PIPSTA106 – QR-CODE PRINT PYTHON CODE TUTORIAL

Contents

Revision History

Revision	Author	Date	Description
1.0	AH	25/11/14	First Release

Difficulty Level:

- This tutorial assumes the user has some Python programming experience
- This tutorial builds on USB configuration code discussed in ***PIPSTA104 – Basic Print Python Code Tutorial***
- This code uses additional, sophisticated imaging libraries
- This code requires a level of programming competence to understand the conversion of a bitmap image into a bit array and subsequent packaging into bespoke printer commands.

Time to Complete:

- Some aspects are complex and require significant time to explain and comprehend.

Who Should Read This Document

This tutorial is beneficial to those who have completed ***PIPSTA104 – Basic Print Python Code Tutorial*** and wish to develop their understanding of the simple text application into something that produces more advanced prints, using the powerful and freely-available Python libraries.

This tutorial covers in detail the operation of the QR-Code demo script from ***PIPSTA006 – Pipsta QR Codes*** for those wishing to understand its operation or adapt the code for their own purposes.

Many aspects are identical to those presented in ***PIPSTA105 – Banner Print Python Code Tutorial***, so it is helpful, though not necessary to have read this tutorial first.

How it Works

The script uses the same methods of establishing a USB connection, and argument-parsing techniques as seen in ***PIPSTA104 – Basic Print Python Code Tutorial***. For brevity, these aspects will not be covered again in this tutorial. Those wishing to understand such aspects should read the ***PIPSTA104 – Basic Print Python Code Tutorial***.

The script takes a single *command-line argument*, though –in a departure from the methods presented in **BasicPrint.py** and **banner.py**—this time the text to encode is held in a file whose

filename (and path) are passed as the command line argument. In practice, this means that this will permit many lines of text to be encoded.

In overview, the following aspects are common between **banner.py and qr.py**:

- *ParseArguments()* and *SetupUSB()* work in much the same way as outlined in **banner.py**
- Both scripts make use of *Pillow* and *bitarray* libraries

As the conversion process from a bitmap to an array of bits suitable for transmission to the printer is effectively identical (albeit with no rotation required for **qr.py**):

- *validate_image()* works in much the same way as *check_image()* in **banner.py**
- *convert_image_to_printer_format()* works in the same way as *convert_image()* in **banner.py**
- *print_image()* also works in the same way as in **banner.py**

- Novel aspects of this script are:
  - The library *qrcode* is used to make a bitmap image of the data to be encoded. In its simplest implementation, this is simply a case of calling *qrcode.make(data)*
  - A *wrapper* function, *send_to_printer(data)* has been placed around the image conversion functions and USB communcations functions.

In the main() function itself:

- **main():**
  - There is a check to ensure that Python2.x is being used. This is a requirement of the *bitarray* library
  - There is a check to ensure that the script is running on a Linux system
  - A function is called to look for a device of the appropriate *VID* and *PID*
  - *parse_arguments()* is called and to get the text data (provided as an argument containing the path and filename of text to be encoded, or from the default file *qr.txt* which appears in the same folder as the Python script itself.)
  - For diagnostic purposes, a logging provision is started (the log for which is stored locally on the Raspberry Pi)

Then, in the *send_to_printer()* function:

- **send_to_printer()**
  - USB is set-up
  - A *Try-Except-Finally* block is established to handle exceptions; specifically data overflow errors as thrown by the qrcode encoding process. Note that the *finally* block will be executed regardless of whether execution happened without a problem, or whether an exception was handled.
  - A command is send to the printer to flash the LED green for the duration of the encoding and printing
  - The QR Code image is made and subsequently scaled to give the best fit over the 384 dot width the printer is capable of. Note that –in addition to the black and white area— the image will also be surrounded by a 'quiet zone'. This has been retained in this script such that the QR Code could still be scanned even if the paper/label was subsequently stuck to –say—a black surface.

- The image is checked to ensure it is acceptable in terms of being multiples of 24 x 8 dots
- The image is converted to a bitarray and issued as a series of bulk transfer graphics commands to the printer.
- 5 carriage returns are sent to feed the paper past the tearbar
- The green LED flashing mode is turned off
- *send_to_printer()* returns
- The script exits

Altering the Script

Unlike with the banner script, the image produced is comparatively small and –as such—processing and printing delays are not too onerous.

Simple adjustments would include:

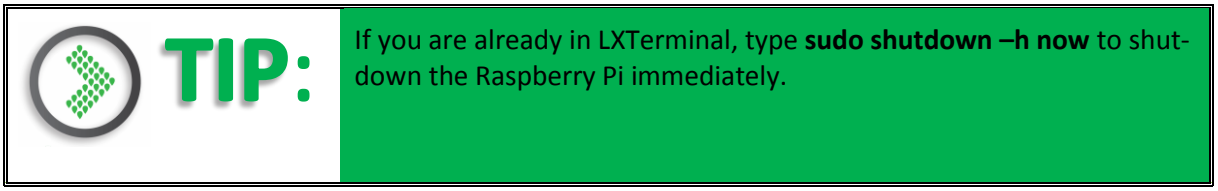
- Encoding web-addresses to take users directly to specific pages. By producing QR Codes on Linerless Label media, existing posters and displays can be significantly enhanced

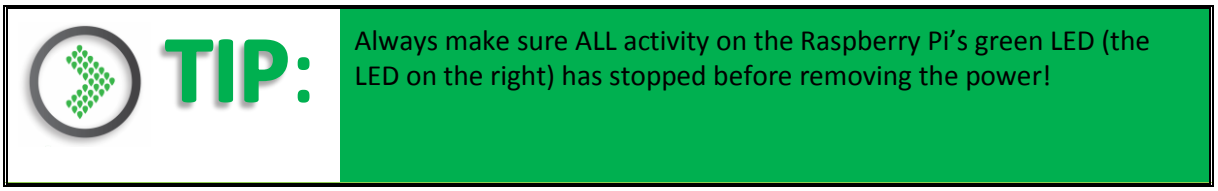
'Speculative' adjustments would include:

- Adjusting the error correction settings of the QR Code and *intentionally* introducing a logo to the centre of the image (using Pillow to apply the logo over the top of the QR Code). Error correction ought to then still deliver a readable QR Code, but one that is more visually appealing and interesting than a default code.

Shutting Pipsta Down Safely

Whilst the printer is resilient when it comes to powering down, the Raspberry Pi must undergo a strict shutdown process to avoid corrupting the Micro SD card. The most straightforward method of doing this is to double-click the 'Shutdown' icon on the desktop.





■End of Document■