bases: object

This class implements a DVI Machine.

_adjust_opcode_counts(*opcode_program*, *virtual_font*, *characters*)

_adjust_opcode_counts_for_virtual_characters(*opcode_program*)

_load_dvi_fonts()
Load the fonts used by the DVI program.

_logger = <logging.Logger object>

_reset()
Reset the machine.

begin_run_page()

compute_page_bounding_box(*page_index*)

count_opcodes(*opcode_program*)

current_colour
Return the current colour.

current_dvi_font
Return the current dvi font.

current_font
Return the current font.

current_font_id

end_run_page()

is_current_font_virtual

load_dvi_program(*dvi_program*, *load_fonts=True*)
Load a *DviProgam* instance.

paint_char(*x*, *y*, *char_bounding_box*, *font*, *dvi_font*, *char_code*)

paint_rule(*x*, *y*, *width*, *height*)

pop_colour(*n=1*)
Pop *n* level in the colour stack.

pop_registers(*n=1*)
Pop *n* level in the register set stack.

push_colour(*colour*)
Push the current colour.

push_registers(*reset=False*)
Push the register set.

registers
Return the current register set.

run_page(*page_index*, ***kwargs*)

run_subroutine(*subroutine*)

simplify_dvi_program()
Simplify the DVI program.

class PyDvi.Dvi.DviMachine.**DviSimplifyMachine**(*font_manager*)
Bases: *PyDvi.Dvi.DviMachine.DviMachine*

process_page_xxx_opcodes(*program_page*)
Process the xxx opcodes in the page program.

simplify(*simplify_opcodes=False*)
Simplify the program.

simplify_page(*program_page*)
Simplify the page.

transform_xxx_colour(*program_page*, *xxx_code*)
Transform a xxx colour opcode.

transform_xxx_paper_orientation(*program_page*, *xxx_code*)
Transform a xxx paper orientation opcode.

transform_xxx_paper_size(*program_page*, *xxx_code*)
Transform a xxx paper size opcode.

xxx_colour = 'color '
Defines colour special

xxx_landscape = '! /landplus90 true store'
Defines landscape special

xxx_papersize = 'papersize='
Defines papersize special

DviParser

This module implements a DVI Stream Parser.

class PyDvi.Dvi.DviParser.**DviParser**
Bases: object

This class implements a DVI Stream Parser.

_logger = <logging.Logger object>

_process_pages_backward()
Process the pages in backward order.

_process_postambule()
Process the postamble where we get the number of pages and the fonts.

_process_preambule()
Process the preamble where we get the magnification.

_reset()
Reset the DVI parser.

process_page()

process_page_forward()

process_stream(*stream*)
Process a DVI stream and return a DviProgam instance.

class PyDvi.Dvi.DviParser.**DviSubroutineParser**(*stream*)
Bases: object

parse()

class PyDvi.Dvi.DviParser.**OpcodeParser_fnt_def**(*opcode*)
Bases: *PyDvi.OpcodeParser.OpcodeParser*

This class parse the fnt_def opcode.

base_opcode = 243

read_parameters(*dvi_parser*)

class PyDvi.Dvi.DviParser.**OpcodeParser_font**(*opcode*)
Bases: *PyDvi.OpcodeParser.OpcodeParser*

This class parse the font opcode.

read_parameters(*dvi_parser*)

class PyDvi.Dvi.DviParser.**OpcodeParser_set_char**(*opcode*)
Bases: *PyDvi.OpcodeParser.OpcodeParser*

This class parse the set_char opcode.

read_parameters(*dvi_parser*)

class PyDvi.Dvi.DviParser.**OpcodeParser_xxx**(*opcode*)
Bases: *PyDvi.OpcodeParser.OpcodeParser*

This class parse the xxx opcode.

base_opcode = 239

read_parameters(*dvi_parser*)

Font

AfmParser

Reference: Adobe Font Metrics File Format Specification, Version 4.1, 7 October 1998

All measurements in AFM files are given in terms of units equal to 1/1000 of the scale factor (point size) of the font being used. To compute actual sizes in a document (in points; with 72 points = 1 inch), these amounts should be multiplied by (scale factor of font) / 1000.

class PyDvi.Font.AfmParser.**AfmParser**(*filename*)
Bases: object

_get_values(*types*, *line*)

_logger = <logging.Logger object>

_parse(*stream*)

_parse_end(*line*)

_parse_key_values(*keys*, *line*)

_parse_key_values_list(*keys*, *line*)

_parse_start(*line*)

static parse(*filename*)

exception PyDvi.Font.AfmParser.**BadAfmFile**
Bases: exceptions.NameError

PyDvi.Font.AfmParser.**boolean**(*x*)

PyDvi.Font.AfmParser.**hex**(*x*)

Encoding

This module handles TeX encoding file.

An encoding file map the glyph index with its symbolic name. It uses the `.enc` extension.

For example, the content of `cork.enc` is:

```
/CorkEncoding [ % now 256 chars follow
% 0x00
  /grave /acute /circumflex /tilde /dieresis /hungarumlaut /ring /caron
  /breve /macron /dotaccent /cedilla
  /ogonek /quotesinglbase /guilsinglleft /guilsinglright
...
% 0xF0
  /eth /ntilde /ograve /oacute /ocircumflex /otilde /odieresis /oe
  /oslash /ugrave /uacute /ucircumflex /udieresis /yacute /thorn /germandbls
] def
```

The percent character is used for comment as for TeX.

The content of this file can be parsed using:

```
cork_encoding = Encoding('/usr/share/texmf/fonts/enc/dvips/base/cork.enc')
```

The encoding's name can be retrieved using:

```
>>> cork_encoding.name
'CorkEncoding'
```

The number of glyphes can be obtained using the function `len()`:

```
>>> len(cork_encoding)
256
```

The index of the glyph `eth` can be retrieved using:

```
>>> cork_encoding['eth']
0xF0
```

and reciprocally:

```
>>> cork_encoding[0xF0]
'eth'
```

The methods `to_index()` and `to_name()` are used internally for this purpose.

class PyDvi.Font.Encoding.**Encoding**(*filename*)

Bases: `object`

Parse an encoding file and store the association between the index and the glyph's name.

Public attributes are:

`name`

_parse_glyph_names(*line*)

Find glyph names in *line*, as '/grave /acute /circumflex ...'.

_parse_name(*line*)

Find the encoding name at the left of the line, as '/CorkEncoding'.

print_summary()

to_index(*name*)
Return the glyph index corresponding to the symbolic name.

to_name(*i*)
Return the symbolic name corresponding to the glyph index *i*.

Font

This module provides a base class for font type managed by the font manager.

class PyDvi.Font.Font.**Font**(*font_manager*, *font_id*, *name*)
Bases: object

This class is a base class for font managed by the Font Manager.

Class attributes to be defined in subclass:

font_type font type enumerate

font_type_string description of the font type

extension file extension

To create a *Font* instance use:

```
font = Font(font_manager, font_id, name)
```

where *font_manager* is a PyDvi.FontManager.FontManager instance, *font_id* is the font id provided by the font manager and *name* is the font name, "cmr10" for example.

_find_font(*kpsewhich_options=None*)
Find the font file location in the system using Kpathsea.

_find_tfm()
Find the TFM file location in the system using Kpathsea and load it.

basename()
Return the basename.

extension = **None**

font_type = **None**

font_type_string = **None**

is_virtual

print_header()

print_summary()

PyDvi.Font.Font.**font_types**
Font Type Enumerate

alias of FontTypes

PyDvi.Font.Font.**sort_font_class**(**args*)
Sort a list of *Font* instance by font type enumerate.

FontManager

FontMap

This module handles font map files.

A font map file gives the correspondance between the TeX PK fonts and their PostScript equivalents. These files use the `.map` extension.

For example, the file `pdftex.map` contains lines like this one:

```
futbo8r Utopia-Bold ".167 SlantFont TeXBase1Encoding ReEncodeFont" <8r.enc <putb8a.
↪pfb
```

Each line describes a PK font using the following format:

```
PK_FONT_NAME PS_FONT_NAME "PostScript snippet" <FILE_NAME1 <FILE_NAME2
```

The first word is the TeX font name and the second word is the PostScript font name. The PostScript font name can be omitted if it is the same than for TeX. The word starting by "<" are filenames to be included in the PostScript file. A filename with the extension `.enc` is an encoding file and a filename with the extension `.pfb` is a Printer Font Binary file. The text enclosed by double quotes is optional and gives a PostScript snippet to be inserted in the PostScript file. It can be placed at the end of the line.

The percent character is used for comment as for TeX.

References:

- The Font Installation Guide Using Postscript fonts to their full potential with Latex. Originally written by Philipp Lehman. December 2004. Revision 2.14. cf. Creating map files Part. http://www.ctan.org/tex-archive/info/Type1fonts/fontinstallationguide
- updmap(1) - Update font map files for TeX output drivers.
- updmap.cfg(5) - Configuration of font mapping/inclusion for dvips and friends

To parse the font map `pdftex.map` do:

```
font_map = FontMap('/usr/share/texmf/fonts/map/pdftex/updmap/pdftex.map')
```

Each entry is stored in a *FontMapEntry* instance and can be retrieved using its TeX name as key:

```
font_map_entry = font_map['futbo8r']
```

The `.pfb` file name and the other parameters are stored as attributes:

```
>>> font_map_entry.pfb_filename
'putb8a.pfb'
```

class PyDvi.Font.FontMap.**FontMap**(*filename*)

Bases: object

This class parses a font map file.

static _parse_effects(*ps_snippet*)

Parse the PostScript snippet.

_parse_line(*line*)

Parse a line.

_register_entry(*font_map_entry*)

Register a font map entry.

print_summary()

class PyDvi.Font.FontMap.**FontMapEntry**(*tex_name*, *ps_font_name*, *ps_snippet*, *effects*, *encoding*, *pfb_filename*)

Bases: object

This class encapsulates a font map entry.

Public attributes:

tex_name

ps_font_name

ps_snippet

effects

encoding

pfb_filename

print_summary()

PkFont

PkFontParser

PkGlyph

Tfm

This module handles TeX Font Metric.

The class PyDvi.Tfm handles the font's metric. To get a PyDvi.Tfm instance for a particular font use the static method PyDvi.TfmParser.TfmParser.parse(). For example use this code for the font "cmr10":

```
tfm = TfmParser.parse('cmr10', '/usr/share/texmf/fonts/tfm/public/cm/cmr10.tfm')
```

The number of characters in the font can be obtained using the function len():

```
>>> len(tfm)
128
```

Each character's metric is stored in a *TfmChar* instance that can be accessed using the char code as index on the *Tfm* class instance. For example to get the metric of the character "A" use:

```
tfm[ord('A')]
```

class PyDvi.Font.Tfm.**Tfm**(*font_name*, *filename*, *smallest_character_code*, *largest_character_code*, *checksum*, *design_font_size*, *character_coding_scheme*, *family*)

Bases: object

This class encapsulates a TeX Font Metric for a font.

Public attributes:

font_name font's name

filename ".tfm" filename

smallest_character_code smallest character code of the font

largest_character_code largest character code of the font

checksum checksum of the tfm file

design_font_size design font size

character_coding_scheme character coding scheme

family font's family

slant

spacing

space_stretch

space_shrink

x_height

quad

extra_space

In addition for Math font, the following public attributes are available:

um1

num2

num3

denom1

denom2

sup1

sup2

sup3

sub1

sub2

supdrop

subdrop

delim1

delim2

axis_height

default_rule_thickness

big_op_spacing

The number of characters can be queried using `len()`. The *TfmChar* instance for a character code *char_code* can be set or get using the operator [].

add_lig_kern(*obj*)
Add a ligature/kern program *obj*.

get_lig_kern_program(*i*)
Return the ligature/kern program at index *i*.

print_summary()

set_font_parameters(*parameters*)
Set the font parameters.

set_math_extension_parameters(*parameters*)
Set the math extension parameters.

set_math_symbols_parameters(*parameters*)
Set the math symbols parameters.

class PyDvi.Font.Tfm.**TfmChar**(*tfm*, *char_code*, *width*, *height*, *depth*, *italic_correction*, *lig_kern_program_index=None*, *next_larger_char=None*)

Bases: object

This class encapsulates a TeX Font Metric for a Glyph.

Public attributes:

char_code

width

height

depth

italic_correction

chr()
Return the character string from its char code if it is printable.

get_lig_kern_program()
Get the ligature/kern program of the character.

next_larger_tfm_char()
Return the *TfmChar* instance for the next larger char if it exists else return None.

print_summary()

printable = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!"#$%&\'()*+,-./
List of the printable characters.

scaled_depth(*scale_factor*)
Return the scaled depth by *scale_factor*.

scaled_dimensions(*scale_factor*)
Return the 3-tuple made of the scaled width, height and depth by *scale_factor*.

scaled_height(*scale_factor*)
Return the scaled height by *scale_factor*.

scaled_width(*scale_factor*)
Return the scaled width by *scale_factor*.

class PyDvi.Font.Tfm.**TfmExtensibleChar**(*tfm*, *char_code*, *width*, *height*, *depth*, *italic_correction*, *extensible_recipe*, *lig_kern_program_index=None*, *next_larger_char=None*)
Bases: *PyDvi.Font.Tfm.TfmChar*

This class encapsulates a TeX Font Metric for an extensible Glyph.

Public attributes:

top

mid

bot

rep

class PyDvi.Font.Tfm.**TfmKern**(*tfm*, *index*, *stop*, *next_char*, *kern*)
Bases: PyDvi.Font.Tfm.TfmLigKern

This class represents a Kerning Program Instruction.

Public Attributes:

next_char next character

kern kerning value

class PyDvi.Font.Tfm.**TfmLigature**(*tfm*, *index*, *stop*, *next_char*, *ligature_char_code*, *number_of_chars_to_pass_over*, *current_char_is_deleted*, *next_char_is_deleted*)
Bases: PyDvi.Font.Tfm.TfmLigKern

This class represents a Ligature Program Instruction.

Public Attributes:

next_char next character

ligature_char_code ligature character code

current_char_is_deleted the current characters must be deleted of the stream

next_char_is_deleted the next characters must be deleted of the stream

number_of_chars_to_pass_over number of characters to pass over

TfmParser

The `TfmParser` module provides a tool to parse TeX Font Metric file. TFM files contain the metrics for TeX fonts. They have the ".tfm" extension.

To parse a TFM file and get a `PyDvi.Tfm` instance, use the static method *`TfmParser.parse()`*. For example use this code for the font "cmr10":

```
tfm = TfmParser.parse('cmr10', '/usr/share/texmf/fonts/tfm/public/cm/cmr10.tfm')
```

The TFM file format in descriped in the `tftopl.web` file from Web2C. Part of this documentation comes from this file.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words. Note that the bytes are considered to be unsigned numbers.

class `PyDvi.Font.TfmParser.`**TfmParser**(*font_name*, *filename*)
Bases: `object`

This class parse a TFM file.

_position_in_table(*table*, *index*)
Return a pointer to the word element at *index* in the table *table*.

_print_summary()

_process_char(*c*)
Process the character code *c* in the character information table.

_read_char_info(*c*)
Read the character code *c* data in the character information table.

_read_characters()
Next comes the char info array, which contains one char info word per character. Each char info word contains six fields packed into four bytes as follows.

- first byte: `width_index` (8 bits)
- second byte: `height_index` (4 bits) times 16, plus depth index (4 bits)
- third byte: `italic_index` (6 bits) times 4, plus tag (2 bits)
- fourth byte: `remainder` (8 bits)

The actual width of a character is `width[width_index]`, in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the TFM format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation `width[0] = height[0] = depth[0] = italic[0] = 0` should always hold, so that an index of zero implies a value of zero. The width index should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between `bc` and `ec` and has a nonzero width index.

The tag field in a char info word has four values that explain how to interpret the remainder field.

- `tag = 0` (`no_tag`) means that remainder is unused.

- `tag = 1` (`lig_tag`) means that this character has a ligature/kerning program starting at `lig_kern[remainder]`.
- `tag = 2` (`list_tag`) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The remainder field gives the character code of the next larger character.
- `tag = 3` (`ext_tag`) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in `exten[remainder]`.
- `no_tag = 0` vanilla character
- `lig_tag = 1` character has a ligature/kerning program
- `list_tag = 2` character has a successor in a charlist
- `ext_tag = 3` character is extensible

_read_extensible_recipe(*index*)
Return the extensible recipe, four numbers, at index *index*.

Extensible characters are specified by an extensible recipe, which consists of four bytes called top, mid, bot, and rep (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If top, mid, or bot are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

_read_fix_word_in_table(*table*, *index*)
Return the fix word in table *table* at index *index*.

_read_font_parameters()
The final portion of a TFM fie is the param array, which is another sequence of fix word values.

- param[1] = `slant` is the amount of italic slant, which is used to help position accents. For example, slant = .25 means that when you go up one unit, you also go .25 units to the right. The slant is a pure number; it's the only fix word other than the design size itself that is not scaled by the design size.
- param[2] = `space` is the normal spacing between words in text. Note that character " " in the font need not have anything to do with blank spaces.
- param[3] = `space_stretch` is the amount of glue stretching between words.
- param[4] = `space_shrink` is the amount of glue shrinking between words.
- param[5] = `x_height` is the height of letters for which accents don't have to be raised or lowered.
- param[6] = `quad` is the size of one em in the font.
- param[7] = `extra_space` is the amount added to param[2] at the ends of sentences.

When the character coding scheme is `TeX math symbols`, the font is supposed to have 15 additional parameters called `num1`, `num2`, `num3`, `denom1`, `denom2`, `sup1`, `sup2`, `sup3`, `sub1`, `sub2`, `supdrop`, `subdrop`, `delim1`, `delim2`, and `axis_height`, respectively. When the character coding scheme is `TeX math extension`, the font is supposed to have six additional parameters called `defaul_rule_thickness` and `big_op_spacing1` through `big_op_spacing5`.

_read_four_byte_numbers_in_table(*table*, *index*)
Return the four numbers in table *table* at index *index*.

_read_header()
The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, and for TFM files to be used with Xerox printing software it must contain at least 18 words, allocated as described below.

`header[0]` is a 32-bit check sum that TEX will copy into the DVI output file whenever it uses the font. Later on when the DVI file is printed, possibly on another computer, the actual font that gets used

is supposed to have a check sum that agrees with the one in the TFM file used by TEX. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the TFM file, no check is made.) The actual relation between this check sum and the rest of the TFM file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

`header[1]` is a fix word containing the design size of the font, in units of TEX points (7227 TEX points = 254 cm). This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TEX user asks for a font "at delta pt", the effect is to override the design size and replace it by delta, and to multiply the x and y coordinates of the points in the font image by a factor of delta divided by the design size. All other dimensions in the TFM file are fix word numbers in design-size units. Thus, for example, the value of `param[6]`, one em or `\quad`, is often the fix word value `2**20 = 1.0`, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, `header[1]` and `param[1]` are the only fix word entries in the whole TFM file whose first byte might be something besides 0 or 255.

`header[2 ... 11]`, if present, contains 40 bytes that identify the character coding scheme. The first byte, which must be between 0 and 39, is the number of subsequent ASCII bytes actually relevant in this string, which is intended to specify what character-code-to-symbol convention is present in the font. Examples are ASCII for standard ASCII, TeX text for fonts like cmr10 and cmti9, TeX math extension for cmex10, XEROX text for Xerox fonts, GRAPHIC for special-purpose non- alphabetic fonts, UNSPECIFIED for the default case when there is no information. Parentheses should not appear in this name. (Such a string is said to be in BCPL format.)

`header[12 ... 16]`, if present, contains 20 bytes that name the font family (e.g., CMR or HELVETICA), in BCPL format. This field is also known as the "font identifier."

`header[17]`, if present, contains a first byte called the `seven_bit_safe_flag`, then two bytes that are ignored, and a fourth byte called the *face*. If the value of the fourth byte is less than 18, it has the following interpretation as a "weight, slope, and expansion": Add 0 or 2 or 4 (for medium or bold or light) to 0 or 1 (for roman or italic) to 0 or 6 or 12 (for regular or condensed or extended). For example, 13 is `0+1+12`, so it represents medium italic extended. A three-letter code (e.g., MIE) can be used for such face data.

`header[18 ... whatever]` might also be present; the individual words are simply called `header[18]`, `header[19]`, etc., at the moment.

`_read_lengths`()

The fist 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

- lf = length of the entire file, in words;
- lh = length of the header data, in words;
- bc = smallest character code in the font;
- ec = largest character code in the font;
- nw = number of words in the width table;
- nh = number of words in the height table;
- nd = number of words in the depth table;
- ni = number of words in the italic correction table;
- nl = number of words in the lig/kern table;
- nk = number of words in the kern table;
- ne = number of words in the extensible character table;
- np = number of font parameter words.

They are all nonnegative and less than 2**15. We must have `bc - 1 <= ec <= 255`, `ne <= 256`, and

`lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np`.

Note that a font may contain as many as 256 characters (if `bc = 0` and `ec = 255`), and as few as 0 characters (if `bc = ec + 1`).

The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification:

header	array [0 ... lh - 1]	of stuff
char info	array [bc ... ec]	of char info word
width	array [0 ... nw - 1]	of fix word
height	array [0 ... nh - 1]	of fix word
depth	array [0 ... nd - 1]	of fix word
italic	array [0 ... ni - 1]	of fix word
lig kern	array [0 ... nl - 1]	of lig kern command
kern	array [0 ... nk - 1]	of fix word
exten	array [0 ... ne - 1]	of extensible recipe
param	array [1 ... np]	of fix word

_read_lig_kern_programs()

The lig kern array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word is a lig kern command of four bytes.

- first byte: `skip_byte`, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.
- second byte: `next_char`, "if `next_char` follows the current character, then perform the operation and stop, otherwise continue."
- third byte: `op_byte`, indicates a ligature step if less than 128, a kern step otherwise.
- fourth byte: `remainder`.

In a kern step, an additional space equal to `kern[256 * (op_byte + 128) + remainder]` is inserted between the current character and next char. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having `op_byte` codes `4a+2b+c` where `0 <= a <= b+c` and `0 <= b; c <= 1`. The character whose code is remainder is inserted between the current character and next char; then the current character is deleted if `b = 0`, and next char is deleted if `c = 0`; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

Notice that if `a = 0` and `b = 1`, the current character is unchanged; if `a = b` and `c = 1`, the current character is changed but the next character is unchanged.

If the very first instruction of the lig kern array has `skip_byte = 255`, the `next_char` byte is the so-called right boundary character of this font; the value of `next_char` need not lie between `bc` and `ec`. If the very last instruction of the lig kern array has `skip_byte = 255`, there is a special ligature/kerning program for a left boundary character, beginning at location `256 * op_byte + remainder`. The interpretation is that TEX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's `lig_kern` program has `skip_byte > 128`, the program actually begins in location `256 * op_byte + remainder`. This feature allows access to large lig kern arrays, because the first instruction must otherwise appear in a location `<= 255`.

Any instruction with `skip_byte > 128` in the lig kern array must have `256 * op_byte + remainder < nl`. If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

_seek_to_table(*table*)

Seek to the table *table*.

static parse (*font_name*, *filename*)
Parse the TFM `filename` for the font *font_name* and return a `PyDvi.Tfm` instance.

static word_ptr (*base*, *index*)
Compute the pointer to the word element index *index* from the base *base*. A word element has a size of 32-bit.

```
prt = base + 4*index
```

Type1Font

VirtualCharacter

class PyDvi.Font.VirtualCharacter.**VirtualCharacter** (*char_code*, *width*, *dvi*)
Bases: object

subroutine

VirtualFont

class PyDvi.Font.VirtualFont.**VirtualFont** (*font_manager*, *font_id*, *name*)
Bases: *PyDvi.Font.Font.Font*

This class implements the virtual font type in the font manager.

_set_preambule_data (*vf_id*, *comment*, *design_font_size*, *checksum*)
Set the preambule data from the Virtual Font Parser.

extension = 'vf'

font_type = 1

font_type_string = 'TeX Virtual Font'

load_dvi_fonts ()

print_summary ()

register_character (*character*)

register_font (*font*)
Register a `DviFont` instance.

update_font_id_map ()

VirtualFontParser

The VirtualFontParser module provides a tool to parse virtual font files. They have the ".vf" extension.

class PyDvi.Font.VirtualFontParser.**VirtualFontParser** (*virtual_font*)
Bases: object

_logger = <logging.Logger object>

_process_file ()
Process the characters.

_process_preambule ()
Process the preamble.

opcode_parser_set

static parse (*virtual_font*)

Kpathsea

This module provides a wrapper for the **Kpathsea** library, cf. http://www.tug.org/kpathsea.

PyDvi.Kpathsea.**kpsewhich**(*filename*, *file_format=None*, *options=None*)

Wrapper around the **kpsewhich** command, cf. kpsewhich(1).

file_format used to specify the file format, see **kpsewhich** help for the file format list.

options additional option for **kpsewhich**.

Examples:

```
>>> kpsewhich('cmr10', file_format='tfm')
'/usr/share/texmf/fonts/tfm/public/cm/cmr10.tfm'
```

Logging

Logging

PyDvi.Logging.Logging.**setup_logging**(*application_name='PyDvi'*, *config_file='logging.yml'*)

OpcodeParser

This modules provides tools to parse TeX stream like DVI file and PK Font.

class PyDvi.OpcodeParser.**OpcodeParserSet**(*opcode_definitions*)

Bases: list

This class defines an opcode parser set.

The parameter *opcode_definitions* is a tuple of 'opcode definition'.

An opcode definition is a tuple that corresponds to the parameters of the *OpcodeParser* constructor. Except that the opcode byte can be a list that defines a range of opcode bytes. In this case the opcode is duplicated in the opcode range. Moreover the parameter's definition can be a list that defines a range for a mono parameter set of opcodes. For example [1,4] will create successively an opcode with 1 to 4-byte unsigned parameter with an incremental opcode byte starting from the one specified.

Usage summary:

```
opcode_definitions : (opcode_definition, ...)

opcode_definition :
  (opcode_indexes, opcode_name, opcode_description, opcode_parameters=(),␣
↪opcode_class=None) |
  (opcode_indexes, opcode_parser_class),

opcode_indexes :
  index |
  [lower_index, upper_index] # duplicate the opcode in the range

opcode_parameters :
  (p0, p1, ...) |
  ([lower_n, upper_n]) # opcode at [index + i] has parameter p[i]
```

_init_opcode_parser(*opcode_definition*)

Build the set.

class PyDvi.OpcodeParser.**OpcodeParser**(*opcode*, *name*, *description*, *parameters=()*, *opcode_class=None*)

Bases: object

This class implements an opcode parser.

The argument *opcode* defines the opcode byte.

The argument *name* and *description* defines the name and a description string, respectively.

The argument *parameters* is a tuple that defines the parameters of the opcode. Each item is an integer that gives the number of bytes of the parameter. If this number is negative then the parameter is a signed integer. For example `(2, -3)` defines an opcode having as parameters a 2-byte unsigned integer followed by a 3-byte signed integer.

The optional *opcode_class* defines an `PyDvi.DviMachine.Opcode` subclass for the opcode.

_init_parameter_readers(*parameters*)

read_parameters(*opcode_parser*)
Read the opcode parameters.

to_opcode(*args*)
Return an an `PyDvi.DviMachine.Opcode` subclass isntance.

TeXUnit

This module provides functions to convert units used in the TeX world:

- **mm** stands for milimetre,
- **in** stands for inch which corresponds to 25.4 mm,
- **pt** stands for TeX point, there is 72.27 pt in one inch,
- **sp** stands for scale point, there is 2**16 sp in one pt,
- **dpi** stands for dot per inch.

The DVI format uses the measure 100 nm as base unit. A scaled point is defined as a fraction:

- num = 2.54 * 1e7 = 25400000
- den = 7227 * 2**16 = 473628672
- 1 sp = num/den = 5.4 nm

For a resolution of 1200 dpi, a pixel measures 21 um.

`PyDvi.TeXUnit.`**dpi2mm**(*x*)
Convert dpi to mm

`PyDvi.TeXUnit.`**in2mm**(*x*)
Convert in to mm

`PyDvi.TeXUnit.`**in2pt**(*x*)
Convert in to pt

`PyDvi.TeXUnit.`**in2sp**(*x*)
Convert in to sp

`PyDvi.TeXUnit.`**mm2in**(*x*)
Convert mm to in

`PyDvi.TeXUnit.`**pt2in**(*x*)
Convert in to pt

`PyDvi.TeXUnit.`**pt2mm**(*x*)
Convert pt to mm

`PyDvi.TeXUnit.`**pt2sp**(*x*)
Convert pt to sp

PyDvi.TeXUnit.**sp2in**(*x*)
Convert sp to in

PyDvi.TeXUnit.**sp2mm**(*x*)
Convert sp to mm

PyDvi.TeXUnit.**sp2pt**(*x*)
Convert sp to pt

PyDvi.TeXUnit.**sp2dpi**(*x*)
Convert sp to dpi

TexDaemon

This module provides a TeX Daemon interface.

class PyDvi.TexDaemon.**TexDaemon**(*working_directory*, *tex_format*, *start_code*, *done_code*, *done_string*)
Bases: *PyDvi.Tools.DaemonSubprocess.DaemonSubprocess*

This class implements a TeX Daemon Subprocess.

The TeX process run in the working directory *working_directory*.

The argument *tex_format* specify the format to be used by TeX.

The argument *start_code* defines the code to be executed first by TeX.

The argument *done_code* defines to code to be appended to the input code. This code must print on the standard output a *done_string* string in order to indicate the page was successfully and completely ship-out.

_process(*input_string*, *done_string*)
Process the input string.

fifos = ('texput.tex', 'texput.log', 'texput.dvi')
Defines the FIFOs used by the TeX daemon to communicate.

make_args()
Return the TeX calling sequence.

process(*input_string*)
Process the input string and return a dictionary with 'dvi', 'stdout', 'logfile' and 'stderr' entries.

start()
Start the TeX daemon.

Tools

DaemonSubprocess

This module provides functions to run Daemon process.

class PyDvi.Tools.DaemonSubprocess.**DaemonSubprocess**(*working_directory*)
Bases: object

This class implements a Daemon sub-process.

fifos = ()
List of fifos to be created.

kill()
Send Kill signal to the child process.

make_args()
Return the args for Popen. To be implemented in subclass.

restart()
Restart the child process.

start()
Start the child process.

stop()
Stop the child process.

exception PyDvi.Tools.DaemonSubprocess.**SubprocessError**
Bases: exceptions.EnvironmentError

PyDvi.Tools.DaemonSubprocess.**make_nonblocking**(*fd*)
Makes a file descriptor non-blocking.

When a non-blocking file is read, the read does not wait for end-of-file. Instead, the read can return just as soon as there is nothing left to read. This might be because a buffer is empty.

See Python Cookbook, Recipe 6.6.

EnumFactory

This module provides an implementation for enumerate.

The enum factory *EnumFactory()* builds a enumerate from a list of names and assigns to these constants a value from 0 to N-1, where N is the number of constants:

```
enum1 = EnumFactory('Enum1', ('cst1', 'cst2'))
```

then we can get a constant's value with:

```
enum1.cst1
```

and the number of constants using:

```
len(enum1)
```

The enum factory *ExplicitEnumFactory()* permits to specify the values of the constants:

```
enum2 = ExplicitEnumFactory('Enum2', {'cst1':1, 'cst2':3})
```

We can test if a value is in the enum using:

```
constant_value in enum2
```

PyDvi.Tools.EnumFactory.**EnumFactory**(*cls_name*, *constant_names*)
Return an EnumMetaClass instance, where *cls_name* is the class name and *constant_names* is an iterable of constant's names.

PyDvi.Tools.EnumFactory.**ExplicitEnumFactory**(*cls_name*, *constant_dict*)
Return an ExplicitEnumMetaClass instance, where *cls_name* is the class name and *constant_dict* is a dict of constant's names and their values.

FuncTools

PyDvi.Tools.FuncTools.**repeat_call**(*func*, *count*)
Call the function *func count* times and return the output as a list.

PyDvi.Tools.FuncTools.**get_filename_extension**(*filename*)
Return the filename extension.

Interval

class PyDvi.Tools.Interval.**Interval**(**args*)
Bases: object

One-dimension Interval

Initialise an interval

- Interval(inf, sup)
- else args must support the __getitem__ interface, e.g.:
- Interval((inf, sup))
- Interval([inf, sup])
- Interval(interval_instance)

_check_arguments(*args*)

static _intersection(*i1*, *i2*)

static _union(*i1*, *i2*)

copy()
Return a clone of the interval

enlarge(*dx*)
Enlarge the interval of dx

intersect(*i1*, *i2*)
Does the interval intersect with i2?

is_empty()

is_included_in(*i1*, *i2*)
Is the interval included in i1?

length()
Return sup - inf

middle()
Return interval middle

print_object()
Print the interval

zero_length()
Return sup == inf

class PyDvi.Tools.Interval.**IntervalInt**(**args*)
Bases: *PyDvi.Tools.Interval.Interval*

One-dimension Integer Interval

Initialise an interval

array must support the __getitem__ interface

length()
Return sup - inf +1

class PyDvi.Tools.Interval.**Interval2D**(*x*, *y*)
Bases: object

Two-dimension Interval

Initialise a 2D interval

x and y must support the __getitem__ interface

area()
Return the area

bounding_box()
Return the corresponding bounding box (x.inf, y.inf, x.sup, y.sup)

copy()
Return a clone of the interval

enlarge(*dx*)
Enlarge the interval of dx

intersect(*i2*)
Does the interval intersect with i2?

is_empty()

is_included_in(*i2*)
Is the interval included in i2?

middle()
Return interval middle

print_object()
Print the interval

size()
Return the horizontal and vertical size

class PyDvi.Tools.Interval.**IntervalInt2D**(*x*, *y*)
Bases: *PyDvi.Tools.Interval.Interval2D*

Two-dimension Integer Interval

Initialise a 2D Integer interval

x and y must support the __getitem__ interface

Logging

PyDvi.Tools.Logging.**format_card**(*text*, *centered=False*, *width=80*, *rule_char='#'*, *newline=False*, *border=False*, *bottom_rule=True*)
Format the string *text* as a card:

```
**************************************************
*
*                      Title
*
* xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx
* xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx
* xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx
* xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx
*
**************************************************
```

centered center the text in the card

width width of the card

rule_char character used to draw the rule

newline insert a new line before the card

border draw a left vertical rule

bottom_rule draw a bottom horizontal rule

PyDvi.Tools.Logging.**print_card**(*text*, ***kwargs*)
Print the string *text* formated by *format_card()*. The remaining keyword arguments *kwargs* are passed to *format_card()*.

PyDvi.Tools.Logging.**remove_enclosing_new_line**(*text*)
Return a copy of the string *text* with leading and trailing newline removed.

Path

PyDvi.Tools.Path.**find**(*file_name*, *directories*)

PyDvi.Tools.Path.**parent_directory_of**(*file_name*, *step=1*)

PyDvi.Tools.Path.**to_absolute_path**(*path*)

RevisionVersion

class PyDvi.Tools.RevisionVersion.**RevisionVersion**(*version*)
Bases: object

This class implements a revision version of the form vx.y.z where x, y and z are the major, minor and revision number, respectively.

To compare two version, the version x.y.z is converted to an integer using the following formulae: (x * scale + y) * scale + z. Thus x, y and z must be less than the *scale*.

version could be a version string or a sequence of three integers.

Examples:

```
RevisionVersion('v0.1.2')
RevisionVersion((0,1,2))
RevisionVersion([0,1,2])
```

Two Instances can be compared using operator: ==, <, >, <=, >=.

An instance can be formated using `str()` function.

scale = 1000000
default scale value

Stream

class PyDvi.Tools.Stream.**AbstractStream**
Bases: object

Abstract class to read DVI, PK, TFM and VF streams.

The followings methods are abstracts:

- *read()*
- *seek()*
- *tell()*

and must be implemented in subclass.

In the followings methods, the *position* argument is used to specify a position in the stream for the read operation. If *position* is not None, it seeks to the specified position before to read the stream else it reads from the current position. See also *read_bytes()*.

read(*number_of_bytes*)
Read *n* bytes from the current position.

read_bcpl(*position=None*)
Read a BCPL string.

The BCPL string format comes from the Basic Combined Programming Language. The length of the string is given by the first byte, thus its length is limited to 256 characters.

read_big_endian_number(*number_of_bytes*, *signed=False*, *position=None*)
Read a signed or an unsigned integer encoded in big endian order with *number_of_bytes* bytes, cf. *read_bytes()*.

read_byte_numbers(*number_of_bytes*, *position=None*)
Read *number_of_bytes* times 8-bit unsigned integers, cf. *read_bytes()*.

read_bytes(*number_of_bytes*, *position=None*)
Read *number_of_bytes* bytes from the optional position or the current position. If *position*, it seeks to the specified position and thus change the current position in the stream.

read_fix_word(*position=None*)
Read a fix word.

read_four_byte_numbers(*position=None*)
Read four 8-bit unsigned integers, cf. *read_bytes()*.

read_signed_byte1(*position=None*)
Read a 1-byte signed integer, cf. *read_big_endian_number()*.

read_signed_byte2(*position=None*)
Read a 2-byte signed integer, cf. *read_big_endian_number()*.

read_signed_byte3(*position=None*)
Read a 3-byte signed integer, cf. *read_big_endian_number()*.

read_signed_byte4(*position=None*)
Read a 4-byte signed integer, cf. *read_big_endian_number()*.

read_signed_byten = (<function read_signed_byte1>, <function read_signed_byte2>, <function read_signed_byt
This tuple permits to get the *read_signed_byte* method with the number of bytes as index.

read_three_byte_numbers(*position=None*)
Read three 8-bit unsigned integers, cf. *read_bytes()*.

read_unsigned_byte1(*position=None*)
Read a 1-byte unsigned integer, cf. *read_big_endian_number()*.

read_unsigned_byte2(*position=None*)
Read a 2-byte unsigned integer, cf. *read_big_endian_number()*.

read_unsigned_byte3(*position=None*)
Read a 3-byte unsigned integer, cf. *read_big_endian_number()*.

read_unsigned_byte4(*position=None*)
Read a 4-byte unsigned integer, cf. *read_big_endian_number()*.

read_unsigned_byten = (<function read_unsigned_byte1>, <function read_unsigned_byte2>, <function read_uns
This tuple permits to get the read_unsigned_byte method with the number of bytes as index.

seek(*postion*, *whence*)
Seek to position.

tell()
Tell the current position.

class PyDvi.Tools.Stream.**StandardStream**
Bases: *PyDvi.Tools.Stream.AbstractStream*

Abstract stream class.

The attribute `stream` must be defined in subclass.

read(*number_of_bytes*)
Read *n* bytes from the current position and return a `bytearray`

seek(*postion*, *whence=0*)
Seek to position.

tell()
Tell the current position.

class PyDvi.Tools.Stream.**FileStream**(*filename*)
Bases: *PyDvi.Tools.Stream.StandardStream*

class PyDvi.Tools.Stream.**ByteStream**(*string_bytes*)
Bases: *PyDvi.Tools.Stream.StandardStream*

end_of_stream()

PyDvi.Tools.Stream.**to_fix_word**(*x*)
Convert *x* to a fix word.

A fix word is a 32-bit representation of a binary fraction. A fix word is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a fix word, exactly 12 are to the left of the binary point; thus, the largest fix word value is 2048 - 2**-20, and the smallest is -2048.

```
fix word = x / 2**20
```

TexCommentedFile

class PyDvi.Tools.TexCommentedFile.**TexCommentedFile**(*filename*)
Bases: `file`

This class permits to iterate over lines of a text file and to skip commented line by '%'.

concatenate_lines()
Concatenate the lines and return the corresponding string.

Version

PyDvi.Version.**pydvi = <PyDvi.Tools.RevisionVersion.RevisionVersion object>**
defines the PyDvi revision version.

PyDviPng

Config

ConfigInstall

DviMachine

Logging

Logging

PyDviGui

Config

Config

ConfigInstall

Messages

DviGlViewer

ApplicationStatusBar

DviMachine

GlWidgetV4

MainWindow

PrimitiveVertexArray

ShaderProgramesV4

TextVertexArray

TextureFont

DviViewer

MainWindow

FontViewer

FontInfoTableModel

Glyph

GlyphGraphicsView

GlyphInfoTableModel

InfoTableModel

MainWindow

Logging

Logging

Tools

Math

Path

Platform

Singleton

Widgets

ApplicationBase

IconLoader

MainWindowBase

ui

dvi_viewer_ui

font_viewer_ui

pydvi_rc

Indexes

- genindex
- modindex
- search

Bibliography

This software was designed according to the official TeX Web2C documentation (http://www.tug.org/web2c/).

Relevant files are:

- `dvicopy.web`
- `dvitype.web`
- `pktype.web`
- `pltotf.web`
- `tftopl.web`
- `vftovp.web`

A copy of this documentation is available here and later in the API documentation:

Device-Independent File Format

The Device-independent file format is described in the `dvitype.web` file from Web2C. Part of this documentation comes from this file.

The DVI format was designed by David R. Fuchs in 1979.

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the `set_rule` command has two parameters, each of which is four bytes long. Parameters are usually regarded as

non negative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2**15 and 2**15 -1.

A DVI file consists of a "preamble", followed by a sequence of one or more "pages", followed by a "postamble". The preamble is simply a `pre` command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a `bop` command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an `eop` command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore `nop` commands and `fnt_def` commands (which are allowed between any two commands in the file), each `eop` command is immediately followed by a `bop` command, or by a post command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are "pointers". These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a `bop` command points to the previous `bop`; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the `bop` that starts in byte 1000 points to 100 and the `bop` that starts in byte 2000 points to 1000. (The very first `bop`, i.e. the one that starts in byte 100, has a pointer of -1.)

The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font `f` is an integer; this value is changed only by `fnt` and `fnt_num` commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, `h` and `v`. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of `(h, v)` would be `(h, -v)`. (c) The current spacing amounts are given by four numbers `w`, `x`, `y`, and `z`, where `w` and `x` are used for horizontal spacing and where `y` and `z` are used for vertical spacing. (d) There is a stack containing `(h, v, w, x, y, z)` values; the DVI commands `push` and `pop` are used to change the current level of operation. Note that the current font `f` is not pushed and popped; the stack contains only information about positioning.

The values of `h`, `v`, `w`, `x`, `y`, and `z` are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing `h` by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

Here is a list of all the commands that may appear in a DVI file. Each command is specified by its symbolic name (e.g. `bop`), its opcode byte (e.g. 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, `p[4]` means that parameter `p` is four bytes long.

`set_char_0 0`. Typeset character number 0 from font `f` such that the reference point of the character is at `(h, v)`. Then increase `h` by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that `h` will advance after this command; but `h` usually does increase.

`set_char 1` through set char 127 (opcodes 1 to 127). Do the operations of `set_char_0`; but use the character whose number matches the opcode, instead of character 0.

`set1 128 c[1]`. Same as `set_char_0`, except that character number `c` is typeset. TEX82 uses this command for characters in the range `128 <= c < 256`.

`set2 129 c[2]`. Same as `set1` , except that `c` is two bytes long, so it is in the range `0 <= c < 65536`. TEX82 never uses this command, which is intended for processors that deal with oriental languages; but DVItype will allow character codes greater than 255, assuming that they all have the same width as the character whose code is `c` mod 256.

`set3 130 c[3]`. Same as `set1`, except that `c` is three bytes long, so it can be as large as 2**24 -1.

`set4 131 c[4]`. Same as `set1`, except that `c` is four bytes long, possibly even negative. Imagine that.

`set_rule 132 a[4] b[4]`. Typeset a solid black rectangle of height `a` and width `b`, with its bottom left corner at `(h, v)`. Then set `h = h + b`. If either `a <= 0` or `b <= 0`, nothing should be typeset. Note that if `b < 0`, the value of `h` will decrease even though nothing else happens. Programs that typeset from DVI files

should be careful to make the rules line up carefully with digitised characters, as explained in connection with the rule pixels subroutine below.

`put1 133 c[1]`. Typeset character number `c` from font `f` such that the reference point of the character is at `(h, v)`. (The `put` commands are exactly like the `set` commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

`put2 134 c[2]`. Same as `set2`, except that `h` is not changed.

`put3 135 c[3]`. Same as `set3`, except that `h` is not changed.

`put4 136 c[4]`. Same as `set4`, except that `h` is not changed.

`put_rule 137 a[4] b[4]`. Same as `set_rule`, except that `h` is not changed.

`nop 138`. No operation, do nothing. Any number of `nop`'s may occur between DVI commands, but a `nop` cannot be inserted between a command and its parameters or between two parameters.

`bop 139 c0[4] c1[4] ... c9[4] p[4]`. Beginning of a page: Set `(h, v, w, x, y, z) = (0, 0, 0, 0, 0, 0)` and set the stack empty. Set the current font `f` to an undefined value. The ten `ci` parameters can be used to identify pages, if a user wants to print only part of a DVI file; TEX82 gives them the values of `count0 ... count9` at the time `shipout` was invoked for this page. The parameter `p` points to the previous `bop` command in the file, where the first `bop` has `p = -1`.

`eop 140`. End of page: Print what you have read since the previous `bop`. At this point the stack should be empty. (The DVI-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by `v` coordinate and (for fixed `v`) by `h` coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question. DVItype does not do such sorting.)

`push 141`. Push the current values of `(h, v, w, x, y, z)` onto the top of the stack; do not change any of these values. Note that `f` is not pushed.

`pop 142`. Pop the top six values off of the stack and assign them to `(h, v, w, x, y, z)`. The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a pop command.

`right1 143 b[1]`. Set `h = h + b`, i.e. move right `b` units. The parameter is a signed number in two's complement notation, `-128 <= b < 128`; if `b < 0`, the reference point actually moves left.

`right2 144 b[2]`. Same as `right1`, except that `b` is a two-byte quantity in the range `-32768 <= b < 32768`.

`right3 145 b[3]`. Same as `right1`, except that `b` is a three-byte quantity in the range `-2**23 <= b < 2**23`.

`right4 146 b[4]`. Same as `right1`, except that `b` is a four-byte quantity in the range `-2**31 <= b < 2**31`.

`w0 147`. Set `h = h + w`; i.e. move right `w` units. With luck, this parameter-less command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how `w` gets particular values.

`w1 148 b[1]`. Set `w = b` and `h = h + b`. The value of `b` is a signed quantity in two's complement notation, `-128 <= b < 128`. This command changes the current `w` spacing and moves right by `b`.

`w2 149 b[2]`. Same as `w1`, but `b` is a two-byte-long parameter, `-32768 <= b < 32768`.

`w3 150 b[3]`. Same as `w1`, but `b` is a three-byte-long parameter, `-2**23 <= b < 2**23`.

`w4 151 b[4]`. Same as `w1`, but `b` is a four-byte-long parameter, `-2**31 <= b < 2**31`.

`x0 152`. Set `h = h + x`; i.e. move right `x` units. The `x` commands are like the `w` commands except that they involve `x` instead of `w`.

`x1 153 b[1]`. Set `x = b` and `h = h + b`. The value of `b` is a signed quantity in two's complement notation, `-128 <= b < 128`. This command changes the current `x` spacing and moves right by `b`.

`x2 154 b[2]`. Same as `x1`, but `b` is a two-byte-long parameter, `-32768 <= b < 32768`.

x3 155 b[3]. Same as x1, but b is a three-byte-long parameter, -2**23 <= b < 2**23.

x4 156 b[4]. Same as x1, but b is a four-byte-long parameter, -2**31 <= b < 2**31.

down1 157 a[1]. Set v = v + a, i.e. move down a units. The parameter is a signed number in two's complement notation, -128 <= a < 128; if a < 0, the reference point actually moves up.

down2 158 a[2]. Same as down1, except that a is a two-byte quantity in the range -32768 <= a < 32768.

down3 159 a[3]. Same as down1, except that a is a three-byte quantity in the range -2**23 <= a < 2**23.

down4 160 a[4]. Same as down1, except that a is a four-byte quantity in the range -2**31 <= a < 2**31.

y0 161. Set v = v + y; i.e. move down y units. With luck, this parameter-less command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how y gets particular values.

y1 162 a[1]. Set y = a and v = v + a. The value of a is a signed quantity in two's complement notation, -128 <= a < 128. This command changes the current y spacing and moves down by a.

y2 163 a[2]. Same as y1, but a is a two-byte-long parameter, -32768 <= a < 32768.

y3 164 a[3]. Same as y1, but a is a three-byte-long parameter, -2**23 <= a < 2**23.

y4 165 a[4]. Same as y1, but a is a four-byte-long parameter, -2**31 <= a < 2**31.

z0 166. Set v = v + z; i.e. move down z units. The z commands are like the y commands except that they involve z instead of y.

z1 167 a[1]. Set z = a and v = v + a. The value of a is a signed quantity in two's complement notation, -128 <= a < 128. This command changes the current z spacing and moves down by a.

z2 168 a[2]. Same as z1, but a is a two-byte-long parameter, -32768 <= a < 32768.

z3 169 a[3]. Same as z1, but a is a three-byte-long parameter, -2**23 <= a < 2**23.

z4 170 a[4]. Same as z1, but a is a four-byte-long parameter, -2**31 <= a < 2**31.

fnt_num_0 171. Set f = 0. Font 0 must previously have been defined by a fnt_def instruction, as explained below.

fnt_num_1 through fnt_num_63 (opcodes 172 to 234). Set f = 1, ... , f = 63, respectively.

fnt1 235 k[1]. Set f = k. TEX82 uses this command for font numbers in the range 64 <= k < 256.

fnt2 236 k[2]. Same as fnt1, except that k is two bytes long, so it is in the range 0 <= k < 65536. TEX82 never generates this command, but large font numbers may prove useful for specifications of colour or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 k[3]. Same as fnt1, except that k is three bytes long, so it can be as large as 2**24 - 1.

fnt4 238 k[4]. Same as fnt1, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 239 k[1] x[k]. This command is undefined in general; it functions as a (k+2)-byte nop unless special DVI-reading programs are being used. TEX82 generates xxx1 when a short enough special appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 k[2] x[k]. Like xxx1, but 0 <= k < 65536.

xxx3 241 k[3] x[k]. Like xxx1, but 0 <= k < 224.

xxx4 242 k[4] x[k]. Like xxx1, but k can be ridiculously large. TEX82 uses xxx4 when xxx1 would be incorrect.

fnt_def1 243 k[1] c[4] s[4] d[4] a[1] l[1] n[a + l]. Define font k, where 0 <= k < 256; font definitions will be explained shortly.

`fnt_def2 244 k[2] c[4] s[4] d[4] a[1] l[1] n[a + l]`. Define font `k`, where `0 <= k < 65536`.

`fnt_def3 245 k[3] c[4] s[4] d[4] a[1] l[1] n[a + l]`. Define font `k`, where `0 <= k < 224`.

`fnt_def4 246 k[4] c[4] s[4] d[4] a[1] l[1] n[a + l]`. Define font `k`, where `-2**31 <= k < 2**31`.

`pre 247 i[1] num[4] den [4] mag[4] k[1] x[k]`. Beginning of the preamble; this must come at the very beginning of the file. Parameters `i`, `num`, `den`, `mag`, `k`, and `x` are explained below.

`post 248`. Beginning of the postamble, see below.

`post_post 249`. Ending of the postamble, see below.

Commands 250-255 are undefined at the present time.

The preamble contains basic information about the file as a whole. As stated above, there are six parameters: `i[1] num[4] den [4] mag[4] k[1] x[k]`.

The `i` byte identifies DVI format; currently this byte is always set to 2. (The value `i = 3` is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting.

The next two parameters, `num` and `den`, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 1e-7 meters. (For example, there are exactly 7227 TEX points in 254 centimetres, and TEX82 works with scaled points where there are 216 sp in a point, so TEX82 sets `num = 25400000` and `den = 7227 * 2**16 = 473628672`.)

The `mag` parameter is what TEX82 calls `mag`, i.e. 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore `mn/1000d`. Note that if a TEX source document does not call for any `true` dimensions, and if you change it only by specifying a different `mag` setting, the DVI file that TEX creates will be completely unchanged except for the value of `mag` in the preamble and postamble. (Fancy DVI-reading programs allow users to override the mag setting when a DVI file is being printed.)

Finally, `k` and `x` allow the DVI writer to include a comment, which is not interpreted further. The length of comment `x` is `k`, where `0 <= k < 256`.

Font definitions for a given font number `k` contain further parameters `c[4] s[4] d[4] a[1] l[1] n[a + l]`.

The four-byte value `c` is the check sum that TEX (or whatever program generated the DVI file) found in the TFM file for this font; `c` should match the check sum of the font found by programs that read this DVI file.

Parameter `s` contains a fixed-point scale factor that is applied to the character widths in font `k`; font dimensions in TFM files and other font files are relative to this quantity, which is always positive and less than 2**27. It is given in the same units as the other dimensions of the DVI file. Parameter `d` is similar to `s`; it is the "design size", and (like `s`) it is given in DVI units. Thus, font `k` is to be used at `mag * s/1000d` times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length `a + l`. The number `a` is the length of the "area" or directory, and `l` is the length of the font name itself; the standard local system font area is supposed to be used when `a = 0`. The `n` field contains the area in its first `a` bytes.

Font definitions must appear before the first use of a particular font number. Once font `k` is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like `nop` commands, font definitions can appear before the first `bop`, or between an `eop` and a `bop`.

The last page in a DVI file is followed by `post`; this command introduces the postamble, which summarises important facts that TEX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form:

```
post p[4] num[4] den [4] mag[4] l[4] u[4] s[2] t[2]
font definitions
post_post q[4] i[1] 223's[>=4]
```

Here p is a pointer to the final bop in the file. The next three parameters, num, den, and mag, are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to centre the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u are often ignored.

Parameter s is the maximum stack depth (i.e. the largest excess of push commands over pop commands) needed to process this file. Then comes t, the total number of pages (bop commands) present.

The postamble continues with font definitions, which are any number of fnt_def commands as described above, possibly interspersed with nop commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a fnt command, and once in the postamble.

The last part of the postamble, following the post_post byte that signifies the end of the font definitions, contains q, a pointer to the post command that started the postamble. An identification byte i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223. TEX puts out four to seven of these trailing bytes, until the total length of the file length is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TEX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (post); now the postamble can be read, so the DVI reader discovers all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

Packet Font File Format

The Packet Font file format is described in the pktype.web file from Web2C. Part of this documentation comes from this file.

The packed file format is a compact representation of the data contained in a GF file. The information content is the same, but packed (PK) files are almost always less than half the size of their GF counterparts. They are also easier to convert into a raster representation because they do not have a profusion of paint, skip, and new_row commands to be separately interpreted. In addition, the PK format expressly forbids special commands within a character. The minimum bounding box for each character is explicit in the format, and does not need to be scanned for as in the GF format. Finally, the width and escapement values are combined with the raster information into character "packets", making it simpler in many cases to process a character.

A PK file is organised as a stream of 8-bit bytes. At times, these bytes might be split into 4-bit nybbles or single bits, or combined into multiple byte parameters. When bytes are split into smaller pieces, the "first" piece is always the most significant of the byte. For instance, the first bit of a byte is the bit with value 128; the first nybble can be found by dividing a byte by 16. Similarly, when bytes are combined into multiple byte parameters, the first byte is the most significant of the parameter. If the parameter is signed, it is represented by two's-complement notation.

The set of possible eight-bit values is separated into two sets, those that introduce a character definition, and those that do not. The values that introduce a character definition range from 0 to 239; byte values above 239 are interpreted as commands. Bytes that introduce character definitions are called flag bytes, and various fields within the byte indicate various things about how the character definition is encoded. Command bytes have zero or more parameters, and can never appear within a character definition or between parameters of another command, where they would be interpreted as data.

A PK file consists of a preamble, followed by a sequence of one or more character definitions, followed by a postamble. The preamble command must be the first byte in the file, followed immediately by its parameters. Any number of character definitions may follow, and any command but the preamble command and the postamble command may occur between character definitions. The very last command in the file must be the postamble.

The packed file format is intended to be easy to read and interpret by device drivers. The small size of the file reduces the input/output overhead each time a font is loaded. For those drivers that load and save each font file into memory, the small size also helps reduce the memory requirements. The length of each character packet is specified, allowing the character raster data to be loaded into memory by simply counting bytes, rather than interpreting each command; then, each character can be interpreted on a demand basis. This also makes it possible for a driver to skip a particular character quickly if it knows that the character is unused.

First, the command bytes will be presented; then the format of the character definitions will be defined. Eight of the possible sixteen commands (values 240 through 255) are currently defined; the others are reserved for future extensions. The commands are listed below. Each command is specified by its symbolic name (e.g., `pk_no_op`), its opcode byte, and any parameters. The parameters are followed by a bracketed number telling how many bytes they occupy, with the number preceded by a plus sign if it is a signed quantity. (Four byte quantities are always signed, however.)

`pk_xxx1 240 k[1] x[k]`. This command is undefined in general; it functions as a `(k + 2)`-byte `no_op` unless special PK-reading programs are being used. METAFONT generates `xxx` commands when encountering a special string. It is recommended that `x` be a string having the form of a keyword followed by possible parameters relevant to that keyword.

`pk_xxx2 241 k[2] x[k]`. Like `pk_xxx1`, but `0 <= k < 65536`.

`pk_xxx3 242 k[3] x[k]`. Like `pk_xxx1`, but `0 <= k < 224`. METAFONT uses this when sending a special string whose length exceeds 255.

`pk_xxx4 243 k[4] x[k]`. Like `pk_xxx1`, but `k` can be ridiculously large; `k` mustn't be negative.

`pk yyy 244 y[4]`. This command is undefined in general; it functions as a five-byte `no_op` unless special PK reading programs are being used. METAFONT puts scaled numbers into `yyy`'s, as a result of `numspecial` commands; the intent is to provide numeric parameters to `xxx` commands that immediately precede.

`pk post 245`. Beginning of the postamble. This command is followed by enough `pk_no_op` commands to make the file a multiple of four bytes long. Zero through three bytes are usual, but any number is allowed. This should make the file easy to read on machines that pack four bytes to a word.

`pk_no_op 246`. No operation, do nothing. Any number of `pk_no_op`'s may appear between PK commands, but a `pk_no_op` cannot be inserted between a command and its parameters, between two parameters, or inside a character definition.

`pk_pre 247 i[1] k[1] x[k] ds [4] cs [4] hppp[4] vppp[4]`. Preamble command. Here, `i` is the identification byte of the file, currently equal to 89. The string `x` is merely a comment, usually indicating the source of the PK file. The parameters `ds` and `cs` are the design size of the file in 1/2**20 points, and the checksum of the file, respectively. The checksum should match the TFM file and the GF files for this font. Parameters `hppp` and `vppp` are the ratios of pixels per point, horizontally and vertically, multiplied by 2**16; they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes". Usually, the name of the PK file is formed by concatenating the font name (e.g., cmr10) with the resolution at which the font is prepared in pixels per inch multiplied by the magnification factor, and the letters `pk`. For instance, cmr10 at 300 dots per inch should be named `cmr10.300pk`; at one thousand dots per inch and `magstephalf`, it should be named `cmr10.1095pk`.

The PK format has two conflicting goals: to pack character raster and size information as compactly as possible, while retaining ease of translation into raster and other forms. A suitable compromise was found in the use of run-encoding of the raster information. Instead of packing the individual bits of the character, we instead count the number of consecutive "black" or "white" pixels in a horizontal raster row, and then encode this number. Run counts are found for each row from left to right, traversing rows from the top to bottom. This is essentially the way the GF format works. Instead of presenting each row individually, however, we concatenate all of the horizontal raster rows into one long string of pixels, and encode this row. With knowledge of the width of the bit-map, the original character glyph can easily be reconstructed. In addition, we do not need special commands to mark the end of one row and the beginning of the next.

Next, we place the burden of finding the minimum bounding box on the part of the font generator, since the characters will usually be used much more often than they are generated. The minimum bounding box is the smallest rectangle that encloses all "black" pixels of a character. We also eliminate the need for a special end of character marker, by supplying exactly as many bits as are required to fill the minimum bounding box, from which the end of the character is implicit.

Let us next consider the distribution of the run counts. Analysis of several dozen pixel files at 300 dots per inch yields a distribution peaking at four, falling off slowly until ten, then a bit more steeply until twenty, and then asymptotically approaching the horizontal. Thus, the great majority of our run counts will fit in a four-bit nybble. The eight-bit byte is attractive for our run-counts, as it is the standard on many systems; however, the wasted four bits in the majority of cases seem a high price to pay. Another possibility is to use a Huffman-type encoding scheme with a variable number of bits for each run-count; this was rejected because of the overhead in fetching and examining individual bits in the file. Thus, the character raster definitions in the PK file format are based on the four-bit nybble.

An analysis of typical pixel files yielded another interesting statistic: Fully 37% of the raster rows were duplicates of the previous row. Thus, the PK format allows the specification of repeat counts, which indicate how many times a horizontal raster row is to be repeated. These repeated rows are taken out of the character glyph before individual rows are concatenated into the long string of pixels.

For elegance, we disallow a run count of zero. The case of a null raster description should be gleaned from the character width and height being equal to zero, and no raster data should be read. No other zero counts are ever necessary. Also, in the absence of repeat counts, the repeat value is set to be zero (only the original row is sent.) If a repeat count is seen, it takes effect on the current row. The current row is defined as the row on which the first pixel of the next run count will lie. The repeat count is set back to zero when the last pixel in the current row is seen, and the row is sent out.

This poses a problem for entirely black and entirely white rows, however. Let us say that the current row ends with four white pixels, and then we have five entirely empty rows, followed by a black pixel at the beginning of the next row, and the character width is ten pixels. We would like to use a repeat count, but there is no legal place to put it. If we put it before the white run count, it will apply to the current row. If we put it after, it applies to the row with the black pixel at the beginning. Thus, entirely white or entirely black repeated rows are always packed as large run counts (in this case, a white run count of 54) rather than repeat counts.

Now we turn our attention to the actual packing of the run counts and repeat counts into nybbles. There are only sixteen possible nybble values. We need to indicate run counts and repeat counts. Since the run counts are much more common, we will devote the majority of the nybble values to them. We therefore indicate a repeat count by a nybble of 14 followed by a packed number, where a packed number will be explained later. Since the repeat count value of one is so common, we indicate a repeat one command by a single nybble of 15. A 14 followed by the packed number 1 is still legal for a repeat one count. The run counts are coded directly as packed numbers.

For packed numbers, therefore, we have the nybble values 0 through 13. We need to represent the positive integers up to, say, 2**31 - 1. We would like the more common smaller numbers to take only one or two nybbles, and the infrequent large numbers to take three or more. We could therefore allocate one nybble value to indicate a large run count taking three or more nybbles. We do this with the value 0.

We are left with the values 1 through 13. We can allocate some of these, say `dyn_f`, to be one-nybble run counts. These will work for the run counts 1 .. `dyn_f`. For subsequent run counts, we will use a nybble greater than `dyn_f`, followed by a second nybble, whose value can run from 0 through 15. Thus, the two-nybble values will run from `dyn_f +1 .. (13 - dyn_f) * 16 + dyn_f`. We have our definition of large run count values now, being all counts greater than `(13 - dyn_f) * 16 + dyn_f`.

We can analyse our several dozen pixel files and determine an optimal value of `dyn_f`, and use this value for all of the characters. Unfortunately, values of `dyn_f` that pack small characters well tend to pack the large characters poorly, and values that pack large characters well are not efficient for the smaller characters. Thus, we choose the optimal `dyn_f` on a character basis, picking the value that will pack each individual character in the smallest number of nybbles. Legal values of `dyn_f` run from 0 (with no one-nybble run counts) to 13 (with no two-nybble run counts).

Our only remaining task in the coding of packed numbers is the large run counts. We use a scheme suggested by D. E. Knuth that simply and elegantly represents arbitrarily large values. The general scheme to represent an integer `i` is to write its hexadecimal representation, with leading zeros removed. Then we count the number of digits, and

prepend one less than that many zeros before the hexadecimal representation. Thus, the values from one to fifteen occupy one nybble; the values sixteen through 255 occupy three, the values 256 through 4095 require five, etc.

For our purposes, however, we have already represented the numbers one through `(13 - dyn_f) * 16 + dyn_f ``. In addition, the one-nybble values have already been taken by our other commands, which means that only the values from sixteen up are available to us for long run counts. Thus, we simply normalise our long run counts, by subtracting ``(13 - dyn_f) * 16 + dyn_f +1` and adding 16, and then we represent the result according to the scheme above.

The final algorithm for decoding the run counts based on the above scheme looks like this, assuming that a procedure called `get_nyb` is available to get the next nybble from the file, and assuming that the global repeat count indicates whether a row needs to be repeated. Note that this routine is recursive, but since a repeat count can never directly follow another repeat count, it can only be recursive to one level:

```
function pk packed num: integer ;
  var i; j: integer;
  begin i <- get nyb;
    if i = 0 then
      begin repeat j <- get_nyb; incr(i);
      until j != 0;
      while i > 0 do
        begin j <- j * 16 + get_nyb; decr(i);
        end;
      pk_packed_num <- j - 15 + (13 - dyn_f ) * 16 + dyn_f;
      end
     else if i <= dyn_f then pk_packed_num <- i
       else if i < 14 then pk_packed_num <- (i - dyn_f - 1) * 16 + get_nyb + dyn_f␣
↪+ 1
         else begin if repeat count != 0 then abort('Second repeat count for this␣
↪row!');
           repeat_count <- 1; { prevent recursion more than one level }
           if i = 14 then repeat_count <- pk_packed_num;
           send_out(true, repeat count); pk_packed_num <- pk_packed_num;
           end;
  end;
```

For low resolution fonts, or characters with “gray” areas, run encoding can often make the character many times larger. Therefore, for those characters that cannot be encoded efficiently with run counts, the PK format allows bit-mapping of the characters. This is indicated by a `dyn_f` value of 14. The bits are packed tightly, by concatenating all of the horizontal raster rows into one long string, and then packing this string eight bits to a byte. The number of bytes required can be calculated by `(width * height + 7) div 8`. This format should only be used when packing the character by run counts takes more bytes than this, although, of course, it is legal for any character. Any extra bits in the last byte should be set to zero.

At this point, we are ready to introduce the format for a character descriptor. It consists of three parts: a flag byte, a character preamble, and the raster data. The most significant four bits of the flag byte yield the `dyn_f` value for that character. (Notice that only values of 0 through 14 are legal for `dyn_f`, with 14 indicating a bit mapped character; thus, the flag bytes do not conflict with the command bytes, whose upper nybble is always 15.) The next bit (with weight 8) indicates whether the first run count is a black count or a white count, with a one indicating a black count. For bit-mapped characters, this bit should be set to a zero. The next bit (with weight 4) indicates whether certain later parameters (referred to as size parameters) are given in one-byte or two-byte quantities, with a one indicating that they are in two-byte quantities. The last two bits are concatenated on to the beginning of the packet-length parameter in the character preamble, which will be explained below.

However, if the last three bits of the flag byte are all set (normally indicating that the size parameters are two-byte values and that a 3 should be prepended to the length parameter), then a long format of the character preamble should be used instead of one of the short forms.

Therefore, there are three formats for the character preamble; the one that is used depends on the least significant three bits of the flag byte. If the least significant three bits are in the range zero through three, the short format is used. If they are in the range four through six, the extended short format is used. Otherwise, if the least significant bits are all set, then the long form of the character preamble is used. The preamble formats are explained below.

Short form: `flag [1] pl [1] cc[1] tfm [3] dm[1] w[1] h[1] hoff [+1] voff [+1]`. If this format of the character preamble is used, the above parameters must all fit in the indicated number of bytes, signed or unsigned as indicated. Almost all of the standard TEX font characters fit; the few exceptions are fonts such as cminch.

Extended short form: `flag [1] pl [2] cc[1] tfm [3] dm[2] w[2] h[2] hoff [+2] voff [+2]`. Larger characters use this extended format.

Long form: `flag [1] pl [4] cc[4] tfm [4] dx [4] dy [4] w[4] h[4] hoff [4] voff [4]`. This is the general format that allows all of the parameters of the GF file format, including vertical escapement.

The flag parameter is the flag byte. The parameter `pl` (packet length) contains the offset of the byte following this character descriptor, with respect to the beginning of the tfm width parameter. This is given so a PK reading program can, once it has read the flag byte, packet length, and character code (`cc`), skip over the character by simply reading this many more bytes. For the two short forms of the character preamble, the last two bits of the flag byte should be considered the two most-significant bits of the packet length. For the short format, the true packet length might be calculated as `(flag mod 4) * 256 + pl`; for the short extended format, it might be calculated as `(flag mod 4) * 65536 + pl`.

The `w` parameter is the width and the `h` parameter is the height in pixels of the minimum bounding box. The `dx` and `dy` parameters are the horizontal and vertical escapements, respectively. In the short formats, `dy` is assumed to be zero and `dm` is `dx` but in pixels; in the long format, `dx` and `dy` are both in pixels multiplied by 2**16. The `hoff` is the horizontal offset from the upper left pixel to the reference pixel; the `voff` is the vertical offset. They are both given in pixels, with right and down being positive. The reference pixel is the pixel that occupies the unit square in METAFONT; the METAFONT reference point is the lower left hand corner of this pixel.

Virtual Font File Format

The Virtual Font file format is described in the `vftovp.web` file from Web2C. Part of this documentation comes from this file.

The idea behind VF files is that a general interface mechanism is needed to switch between the myriad font layouts provided by different suppliers of typesetting equipment. Without such mechanism, people must go to great lengths writing inscrutable macros whenever they want to use typesetting conventions based on one font layout in connection with actual fonts that have another layout. This puts an extra burden on the typesetting system, interfering with the other things it needs to do (like kerning, hyphenation, and ligature formation).

These difficulties go away when we have a "virtual font," i.e., a font that exists in a logical sense but not a physical sense. A typesetting system like TEX can do its job without knowing where the actual characters come from; a device driver can then do its job by letting a VF file tell what actual characters correspond the characters TEX imagined were present. The actual characters can be shifted and/or magnified and/or combined with other characters from many different fonts. A virtual font can even make use of characters from virtual fonts, including itself.

Virtual fonts also allow convenient character substitutions for proofreading purposes, when fonts designed for one output device are unavailable on another.

A VF file is organised as a stream of 8-bit bytes, using conventions borrowed from DVI and PK files. Thus, a device driver that knows about DVI and PK format will already contain most of the mechanisms necessary to process VF files. We shall assume that DVI format is understood; the conventions in the DVI documentation (see, for example, TEX: The Program, part 31) are adopted here to define VF format.

A preamble appears at the beginning, followed by a sequence of character definitions, followed by a postamble. More precisely, the first byte of every VF file must be the first byte of the following "preamble command": `pre 247 i[1] k[1] x[k] cs [4] ds [4]`. Here `i` is the identification byte of VF, currently 202. The string `x` is merely a comment, usually indicating the source of the VF file. Parameters `cs` and `ds` are respectively the check sum and the design size of the virtual font; they should match the first two words in the header of the TFM file, as described below.

After the `pre` command, the preamble continues with font definitions; every font needed to specify "actual" characters in later set char commands is defined here. The font definitions are exactly the same in VF files as they

are in DVI files, except that the scaled size s is relative and the design size d is absolute:

- fnt def1 243 k[1] c[4] s[4] d[4] a[1] l[1] n[a + l]. Define font k, where 0 k < 256.
- fnt def2 244 k[2] c[4] s[4] d[4] a[1] l[1] n[a + l]. Define font k, where 0 k < 65536.
- fnt def3 245 k[3] c[4] s[4] d[4] a[1] l[1] n[a + l]. Define font k, where 0 k < 2**24.
- fnt def4 246 k[4] c[4] s[4] d[4] a[1] l[1] n[a + l]. Define font k, where 2**31 k < 2**31.

These font numbers k are "local"; they have no relation to font numbers defined in the DVI file that uses this virtual font. The dimension s, which represents the scaled size of the local font being defined, is a fix word relative to the design size of the virtual font. Thus if the local font is to be used at the same size as the design size of the virtual font itself, s will be the integer value 2**20. The value of s must be positive and less than 2**24 (thus less than 16 when considered as a fix word). The dimension d is a fix word in units of printer's points; hence it is identical to the design size found in the corresponding TFM file.

The preamble is followed by zero or more character packets, where each character packet begins with byte that is < 243. Character packets have two formats, one long and one short:

- long char 242 pl [4] cc [4] tfm [4] dvi [pl]. This long form specifies a virtual character in the general case.
- short char0 ... short char241 pl [1] cc [1] tfm [3] dvi [pl]. This short form specifies a virtual character in the common case when 0 pl < 242 and 0 cc < 256 and 0 tfm < 2**24.

Here pl denotes the packet length following the tfm value; cc is the character code; and tfm is the character width copied from the TFM file for this virtual font. There should be at most one character packet having any given cc code.

The dvi bytes are a sequence of complete DVI commands, properly nested with respect to push and pop. All DVI operations are permitted except bop, eop, and commands with opcodes 243. Font selection commands (fnt_num0 through fnt4) must refer to fonts defined in the preamble.

Dimensions that appear in the DVI instructions are analogous to fix word quantities; i.e., they are integer multiples of 2**20 times the design size of the virtual font. For example, if the virtual font has design size 10 pt, the DVI command to move down 5 pt would be a down instruction with parameter 2**19. The virtual font itself might be used at a different size, say 12 pt; then that down instruction would move down 6 pt instead. Each dimension must be less than 2**24 in absolute value.

Device drivers processing VF files treat the sequences of dvi bytes as subroutines or macros, implicitly enclosing them with push and pop. Each subroutine begins with w = x = y = z = 0, and with current font f the number of the first-defined in the preamble (undefined if there's no such font). After the dvi commands have been performed, the h and v position registers of DVI format and the current font f are restored to their former values; then, if the subroutine has been invoked by a set char or set command, h is increased by the TFM width (properly scaled)—just as if a simple character had been typeset.

- long char = 242 { VF command for general character packet }
- set char 0 = 0 { DVI command to typeset character 0 and move right }
- set1 = 128 { typeset a character and move right }
- set rule = 132 { typeset a rule and move right }
- put1 = 133 { typeset a character }
- put rule = 137 { typeset a rule }
- nop = 138 { no operation }
- push = 141 { save the current positions }
- pop = 142 { restore previous positions }

- right1 = 143 { move right }
- w0 = 147 { move right by w }
- w1 = 148 { move right and set w }
- x0 = 152 { move right by x }
- x1 = 153 { move right and set x }
- down1 = 157 { move down }
- y0 = 161 { move down by y }
- y1 = 162 { move down and set y }
- z0 = 166 { move down by z }
- z1 = 167 { move down and set z }
- fnt num 0 = 171 { set current font to 0 }
- fnt1 = 235 { set current font }
- xxx1 = 239 { extension to DVI primitives }
- xxx4 = 242 { potentially long extension to DVI primitives }
- fnt def1 = 243 { define the meaning of a font number }
- pre = 247 { preamble }
- post = 248 { postamble beginning }
- improper DVI for VF 139, 140, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255

The character packets are followed by a trivial postamble, consisting of one or more bytes all equal to post (248). The total number of bytes in the file should be a multiple of 4.

Other relevant information could be found here:

Relevant Softwares

Table of Contents

TeX Engine

- LuaTeX - an extended version of pdfTeX using Lua as an embedded scripting language

  http://www.luatex.org

Scientific Plotting

- Matplotlib - Python 2D plotting library

  http://matplotlib.sourceforge.net
  cf. dviread.py

DVI to Image Converter

- Dvipng - A DVI-to-PNG converter

  http://savannah.nongnu.org/projects/dvipng

- MathTran - Translation of math content

  http://www.mathtran.org
  http://sourceforge.net/projects/mathtran

- jsMath - A Method of Including Mathematics in Web Pages

  http://www.math.union.edu/~dpvc/jsmath

- Texvc - MediaWiki TeX Converter

  http://www.mediawiki.org/wiki/Manual:Enable_TeX
  http://www.mediawiki.org/wiki/Texvc

DVI Viewer

- Okular - universal document viewer based on KPDF for KDE 4

  http://okular.kde.org/

- Evince - document viewer for multiple document formats

  http://projects.gnome.org/evince/?guid=ON

- xdvi - program for displaying DVI files on X-Window

  http://math.berkeley.edu/~vojta/xdvi.html
  http://xdvi.sourceforge.net

- Active-DVI - Unix-platform DVI previewer and a programmable presenter for slides written in LaTeX

  http://advi.inria.fr

DVI Tools

- Hacking DVI files: Birth of DVIasm

  http://www.ctan.org/tex-archive/dviware/dviasm
  http://www.tug.org/pracjourn/2007-1/cho/cho.pdf

TeX Fonts

- The Latin Modern (LM) Family of Fonts

  http://www.gust.org.pl/projects/e-foundry/latin-modern

- The TeX Gyre (TG) Collection of Fonts

  http://www.gust.org.pl/projects/e-foundry/tex-gyre

Font Viewer

- FontMatrix

  http://www.fontmatrix.net IS DEAD
  http://en.wikipedia.org/wiki/Fontmatrix
  http://www.ohloh.net/p/fontmatrix

- TFMPK - TEX's Fonts Viewer

  http://www.medini.org/software/tfmpk

Python Module Index

p

Index

Symbols

A

B

C

D

E

P

R

S

T

U

V

W

X

Z