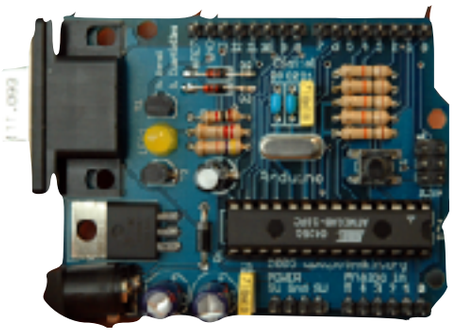
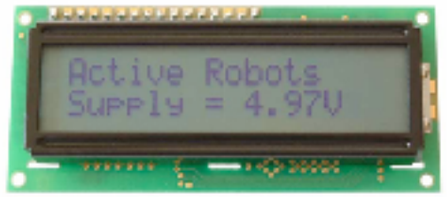
EECS 159A/CSE 181A
Layers of Communication

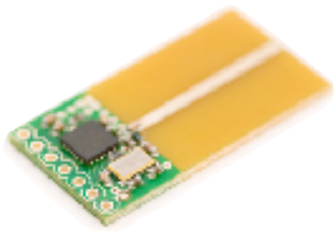
You may have a system architecture, but…

- You still need to write code…
  - to "talk <u>to</u>" a device
  - to "talk <u>through</u>" a device to another subsystem
  - to track the state of your application
- Need different styles of communication
  - sending and receiving, pushing and pulling, notification
  - define the API

Example: ID card scanner







UART

scanner

pair/wake

interrupt

on/off

supply power

Keypad (columns)

GPIO

(RESET) PC6 1
(RXD) PD0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
(XTAL1/TOSC1) PB6 9
(XTAL2/TOSC2) PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14

28 PC5 (ADC5/SCL)
27 PC4 (ADC4/SDA)
26 PC3 (ADC3)
25 PC2 (ADC2)
24 PC1 (ADC1)
23 PC0 (ADC0)
22 GND
21 AREF
20 AVCC
19 PB5 (SCK)
18 PB4 (MISO)
17 PB3 (MOSI/OC2)
16 PB2 (SS/OC1B)
15 PB1 (OC1A)

I2C

LCD

RTC

Keypad (rows)

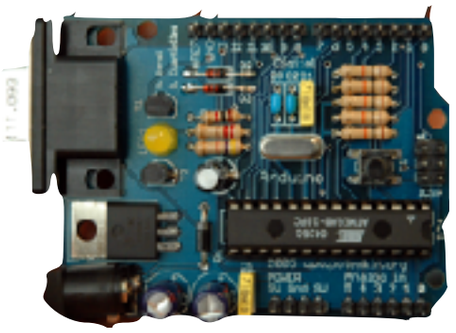
GPIO

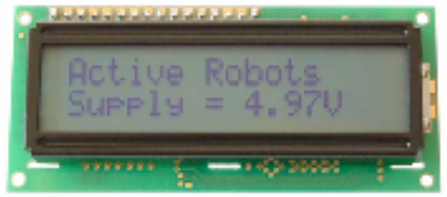
SPI bus

RF

flash

Example: ID card scanner





scanner

UART

pair/wake

interrupt

on/off

supply power

Keypad (columns)

GPIO

(RESET) PC6	1	28	PC5 (ADC5/SCL)
(RXD) PD0	2	27	PC4 (ADC4/SDA)
(TXD) PD1	3	26	PC3 (ADC3)
(INT0) PD2	4	25	PC2 (ADC2)
(INT1) PD3	5	24	PC1 (ADC1)
(XCK/T0) PD4	6	23	PC0 (ADC0)
VCC	7	22	GND
GND	8	21	AREF
(XTAL1/TOSC1) PB6	9	20	AVCC
(XTAL2/TOSC2) PB7	10	19	PB5 (SCK)
(T1) PD5	11	18	PB4 (MISO)
(AIN0) PD6	12	17	PB3 (MOSI/OC2)
(AIN1) PD7	13	16	PB2 (SS/OC1B)
(ICP1) PB0	14	15	PB1 (OC1A)

I2C

LCD

RTC

Keypad (rows)

GPIO

SPI bus

RF

flash

Data from the Scanner

- Depends, but normally it's just ASCII data
  - 3 Tracks of data, emulates a keyboard device
- Sample data (text string)
  - %B6038380012345678^ANTEATER/PETER Z
    ^4912120000000000 000 ?;
    6038380006514029=4912120000000000000?
- just need to extract the 8-digit ID and name
- comes in on the serial port (UART)

How to read data from the serial port?

- Depends on the runtime support, if any
  - Arduino example:

```
int incomingByte = 0;   // for incoming serial data

void setup() {
        Serial.begin(9600); // set the baud rate
}

void loop() {
        if (Serial.available() > 0) {
                // read the incoming byte:
                incomingByte = Serial.read();
        }
}
```

Serial port functions

```
int incomingByte = 0;

void setup() {
        Serial.begin(9600);
}
```

```
if (Serial.available() > 0) {
   // read the incoming byte:
   incomingByte = Serial.read();
}
```

- Configures the serial port to 9600 baud, sets up the interrupt service routine (ISR) for the serial port to buffer incoming bytes
- Tests the buffered byte count
- Dequeues the next byte received

Driver vs. HAL

- Driver
  - Steps required to configure & access the I/O controller
  - Hopefully independent of architecture by calling HAL
- HAL: hardware-abstraction layer
  - isolates architecture-dependent code behind API
  - Different architectures have their own HAL definition
  - HAL API should be the same across architectures

SFRs for UART on ATmega8

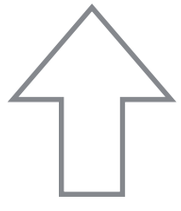
- UDR: data registers
  - same name for Rx & Tx
- UCSRA: status register A
  - Rx/Tx complete, buffer empty, frame error, data overrun, parity error, double Tx speed, multi-processor mode
- UCSRB: status register B
  - Interrupt enable on Rx, Tx, empty; Rx enable, Tx enable, char size, 9th data bit for Rx & Tx
- UCSRC: status register C
  - USART register select, mode select, parity mode, stop bit select, char size, polarity
- UBBR: baud rate register

UART Driver vs. HAL

Driver

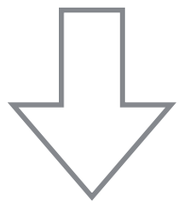
```
#include “atega8/hal.h”

void Serial::begin(int baud) {
    hal_setBaudRate(baud);
    hal_initRxQueue();
    hal_enableRxTx();
}
```

architect independent



architect dependent

(could be same MCU but different board,
or different MCU altogether, or software UART using GPIO!)



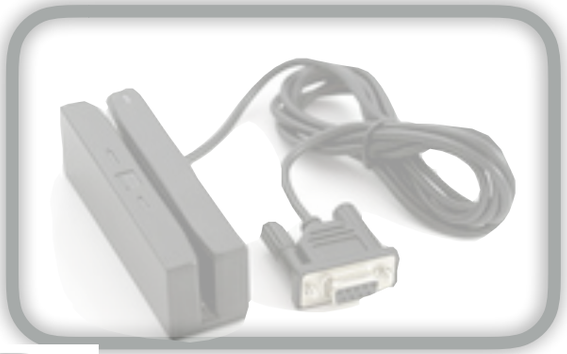
HAL for ATmega8

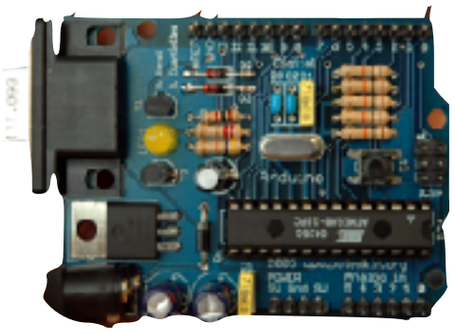
```
#include “atega8/hal.h”
#define BAUDRATE(baud) \
   ((F_CPU)/((baud)*16UL)-1)
void hal_setBaudRate(int baud) {
   UBRRH = (baud >> 8);
   UBRRL = BAUDRATE(baud);
}

void hal_enableRxTx() {
    UCSRB|= (1<<TXEN)|(1<<RXEN);
}
```

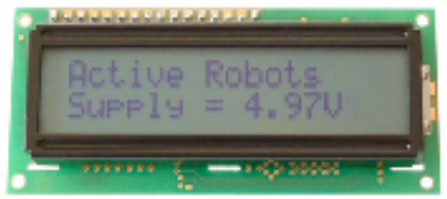
HAL for another arch.

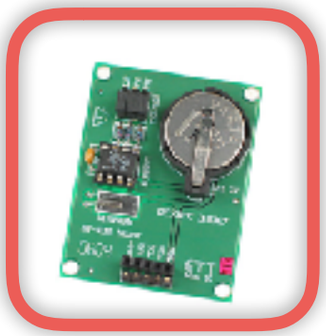
```
#include “myMCU/hal.h”
#define BAUDRATE(baud) \
   ((…))
void hal_setBaudRate(int baud) {
   …
}

void hal_enableRxTx() {
    …
}
```

Example: RTC









UART

scanner

pair/wake

interrupt

on/off

supply power

Keypad (columns)

GPIO

(RESET) PC6 1
(RXD) PD0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
(XTAL1/TOSC1) PB6 9
(XTAL2/TOSC2) PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14

28 PC5 (ADC5/SCL)
27 PC4 (ADC4/SDA)
26 PC3 (ADC3)
25 PC2 (ADC2)
24 PC1 (ADC1)
23 PC0 (ADC0)
22 GND
21 AREF
20 AVCC
19 PB5 (SCK)
18 PB4 (MISO)
17 PB3 (MOSI/OC2)
16 PB2 (SS/OC1B)
15 PB1 (OC1A)

LCD

RTC

I2C

Keypad (rows)

GPIO

SPI bus

RF

flash

Example RTC: DS1307

- Data Sheet:
  https://datasheets.maximintegrated.com/en/ds/DS1307.pdf
- Supported by Arduino library <DS1307RTC.h>
- Format of Transactions over I2C
  - I2C protocol requires slave ack every byte until the last
- Backed by coin-cell battery
- Tracks time fields
  - ss:mm:hh:DoW/dd/mm/yy
  - can read and write most fields

address	content (RTC)
00h	10sec, sec
01h	10min, min
02h	10hour, hour
03h	day of week
04h	10date, date
05h	leapyr, 10mon, mon
06h	10year, year

<DS1307RTC.h> API

RTC.get();
Reads the current date & time as a 32 bit "time_t" number. Zero is returned if the DS1307 is not running or does not respond.
RTC.set(t);
Sets the date & time, using a 32 bit "time_t" number. Returns true for success, or false if any error occurs.
RTC.read(tm);
Read the current date & time as TimeElements variable. See the Time library for TimeElements details. Returns true on success, or false if the time could not be read.
RTC.write(tm);
Sets the date & time, using a TimeElements variable. Returns true for success, or false if the time could not be written.
RTC.chipPresent();
Returns true if a DS1307 compatible chip was present after using the 4 functions. If an error occurs, this can be used to distinguish between a DS1307 that is not running vs no chip connected at all.

Driver & interface layers for RTC

MCU side

RTC side

RTC Driver

program logic

`RTC::get()` `RTC::set()`

I2C Driver

`I2C::read()` `I2C::write()`

address	content (RTC)
00h	10sec, sec
01h	10min, min
02h	10hour, hour
03h	day of week
04h	10date, date
05h	leapyr, 10mon, mon
06h	10year, year

responds to r/w
at given address

software or logic

I2C controller

start cond → target addr. → byte r/w → stop cond.

start cond → target addr. → byte r/w → stop cond.

hardware

I2C Wires

SDA

SCL

Steps in setting time

MCU side

RTC side

RTC Driver

I2C Driver

I2C controller

I2C Wires

software or logic

hardware

program logic

`RTC::get()` `RTC::set()`

`I2C::read()` `I2C::write()`

start cond → target addr. → byte r/w → stop cond.

`addrPtr`

responds to r/w at given address

start cond → target addr. → byte r/w → stop cond.

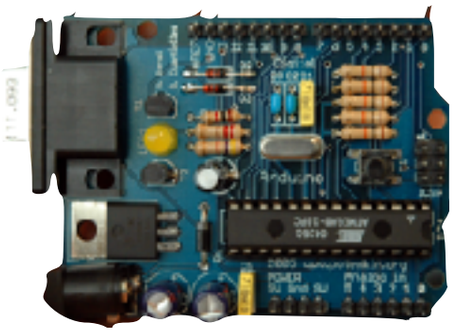
address	content (RTC)
00h	10sec, sec
01h	10min, min
02h	10hour, hour
03h	day of week
04h	10date, date
05h	leapyr, 10mon, mon
06h	10year, year

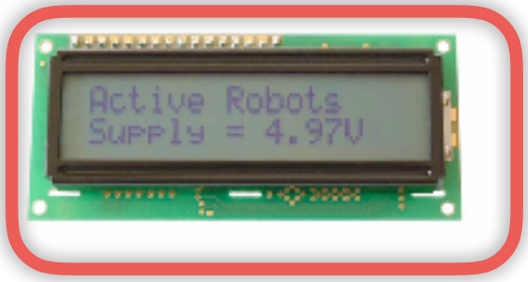
(1) (2) (3) (4) (5) (6)

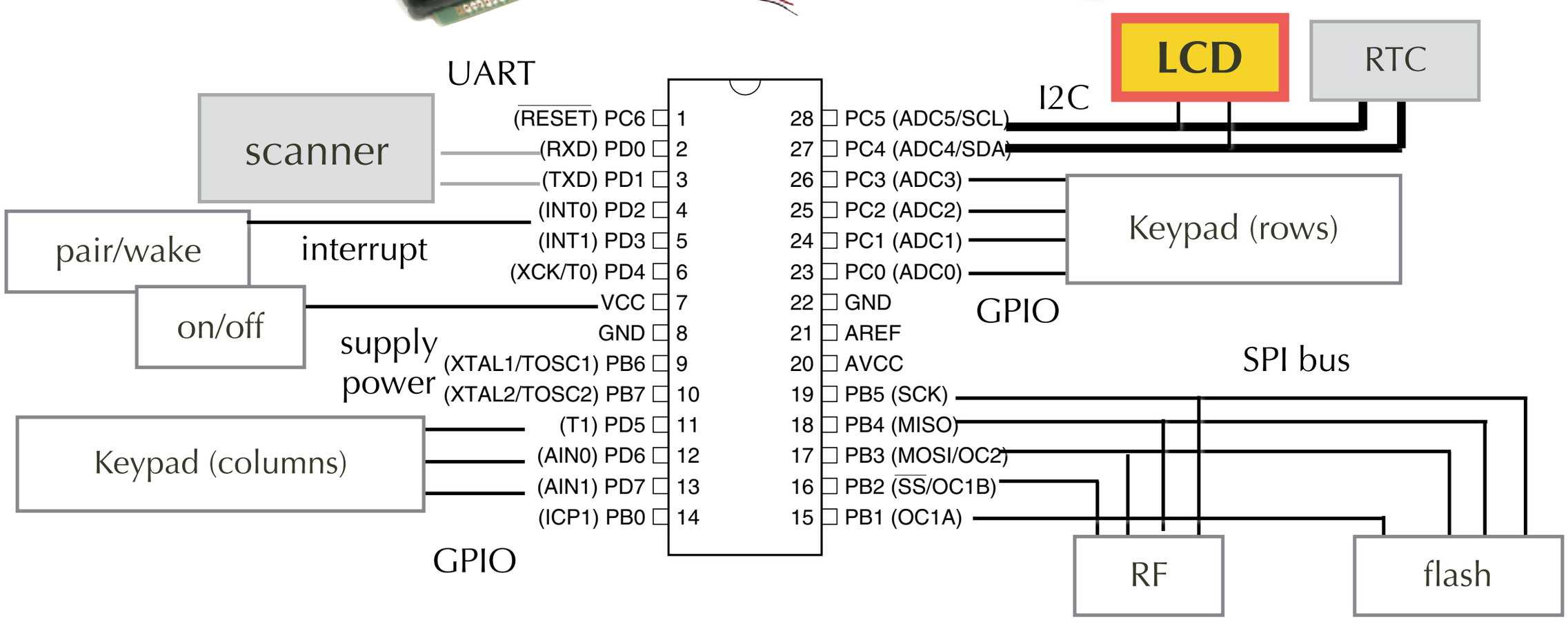
SDA

SCL

Example: LCD




Active Robots
Supply = 4.97V


UART
scanner
pair/wake
interrupt
on/off
supply
power
Keypad (columns)
GPIO
(RESET) PC6 1
(RXD) PD0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
(XTAL1/TOSC1) PB6 9
(XTAL2/TOSC2) PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14
28 PC5 (ADC5/SCL)
27 PC4 (ADC4/SDA)
26 PC3 (ADC3)
25 PC2 (ADC2)
24 PC1 (ADC1)
23 PC0 (ADC0)
22 GND
21 AREF
20 AVCC
19 PB5 (SCK)
18 PB4 (MISO)
17 PB3 (MOSI/OC2)
16 PB2 (SS/OC1B)
15 PB1 (OC1A)
I2C
LCD
RTC
Keypad (rows)
GPIO
SPI bus
RF
flash

LCD Commands

#	Command Description
1	Write a byte or multiple bytes of data
2	Display on, cursor off
3	Display and cursor off
4	Set cursor position
5	Cursor home
6	Reset cursor position
7	Move cursor left one place
8	Move cursor right one place
9	Underline cursor on
10	Display - on, cursor - off, set cursor wink
11	Display - on, cursor - off, set cursor blink
12	Clear screen
13	Load custom character
14	Set display for 2 lines and 16 characters
15	Move cursor/shift display right (cursor position auto increment)
16	Move cursor/shift display right (cursor position auto decrement)

Driver & interface layers for LCD

MCU side

LCD side

software or logic

LCD Driver

program logic

```
LCD::print(char*)
LCD::move(row,col)
LCD::cursor(bool)
LCD::display(bool)
```

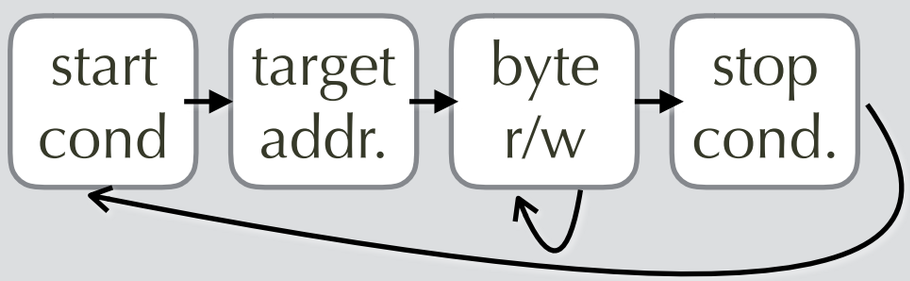
#	Command Description
1	Write a byte or multiple bytes of data
2	Display on, cursor off
3	Display and cursor off
4	Set cursor position
5	Cursor home
6	Reset cursor position
7	Move cursor left one place
8	Move cursor right one place
9	Underline cursor on
10	Display - on, cursor - off, set cursor wink
11	Display - on, cursor - off, set cursor blink
12	Clear screen
13	Load custom character
14	Set display for 2 lines and 16 characters
15	Move cursor/shift display right (cursor position auto
16	Move cursor/shift display right (cursor position auto

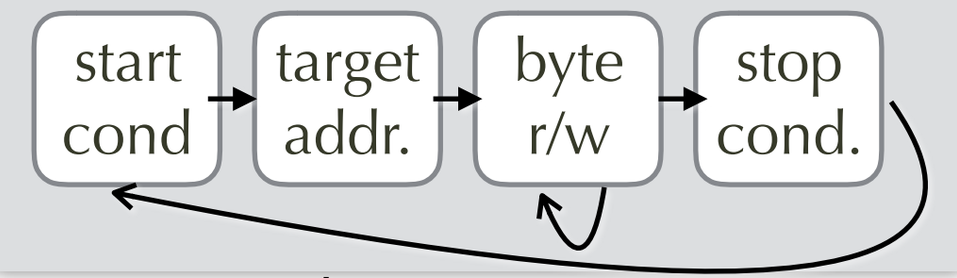
I2C Driver

```
I2C::read()  I2C::write()
```

dispatcher

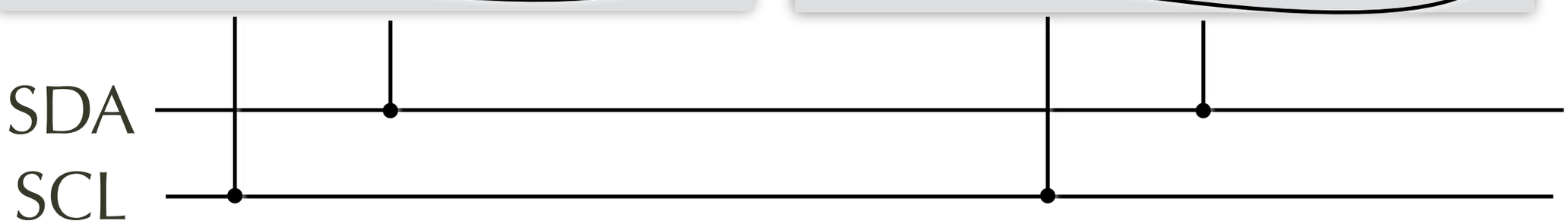
I2C controller





hardware

I2C Wires



LCD & RTC sharing I2C bus

MCU

LCD

RTC

program logic

called by both LCD & RTC

LCD Driver

`LCD::print(char*)`

`LCD::move(r,c)`

`LCD::cursor(bool)`

`LCD::display(bool)`

RTC Driver

`RTC::get()`

`RTC::set(t)`

#	Command Description
1	Write a byte or
2	Display on, cursor off
3	Display and cursor off
4	Set cursor position
5	Cursor home
6	Reset cursor position
7	Move cursor left one
8	Move cursor right one
9	Underline cursor on
10	Display - on, cursor -
11	Display - on, cursor -
12	Clear screen
13	Load custom character
14	Set display for 2 lines
15	Move cursor/shift
16	Move cursor/shift

addr	content (RTC)
00h	10sec, sec
01h	10min, min
02h	10hour, hour
03h	day of week
04h	10date, date
05h	leapyr, 10mon,
06h	10year, year

software or logic

I2C Driver

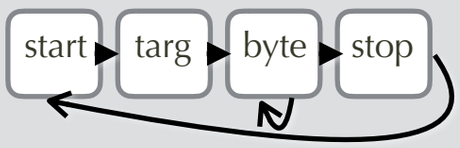
`I2C::read()`

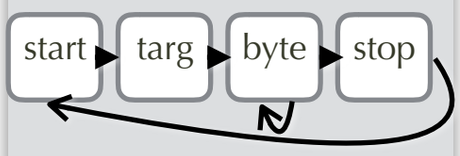
`I2C::write()`

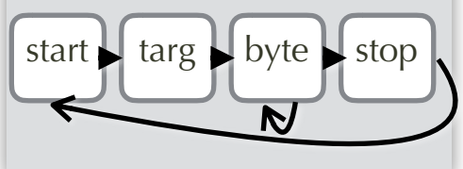
dispatcher

dispatcher

I2C controller







hardware

I2C Wires

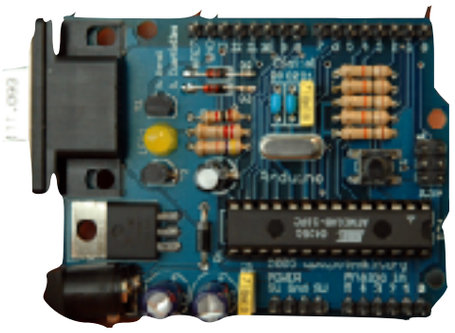
SDA

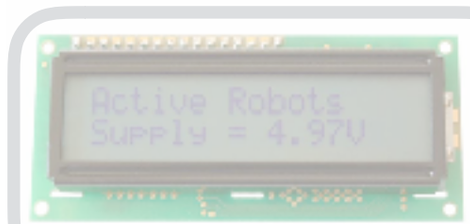
SCL

Issues with sharing I2C driver

- Code reuse
  - Once I2C works, easy to build on it
- No Reentrance
  - Same code should not be called before another invocation completes
  - Why? the (I2C) bus is still busy!
  - maybe some global variables may get updated improperly

Example: flash





UART

scanner

pair/wake

interrupt

on/off

supply power

Keypad (columns)

GPIO

(RESET) PC6 1
(RXD) PD0 2
(TXD) PD1 3
(INT0) PD2 4
(INT1) PD3 5
(XCK/T0) PD4 6
VCC 7
GND 8
(XTAL1/TOSC1) PB6 9
(XTAL2/TOSC2) PB7 10
(T1) PD5 11
(AIN0) PD6 12
(AIN1) PD7 13
(ICP1) PB0 14

28 PC5 (ADC5/SCL)
27 PC4 (ADC4/SDA)
26 PC3 (ADC3)
25 PC2 (ADC2)
24 PC1 (ADC1)
23 PC0 (ADC0)
22 GND
21 AREF
20 AVCC
19 PB5 (SCK)
18 PB4 (MISO)
17 PB3 (MOSI/OC2)
16 PB2 (SS/OC1B)
15 PB1 (OC1A)

I2C

LCD

RTC

Keypad (rows)

GPIO

SPI bus

RF

flash

Flash memory

- Interfaces
  - built-in to MCU (access via SFR)
  - SPI: 1-bit full-duplex
  - SD or MMC: 4-bit half-duplex
  - parallel: 8-bit, 16-bit, 32-bit data, separate address
- Types and access granularity
  - NAND: page read/write, block (sector) erase
  - NOR: word read, page write, block erase
- Format
  - raw data storage vs. file system (e.g., FAT)

Example: Cypress S25FL512S

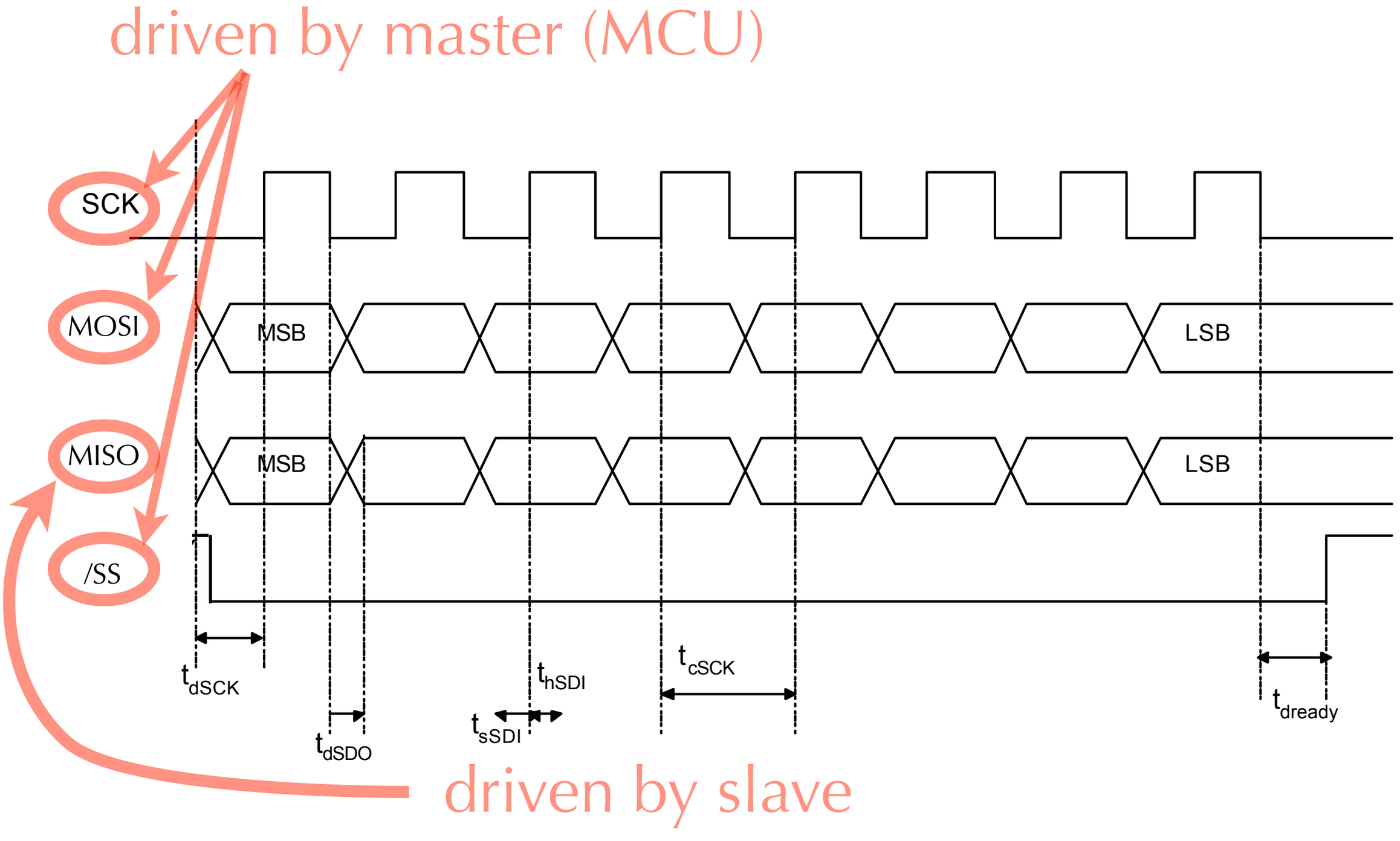
- Serial flash, 512Mb SPI flash
- Command categories
  - Device ID
  - Register access
  - Read, program (write), erase flash array
  - Reset
  - Protect

http://www.cypress.com/documentation/datasheets/s25fl512s-512-mbit-64-mbyte-30v-spi-flash-memory

SPI: Master-Slave protocol

- Master
  - Asserts Slave Select (/SS)
  - Drives the clock (SCK)
- Data
  - Master and slave swap data, 1 bit/clock
  - Full duplex

SPI timing example

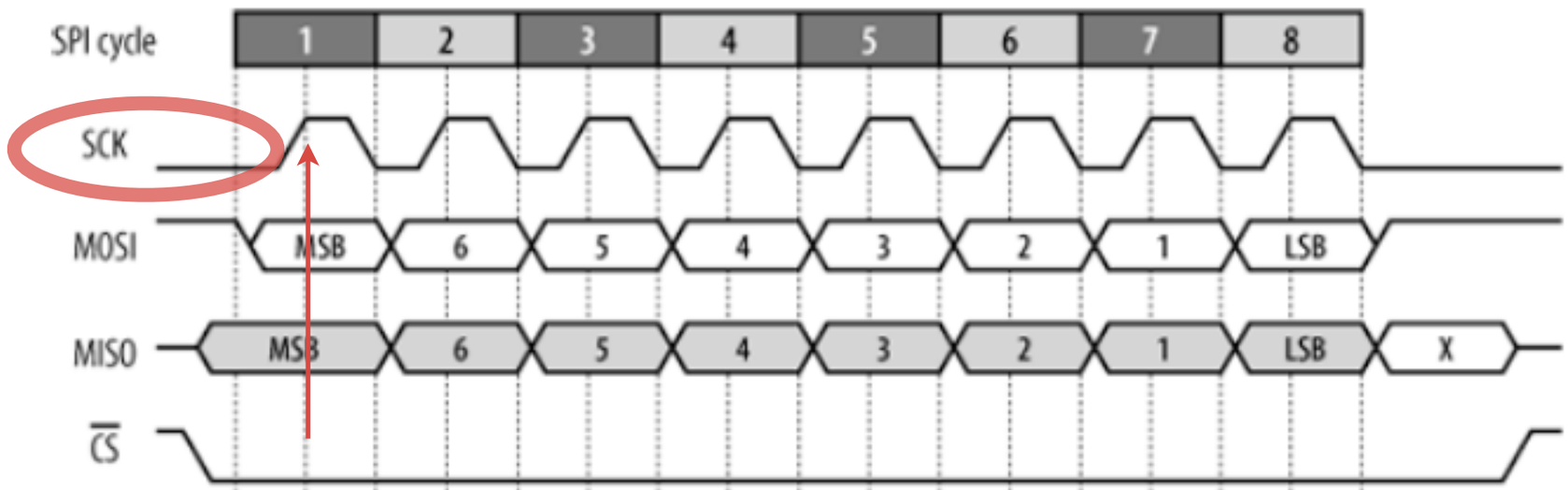


Four SPI modes -- need to know which one!

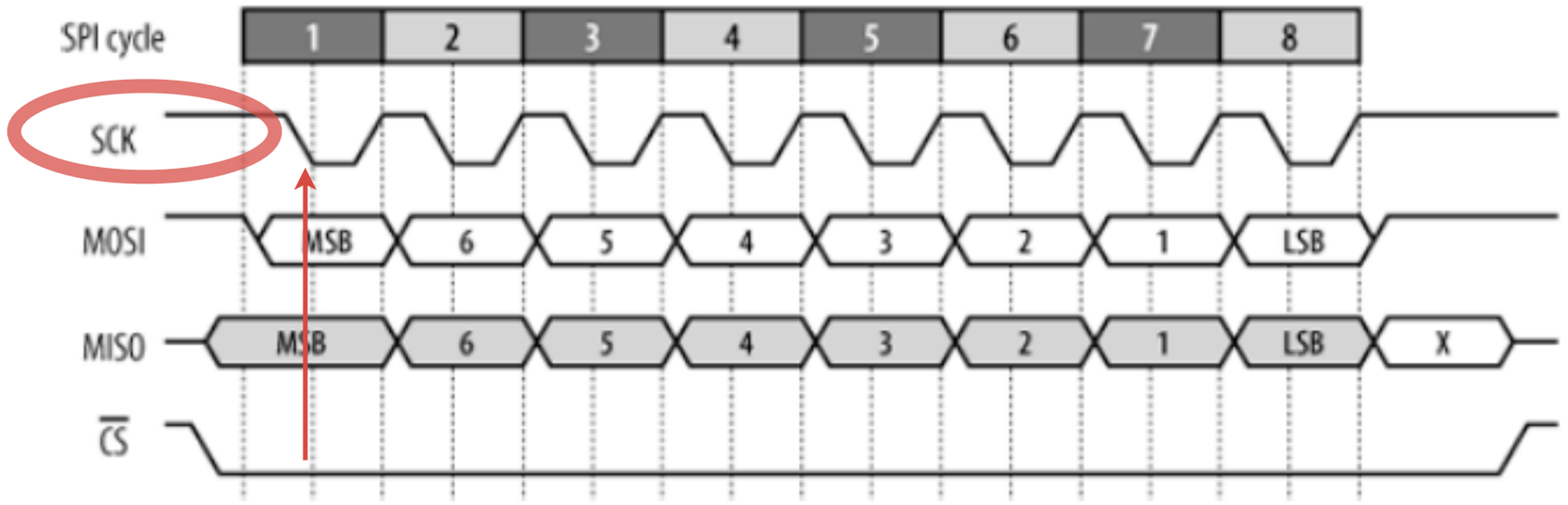
- SCK Polarity
  - Low when idle, data valid on rising edge
  - High when idle, data valid on falling edge
- SCK Phase
  - Phase zero: first edge after /CS asserted
  - Phase one: second edge after /CS asserted

Phase-zero SCK

low polarity, rising edge

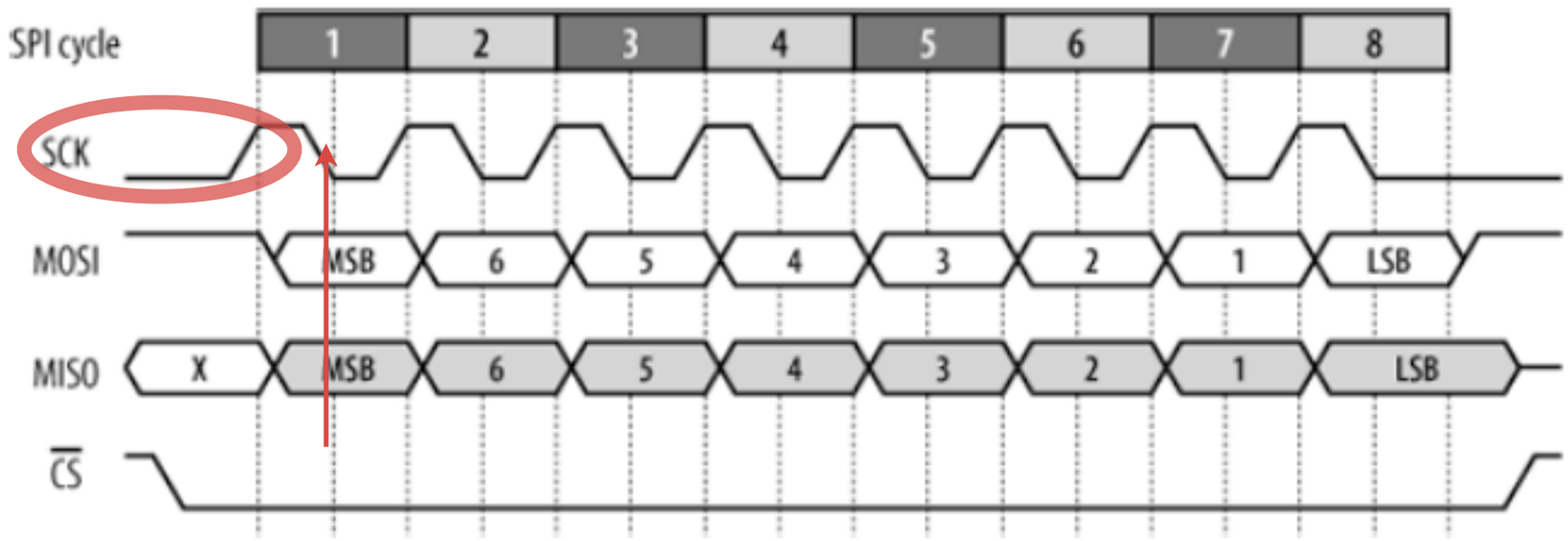


high polarity, falling edge

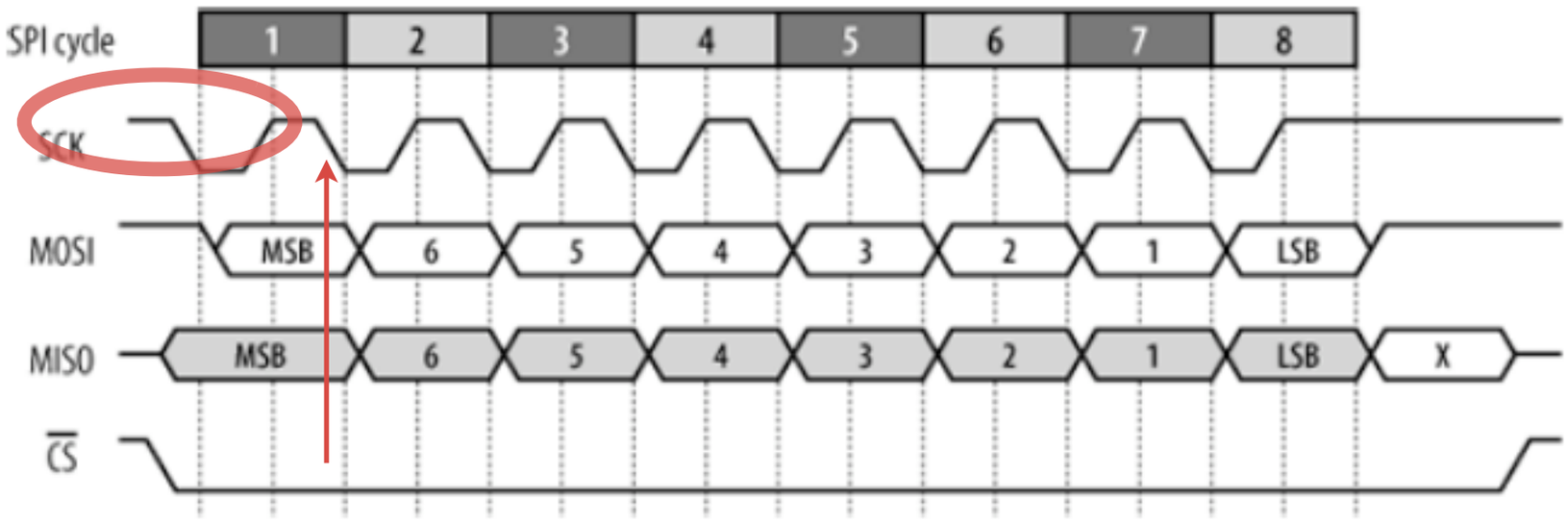


Phase-one SCK

low polarity, after rising edge



high polarity, after falling edge



How to generate SPI waveform?

- Use the built-in SPI Controller on MCU
  - software reads/writes hardware registers (also known as Special Function Registers (SFR))
  - causes hardware action
    - configure phasing & polarity
    - transfer (send and receive) data bytes
    - bit to indicate data ready
- Or, implement SPI protocol in software
  - read/write GPIO pins to make the waveforms

Driver for SPI flash

MCU side

flash side

flash Driver

program logic

```
flash::deviceID()
flash::read(addr, buf)
flash::write(addr,buf)
flash::erase(addr)
flash::reset()
flash::protect()
```

flash array

Command
Device ID
Register access
Read, program (write), erase flash
Reset
Protect

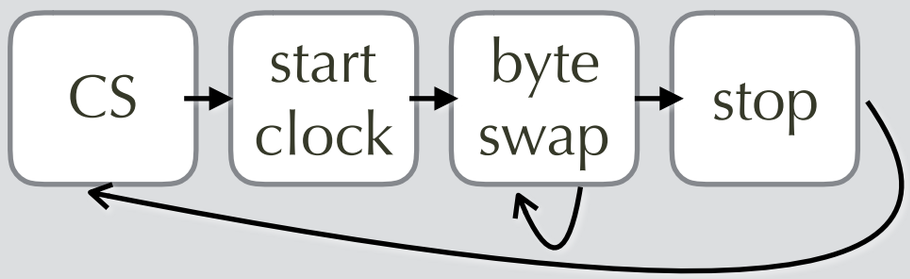
SPI Driver

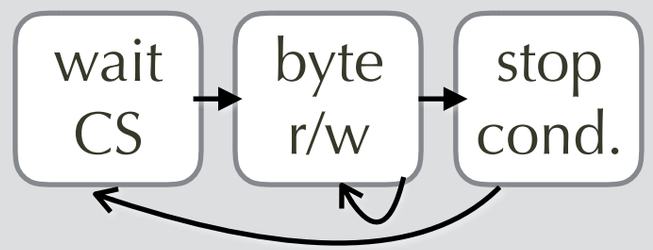
```
SPI::byteswap(byte, CS)
```

dispatcher

software or logic

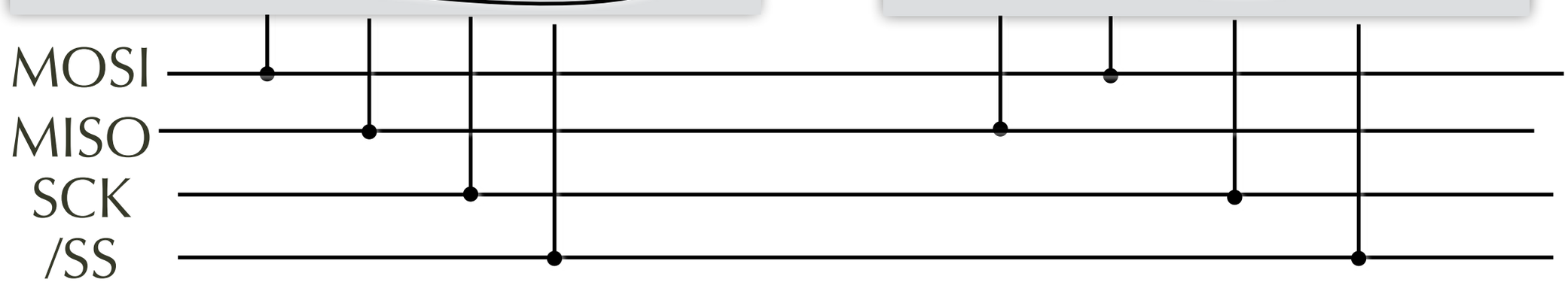
SPI controller



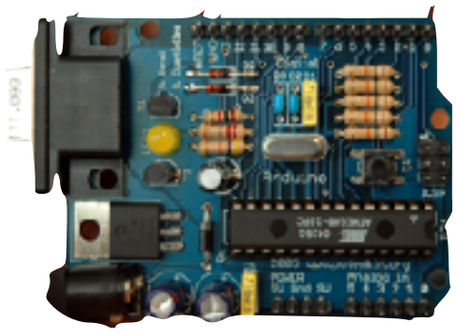


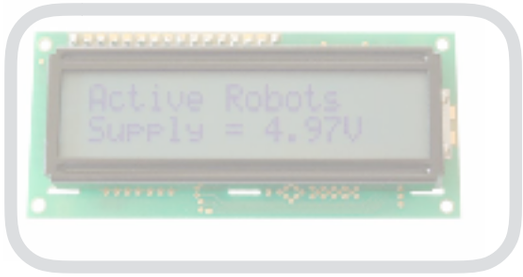
hardware

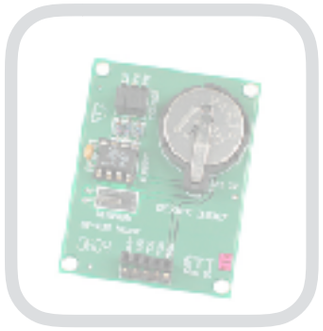
SPI Wires

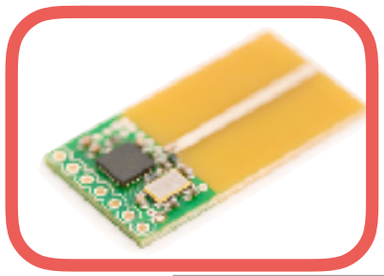


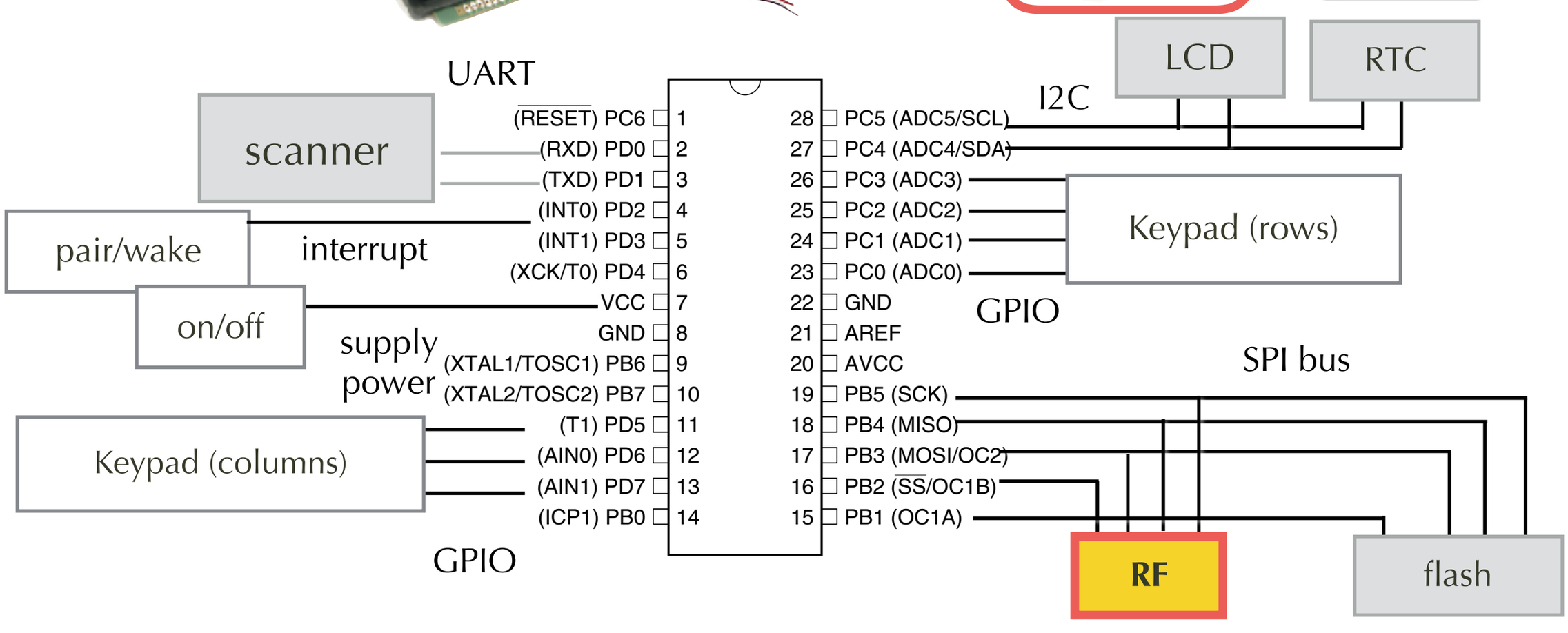
Example: RF module











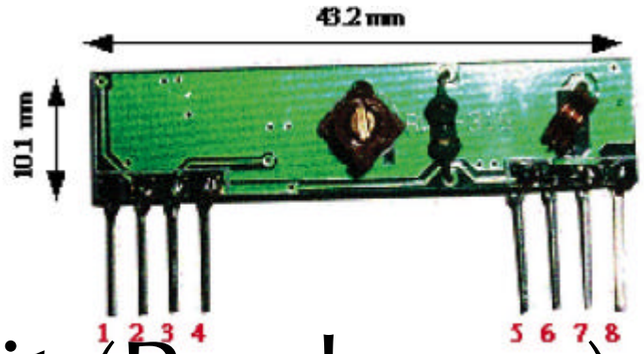
Many choices of RF Protocols

- Analog radio (PHY only)
- Proprietary: RFM TR1100, CC1000
- Bluetooth / 802.15.1
- Bluetooth Low Energy / Wibree
- ZigBee / 802.15.4
- Z-Wave, Dash-7, others...

RF Frequency bands

- 2.4 GHz
  - Worldwide license-free operation
- 900 MHz
  - Longer range than 2.4GHz, not worldwide license-free
- 433 MHz
  - Longer range, penetrates walls better

UART over AM Radio



pin 1 : Gnd
pin 2 : Digitak Output
pin 3 : Linear Output
pin 4 : Vcc
pin 5 : Vcc
pin 6 : Gnd
pin 7 : Gnd
pin 8 :ANT (About 30 - 35 cm)

Moulation : AM
Supply Voltage : 5v dc

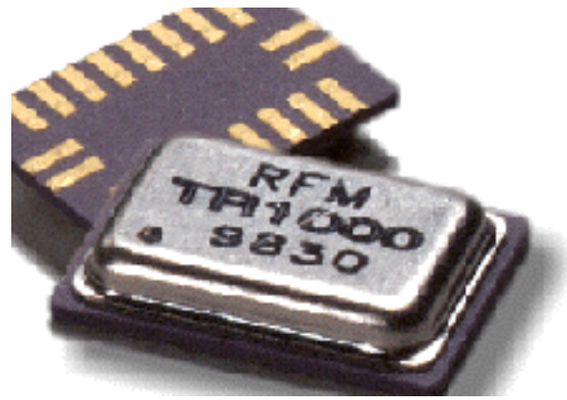
- 315 or 433 MHz
- Separate Rx and Tx unit (Rx shown)
- Can hook up UART line, AM radio
  - point-to-point only, but not SPI, I2C
- Simple, ~500 m range; low data rate (4.8Kbps)
- Point-to-point only; >2 can't coexist

http://www.HobbyEngineering.com/specs/WENSHING-RWS3716C.pdf

RFM TR1000

(used in original Mica motes)

- Digital radio, 916.5 MHz hybrid transceiver
  - modulates-demodulates bit
  - 115.2 kbps
  - improvement over AM module



- May need external support
  - Separate Tx power controller (DS 1804)
  - Serialization accelerator, Timing accelerator
  - Media access control, often done in software

http://wireless.murata.com/datasheet?/RFM/data/tr1000.pdf

CC1000

(used in Mica2 motes)

- Digital transceiver
  - Modulation and demodulation
  - Hardware codec (Manchester)
  - Hardware synchronization
  - Byte interface with MCU
  - Power control
- Software MAC
  - AVR(Atmega 128L)
  - Protocol processing

http://www.ti.com/product/cc1000

CC2420

(used in MicaZ motes and Telos motes)

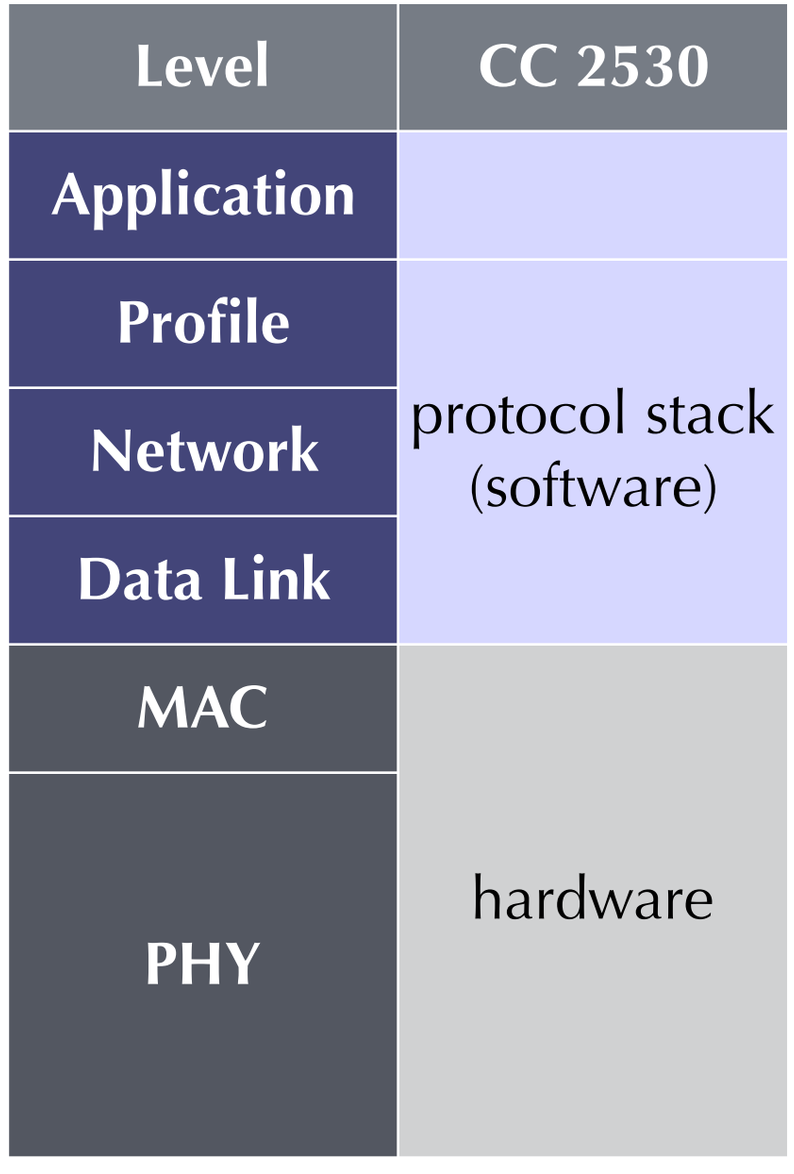
- PHY/MAC for 802.15.4
  - Handles packets instead of just bytes
  - Modulation, demodulation, Codec, encryption
  - Clear channel assessment, link quality, packet timing
  - 7mm x 7mm package
- "ZigBee Ready"

Levels of Abstraction supported by the chip or module

Level	Concept	AM radio	TR 1000	CC 1000	CC 2420	CC 2530
Application	behavior					
Profile	format					protocol stack (software)
Network	topology					
Data Link	connection					
MAC	packets				v	hardware
PHY	bytes			v	v	
PHY	bits		v	v	v	
PHY	volts	v	v	v	v	

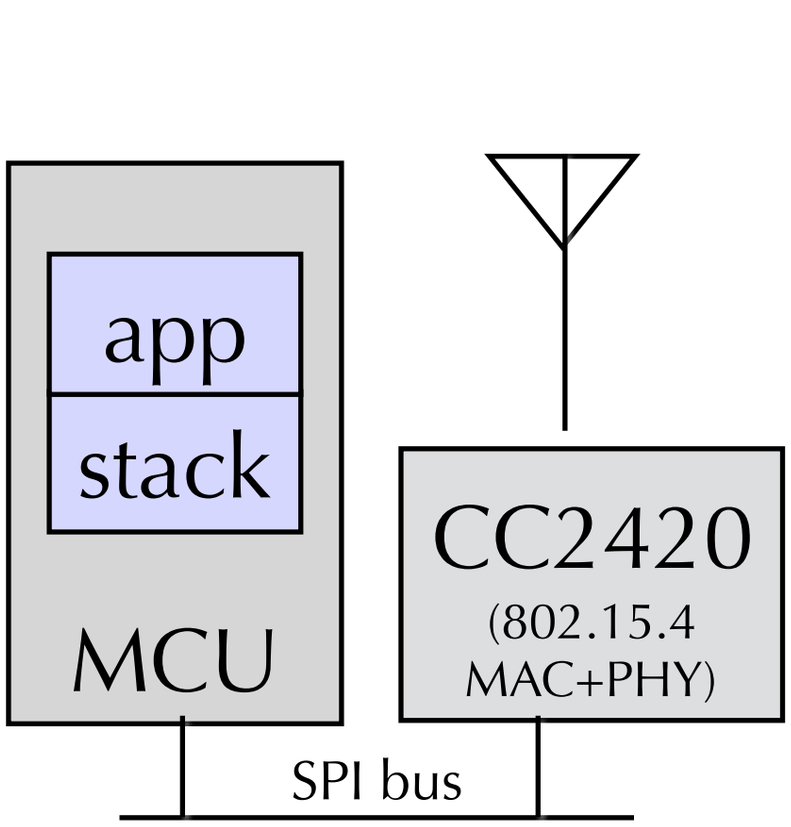
CC 2530

- Integrated MCU+MAC+PHY
- Why MCU?
  - to run protocol stack
    - May also run application! (but limited room)
- Alternative stacks
  - ZigBee, RF4CE, 6loWPAN, WirelessHART, ...

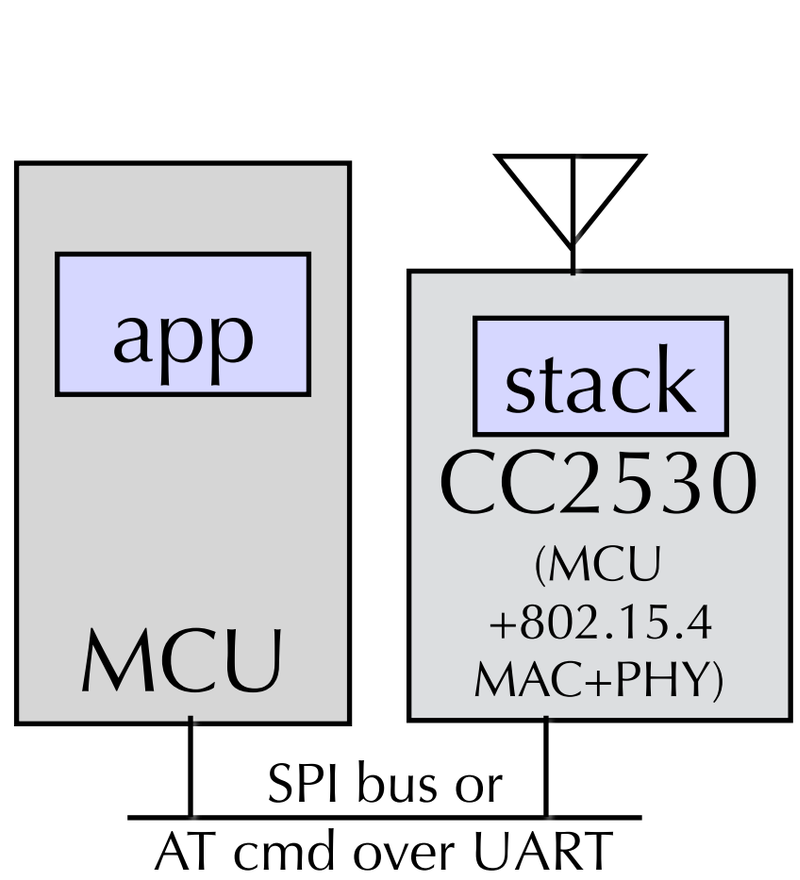


http://www.ti.com/product/cc2530

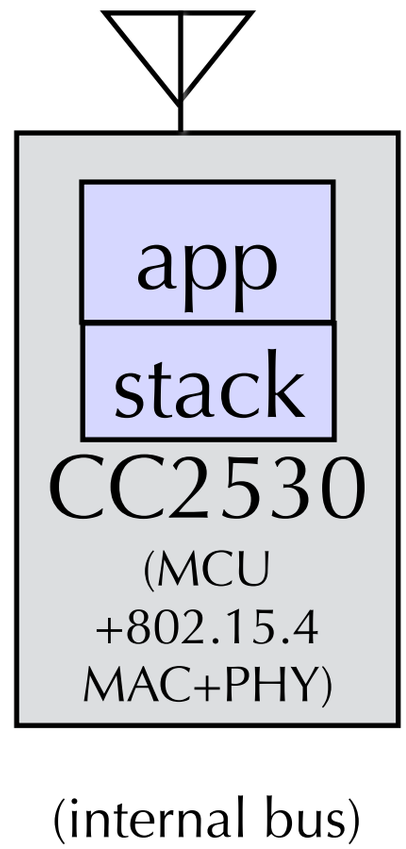
Possible Configurations



"Host-stack" configuration

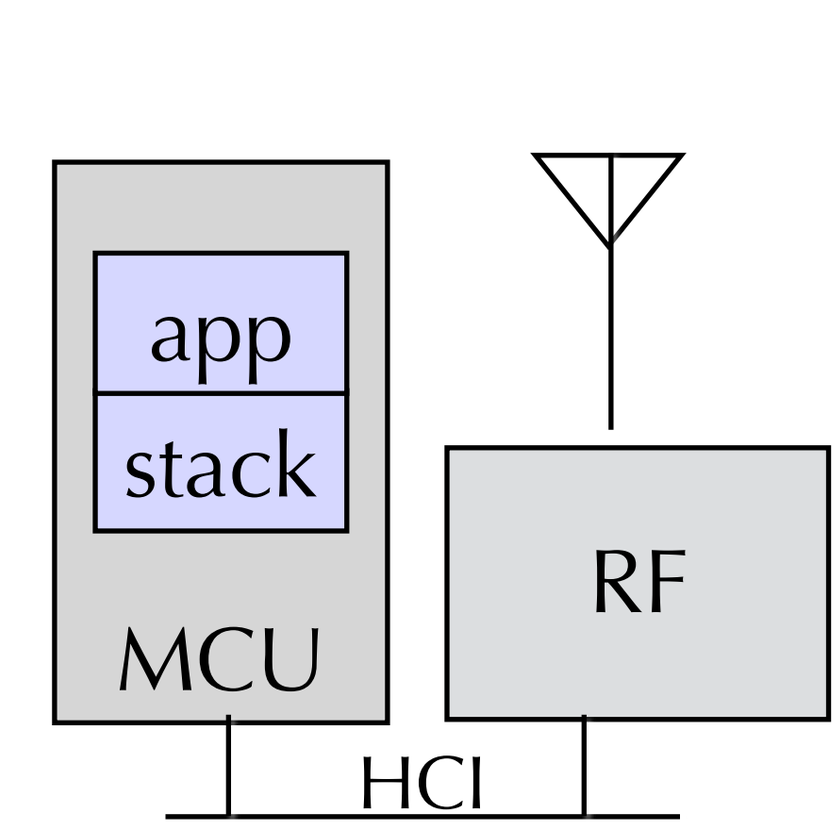


"Network Co-Processor" configuration

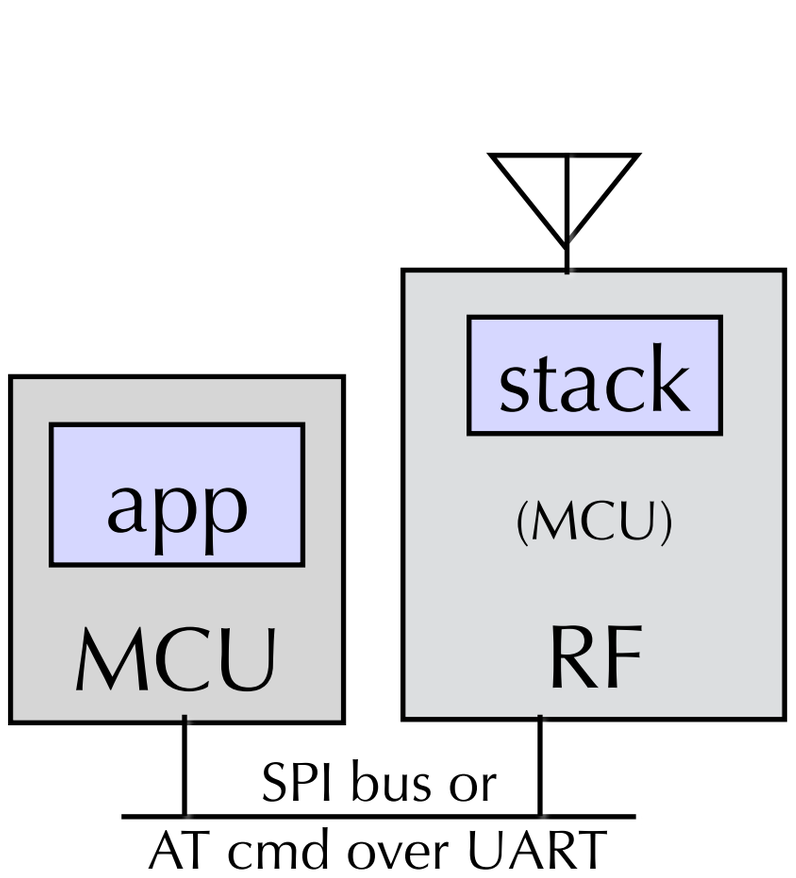


"Single-Chip" configuration

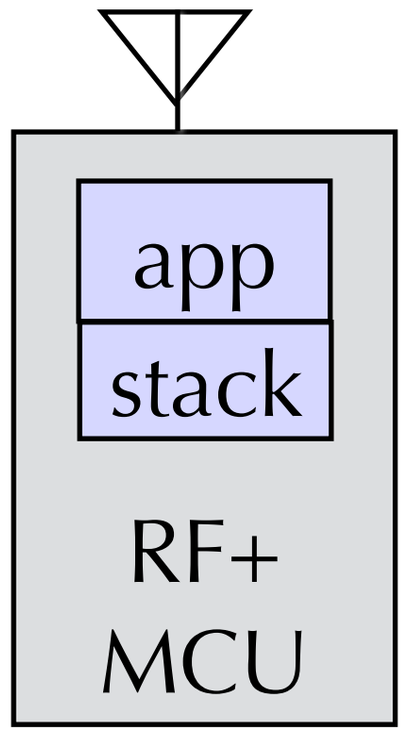
What about Bluetooth and WiFi?



How most PC, Linux, general-purpose systems do Wi-Fi & BT



Many embedded systems today



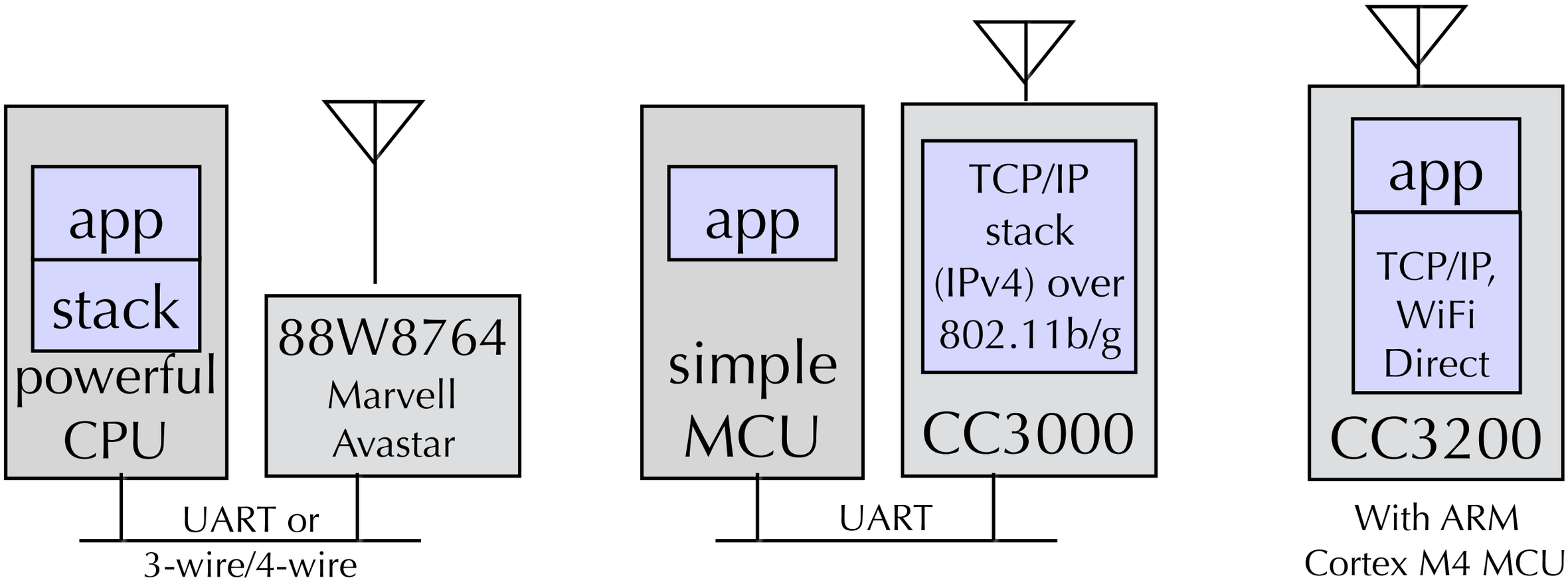
Starting to appear on market for embedded systems

CC2560 and CC2564

http://www.ti.com/product/cc2560

- CC2560: Bluetooth BR/EDR (“Classic”)
  - RF + stack
  - Includes audio codec
- CC2564: Dual-mode Bluetooth 4.0
  - BR/EDR
  - Bluetooth Low Energy (BLE)
  - ANT protocol (but not BLE at the same time)

Wi-Fi comparisons

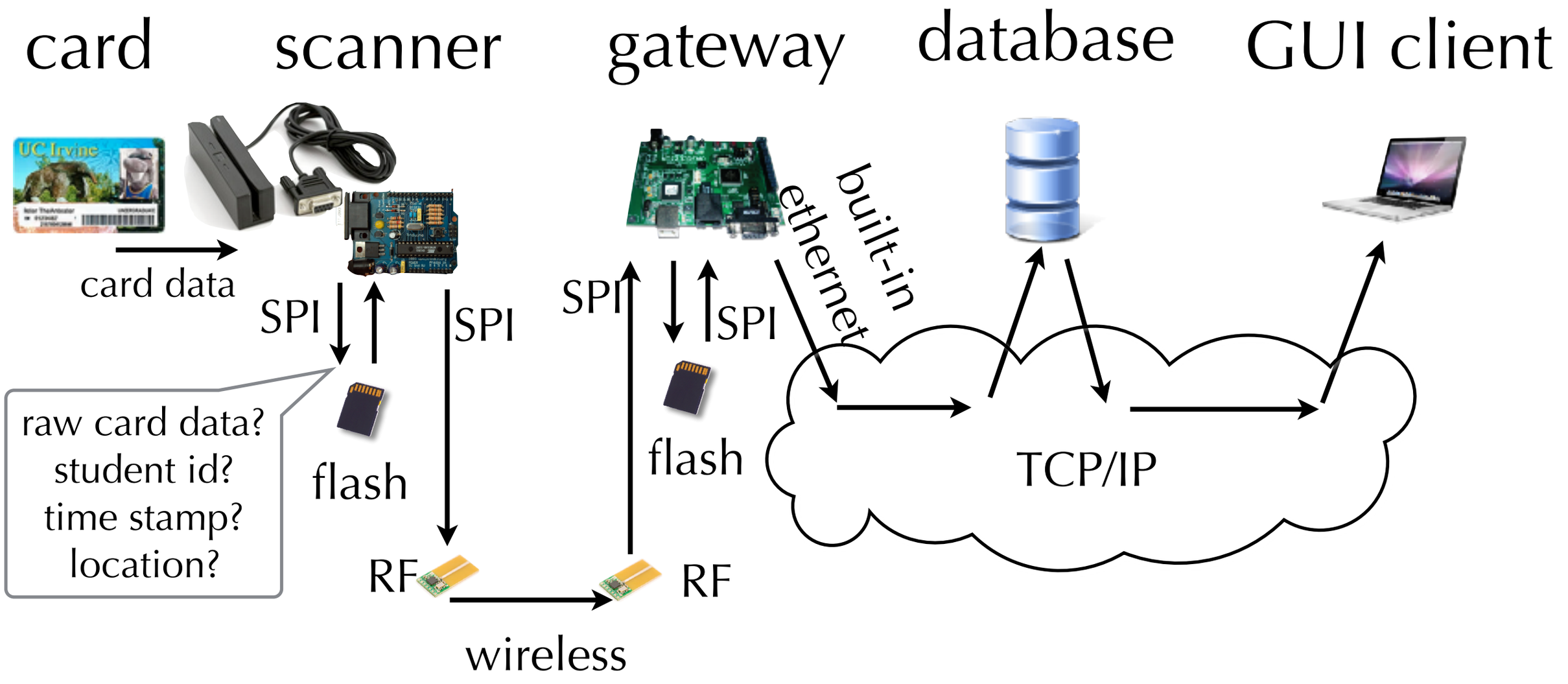


"Host-stack" configuration (for most PCs)

"Network Processor" configuration

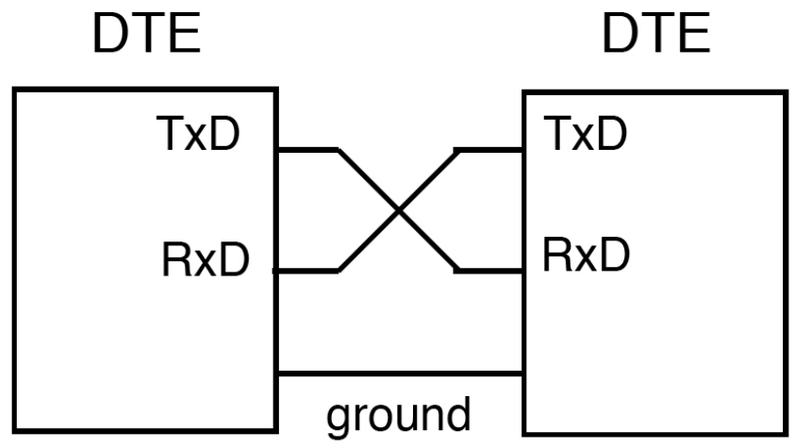
"Single-Chip" configuration

Generalizing to Higher-Level Communication



Minimal serial connection

- 3 wires:
  - TxD (transmitted data)
  - RxD (received data)
  - GND (ground)



- Crossover
  - TxD connected to RxD of the other, and vice versa

Asynchronous hardware protocol

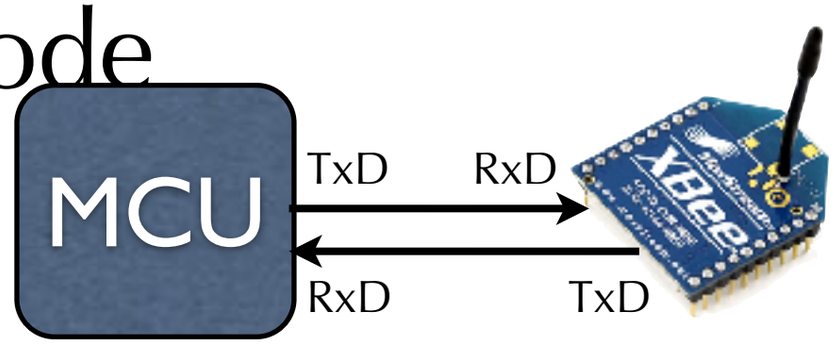
- During no signal: kept high
  To start: "start bit" goes low
- Each side locally generates its own clock
  - both sides must agree on clock rate
    => synchronous as hardware during data transfer phase, no acknowledgment!
    => sender doesn't know if receiver got it
- 1 or 2 Stop bits (high), space (low)

Drivers over UART devices

- Send/parse bytes!
  - Usually no separate interrupt line from device
    -- everything pushed from TxD
- Driver: FSMs
  - One for TxD, one for RxD
  - Need to track timing and state

e.g. Driver: FSMs to handle AT Commands

- Initially: Command mode
- Data mode: pass thru



  - "+++" followed by 1-second pause => switch to Command mode
- Command mode: interpret commands
  - "O" to switch to data mode

Trade-offs betw SPI & UART for RF modules

- SPI
  - Share SPI bus with other SPI devices
  - Easy to work with: /CS delimiter, Command followed by data
  - Separate line needed for slave to interrupt MCU
- UART / AT-commands
  - Can type interactively from PC terminal program (assume RS232/UART conversion)
  - Need to track command/data mode
  - Dedicated UART to the RF, not sharable

Idea for RF Driver API (1/3)

- Define your own API: Config, Tx, Rx
- int RF_Driver_Config(params..);
  - turn on, turn off
  - get various status
  - set self ID, set recipient ID, set payload size, ..

Idea for RF Driver API (2/3)

- int RF_write(octet *buf, int *len);
  - called by user to write #bytes to transceiver (over SPI or AT commands over UART, etc; user shouldn't care)
  - non-blocking? returns status code (e.g., queued_ok; buffer-overflow; ..)
- Issues: reentrant?

Idea for RF Driver API (3/3)

- int RF_read(octet *buf, int *len);
  - actively read the buffer content, returns status code, sets nbytes
- int RF_Rx(void(*p)(octet *buf, int *nbytes))
  - registers callback when it receives data
  - the callback should update FSM

Layers of Abstraction

software

Application — *submit entry, configure, set clock, lookup by name/time/id/...*

(presentation) — *encryption?*

Session — *grant access(deputize), refresh, close session*

Transport — *send, receive, abort*

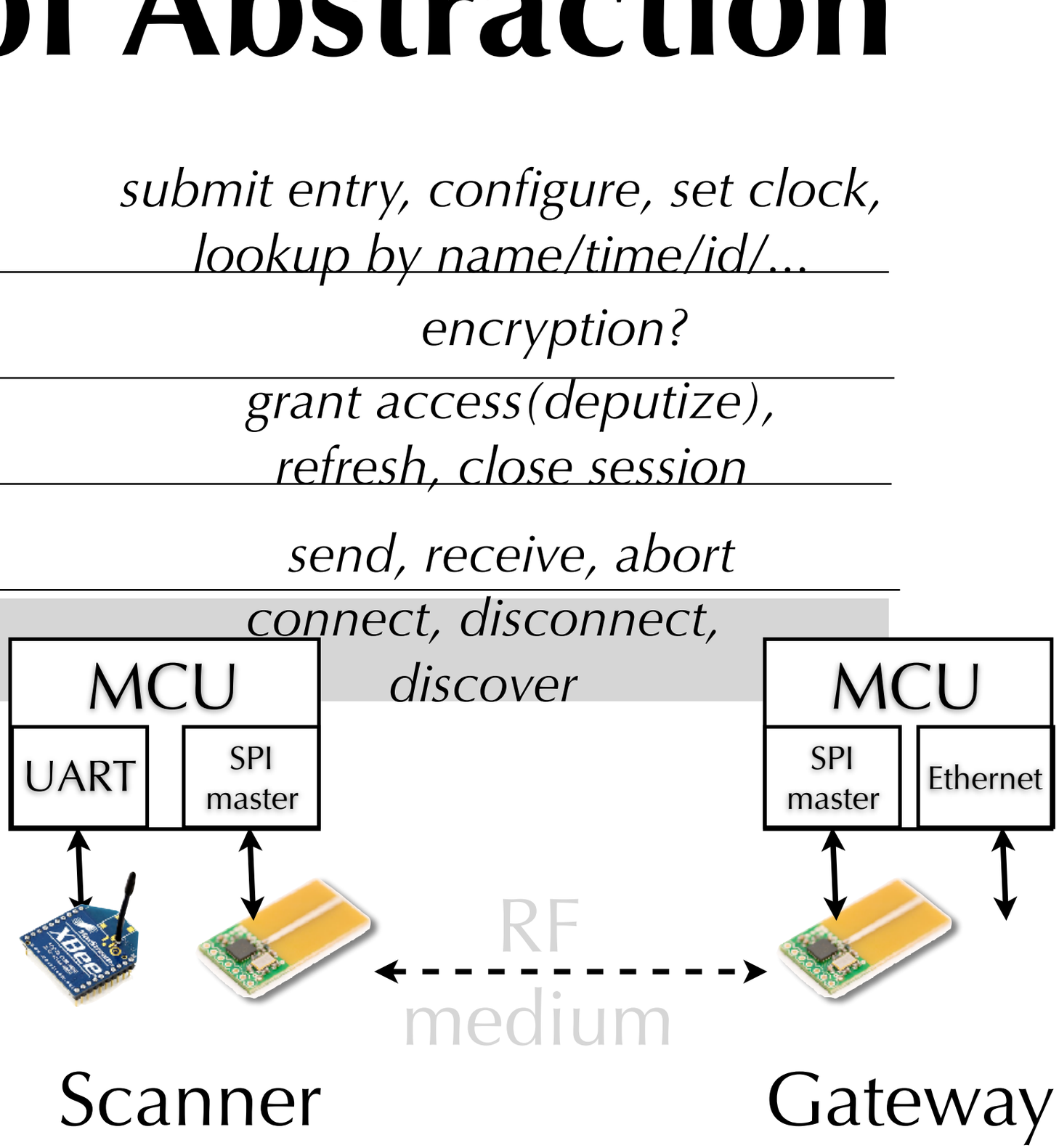
Network — *connect, disconnect, discover*

Driver

hw transactions

MAC/Physical

hardware



From last week: driver wrapping

- Blocking vs. Nonblocking calls
  - Blocking: doesn't return until completion
  - Nonblocking: queue it, separate call to check completion
- Callback: register a function pointer
  - The driver calls the callback function (that you set)

Function pointers in C

- A regular function
  - `int foo(int i) { return i + 2;}`
- A pointer to a function like foo
  - `int (*ptr)(int i);`
  - `ptr` is the name of the variable
  - you can say `ptr = foo;`
  - to call, can do either `ptr(23);` or `(*ptr)(23);`

Network Layer

- Connection model
  - Symmetric? active/passive connection? Discovery of what's available to connect?
- Node addressing
  - hardware ID vs network address? DHCP-like assignment or statically configured?
- [no need] Message forwarding

Network Layer API

- Static: the user configures the gateway ID, scanner ID
  - SetGatewayID(id_t)
  - SetSelfID(id_t)
  - node_t Connect(id_t)
  - Disconnect(node_t)
- Dynamic: node/gateway discover each other
  - Discover(id_t[ ])
  - GetGatewayID(id_t*)
  - GetSelfID(id_t*)
  - Connect, Disconnect (same as static version)

typedef

id_t:	hardware address
node_t:	network address

Transport Layer

- Reliable transmission vs. unreliable (for nRF24L01, just configure hardware)
  - In-order delivery
- Optional (not needed here)
  - congestion avoidance
  - flow control (stop sender if buffer full)
  - byte orientation, ports

Transport Layer API (1/2)

- SendTo(node_t dest, octet *buf, int *nb)
  - Send packet to node; could fail if node handle is invalid, or lost connection
- Q: blocking or nonblocking?
  - If nonblocking, need another function to check status of send

Transport Layer API (2/2)

- Blocking version
  - RcvFrom(node_t, octet *buf, int *nb)
  - wait to receive a packet from node; or, from anyone
    - For testing only! Don't do this in real code.
- Nonblocking
  - RcvOn(void (*callback)(node_t, octet *buf, int nb))
  - register a callback when a packet is received.
  - might want to filter unregistered calls

Session Layer

- Open/Close a session
  - Request: with ID, password, key exchange...
  - Reply: Grant for a duration, set duration, time-synchronize, or deny request
  - Close session
- Refresh session periodically (keep alive), time-resynchronize

Presentation Layer (data representation)

- Encryption/Decryption
- Byte ordering
- Data structure matching (could be XML)
- Provide the same Send/Receive calls - pass through if no encryption or encoding

Application Layer

- Should accept commands almost any time
  - Configure, Process data query
- Track state of operations
  - Manage connectivity
  - Scan, log data, transmit when possible
  - Manage power!

Application Layer - program (scanner)

```
init(devices, network)
repeat forever {
  if ((e=deq())!=nil) {
    sleep; continue;
  }
  switch (type(e)) {
  case SCAN_EVT:
    readData(&d);
    enq(LOG_EVT, d);
    enq(TX_EVT, d);
    break;
```

```
  case TX_EVT:
    if (!connected()) {
      tryReconnect();
      requeue(e);
    } else { TX(...); }
    break;
  case LOG_EVT:
    if (SD_busy()) {
      requeue(e);
    } else { SD_wr(...); }
    break;
```