Meeting Assembly — Hello World Arduino Blinking Code

Simulating in Atmel Studio 7 #arduSerie — 27

J3 · Follow
Published in Jungletronics · 14 min read · Feb 27, 2017

On this page, I'll tell you how an Arduino's blink program works in assembly, how to create a project on Atmel Studio 7 and some details about the AVR.

Now it's time to have a look at what we want to do and how it can be done.

1 — Tutorial Objectives :

After completing this AVR microcontroller tutorial readers should be able to:

. Write assembly code to initialize an AVR ATmega 328P and run code on it
. Understand how it works line by line
. Trying this in the simulator of Atmel Studio 7
. Upload the code to Arduino Uno directly from Atmel Studio 7 IDE

2 — Ten (10) Asm Instructions used:

AVR Instruction Set — Manual

Instruction : Cycle : ***Descriptions***

ldi : 1 : *Load Immediate Into*; Loads an 8-bit constant directly to regs.16 to 31.

cbi : 1 : *Clear Bit In I/O Register* — Clears a specified bit in an I/O register.

sbi : 1 : *Set Bit in I/O Register* — Sets a specified bit in an I/O Register.

out : 1 : *Store Register to I/O Location* — Stores data from register Rr in the Register File to I/O Space (Ports, Timers, Configuration Registers, etc.).

dec : 1 : *Decrement* — Subtracts one from the contents of register Rd and places the result in the destination register Rd.

adiw : 2 : *Add Immediate to Word* — Adds an immediate value (0–63) to a register pair and places the result in the register pair.

brne : 2 : *Branch if Not Equal* — Conditional relative branch. Tests the Zero Flag (Z) and branches relatively to PC if Z is cleared.

rcall : 1 : *Relative Call to Subroutine* — Relative call to an address within PC

ret : 1 : *Return from Subroutine* — Returns from the subroutine.

rjmp : 1 : *Relative Jump* — Relative jump to an address.

3 — Solution Index:

Point 00 — MOTIVATION

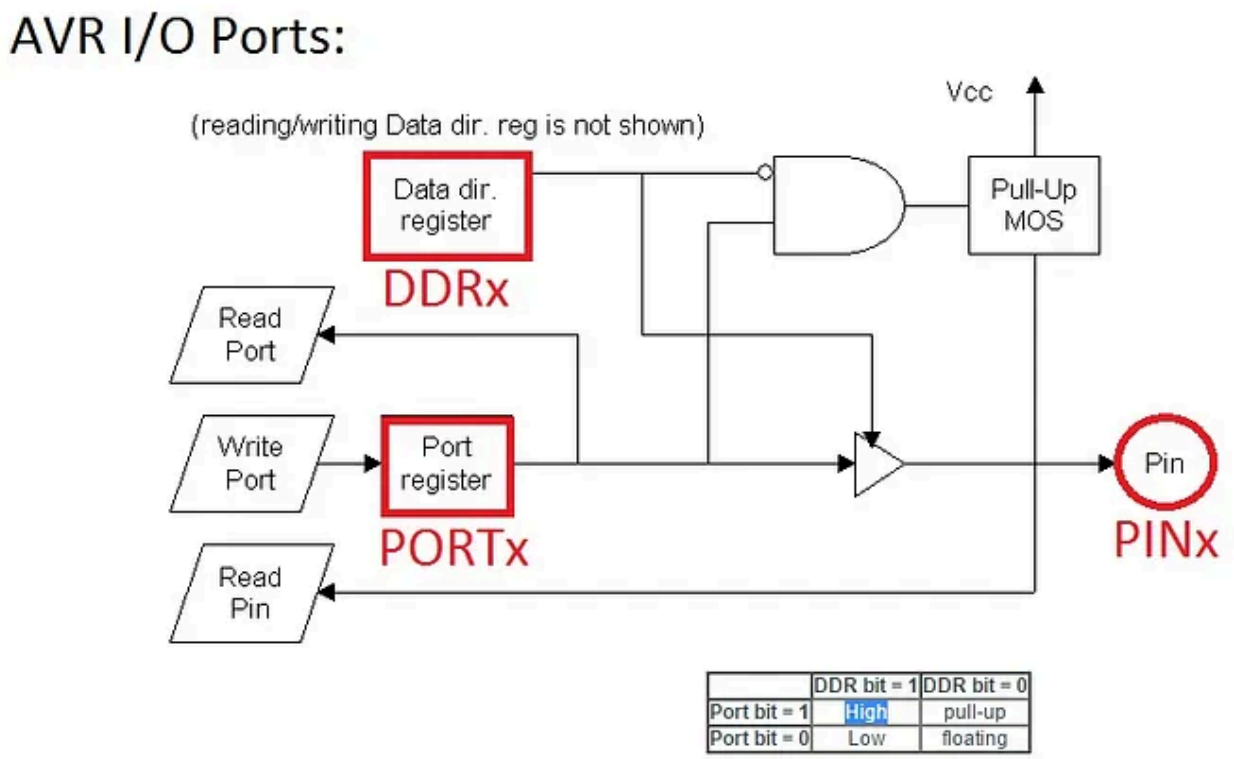
The simulator is very important because it allows us to step through a program and software code line by line and analyze and understand what your code is doing before you actually upload it to your board and running it. Though the hardware is as important as the software it only tells you if your code is running or not.
I would like to see Arduino Internals (under the hood) and Atmel Studio 7 made it as open source. Thanks to Atmel team!
This video is like a recollection for me, so that sometime later I can keep the most important information avoiding forgetting it all.
And sharing comes from the passion I treat the DIY community. So here we go! Let's get started!

Point 01 — Open Datasheet — Prepare a solution

in order that any project goes further one must read the (ATmega328P) datasheet. Let's open the document and make the first *Close Encounters of the Third Kind* :-)

If you click on this flag (top right) and head up to **Register Summary** you'll find all the records and objects of our studies. Here you find the three main registers of Atmel AVR i/o ports:

```
PORTx, DDRx and PINx.
```



source: http://www.avrbeginners.net/architecture/ioports/ioports.html

Lecture#02 Arduino Uno — Circuitry Inside

Atmega328p Generic IO Port Operation

medium.com

Here's a drawing of their basic functionality: as you can see, there's an internal pull-up for every pin (on PIC, 18-K's Families forward). It can be activated by setting the DDR bit of the pin to 0 and the Port bit to 1. A cleared DDR bit means that the pin is an input pin. So the pin is disconnected from the Port register (see the driver in the drawing?) and the pin is floating. In this case, the Port bit controls the pull-up.

We highlight the configuration we will take in our project: we put the DDRx to 1 and we will switch PORTx between the high and low states to blink the LED.

Back to datasheet from the **Instruction Set Summary,** we will use in our code only 10 out of 131 (yep! just one hundred thirty-one! ugh…) Powerful Instructions that are **cbi**, **ldi**, **out**, **dec**, **adwi**, **brne**, **rcall**, **ret**, **rjmp** and **sbi** (details above).
Now, open **Atmel Studio 7:**

```
File > New > Project/Solution
```

Dismiss the **start page**, choose **Atmel Studio Solution > Blank Solution** from the **installed** tab. Name it as you wish…I'll give you it the name **Solution_AVR1**, configure the directory of your project so you know where you place it.

Now on **Solution Explorer,** right-click and choose **Add > New Project**, choose **Assembler > AVR Assembler Project,** and on **Device Selection**, type 328p and choose **ATmega328P** and name it as you wish. As soon as you open the project you should put the **Block Comment Header** (BCH) so that anyone knows what the code is for (as good programming practices). see links for an awesome BCH website with examples in this :D

```
;
; _27_arduserie_Simulae_328P.asm
; ***here goes the BCH****
;
```

We will follow an academic standard of opening two projects. One for simulation and another to be loaded into the hardware.

Point 02 — Open a Simulation Project

_27_arduserie_Simulae_328P for simulation purposes. Cool!

Point 03 — Open a Target Project

_27_arduserie_Target_328P for aim at the target which is the hardware code, Arduino Uno, of course…

Point 04 — Init your code

Let's have look at probably the easiest program possible:

```
.ORG 0x000  ; the next instruction has to be written to add 0x0000

            ; infinite loop
START:      ; this is a label "START"
rjmp START  ; Relative JuMP to START
```

When the instruction is executed, the CPU (**ALU — Arithmetic Logic Unit**) will jump to **START**. The jump will be repeated over and over, resulting in an infinite loop. Alright, that's not a big deal…

Point 05 — Do the Math

Let's continue the code development.
Assuming that our Arduino AVR is running at 16 MHz (16 Million clock cycles per second), how long does all this take?

```
T = 1/ F then T = 1 / 16E6
T = 0,0000000625 seconds, not precisely 0,06 us
```

That's pretty fast! We can not see at this frequency the blinking of the LED.
So coming to the answer it has been observed experimentally that the human eye can't make out a difference in the picture frame if it appears for less than 16ms to 13ms (0,016-0,013s). Hence purely based on this we can say sampling frequency is 60Hz to 80Hz (about 0,06 & 0,08 MHz).
So we will need to know how many times we will loop to in terms of a cycle of say half a second off and half a second on so that the LED flashes every half a second at a frequency of 16 MHz we need x cycles. Here's how:

```
if 1 cycle takes    -----------0,0000000625 seconds
x  cycles will take -----------0,5          seconds
this makes 0,5 / 0,0000000625 s = 8 000 0000 cycles
```

How to achieve all these cycles if I can count only up to 255 (remember, we have an 8-bit chip)... magic? Magic wand?
Relax, with the technique that I will present everything will become easy.

For this, we will have to implement **two loops**: inner and *outloop*. See how: Calculations - The inner loop is treated like one BIG instruction needing 262145 clock cycles. See: Inner loop first - as you know registers can be used in pairs, allowing to work with values from 0 to 65 535. That's a **word**.
The following piece of code clears registers 24 and 25 and increments them in a loop until they overflow to zero again.
When that condition occurs, the loop doesn't go around again. Let's go ahead!

```
clr r24             ; clr needs 1 cycle
clr r25             ; clr needs 1 cycle

DELAY_05:           ; we need .5s delay
   adiw r24, 1      ; adiw needs 2 cycles and
```

```
    brne DELAY_05     ; brne needs 2 cycles if the branch is done
                      ; and 1 otherwise
```

Every time the registers don't overflow the loop takes adiw(2) + brne(2) = 4 cycles.
This is done 0xFFFF (65 535) times before the overflow occurs. The next time the loop only needs 3 cycles, because no branch is done.
This adds up to 4 * 65 535(looping) + 3(overflow) + 2(clr) = 262 145 cycles. This is still not enough: 8 000 000/262 145 ~ 30.51.

The "outer" loop will be down-counting from 31 to zero using R16.

```
ldi r16, 31

OUTER_LOOP:          ; outer loop label
   ldi r24, 0          ; clear register 24
   ldi r25, 0          ; clear register 25

DELAY_05:            ; the loop label
   adiw r24, 1         ; "add immediate to word": r24:r25 are
                     ; incremented
   brne DELAY_05
   dec r16             ; decrement r16
   brne OUTER_LOOP     ; load r16 with 8
```

The overall loop needs: 262 145 (inner loop) + 1 (dec) + 2 (brne) = 262 148 * 31 = 8 126 588 cycles.
This is more like what we want, but 126 588 cycles too long.

This is where the **fine-tuning** comes in — we need to change the initial value of r24:r25.
The outer loop is executed 31 times and includes the "big-inner-loop-instruction".

We have to **subtract some cycle**s from the inner loop: 126 588 / 31 = 4 083 cycles per inner loop.
This is what the inner loop has to be shorter. Every iteration of the inner loop takes 4 cycles (the last one takes 3 but that's not so important), so let's divide those 4 083 by 4.
That's 1 020.8 or 1 021 fewer iterations.

This is our new initialization value for r24:r25!
Now, if you want, do all those calculations again: The result is 8 000 000 clock cycles!

Now just put this into a separate routine and call it from the main LED flashing loop. Here is a complete program:

```
.ORG 0x0000             ; the next instruction has to be written to
                        ; address 0x0000
rjmp START              ; the reset vector: jump to "main"
START:
ldi r16, low(RAMEND)    ; set up the stack
out SPL, r16
ldi r16, high(RAMEND)
out SPH, r16
ldi r16, 0xFF           ; load register 16 with 0xFF (all bits 1)
out DDRB, r16           ; write the value in r16 (0xFF) to Data
                        ; Direction Register B
LOOP:
  sbi PortB, 5          ; switch off the LED
  rcall delay_05        ; wait for half a second
  cbi PortB, 5          ; switch it on
```

```
    rcall delay_05        ; wait for half a secon
    rjmp LOOP             ; jump to loop

DELAY_05:                 ; the subroutine:
    ldi r16, 31           ; load r16 with 31
OUTER_LOOP:               ; outer loop label
    ldi r24, low(1021)    ; load registers r24:r25 with 1021, our new
                          ; init value
    ldi r25, high(1021)   ; the loop label
DELAY_LOOP:               ; "add immediate to word": r24:r25 are
                          ; incremented
    adiw r24, 1           ; if no overflow ("branch if not equal"), go
                          ; back to "delay_loop"
    brne DELAY_LOOP
    dec r16               ; decrement r16
    brne OUTER_LOOP       ; and loop if outer loop not finished
    ret                   ; return from subroutine
```

Point 06 — Run a Simulation on Atmega 238P

So let's go ahead and start a debugging session or simulate.
You just need to click **Build > Build Solution** and proceed on by clicking on the **Start Debugging and Break** (Alt + F5) button of the **Debug** menu.
Let's just do that and see what happens...

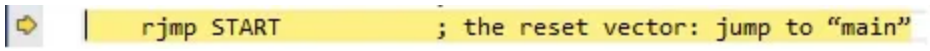
Oops: error? **Please select connected tools and interface and try it again...** that's for the purpose, click on **Continue** and in **Tool > Selected debugger/programmer** choose **Simulator.**

Take advantage and go to **Toolchain** and under **General** disable **Generate HEX file.** This is a security feature. As we are simulating here we do not want

to put this file on the chip, right?

Ctrl + S, and dismiss the Properties window and **Alt + F5** again.

So the debugger has started and break means to pause in the first line (or whatever you wish to...). Select in **I/O** window **PORTB**.



Notice a **yellow arrow** stops right at the address zero. It points to the instruction that will be executed next (pressing **Reset** brings here to), but has not been yet executed.
If you want to step through the code line by line there are basically two key commands you must be aware of: **Step Over** (F10) and **Step Into** (F11).
Let see how this works.
If I hit **F11** it jumps to the next instruction. The counter program starts counting. A machine cycle is passed.
In the Arduino frequency rate it takes a mere 0.06 microseconds.
In the next step, we will notice that the **R16** register will change its value. It will be **0xFF** the value of the first nibble out of Stack pointer.

To understand, it better go to this link.

These operations allow us to use 16 bits (one word) to do the counting required.

Note:

> *The AVR 8-bits microcontroller stack pointer can either consist of a single I/O register SPL (Stack Pointer Low) or two (2) I/O registers SPL and SPH (Stack Pointer High). The size of the stack pointer depends on the amount of data memory a microntroller contains. If the entire data memory can be addressed using 8-bits then the stack pointer is 8-bits wide i.e. SPL only, otherwise the stack pointer is consist of SPL and SPH.*

Alright. Now we need to configure our **DDRB**. With 0xFF all bits are 1, as output.
Now it's only necessary to set the **PORTB** up and down alternately using **sb**i to set bit and **cbi** to clears it.

Now we are using **rcall** to jump to loop **delay_05** that waits for half a second. For this hit now **F10 — Step Over**.

After a brief second, the clock will stop in the cycle of number **8 million** according to our calculations above.

Now if we calculate the **StopWatch** will be scoring exactly half a second. Look:

```
500000000 us = 0,5 seconds
```

Awesome!!!

When you're done stop debugging by click on **Stop debugging** button (**Ctrl+Shift+F5**).

We finally got to turn assembly language into **Hello world**!!!.

Point 07 — Run On Real Board — Arduino UNO

Copy and paste this code to the **target** project.
With our previous video, we already have the necessary tools to load this code for the Arduino directly from Atmel Studio 7 IDE. Just go to the **Tools** menu, hit:

```
Send To ArduinoUNO
```

and there you go! Here is the video for this setup.

Point 08 — Take your Simulation project as Template

Now let's turn the entire project into a template. For this use the code of the simulation.


```
/*============================================================
 | Project: ASSIGNMENT NUMBER AND TITLE
 |
 | Author: STUDENT'S NAME HERE
 | Language: NAME OF LANGUAGE IN WHICH THE PROGRAM IS WRITTEN AND
 |           THE NAME OF THE COMPILER USED TO COMPILE IT WHEN IT
```

```
|           WAS TESTED
| Solution: YOUR SOLUTION ATMEL STUDIO 7 NAME
|
| Projects:  _##_####_Simulation_ATMEGA328P &
|            _##_####_Target_ATMEGA328P
|
| To Compile: EXPLAIN HOW TO COMPILE THIS PROGRAM
|
| Software: NAME AND TITLE OF THE CLASS FOR WHICH THIS PROGRAM WAS
|           WRITTEN
| OS Version: YOUR COMPUTER PROGRAM VERSION NUMBER
|
| Platform: YOUR COMPUTER PROGRAM
|
| Instructor: NAME OF YOUR COURSE'S INSTRUCTOR
|
| Due Date: DATE AND TIME THAT THIS PROGRAM IS/WAS DUE TO BE
|           SUBMITTED
|
+ ------------------------------------------------------------------
|
| Description: DESCRIBE THE PROBLEM THAT THIS PROGRAM WAS WRITTEN
|              TO SOLVE.
|
| Input: DESCRIBE THE INPUT THAT THE PROGRAM REQUIRES.
|
| Output: DESCRIBE THE OUTPUT THAT THE PROGRAM PRODUCES.
|
| Algorithm: OUTLINE THE APPROACH USED BY THE PROGRAM TO SOLVE THE
| PROBLEM.
|
| Required Features Not Included: DESCRIBE HERE ANY REQUIREMENTS OF
| THE ASSIGNMENT THAT THE PROGRAM DOES NOT ATTEMPT TO SOLVE.
|
| Known Bugs: IF THE PROGRAM DOES NOT FUNCTION CORRECTLY IN SOME
| SITUATIONS, DESCRIBE THE SITUATIONS AND PROBLEMS HERE.
|
*==================================================================*/

.INCLUDE "m328pdef.inc"  ; this are automatically included
.ORG 0x0000              ; initial instruction
    rjmp START           ; the reset vector: jump to “main”

      ;***place your data here***

START:
      ;***your code goes her***
LOOP:
      ;***loop routine***
   rjmp LOOP
```

```
        ;***your procedure here***
.EXIT                   ; tells assembler it's over here!
```

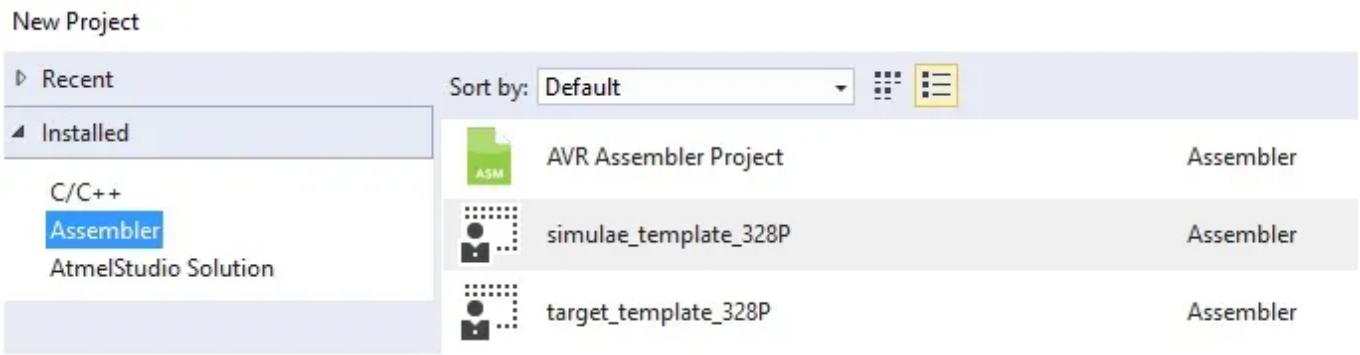
Crtl+s. Now hit **File > Export Template...**
Choose **Template Type**, hit **Project Template.**
Hit **Next.**
Choose **Template name** (simulae_template_328P), **Template description, t**ype a little text to remind you later (simulation/target project template for Arduino Uno) and **Output location** (new directory). Leave everything in the default setting.
You will end up with a dot zip file, in my case, C:\Users\giljr\Documents\Visual Studio 2015\My Exported Templates.

Repeat the same procedure to the target project. Name it as target_template_328P.

Cool!!! **reboot** Atmel Studio 7 IDE and navigate **File> New > Project** n boom!!! There you go!!!

Point 09— Take your Target project as Template

Repeat the same procedure for the target project.



Point 10 — Prepare The next Project

File > Import > Project Template and select your .zip file of the project of interest.

Save as '**Solution_AVR2**' and '**_28_arduserie_Simulation_ATMEGA328P**' and '**_28_arduserie_Target_ATMEGA328P**'

For the next project lets work with '**Timers and Counters**'.

Let's go deeper and deeper inside the Arduino board!!!

But this is issues for the next video...thanks for your time!!! bye!!!

BTW — Some Important Concepts:

What is assembly?
Assembler is a low-level language. An assembly (or assembler) language, often abbreviated **asm**, is a low-level programming language for a computer, or another programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler (Wikipedia).

What is a stack?
A stack is a consecutive block of data memory allocated by the programmer. This block of memory can be used both by the microcontroller internal control as well as the programmer to store data temporarily. The stack operates with a Last In First Out (**LIFO**) mechanism, i.e the last thing stored on the stack is the first thing to be retrieved from the stack.

What is the Stack Pointer?
The stack pointer is basically a register or registers that hold either "the

memory address of the last location on that stack where data was stored" or "the memory address of the next available location on the stack to store data." The definition of the stack pointer depends on the design of the microcontroller. In AVR microcontrollers, such as the ATMega8515, ATMega16, ATTiny13, etc., the stack pointer holds the address on the next available location on the stack available to store data.
The AVR 8-bits microcontroller stack pointer can either consist of a single I/O register SPL (Stack Pointer Low) or two (2) I/O registers **SPL** and **SPH** (Stack Pointer High). The size of the stack pointer depends on the amount of data memory a microcontroller contains. If the entire data memory can be addressed using 8-bits then the stack pointer is 8-bits wide i.e. SPL only, otherwise, the stack pointer is consists of SPL and SPH.

What the hell means RAMEND?
RAM END is a label that represents the address of the last memory location in SRAM. To use this label you MUST ensure that you include the definition header file for the specific microcontroller. The functions low() and high() are used by the assembler to return the low byte and high byte respectively of a 16-bit word. Remember we are dealing with an 8-bit microcontroller that can only handle only 8-bits at a time. RAMEND is a 16-bit word and so we use the functions to split it.

Download All Project Archives

References:

Lecture 1: Using, Setting Up and Simulating w/ Atmel Studio 7 (Thanks to Mr. Santos for awesome lesson!!! —I wish I could watch your classes live!!!)

AVR Microcontroller Stack and Stack Pointer

An asm Introduction And The Embedded “Hello World”!!!!

Block Comment Templates with Examples

I/O Ports

AVR Memory Organization

Mikrochip’s Section 8. Interrupts

[Edited @ nov 2019 using Grammarly powered corrections]

Programming Arduino Assembly Atmel Studio 7 Hello World

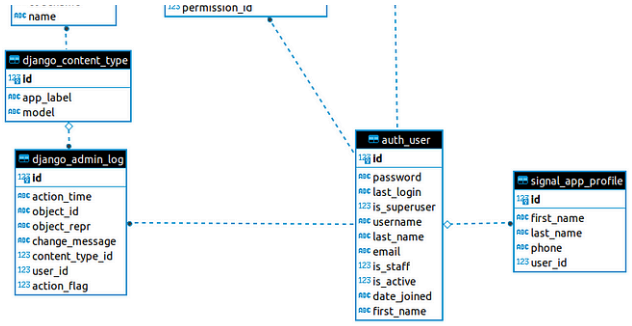
Written by J3

Follow

1.1K Followers · Editor for Jungletronics

Hi, Guys o/ I am J3! I am just a hobby-dev, playing around with Python, Django, Ruby, Rails, Lego, Arduino, Raspy, PIC, AI... Welcome! Join us!

More from J3 and Jungletronics



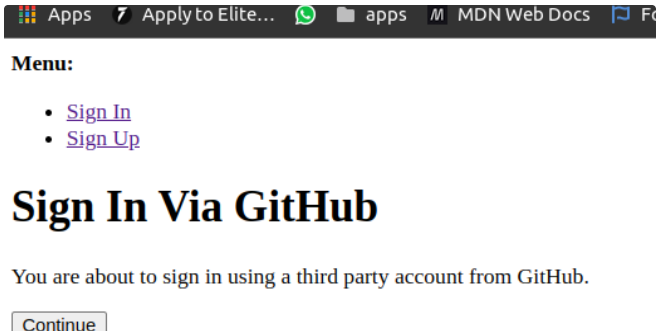
J3 in Jungletronics

How Django Signals Work

Stand-alone Project to show how-to #PureDjango—Episode #00

7 min read · Apr 6, 2023

48 2



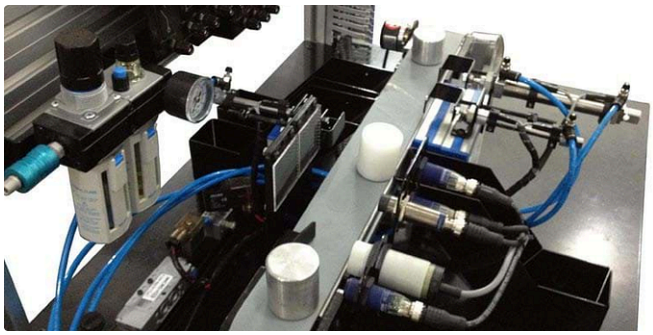
J3 in Jungletronics

All-Auth with Django-Allauth

Django-allauth Tutorial #PureDjango—Episode #01

8 min read · Apr 14, 2023

18



J3 in Jungletronics

28BYJ-48 Stepper Motor-Peak RPM

RPM Configurations & Industrial Conveyor Belt — ArduSerie#68

4 min read · Jan 27, 2019

24 4

J3 in Jungletronics

Mosquitto — ACLs

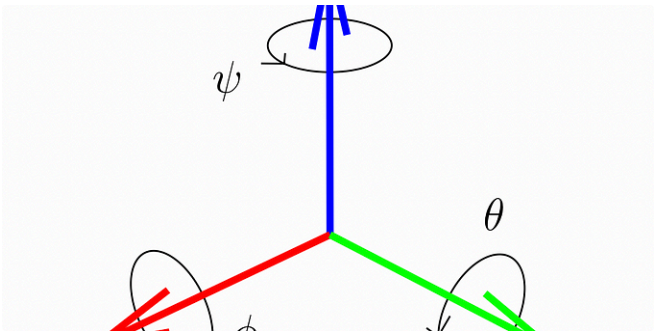
Wildcards & ACL — Access Control Lists — MQTT — Episode #03

7 min read · May 2, 2020

10 2

See all from J3 See all from Jungletronics

Recommended from Medium



nuclei_av

Malware Analysis Handbook—1

I have been learning Malware since March this year. Hence will be starting a Malware...

9 min read · 5 days ago

76

Callum Bruce in Better Programming

Let's Build an Arduino-based Kalman Filter for Attitude...

A practical guide to attitude determination with Kalman filtering

✦ · 15 min read · Jul 19, 2023

390 4

Lists

General Coding Knowledge

20 stories · 1184 saves

Coding & Development

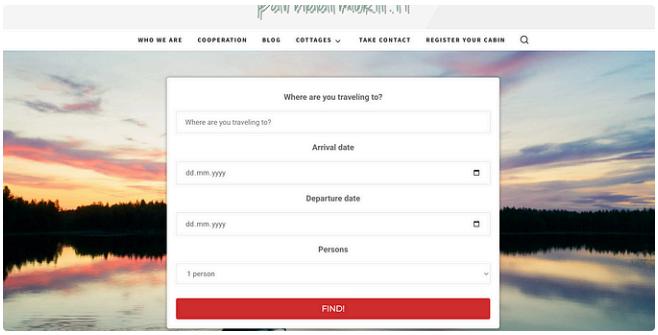
11 stories · 591 saves

Stories to Help You Grow as a Software Developer

19 stories · 1027 saves

ChatGPT

21 stories · 612 saves



Somnath Singh in Level Up Coding

The Era of High-Paying Tech Jobs is Over

The Death of Tech Jobs.

✦ · 14 min read · Mar 31, 2024

10.1K 260

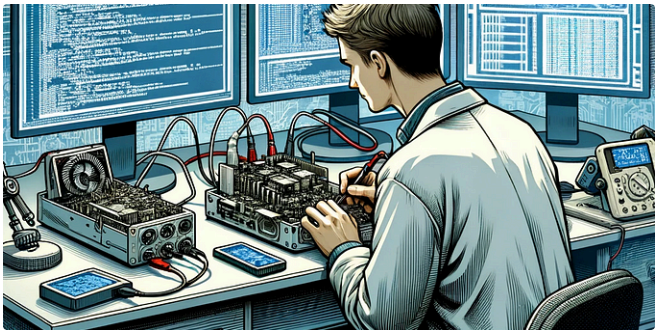
Artturi Jalli

I Built an App in 6 Hours that Makes $1,500/Mo

Copy my strategy!

✦ · 3 min read · Jan 23, 2024

17.5K 191



Hazel Paradise

How I Create Passive Income With No Money

many ways to start a passive income today

5 min read · Mar 27, 2024

9.8K 232

Path Cybersec [Slava Moskvin]

Extracting Firmware: Every Method Explained

8 min read · Jan 2, 2024

105