How to print QR codes on a thermal printer

Last updated December 04, 2016

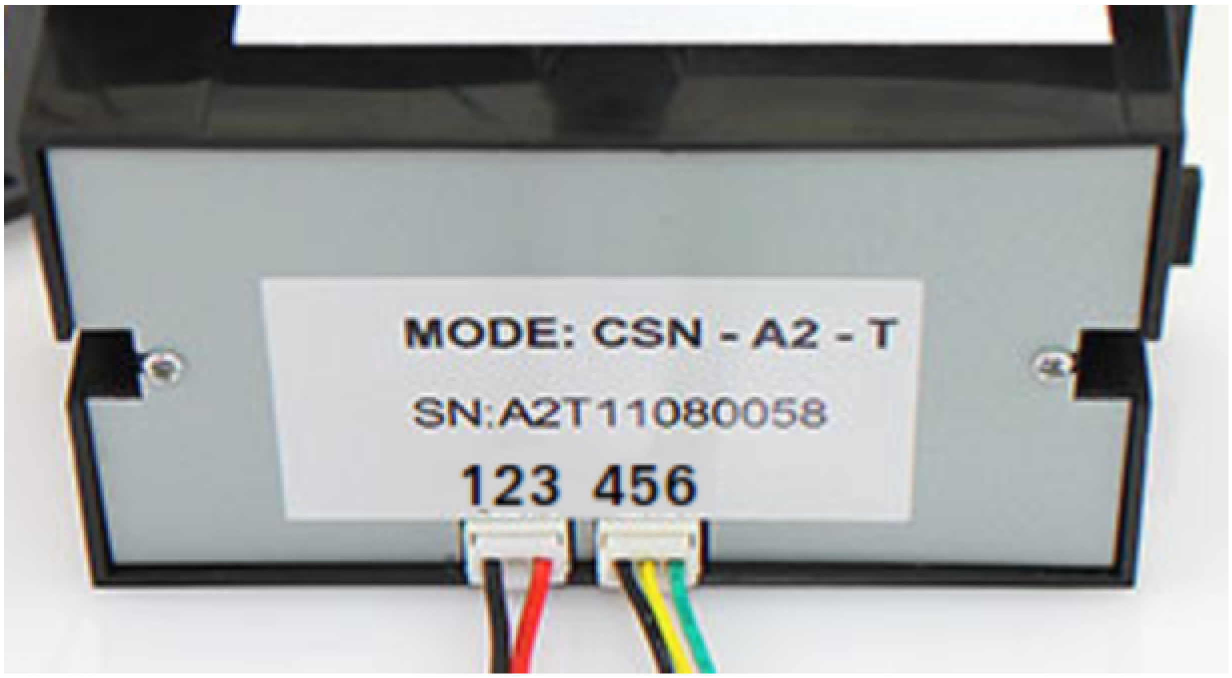
Table of Contents

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.

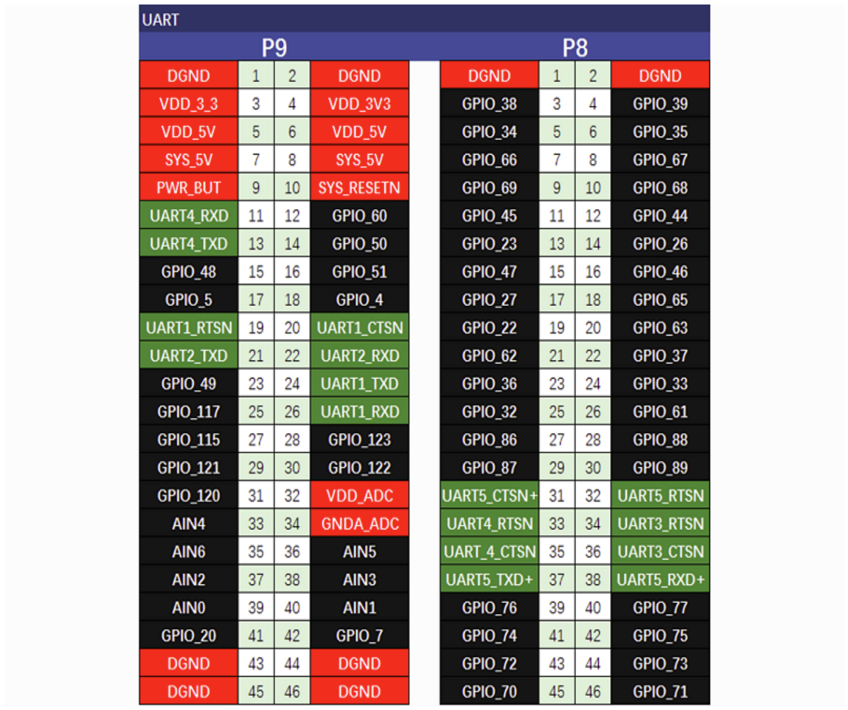
Wiring:

- 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.



UART

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3_3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
UART4_RXD	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
UART4_TXD	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
UART1_RTSN	19	20	UART1_CTSN	GPIO_22	19	20	GPIO_63
UART2_TXD	21	22	UART2_RXD	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	UART1_TXD	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	UART1_RXD	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_123	GPIO_86	27	28	GPIO_88
GPIO_121	29	30	GPIO_122	GPIO_87	29	30	GPIO_89
GPIO_120	31	32	VDD_ADC	UART5_CTSN+	31	32	UART5_RTSN
AIN4	33	34	GNDA_ADC	UART4_RTSN	33	34	UART3_RTSN
AIN6	35	36	AIN5	UART_4_CTSN	35	36	UART3_CTSN
AIN2	37	38	AIN3	UART5_TXD+	37	38	UART5_RXD+
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

- 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) 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) but this will not be covered in this guide.
- We can now print simple strings to the printer by writing to the device file.

```
#echo I want to be the very best! > /dev/tty05
```

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

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

int fd;
fd = open("/dev/tty05", O_WR | O_NOCTTY | O_NDELAY);
char *output = "Like no one ever was";
write(fd, output, strlen(output));
```

Library Setup:

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

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 bitmaps 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.

```
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 printers 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://cdn-shop.adafruit.com/datasheets/A2-user+manual.pdf
https://github.com/adafruit/Adafruit-Thermal-Printer-Library
http://fukuchi.org/works/qrencode/manual/index.html