Cyber-Physical Systems

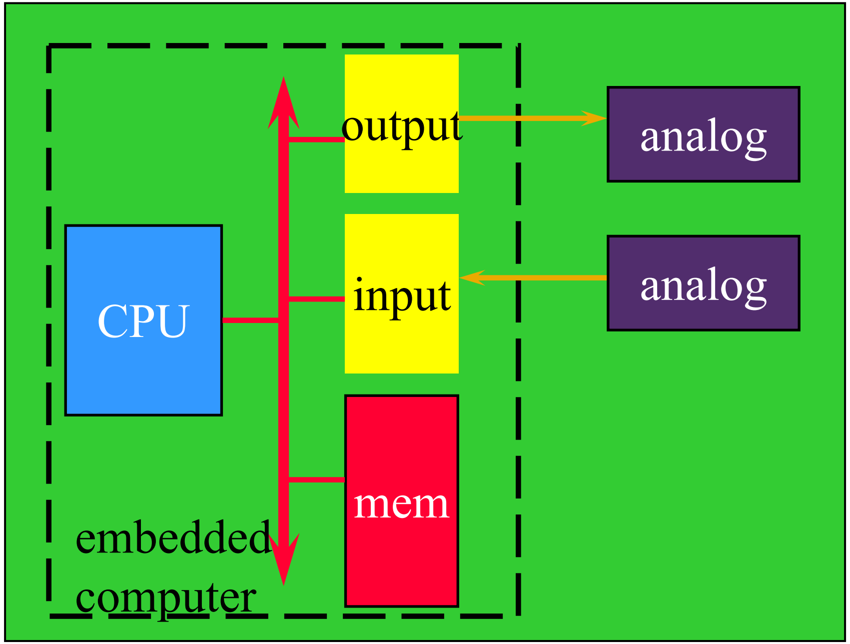
Basic I/O with RPi

UNIVERSITY
AT ALBANY
State University of New York

IECE 553/453, ICSI 553 – Fall 2022
Prof. Dola Saha

Embedded System

- Embedded computing system: any device that includes a processing system but is NOT a general-purpose computer.
- Often application specific: takes advantage of application characteristics to optimize the design
- Might have real-time requirements
- Might be power constrained



Connecting Analog and Digital Worlds

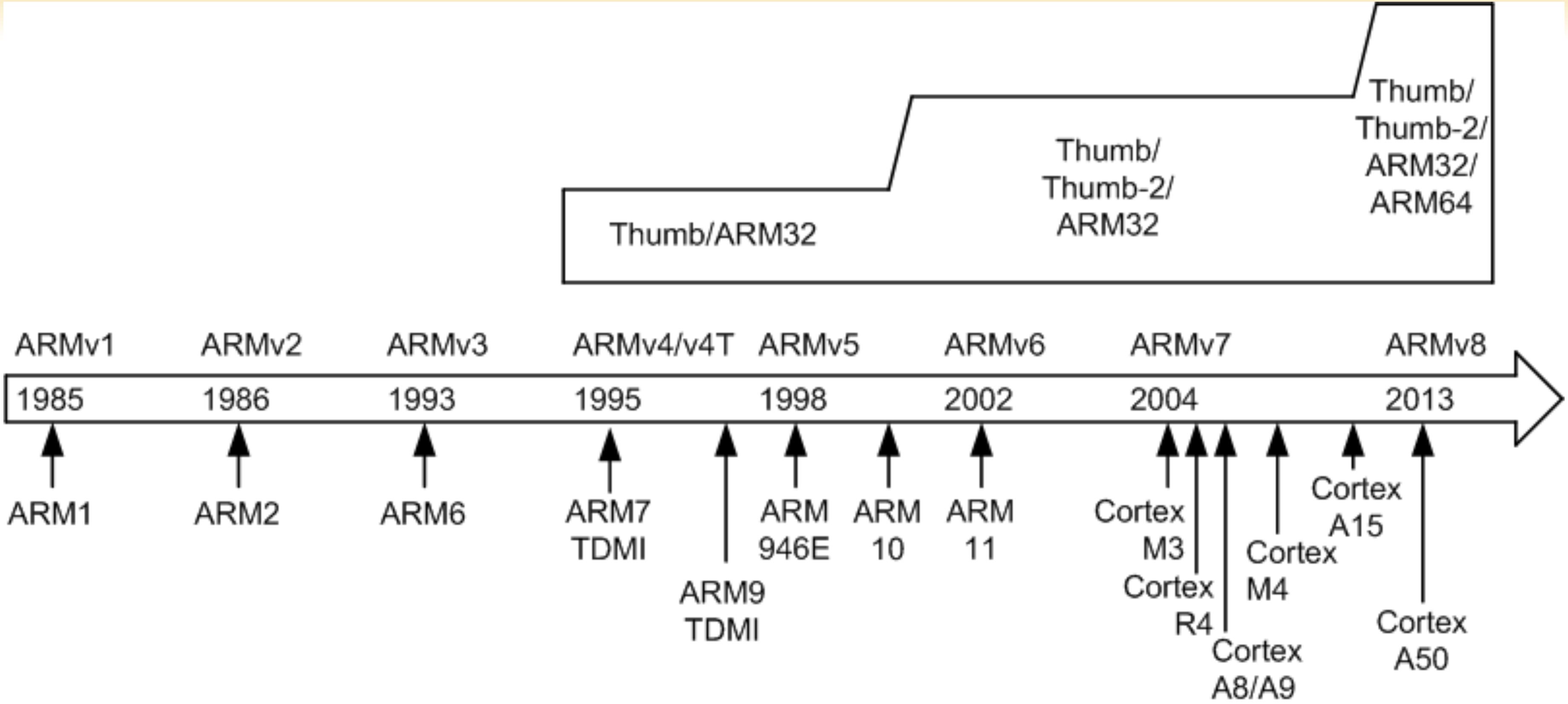
- Cyber
  - Digital
  - Discrete in Time
  - Sequential

- Physical
  - Continuum
  - Continuous in time
  - Concurrent

Practical Issues

- Analog vs. digital
- Wired vs. wireless
- Serial vs. parallel
- Sampled or event triggered
- Bit rates
- Access control, security, authentication
- Physical connectors
- Electrical requirements (voltages and currents)

History of ARM Processor



UNIVERSITY AT ALBANY
State University of New York

ARM Cortex Processors

ARM Cortex-A family:

Applications processors

Support OS and high-performance applications, such as smartphones, Smart TV

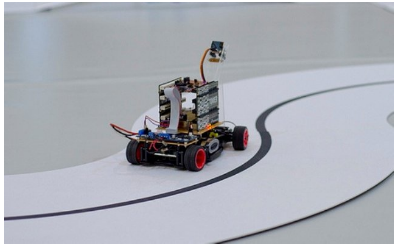
ARM Cortex-R family:

Real-time processors with high performance and high reliability

Support real-time processing and mission-critical control

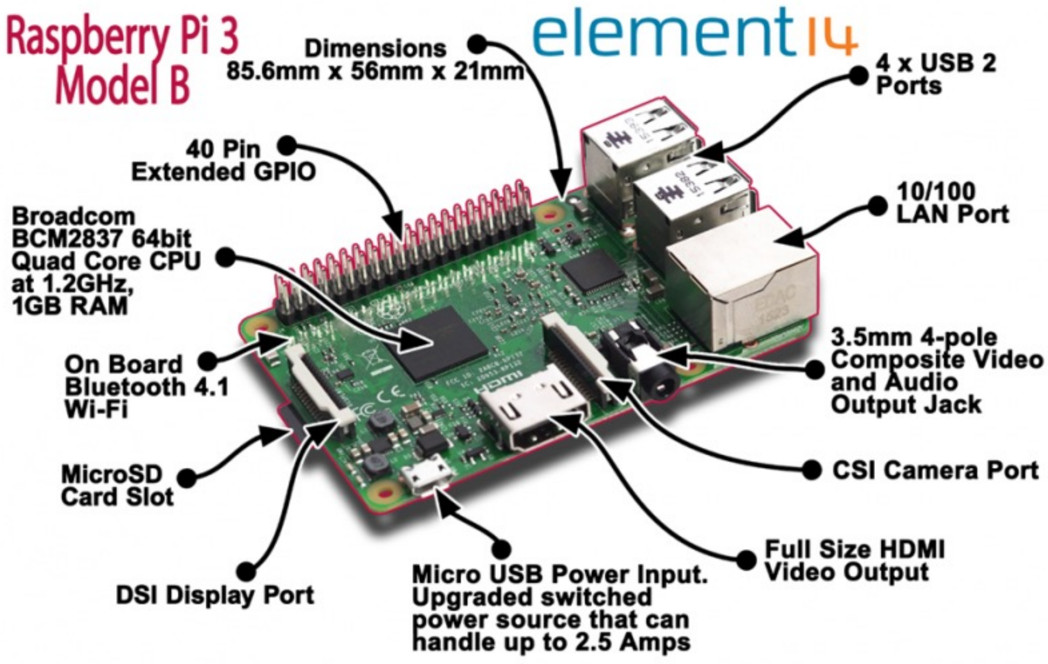
ARM Cortex-M family:

Microcontroller, energy-efficient, cost-sensitive, support SoC



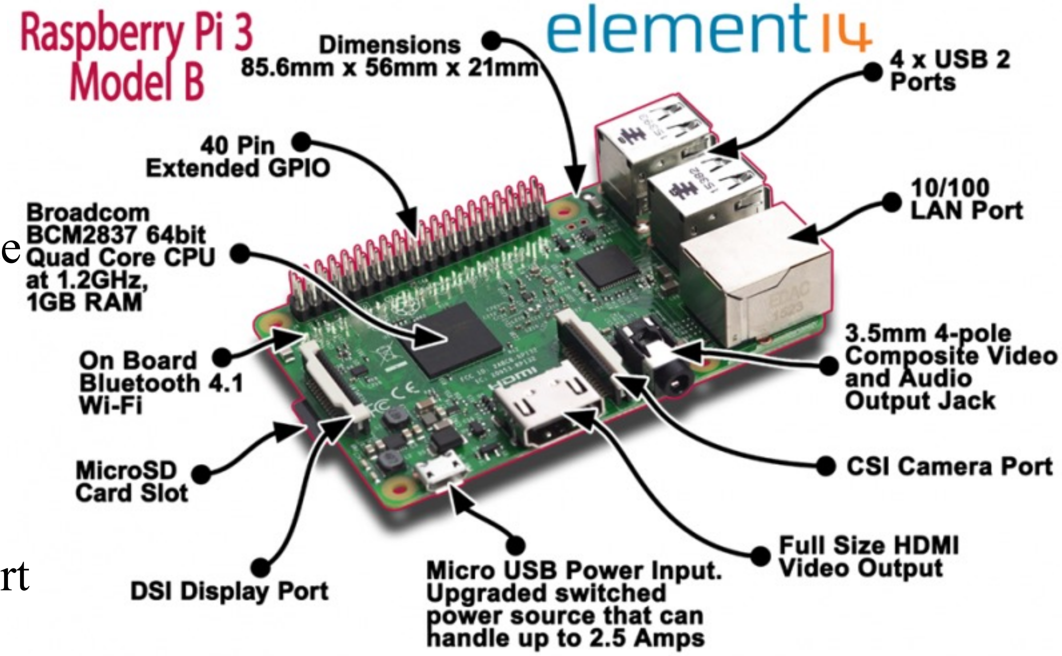
Raspberry Pi – Know your board

- The Raspberry Pi 3 Model B+
  - Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
  - 1GB LPDDR2 SDRAM
  - 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
  - Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)
  - Extended 40-pin GPIO header
  - Full-size HDMI



Raspberry Pi – Know your board

- The Raspberry Pi 3 Model B+
  - CSI camera port for connecting a Raspberry Pi camera
  - DSI display port for connecting a Raspberry Pi touchscreen display
  - 4-pole stereo output and composite video port
  - Micro SD port for loading your operating system and storing data
  - 5V/2.5A DC power input
  - Power-over-Ethernet (PoE) support (requires separate PoE HAT)

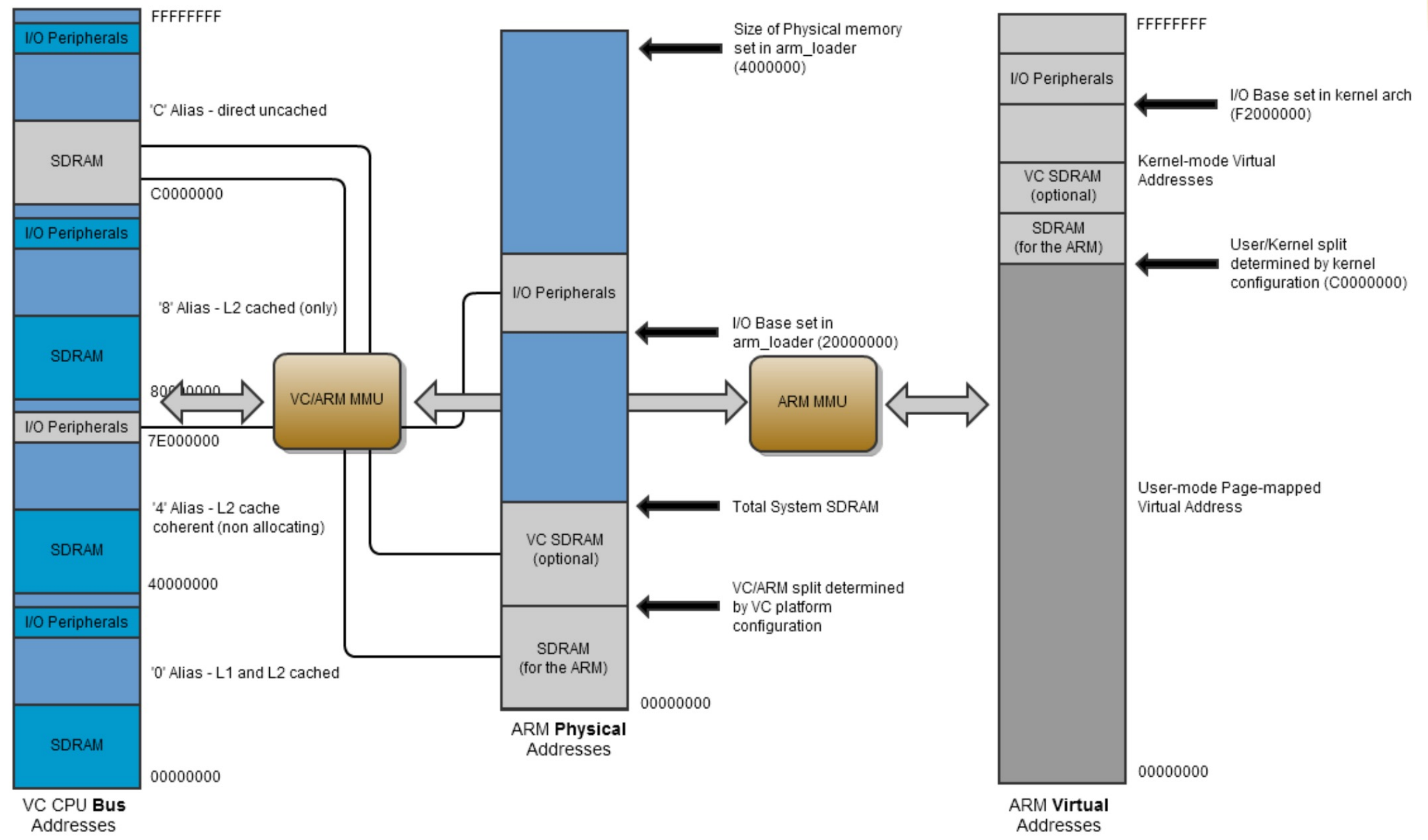


CAUTION!!

- Do not shutdown the RPi by pulling the USB cable, use a software shutdown procedure
- Do not place powered RPi on metal surfaces. If you short the pins underneath the GPIO header, you can destroy the board
- Do not connect circuits that source/sink other than very low currents
- The GPIO pins are 3.3V tolerant, not 5V
- Carefully check pin numbers, don’t short the GPIO pins

ARM Peripherals

BCM2837 Manual

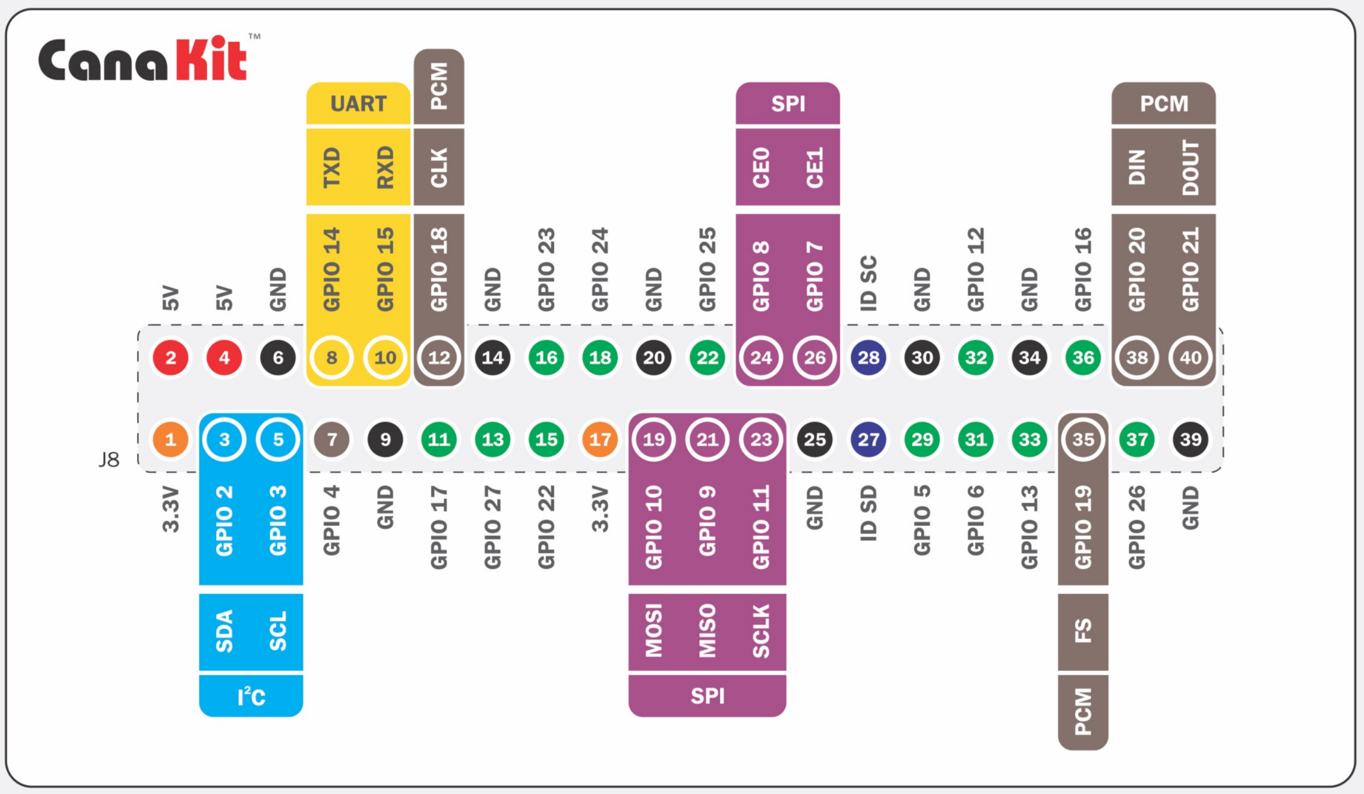


Address Mapping

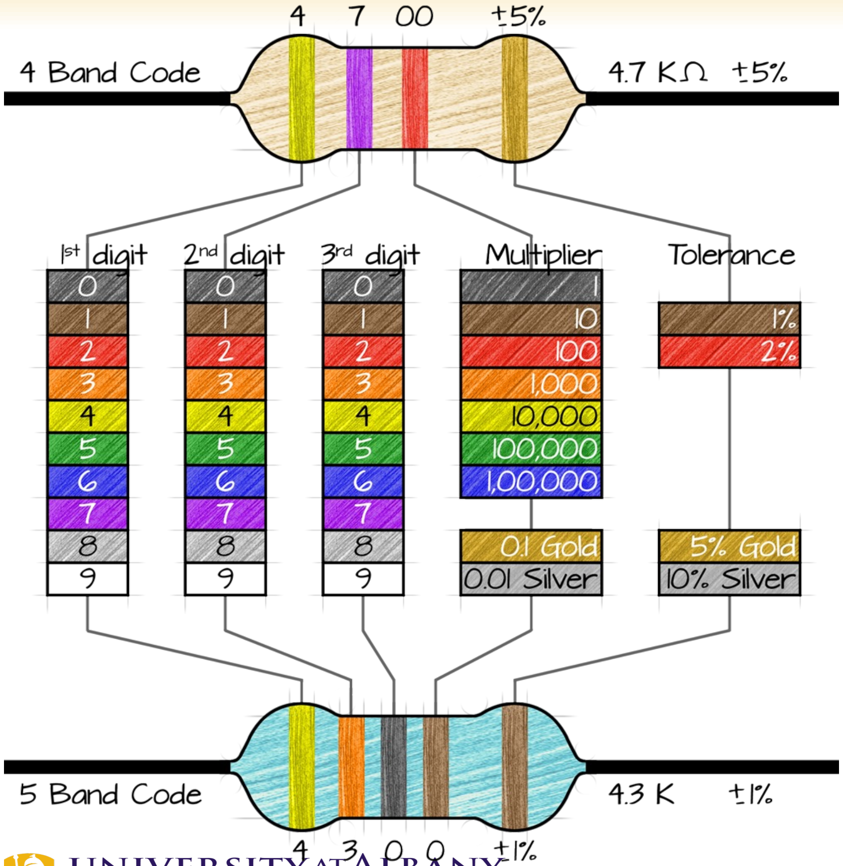
- Addresses in ARM Linux are:
- issued as virtual addresses by the ARM core,
- mapped into a physical address by the ARM/MMU,
- mapped into a bus address by the ARM mapping MMU,
- used to select the appropriate peripheral or location in RAM.

GPIO Pins

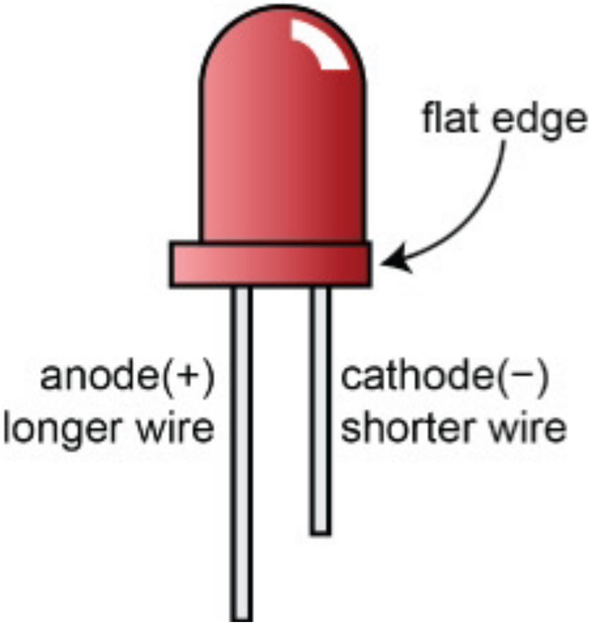
- https://pinout.xyz



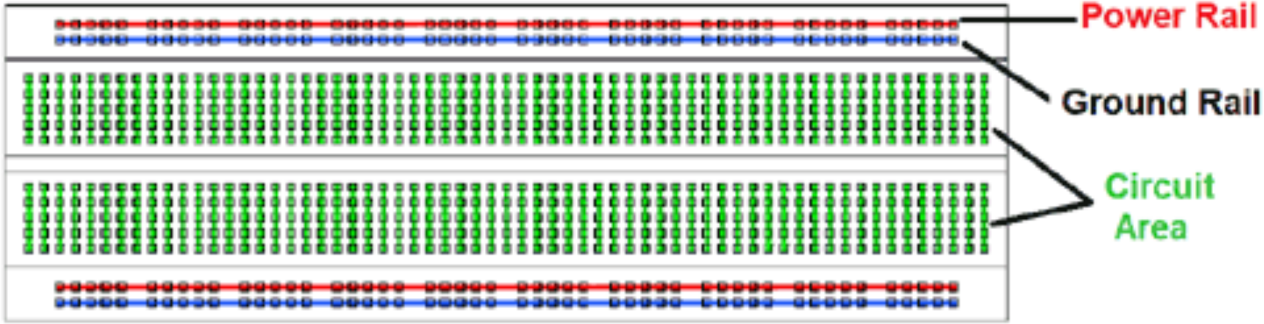
Resistors and LEDs

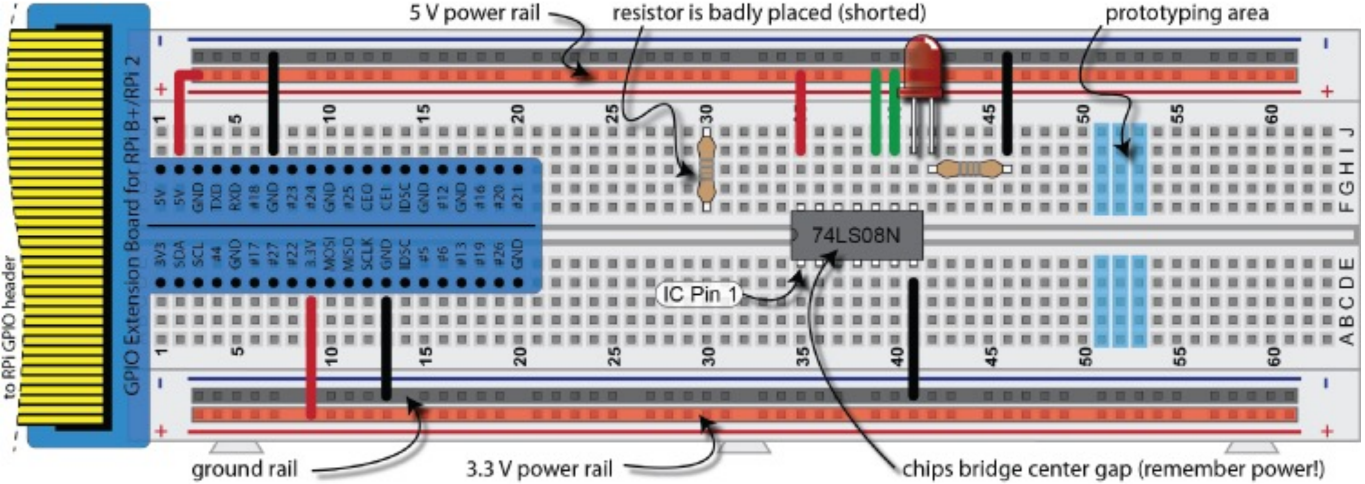


1st digit	2nd digit	3rd digit	Multiplier	Tolerance
0	0	0	1	
1	1	1	10	1%
2	2	2	100	2%
3	3	3	1,000	
4	4	4	10,000	
5	5	5	100,000	
6	6	6	1,00,000	
7	7	7		
8	8	8	0.1 Gold	5% Gold
9	9	9	0.01 Silver	10% Silver

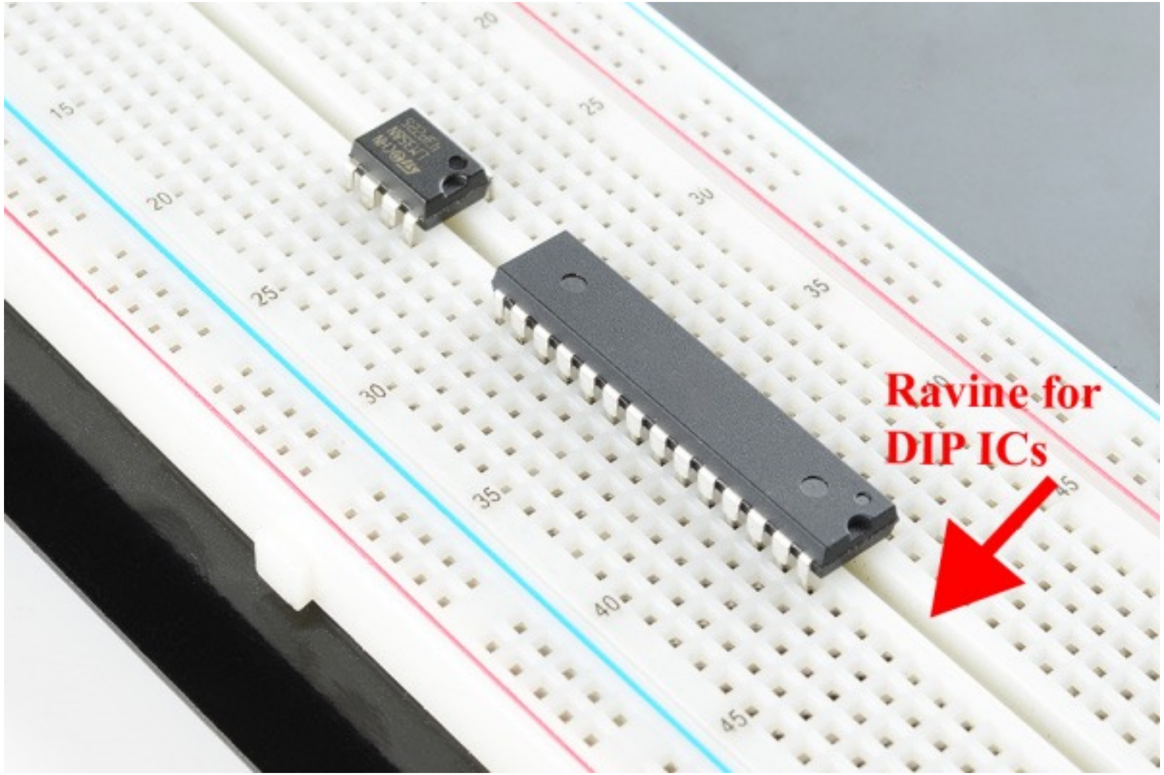


Breadboard Connections



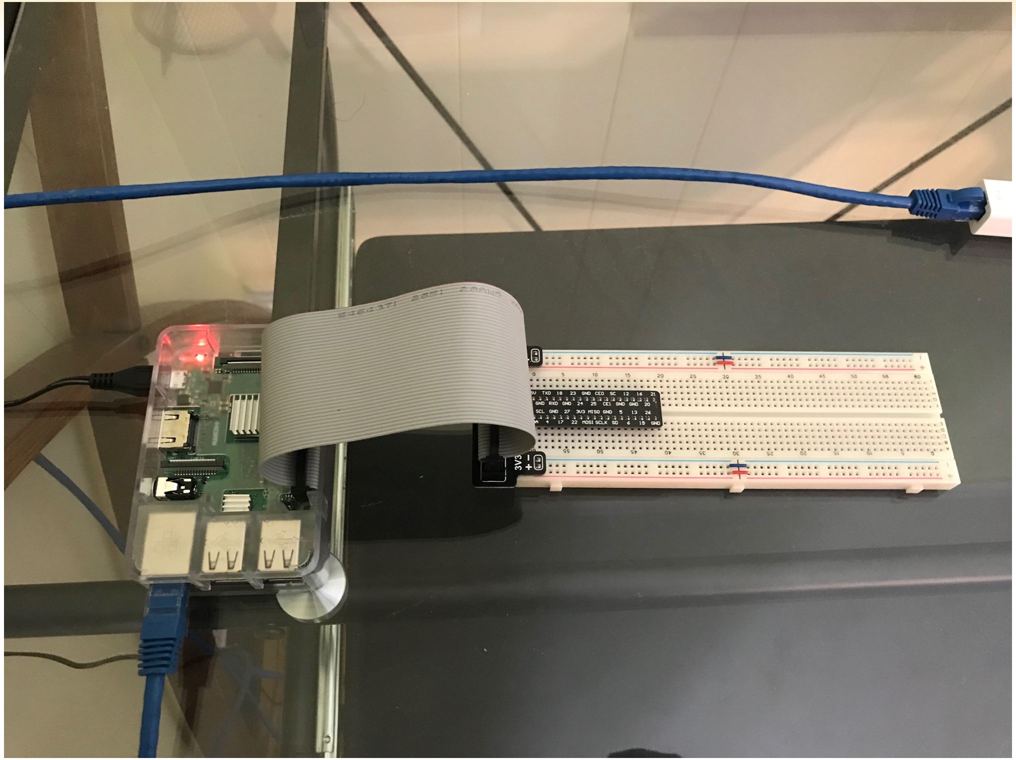


Dual In-Line Package or DIP

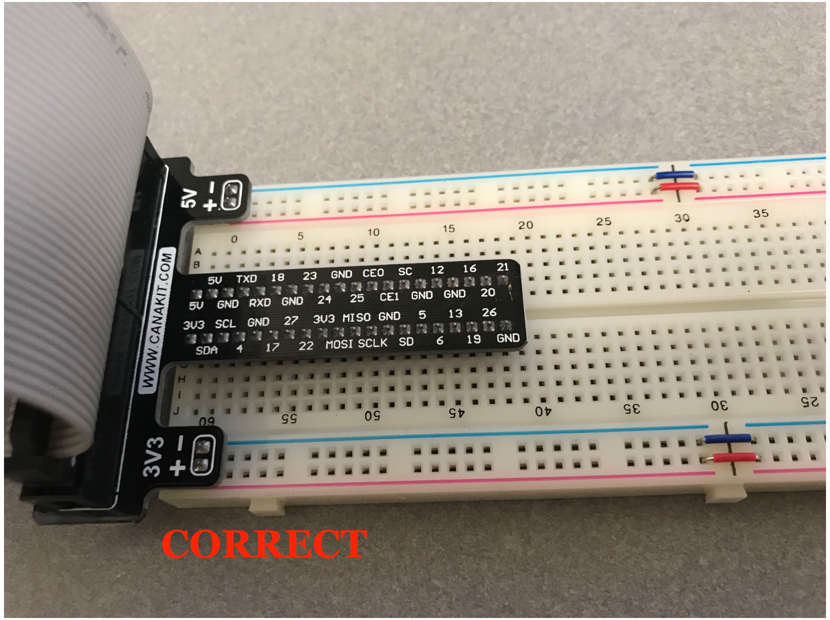

Ravine for DIP ICs

GPIO

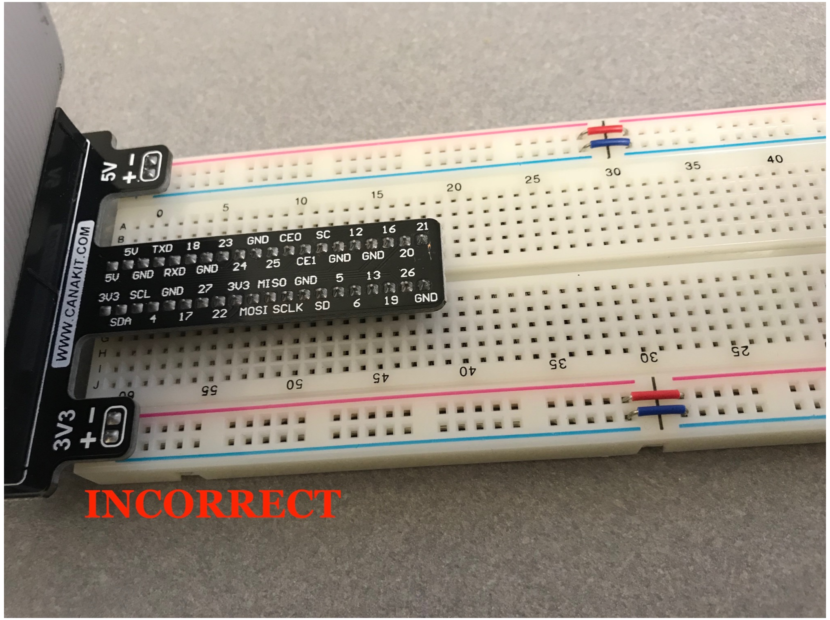
- GPIO to Breadboard Interface Board
- GPIO Ribbon Cable
- Breadboard



Convention



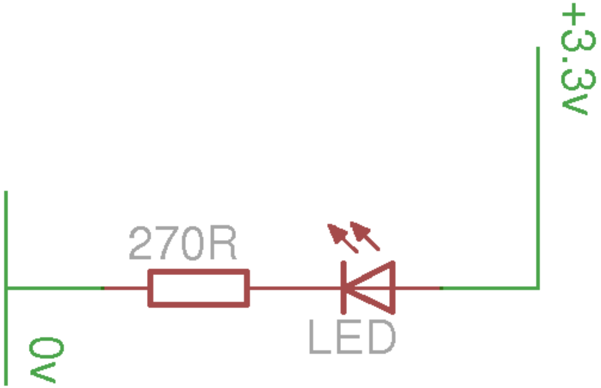
CORRECT

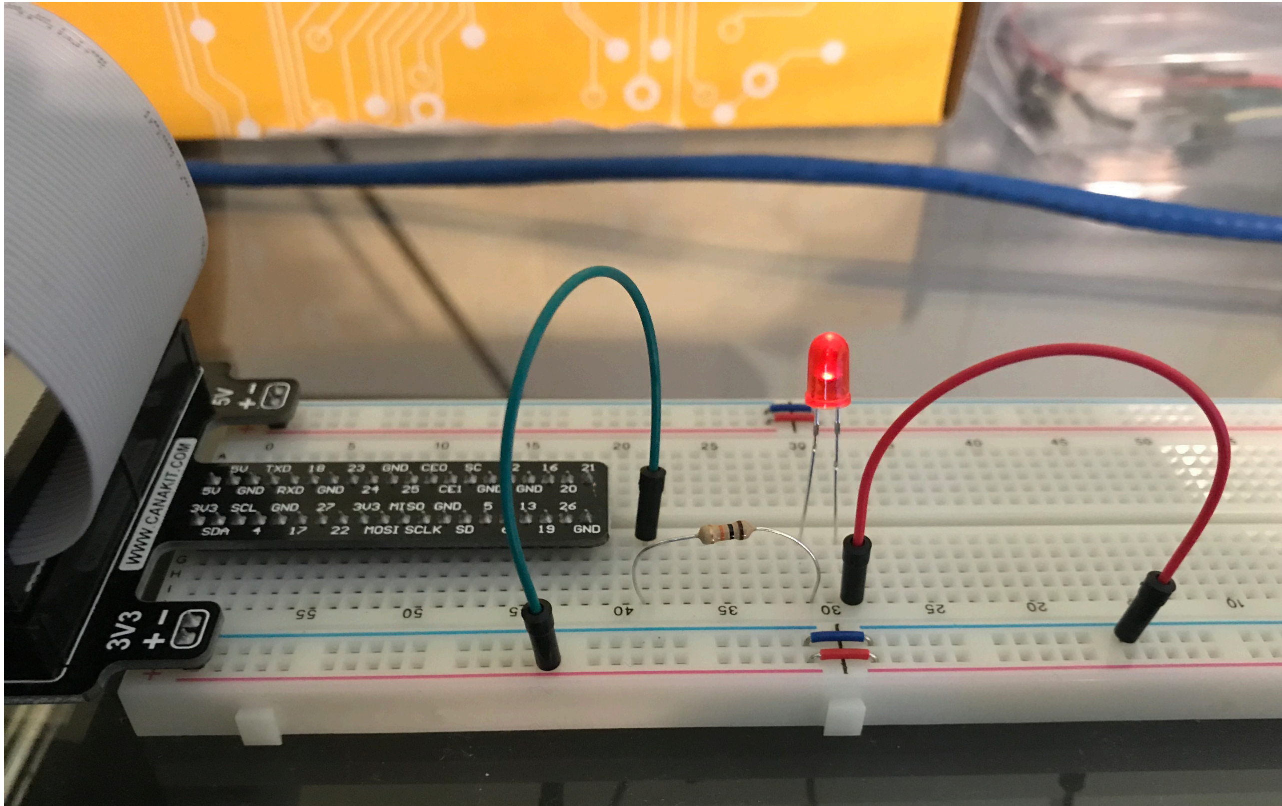


INCORRECT

Circuit to Breadboard

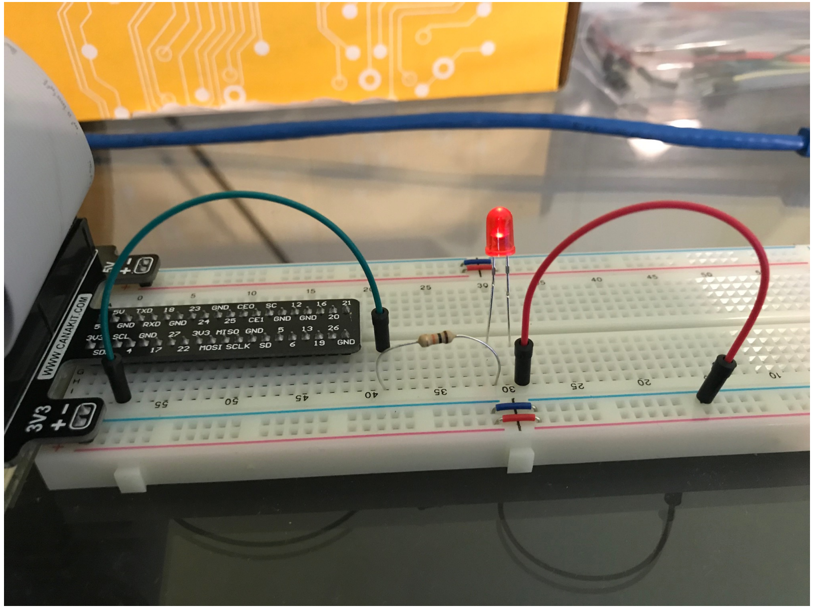
- Use 3V





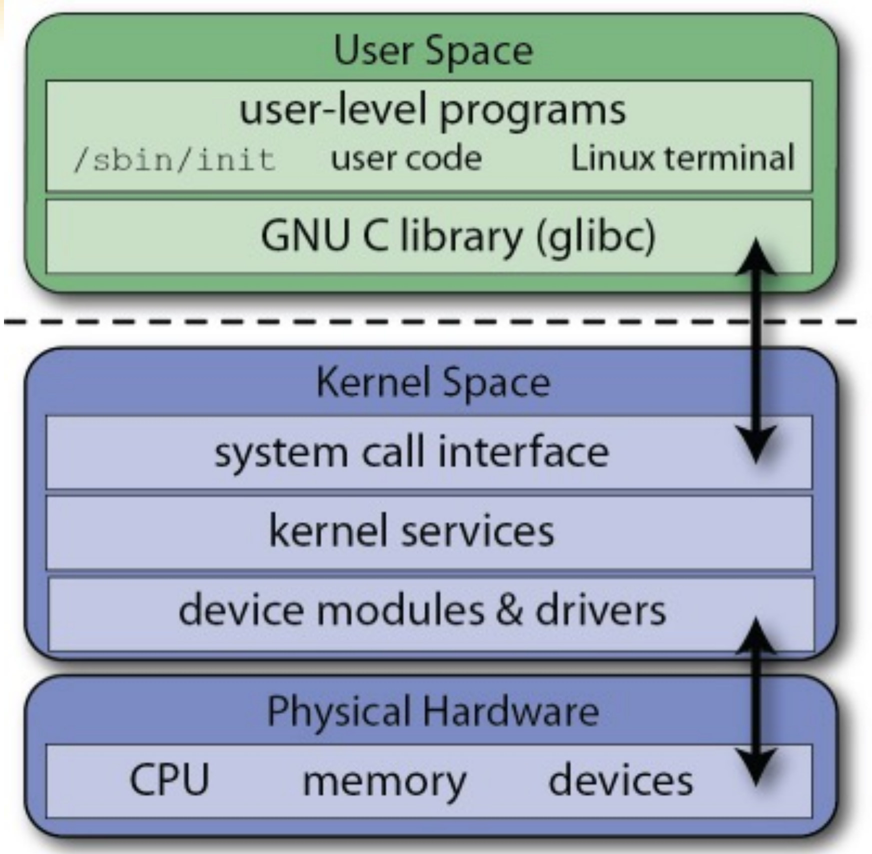
Circuit to Breadboard

- Use GPIO pin



sysfs – pseudo-filesystem

- The **sysfs** filesystem is a pseudo-filesystem which provides an interface to kernel data structures.
- The files under **sysfs** provide information about devices, kernel modules, filesystems, and other kernel components.



Linux Kernel vs User Space

- The Linux kernel runs in an area of system memory called the *kernel space*
- Regular user applications run in an area of system memory called *user space*
- A hard boundary between these two spaces prevents
  - User applications from accessing memory and resources required by the Linux kernel
  - Linux kernel from crashing due to badly written user code
  - Interfering one user’s applications with another
  - Provides a degree of security.

sysfs

- Paths in sysfs (/sys/class/gpio)
  - Control interfaces used to get userspace control over GPIOs
    - export
    - unexport
  - GPIOs themselves
  - GPIO controllers ("gpiochip" instances)
- GPIO signals have paths like /sys/class/gpio/gpioN/
  - "direction" - reads as either "in" or "out”
  - "value" - reads as either 0 (low) or 1 (high)
  - "edge" - reads as either "none", "rising", "falling", or "both”
  - "active_low" - reads as either 0 (false) or 1 (true)

Steps to perform I/O using sysfs

- Export the pin.
- Set the pin direction (input or output).
- If an output pin, set the level to low or high.
- If an input pin, read the pin's level (low or high).
- When done, unexport the pin.

Exporting GPIO control to userspace

- "export"
  - Userspace may ask the kernel to export control of a GPIO to userspace by writing its number to this file.
  - Example: "echo 19 > export" will create a "gpio19" node for GPIO #19, if that's not requested by kernel code.
- "unexport"
  - Reverses the effect of exporting to userspace.
  - Example: "echo 19 > unexport" will remove a "gpio19" node exported using the "export" file.

Control GPIO with Linux

- Become the sudo user

```
dsaha@sahaPi:~ $ sudo su
```

- Go to the GPIO folder and list the contents

```
root@sahaPi:/home/dsaha# cd /sys/class/gpio/
root@sahaPi:/sys/class/gpio# ls
export gpiochip0  gpiochip128 unexport
```

- Export gpio 4

```
root@sahaPi:/sys/class/gpio# echo 4 > export
root@sahaPi:/sys/class/gpio# ls
export gpio4  gpiochip0  gpiochip128  unexport
```

Control GPIO with Linux

- Go to the gpio4 folder and list contents

  ```
  root@sahaPi:/sys/class/gpio# cd gpio4/
  root@sahaPi:/sys/class/gpio/gpio4# ls
  active_low  device  direction  edge  power  subsystem  uevent  value
  ```

- Set direction (in or out) of pin

  ```
  root@sahaPi:/sys/class/gpio/gpio4# echo out > direction
  ```

- Set value to be 1 to turn on the LED

  ```
  root@sahaPi:/sys/class/gpio/gpio4# echo 1 > value
  ```

Control GPIO with Linux

- Set value to be 0 to turn off the LED
  - root@sahaPi:/sys/class/gpio/gpio4# echo 0 > value
- Check the status (direction and value) of the pin
  - root@sahaPi:/sys/class/gpio/gpio4# cat direction
  - out
  - root@sahaPi:/sys/class/gpio/gpio4# cat value
  - 0

Control GPIO with Linux

- Ready to give up the control? Get out of gpio4 folder and list contents, which shows gpio4 folder
  - `root@sahaPi:/sys/class/gpio/gpio4# cd ../`
  - `root@sahaPi:/sys/class/gpio# ls`
  - `export gpio4  gpiochip0  gpiochip128  unexport`
- Unexport gpio 4 and list contents showing removal of gpio4 folder
  - `root@sahaPi:/sys/class/gpio# echo 4 > unexport`
  - `root@sahaPi:/sys/class/gpio# ls`
  - `export gpiochip0  gpiochip128 unexport`

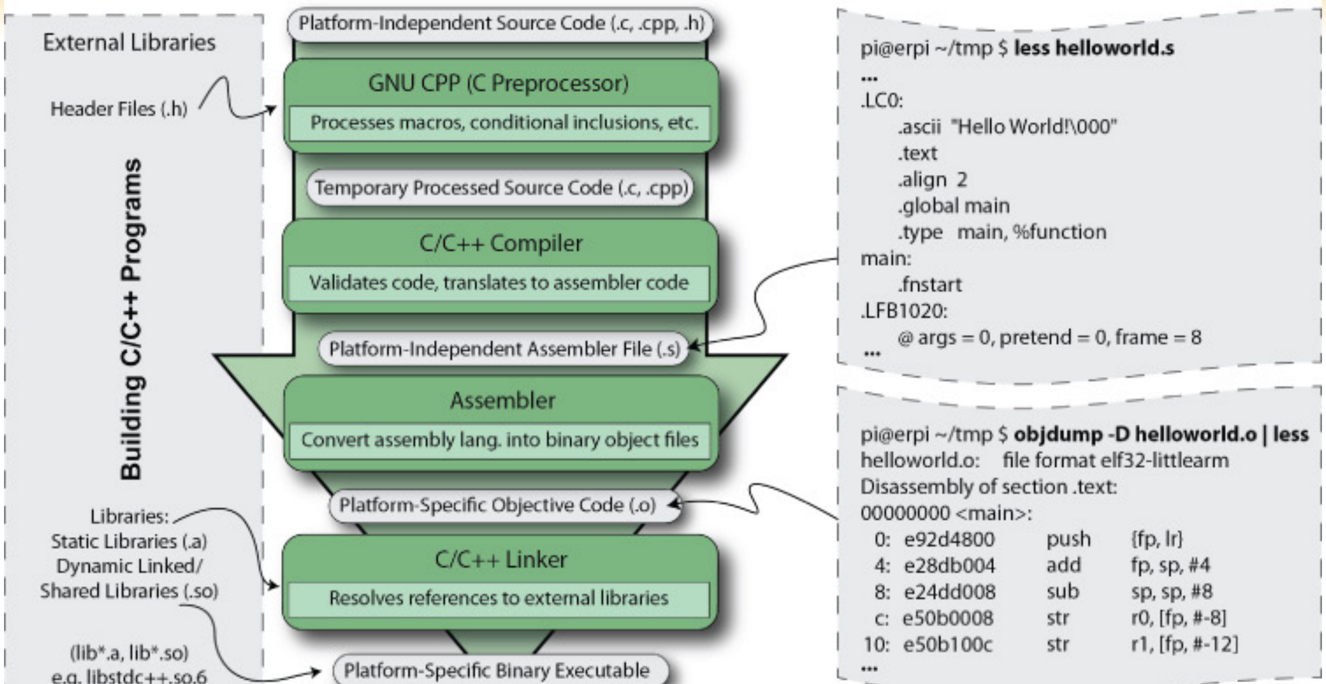
Program

- Bash Script
  - exploringrpi/chp05/bashLED/bashLED
- Python Code
  - exploringrpi/chp05/pythonLED/python2LED.py
- C code
  - exploringrpi/chp05/makeLED/makeLED.c

C/C++

ADVANTAGES	DISADVANTAGES
You can build code directly on the RPi or you can **cross-compile code** using professional toolchains. Runtime environments do not need to be installed.	Compiled **code is not portable**. Code compiled for your x86 desktop will not run on the RPi ARM processor.
C++ has full **support** for procedural programming, OOP, and support for generics through the use of STL (Standard Template Library).	Many consider the languages to be complex to master. There is a tendency to need to know everything before you can do anything.
It gives the best **computational performance**, especially *if optimized*. However, optimization can be difficult and can reduce the portability of your code.	The use of pointers and the low-level control available makes code prone to memory leaks. With careful coding these can be avoided and can lead to efficiencies over dynamic memory management schemes.
Can be used for high-performance user-interface application development on the RPi using third-party libraries. Libraries such as Qt and Boost provide extensive additional libraries for components, networking, etc.	By default, C and C++ do not support graphical user interfaces, network sockets, etc. Third-party libraries are required.
Offers **low-level access** to glibc for integrating with the Linux system. Programs can be setuid to root.	Not suitable for scripting (there is a C shell, csh, that does have syntax like C). You can integrate with Lua. Not ideal for web development either.
The **Linux kernel is written in C** and having knowledge of C/C++ can help if you ever have to write device drivers or contribute to Linux kernel development.	C++ attempts to span from low-level to high-level programming tasks, but it can be difficult to write very scalable enterprise or web applications.
The C/C++ languages are ISO standards, not owned by a single company.	

Building C/C++ Applications



External Libraries

Header Files (.h)

Building C/C++ Programs

Libraries:
Static Libraries (.a)
Dynamic Linked/
Shared Libraries (.so)

(lib*.a, lib*.so)
e.g. libstdc++.so.6

Platform-Independent Source Code (.c, .cpp, .h)

GNU CPP (C Preprocessor)
Processes macros, conditional inclusions, etc.

Temporary Processed Source Code (.c, .cpp)

C/C++ Compiler
Validates code, translates to assembler code

Platform-Independent Assembler File (.s)

Assembler
Convert assembly lang. into binary object files

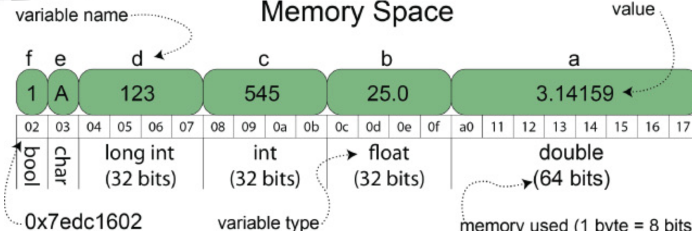
Platform-Specific Objective Code (.o)

C/C++ Linker
Resolves references to external libraries

Platform-Specific Binary Executable

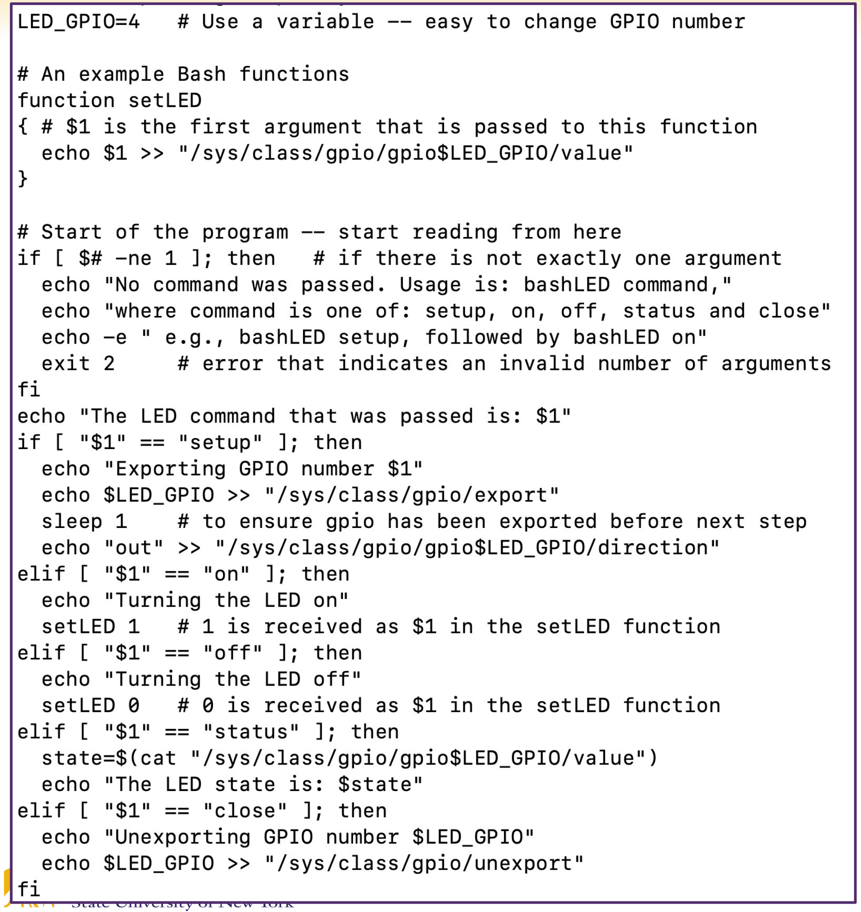
```
pi@erpi ~/tmp $ less helloworld.s
...
.LC0:
      .ascii  "Hello World!\000"
      .text
      .align  2
      .global main
      .type   main, %function
main:
      .fnstart
.LFB1020:
      @ args = 0, pretend = 0, frame = 8
...
```

```
pi@erpi ~/tmp $ objdump -D helloworld.o | less
helloworld.o:     file format elf32-littlearm
Disassembly of section .text:
00000000 <main>:
  0:  e92d4800      push      {fp, lr}
  4:  e28db004      add       fp, sp, #4
  8:  e24dd008      sub       sp, sp, #8
  c:  e50b0008      str       r0, [fp, #-8]
 10:  e50b100c      str       r1, [fp, #-12]
...
```

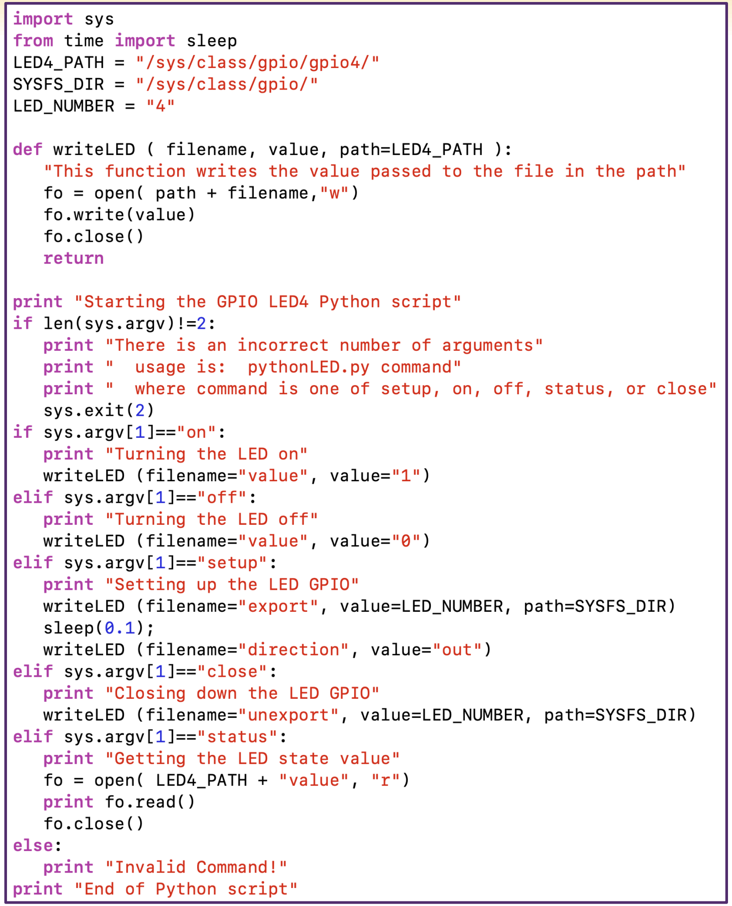


Memory Space

Bash and Python Script



```bash
LED_GPIO=4   # Use a variable -- easy to change GPIO number

# An example Bash functions
function setLED
{ # $1 is the first argument that is passed to this function
  echo $1 >> "/sys/class/gpio/gpio$LED_GPIO/value"
}

# Start of the program -- start reading from here
if [ $# -ne 1 ]; then   # if there is not exactly one argument
  echo "No command was passed. Usage is: bashLED command,"
  echo "where command is one of: setup, on, off, status and close"
  echo -e " e.g., bashLED setup, followed by bashLED on"
  exit 2     # error that indicates an invalid number of arguments
fi
echo "The LED command that was passed is: $1"
if [ "$1" == "setup" ]; then
  echo "Exporting GPIO number $1"
  echo $LED_GPIO >> "/sys/class/gpio/export"
  sleep 1    # to ensure gpio has been exported before next step
  echo "out" >> "/sys/class/gpio/gpio$LED_GPIO/direction"
elif [ "$1" == "on" ]; then
  echo "Turning the LED on"
  setLED 1   # 1 is received as $1 in the setLED function
elif [ "$1" == "off" ]; then
  echo "Turning the LED off"
  setLED 0   # 0 is received as $1 in the setLED function
elif [ "$1" == "status" ]; then
  state=$(cat "/sys/class/gpio/gpio$LED_GPIO/value")
  echo "The LED state is: $state"
elif [ "$1" == "close" ]; then
  echo "Unexporting GPIO number $LED_GPIO"
  echo $LED_GPIO >> "/sys/class/gpio/unexport"
fi
```



```python
import sys
from time import sleep
LED4_PATH = "/sys/class/gpio/gpio4/"
SYSFS_DIR = "/sys/class/gpio/"
LED_NUMBER = "4"

def writeLED ( filename, value, path=LED4_PATH ):
   "This function writes the value passed to the file in the path"
   fo = open( path + filename,"w")
   fo.write(value)
   fo.close()
   return

print "Starting the GPIO LED4 Python script"
if len(sys.argv)!=2:
   print "There is an incorrect number of arguments"
   print "  usage is:  pythonLED.py command"
   print "  where command is one of setup, on, off, status, or close"
   sys.exit(2)
if sys.argv[1]=="on":
   print "Turning the LED on"
   writeLED (filename="value", value="1")
elif sys.argv[1]=="off":
   print "Turning the LED off"
   writeLED (filename="value", value="0")
elif sys.argv[1]=="setup":
   print "Setting up the LED GPIO"
   writeLED (filename="export", value=LED_NUMBER, path=SYSFS_DIR)
   sleep(0.1);
   writeLED (filename="direction", value="out")
elif sys.argv[1]=="close":
   print "Closing down the LED GPIO"
   writeLED (filename="unexport", value=LED_NUMBER, path=SYSFS_DIR)
elif sys.argv[1]=="status":
   print "Getting the LED state value"
   fo = open( LED4_PATH + "value", "r")
   print fo.read()
   fo.close()
else:
   print "Invalid Command!"
print "End of Python script"
```

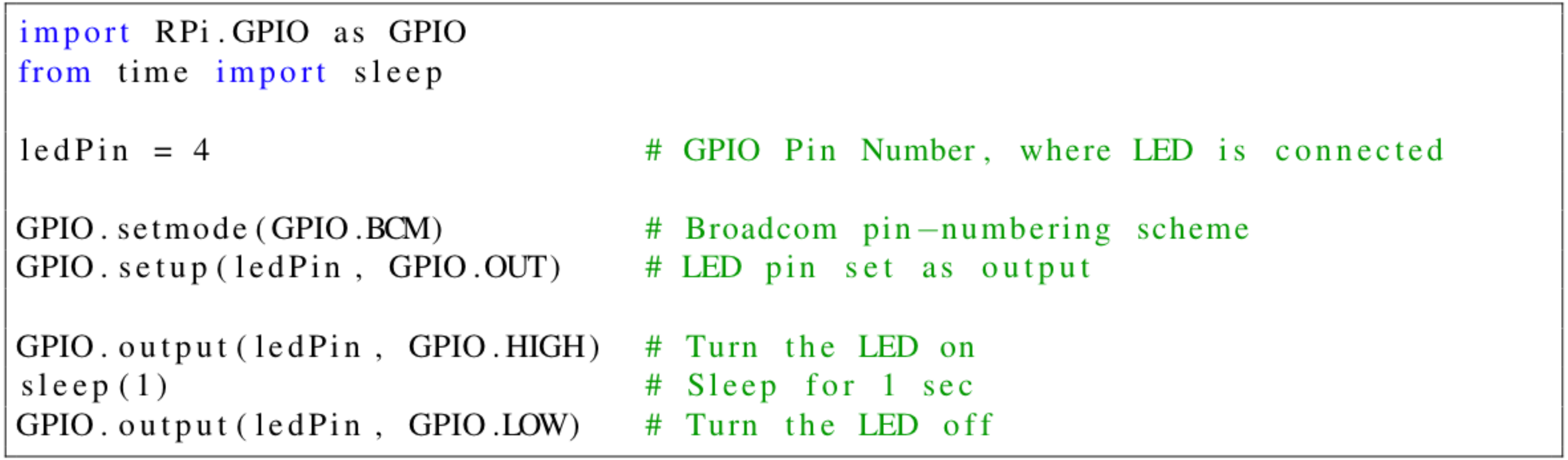
C Program

```
#define GPIO_NUMBER "4"
#define GPIO4_PATH "/sys/class/gpio/gpio4/"
#define GPIO_SYSFS "/sys/class/gpio/"

void writeGPIO(char filename[], char value[]){
   FILE* fp;                           // create a file pointer fp
   fp = fopen(filename, "w+");         // open file for writing
   fprintf(fp, "%s", value);           // send the value to the file
   fclose(fp);                         // close the file using fp
}

int main(int argc, char* argv[]){
   if(argc!=2){                        // program name is argument 1
      printf("Usage is makeLEDC and one of:\n");
      printf("   setup, on, off, status, or close\n");
      printf(" e.g. makeLEDC on\n");
      return 2;                        // invalid number of arguments
   }
   printf("Starting the makeLED program\n");
   if(strcmp(argv[1],"setup")==0){
      printf("Setting up the LED on the GPIO\n");
      writeGPIO(GPIO_SYSFS "export", GPIO_NUMBER);
      usleep(100000);                  // sleep for 100ms
      writeGPIO(GPIO4_PATH "direction", "out");
   }
   else if(strcmp(argv[1],"close")==0){
      printf("Closing the LED on the GPIO\n");
      writeGPIO(GPIO_SYSFS "unexport", GPIO_NUMBER);
   }
```

```
   else if(strcmp(argv[1],"on")==0){
      printf("Turning the LED on\n");
      writeGPIO(GPIO4_PATH "value", "1");
   }
   else if (strcmp(argv[1],"off")==0){
      printf("Turning the LED off\n");
      writeGPIO(GPIO4_PATH "value", "0");
   }
   else if (strcmp(argv[1],"status")==0){
      FILE* fp;       // see writeGPIO function above for description
      char line[80], fullFilename[100];
      sprintf(fullFilename, GPIO4_PATH "/value");
      fp = fopen(fullFilename, "rt");      // reading text this time
      while (fgets(line, 80, fp) != NULL){
         printf("The state of the LED is %s", line);
      }
      fclose(fp);
   }
   else{
      printf("Invalid command!\n");
   }
   printf("Finished the makeLED Program\n");
   return 0;
}
```

Use Rpi Library

- https://sourceforge.net/projects/raspberry-gpio-python/
- Note: Current release does not support SPI, I2C, 1-wire or serial functionality on the RPi yet



```
import RPi.GPIO as GPIO
from time import sleep

ledPin = 4                      # GPIO Pin Number, where LED is connected

GPIO.setmode(GPIO.BCM)          # Broadcom pin-numbering scheme
GPIO.setup(ledPin, GPIO.OUT)    # LED pin set as output

GPIO.output(ledPin, GPIO.HIGH)  # Turn the LED on
sleep(1)                        # Sleep for 1 sec
GPIO.output(ledPin, GPIO.LOW)   # Turn the LED off
```

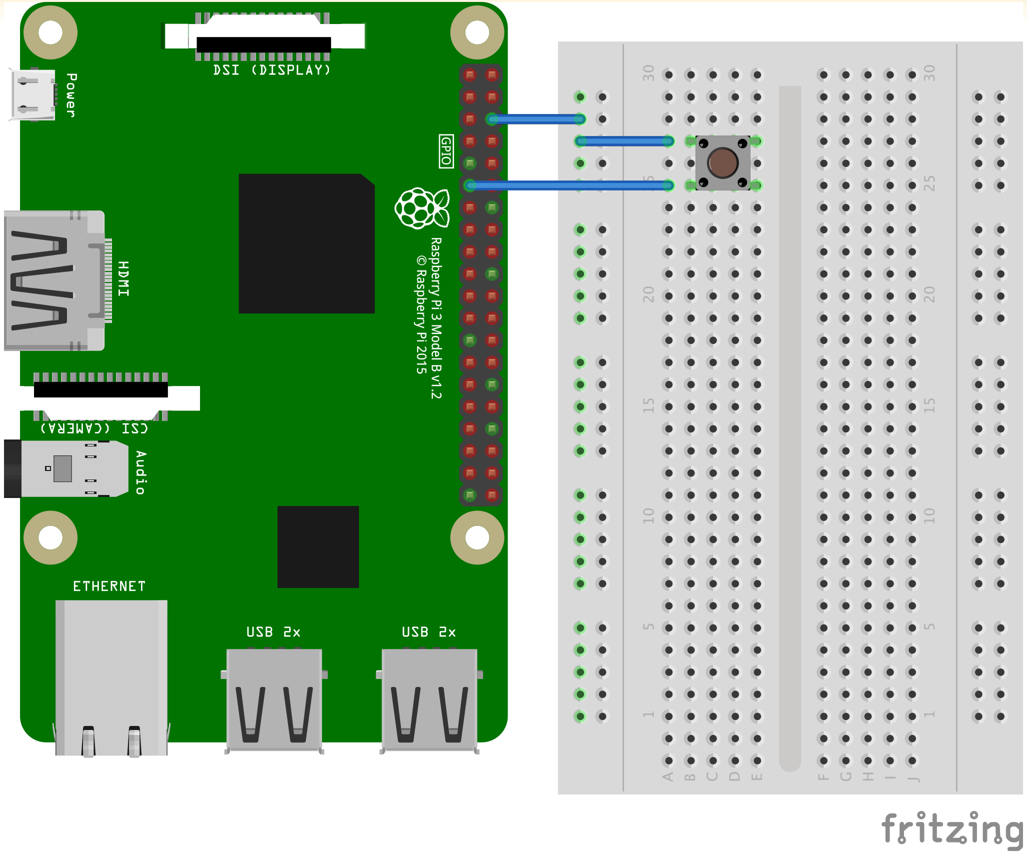
Use gpiozero Library

- https://gpiozero.readthedocs.io/en/stable/

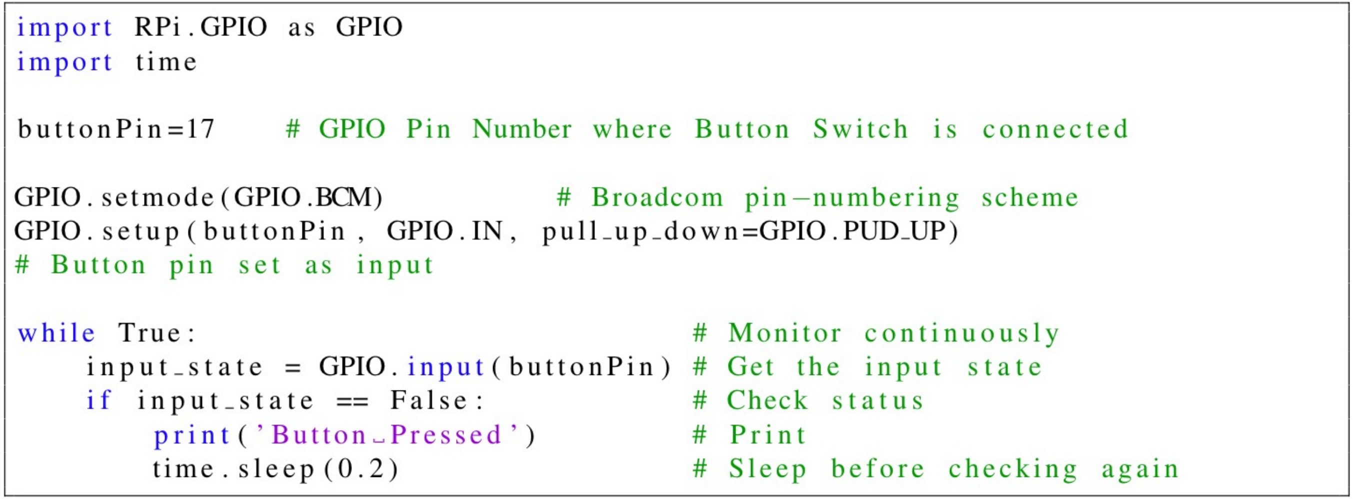
```python
from gpiozero import LED
from time import sleep

led = LED(4)        # GPIO Pin Number
led.on()            # Turn on LED
sleep(1)            # Sleep for 1 sec
led.off()           # Turn off LED
```

GPIO as Input

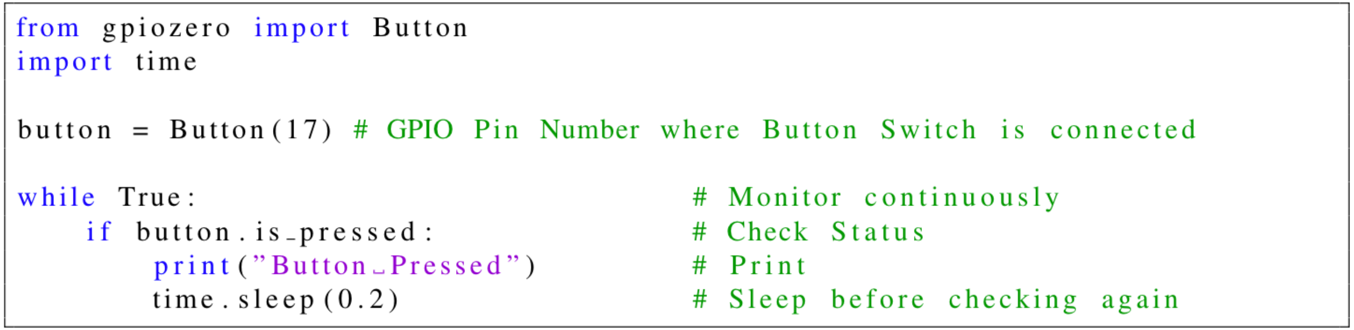
- Push-button Switch



Reading GPIO



```
import RPi.GPIO as GPIO
import time

buttonPin=17     # GPIO Pin Number where Button Switch is connected

GPIO.setmode(GPIO.BCM)             # Broadcom pin-numbering scheme
GPIO.setup(buttonPin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
# Button pin set as input

while True:                             # Monitor continuously
    input_state = GPIO.input(buttonPin) # Get the input state
    if input_state == False:            # Check status
        print(’Button␣Pressed’)         # Print
        time.sleep(0.2)                 # Sleep before checking again
```

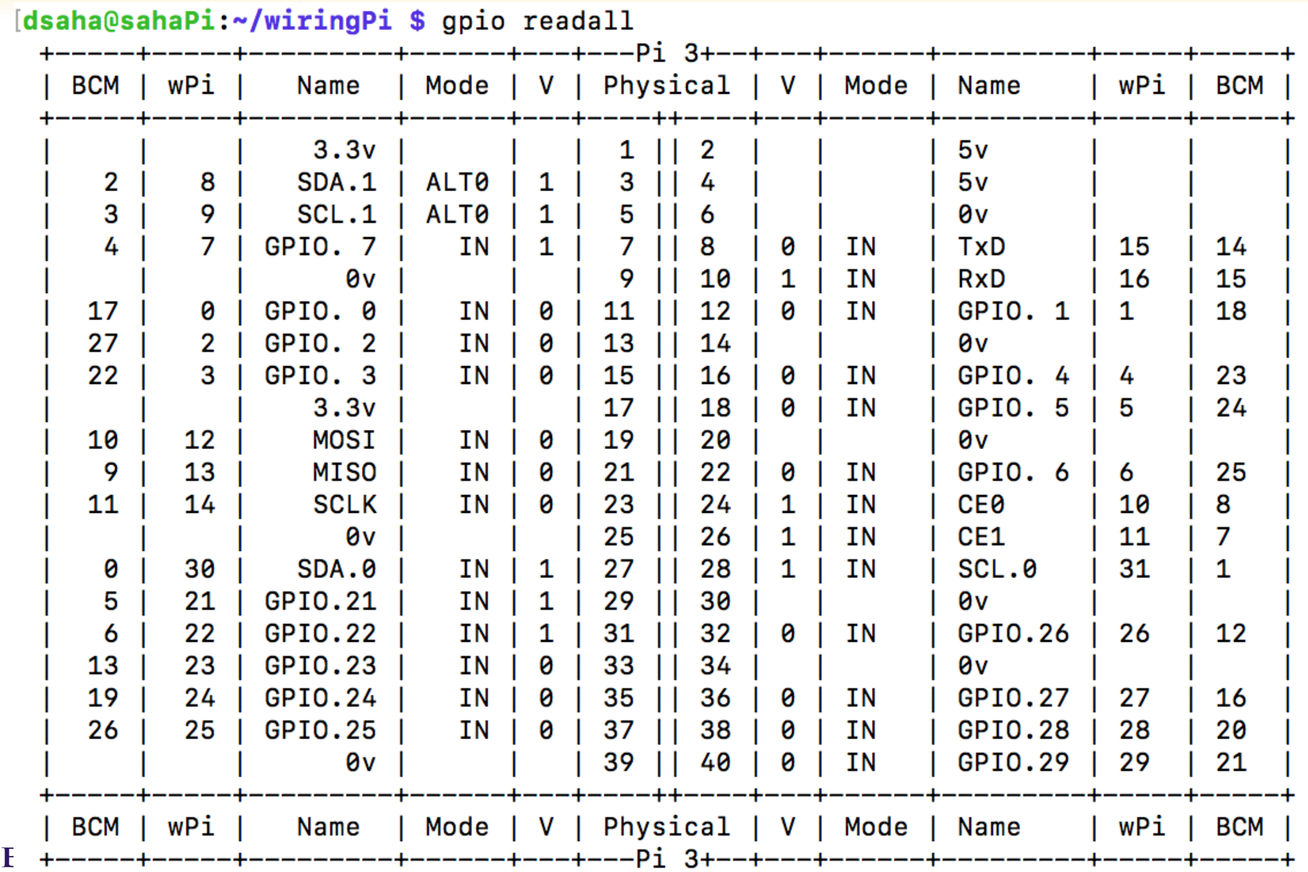


```
from gpiozero import Button
import time

button = Button(17) # GPIO Pin Number where Button Switch is connected

while True:                             # Monitor continuously
    if button.is_pressed:               # Check Status
        print(”Button␣Pressed”)         # Print
        time.sleep(0.2)                 # Sleep before checking again
```

Wiring Pi

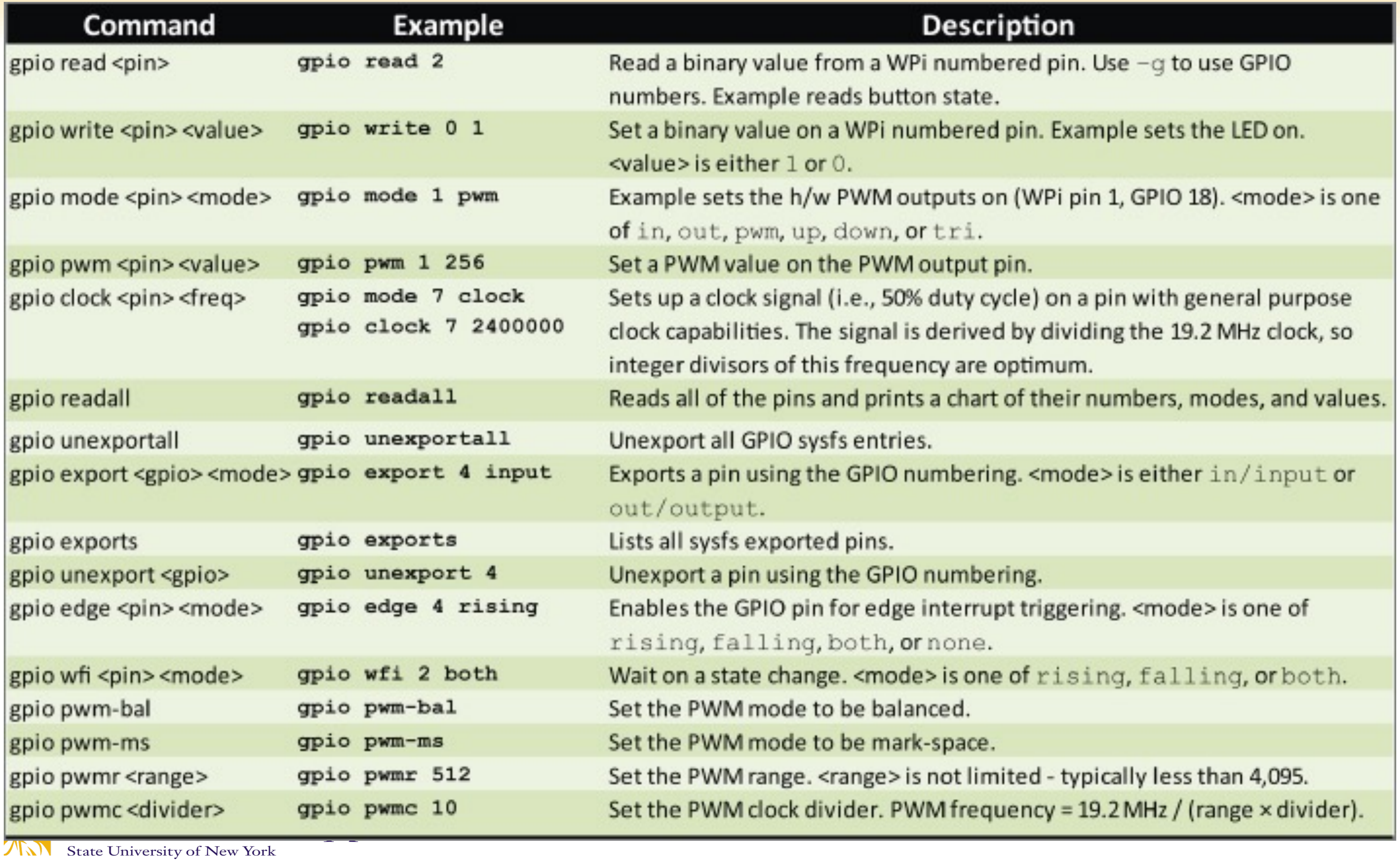
- http://wiringpi.com

WiringPi



```
[dsaha@sahaPi:~/wiringPi $ gpio readall
 +-----+-----+---------+------+---+---Pi 3+--+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 | ALT0 | 1 |  3 || 4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 | ALT0 | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 0 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3+--+---+------+---------+-----+-----+
```

The gpio Command (WiringPi)



Command	Example	Description
gpio read <pin>	`gpio read 2`	Read a binary value from a WPi numbered pin. Use `-g` to use GPIO numbers. Example reads button state.
gpio write <pin> <value>	`gpio write 0 1`	Set a binary value on a WPi numbered pin. Example sets the LED on. <value> is either `1` or `0`.
gpio mode <pin> <mode>	`gpio mode 1 pwm`	Example sets the h/w PWM outputs on (WPi pin 1, GPIO 18). <mode> is one of `in`, `out`, `pwm`, `up`, `down`, or `tri`.
gpio pwm <pin> <value>	`gpio pwm 1 256`	Set a PWM value on the PWM output pin.
gpio clock <pin> <freq>	`gpio mode 7 clock` `gpio clock 7 2400000`	Sets up a clock signal (i.e., 50% duty cycle) on a pin with general purpose clock capabilities. The signal is derived by dividing the 19.2 MHz clock, so integer divisors of this frequency are optimum.
gpio readall	`gpio readall`	Reads all of the pins and prints a chart of their numbers, modes, and values.
gpio unexportall	`gpio unexportall`	Unexport all GPIO sysfs entries.
gpio export <gpio> <mode>	`gpio export 4 input`	Exports a pin using the GPIO numbering. <mode> is either `in/input` or `out/output`.
gpio exports	`gpio exports`	Lists all sysfs exported pins.
gpio unexport <gpio>	`gpio unexport 4`	Unexport a pin using the GPIO numbering.
gpio edge <pin> <mode>	`gpio edge 4 rising`	Enables the GPIO pin for edge interrupt triggering. <mode> is one of `rising`, `falling`, `both`, or `none`.
gpio wfi <pin> <mode>	`gpio wfi 2 both`	Wait on a state change. <mode> is one of `rising`, `falling`, or `both`.
gpio pwm-bal	`gpio pwm-bal`	Set the PWM mode to be balanced.
gpio pwm-ms	`gpio pwm-ms`	Set the PWM mode to be mark-space.
gpio pwmr <range>	`gpio pwmr 512`	Set the PWM range. <range> is not limited - typically less than 4,095.
gpio pwmc <divider>	`gpio pwmc 10`	Set the PWM clock divider. PWM frequency = 19.2 MHz / (range × divider).

wiringPi

- Functions

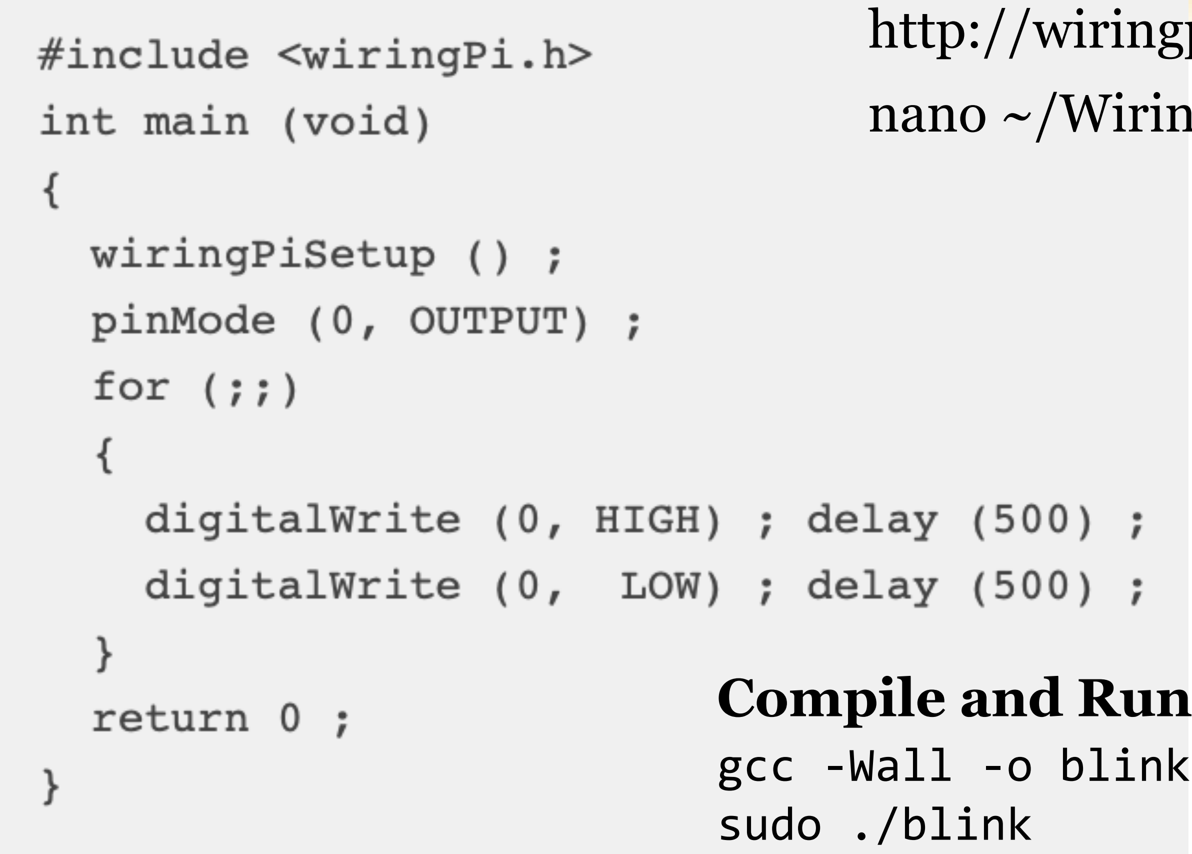
Return	Function Call	Description
Setup		
int	`wiringPiSetup(void)`	Initializes wiringPi. Must be used with root privileges. Returns 0 if successful.
int	`wiringPiSetupGpio(void)`	Same as above. Uses GPIO rather than WPi numbers. Must use root privileges.
int	`wiringPiSetupSys(void)`	Uses sysfs. Root not required if udev rules in place (see end of chapter). You must manually export pins. Slower, as memory-mapping does not work.
int	`wiringPiSetupPhys(void)`	Uses the physical pin numbering on the RPi.
int	`piBoardRev(void)`	Returns the board version (0=n/a, 1=A, 2=B, 3=B+, 4=compute, 5= A+, 6=RPi 2)
GPIO Control		
void	`pinMode(int pin, int mode)`	Sets the pin to be one of `INPUT`, `OUTPUT`, or `PWM_OUTPUT` (on the hardware PWM pins only). Not available if `wiringPiSetupSys()` is used.
int	`getAlt(int pin)`	Get the ALT mode for a pin.
void	`pinModeAlt(int pin, int mode)`	Set the ALT mode for a pin.
void	`digitalWrite(int pin, int value)`	Sets the pin to be one of `HIGH` (1) or `LOW` (0). The pin mode must be OUTPUT.
void	`digitalWriteByte(int value)`	Fast parallel write of 8 bits to the first eight GPIO pins.
int	`digitalRead(int pin)`	Returns the input on a pin and returns either `HIGH` (1) or `LOW` (0).
void	`pullUpDnControl(int pin, int pud)`	Sets the pull-up or pull-down resistor type to be one of `PUD_OFF` (none), `PUD_U` (pull up), or `PUD_DOWN` (pull down). Not available in sysfs mode.
PWM and Timers		
void	`pwmWrite(int pin, int value)`	Sets the PWM output for a h/w PWM pin. Not available in sysfs mode.
void	`pwmSetMode(int mode)`	RPi PWM has two modes `PWM_MODE_BAL` (balanced) or `PWM_MODE_MS` (mark-space ratio). MS mode is most commonly used. BAL affects PWM frequency.
void	`pwmSetRange(unsigned int range)`	Sets the PWM range register. Valid values 2-4,095. Range and divisor affect frequency.
void	`pwmSetClock(int divisor)`	Sets the PWM clock divisor. PWM frequency = 19.2MHz / (divisor × range)
void	`pwmToneWrite(int pin, int freq)`	Set the frequency using the hardware PWM pin.
void	`gpioClockSet(int pin, int freq)`	Sets the frequency on a GPIO clock pin.
Interrupts		
int	`waitForInterrupt(int pin,int timeout)`	Waits for an interrupt. Timeout is set in ms where -1 is none. You must initialize the pin from outside the program, or using `system()` and the `gpio` command.
int	`wiringPiISR(int pin, int edgeType, void (*function)(void));`	Set a callback function (ISR) to be called on an interrupt event, which is one of `INT_EDGE_FALLING`, `INT_EDGE_RISING`, `INT_EDGE_BOTH`, or `INT_EDGE_SETUP`.
int	`piHiPri(int priority)`	Sets the priority of the program (0 to 99) allowing for a reduction in latency. Must be run as root. Returns 0 for success and -1 otherwise.
Helper Functions		
int	`wpiPinToGpio(int wPiPin)`	Converts WPi numbers into GPIO numbers.
int	`physPinToGpio(int physPin)`	Converts physical pin numbers to GPIO numbers.
uint32_t	`millis(void)`	Returns the number of milliseconds since a setup function was called.
uint32_t	`micros(void)`	Returns the number of microseconds since a setup function was called.
void	`delay(unsigned int t_ms)`	Delays for `t_ms` milliseconds. Delay is non-blocking and will exhibit latency.
void	`delayMicroseconds(unsigned int t_us)`	Delays for a number of microseconds.

Table information gleaned from `wiringPi.h` and `wiringPi.c`, which are distributed in the `/wiringPi/` directory of the wiringPi repository

wiringPi Blink LED

http://wiringpi.com/examples/blink/

nano ~/WiringPi/examples/blink.c

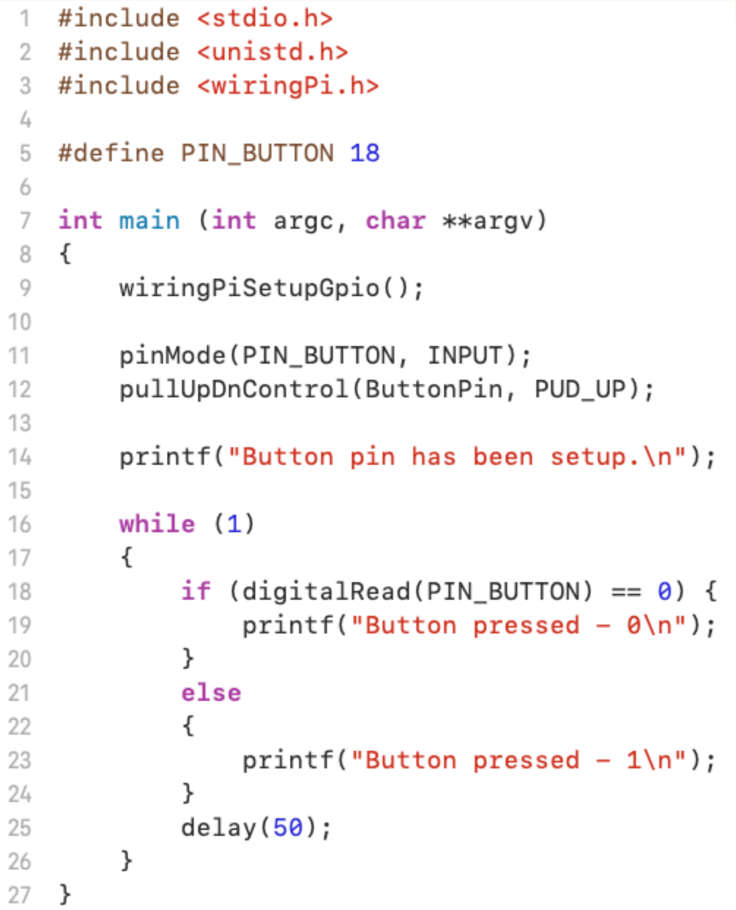


```c
#include <wiringPi.h>
int main (void)
{
  wiringPiSetup () ;
  pinMode (0, OUTPUT) ;
  for (;;)
  {
    digitalWrite (0, HIGH) ; delay (500) ;
    digitalWrite (0,  LOW) ; delay (500) ;
  }
  return 0 ;
}
```

Compile and Run

```
gcc -Wall -o blink blink.c -lwiringPi
sudo ./blink
```

Digital Input - Polling

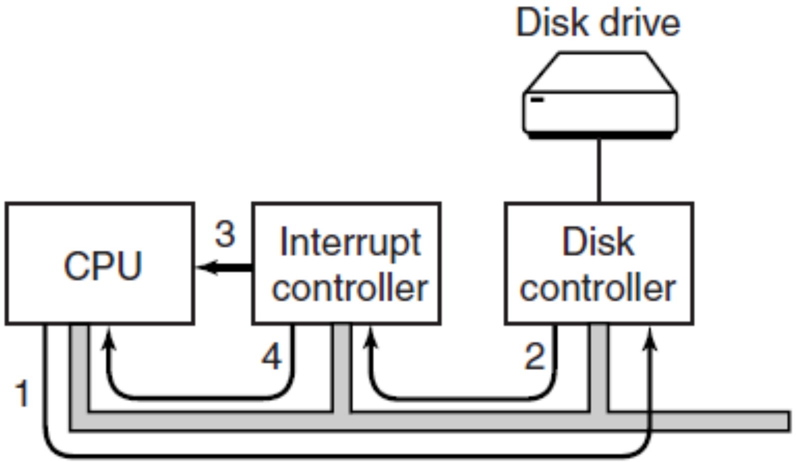


```c
#include <stdio.h>
#include <unistd.h>
#include <wiringPi.h>

#define PIN_BUTTON 18

int main (int argc, char **argv)
{
    wiringPiSetupGpio();

    pinMode(PIN_BUTTON, INPUT);
    pullUpDnControl(ButtonPin, PUD_UP);

    printf("Button pin has been setup.\n");

    while (1)
    {
        if (digitalRead(PIN_BUTTON) == 0) {
            printf("Button pressed - 0\n");
        }
        else
        {
            printf("Button pressed - 1\n");
        }
        delay(50);
    }
}
```

- Continuously check the status

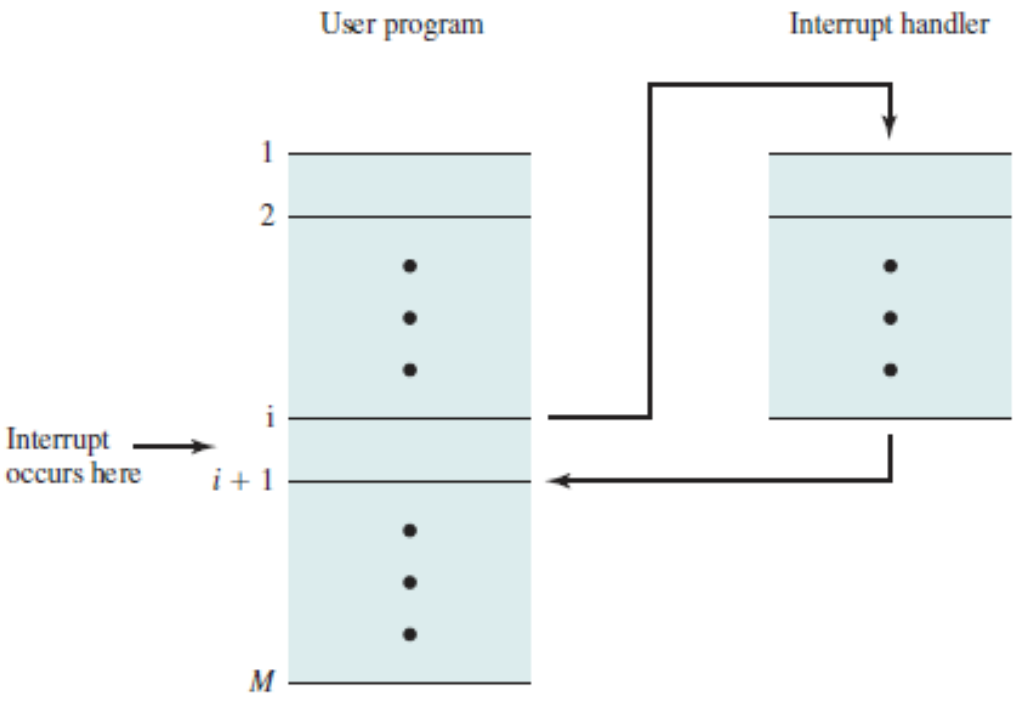
Digital Input – Interrupt

1. The driver tells the controller what to do by writing into its device registers. The controller then starts the device.
2. I/O signals the interrupt controller chip using certain bus lines.
3. It asserts a pin on the CPU chip
4. The interrupt controller puts the number of the device on the bus



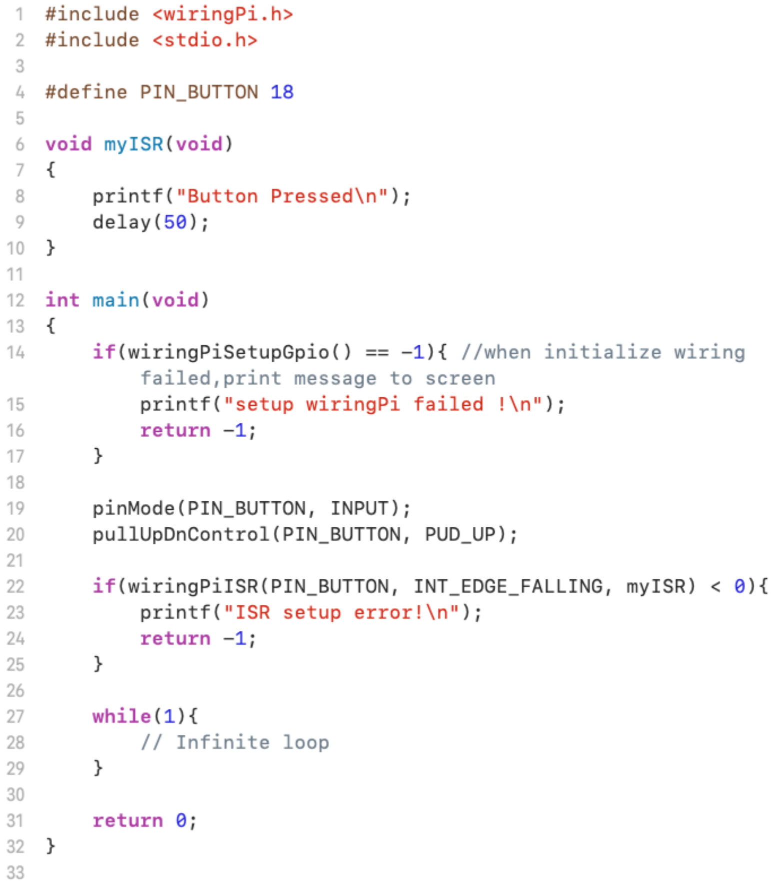
Interrupt Handler

- An interrupt suspends the normal sequence of execution.
- When the interrupt processing is completed, execution resumes



Button Press - Interrupt

- Register for the Interrupt Service Routine
- myISR()
- It is called when the interrupt happens



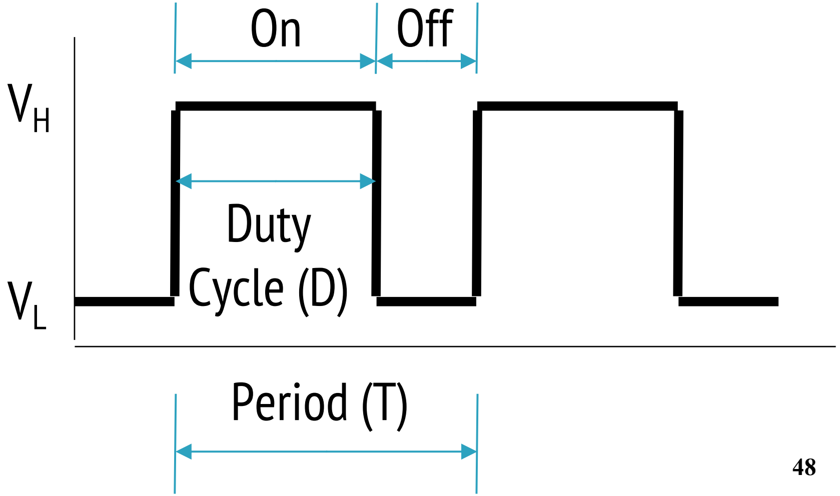
```c
#include <wiringPi.h>
#include <stdio.h>

#define PIN_BUTTON 18

void myISR(void)
{
    printf("Button Pressed\n");
    delay(50);
}

int main(void)
{
    if(wiringPiSetupGpio() == -1){ //when initialize wiring failed,print message to screen
        printf("setup wiringPi failed !\n");
        return -1;
    }

    pinMode(PIN_BUTTON, INPUT);
    pullUpDnControl(PIN_BUTTON, PUD_UP);

    if(wiringPiISR(PIN_BUTTON, INT_EDGE_FALLING, myISR) < 0){
        printf("ISR setup error!\n");
        return -1;
    }

    while(1){
        // Infinite loop
    }

    return 0;
}

```

Analog Output

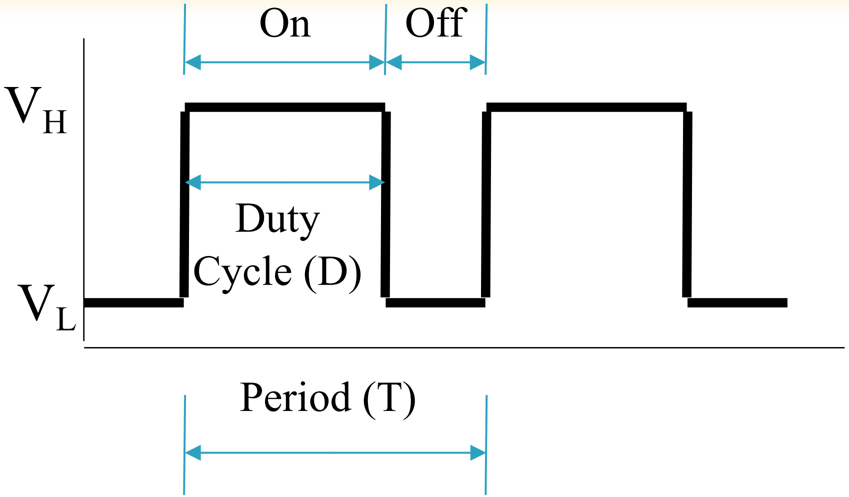
- Pulse Width Modulation (PWM)
  - Technique that conforms a signal width, generally pulses
  - The general purpose is to control power delivery
  - The on-off behavior changes the average power of signal
  - Output signal alternates between on and off within a specified period.
  - If signal toggles between on and off quicker than the load, then the load is not affected by the toggling

PWM – Duty Cycle

- A measure of the time the modulated signal is in its “high” state
- Generally recorded as the percentage of the signal period where the signal is considered on



Duty Cycle Formulation



Duty Cycle is determined by:

$$Duty\ Cycle = \frac{On\ Time}{Period} \times 100\%$$

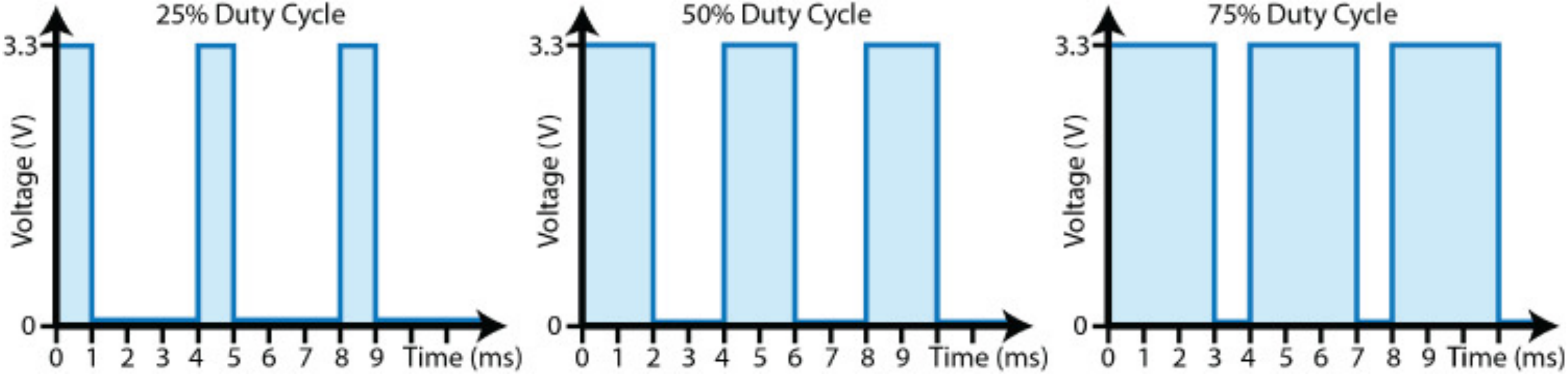
*Average value of a signal can be found as:

$$\bar{y} = \frac{1}{T}\int_0^T f(t)dt$$

$$V_{avg} = D \cdot V_H + (1 - D) \cdot V_L$$

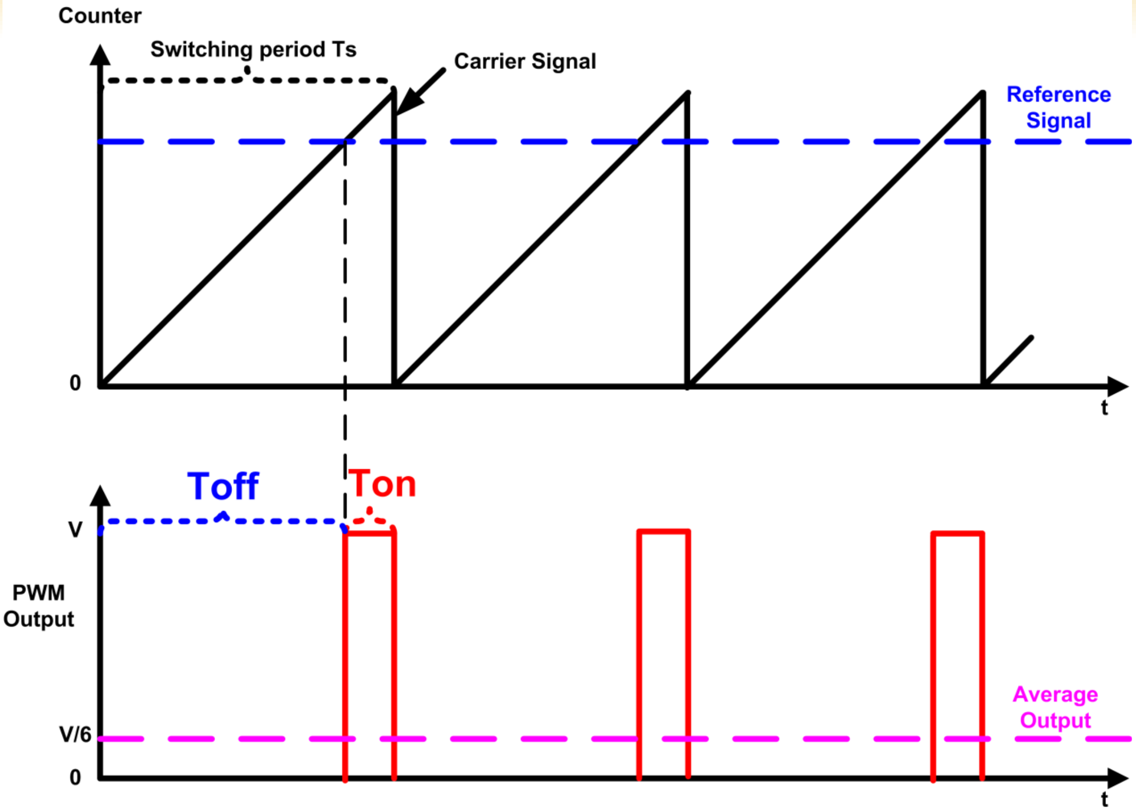
*In general analysis, V_L is taken as zero volts for simplicity.

PWM Duty Cycle



PWM Mode

- Counter counts up to the range provided
- When the counter value is higher than set value, output is high

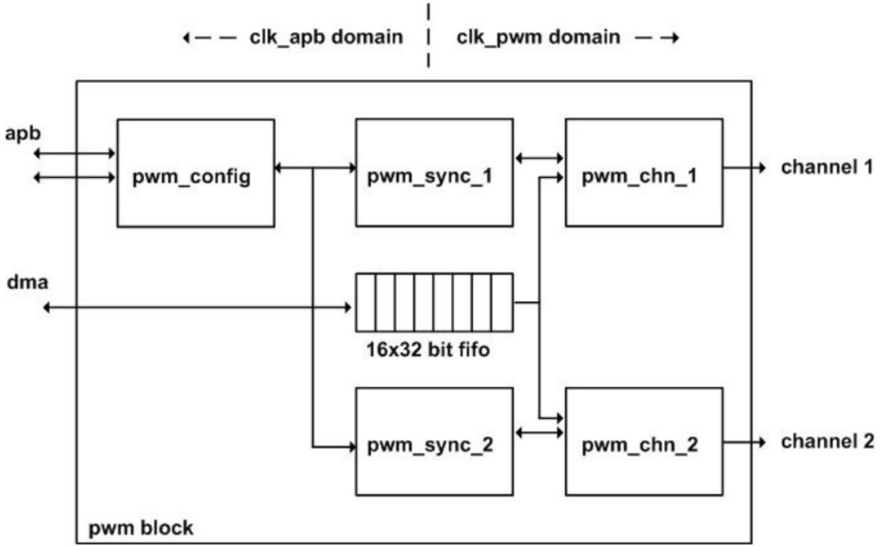


PWM Duty Cycle Calculation

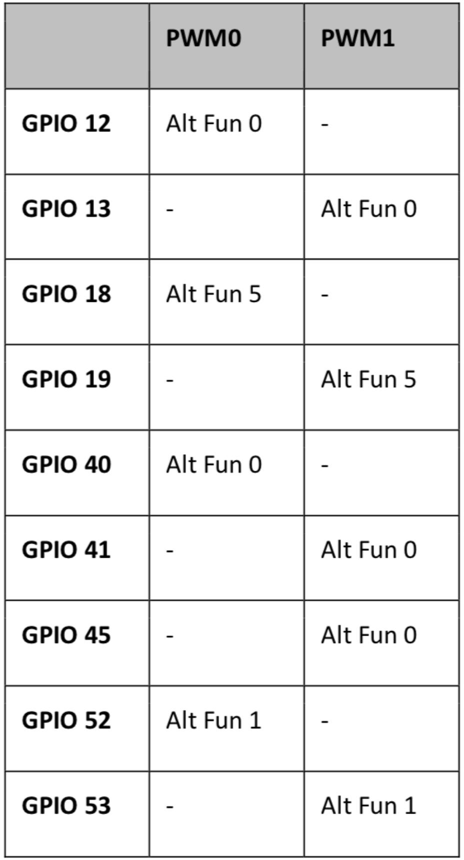
- The PWM device on the RPi is clocked at a fixed base-clock frequency of 19.2 MHz
- Integer divisor and range values are used to tailor the PWM frequency according to application requirements
- $f_{PWM} = 19.2MHz/(divisor \times range)$
- If f_{PWM} is 10KHz (0.01MHz), and range is 128,
  - $divisor = \frac{19.2MHz}{f_{PWM} \times range} = 15$
- Smaller value in range results in poor resolution

PWM Controller

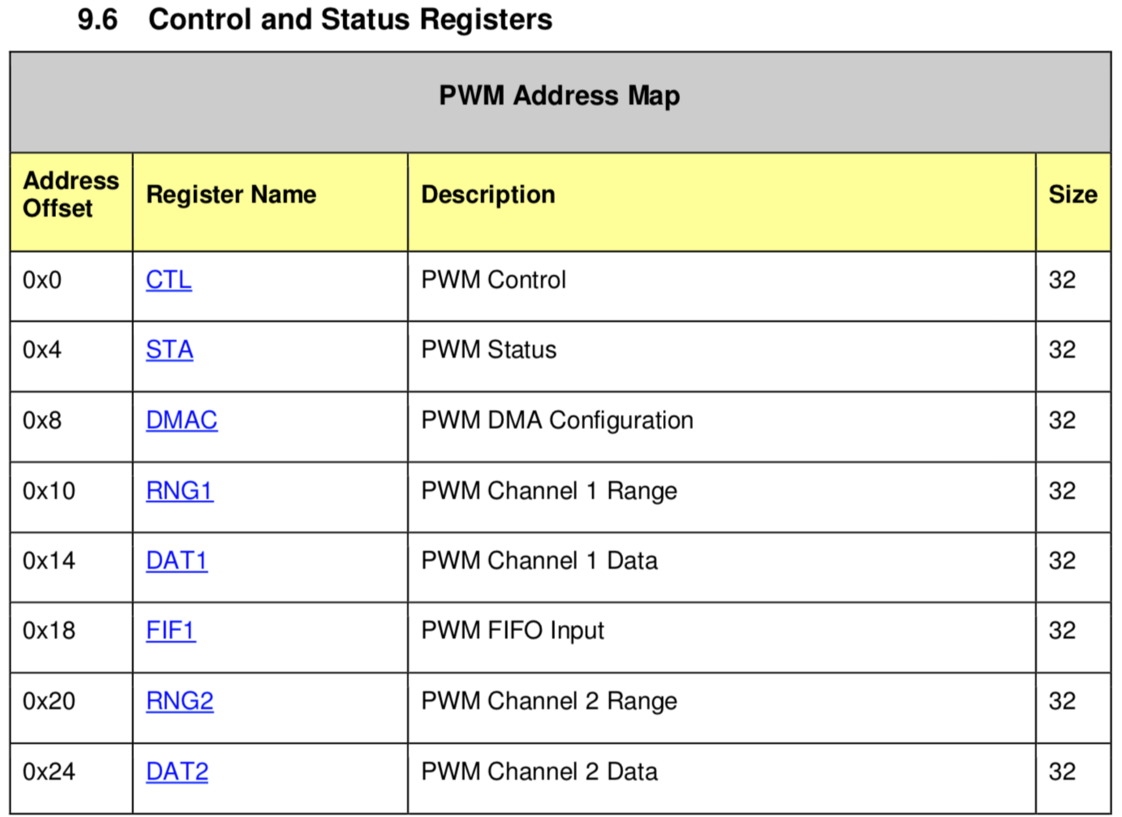
- Two independent output bit-streams, clocked at a fixed frequency



PWM0 and PWM1 Map



	PWM0	PWM1
GPIO 12	Alt Fun 0	-
GPIO 13	-	Alt Fun 0
GPIO 18	Alt Fun 5	-
GPIO 19	-	Alt Fun 5
GPIO 40	Alt Fun 0	-
GPIO 41	-	Alt Fun 0
GPIO 45	-	Alt Fun 0
GPIO 52	Alt Fun 1	-
GPIO 53	-	Alt Fun 1

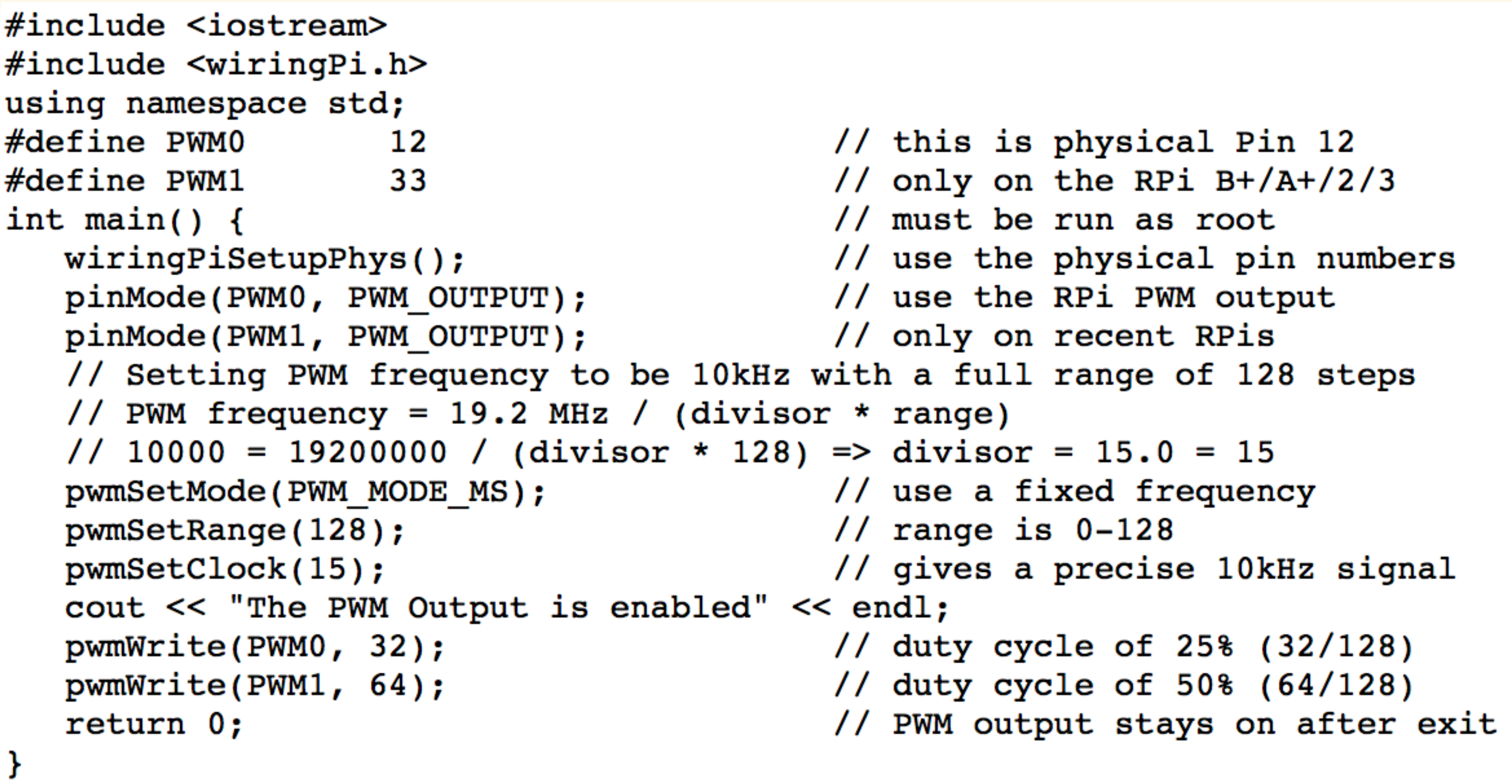


9.6 Control and Status Registers

PWM Address Map

Address Offset	Register Name	Description	Size
0x0	CTL	PWM Control	32
0x4	STA	PWM Status	32
0x8	DMAC	PWM DMA Configuration	32
0x10	RNG1	PWM Channel 1 Range	32
0x14	DAT1	PWM Channel 1 Data	32
0x18	FIF1	PWM FIFO Input	32
0x20	RNG2	PWM Channel 2 Range	32
0x24	DAT2	PWM Channel 2 Data	32

exploringPi/chp06/wiringPi/pwm.cpp



```
#include <iostream>
#include <wiringPi.h>
using namespace std;
#define PWM0       12                    // this is physical Pin 12
#define PWM1       33                    // only on the RPi B+/A+/2/3
int main() {                             // must be run as root
   wiringPiSetupPhys();                  // use the physical pin numbers
   pinMode(PWM0, PWM_OUTPUT);            // use the RPi PWM output
   pinMode(PWM1, PWM_OUTPUT);            // only on recent RPis
   // Setting PWM frequency to be 10kHz with a full range of 128 steps
   // PWM frequency = 19.2 MHz / (divisor * range)
   // 10000 = 19200000 / (divisor * 128) => divisor = 15.0 = 15
   pwmSetMode(PWM_MODE_MS);              // use a fixed frequency
   pwmSetRange(128);                     // range is 0-128
   pwmSetClock(15);                      // gives a precise 10kHz signal
   cout << "The PWM Output is enabled" << endl;
   pwmWrite(PWM0, 32);                   // duty cycle of 25% (32/128)
   pwmWrite(PWM1, 64);                   // duty cycle of 50% (64/128)
   return 0;                             // PWM output stays on after exit
}
```