Introduction to the GPIO System

To qualify as a secure `Thing` within the `IoT`[1], a device must be accessible by an IP address, and preferably encrypted to keep it private. The `RasPi`[2] is a suitable module for this purpose, considering it has `ssh`[3] access and interfaces with simple hardware through its `GPIO`[4] pins. This tutorial describes in detail how you can program the `GPIO` pins, using a simple `bash`[5] script. I consider other ways of doing this – with `C` or `python` – as too complex for beginners.

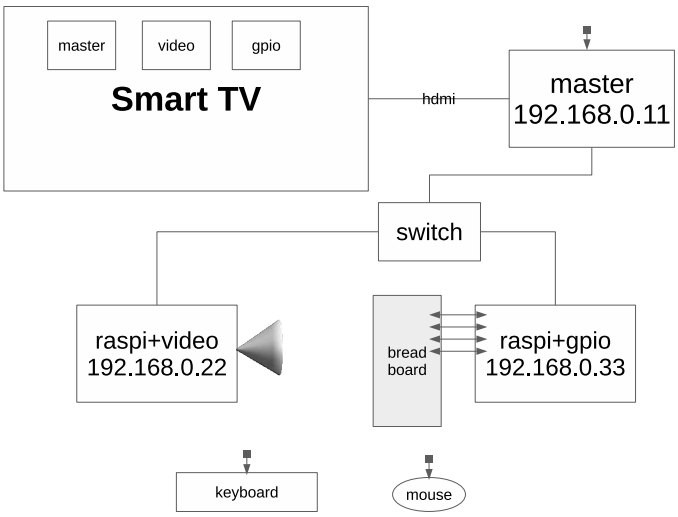
Setting up the Demonstration

Referring to the diagram below, in this talk I shall control everything from an `Intel NUC`[6], using a large smart TV as its video monitor, with a wireless mouse and wireless keyboard.

The three computers are connected by ethernet cable *via* a switch. The master computer boots to IP address `192.168.0.11`, the `RasPi` with the camera module boots to IP address `192.168.0.22` and the `RasPi` with the breadboard attached boots to the IP address `192.168.0.33`.

I'll use `nc`[7] to stream the `RasPi` camera module's view of a breadboard[8] (containing some `LED`[9] circuits) to the large TV so the audience can see the result of programming the pins and hence the `LEDs`. See our talk at `http://linuxlsga.net/stream.pdf`.

Note that I disconnect from the internet. Over the internet you'll have to configure a firewall to let `nc` through, but I cannot deal with internet or firewalls here. While I work in my `LAN`[10], I'll need to disable any firewall that might stop `nc` connecting, thus: `sudo /sbin/iptables -F`.



[1] internet of things
[2] raspberry pi nano computer
[3] the secure shell program
[4] general purpose input/output
[5] bourne-again shell program
[6] next unit of computing
[7] the netcat program
[8] a solderless lashup device for experimenting with circuits
[9] light-emitting diode
[10] local area network

Streaming the Video of the Breadboard

Position the `RasPi` so its camera module can film the breadboard. First, from one terminal emulator on `master`, start listening on port 12345 for video to display:

```
master~$ nc -l -p 12345 | /usr/bin/mpv --no-correct-pts --fps=90 -
Playing: -
[file] Reading from stdin...
```

Then, from another terminal emulator on `master`, log in to `video` and start the camera:

```
master~$ ssh 192.168.0.22
video~$ raspivid -vf -hf -n -w 1024 -h 768 -t 0 -fps 20 -o - | nc 192.168.0.11 12345
```

Return to the `master` desktop on the TV, and continuous video of what is happening on the breadboard shall be playing on `master` with a latency of about 250 milliseconds.

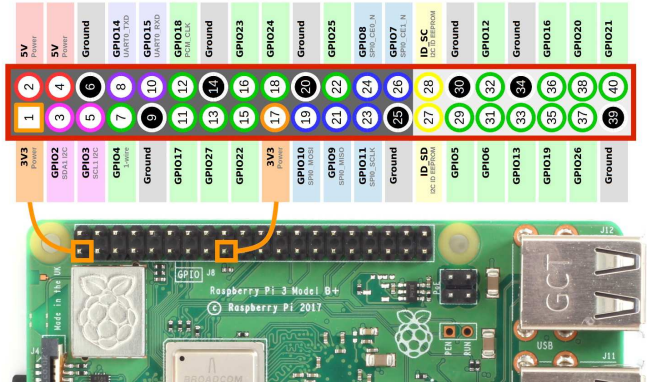
OK, Let's Control those GPIO Pins

My purpose here is to ensure that you can control a pin by setting it to a logical '0' (low voltage) or a logical '1' (high voltage). If the pin is connected to a suitable LED circuit, this will turn off or turn on that LED. If you can do this then you can advance to real projects. The relevant documentation is in the file `sysfs.txt` in the computer kernel source code.

The diagram below shows that for the `RasPi` I have chosen, there are 40 physical pins. The strange thing you'll notice is that the pin names have nothing to do with the pin numbers. Just get used to it. The manufacturer of the computers (Broadcom Corp) has assigned these.

To experiment with and explore these pins, first connect a jumper lead from pin 1 on the Pi (3.3Volts) to the longer leg of a small red LED in the breadboard, then the shorter leg connected in serial with a 330 Ohm resistor to GND, and connect a lead from this end of the resistor to pin 6 (GND) on the Pi. Stand back. When the `RasPi` is turned on and boots up, this red LED should light up, proving that you have got things right thus far !

Next, turn off the Pi, for safety. On the breadboard connect a small green LED through another 330 Ohm resistor to GND, and connect a jumper lead from pin 11 (which is labelled as GPIO17) to the positive (longer leg) of that LED, and connect a jumper from pin 20 (GND) on the Pi, to the GND end of the resistor on the breadboard. We shall turn this green LED on and off at will.



Log in from `master` to the `raspi`. You will need to perform all these commands as root, since you are now dealing with the computer hardware. So let's become root and explore the kernel structure that allows us to come to grips with pin 11 (gpio17).

```
raspi~$ sudo su
# cd /sys/class/gpio
# /bin/ls -F
export gpiochip0@  gpiochip504@  unexport
```

So far there is no apparatus for manipulating pin 11. You create that structure by writing `17` into the export attribute.

```
# echo "17">export
# /bin/ls -F
export gpio17@  gpiochip0@  gpiochip504@  unexport
```

Notice that a new link to a directory for gpio17 has been created. Let's go there.

```
# cd gpio17
# /bin/ls
active_low  device  direction  edge  power  subsystem  uevent  value
```

You will be interested at this stage in `direction` and `value`. The default direction is `in` (meaning you can read the value on the pin) and the default value is `0`, so that nothing on the breadboard gets damaged too much on bootup. You can now control these attributes of pin 11. Let's check the defaults first.

```
# cat direction
in
# cat value
0
```

Now let's take the crucial step: make the direction `out`, so you can write to the pin, and write the value `1` to it, to light the green LED.

```
# echo "out">direction
# cat direction
out
# echo "1">value
# cat value
1
```

This turns the LED on; if it does not ... then back to the drawing board !! But if it does ... well, then — Congratulations !

Now build on this successful experiment by putting all this together. Copy the following more formal `blinking-led` program into a text file (for example, using the program `nano` on the RasPi), name it `blink`, make it executable by executing `chmod +x blink` and execute it by `sudo ./blink`. The green LED should turn on and off every second until you hit CTRL+C, at which time the green LED turns off and the program exits.

```
#!/bin/bash
# program blink
# usage: $ sudo ./blink

# explicit led control from pin gpio17 (physical pin 11)

function error {
    echo "ERROR: $1"
    exit 1
}

[ $(whoami) == root ] || error "need to run as root"

# define pin functions and connections to breadboard
# gpio17 is physical pin 11

function create_pin {
    if [ ! -d /sys/class/gpio/gpio17 ]
    then
        echo "17">/sys/class/gpio/export
    fi
}

function remove_pin {
    if [ -d /sys/class/gpio/gpio17 ]
    then
        echo "17">/sys/class/gpio/unexport
    fi
}

function set_pin_high {
    echo "out" > /sys/class/gpio/gpio17/direction
    echo "1" > /sys/class/gpio/gpio17/value
}

function set_pin_low {
    echo "out" > /sys/class/gpio/gpio17/direction
    echo "0" > /sys/class/gpio/gpio17/value
}

function led {
case "$1" in
    on) set_pin_high ;;
    off) set_pin_low ;;
    *) error "can set pin high/low only" ;;
esac
}
```

```
function cleanup {
    echo "got CTRL+C; turn led off"
    led off
    remove_pin
    exit 0
}

trap 'cleanup' INT

create_pin

while :
do
    led on
    sleep 1
    led off
    sleep 1
done

exit 0
```

Warnings

Remember to GROUND yourself on a kitchen sink before touching your `RasPi`; it is not well protected from discharges of static electricity.

Conclusion

We have tried out two projects dealing with the RasPi hardware.

We can use the PiCam to stream continuous video of a scene back to a master computer.

We can control turning on and off lights from the GPIO pins. Although there is no doubt that the methods for programming the pins is simply weird when compared with my experience of programming the `Intel 8080` and the `MicroChip Technology PIC Series`, when you actually try them out they begin to appear workable.

The methods are tried on a secure local network, not over the internet, in order to illustrate the principles and practicalities involved.
