How to print QR codes on a thermal printer
Last updated December 04, 2016

Table of Contents
1. Introduction
2. Wiring
3. Basic Testing
4. Writing C code
5. Library Setup
6. Library Usage
7. Bitmap Printing
8. Troubleshooting

Introduction:
This guide will cover the usage of the Adafruit mini thermal printer and BeagleBone Green (BBG) for printing QR codes. The printer may be purchased from Adafruit here https://www.adafruit.com/product/597 as well as other retailers. (https://www.sparkfun.com/products/10438 or https://www.seeedstudio.com/Embedded-Thermal-Printer-p-1621.html) Other accessories needed include a power supply and thermal printer. The power supply must be 5V-9V and capable of handling peak currents of 2.0 amps. The thermal paper must be 57mm wide. (This paper can be found at Staples) The cross-compiling and use of the library QRencode will also be covered in this guide.

Wiring:
• The printer has two connectors on the bottom with 3 pins each. The pinout is as follows

1: GND 2: NOTHING 3: 5V 4: GND 5: RX 6: TX
• You may connect the printer to any of the BBG’s UART ports which are found via the following pinout.

<table>
<thead>
<tr>
<th>UART</th>
<th>P9</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DGND</td>
<td>DGND</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>VDD_3.3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>VDD_5V</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>SYS_5V</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>PWR_BUT</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>UART4_RXD</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>UART4_TXD</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>GPIO_4</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>GPIO_5</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>UART1_RTSN</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>UART2_TXD</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>GPIO_49</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>GPIO_117</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>GPIO_115</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>GPO_121</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>GPO_120</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>AIN4</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>AIN6</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>AIN2</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>AIN0</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>GPO_20</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>GPO_22</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>GPO_15</td>
<td>45</td>
<td>46</td>
</tr>
</tbody>
</table>

• Simply connect the RX pin of the printer to the TX pin of the UART port you have chosen to use. This guide assumes the use of UART5 and thus the RX pin is connected to pin 37 of the BBG. However, you can still easily follow the guide still by using the port number you have chosen anywhere you see a 5. The GND pin (pin 4) of the serial connector can be connected any of the DGND pins on the Beaglebone.

**WARNING!**

Do not connect the TX pin of the printer to the BBG! The printer sends it data at 5V whereas the BBG is a 3.3V device. While the printer has no problem receiving 3.3V signals, the 5V from the printer will damage the Beaglebone. The only functions that transmit data from the printer are status functions for checking things such as “Is the device out of paper?”. If this functionality is needed, you can use a level shifter ([https://www.adafruit.com/product/735](https://www.adafruit.com/product/735)) to interface the signal. This guide will not cover how to wire the level shifter.

**Basic Testing:**

• To print the printers test page with some basic info including the printers firmware and baud rate, simply hold the button while powering on the printer.

• First we need to enable the device tree for the UART port. This can be done via the command

  `#echo BB-UART5 > /sys/devices/platform/bone_capemgr/slots`

• Then we need to set the baud rate to match the printer. This will most likely be 19200, however you can double check this by printing the test page as mentioned above.

  `#stty -F /dev/ttyO5 19200` (Troubleshooting, that is the capital letter O, not a zero)

• These steps will need to be done on reboot everytime and must be done in order for the C code provided to function. If one desires it is possible to set the baud rate of the port through C code using termios ([http://pubs.opengroup.org/onlinepubs/7908799/xsh/termios.h.html](http://pubs.opengroup.org/onlinepubs/7908799/xsh/termios.h.html)) but this will not be covered in this guide.

• We can now print simple strings to the printer by writing to the device file.
Writing C code:

- We need to make sure the file does not open as a controlled tty terminal with the flag `O_NOCTTY`, otherwise the OS will send a bunch of commands to the terminal and confuse the printer.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int fd;
fd = open("/dev/ttyO5", O_WR | O_NOCTTY | O_NDELAY);
char *output = "Like no one ever was";
write(fd, output, strlen(output));
```

Library Setup:

- If you are compiling your code on the Beaglebone directly you can simply use `apt-get` to install the library for you.
  ```
  sudo apt-get install libqrencode-dev
  ```
- Then you need only to include `<qrencode.h>` to use the library.
- To cross compile from a host computer, we will need to compile the library ourselves. First download the source ([https://github.com/fukuchi/libqrencode](https://github.com/fukuchi/libqrencode)) then in the terminal navigate to the main directory of the downloaded files and run the following commands.
  ```
  #autogen.sh
  ./configure --without-tools --without-png --enable-thread-safety
  --libdir=DIR1 --includedir=DIR2 CC=arm-linux-gnueabihf-gcc
  --host=arm-linux-gnueabihf
  #make
  #make install
  ```
- This will output qrencode.h to the directory specified by `DIR2` and some shared object files (.so) to `DIR1`. Now we need to add some flags when we compile our own code.
  ```
  -I DIR2 -L DIR1
  ```
- The directories need not be the same as before, so long as the specified directories contain the appropriate .h (for the -I) and .so (for the -L) files.
**Library Usage:**

- **length** = The number of bytes in the data to encode
- **input** = The data to be encode as a QR code, typically a char array, but you can use any single byte datatype.

 /*I have included a cast to unsigned char* as often the data given will be in char* form but the function expects unsigned char*, your mileage may vary. As we have used the encode data function of the library, we are able to have bytes of any value, the library also includes an encodeString function, but this does not allow all values, in particular the input must have no null characters (except the end) */

```c
QRcode *encoded = QRcode_encodeData(length, (unsigned char*)input, 0, QR_ECLEVEL_L);
```

/*A qr code can have 4 levels of error correction, this code uses the lowest level, you can change it to higher levels by changing the last letter in QR_ECLEVEL_L to L (lowest) M, Q, or H (highest) A qr code is always square, so columns = width = rows
Note, the printer has a maximum width of 384 pixels, the library automatically generates the smallest qr code size that will fit the provided data and error correction level, if this value is greater than 384, you will need to split your data into segments */

```c
int rows = encoded->width;
int numBytes = rows * (rows + 7)/8
pixels1bit = malloc(numBytes);
memset(pixels1bit, 0, numBytes);
```

/*The printer needs 1 bit bitmaps, where each bit corresponds to a single pixel that is either black or white. The library uses a similar format, but each byte corresponds to one pixel, with the least significant bit being the image data, and the other bits being padding or extra information. This function packs the given data down into a format the printer can understand */

```c
make1bitBitmap(pixels1bit, encoded->data, rows, rows);
QRcode_free(encoded);
//You may now print the bitmap contained within pixels1bit.
```

```c
void make1bitBitmap(char* output, char *input, int width, int height)
{
    int whatByte = 0;
    int whatBit = 0;

    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {
            /*Check the least significant bit, if it is 1, then the pixel is black */
            if((input[i * width + j] & 1) != 0) {
                output[whatByte] += (1 << (7-whatBit));
                whatBit++;
            }
            if(whatBit % 8 == 0) {
                whatBit = 0;
                whatByte++;
            }
        }
        //Account for widths that may not be multiples of 8
        if(whatBit % 8 != 0) {
            whatByte++;
        }
        whatBit = 0;
    }
}
```
**Bitmap Printing:**

bitmap = An array of chars, with each bit in each char corresponding to a black or white pixel in the image
rows = The number of rows in the bitmap (equivalently the height)
columns = The number of bytes in one row, if given the bitmap width in pixels, you can get the width in bytes by using (width + 7)/8 to round to the next highest multiple of eight and then divide by 8.

```c
for (int i = 0; i < rows; i++) {
    //Tell the printer we are starting a new row of the bitmap
    write(fd, "\x12", 1);
    write(fd, "*", 1);
    /*We’ll write the bitmap 1 row at a time, so this tells the printer we are
     *printing a one row bitmap */
    write(fd, "\x1", 1);
    /*Always padding up to maximum width, so our bitmap width is always 0x30 or
     *48*/
    write(fd, "\x30", 1);

    /*We always use 48 here, the maximum width of the printer is 384 pixels and
     *there are 8 pixels in a byte, meaning 48 rows. If the width is less than 48
     *the printer sometimes works, and other times does weeeird things. Padding
     *with whitespace is an easy and reliable solution */
    for (int k = 0; k < 48; k++) {
        if (k < columns) {
            write(fd, &bitmap[k], 1);
        } else {
            //Add Whitespace
            write(fd, "\x00", 1);
        }
    }
    /*The printer only has a small buffer, if you tried to send all the data at
     *once for even a small image, it would likely fill the buffer and start
     *overwriting other parts of data. This delay needs to be long enough such that
     *the printer can finish printing the row before more data is sent. I have
     *always just used 50ms as an overestimate, but this may cause slower than
     *usual printing, feel free to optimize this value by trial and error */
    nanosleep(&pollingDelay, (struct timespec *) NULL);
}
```
**Troubleshooting:**

1. The printer spits out seemingly random stuff, even when echoing to the dev file
   - Did you remember to set the baud rate for the device? If you consistently forget and waste time before remembering, it might be worth your time looking into termios to set the baud rate from code!
   - Perhaps the baud rate of your model printer isn’t 19200, print a test page and double check.

2. The printer prints part of a barcode, then random nonsense.
   - You’ve likely overrun the printer’s data buffer. Use a longer delay between rows.

3. A pink line appears on the side of the paper
   - The spool of thermal paper is running low, replace it soon.

4. The printer does nothing, the light doesn’t even blink
   - Make sure you’ve connected your power supply polarity correctly, lucky you if it’s wrong, the printer will not burn out, but it also won’t function.

**Useful References:**

[https://github.com/adafruit/Adafruit-Thermal-Printer-Library](https://github.com/adafruit/Adafruit-Thermal-Printer-Library)