ECE 471 – Embedded Systems
Lecture 11

Vince Weaver

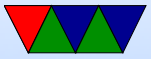
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

27 September 2024

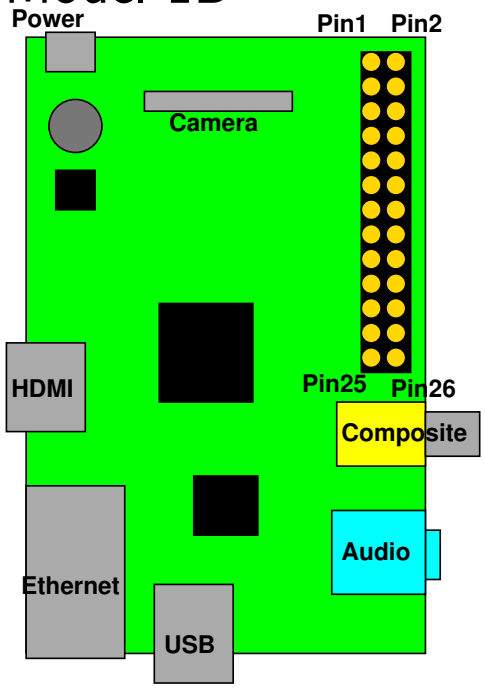
Announcements

- HW#4 will be posted
- Will require an LED, a breadboard, some resistors and some jumper wires.
  I'll provide jumper wires
  If you need breadboard I'll try to track some down
- Remember to comment your code!
- Also be sure your code doesn't crash!

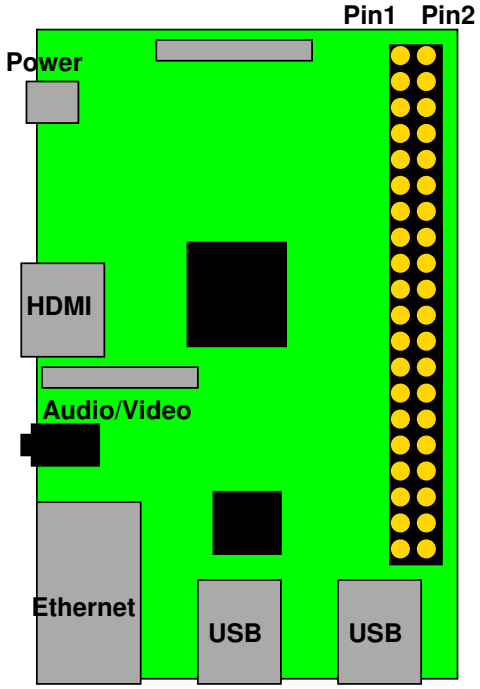


Brief Overview of the Raspberry Pi Board

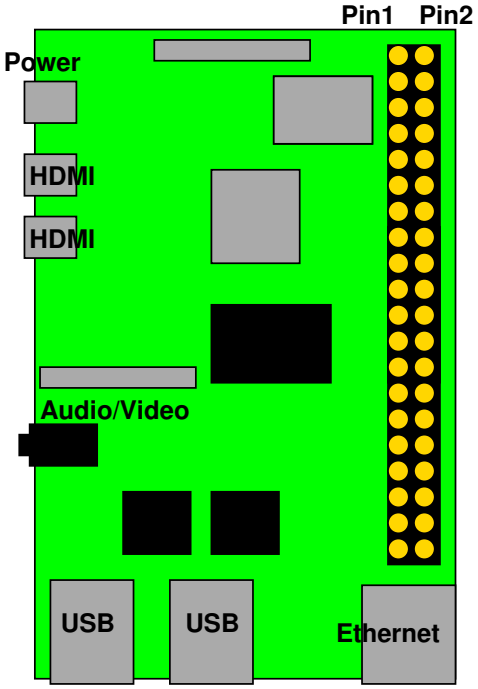
Model 1B

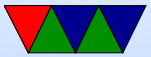


1B+/2B/3B/3B+



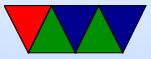
Model 4B/(Pi5 similar)





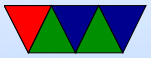
GPIOs

- General-Purpose I/O
- Really useful for embedded systems (does your desktop/laptop have any?)
- Simplest case, just a pin you can do output (set it high/low) or input (read to see if it's high/low)
- There's also a lot more complex things you can do with them.



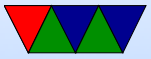
Rasp-pi GPIO Header

- Model 1B has 17 GPIOs (out of 26 pins)
  1B+/2B/3B/4B has 26 (out of 40)
- 3.3V signaling logic. Need level shifter if want 5V or 1.8V
- Linux by default configures some for other purposes (serial, i2c, SPI)



Rasp-pi Header

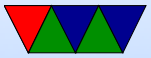
3.3V	1	2	5V
GPIO2 (SDA)	3	4	5V
GPIO3 (SCL)	5	6	GND
GPIO4 (1-wire)	7	8	GPIO14 (UART_TXD)
GND	9	10	GPIO15 (UART_RXD)
GPIO17	11	12	GPIO18 (PCM_CLK)
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10 (MOSI)	19	20	GND
GPIO9 (MISO)	21	22	GPIO25
GPIO11 (SCLK)	23	24	GPIO8 (CE0)
GND	25	26	GPIO7 (CE1)
ID_SD (EEPROM)	27	28	ID_SC (EEPROM)
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21



Review: How to enable GPIO on STM32L

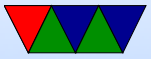
A lot of read/modify/write instructions to read current register values and then to shift/mask to write out updated bitfields.

- Enable GPIO Clock
- Set output mode for GPIO.
- Set GPIO type.
- Set pin clock speed.
- Set pin pull-up/pull-down
- Set or clear GPIO pin.

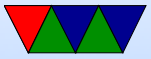


“Bare Metal” on BCM2835 (Rasp-pi)

- Documented in BCM2835 ARM Peripherals Manual
- 53 GPIOs (not all available on board)
- Can use Wiring-Pi or libbcm2835 if you need speed
- Similar to how done on STM32L... but we have an operating system

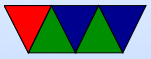


Letting the OS handle it for you

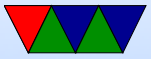


"Old" Linux sysfs GPIO interface

- See the Appendix to these notes for details
- Deprecated with Linux 4.8 in October 2016
- Still there; supposedly to be removed in 2020
- Benefits
  - Could call from shell script
- Downsides
  - String based, had to remember to convert from ASCII
  - If crash/forget to close, GPIO left active
  - Multiple processes at same time, conflict



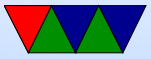
- Some things (like open-drain) couldn't be set
- Slow, especially if writing multiple (lots of syscalls)



“New” Linux GPIO interface

- Introduced with Linux 4.8 (October 2016)
- New way uses ioctls and structs
  - Faster
  - Automatically releases GPIO when program ends

    Note: this can confuse, if you turn an LED on but immediately exit, might look like not working
  - Can set parameters (i.e. pull-up/down) couldn’t before



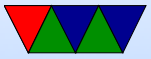
GPIOD utils

- If you have `gpiod` utilities installed you can get info
- If not installed, if on network you can `sudo apt-get install gpiod`
- `gpiodetect`

```
gpiochip0 [pinctrl-bcm2835] (54 lines)
gpiochip1 [raspberrypi-exp-gpio] (8 lines)
```

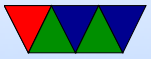
- `gpioinfo`

```
gpiochip0 - 54 lines:
 line   0:      unnamed       unused   input  active-high
 line   1:      unnamed       unused   input  active-high
 ...
```



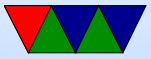
There is a library

- Some Linux interfaces (perf, ALSA,) assume library
- libgpiod library
- We will avoid it
  - embedded systems: may not be room for a library
  - sometimes good to code directly to operating system



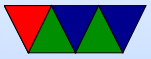
A few low-level Linux Coding Instructions

- These routines are C bindings for the low-level I/O syscalls used by the Linux kernel
- Linux, “everything is a file”
- Operate on file descriptors, just integer value that the kernel uses to track open files internally
- You might remember your program starts with three of these 0 (stdin), 1 (stdout), and 2 (stderr)



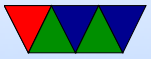
Opening a File

- `int fd=open("/path/to/file",O_RDWR);`
- Note can have many many options `O_RDONLY` and others can be or-ed on `O_TRUNC`, etc.
- Also optional third argument with permissions
- CHECK FOR ERRORS! For open, -1 is an error (remember 0 can be a valid fd value)
  If error happens, can check the `errno` value to see what kind (e.g. EINVAL) and can use `perror()` or `printf("%s",strerror(errno));` to get value



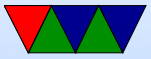
Closing a File with `close()`

- What happens if you forget to close before exiting program?
  No problem, OS will clean up when existing
- What happens if you forget to close in long-running program?
  Can have a file descriptor leak, will run out eventually
- Do you need to check the return value from `close()` for errors?



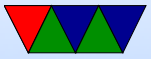
Reading/Writing a File

- `read(int fd, char *buffer, int count)`
  Read count bytes from file fd into pointed-to buffer
- `write(int fd, char *buffer, int count)`
  Write count bytes from buffer into file fd
- Return value is either -1 on error or else number of bytes successfully read/written



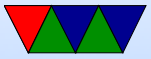
Out of band access

- `ioctl()`
- Takes filedescriptor, an IOCTL number defined in the kernel somewhere, and usually a pointer of some sort
- Not very type safe



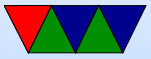
Other syscalls

- There are a huge number, too many to get into here
- Some are useful when manipulating files (`llseek()`)
- Others just use file descriptors because why not `perf_event_open()`

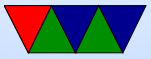


What about fopen()?

- You might have used the FILE *, fopen(), fwrite(), fclose()
- These are buffered. Instead of writing directly to OS the C library buffers a number of writes together and writes all data at once when hit threshold
- Usually good for performance (fewer syscalls), bad if want to write data RIGHT NOW like on embedded system
- Underneath, these routines call into open/read/write/close



anyway

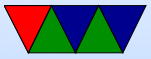


gpio – Opening Device

```
#include "linux/gpio.h"

    int fd,rv;

    /* open first gpio device read/write, check for error */
    fd=open("/dev/gpiochip0",O_RDWR);
    if (fd<0) printf("Error opening %s\n",strerror(errno));
```

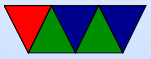


gpio – configure request structure

```
// struct gpiohandle_request {
//       __u32 lineoffsets[GPIOHANDLES_MAX];
//       __u32 flags;
//       __u8 default_values[GPIOHANDLES_MAX];
//       char consumer_label[32];
//       __u32 lines;int fd;}

// configuration values we can or together
// BIAS values added later

// #define GPIOHANDLE_REQUEST_INPUT(1UL << 0)
// #define GPIOHANDLE_REQUEST_OUTPUT(1UL << 1)
// #define GPIOHANDLE_REQUEST_ACTIVE_LOW(1UL << 2)
// #define GPIOHANDLE_REQUEST_OPEN_DRAIN(1UL << 3)
// #define GPIOHANDLE_REQUEST_OPEN_SOURCE(1UL << 4)
// #define GPIOHANDLE_REQUEST_BIAS_PULL_UP (1UL << 5)
// #define GPIOHANDLE_REQUEST_BIAS_PULL_DOWN       (1UL << 6)
// #define GPIOHANDLE_REQUEST_BIAS_DISABLE (1UL << 7)
```



gpio – actually do request

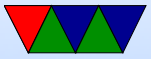
```
struct gpiohandle_request req;

/* clear out struct */
memset(&req,0,sizeof(struct gpiohandle_request));

req.flags = GPIOHANDLE_REQUEST_OUTPUT;  // want it to be output
req.lines =1;          // can group multiple lines together
req.lineoffsets[0] =17;       // gpio number we want
req.default_values[0]=0;      // default value
strcpy(req.consumer_label, "ECE471");  // helpful label

/* get a handle for our requested config */
rv = ioctl(fd, GPIO_GET_LINEHANDLE_IOCTL, &req);
if (rv<0 ) printf("Error ioctl %s\n",strerror(errno));

// req.fd is now a handle for this gpio setup
```



gpio – write/change value of gpio17

```
// struct gpiohandle_data {
//       __u8 values[GPIOHANDLES_MAX]; }

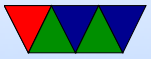
   struct gpiohandle_data data;

   /* set output to 0 */
   data.values[0]=0;

   /* send this data struct to the handle for gpio17 we created */
   rv=ioctl(req.fd,GPIOHANDLE_SET_LINE_VALUES_IOCTL,&data);
   if (rv<0) printf("Error setting value %s\n",strerror(errno));

   /* set output to 1 */
   data.values[0]=1;

   /* send this data struct to the handle for gpio17 we created */
   rv=ioctl(req.fd,GPIOHANDLE_SET_LINE_VALUES_IOCTL,&data);
   if (rv<0) printf("Error setting value %s\n",strerror(errno));
```



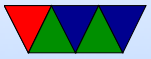
gpio – read current value of gpio17

```
struct gpiohandle_data data;

/* clear out our data */
memset(&data, 0, sizeof(data));

/* read current value into data struct */
rv = ioctl(req.fd, GPIOHANDLE_GET_LINE_VALUES_IOCTL, &data);
if (rv<0) printf("Error! %s\n",strerror(errno));

/* print the result */
printf("%d\n",data[0]);
```



gpio – (optional) getting interface info

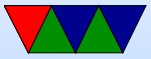
```
#include "linux/gpio.h"

// struct gpiochip_info {
//       char name[32];
//       char label[32];
//       __u32 lines; }

   struct gpiochip_info chip_info;

   /* ask for chipinfo from open file descriptor, put in chip_info struct */
   rv=ioctl(fd,GPIO_GET_CHIPINFO_IOCTL,&chip_info);
   if (rv<0 ) printf("Error ioctl %s\n",strerror(errno));

   /* print summary of what was returned */
   printf("Found %s, %s, %d lines\n",
                chip_info.name,chip_info.label,chip_info.lines);
```



gpio – (optional) get info about line gpio17

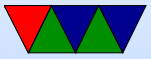
```
// struct gpioline_info {
//       __u32 line_offset;
//       __u32 flags;
//       char name[32];
//       char consumer[32]; }

   struct gpioline_info line_info;

   /* clear struct to 0 before using it */
   /* kernel might not like uninitialized values */
   memset(&line_info,0,sizeof(line_info));

   /* get line info for gpio17 */
   line_info.line_offset=17;   // set GPIO17
   rv=ioctl(fd,GPIO_GET_LINEINFO_IOCTL,&line_info);
   if (rv<0) printf("Error ioctl %s\n",strerror(errno));

   /* print summary of what we learned */
   printf("Offset %d, flags %x, name %s, consumer %s\n",line_info.line_offset,
         line_info.flags, line_info.name, line_info.consumer);
```

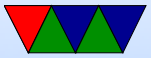


Delay in Linux

- Can we Busy delay (like in ECE271)?
  `for(i=0;i<1000000;i++);`
  Harder to do in C. Why?
  Compiler optimizes away.

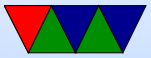
- `usleep()` puts process to sleep for a number of microseconds. But can have issues if want exact delay. Why? OS potentially context switches every 100ms.

- Other ways to implement: Set up PWM? Timers?



Waiting for Input

- Busy loop. Bad, burns CPU / power
- usleep() in loop. Can delay response time.
- Interrupt when ready! poll()



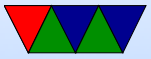
gpio input using interrupts / poll()

NOTE: You don't need to do this for the homework, this is just here in case you are curious how to do it.

```
// struct gpioevent_request {
//         __u32 lineoffset;
//         __u32 handleflags;
//         __u32 eventflags;
//         char consumer_label[32];
//         int fd; }

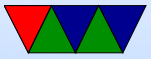
// struct gpioevent_data {
//         __u64 timestamp;
//         __u32 id; }

    struct gpioevent_request ereq;
    struct gpioevent_data edata;
    struct pollfd pfd;
    ssize_t rd;
```



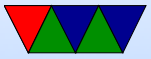
```
/* do this instead of request_line */
memset(&ereq,0,sizeof(struct gpioevent_request));
req.lineoffset=17;
req.handleflags = GPIOHANDLE_REQUEST_INPUT;
req.eventflags = GPIOEVENT_REQUEST_BOTH_EDGES;
rv = ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &req);

pfd.fd = ereq.fd;
pfd.events = POLLIN | POLLPRI;
rv = poll(&pfd, 1, 1000);   // 1000 = timeout 1s
if (rv >0) {
   rd = read(req.fd, &event, sizeof(event));
   printf("Timestamp: %lld   id %d\n",
                             edata.timestamp,edata.id);
}
```

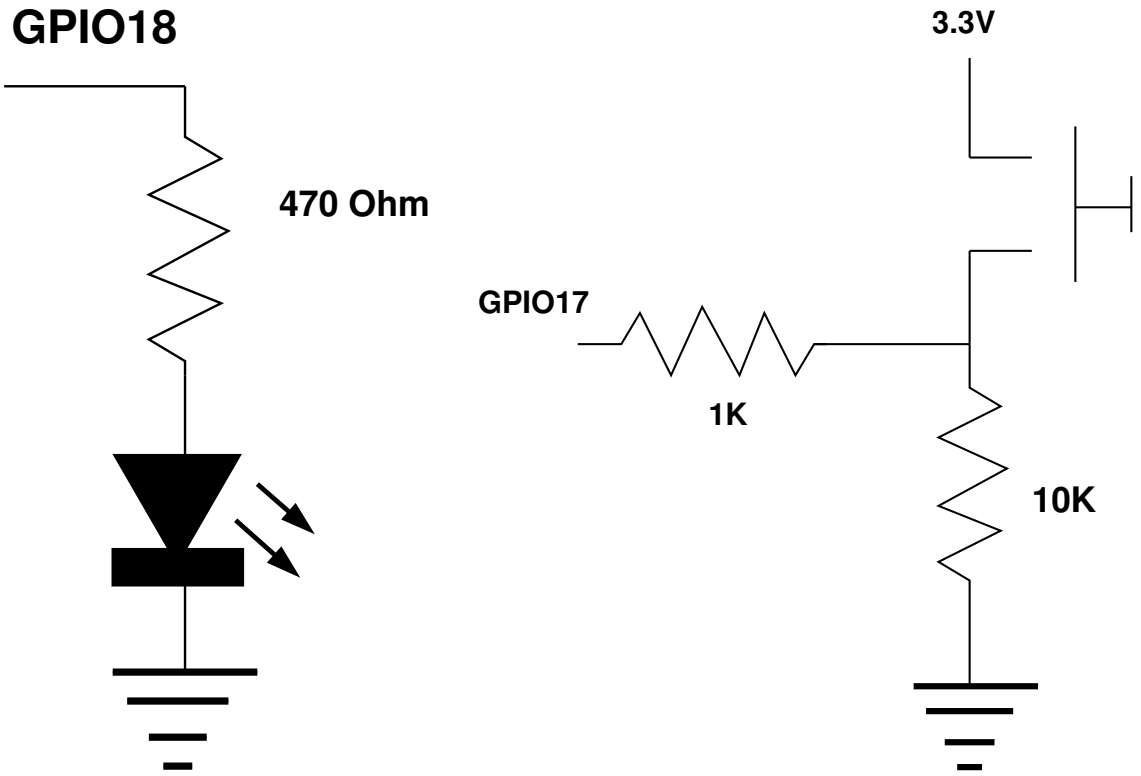


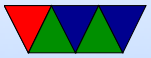
Requirements!

- Comment your code!
- Check for errors, print message and exit() if error.



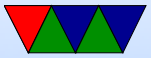
Circuit





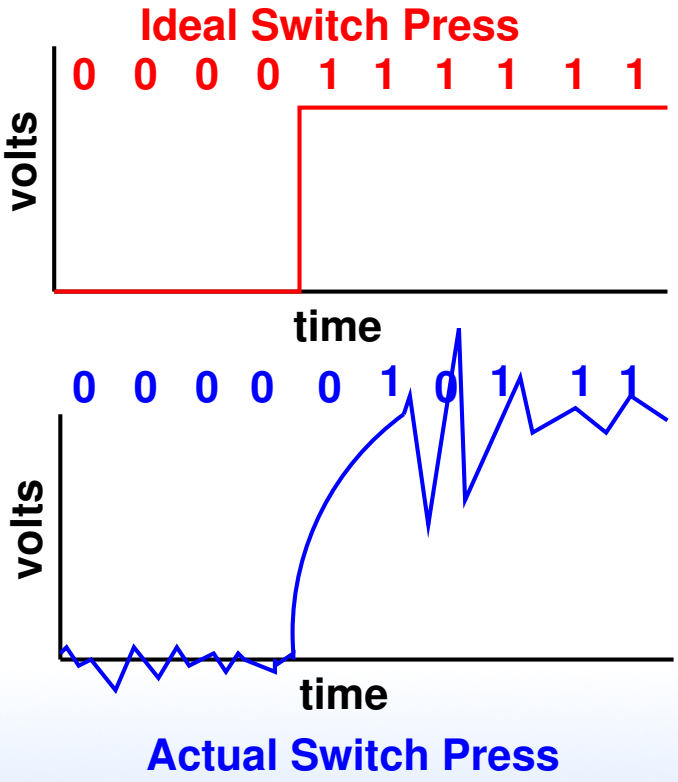
Circuit Discussion

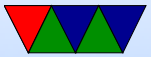
- Pull-up / Pull-down resistor. Why?
- Why the extra 1k resistor? (avoid short if set to output by accident)



Debouncing! Noisy Switches

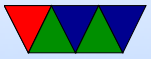
- Noisy switches, have to debounce



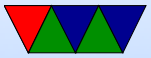


Debouncing!

- Can you fix in hardware? Capacitors?
- Can you fix in software? No built-in debounce like on STM32L
- Algorithms
  - NOTE: for a proper debounce you need to do at least two reads and only report a change if they match! Sleeping after one read is not a debounce!
  - Wait until you get X consecutive values before changing

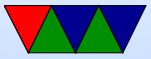


- Get new value, wait short time and check again



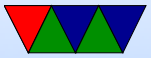
Permissions!

- Linux has permissions on devices to prevent accidental or not-allowed access
- By default your user should be in the “gpio” group to allow you access
- If you get permission denied you can try running your code with “sudo” to over-ride the permissions
- That is generally considered unsafe, you can possibly add yourself to the gpio group to avoid this: `sudo addgroup vince gpio`



- What should your code do if gets permission denied error?

  Not crash, ideally.

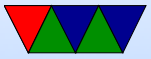


Bypassing Linux for speed

http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/

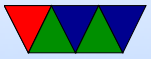
Trying to generate fastest GPIO square wave.

shell	gpio util	40Hz
shell	sysfs	2.8kHz
Python	WiringPi	28kHz
Python	RPi.GPIO	70kHz
C	sysfs (vmw)	400kHz
C	WiringPi	4.6MHz
C	libbcm2835	5.4MHz
C	Rpi Foundation “Native”	22MHz



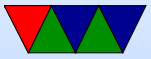
Appendix: Linux userspace sysfs interface

THIS IS INCLUDED FOR HISTORICAL PURPOSES

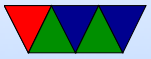


Linux GPIO interface

- `Documentation/gpio/sysfs.txt`
- sysfs and string based



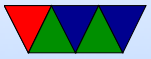
A few low-level Linux Coding Instructions



Enable a GPIO for use

To enable GPIO 17:
write “17” to /sys/class/gpio/export
To disable GPIO 17:
write “17” to /sys/class/gpio/unexport

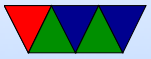
```
char buffer[10];
fd=open("/sys/class/gpio/export",O_WRONLY);
if (fd<0) fprintf(stderr,"\tError enabling\n");
strcpy(buffer,"17");
write(fd,buffer,2);
close(fd);
```



Set GPIO Direction

To make GPIO 17 an input:
write “in” to /sys/class/gpio/gpio17/direction
To make GPIO 17 an output:
write “out” to /sys/class/gpio/gpio17/direction

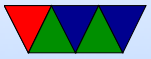
```
fd=open("/sys/class/gpio/gpio17/direction",O_WRONLY);
if (fd<0) fprintf(stderr,"Error!\n");
write(fd,"in",2);
close(fd);
```



Write GPIO Value

To write value of GPIO 17:
write `/sys/class/gpio/gpio17/value`

```
fd=open("/sys/class/gpio/gpio17/value",O_WRONLY);
if (fd<0) fprintf(stderr,"Error opening!\n");
write(fd,"1",1);
close(fd);
```



Read GPIO Value

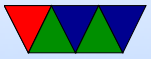
To read value of GPIO 17:

read /sys/class/gpio/gpio17/value

```
char buffer[16];
fd=open("/sys/class/gpio/gpio17/value",O_RDONLY);
if (fd<0) fprintf(stderr,"Error opening!\n");
read(fd,buffer,16);
printf("Read %c from GPIO17\n",buffer[0]);
close(fd);
```

Note: the value you read is ASCII, not an integer.

Also Note, if reading and you do not close after read you will have to rewind using `lseek(fd,0,SEEK_SET);` after your read.

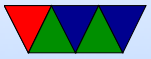


Delay

- Busy delay (like in ECE271).
  `for(i=0;i<1000000;i++);`
  Harder to do in C. Why?
  Compiler optimizes away.

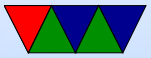
- `usleep()` puts process to sleep for a number of microseconds. But can have issues if want exact delay. Why? OS potentially context switches every 100ms.

- Other ways to implement: Set up PWM? Timers?



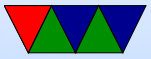
Using fopen instead?

- Need to `fflush()` after writes (linefeed not enough?)
- Need to `frewind()` after reads?



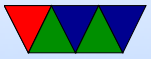
C Pitfalls

- Be careful cut and pasting! Especially the size of strings you are sending with write()
- Know the difference between `'C'` and `"C"`
- Remember the strings we are reading/writing are ASCII '0' and '1' not integers



Waiting for Input

- Busy loop. Bad, burns CPU / power
- usleep() in loop. Can delay response time.
- Interrupt when ready! poll()



GPIO Interrupts on Linux

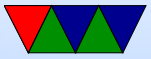
May need a recent version of Raspbian.

First write "rising", "falling", or "both" to /sys/class/gpio/gpio17/edge.

Then open and poll /sys/class/gpio/gpio17/value.

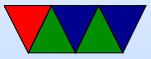
```
struct pollfd fds;
int result;

fd=open("/sys/class/gpio/gpio18/value",O_RDONLY);
fds.fd=fd;
fds.events=POLLPRI|POLLERR;
while(1) {
    result=poll(&fds,1, -1);
    if (result<0) printf("Error!\n");
    lseek(fd,0,SEEK_SET);
read(fd,buffer,1); }
```



Buffered "Stream" I/O

- Slightly higher-level I/O routines in C library
- Buffered I/O
- Still use open/close/read/write underneath
  Can find file descriptor with `fileno()`
- ```
  FILE *f;
  f=fopen("filename","r");
  if (f==NULL) fprintf(stderr,"Error!\n");
  fwrite(buffer,size,members,f);
  fclose(f);
  ```
- Buffered I/O (saves overhead, fewer syscalls, maybe makes I/O faster, but also adds potential delay)
- Use `fflush()` to force buffer flush


  ```

- Use `rewind()` to rewind to beginning of file