Text Mode Layer (tml) - a Python module for HP Prime for simulating a fixed-width font terminal

version 1.00

by Piotr Kowalewski (komame), August-October 2024

What is tml and how does it work?

The ***tml*** (Text Mode Layer) module serves as a convenient replacement for the built-in terminal on the HP Prime for Python, while also offering additional features for managing the terminal.

The module uses graphics mode to simulate a terminal using fixed-width fonts. Since the built-in terminal is restored once the Python program finishes execution, it is not possible to use *tml* directly from the command line. Therefore, it must be imported into a program and can only function until the program terminates. For this reason, *tml* includes its own handling of text printing, screen scrolling, and user input mechanisms. The user input handling has been implemented in such a way that it behaves similarly to how it works on a PC, allowing data to be entered at any position on the screen, rather than being restricted to a dedicated input field at the bottom of the screen (which is a limitation of the built-in terminal on the HP Prime).

A status bar can be displayed at the bottom of the terminal (enabled by default), which can display any text message. The status bar also displays keyboard indicators (Shift/Alpha).

The *tml* module allows the use of various monospaced bitmap fonts (different styles and sizes), but only one can be used for a single *tml* instance. After loading a font, *tml* initializes a terminal with the number of rows and columns that the selected font size permits. The largest supported font width is 35 pixels, and the smallest is 4 pixels, allowing for a terminal with 9 to 80 characters per row, respectively.

It is possible to create custom fonts and define your own key mappings for specific characters (symbols), allowing for direct input of symbols from the keyboard, as well as their correct display, such as for diacritical characters. Refer to the 'Advanced features' section for details.

How to start with tml?

If you are using the built-in Python app or creating your own application based on the Python app, copy the *tml.py* file and one of the selected fonts (from the *Fonts* folder) to your application. Then, in the main file of your application (usually *main.py*), import the *tml* module, which will allow you to initialize the terminal.

It is also possible to use *tml* through the PPL wrapper, but with this approach, it is crucial that all PPL procedure calls are made from Python, not the other way around.

Terminal Initialization

The simplest way to initialize the terminal is to simply create an instance of *tml* without providing any arguments:

```
import tml
t = tml.tml()
```

This approach initializes the *tml* with the following default values:
- using the first font found (details below)
- status bar enabled, with no content ('')
- dark mode disabled

- tab size: 4 characters
- extended character mapping: empty (refer to ‘Advanced features’ section)
- symbol key mapping: empty (refer to ‘Advanced features’ section)
- font buffer: 9 (G9)

From the above, it can be seen that *tml* has 7 arguments, each with its own default value; therefore, it is recommended to reference them by name.

The full constructor is as follows:

```
import tml
t = tml.tml(font, status = '', dark_mode = False, tab_size = 4, ext_char_map = {},
    symb_key_map = {}, grob = 9)
```

Arguments:

- font:

During initialization, *tml* automatically searches for and loads a font file (a file with extension ‘.font’). However, it is also possible to explicitly specify the font to be loaded using the *font* parameter (when specifying the font file name explicitly, omit the '.font' extension.). If the font is not specified and the application contains more than one font file, the first one in alphabetical order will be loaded.

The *tml* module package includes several sample fonts in different styles and sizes:
- `atari8x8` (40 columns, 28 or 30 rows) - atari 8-bit style
- `std5x10` (64 columns, 26 or 24 rows) - standard font
- `std5x12d` (64 columns, 18 or 20 rows) - standard font with diacritical characters
- `med6x12` (53 columns, 18 or 20 rows) - medium font
- `med10x12d` (32 columns, 18 or 20 rows) - medium font with diacritical characters
- `mini4x7` (80 columns, 32 or 34 rows) - hp48/49/50 mini font style

Custom fonts can also be used, as described in ‘Advanced features’ section.

- status:

Accepts a text value to display in the status bar. By default, it is an empty value (''), which causes the status bar to appear but contain no initial content. Entering any text into this parameter will cause it to appear in the status bar, which is located at the bottom of the screen. When the status bar is visible, it occupies the last two rows of the terminal and is not scrollable. To disable the status bar and make the terminal fullscreen, set *status* parameter to `None` suring initialization. Note, however, that this will also hide the keyboard status indicators (Shift and Alpha key states).

- dark_mode:

By default, dark mode is disabled, meaning the terminal background is white and the characters displayed are dark. When dark mode is enabled, the background turns black and the characters are light. To enable dark mode, set the argument to `True`. This parameter can only be set during terminal initialization and cannot be changed later.

- tab_size:

The tab size allows the screen to be divided into vertical segments of a specified size, to which the cursor will move when a tab character (`\t`) is printed. This allows text alignment or easier column division.

- grob:

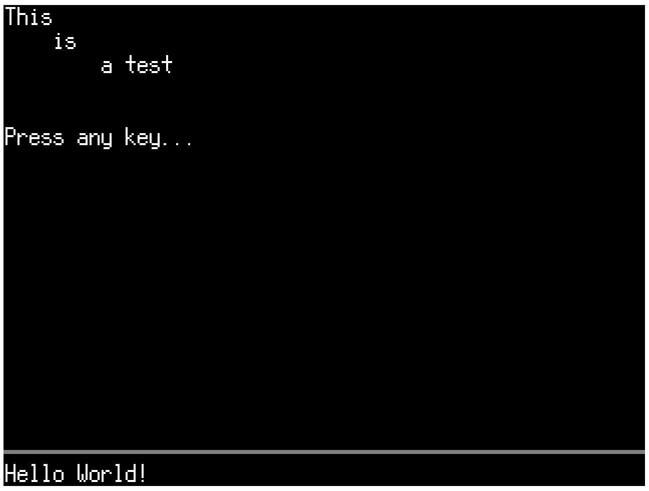
Since the font is loaded from a ‘.png’ file, it must be buffered in one of the HP Prime’s graphic objects (graphic variables). By default, this is 9 (G9), which can be specified during initialization using the

grob argument. Like *dark_mode*, the *grob* value can only be set during terminal initialization and cannot be changed later.
Example of initializing a terminal with 53 columns and 20 rows in dark mode, with the status bar set to `'Hello World!'` and displaying the text `'This\n\tis\n\t\ta test'` using *print*, which will interpret the newline and tab characters:

```
import tml

t = tml.tml(status='Hello World!', font='med6x12', dark_mode=True)
t.print('This\n\tis\n\t\ta test')
t.print('\n\nPress any key...')
t.read_key()
```

Obtained effect on the screen:



```
This
    is
        a test


Press any key...
```

```
Hello World!
```

Available methods

The *tml* module provides the following methods:

- print(*args)
Allows for output of text and other values to the terminal. It accepts arguments and works like the built-in *print* command in MicroPython on HP Prime, additionally interpreting the tab character (`\t`).

- clear()
Clears the terminal screen and places the cursor at position (0,0), which is the top left corner of the screen. It does not take any arguments.

- set_cursor(x, y)
Sets the cursor at position (x, y). The next use of *print* or *input* will start from this position.

- input(prompt, length, alpha, shift, new_line)
The *input* function allows data entry from the user, similar to the built-in *input*, but with some extensions that allow for additional actions during data input. When the edit field appears, the user can move the cursor left and right within the entered text, insert characters, and delete them using the Backspace or Del key. During data entry, modifier keys Shift and Alpha (and their lock states) can be used according to the standard behavior in the Home view. When any modifier key is activated, the

keyboard status is displayed in the status bar (if enabled), where 'SL' stands for Shift-Lock, 'AL' or 'al' indicates Alpha-Lock depending on case, and '^' indicates active Shift (for one-time use).

It is also possible to enter custom characters by holding the Symb key while pressing other keys. However, this requires defining additional key mappings, as mentioned in the 'Advanced features' section.

If Esc or Clear (Shift+Esc) is pressed during editing, the edit field will be cleared, and the cursor will return to the starting position.Pressing Enter ends input and returns the entered data as a string.

Note: Keyboard input may not work correctly on the Virtual Calculator when pressing letter or symbol keys. If you want to test the input method, use exclusively mapped keys (refer to the Computer keyboard mapping section in VC help for details) or use a physical HP Prime.

Input takes following arguments:

- **`prompt`:** Specifies the prompt that will appear on the screen in the data input field.
- **`length`:** By default, *input* creates an edit field that spans the entire line (from the cursor's x position to the right edge of the terminal). However, a maximum edit field length can be specified, preventing more characters than indicated by the *length* argument from being entered.
- **`alpha`:** Initially enables Alpha-Lock when the edit field appears, allowing text to be entered immediately without needing to press the Alpha key.
- **`shift`:** Initially enables Shift or Shift-Lock (if Alpha-Lock was previously activated).
- **`new_line`:** This argument specifies whether pressing Enter should result in moving to a new line. Setting *new_line* to False is useful when collecting data on the last line of the terminal, and you don't want the screen to scroll after pressing either of these keys.

- read_key(code=False)

Pauses the program and waits for any key to be pressed (except Alpha and Shift). By default, it returns the symbol of the pressed key, taking into account the current Alpha and Shift states. If *code* is set to `True`, it returns the key code of the pressed key (0-51).

- get_keys()

Returns a list of codes for currently pressed keys (including Alpha and Shift, as well as when multiple keys are pressed simultaneously). It returns an empty list when no key is pressed.

- set_status(text)

Allows setting the text displayed on the status bar. To clear the status bar, use an empty string: `set_status('')`

- print_xy(x, y, text)

Displays the specified *text* at position (x, y) of the terminal without moving the cursor. For this function, tab and newline characters are not interpreted.

Advanced features

This section is intended for advanced users and describes three *tml* features:

1. Creating custom font files.
2. Mapping graphic symbols from a font file to individual displayable characters.
3. Mapping keyboard keys for direct input of custom symbols.

You will also find information here on how *tml* works at a lower level.

Each *tml* font file is a ‘.png’ file, but slightly modified, as it must contain additional information about the font width and configuration number of the specific font. Internally, such a ‘.png’ file containing the appearance of individual font characters is treated as an array from 0 to 94, where 0 is a space and 94 is a tilde. All characters are arranged side by side in a single row with a fixed pixel width, so when displaying characters using the *print* method, each character can be mapped to the appropriate index and the corresponding section of the font file can be displayed to represent the appearance of the specific character.

By default, only characters within the basic ASCII table range can be displayed and entered in *tml*, i.e., from the space character (code 32) to the tilde (code 126). All characters in this range are handled by the built-in mapping, so there is no need to define additional mappings for either displaying these characters or key mappings for entering them. However, it is possible to use font files with a larger set of characters. This requires using two additional arguments when initializing *tml*: ***ext_char_map*** and ***symb_key_map***.

Creating custom fonts files

To create custom font, create a bitmap with dimensions of `x = 95 * character_width` and `y = character_height`. The bitmap should include at least ASCII characters from code 32 (space) to 126 (tilde) in a single row. The characters should be black on a white background. Grayscale shades are allowed, but other colors should be avoided, as they may cause artifacts during text editing on the screen or when dark mode is enabled.

If you want your font to include more than 95 characters (e.g., if you want to add diacritical marks or dedicated symbols needed in your program), you can add them in any order after the tilde character, meaning that the index of the first additional character will be 95 (as a reminder: the tilde is the 95th character, but since we start counting from zero, the tilde index is 94). Once all the characters are designed, save the bitmap file in ‘.png’ format (you can set the color depth to 1-bit if only black and white color were user).

For the file to be correctly recognized and imported by the *tml* module, a configuration byte must be added at the end of the file. This byte should include the configuration number and the width of a single character in pixels. The configuration number should be stored in the three least significant bits of this byte and should currently always be set to 0 (other configuration numbers are reserved for future use), while the width of a single character should be stored in the five most significant bits as a value decreased by 4. This allows for font widths ranging from 4 to 35 pixels.

For example, if the font file contains characters with a width of 9 pixels, the configuration byte should look like this in binary: 00101000, which is 40 in decimal. You can add this byte on a PC or directly on the HP Prime using the AFilesB command.

Once this byte is added and the file extension is changed to '.font', the font file is ready to be used in *tml*.

Mapping graphic symbols from a font file to individual displayable characters

If your font contains more than the standard 95 characters, you need to define a mapping of Unicode characters to the corresponding indices in your font for *tml* to know which characters to display for the additional symbols outside the standard ASCII range.

For example, let’s assume that the 95th character is the paragraph symbol (§), which is not part of the standard ASCII character set. If the print method encounters this character in the text to be printed, it needs to know its position within your font, i.e., it needs to know its index. To map the character to an index, you need to define a dictionary in the following form: `{ '§' : 95 }` and pass it as the *ext_char_map* argument.

For example, let's assume your font includes symbols for card suits:
♠ (Spade), ♡ (Heart), ♢ (Diamond), ♣ (Club), respectively from index 95 to 98.
In this case, the dictionary defined in *ext_char_map* should look like this:
`{ '♠' : 95, '♡' : 96, '♢' : 97, '♣' : 98 }`

If you have designed custom symbols that do not have corresponding Unicode characters, you can use any Unicode character that you do not need in your program and replace it with your symbol.
It is not necessary to define all the designed characters in the dictionary. However, if a character that has been omitted appears in the text provided to the *print* method, it simply will not be displayed on the screen.

Mapping keyboard keys for direct input of custom symbols

When a mapping for non-standard characters has been defined and they can be displayed on the terminal using *print*, there may sometimes be a need to enter them from the keyboard. In the Home view, to enter a language-specific character such as '**ä**', you can open the 'Chars' tool, locate the symbol, and insert it into the edit field. However, this tool is not accessible when running a program that utilizes the *tml* module. In such cases, the *input* method provides a way to enter certain characters defined in your font through key combinations: `[Symb]+[any_key]`. Similar to font-to-character mapping, it is necessary to define the key mappings and pass them as the *symb_key_map* argument.
This method also allows you to define multiple symbols under one key, which can be useful when dealing with a group of similar symbols, preventing the need to assign different keys for each. For example, in French, the letter 'e' has four diacritical variants: **é**, **è**, **ê**, **ë**. In such a case, it is best to place all four under the `[Symb]+[e]` combination, which will be intuitive and convenient when entering text containing any of these characters.
To input any of these characters into the edit field, press and hold the `[Symb]` key, then additionally press the `[e]` key. The first symbol will appear at the cursor position, but the cursor will not move to the next position until you release the `[Symb]` key. At this point, pressing the `[e]` key repeatedly will cycle through the characters defined under this key, and releasing the `[Symb]` key will confirm the currently displayed character.

Mapping keys to symbols involves using the key number (ranging from 0 to 51) as a key in the dictionary. The corresponding value should be a list of four elements, where each element is either a list of symbols or `None`. Each of these four elements represents a different keyboard state, as the displayed symbols may vary depending on the activation of the Shift, Alpha key, or a combination of both. Thus, the value for each key in the dictionary must be a list of four cases in the following order:

1. when no modifier is active,
2. when only Alpha is active,
3. when only Shift is active,
4. when both are active.

For example, for the key with the letter 'e' and the French diacritical characters, you could create the following dictionary:
`{ 18: [None, ['è','é','ê','ë'], None, ['È','É','Ê','Ë']] }`

This should be understood as follows: when neither Alpha nor Shift is active, pressing `[Symb]+[e]` will not display any symbol (since Alpha is not active, there is no reason for any letters to appear). However, if the Alpha modifier is enabled, pressing `[Symb]+[e]` will display the character '**è**' (lowercase), and each subsequent press of `[e]` will cycle through the symbols in the list until the `[Symb]` key is released. The third value is also `None`, as we do not want to show letter symbols when only Shift is active (this might be a good place to display some unique non-alphanumeric symbols instead). The fourth value indicates what should appear when both Alpha and Shift are active, so in this case, the symbols **È**, **É**, **Ê**, **Ë** (uppercase) will be displayed.

Known issues and limitations

- At this moment, it is not possible to restore the content of the terminal screen after it has been overwritten by external code. This also means that if two parallel instances of *tml* are created, it is not possible to switch between them.
- There is no support for displaying text and background colors in different colors (this is due to the limitations of the *hpprime* library).
- It is also not possible to read the character code from the *tml* screen at specific coordinates.

If you find any bugs, have an interesting suggestion, or simply want to contact me, send a private message to the user **komame** on www.hpmuseum.org/forum/.