

Engineering Best Practices

Test, test, test, and test some more; test as you go

Start from a known working state, take small steps

Make things visible (gdb, printf, logic analyzer)

Methodical, systematic approach. Form hypotheses and perform experiments to confirm

Fast prototyping, embrace automation, one-click build, clean compile, check-in often

Don't let bugs get you down, natural part of engineering, relish the challenge -- you will learn something new!

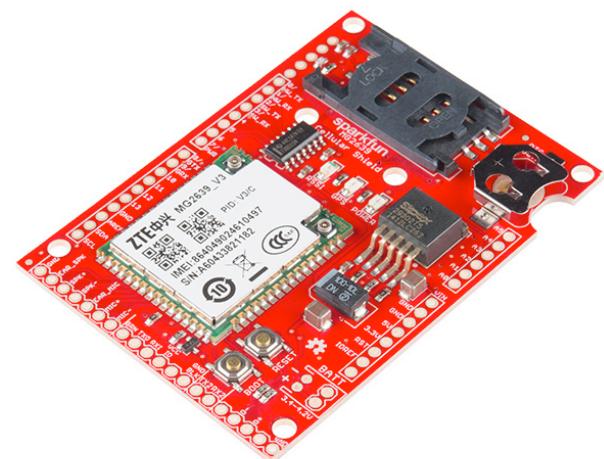
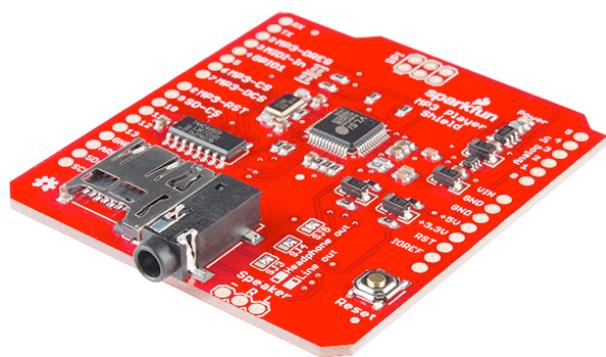
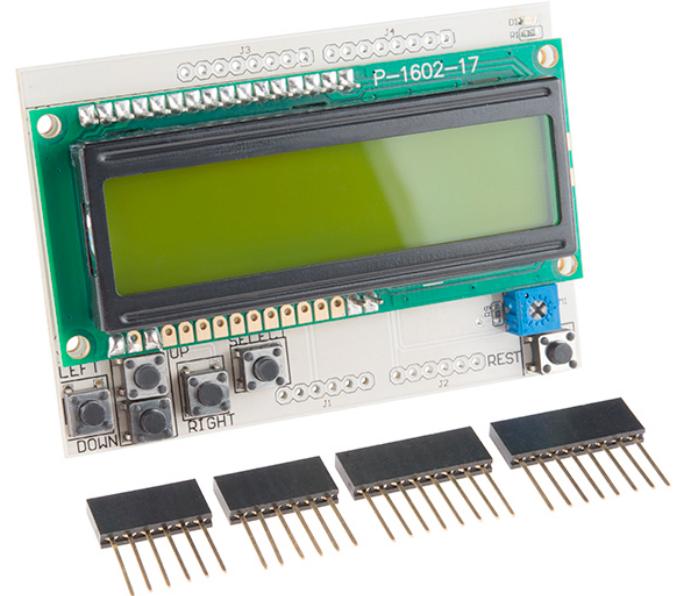
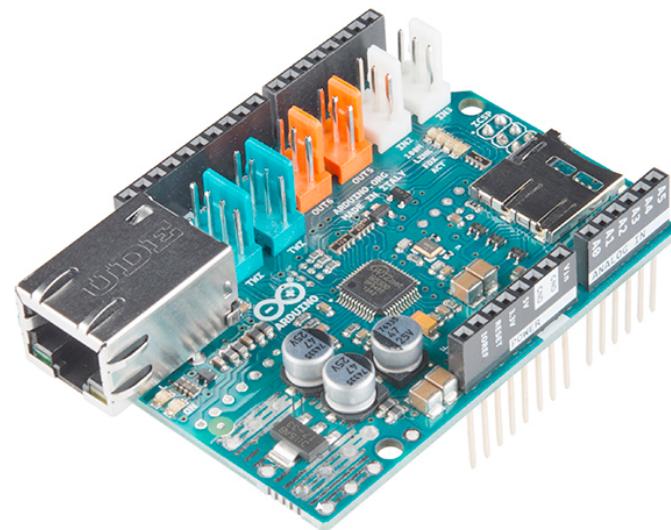
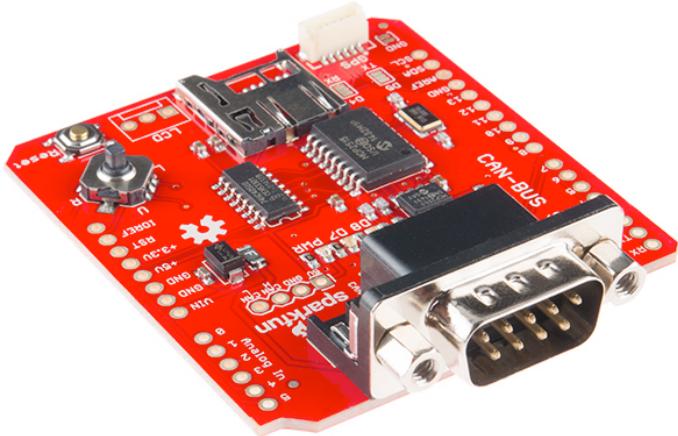
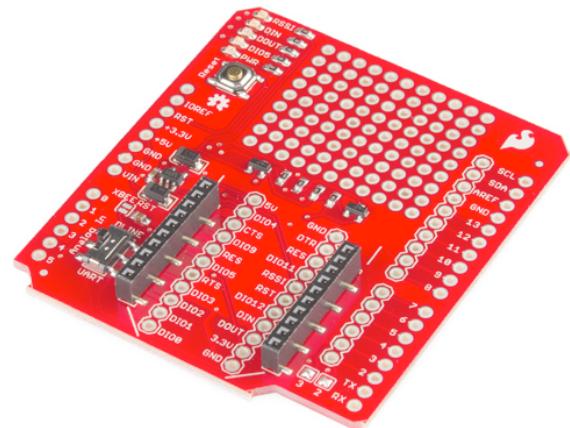
Wellness important! sleep, healthy food, maintain perspective

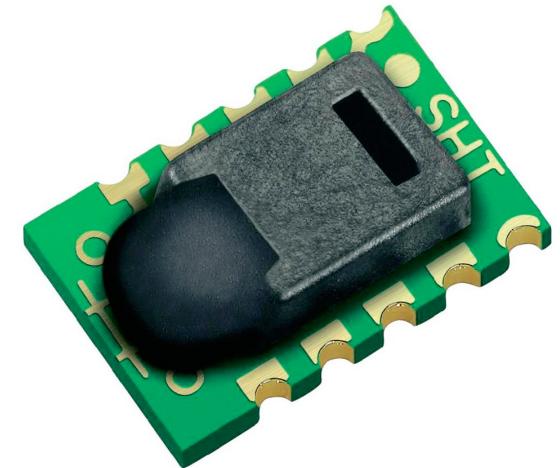
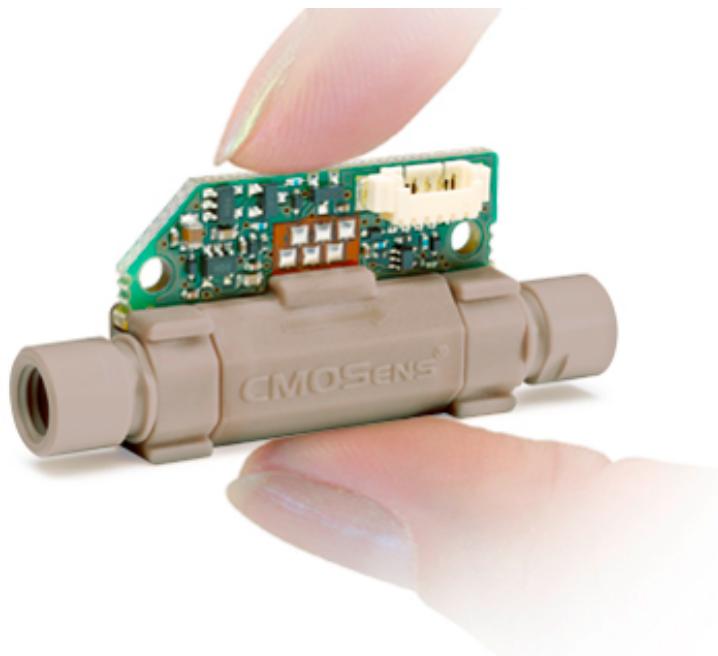
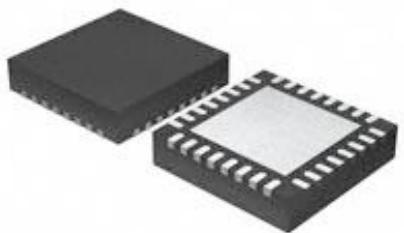
I/O: External Peripherals

PS/2 Keyboard (vs. USB, SPI, I2C)

gpio
timer
uart
printf
malloc
keyboard
shell
fb
gl
console

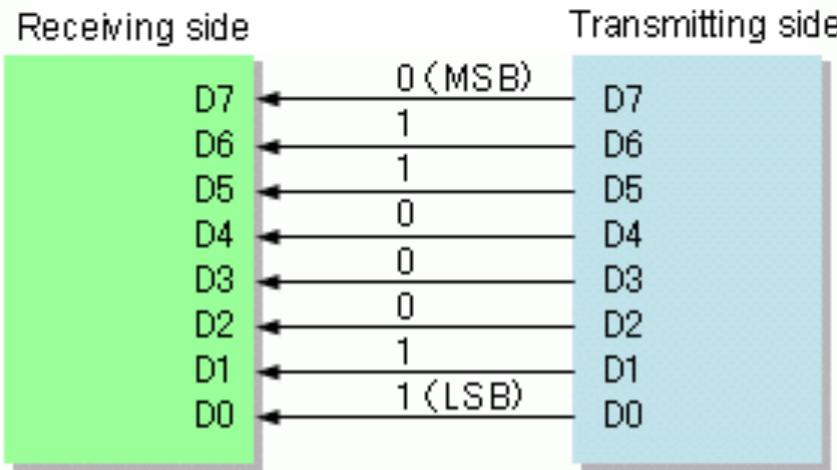




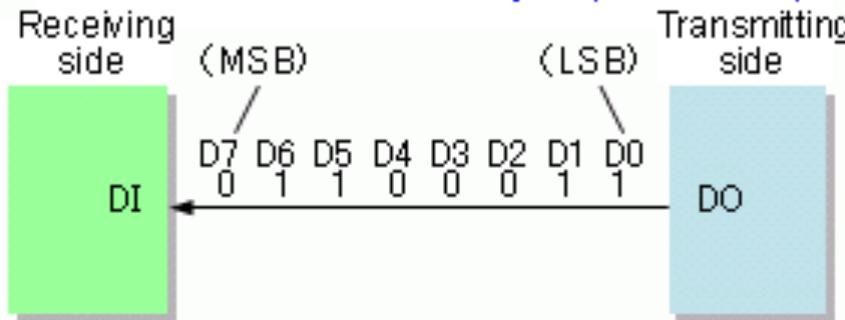


Serial vs. Parallel

Parallel interface example



Serial interface example (MSB first)



UART

Used in your printf & the bootloader

Asynchronous — no clock line

Start bit, (5 to 9) data bits, (0 or 1) parity bit, (1 or 2) stop bits

UART with 8 Databits, 1 Stopbit and no Parity (8-N-1)

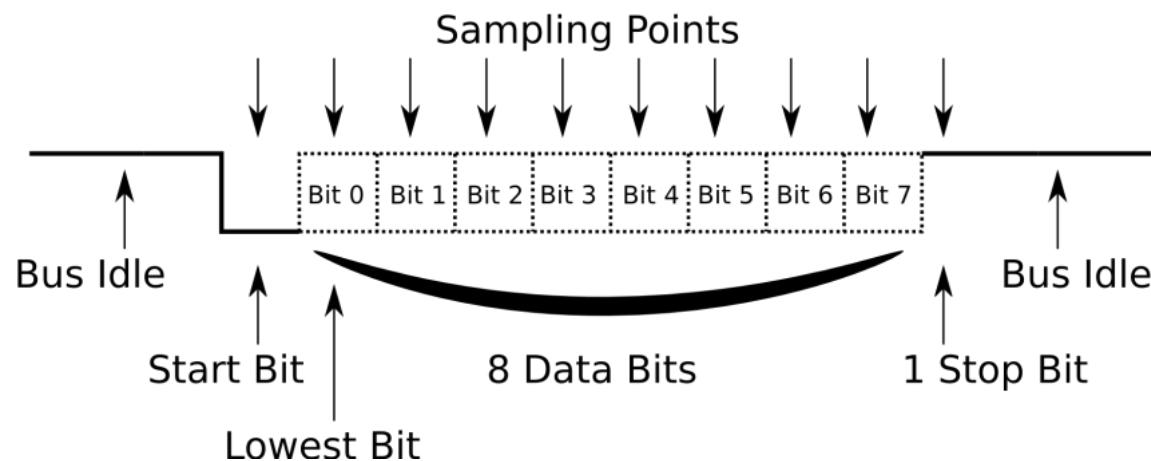
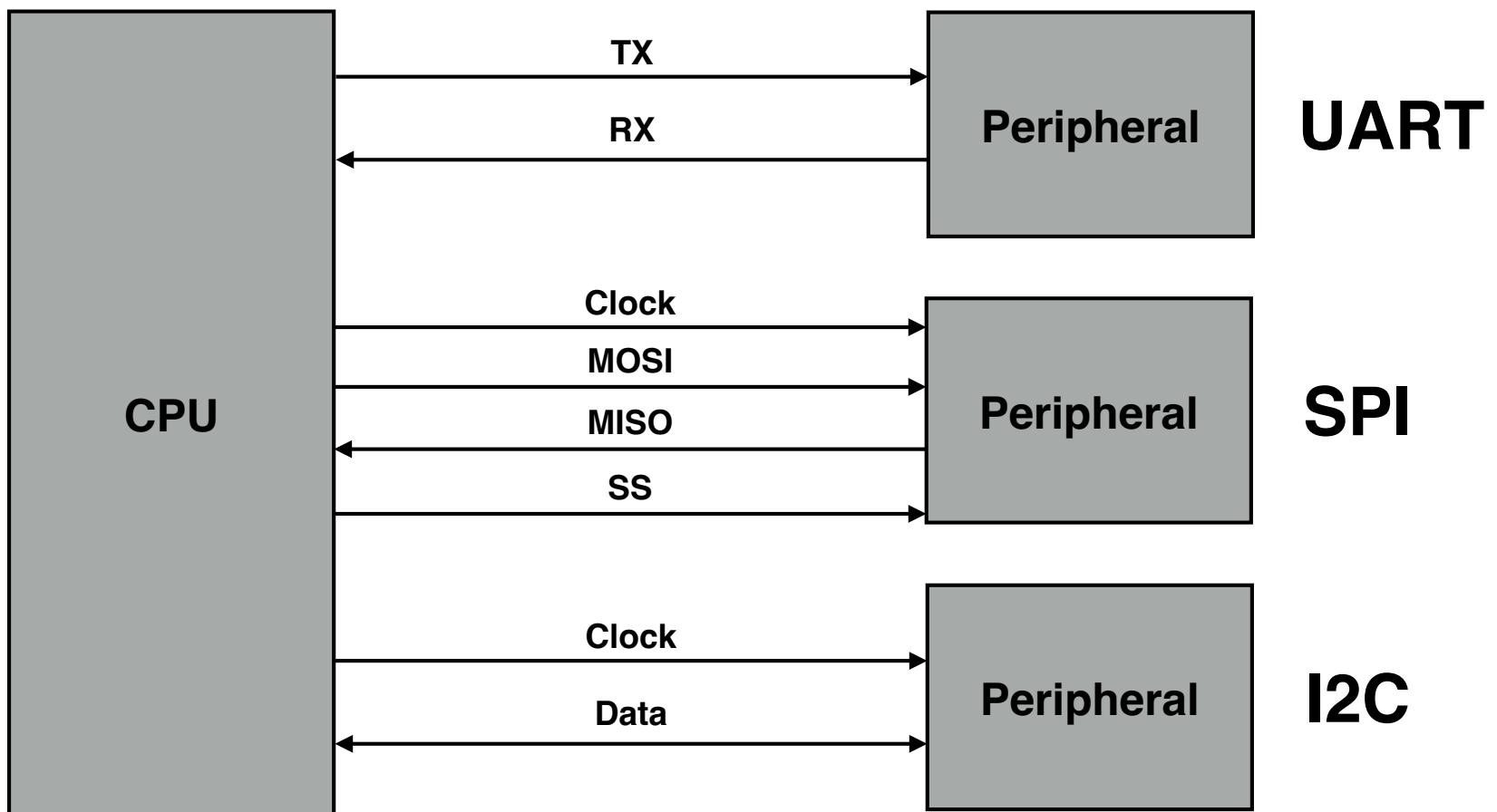


Figure from http://old.myhdl.org/lib/exe/detail.php/projects:rs232image.png?id=projects%3Auart_rs232_receiver_transmitter

Bus Protocols

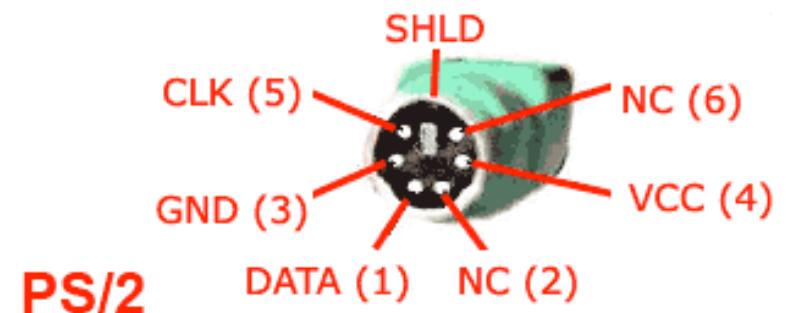




PS/2 Keyboard

- PS/2 is an old serial protocol for keyboards
- Synchronous: CLK and DATA lines

Computer PS/2 ports



PS/2 Protocol

Synchronous protocol, keyboard sends clock

Data changes when clock line goes high

Read data when clock is low

Payload (8-Odd-I)

Start bit (low), 8 data bits (lsb first), 1 parity bit, stop bit (high)

11 bits total

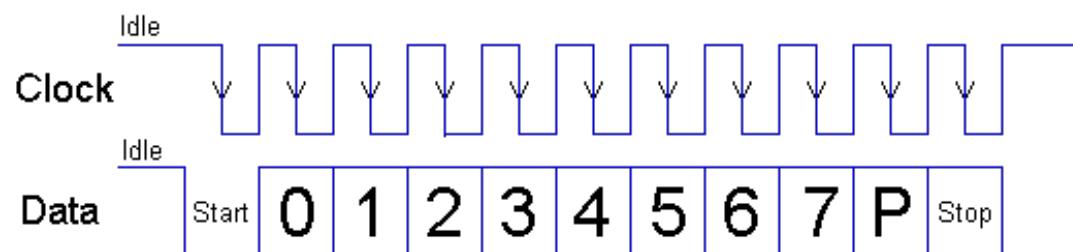


Figure from <http://retired.beyondlogic.org/keyboard/keyboar1.gif>

Parity Bits

Error detection — discard if parity bit wrong

Even parity: parity = XOR of data bits, ensures an even number of 1s (w/ parity bit)

Odd parity: !even parity, ensures an odd number of 1s

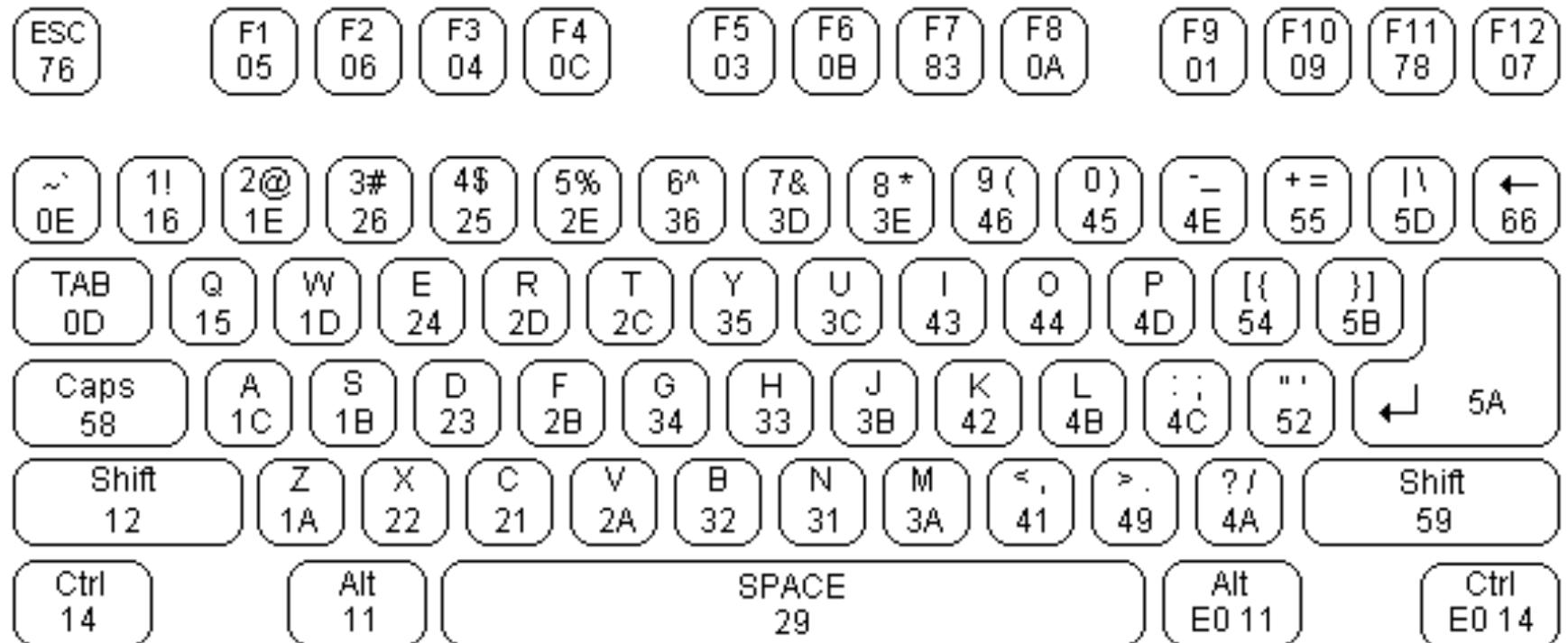
even

data	parity							
1	1	0	1	0	1	1	0	1

odd

data	parity							
1	1	0	1	0	1	1	0	0

Keyboard Data



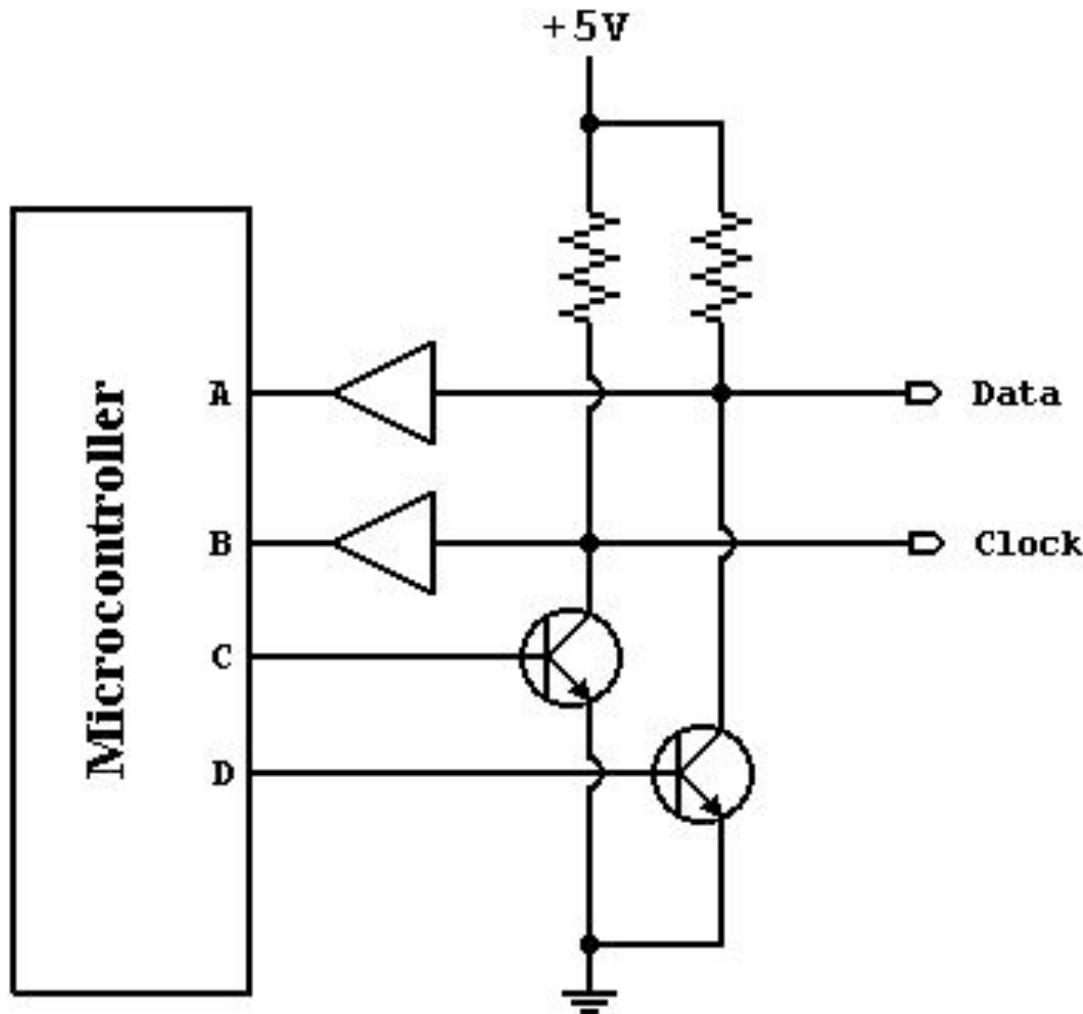
Key	Action	Scan Code
A	Make (down)	0x1C
A	Break (up)	0xF0 0x1C
Shift L	Make (down)	0x12
Shift L	Break (up)	0xF0 0x12

Keyboard Scan Code Demo

scancode/

PS/2 Logic Analyzer Demo

Open Collector



Asserting C will bring clock low

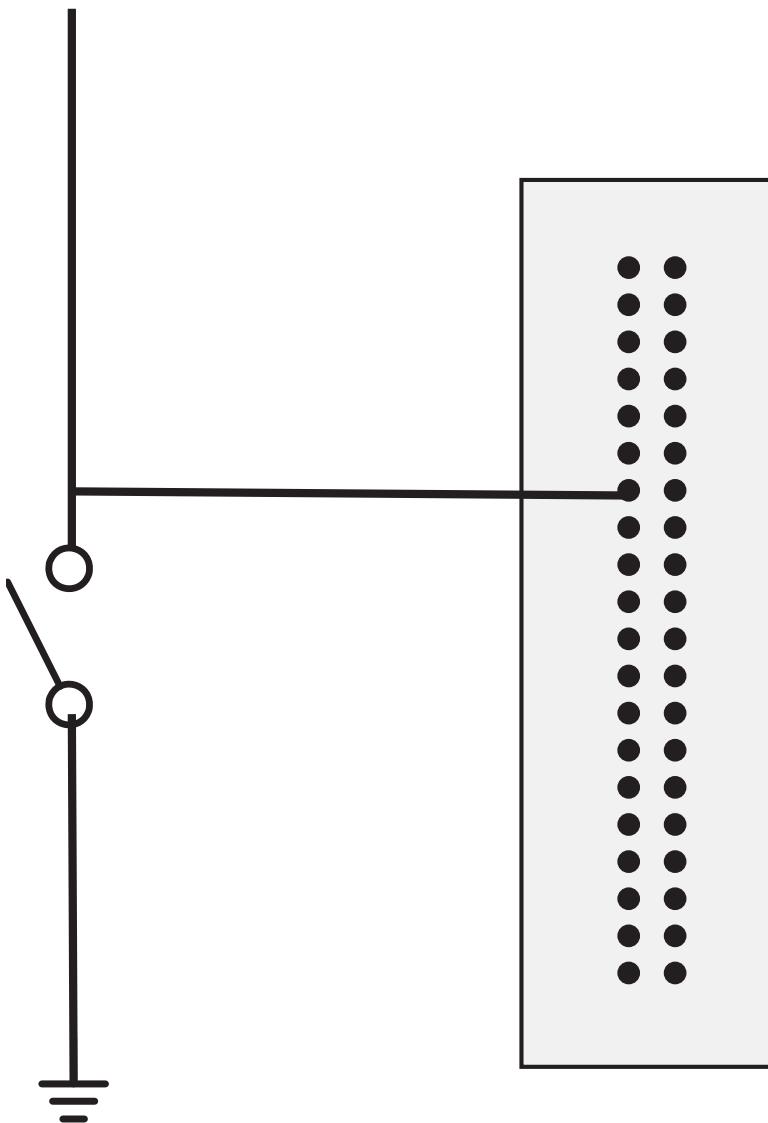
Otherwise goes high from pull-up resistor

Asserting D will bring data low

Otherwise goes high from pull-up resistor

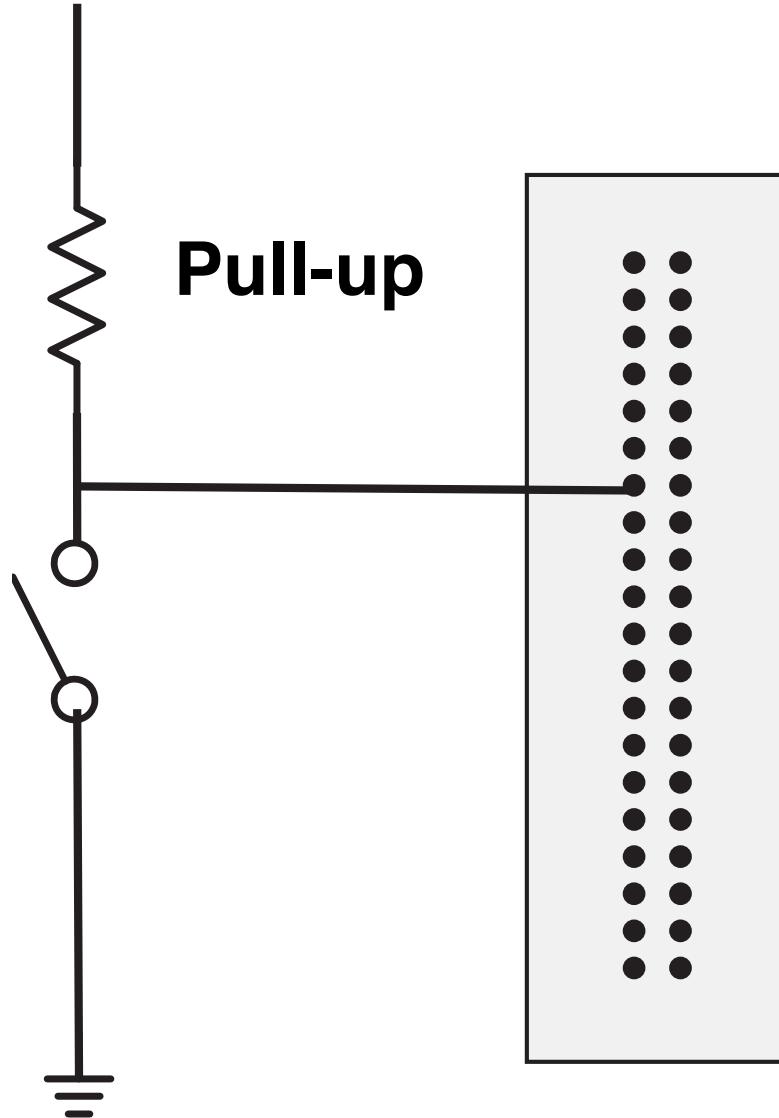
Switch

3.3V



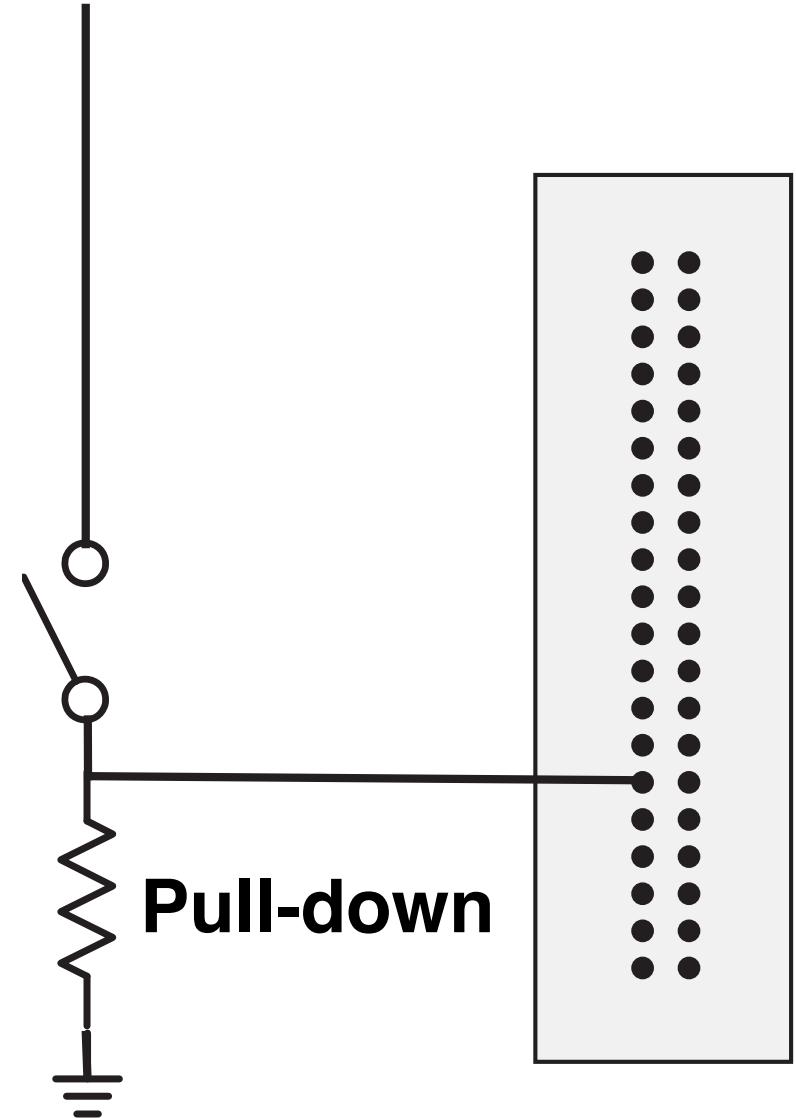
3.3V

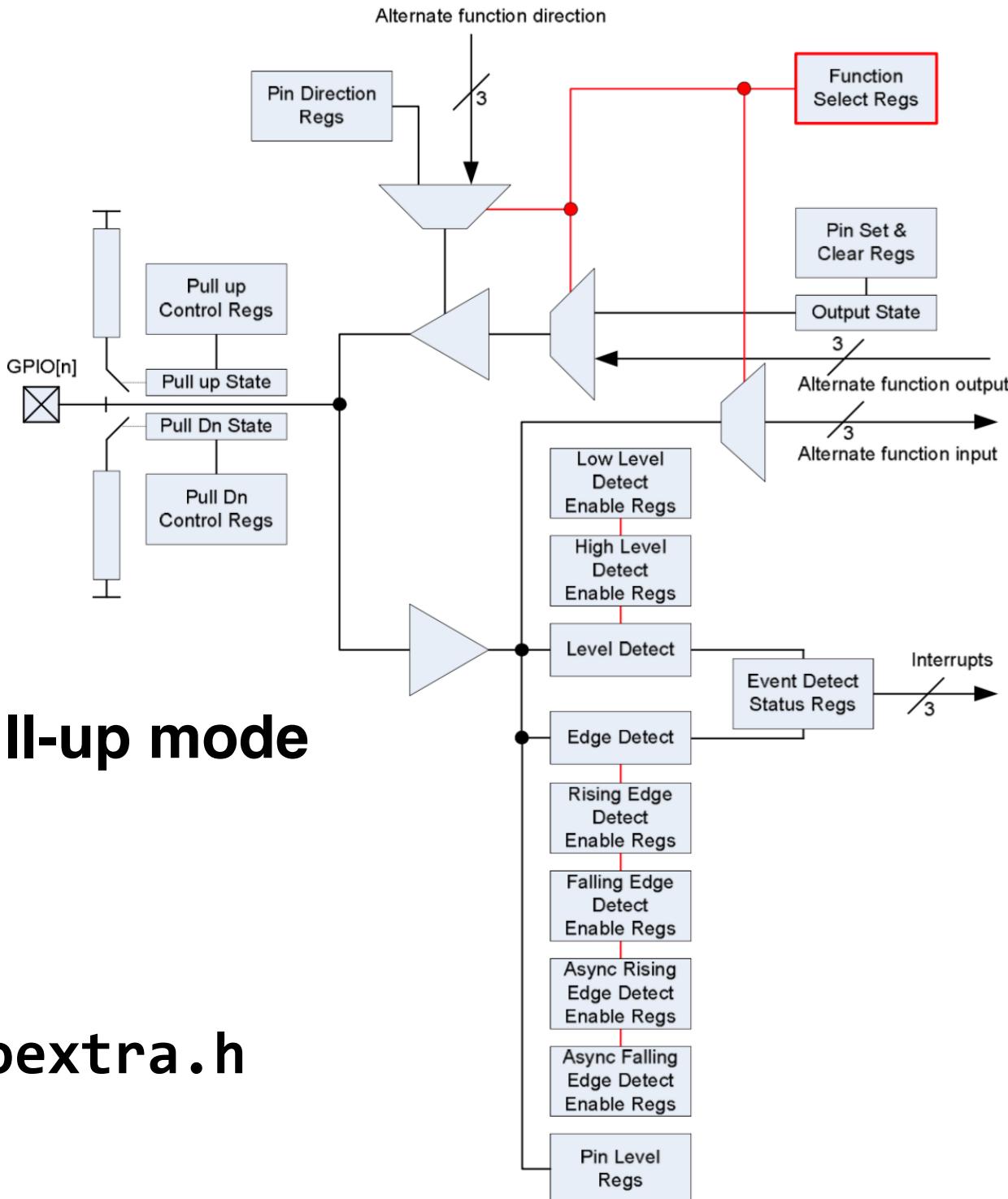
Pull-up



3.3V

Pull-down





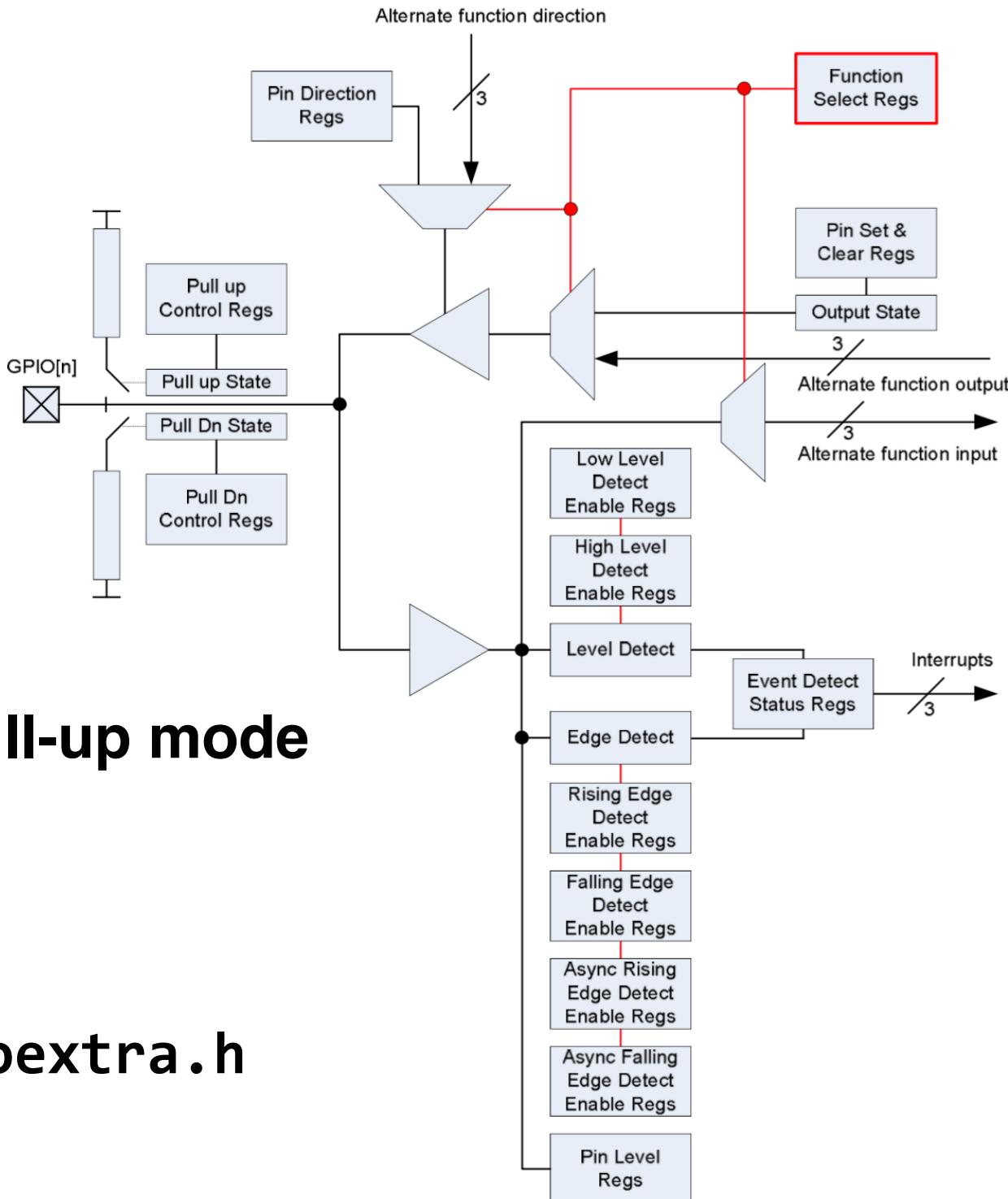
GPIO pull-up mode

See `gpioextra.h`

Figure 6-1 GPIO Block Diagram

Reading PS2 Protocol

ps2/



GPIO pull-up mode

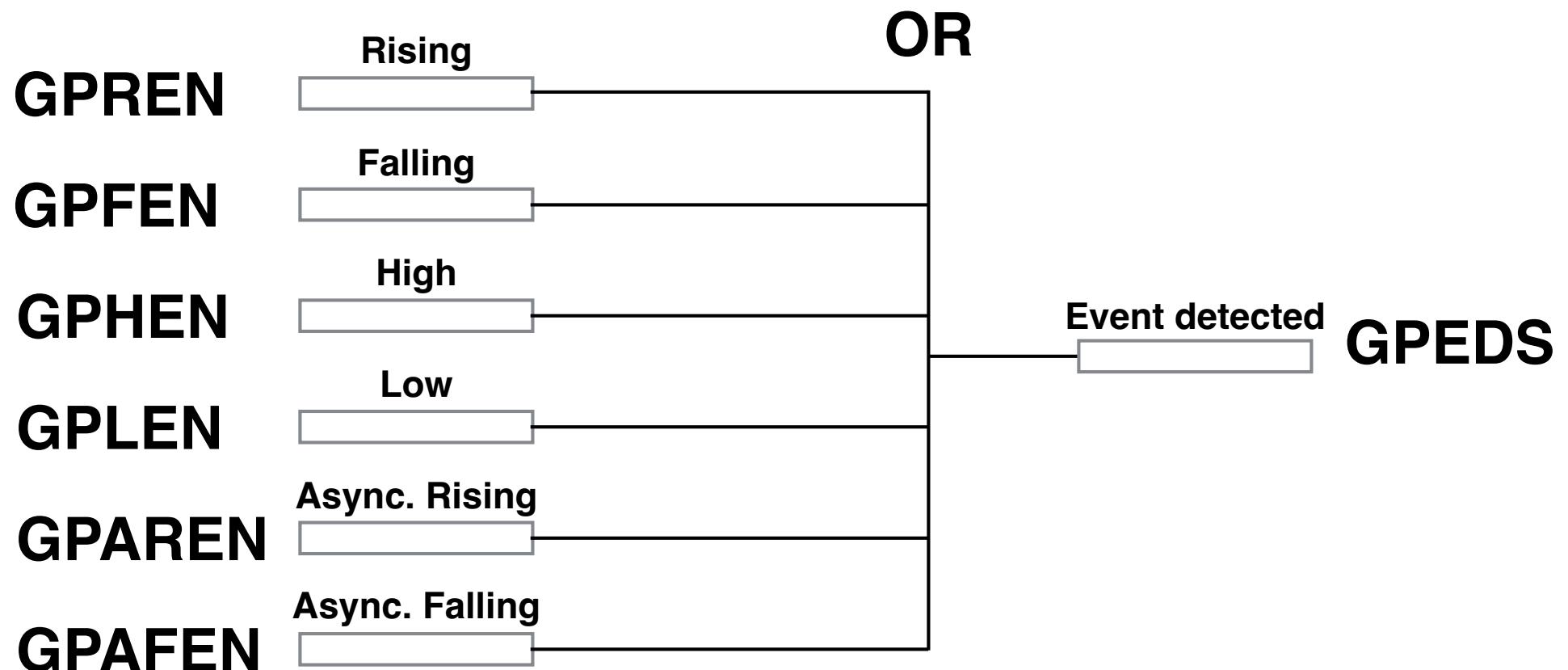
See `gpioextra.h`

Figure 6-1 GPIO Block Diagram

Reading PS2 Protocol

gpioevents/

GPIO Event Detection



GPIO Event Detection

Can detect falling / rising edge, low / high level

Set bit for pin in appropriate GPIO event detect enable register (e.g. GPFEN)

Once enabled, events on that pin will set a bit in the GPIO event detect status register (GPEDS)

Check GPEDS register for event

Clear event by writing 1 to bit in GPEDS

See BCM2835-ARM-Peripherals manual pages 96-100

Keys != Characters

Keyboard scancodes usually converted to ASCII bit stream

Conversion throws away some info (left-shift vs. right-shift, multiple keys pressed, alt/cmd + key, etc.)

Sometimes want the extra info (e.g. games) so interface directly with scancodes

Keys (Scan Codes) ≠ Characters (ASCII)

Scan code numbers ≠ ASCII character codes

- Typically 104 keys
- 127 ASCII character codes

Extra keys

- Special keys - interpreted by the OS or App
 - F1, ..., F12
 - Arrows, insert, delete, home, ...
- Multiple keys with same function
 - Left and right shift, ...
 - Numbers on keypad vs. keyboard

```
% man ascii
```

```
...
```

00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0a	nl	0b	vt	0c	np	0d	cr	0e	so	0f	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1a	sub	1b	esc	1c	fs	1d	gs	1e	rs	1f	us
20	sp	21	!	22	"	23	#	24	\$	25	%	26	&	27	'
28	(29)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[5c	\	5d]	5e	^	5f	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c		7d	}	7e	~	7f	del

```
...
```

```
...
```

Layers (keyboard.h)

`unsigned char keyboard_read_scancode(void)`

returns when receives well-formed scancode

`key_action_t keyboard_read_sequence(void)`

returns key pressed or released

sequence of up to 3 scan codes (includes one or both of
PS2_CODE_RELEASE=0xF0 and PS2_CODE_EXTEND = 0xE0)

`key_event_t keyboard_read_event(void)`

returns key along with modifier key state

`unsigned char keyboard_read_next(void)`

returns ASCII character, modifier and special keys >= 0x90

Keyboard Viewer



Keyboard Viewer

[Shift]



Keyboard Viewer

[CAPS Lock]



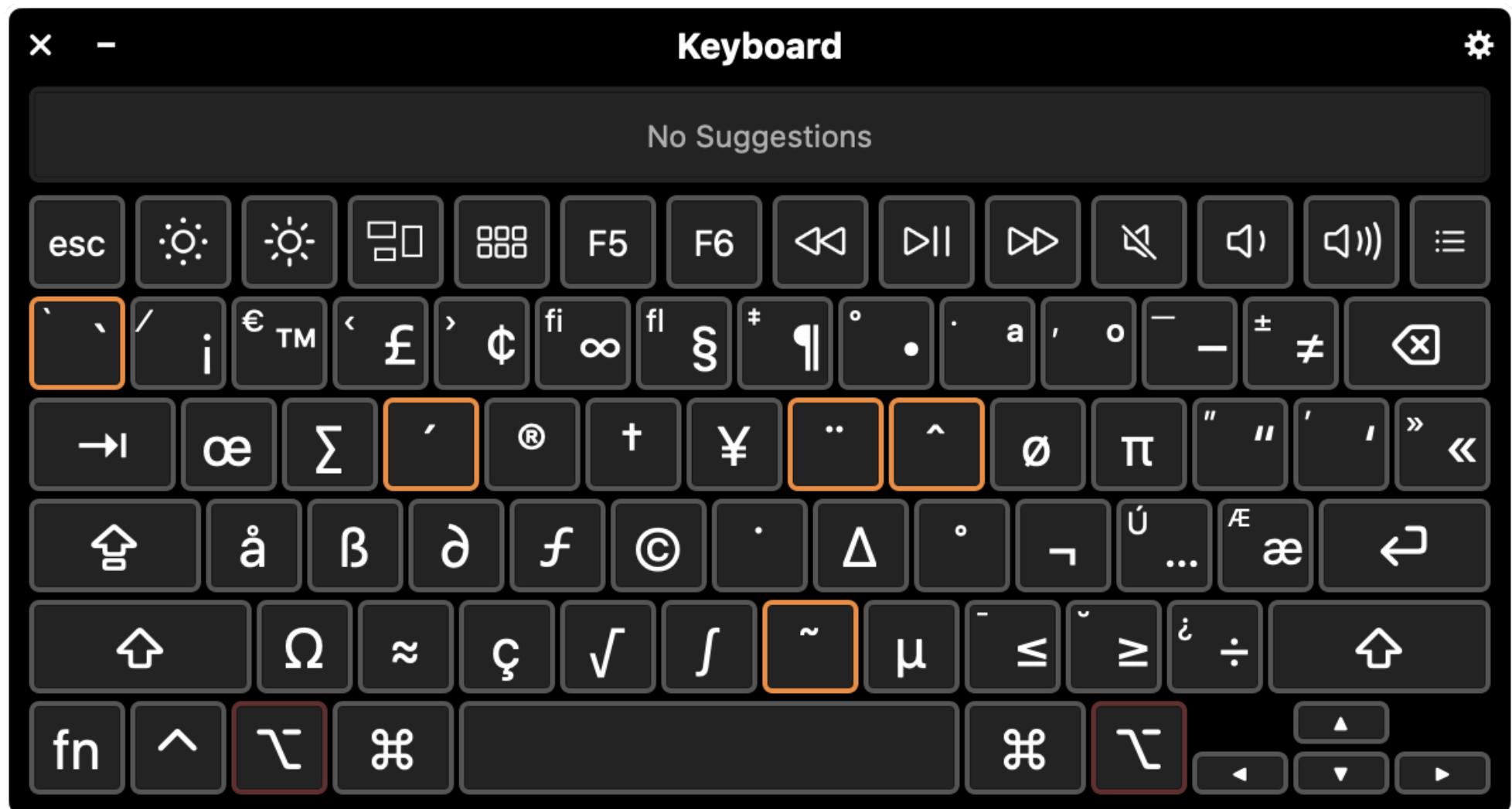
Keyboard Viewer

[CAPS Lock + SHIFT]



Keys ≠ Characters

[OPTION] (orange keys are **dead keys**)



Keys ≠ Characters

[OPTION '']



USB vs. PS/2

USB bus is 480Mbps-2Gbps

PS/2 is 10-16.7 kbps

Gamers preferred PS/2 to early USB keyboards: why?

Polling vs. Interrupts

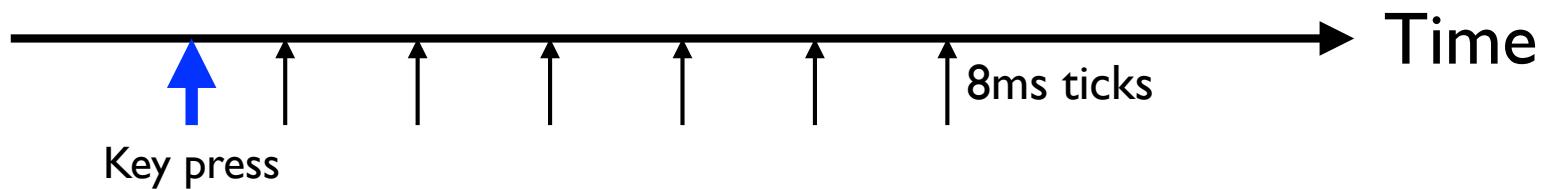
USB protocol uses *polling*: controller asks the keyboard at 125-1000Hz if there are any key presses: 1-8ms delay (old keyboards were 8ms)

Also, USB uses a shared bus: other devices transferring to/from the PC can delay the keyboard

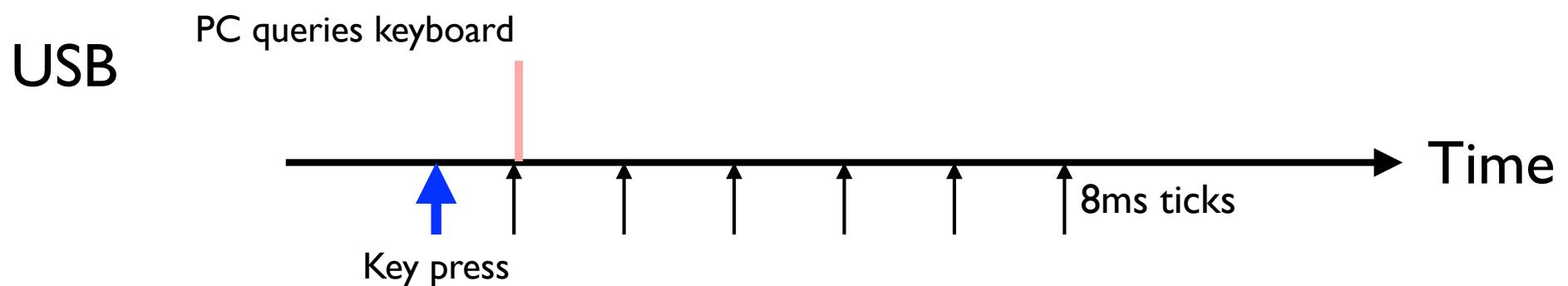
PS/2 protocol uses *interrupts*: when the PS/2 scan code is ready, it tells the processor immediately (assignment 7)

Polling vs. Interrupts

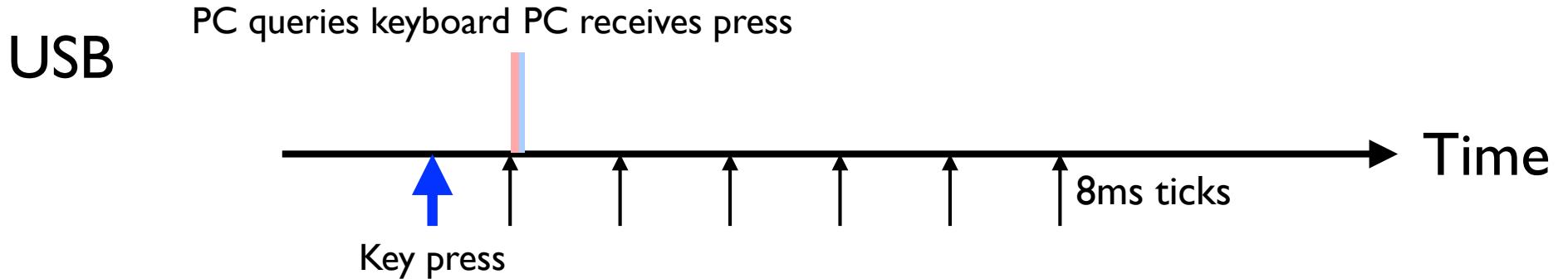
USB



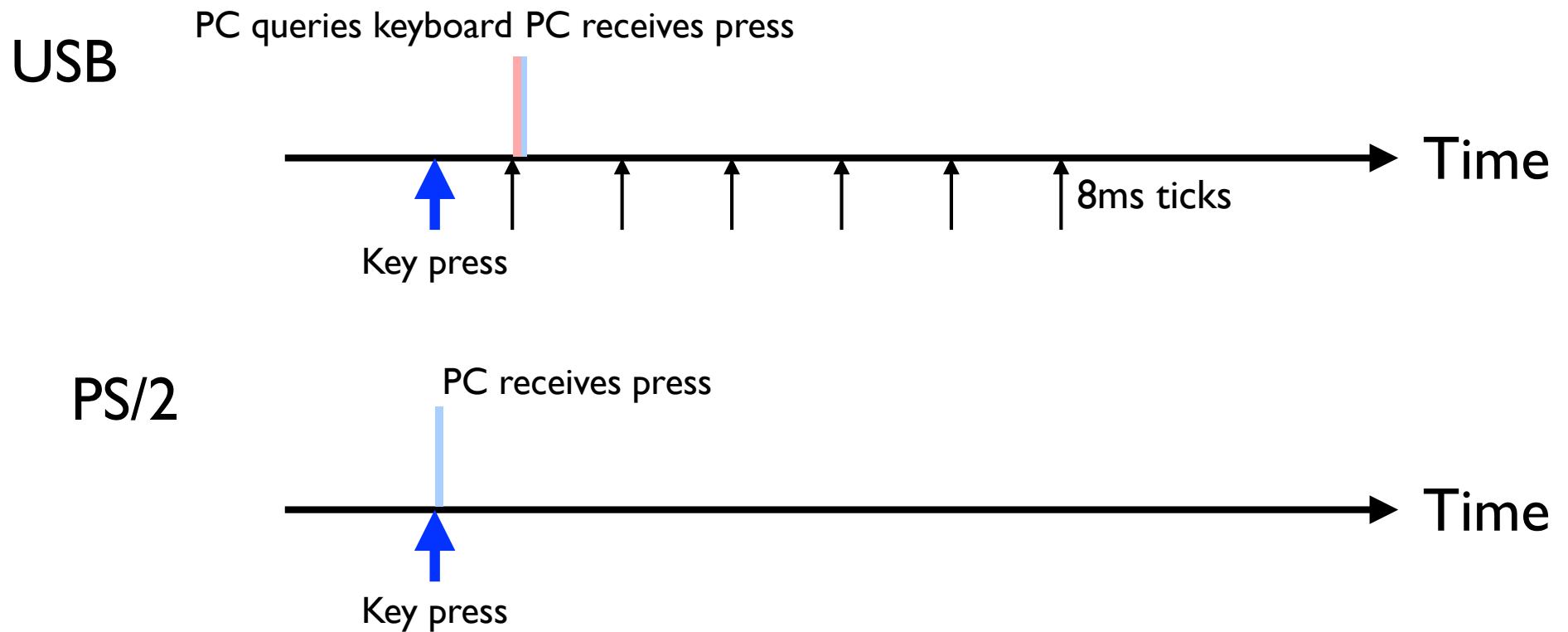
Polling vs. Interrupts



Polling vs. Interrupts



Polling vs. Interrupts



Latency vs. Throughput

USB protocol has higher *throughput* than PS/2 (sends data at Mbps vs. kbps)

But early USB keyboards had higher *latency* because they would poll at 8ms intervals

Modern USB keyboards poll at 1ms; a PS/2 scan code takes about 1ms to send, so unless your USB has lots of data going across it, polling doesn't matter

USB

Extremely complex protocol!

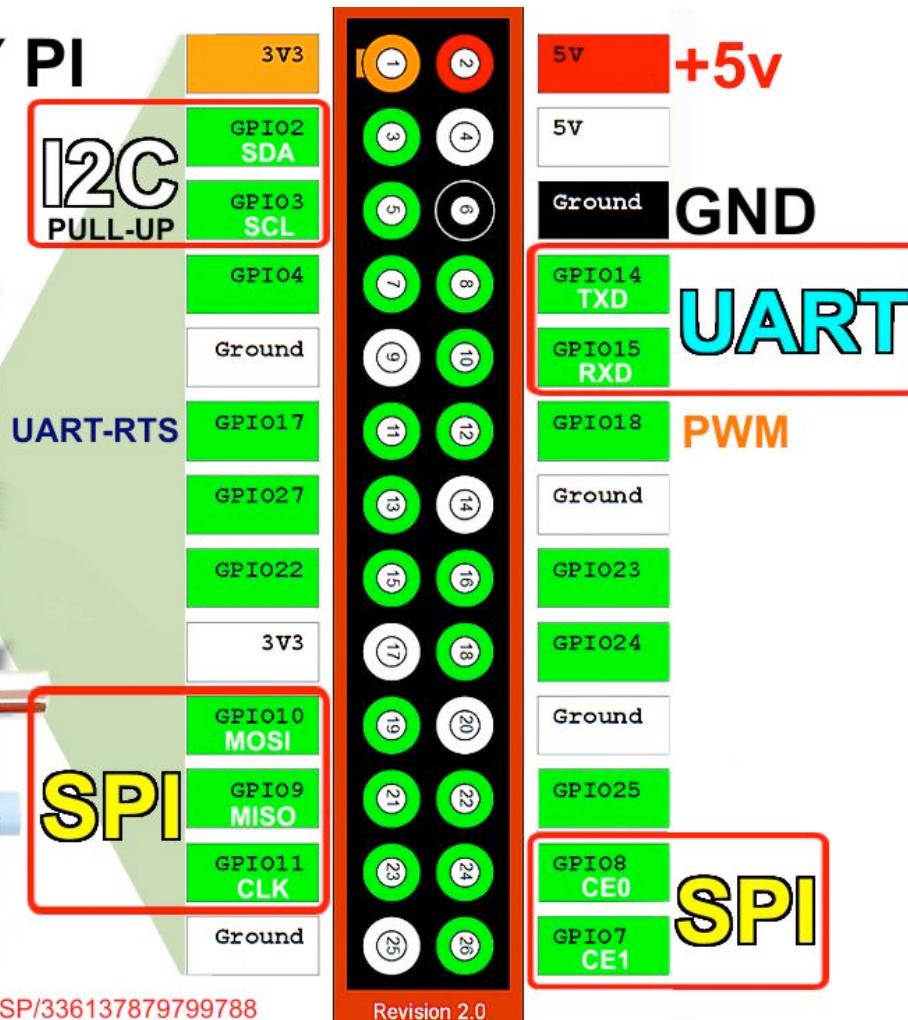
Devices can be plugged in dynamically, extensible topology (adding hubs), your computer 'just knows' what a device can do

Event # 2 16.558,844 s	DATA LINE HIGH D+ (PLUGIN)	DELAY 2,998 us
Event # 3 16.561,843 s	SUSPEND OK	IDLE 202,277 us
Event # 4 16.764,121 s	Start of RESET OK	Duration 50,985 us End of RESET FULL SPEED LINK IDLE 0.42 us
#18...28 16.826,881 s	FS Control Transfer Get Device Descriptor	Addr 0x00 Endp 0x0 Data (8 bytes) 12 01 10 01 00 00 00 08 Status OK
#31...49 16.830,882 s	FS Control Transfer Get Device Descriptor	Addr 0x00 Endp 0x0 Data (18 bytes) 12 01 10 01 00 00 00 08... Status OK
#52...58 16.836,884 s	FS Control Transfer Set Address (0x01)	Addr 0x00 Endp 0x0 Data (0 bytes) Status OK
#61...71 16.839,886 s	FS Control Transfer Get Configuration Descriptor	Addr 0x01 Endp 0x0 Data (8 bytes) 09 02 29 00 01 01 00 80 Status OK
#74...104 16.843,887 s	FS Control Transfer Get Configuration Descriptor	Addr 0x01 Endp 0x0 Data (41 bytes) 09 02 29 00 01 01 00 80... Status OK
#107...113 16.852,891 s	FS Control Transfer Set Configuration (0x01)	Addr 0x01 Endp 0x0 Data (0 bytes) Status OK
#116...194 16.855,892 s	FS Control Transfer Get HID Report Descriptor	Addr 0x01 Endp 0x0 Data (139 bytes) 05 01 09 04 A1 01 A1 02... Status OK

Raspberry Pi Header Pins

RASPBERRY PI Revision 2 Pinout

<http://www.pinballsp.com>



<https://www.facebook.com/pages/PinballSP/336137879799788>