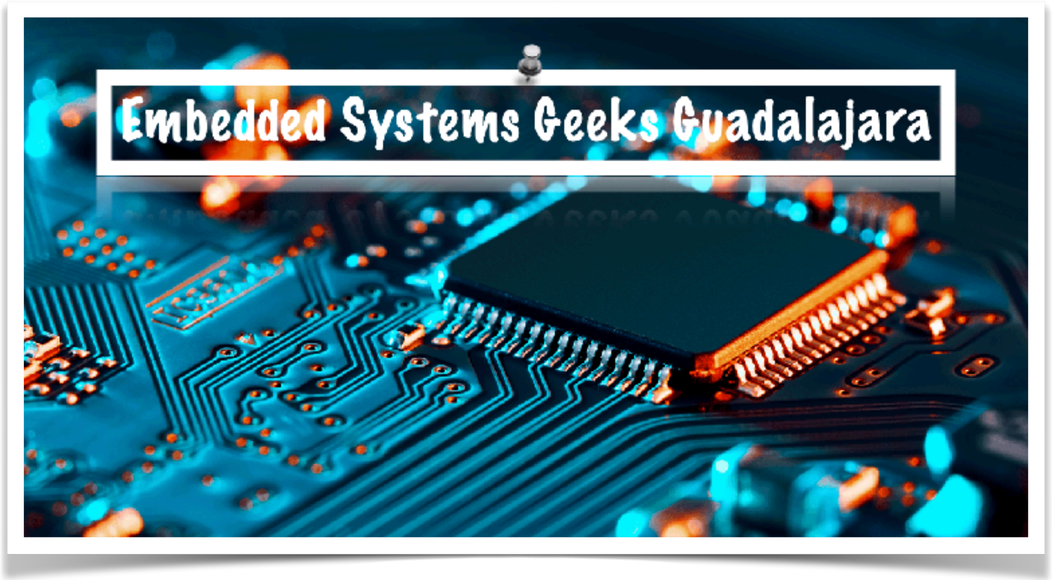
Embedded Systems Geeks GDL
Arduino & AVR: a deep(er) dive

Dr Frank Zeyda - 19 September 2023 @ HackerGarage GDL



Frank Zeyda

Dipl.-Inform. BSc PhD PgCLTHE

Independent Consultant
Safety-Critical Systems

https://www.linkedin.com/in/frank-zeyda/

frank.zeyda@gmail.com

Research Interests:

- Semantic Foundations / Unifying Theories (UTP)
- Formal Methods and Verification
- Rigorous Digital Engineering
- Automated Theorem Proving (Isabelle/HOL)

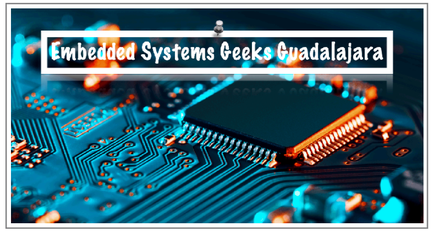
Academic Involvement (2001 to 2018):

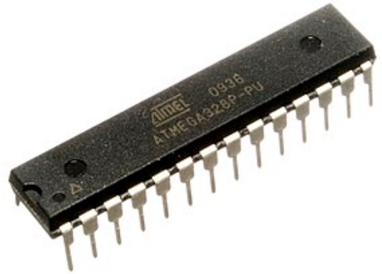
- PhD from Teesside University (UK) in 2007.
- Research Associate/Fellow at the University of York (UK).
- Senior Lecturer at Teesside University and the University of York (UK).
- Lead RA on several EPSRC-funded and EC-funded research projects.
- Most publications are listed on: **https://dblp.org/pid/23/6727.html**

Industrial Involvement (1999 to 2000 & 2018 to 2022):

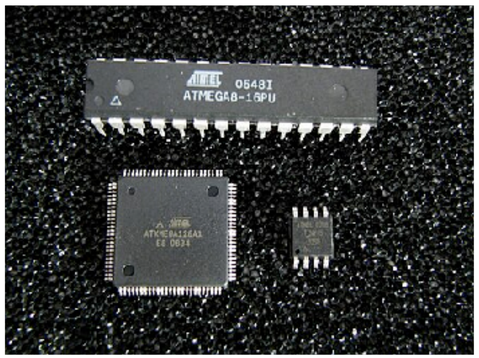
- Siemens Mobility / Transportation Systems (Germany)
- Verified Systems International GmbH (Germany)
- Galois, Inc. (U.S.) (consultant on industrial R&D projects)

Introduction

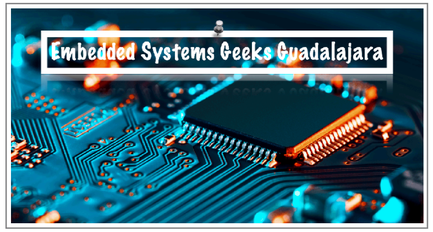




- Topic of this talk: learn more about the **AVR architecture**
- Arduino: name of the “brand” and prototyping boards
- AVR: architecture of the underlying microcontroller family
- Commonly accepted AVR stands for **A**lf and **V**egard's **R**ISC processor, although Atmel denies it is an acronym (conceived by **two students**)
- Original AVR MCU developed by Nordic VLSI / Semiconductors (Norway) and later sold to Atmel Norway (subsidiary of Atmel).
- First AVR microcontrollers (8 bit) emerged in 1997.
- Atmel had shipped 500 million AVR flash microcontrollers.

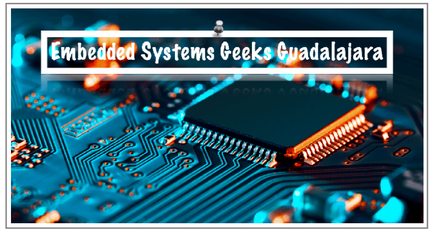


Overview of the talk



- Arduino & AVR Architecture 101
- Knowing your Chip
- **Example**: OLED Device Driver
- Communication Protocols
- Miscellaneous Topics
- Conclusion & Further Reading

AVR Architecture 101



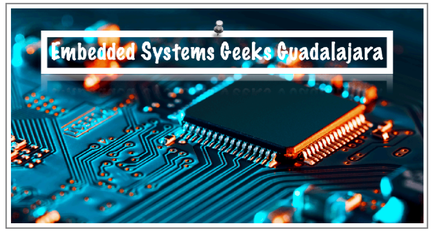
- Modified **Harvard** architecture: programs and data are stored in separate memories.
- Compare this to **von Neumann** architecture, where **all memory** is capable of storing program instructions
- Example: ATmega328P (Arduino Uno) has …

  ➡ 32 KB of ISP flash memory for programs, but only

  ➡ 2 KB of static RAM (SRAM) for dynamic data
- Flash memory is usually **protected** during program executions (why?) and not meant to be written.

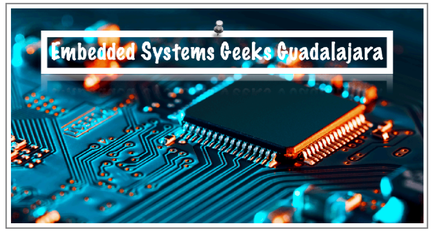
  ✦ But, apparently, this may *still* be possible with a custom boot loader …

Let’s talk more about memory



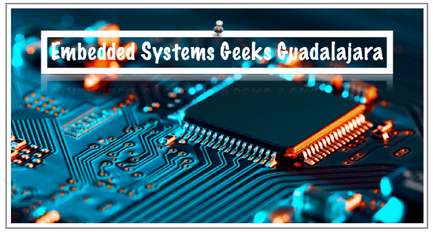
- When doing **Desktop** / **Cloud** application development, we usually do not think (very much) about memory (my MacBook has **64 GB**).
- Perhaps those developing **web** or **mobile** apps are more tuning into this.
- But this is still nothing compared to the restrictions on some MCUs.
- **Problem**: How to work with:
  - **32 KB** of program space (flash, **read-only**)
  - **2 KB** of (S)RAM for data (volatile, **read/write**)
  - **1 KB** of EEPROM (non-volatile, **read/write**)
- **Solution**: We have to be creative!

Let’s talk more about memory



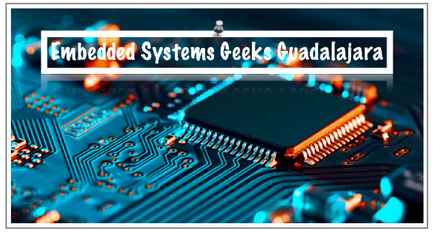
- Memory utilisation economy might imply …
  - ✓ Strip dead **unused** functions / code from executable
  - ✓ Only put into SRAM what needs to be there … (e.g., constant data can be in program memory)
  - ✓ Beware of using **large** data structures (arrays, look-up tables, etc.)
    - ➡ E.g., ask yourself: do we **really need** a frame buffer?
  - ✓ Strip all debug symbols and debug information (no gcc -g …)
  - ✓ Code thoughtful and efficiently (modularisation, optimization, reuse)
    - ➡ What matters is the size of the binary, not the source code.

Let’s talk more about memory

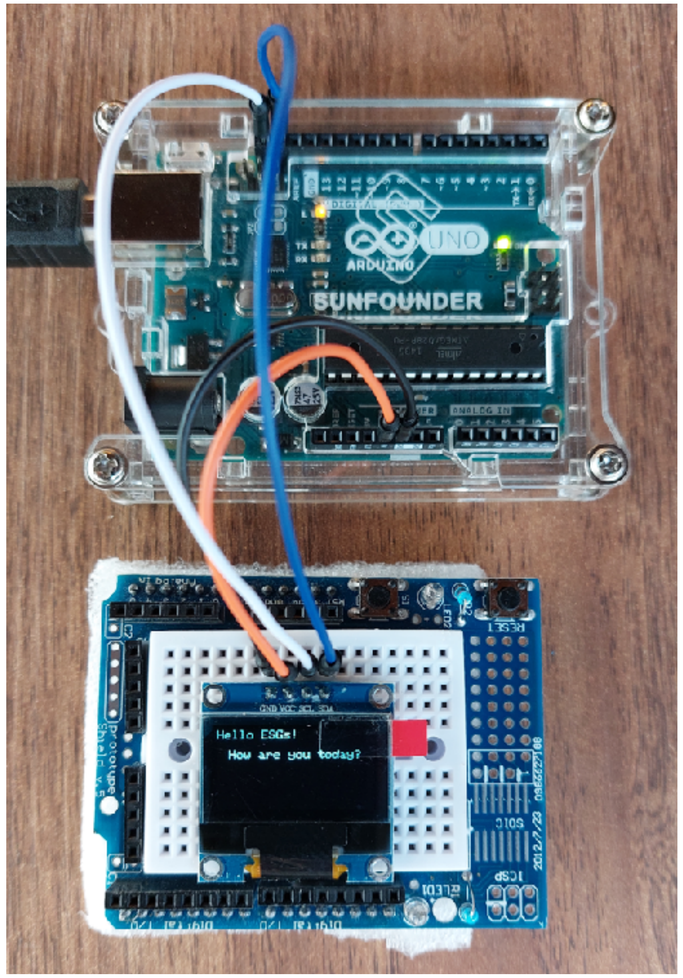


- Some personal experiences:
  - Third-party and open-source code is often needlessly bloated.
  - **Example**: Adafruit GFX library takes away 1 KB of SRAM for a frame buffer, i.e., when using a 128x64 pixel OLED display (display needs **no refresh**).
- Sometimes, ready-made components do **more than needed**
  - **Example**: full I²C compliance must support multiple masters & slaves, arbitration, clock stretching, read/write, etc.
  - But if there exists only **one** master & slave with a simple one-directional communication pattern, we do not need all of that …
- Recursion can be **dangerous** due to the limited stack size.
- We often have a trade-off between **speed** and (memory) **resource efficiency.**

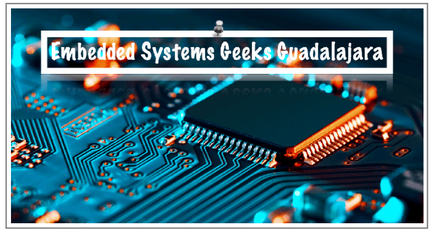
Example: Not using flash memory




```
1   #include "oled.hpp"
2
3   /* Implicitly, all constants are put into SRAM. */
4   static const char s1[] = "Hello ESGs!";
5   static const char s2[] = "How are you today?";
6
7   /* The setup() function is called only once by the AVR run-time. */
8   void setup() {
9     oled::init();
10    oled::clear();
11  }
12
13  /* The loop() function is called repetitively by the AVR run-time. */
14  void loop() {
15    oled::move(0, 0);
16    oled::print(s1);
17    oled::move(2, 10);
18    oled::print(s2);
19    while (true) { /* wait forever */ }
20  }
```



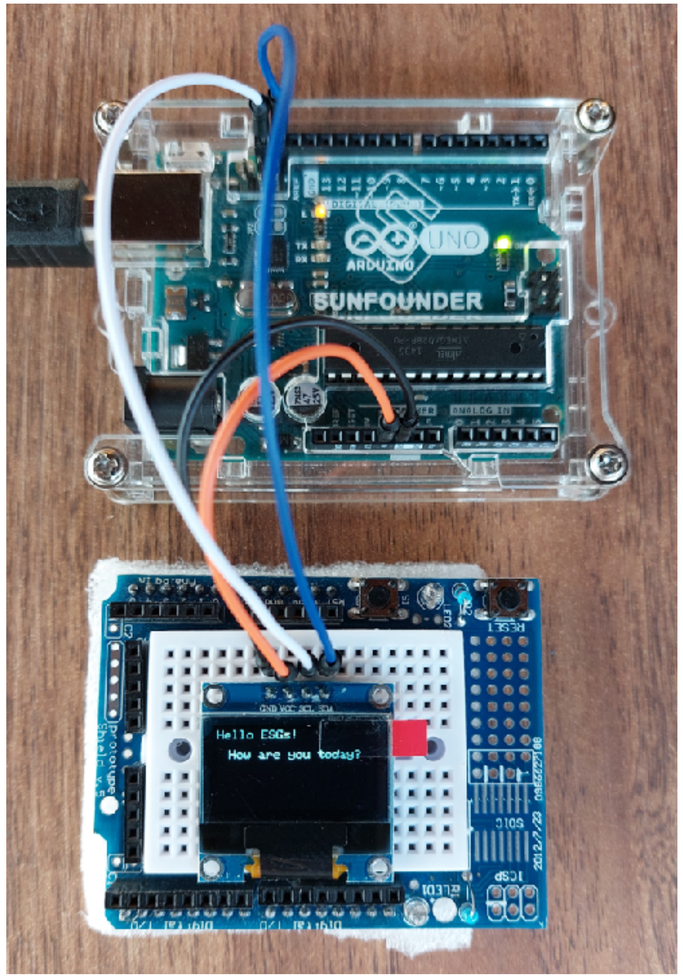
Example: Not using flash memory



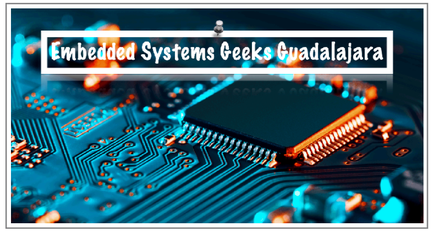

```
1   #include "oled.hpp"
2
3   /* Using the below with oled::print_p(...) saves precious SRAM. */
4   static const char PROGMEM s1[] = "Hello ESGs!";
5   static const char PROGMEM s2[] = "How are you today?";
6
7   /* The setup() function is called only once by the AVR run-time. */
8   void setup() {
9     oled::init();
10    oled::clear();
11  }
12
13  /* The loop() function is called repetitively by the AVR run-time. */
14  void loop() {
15    oled::move(0, 0);
16    oled::print_p(s1);
17    oled::move(2, 10);
18    oled::print_p(s2);
19    while (true) { /* wait forever */ }
20  }
```

Add attribute for linker to place data into flash

Custom print function that read strings from program/flash memory

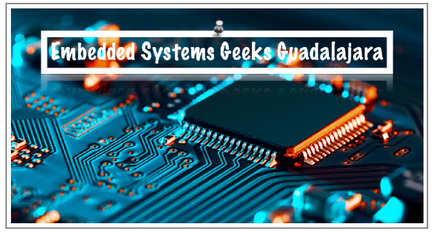


Difference in Compilation Output



- TODO: Show difference in compilation output here!

Accessing Program Memory (API)

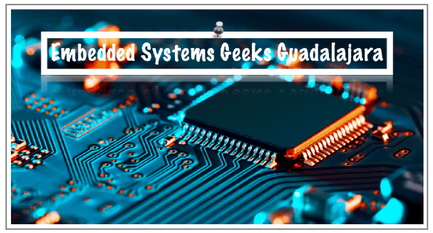


- Side-by-side comparison of `print(…)` and `print_p(…)`:

```
uint8_t print(const char* s) {
  if (s != nullptr) {
    while (*s != '\0') {
      const uint8_t error = emit(*(s++));
      if (error != 0) { return error; }
    }
  }
  return 0; // no error if reaching here
}
```

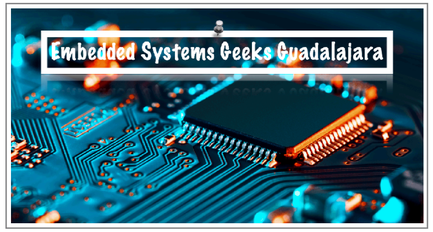
```
uint8_t print_p(const char* s) {
  if (s != nullptr) {
    char c = pgm_read_byte(s++);
    while (c != '\0') {
      const uint8_t error = emit(c);
      if (error != 0) { return error; }
      c = pgm_read_byte(s++);
    }
  }
  return 0; // no error if reaching here
}
```

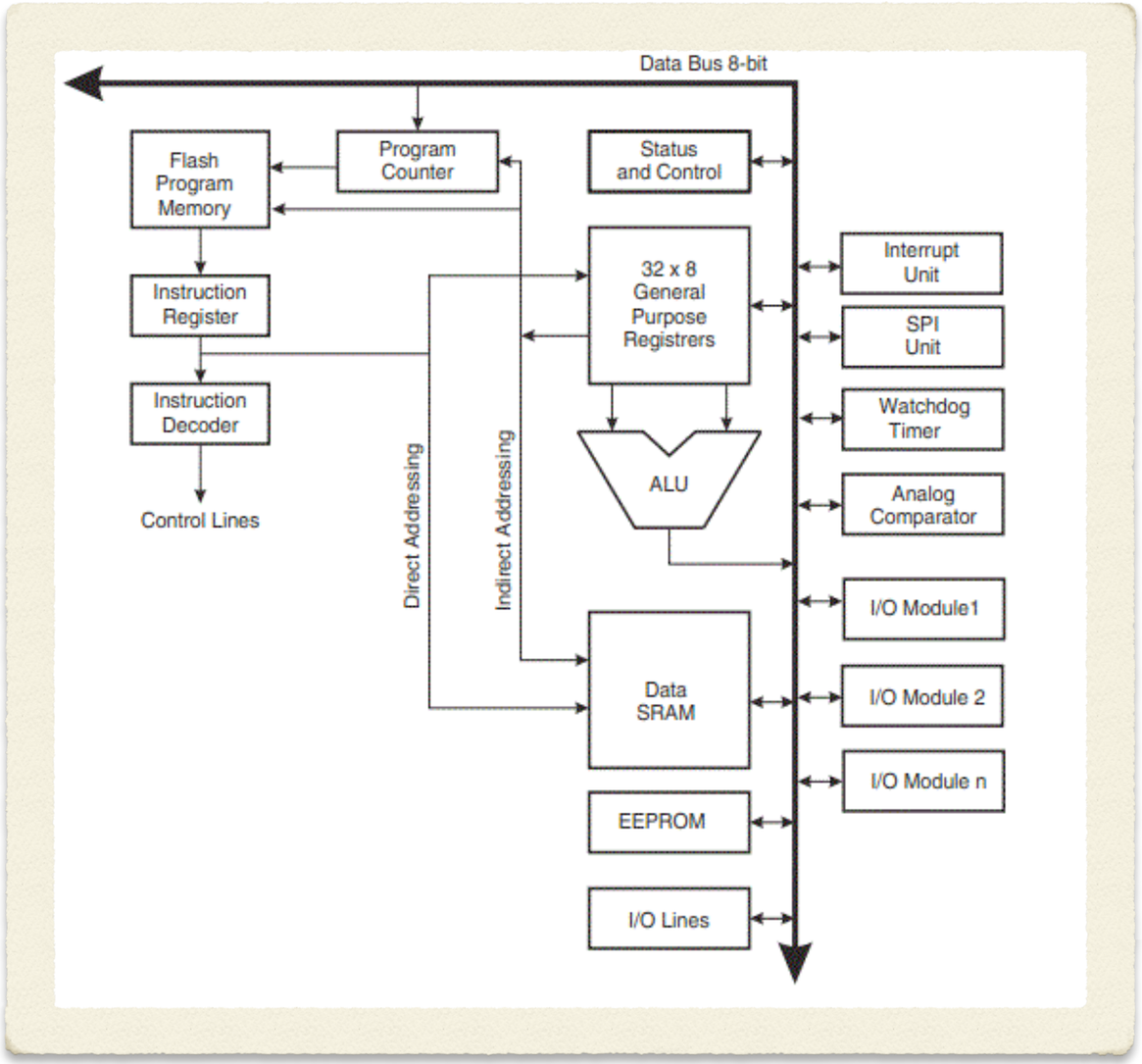
Memory on the Atari 2600



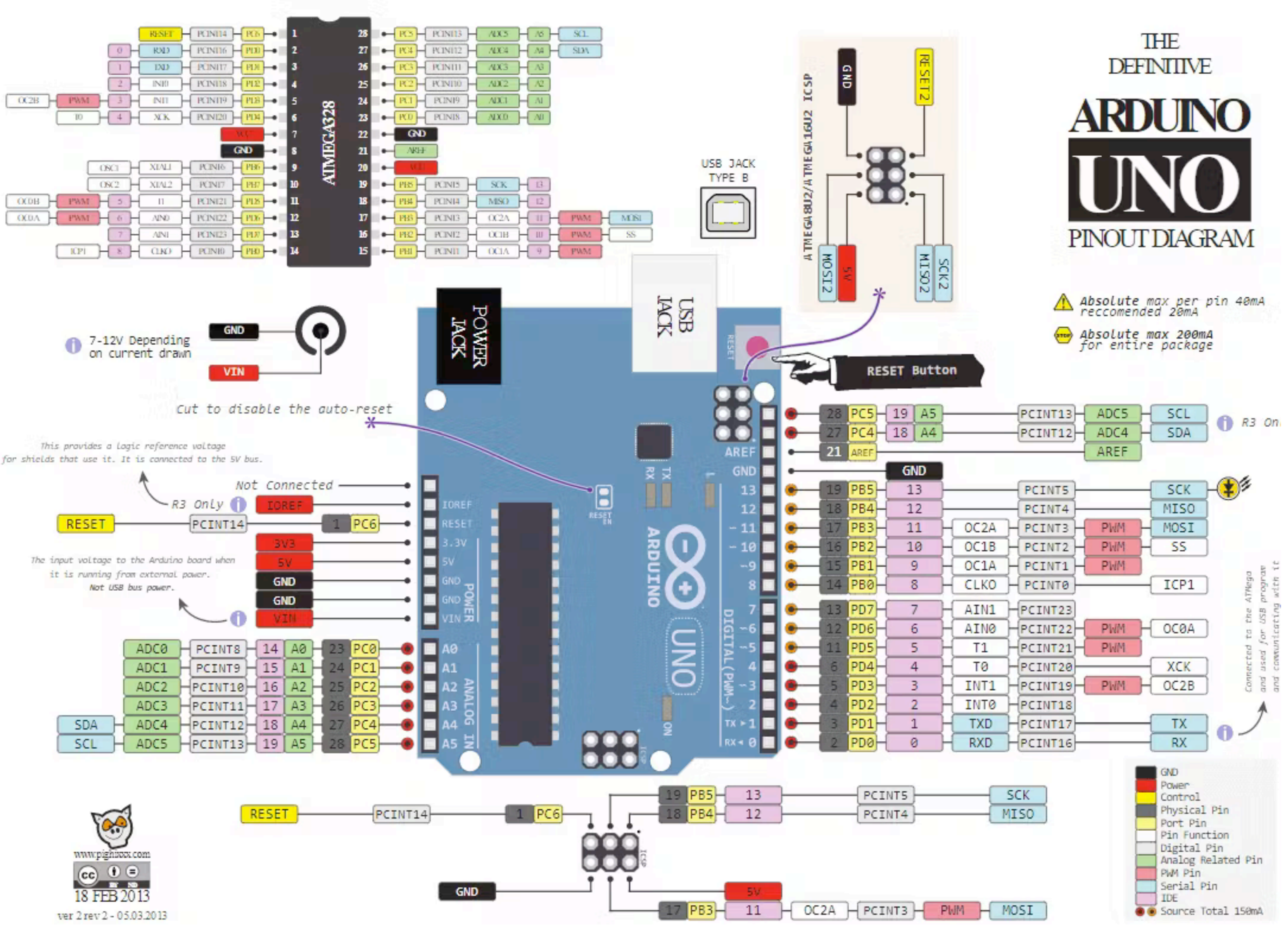
- But consider yourself lucky. On the Atari 2600 …
  - … we only had **128 bytes** of writable memory
  - … most early games had to fit into 4 KB of program memory
  - … the console had **no frame buffer** — the CPU had to write data into the Audio & Video Processor (**TIA**) during HW scan-line generation.
  - … this leaves limited time to run **game logic** (**mostly V/H blank**)
  - … and there was no **DMA** and surely no **GPU** or **coprocessors** …
- Still developers managed to create some cool games for the system.

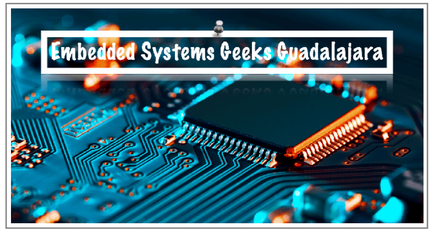
ATmega328 Internals



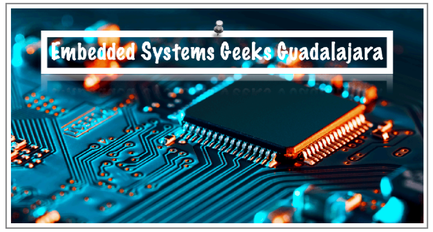


- **32** General Purpose Registers
  - **8-bit** data width
- 20 MHz clock frequency
- Interrupt Control Unit (**ICU**)
- 8-bit / 16-bit Timers
- Programmable serial **U(S)ART**
- Master/slave **SPI** serial interface
- Byte-oriented 2-wire serial interface
  - Phillips **I²C** compatible
- **PWM** & 10-bit **ADC** channels (six)
- 23 programmable I/O lines
- Watchdog Timer


THE DEFINITIVE
ARDUINO UNO
PINOUT DIAGRAM
ATMEGA328
ATMEGA8U2/ATMEGA16U2 ICSP
USB JACK TYPE B
POWER JACK
USB JACK
RESET Button
7-12V Depending on current drawn
GND
VIN
Cut to disable the auto-reset
Absolute max per pin 40mA reccomended 20mA
Absolute max 200mA for entire package
This provides a logic reference voltage for shields that use it. It is connected to the 5V bus.
Not Connected
R3 Only
The input voltage to the Arduino board when it is running from external power.
Not USB bus power.
Connected to the ATMega and used for USB program and communicating with it
www.pighixxx.com
18 FEB 2013
ver 2 rev 2 - 05.03.2013
GND
Power
Control
Physical Pin
Port Pin
Pin Function
Digital Pin
Analog Related Pin
PWM Pin
Serial Pin
IDE
Source Total 150mA


Embedded Systems Geeks Guadalajara

The Anatomy of an MCU



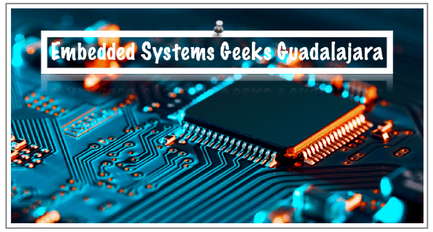
- We already learned that MCUs can be resource-constrained:

  ➡ The AVR family is also includes larger memory MCUs (**megaAVR**, etc.)
- On the other hand, many MCUs come with a lot of useful embedded components to communicate with **other** hardware devices:

  ➡ Natively speaking UART, SPI, I²C, even IEEE 802.11 WiFi

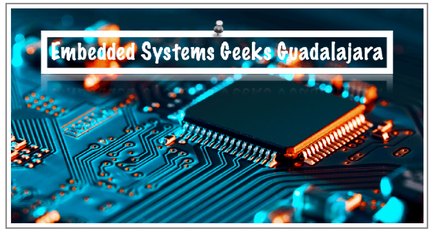
  ➡ See this LINK for an useful set of slides on this topic.
- The **RP2040** MCU (Raspberry Pico) even provides state-machine **coprocessors** to implement your own low-latency protocols (e.g. VGA) …
- The enables us to trade **bit-banging** for **byte-banging** …
- Together with DMA, this causes next-to-no CPU load (**no busy waiting**)

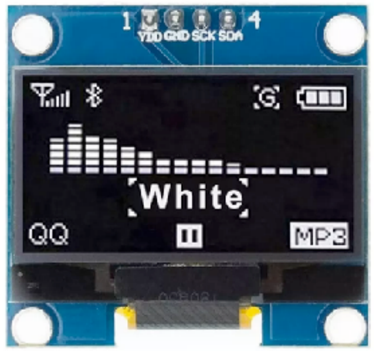
Knowing your Chip



- Application developers commonly have to study **API** and **Framework** documents ….
- Hardware engineers are used to spend a lot of time reading **data sheets**.
- These can be intimidating at first sight …
  - Often go into **thousands** of pages (Pepe?).
- Data sheets are almost **never read** from start to finish …
  - Typically, you just pick the section that is relevant to your problem!
- Data sheets vary in terms of quality, completeness, usability, etc.
  - Do not dispair. Sometimes, **wikis**, **blogs**, and **forums** offer more help.
- Below a link to the data sheet for the **ATmega328** (294 pages)**:** https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

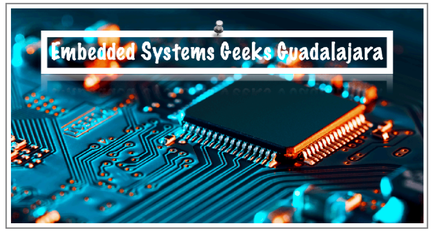
Knowing your Chip (cont'd)

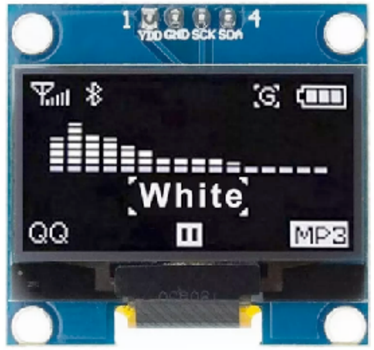




- **Example**: assume you want to drive an **OLED** display via **I²C**
- These are the **questions** you need to answer:
  - Does the chip natively support **I²C** or do we have to implement it *from scratch* (e.g, via bit-banging)?
  - Does the Arduino API already provide a general **driver** for **I²C**?
  - What is the *particular* **protocol** to communicate with the display?
  - Does a **library** for communication with the display already exists?
  - If so, does it fulfil our requirements in terms of **capabilities**, **resources**, **configurability**, and so on.

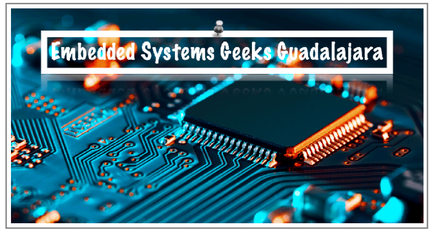
Knowing your Chip (cont'd)





- **Example**: assume you want to drive an **OLED** display via **I²C**
- These are the **questions** you need to answer:
  - Does the chip natively support **I²C** or do we have to implement it *from scratch* (e.g, via bit-banging)? → **YES**
  - Does the Arduino API already provide a general **driver** for **I²C**? → **YES**
  - What is the *particular* **protocol** to communicate with the display?
    - Uses the SDD1306 driver/controller chip (SSD1306.pdf)
  - Does a **library** for communication with the display already exists? → **YES**
  - If so, does it fulfil our requirements in terms of **capabilities**, **resources**, **configurability**, and so on. → **YES/NO? DEPENDS ON PROJECT …**

Knowing your Chip (cont'd)

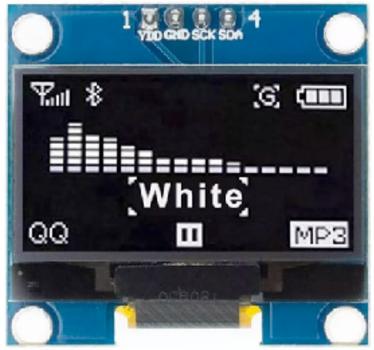


- **Advice**:
  - Using a ready-made library can sometimes safe us **time** and **trouble** to get something working (more) quickly.
  - If we want a more robust / specialised / maintainable / bespoke driver, we **might** have to write it ourselves.
  - Like with all third-party and open-source software, expect **bugs**.
  - Embedded programming gives us the choice to either …

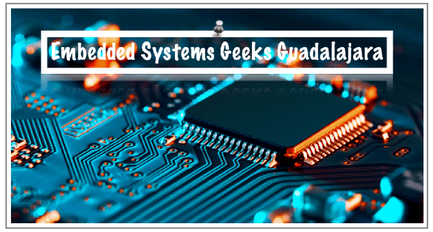
✓ work at a **higher** abstraction level via API / components

✓ work at a **lower** abstraction level via direct I/O access

✓ try to balance **effort** against **benefit** of reinventing stuff (!)

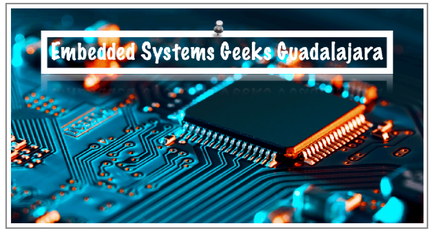


Example: OLED SSD1306 Driver



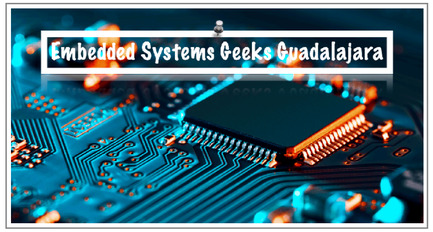
- Driver that I started to write with (memory) economy in mind.
- In most cases, we do **not** need a **frame buffer** …
- Safes us **1 KB** of precious SRAM, i.e, for a 128x64 dots OLED.
- Also a good exercise to learn about **I²C** (more on this to follow)
- **Question**: When *do* we need a frame buffer?
  - ➡ Device does **not hold** display data by itself (needs refresh)
  - ➡ Data can only be written in **words** (not individual **bits**) …
    … and we require to draw objects on top of each other!
  - ➡ We need to **read out** display data for one reason or another
    (e.g., to save the image in a paint program, hibernate device, etc.)

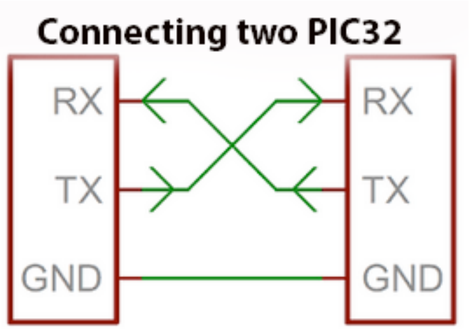
Communication Protocols



- How do protocols *in general* differ from each other?
  - **analogue** (modulated) vs **digital** (low / high)
  - **synchronous** (clocked) vs **asynchronous** (self-clocking)
  - permissible data **transmission rate** (frequency)
  - number of **signals** / **wires** required; support for duplex?
  - number of **senders** (masters) and **receivers** (slaves) allowed
  - built-in **error checking** mechanisms?
  - **electrical characteristics** (voltage, sensitivity to noise, and so on)
  - underlying **protocol specification** and **standardisation**

Protocol Example: UART



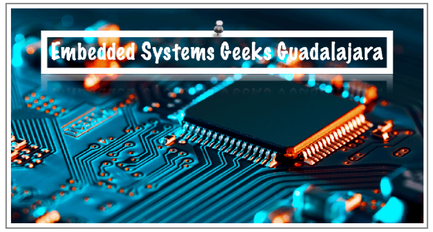


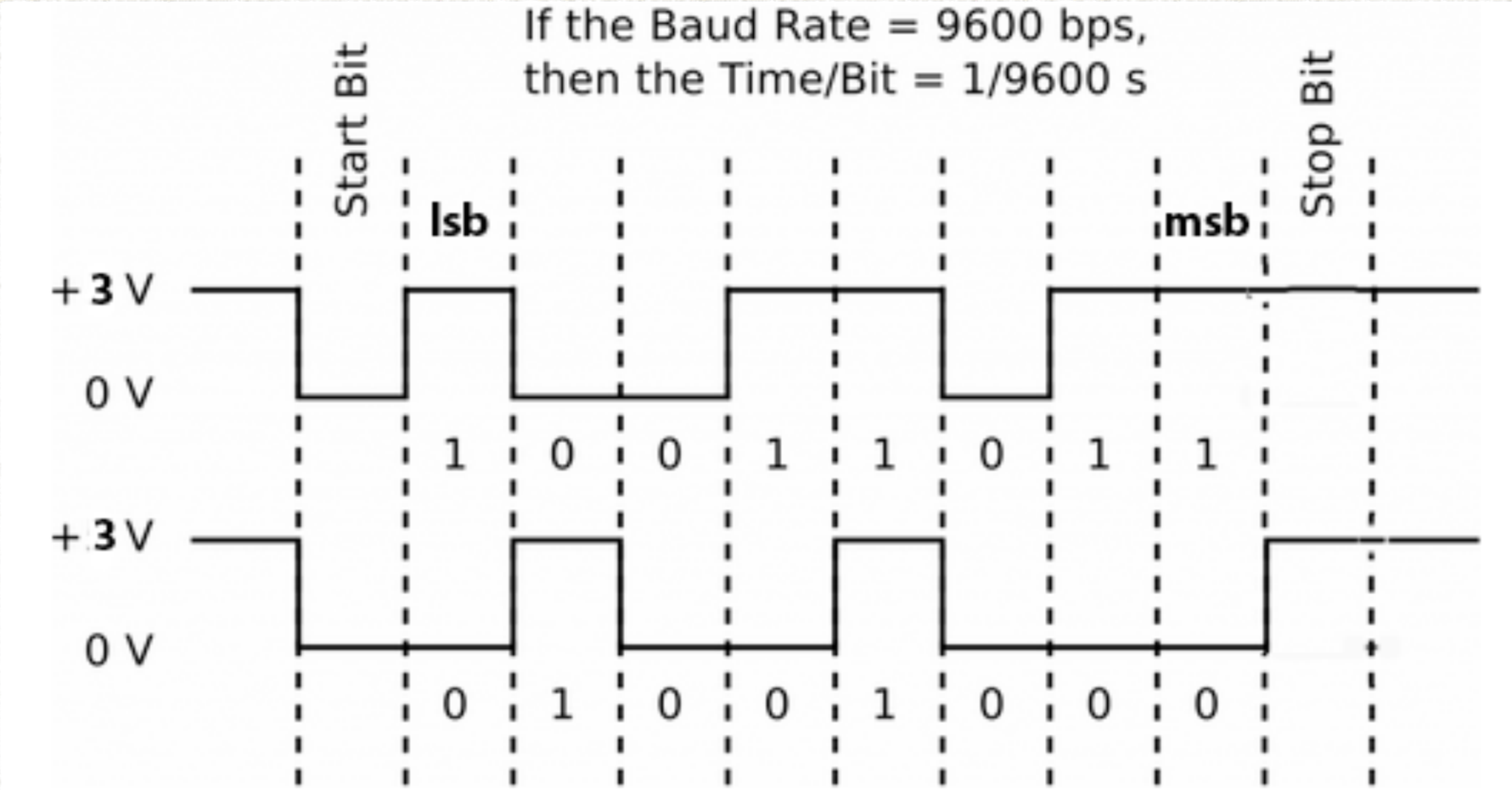
- UART = **U**niversal **A**synchronous **R**eceiver-**T**ransmitter
- Requires two wires for full bidirectional transmission:

  ➡ **TX** (transmit) and **RX** (receive), plus **GND** (ground wire).
- **Asynchronous**, using a fixed predetermined BAUD rate.
- Typical speeds are 9,600, 19200, 38400, 57600, and 115200 bps.
- Connects **two devices**, no multi-master or multi-slave by design.
- Supports error checking via a **party bit**.
- Supported natively by the Arduino / AVR family (!)

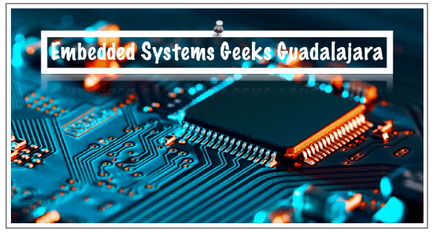
  ➡ Actually used, AFAIK, to upload programs via the Arduino boot-loader …

Protocol Example: UART



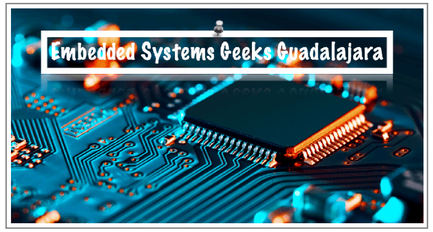


Protocol Example: I²C

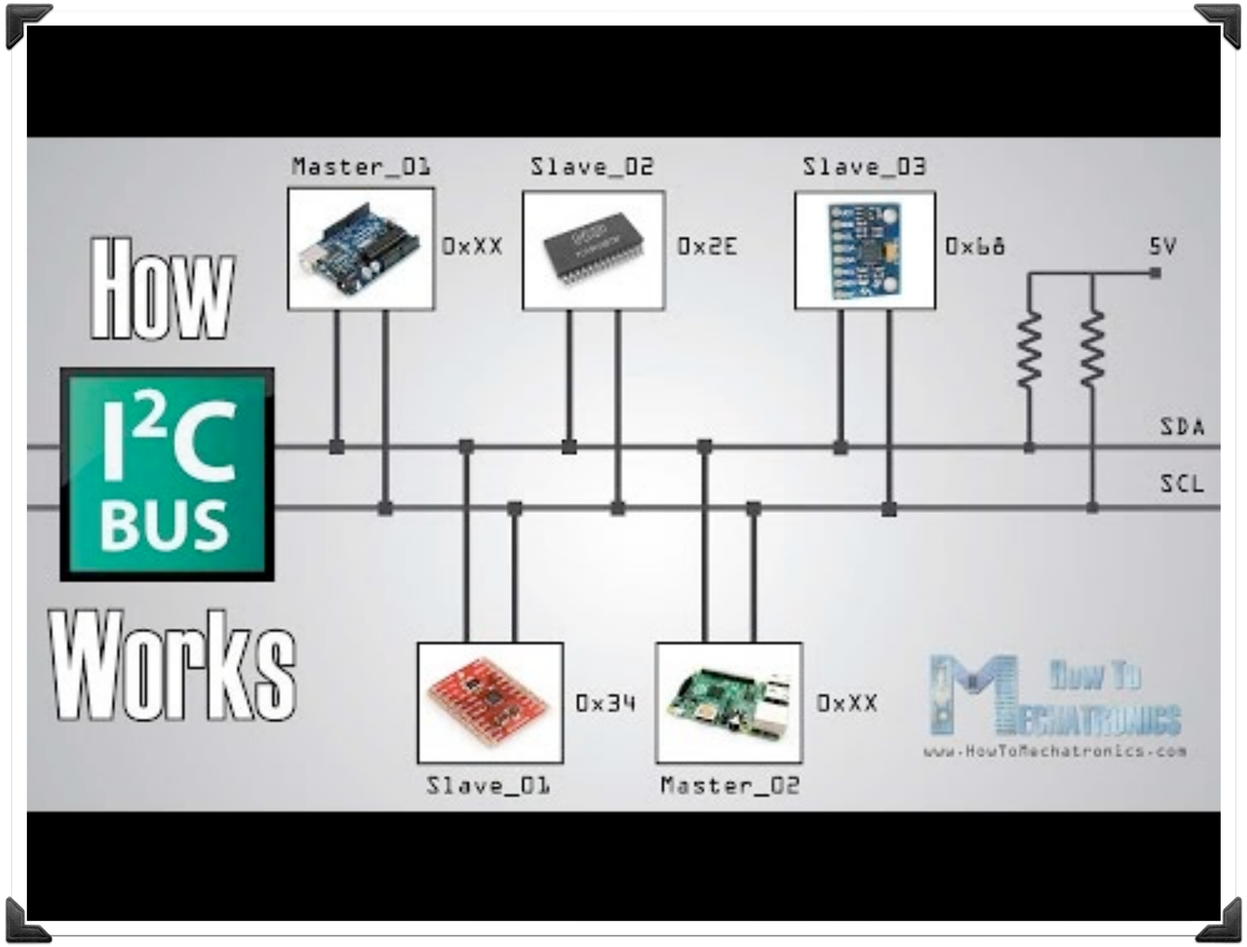


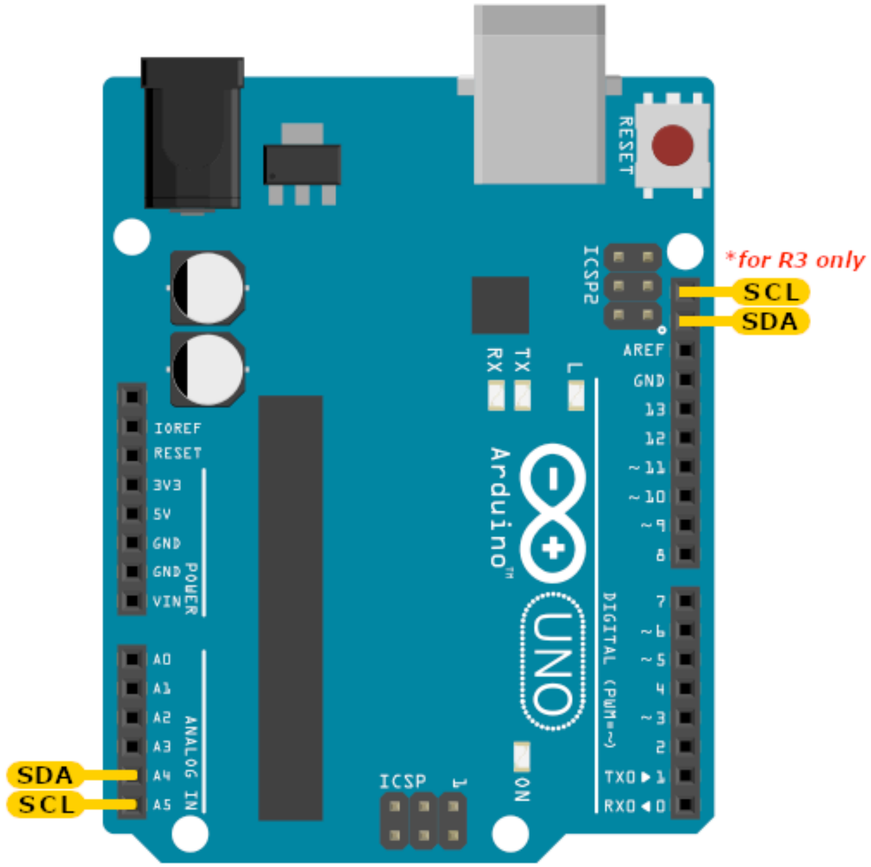
- Supports **multiple masters** and **multiple slaves**.
- Requires two wires (SDA and SCL) plus GND (ground).
- Each slave can be talked to individually via its *address*.
- Masters can send at the same time (outputs use open-drain).
  - To output a 1 go into **tristate** (high impedance)
  - To output a 0 set to **GND**
  - Requires a **pull-up resistor** (the MCU already provides this)
- Supported speeds are: **100** kbit/s (**standard mode**), up to **400** kbit/s (**fast mode**), and up to **1** Mbit/s (**fast mode plus**).
- Many components speak I²C (memories, ADC/DAC, RGB LEDS, …)

Protocol Example: I²C

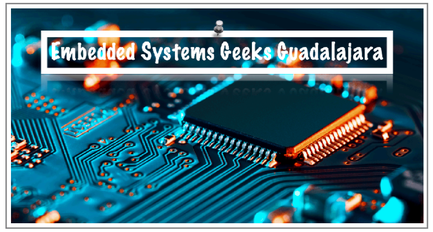


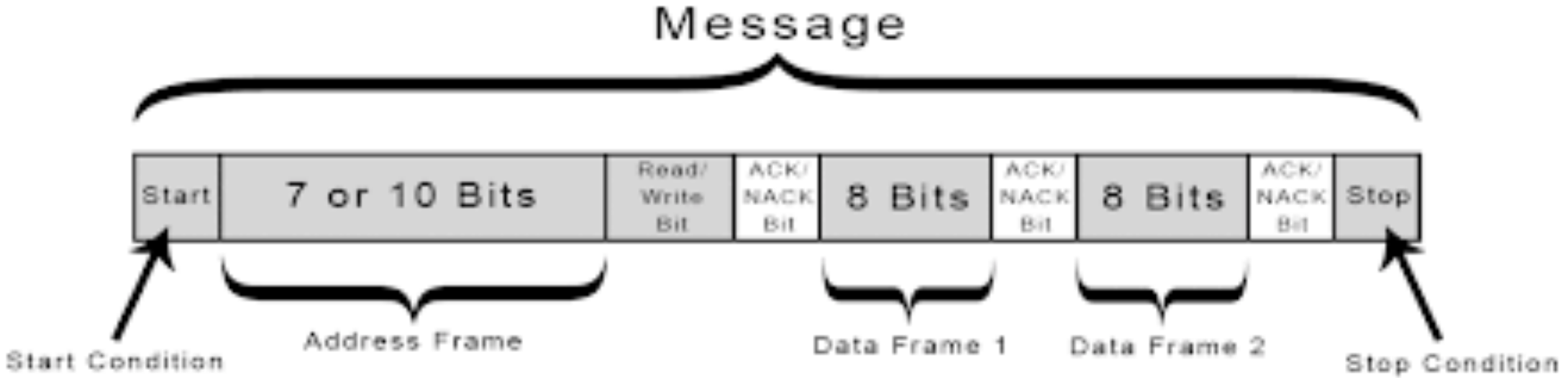
- Many devices can be connected to SDA and SCL.

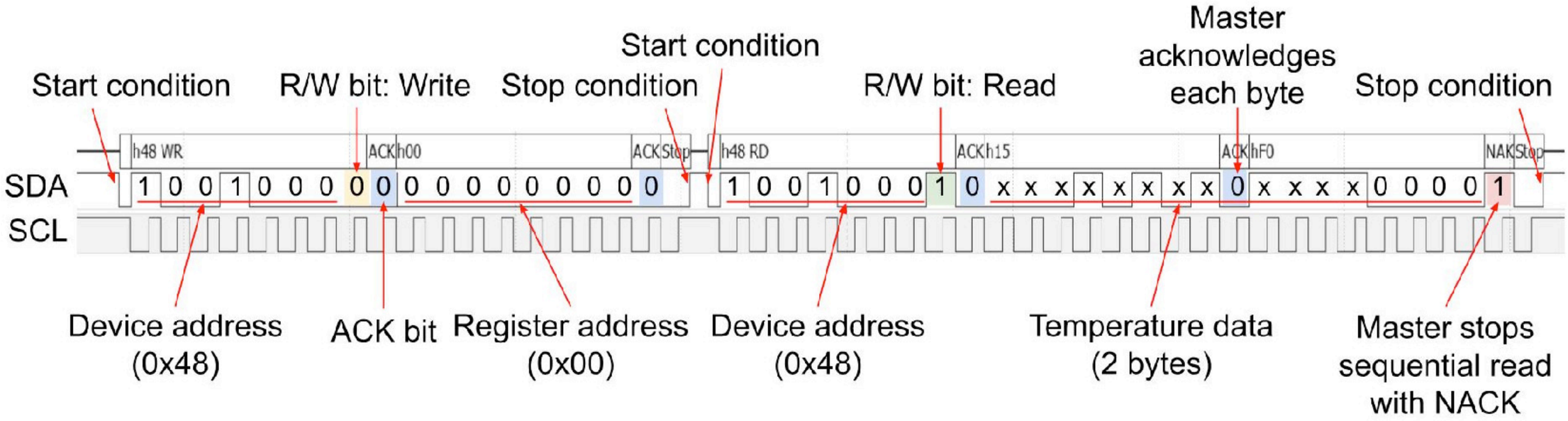




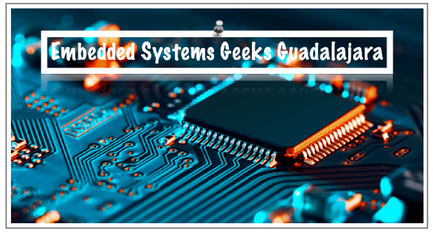
Protocol Example: I²C







I²C on Arduino

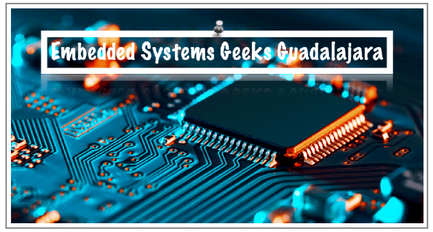


- Supported by the **I²C** / **TWI** library (Wire.h).
- Takes advantage of the native chip support, using interrupts.

  ➡ But there might be some limitations that require **busy waiting** …
- Programmer just has to focus on the transmitted & received data words, and not any details related to the hardware protocol.
- Implementing **I²C** in a fully complaint manners it no easy, e.g.,

  ➡ How to *detect* and *deal with* multiple masters sending **at once**?
- Using the correct PIN output setting is crucial (**open drain** with **pull-up**).
- **I²C** is usually only used for on-board hardware (short connections)

I²C on Arduino (cont'd)

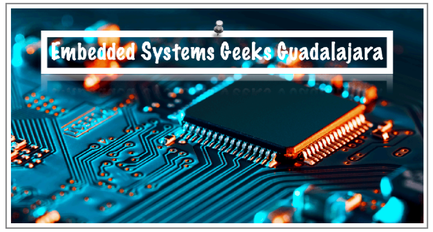


- Arduino **Wire** library code example:

```
13  #include <Wire.h>
14
15  void setup()
16  {
17    Wire.begin(); // join i2c bus (address optional for
18  }
19
20  byte x = 0;
21
22  void loop()
23  {
24    Wire.beginTransmission(4); // transmit to device #4
25    Wire.write("x is ");        // sends five bytes
26    Wire.write(x);              // sends one byte
27    Wire.endTransmission();    // stop transmitting
28
29    x++;
30    delay(500);
31  }
```

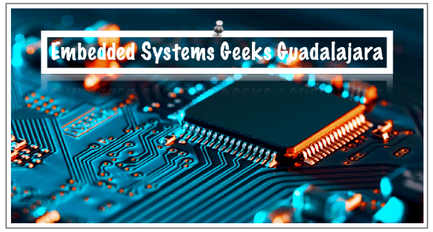
https://docs.arduino.cc/learn/communication/wire

Example Revisited: SSD1306 Driver



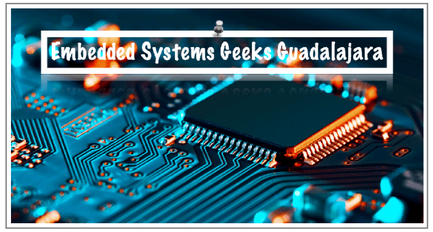
- Have a quick look at the source code in the **Arduino IDE** if time …

Protocols: Summary



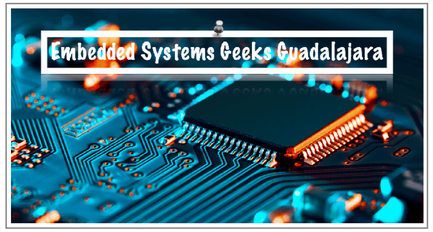
- We have only discussed **two** common protocols here, but there exist many more …
- Recommendation: look into them when you have a **concrete** need !
- Most of them are **not** difficult to understand, but others may be *a little more challenging*. E.g.,
  - ➡ Composite analogue video signals (PAL, NTSC, SECAM)
  - ➡ Other graphics signal standards (VGA, DVI, HDMI)
- You might reach the limits of the Arduino MCUs with that …
  - ➡ **Linus Åkesson** did some cool stuff related to this: http://www.linusakesson.net/scene/phasor/index.php

Misc Topic: Timing



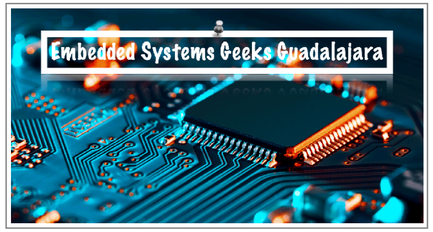
- **Timing** is rather important when programming for embedded systems:
  - Devices often need to **synchronise** with each other.
  - Communication **protocols** have hard timing constraints.
  - Delays are sometimes implemented in pure software.
  - Watchdogs need to be triggered to keep the system alive.
  - And *unresponsive system* often means **death** …
  - Data that is **sent** or **arrives late** may not be of use anymore …
- Games development in the 80s:
  - Count CPU clock cycles to “race the beam”.

Misc Topic: Power/Load



- Electrical devices consumer power when they are in operation.
- Power is determined by **both** voltage and current: **P = U*I**
- **Problem**: output PINs of an MCU can only support current flow up to a certain threshold.
  - Anything above this can **damage** / **destroy** the chip 
- Equally, connecting and output to GND and writing a **high value** (1) can cause instant death … (depends on internal safe-guarding mechanisms)
- Hence, always check that the **consumer** do not draw more current than permitted. Outputs must **never** be connected **directly** to V_{cc} or GND.
- Where do we find this information? In the **data sheet** !

DC/AC Characterstics

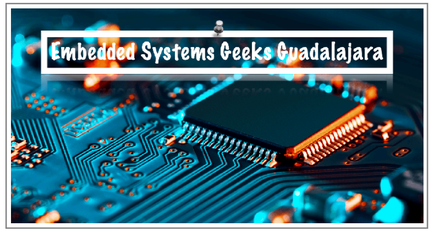


- Maximum Absolute Ratings for **ATmega328P** (**page 258**)

Parameters	Min.	Typ.	Max.	Unit
Operating temperature	–55		+125	°C
Storage temperature	–65		+150	°C
Voltage on any pin except $\overline{\text{RESET}}$ with respect to ground	–0.5		V_{CC} + 0.5	V
Voltage on $\overline{\text{RESET}}$ with respect to ground	–0.5		+13.0	V
Maximum operating voltage		6.0		V
DC current per I/O pin		40.0		mA
DC current V_{CC} and GND pins		200.0		mA
Injection current at V_{CC} = 0V		±5.0(1)		mA
Injection current at V_{CC} = 5V		±1.0		mA

Note: 1. Maximum current per port = ±30mA

DC/AC Characterstics



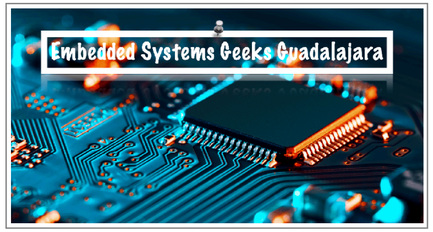
- Maximum Absolute Ratings for **ATmega328P** (**page 258**)

Parameters	Min.	Typ.	Max.	Unit
Operating temperature	–55		+125	°C
Storage temperature	–65		+150	°C
Voltage on any pin except $\overline{\text{RESET}}$ with respect to ground	–0.5		V_{CC} + 0.5	V
Voltage on $\overline{\text{RESET}}$ with respect to ground	–0.5		+13.0	V
Maximum operating voltage		6.0		V
DC current per I/O pin		40.0		mA
DC current V_{CC} and GND pins		200.0		mA
Injection current at V_{CC} = 0V		±5.0(1)		mA
Injection current at V_{CC} = 5V		±1.0		mA

Note: 1. Maximum current per port = ±30mA

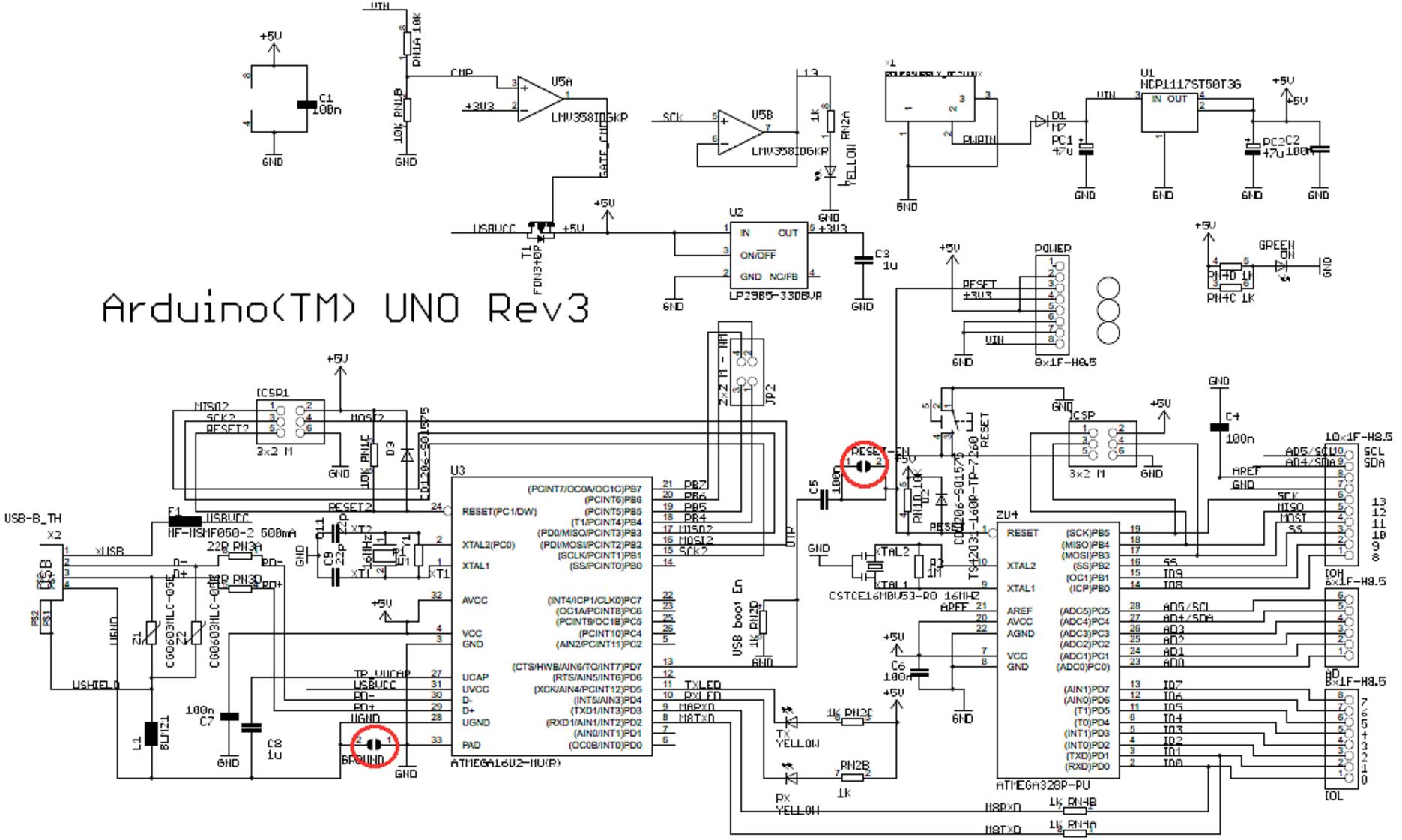
Max. permissible load on I/O PINs

Misc Topic: Schematics

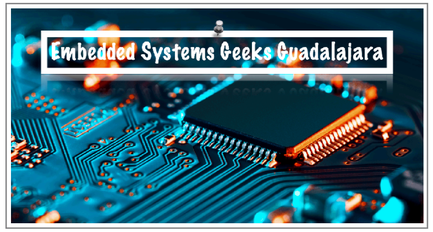


- To **really understand** a development **board** or **device**, we do not get around reading semantics.
- While an MCU includes *most things we need* for deploying *simple* embedded applications, we often profit from additional components.
- Example: USB → UART **bridge** (FTDI), **graphics** signal generators, Ethernet **PHY**, WiFi **transceivers**, MOSFET **drivers** for motors, etc.
- Again, not **rocket science** 🚀 to understand them, but requires basic knowledge in electrical **symbols** and **conventions** of circuit design.

Arduino UNO Schematics



Conclusion



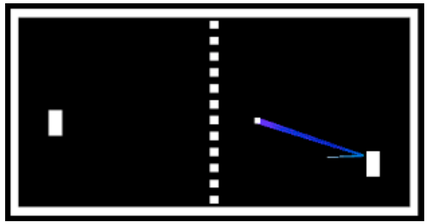
- We had a brief look at the Arduino and AVR **architecture**.
- We understood *a little more* the **constraints** when developing for MCU.
- We examine several hardware communication **protocols**.
- We discussed several miscellaneous topics (**timing**, **load**, **schematics**)
- This should give you a **good start** to embark **on your own quest**:

  ➡ We cannot discuss everything but there is a lot of material on the **WEB**.

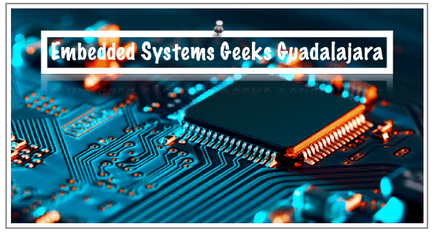
  ➡ Hardware engineering and programming is not rocket science …

  ➡ … but it requires a certain shift in **thinking** and **paradigm**.

  ➡ A suggestion: Let's develop an **arcade game** together !

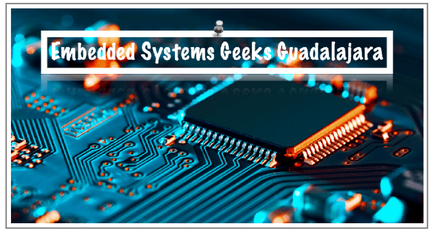


Further Reading & Study



- **First stop**: **Documentation** on https://www.arduino.cc/
  - Language comparison: Arduino *Wiring* Language vs plain C/C++ https://docs.arduino.cc/tutorials/generic/language-comparison
  - Arduino Language Reference which can be found on the above website.
- **Next**: Try out some of the examples that come **preinstalled** with the Arduino IDE: https://docs.arduino.cc/library-examples/
- Have a brief look at the **data sheet** of the ATmega38P
- Arduino Wiki of **NUS** (National University of Singapore)
- Arduino Cookbook (especially, if you are new to programming)
- Any **good book** on C/C++ programming

Thanks for Listening!



Time for questions and comments.