Practical Firmware Reversing and Exploit Development for AVR-based Embedded Devices

Alexander @dark_k3y Bolshev

Boris @dukeBarman Ryutin

; cat /dev/user(s)

- Alexander Bolshev (@dark_key), Security Researcher, Ph.D., Assistant Professor @ SPbETU
- Boris Ryutin (@dukeBarman), radare2 evangelist, Security Engineer @ ZORSecurity

Agenda

Hour 1

- Part 1: Quick **RJMP** to AVR + Introduction example

Hours 2-3:

- Part 2: Pre-exploitation
- Part 3: Exploitation and ROP-chains building
- Part 4: Post-exploitation and tricks

Hour 4:

- Mitigations
- **CFP! (Powered by Roman Bazhin)**

If you have a question, please interrupt and ask immediately

Disclaimer:

1) Workshop is **VERY** fast-paced.
2) Workshop is highly-practical
3) You may encounter information overflow

Image Credit: Marac Kolodzinski

Part 1: What is AVR?

AVR

- **A**lf (Egil Bogen) and **V**egard (Wollan)’s **R**ISC processor
- Modified Harvard architecture 8-bit RISC single-chip microcontroller
- Developed by Atmel in 1996 (now Dialog/Atmel)

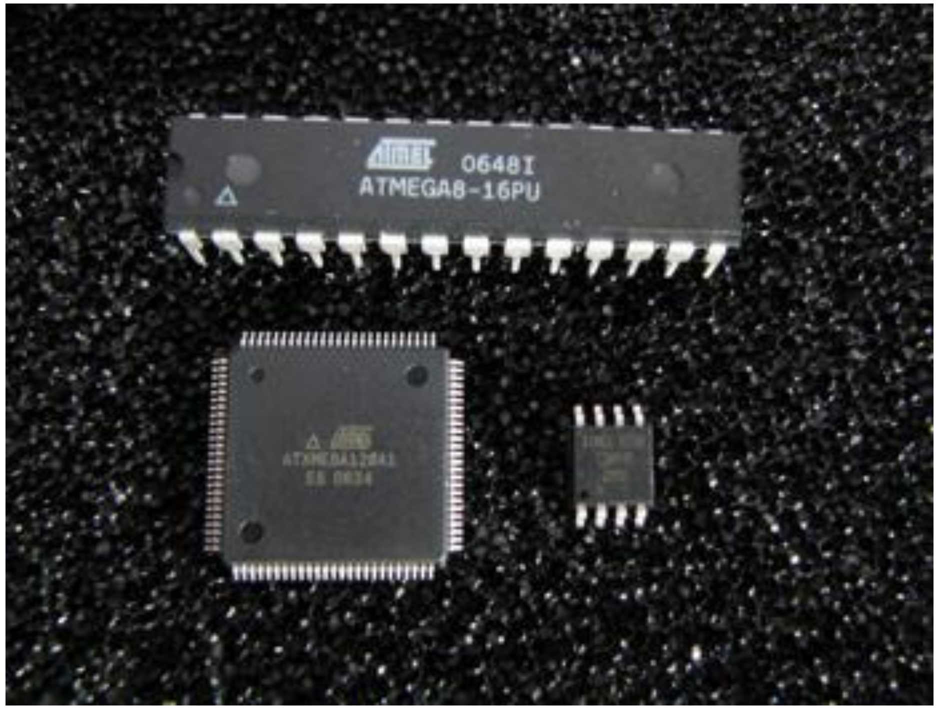
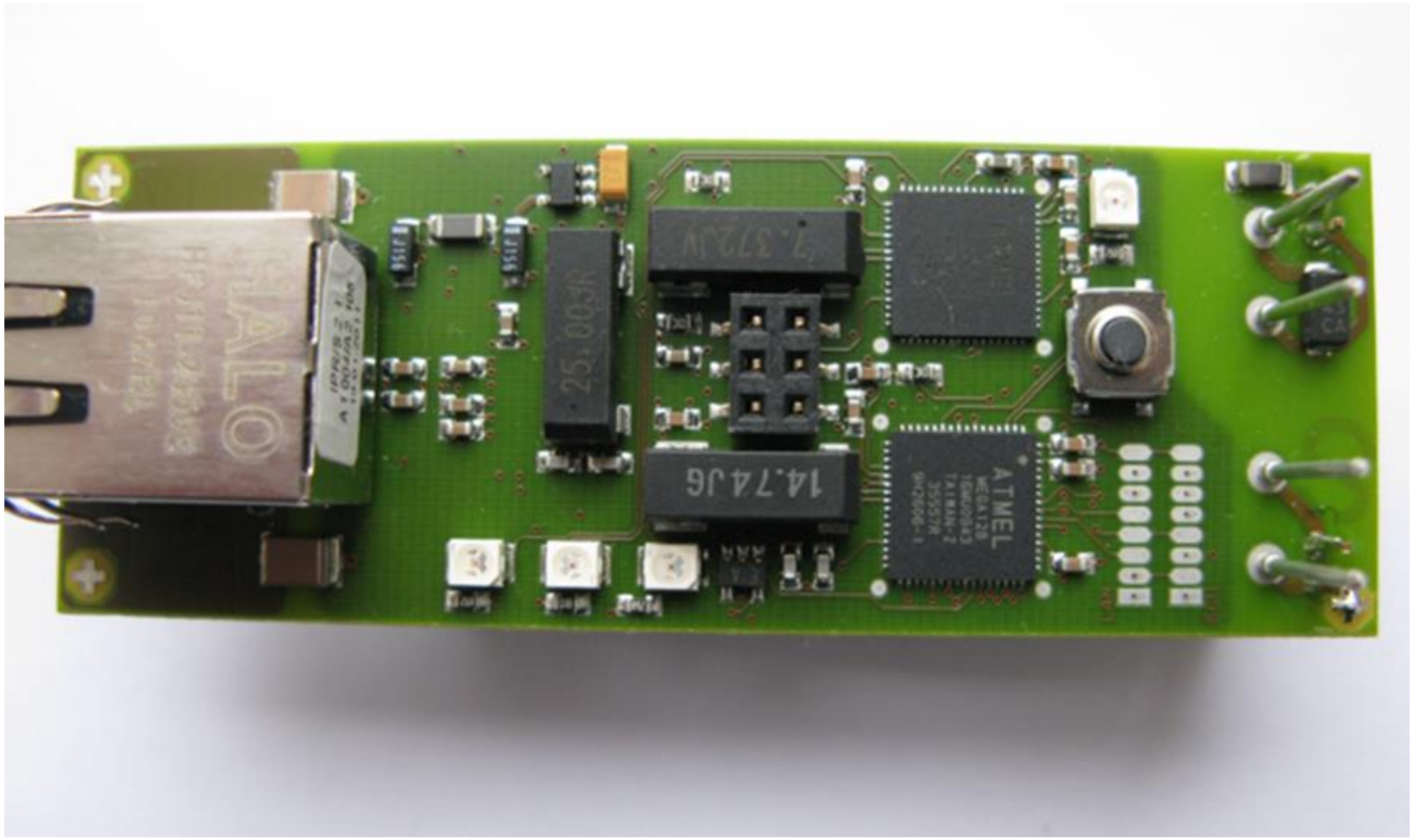


Image: https://de.wikipedia.org/wiki/Atmel AVR

AVR is almost everywhere

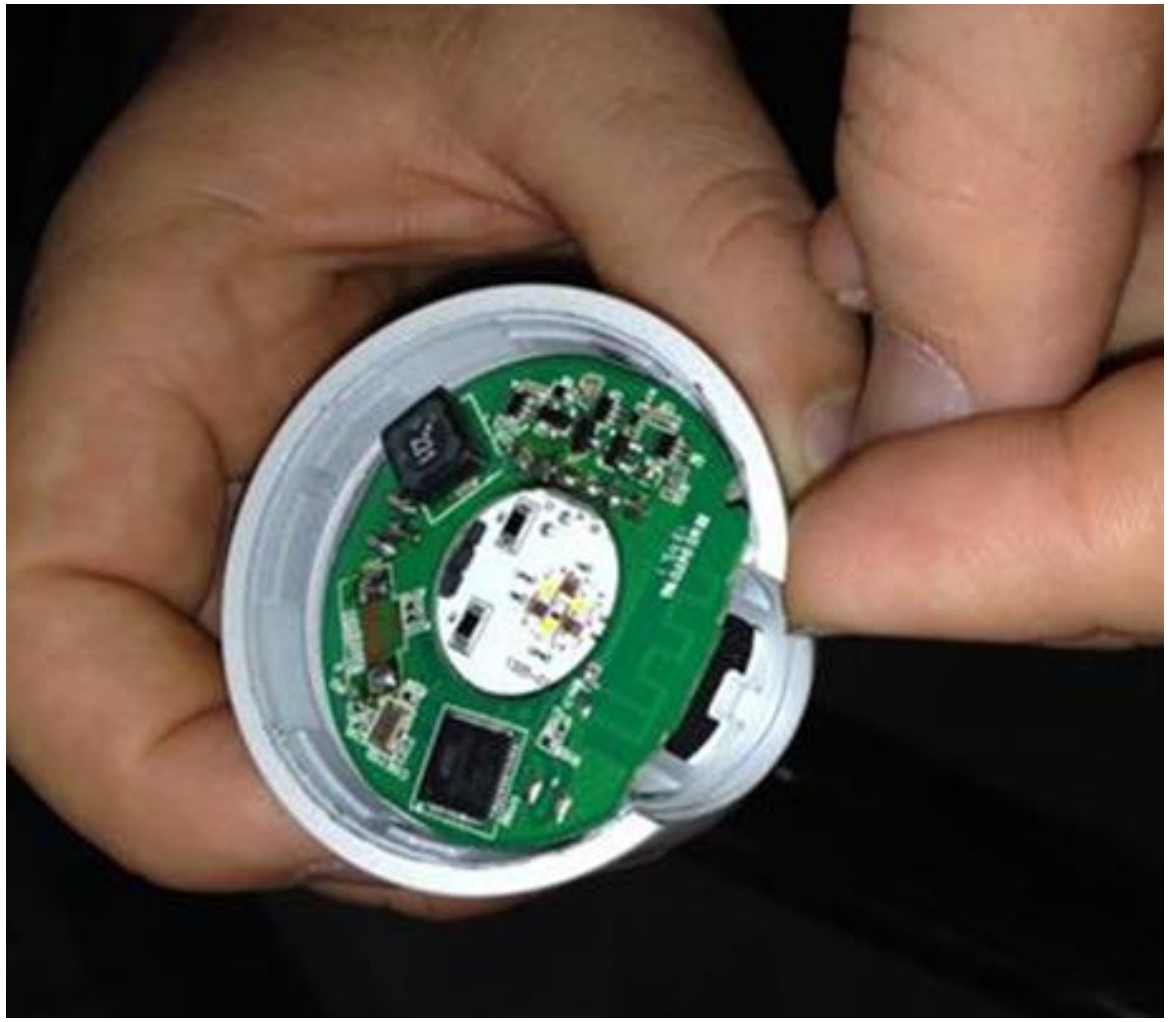
- Industrial PLCs and gateways
- Home electronics: kettles, irons, weather stations, etc
- IoT
- HID devices (ex.: Xbox hand controllers)
- Automotive applications: security, safety, powertrain and entertainment systems.
- Radio applications (and also Xbee and Zwave)
- Arduino platform
- Your new shiny IoE fridge ;)

AVR inside industrial gateway



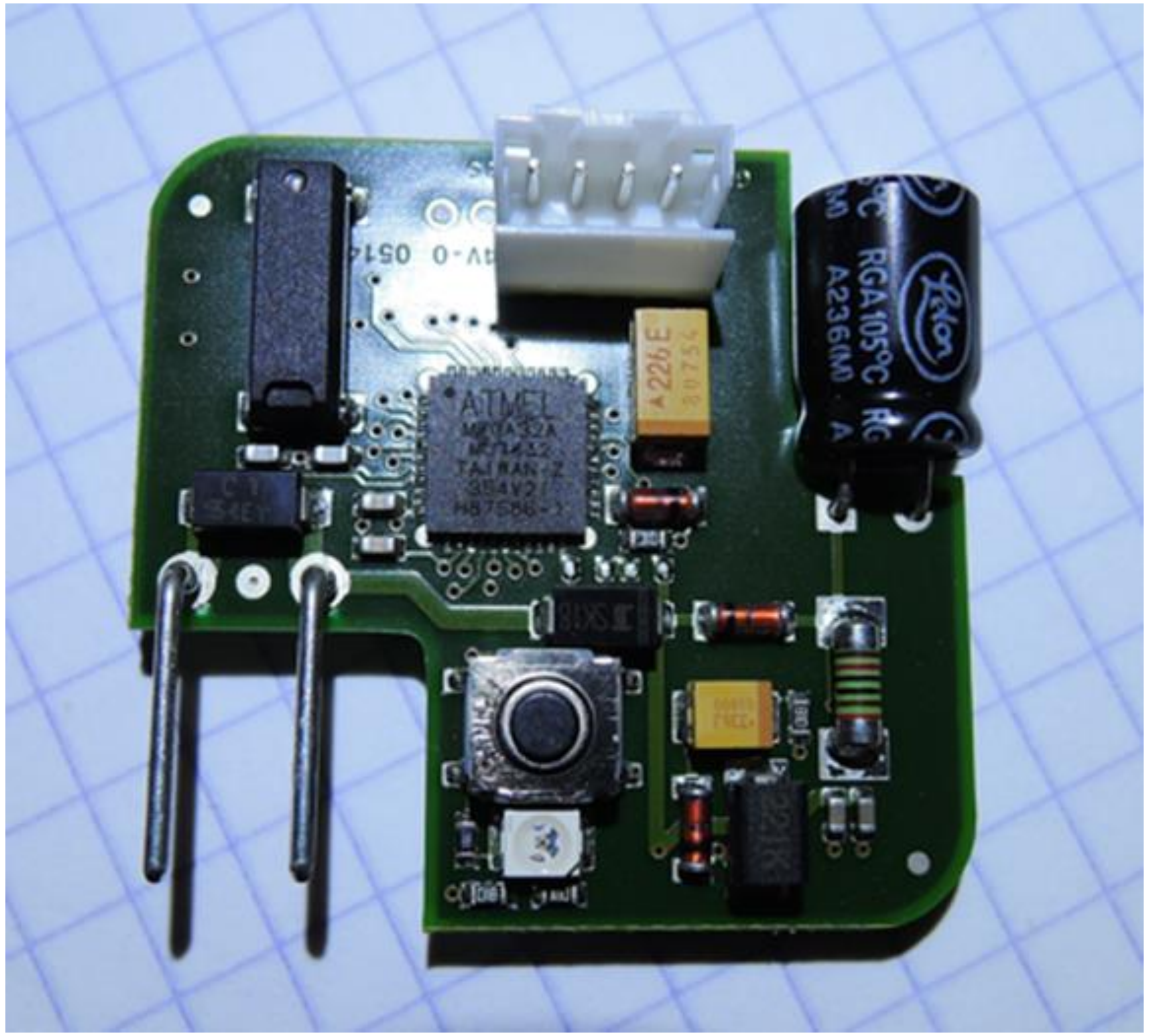
Synapse IoT module with Atmega128RFA1 inside

Philips Hue Bulb



http://www.eetimes.com/document.asp?doc_id=1323739&image_number=1

AVR inside home automation dimmer



Harvard Architecture

Harvard Architecture

- Physically separated storage and signal pathways for instructions and data
- Originated from the Harvard Mark I relay-based computer

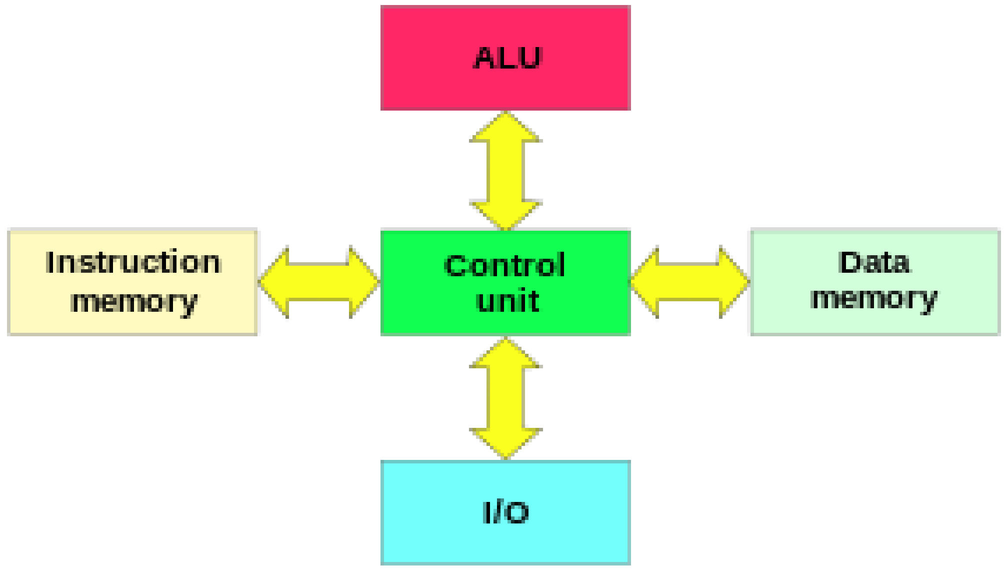


Image: https://en.wikipedia.org/wiki/Harvard architecture

Modified Harvard architecture…

…allows the contents of the instruction memory to be accessed as if it were data[1]

[1]but not the data as code!

DEMO

Introduction example:
We’re still able to exploit!

AVR “features”

AVR-8

- MCU (MicroController Unit) -- single computer chip designed for embedded applications
- Low-power
- Integrated RAM and ROM (SRAM + EEPROM + Flash)
- Some models could work with external SRAM
- 8-bit, word size is 16 bit (2 bytes)
- Higher integration
- Single core/Interrupts
- Low-freq (<20MHz in most cases)

Higher Integration

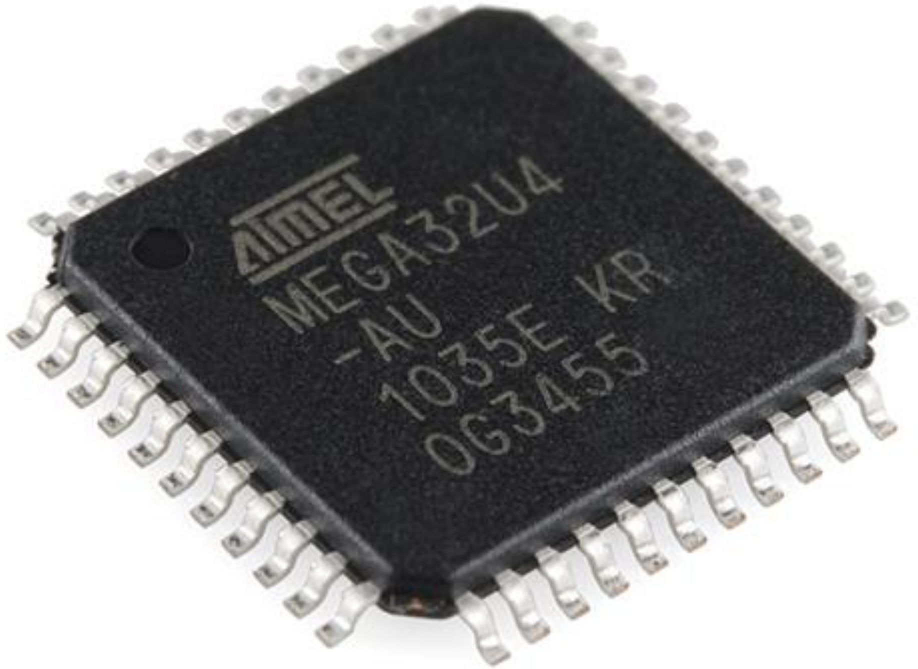
- Built-in SRAM, EEPROM an Flash
- GPIO (discrete I/O pins)
- UART(s)
- I^2C, SPI, CAN, …
- ADC
- PWM or DAC
- Timers
- Watchdog
- Clock generator and divider(s)
- Comparator(s)
- In-circuit programming and debugging support

AVRs are very different

- AtTiny13
- Up to 20 MIPS Througput at 20 MHz
- 64 SRAM/64 EEPROM/1k Flash
- Timer, ADC, 2 PWMs, Comparator, internal oscillator
- 0.24mA in active mode, 0.0001mA in sleep mode

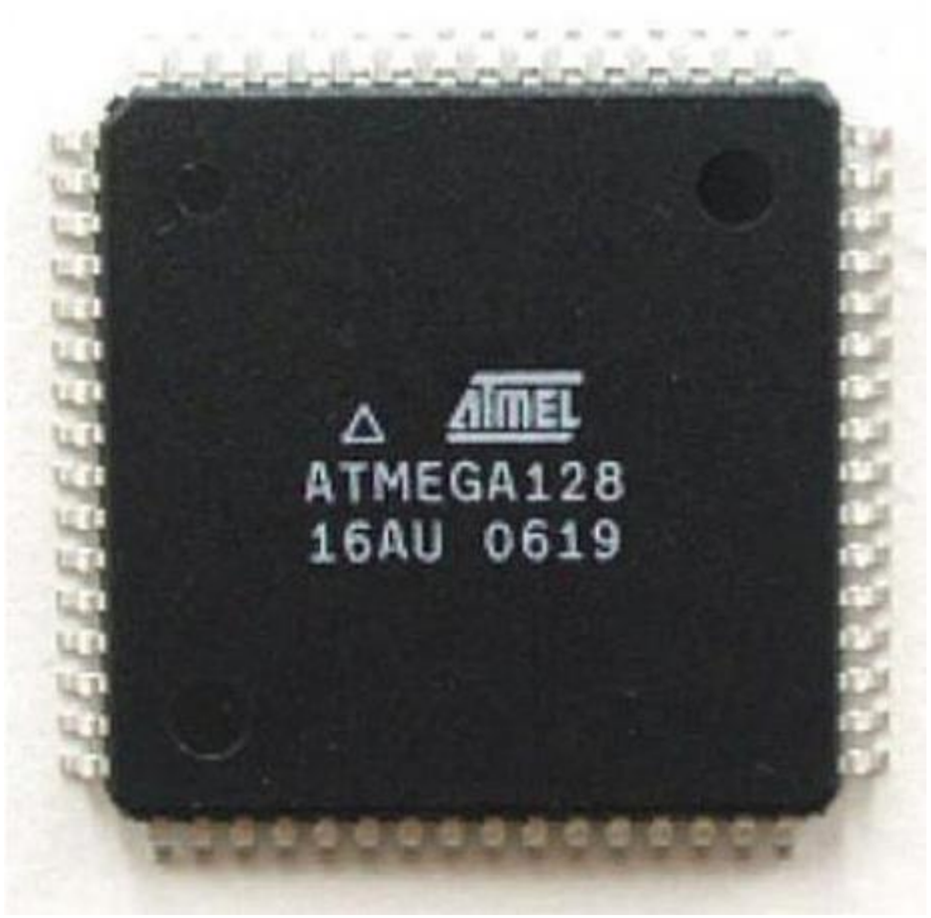
AVRs are very different

- Atmega32U4
- 2.5k SRAM/1k EEPROM/32k Flash
- JTAG
- USB
- PLL, Timers, PWMs, Comparators, ADCs, UARTs, Temperatures sensors, SPI, I^2C, … => tons of stuff



AVRs are very different

- Atmega128
- 4k SRAM/4k EEPROM/128k Flash
- JTAG
- Tons of stuff:…

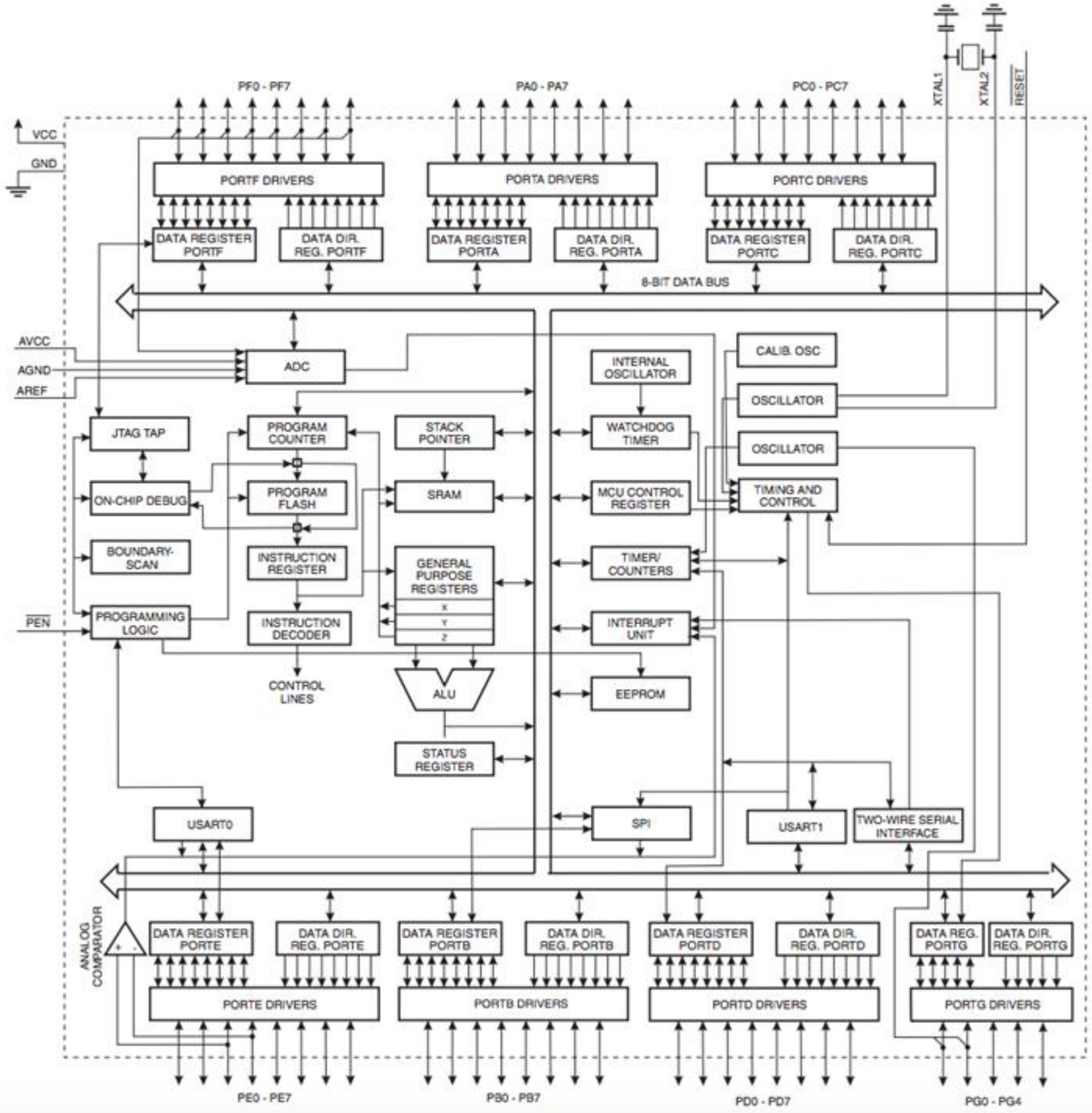


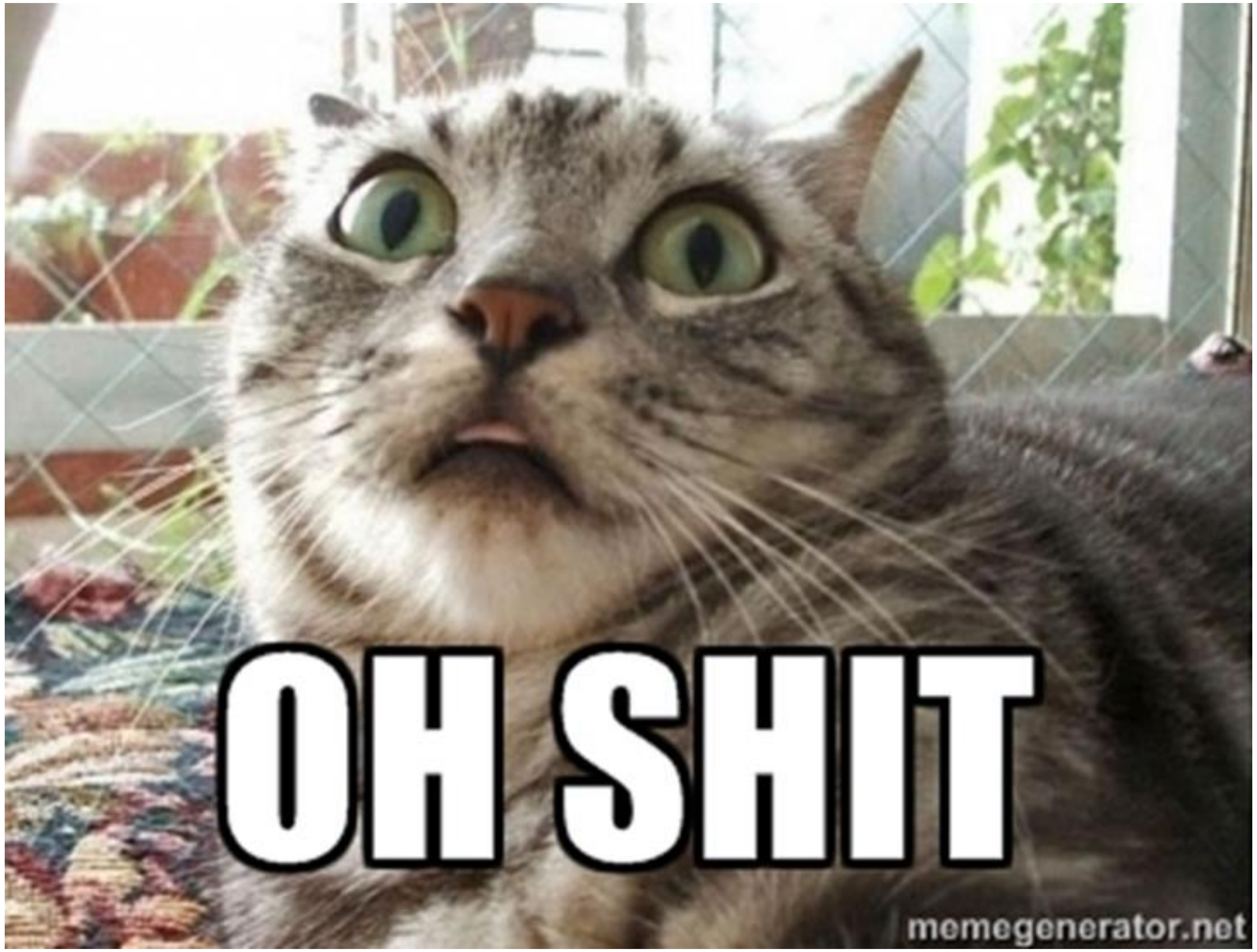
In the rest of the workshop we will focus on this chip

Why Atmega128?

- Old, but very widespread chip.
- At90can128 – popular analogue for CAN buses in automotive application
- **Cheap JTAG programmer**
- Much SRAM == ideal for ROP-chain construction training

Let’s look to the architecture of Atmega128…


PF0 - PF7
PA0 - PA7
PC0 - PC7
XTAL1
XTAL2
RESET
VCC
GND
PORTF DRIVERS
PORTA DRIVERS
PORTC DRIVERS
DATA REGISTER PORTF
DATA DIR. REG. PORTF
DATA REGISTER PORTA
DATA DIR. REG. PORTA
DATA REGISTER PORTC
DATA DIR. REG. PORTC
8-BIT DATA BUS
AVCC
AGND
AREF
ADC
CALIB. OSC
INTERNAL OSCILLATOR
OSCILLATOR
OSCILLATOR
JTAG TAP
PROGRAM COUNTER
STACK POINTER
WATCHDOG TIMER
TIMING AND CONTROL
ON-CHIP DEBUG
PROGRAM FLASH
SRAM
MCU CONTROL REGISTER
BOUNDARY-SCAN
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
TIMER/ COUNTERS
PEN
PROGRAMMING LOGIC
INSTRUCTION DECODER
INTERRUPT UNIT
CONTROL LINES
ALU
EEPROM
STATUS REGISTER
USART0
SPI
USART1
TWO-WIRE SERIAL INTERFACE
ANALOG COMPARATOR
+
-
DATA REGISTER PORTE
DATA DIR. REG. PORTE
DATA REGISTER PORTB
DATA DIR. REG. PORTB
DATA REGISTER PORTD
DATA DIR. REG. PORTD
DATA REG. PORTG
DATA DIR. REG. PORTG
PORTE DRIVERS
PORTB DRIVERS
PORTD DRIVERS
PORTG DRIVERS
PE0 - PE7
PB0 - PB7
PD0 - PD7
PG0 - PG4


OH SHIT
memegenerator.net

Ok, ok, let's simplify a bit ☺

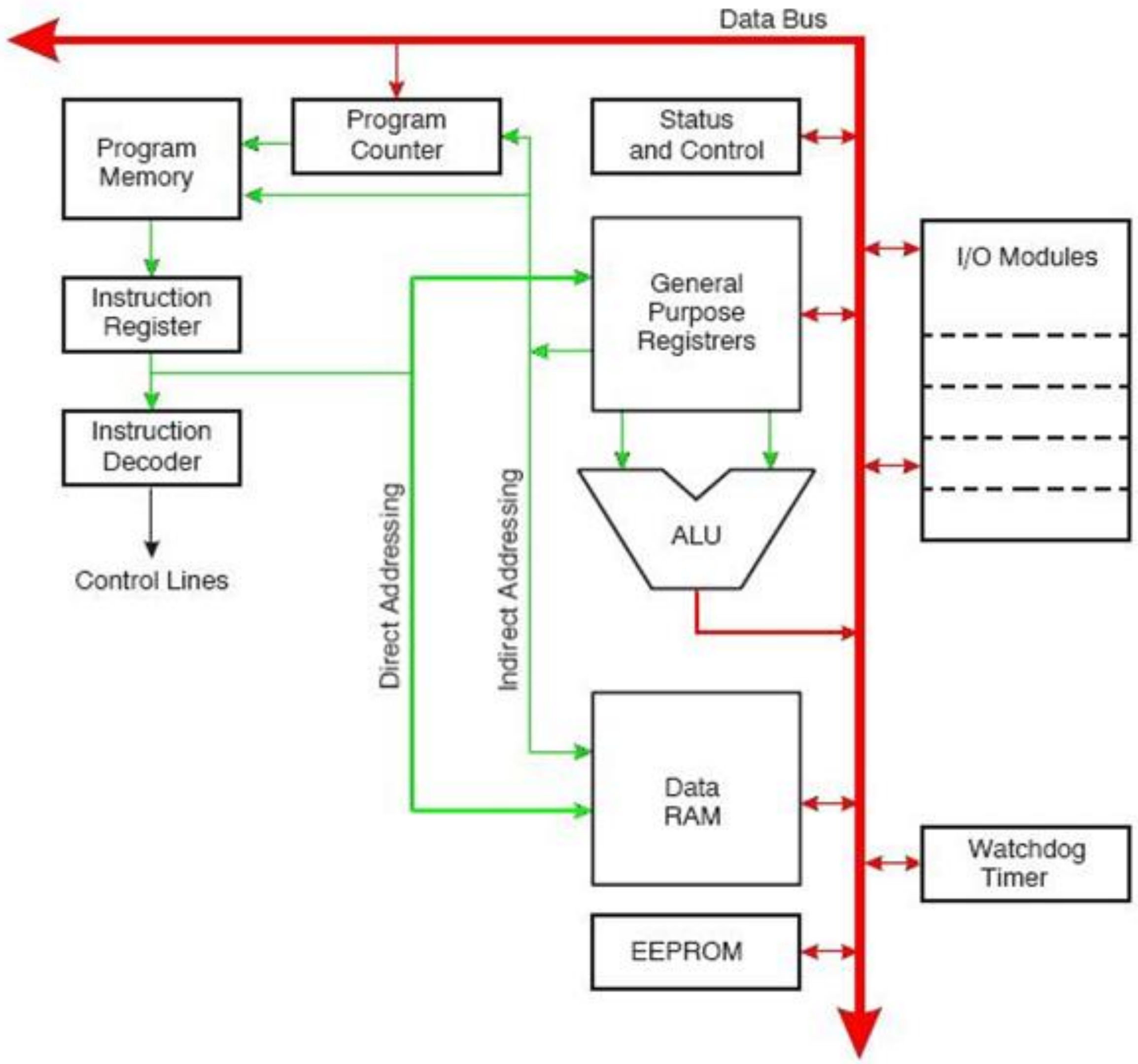


Image: http://www.cs.jhu.edu/~jorgev/cs333/usbkey/uC_3.JPG

Note: code is <u>separated</u> from data

Memory map

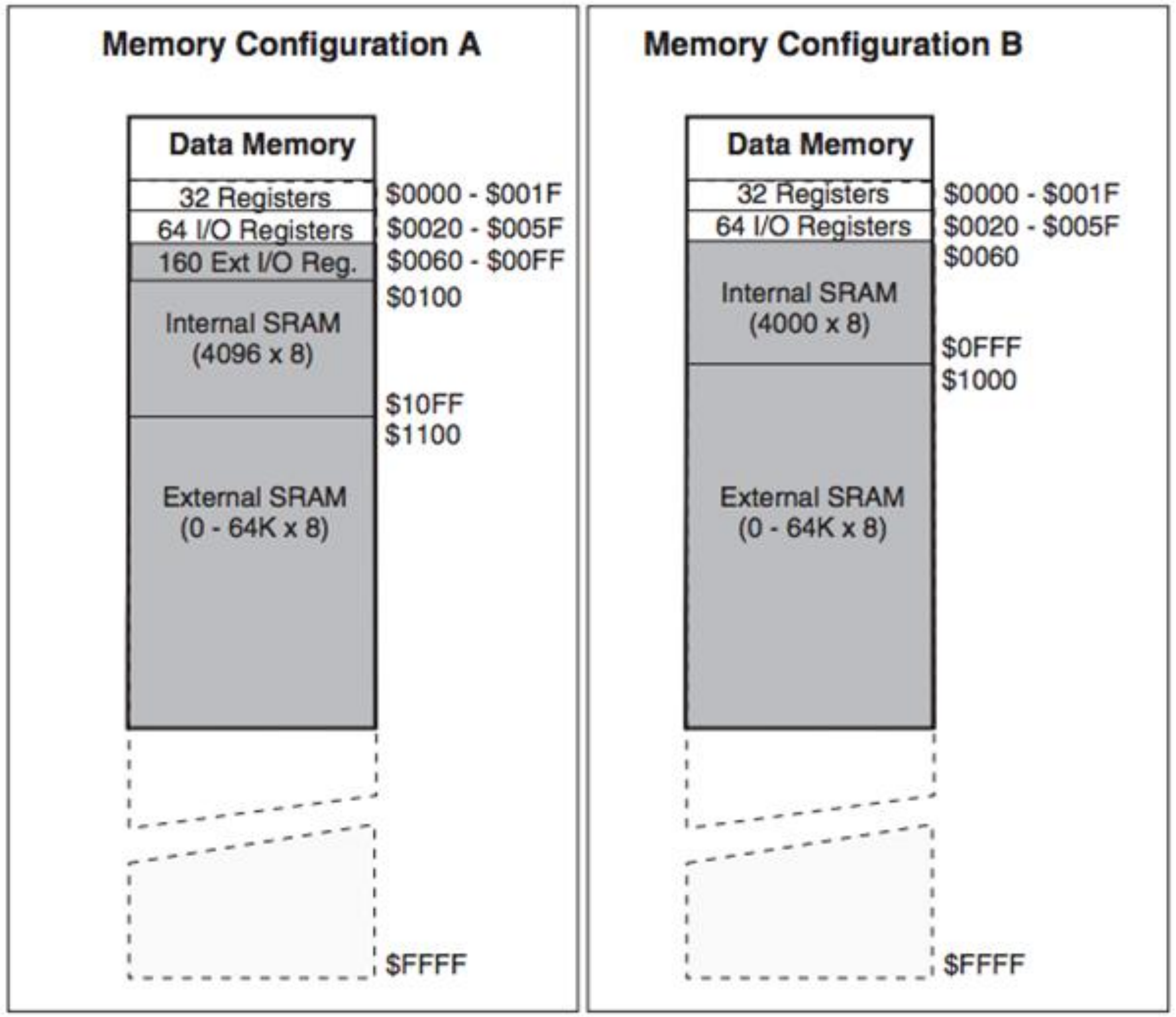
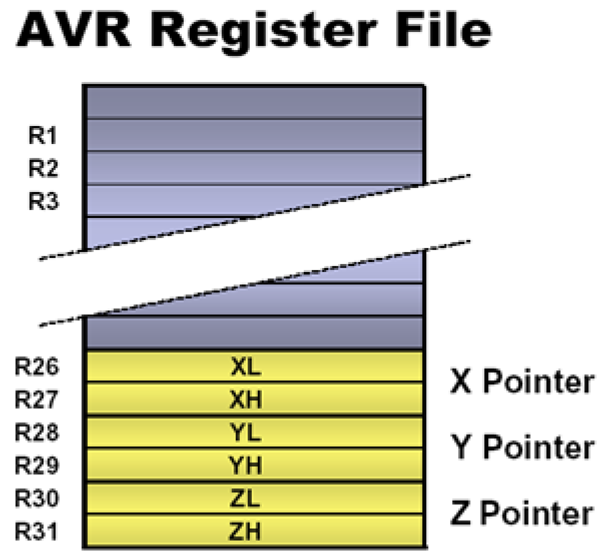


Figure 9. Data Memory Map

Memory: registers

- R1-R25 – GPR
- X,Y,Z – pair “working” registers, e.g. for memory addressing operations
- I/O registers – for accessing different “hardware”



Memory: special registers

- PC – program counter, 16-bit register
- SP – stack pointer, 16-bit register (SPH:SPL)
- SREG – status register (8-bit)

Memory addressing

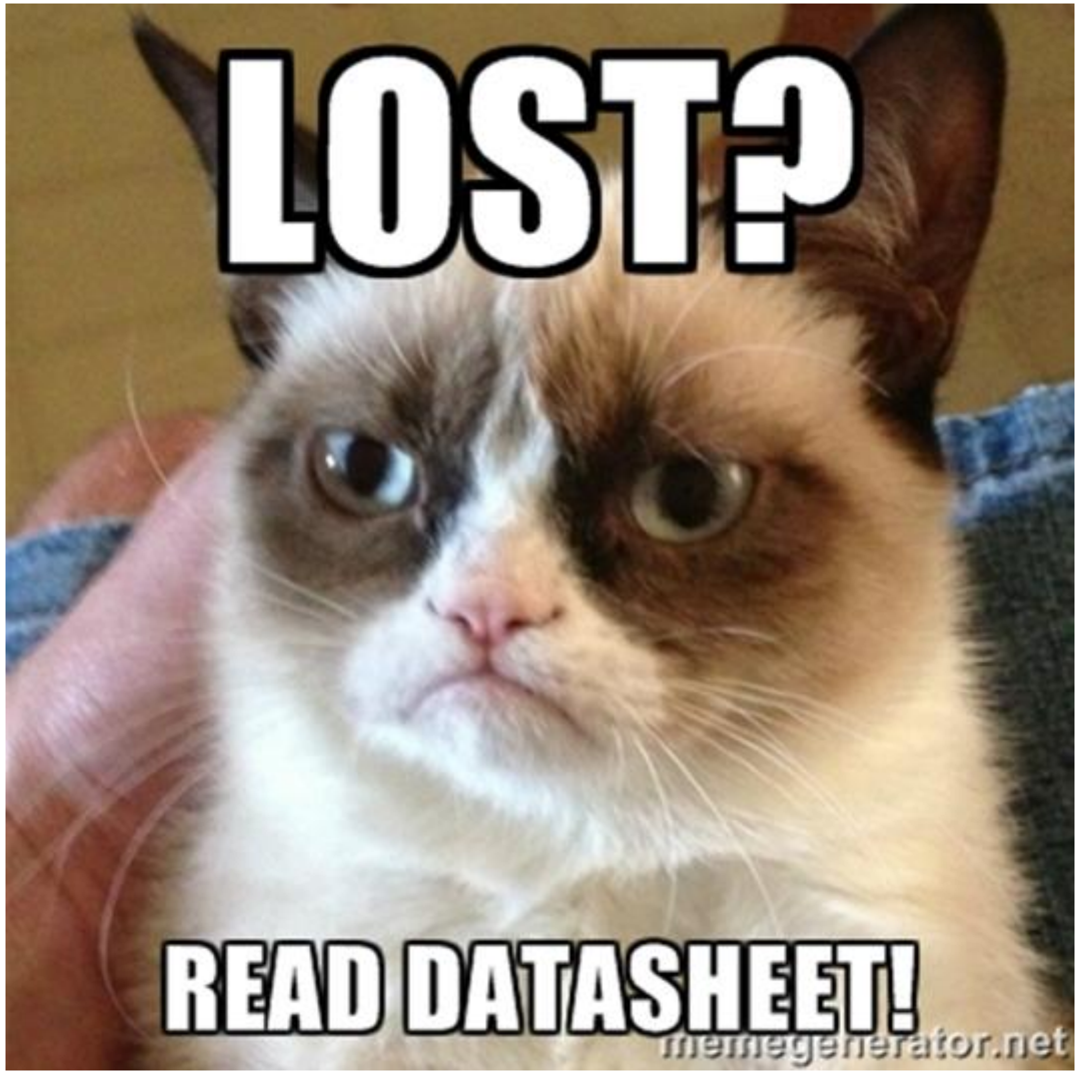
- SRAM/EEPROM – 16-bit addressing, 8-bit element
- Flash – 16(8)-bit addressing, 16-bit element

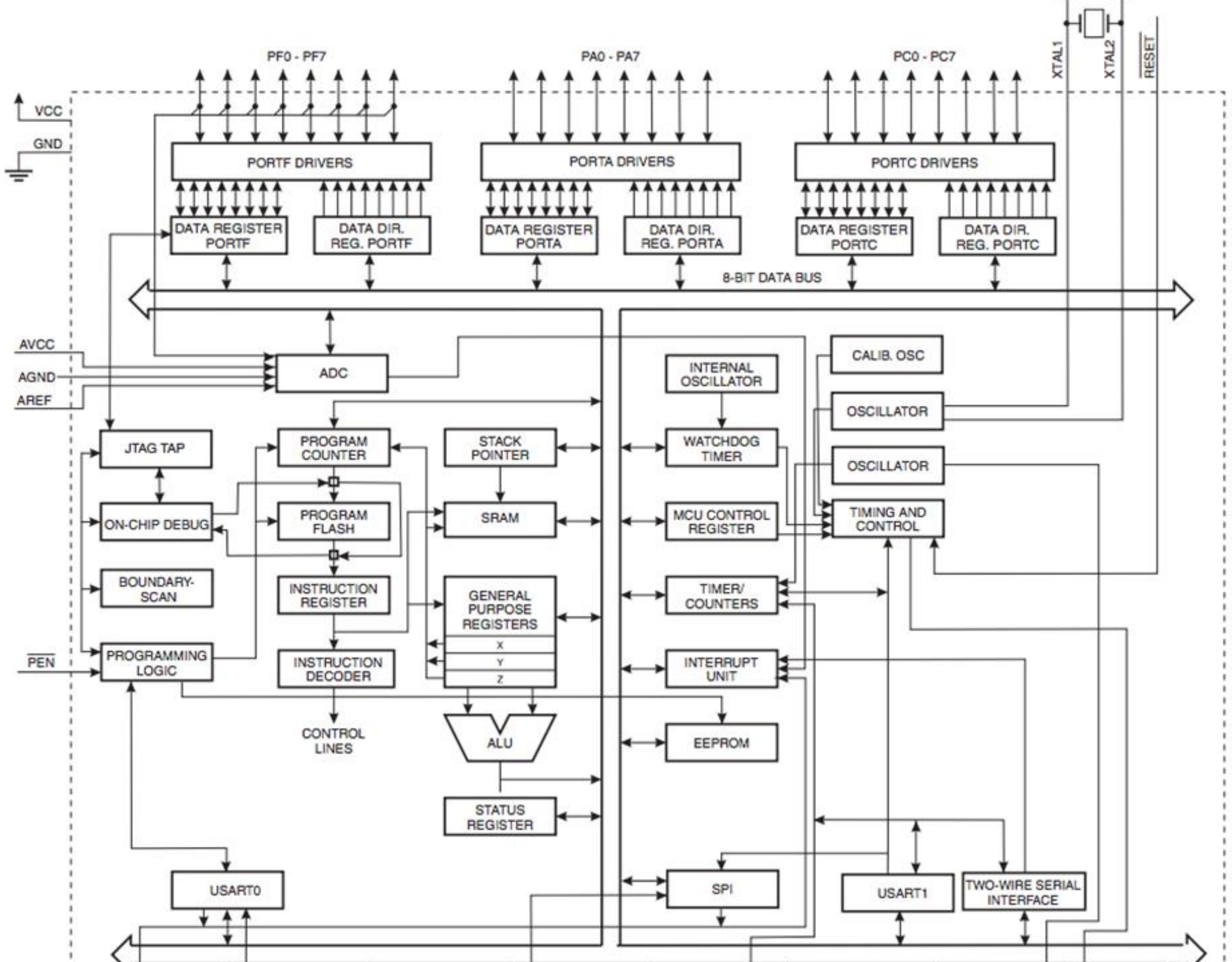
LPM command!

Memory addressing directions

- Direct to register
- Direct to I/O
- SRAM **direct**
- SRAM **indirect** (pre- and post- increment)
- Flash direct

Datasheets are your best friends!





PF0 - PF7
PA0 - PA7
PC0 - PC7
XTAL1
XTAL2
RESET
VCC
GND
PORTF DRIVERS
PORTA DRIVERS
PORTC DRIVERS
DATA REGISTER PORTF
DATA DIR. REG. PORTF
DATA REGISTER PORTA
DATA DIR. REG. PORTA
DATA REGISTER PORTC
DATA DIR. REG. PORTC
8-BIT DATA BUS
AVCC
AGND
AREF
ADC
CALIB. OSC
INTERNAL OSCILLATOR
OSCILLATOR
OSCILLATOR
JTAG TAP
PROGRAM COUNTER
STACK POINTER
WATCHDOG TIMER
TIMING AND CONTROL
ON-CHIP DEBUG
PROGRAM FLASH
SRAM
MCU CONTROL REGISTER
BOUNDARY-SCAN
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
TIMER/ COUNTERS
PEN
PROGRAMMING LOGIC
INSTRUCTION DECODER
INTERRUPT UNIT
CONTROL LINES
ALU
EEPROM
STATUS REGISTER
USART0
SPI
USART1
TWO-WIRE SERIAL INTERFACE

Interrupts

- Interrupts normal process of code execution for handling something or reacting to some event
- Interrupt handler – procedure to be executed after interrupt; address stored in the interrupt vector
- Examples of interrupts:
  - Timers
  - Hardware events
  - Reset

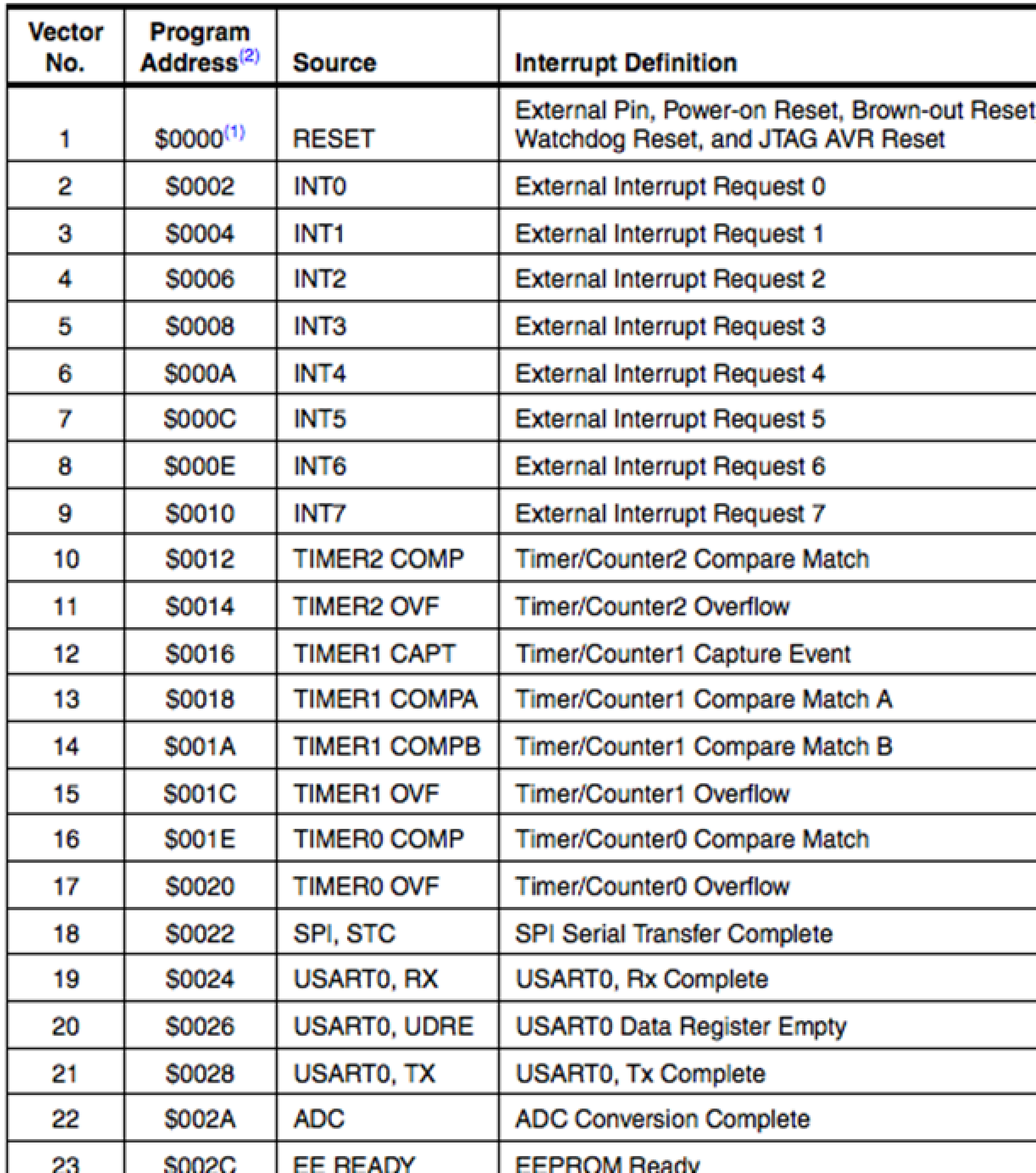


Table 23. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	USART0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete
22	$002A	ADC	ADC Conversion Complete
23	$002C	EE READY	EEPROM Ready

AVR assembly

In a very quick manner

Instruction types

- Arithmetic and logic
- Bit manipulation/test
- Memory manipulation
- Unconditional jump/call
- Branch commands
- SREG manipulation
- Special (watchdog, etc)

Instruction mnemonics

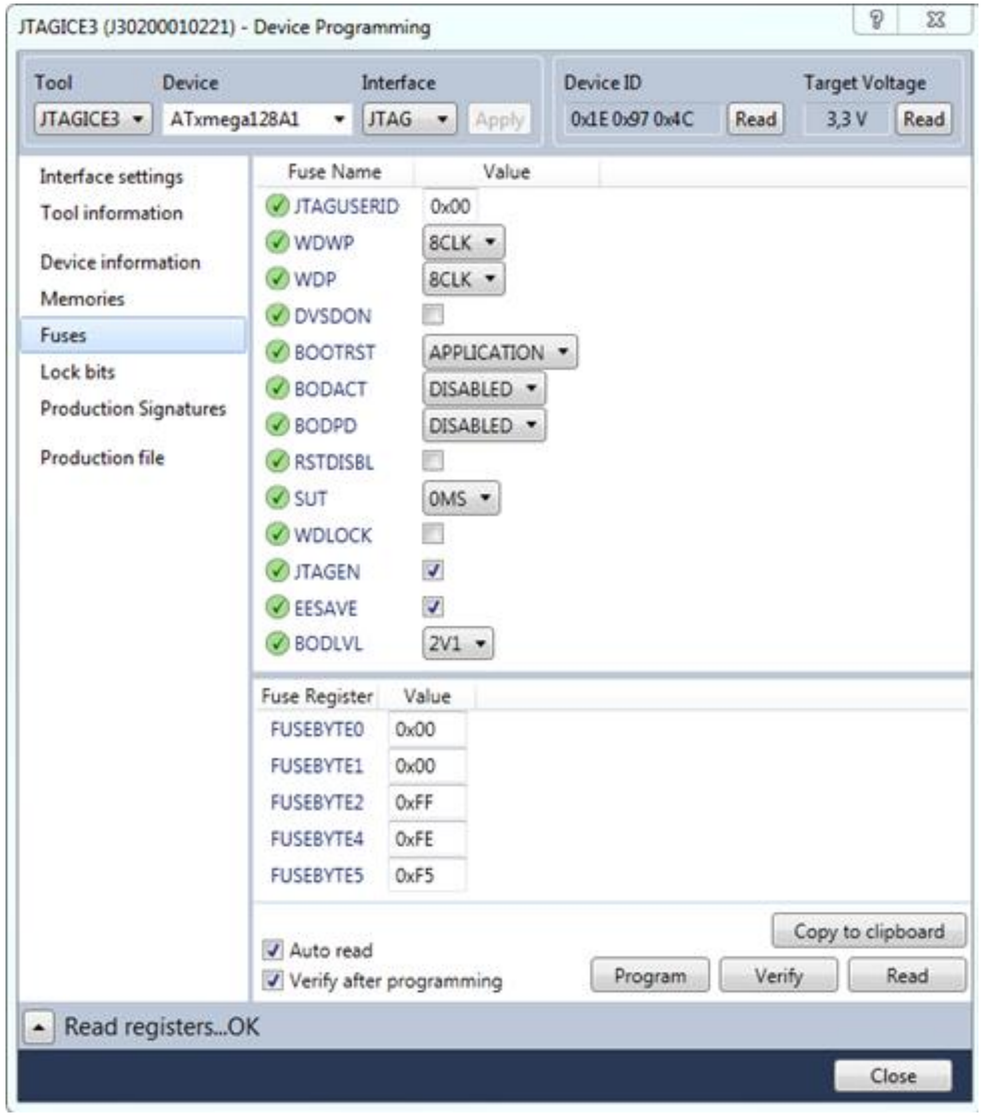
```
mov       r16,r0        ; Copy r0 to r16
out       PORTA, r16    ; Write r16 to PORTA
```

16-bit long
“Intel syntax” (destination **before** source)

A bit more about architecture

Fuses and Lock Bits

- Several bytes of permanent storage
- Set internal hardware and features configuration, including oscillator (int or ext), bootloader, pin, ability to debug/programm, etc.
- 2 lock bits controls programming protection.



JTAGICE3 (J30200010221) - Device Programming

Tool	Device	Interface	Device ID	Target Voltage
JTAGICE3	ATxmega128A1	JTAG	0x1E 0x97 0x4C	3,3 V

Fuse Name	Value
JTAGUSERID	0x00
WDWP	8CLK
WDP	8CLK
DVSDON	☐
BOOTRST	APPLICATION
BODACT	DISABLED
BODPD	DISABLED
RSTDISBL	☐
SUT	0MS
WDLOCK	☐
JTAGEN	✓
EESAVE	✓
BODLVL	2V1

Fuse Register	Value
FUSEBYTE0	0x00
FUSEBYTE1	0x00
FUSEBYTE2	0xFF
FUSEBYTE4	0xFE
FUSEBYTE5	0xF5

✓ Auto read
✓ Verify after programming

Read registers...OK

AVR bootloader – what is it?

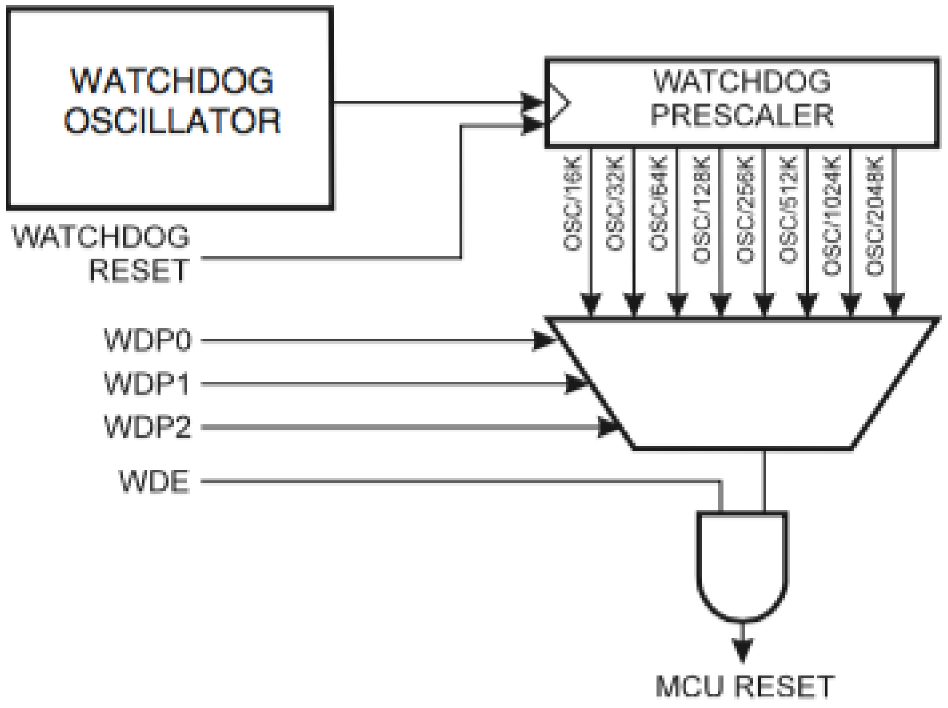
- Part of code that starts **BEFORE** RESET interrupt.
- Could be used for self-programmable (i.e. without external device) systems, in case you need to supply firmware update for your IoT device.
- Bootloader address and behavior configured via FUSEs.
- BLB lock bits controls bootloader ability to update application and/or bootloader parts of flash.

AVR bootloaders

- Arduino bootloader
- USB bootloaders (AVRUSBBoot)
- Serial programmer bootloaders (STK500-compatible)
- Cryptobootloaders
- …
- Tons of them!

Watchdog

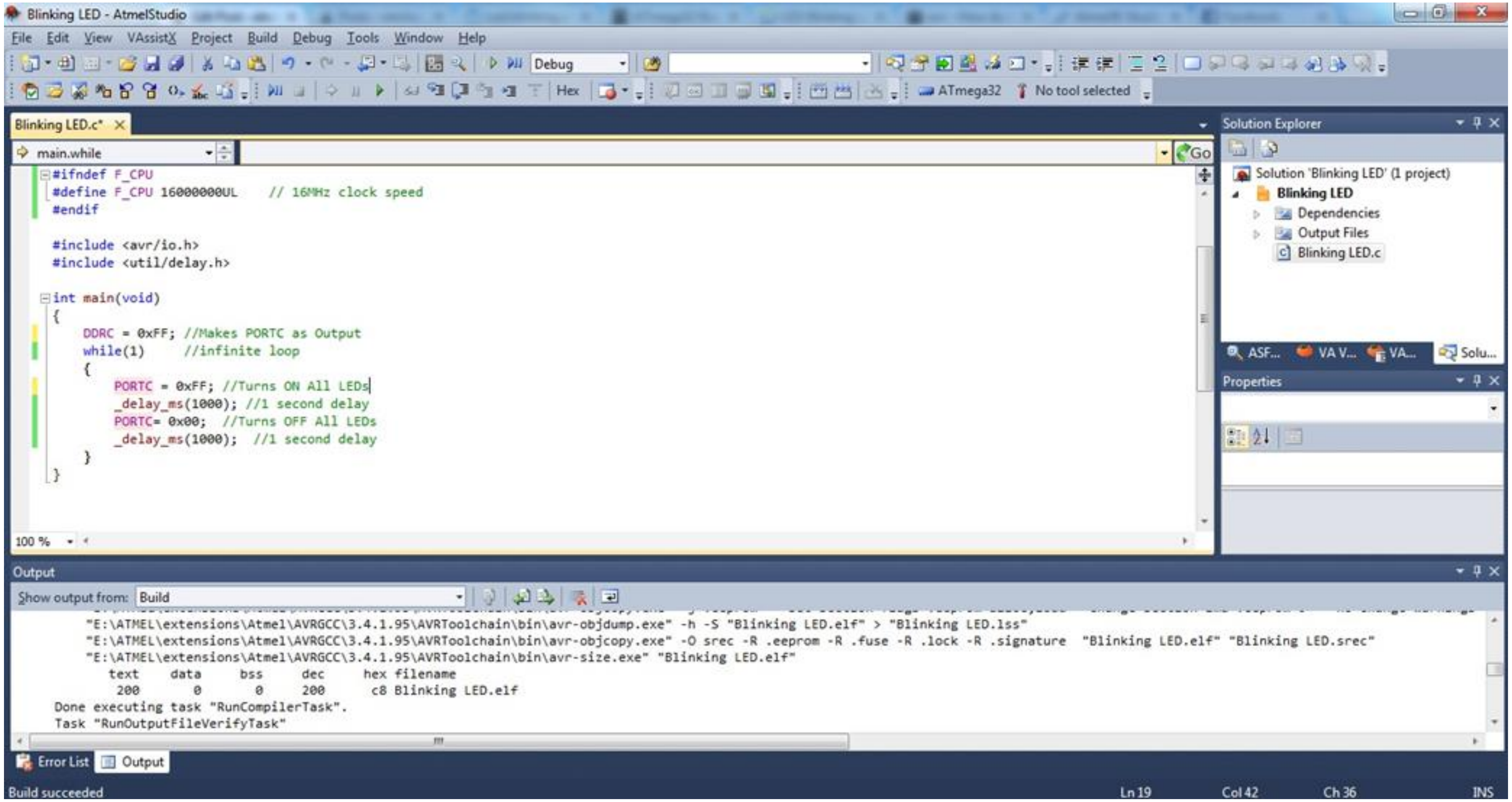
- Timer that could be used for interrupt or reset device.
- Cleared with **WDR** instruction.



http://ardiri.com/blog/entries/20141028/watchdog.jpg

Development for AVR

Atmel studio



```c
#ifndef F_CPU
#define F_CPU 16000000UL    // 16MHz clock speed
#endif

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRC = 0xFF; //Makes PORTC as Output
    while(1)     //infinite loop
    {
        PORTC = 0xFF; //Turns ON All LEDs
        _delay_ms(1000); //1 second delay
        PORTC= 0x00;  //Turns OFF All LEDs
        _delay_ms(1000);  //1 second delay
    }
}
```

```
        "E:\ATMEL\extensions\Atmel\AVRGCC\3.4.1.95\AVRToolchain\bin\avr-objdump.exe" -h -S "Blinking LED.elf" > "Blinking LED.lss"
        "E:\ATMEL\extensions\Atmel\AVRGCC\3.4.1.95\AVRToolchain\bin\avr-objcopy.exe" -O srec -R .eeprom -R .fuse -R .lock -R .signature  "Blinking LED.elf" "Blinking LED.srec"
        "E:\ATMEL\extensions\Atmel\AVRGCC\3.4.1.95\AVRToolchain\bin\avr-size.exe" "Blinking LED.elf"
           text    data     bss     dec     hex filename
            200       0       0     200      c8 Blinking LED.elf
    Done executing task "RunCompilerTask".
    Task "RunOutputFileVerifyTask"
```

AVR-GCC

- Main compiler/debugger kit for the platform
- Used by Atmel studio
- Use “AVR libc” -- http://www.nongnu.org/avr-libc/
- Several optimization options, several memory models

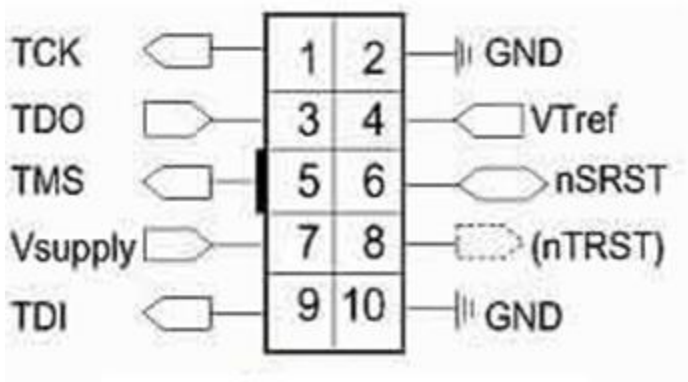
Other tools

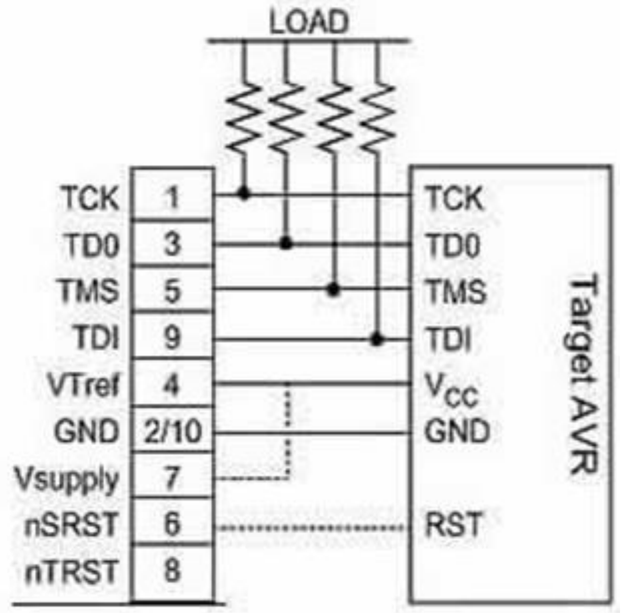
- Arduino
- CodeVision AVR
- IAR Embedded workbench

Debugging AVR

JTAG

- Joint Test Action Group (JTAG)
- Special debugging interface added to a chip
- Allows testing, debugging, firmware manipulation and boundary scanning.
- Requires external hardware





JTAG for AVRs

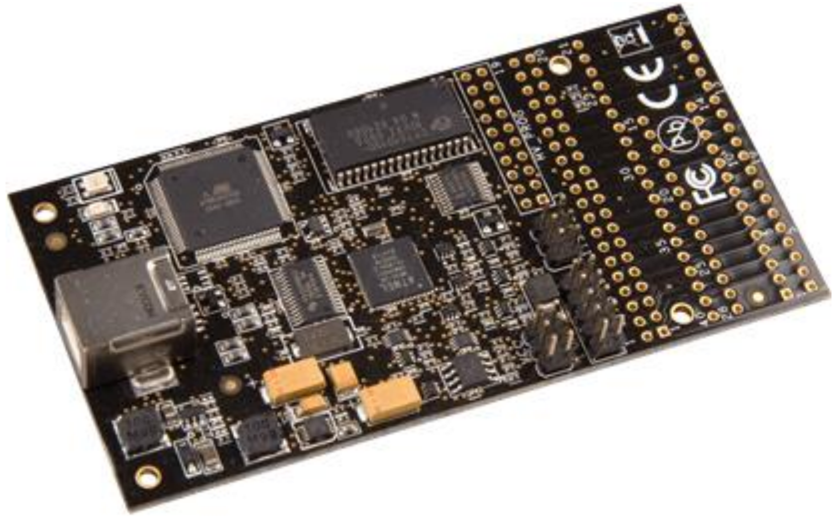
AVR JTAGIce3

AVR JTAG mkII

AVR JTAG mkI

Atmel ICE3

AVR Dragon



Avarice

- Open-source interface between AVR JTAG and GDB
- Also allow to flash/write eeprom, manipulate fuse and lock bits.
- Could *capture* the exeuction flow to restore the firmware
- Example usage:

```
avarice --program --file test.elf --part atmega128 --jtag /dev/ttyUSB0 :4444
```

AVR-GDB

- Part of “nongnu” AVR gcc kit.
- Roughly ported standard gdb to AVR platform
- Doesn’t understand Harvard architecture, i.e. to read flash you will need to resolve it by reference of $pc:

```
(gdb) x/10b $pc + 100
```

Simulators

- Atmel Studio simulator
- Proteus simulator
- Simavr
- Simulavr

VM access:

Login: radare

Password: radare

Ex 1.1: Hello world!

EXAMPLE

Real hardware

```
cd /home/radare/workshop/ex1.1
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex1.1.hex -g :4242
avr-gdb
```

Communication: `cutecom` or `screen /dev/ttyUSB1 9600`

Simulator

```
cd /home/radare/workshop/ex1.1_simulator
simulavr -P atmega128 -F 16000000 -f build-crumbuino128/ex1.1.elf
avr-gdb
```

Ex 1.2: Blink!

EXAMPLE

Real hardware

```
cd /home/radare/workshop/ex1.2
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex1.2.hex -g :4242
avr-gdb
```

AVR RE

Reverse engineering AVR binaries

Pure disassemblers:

- avr-objdump – gcc kit standard tool
- Vavrdisasm -- https://github.com/vsergeev/vavrdisasm
- ODAweb -- https://www.onlinedisassembler.com/odaweb/

“Normal” disassemblers:

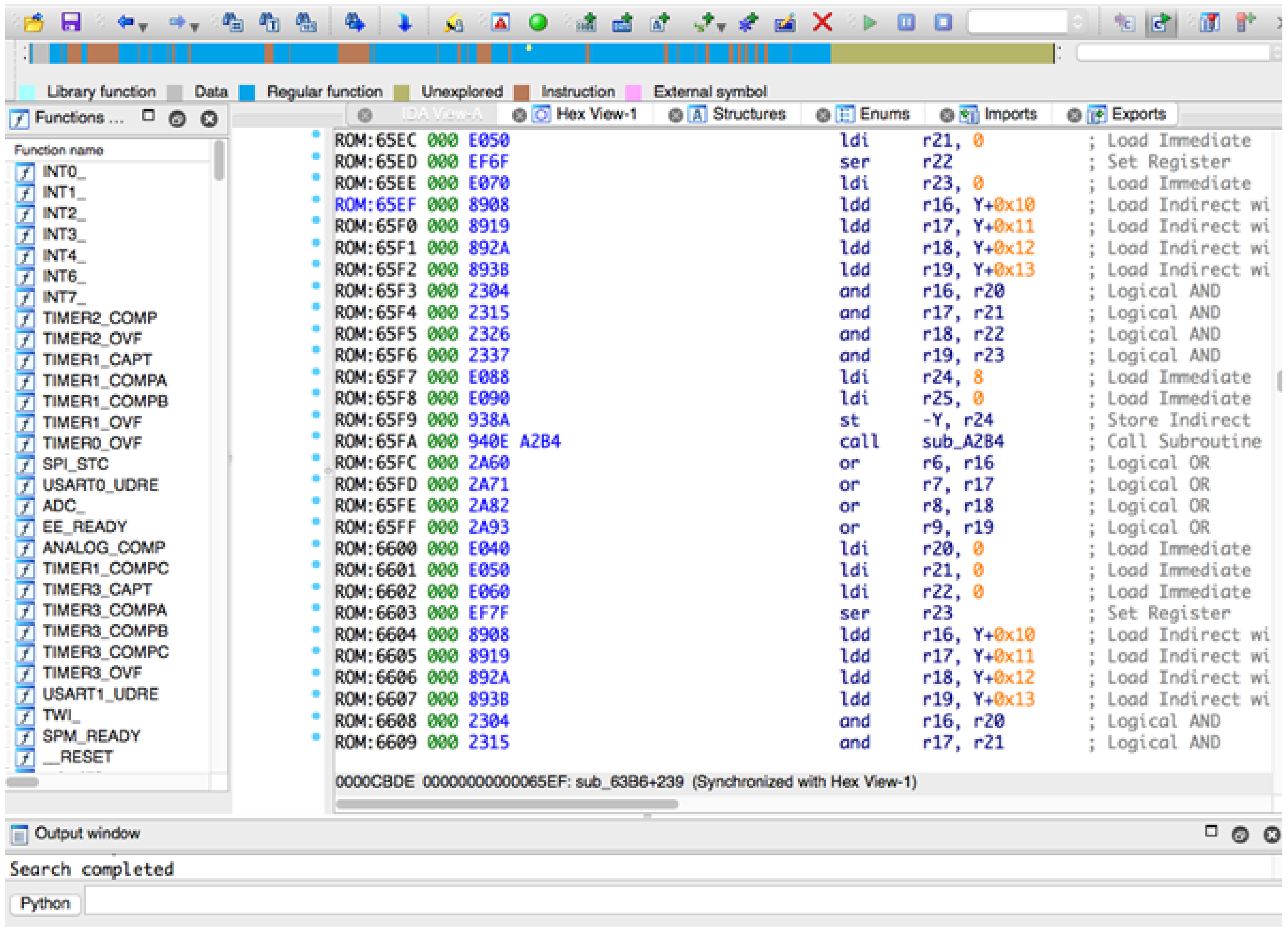
- IDA Pro
- Radare

IDA PRO: AVR specifics

- Incorrect AVR elf-handling
- Incorrect LPM command behavior
- Addressing issues
- Sometimes strange output

...

- However, usable, but “with care”



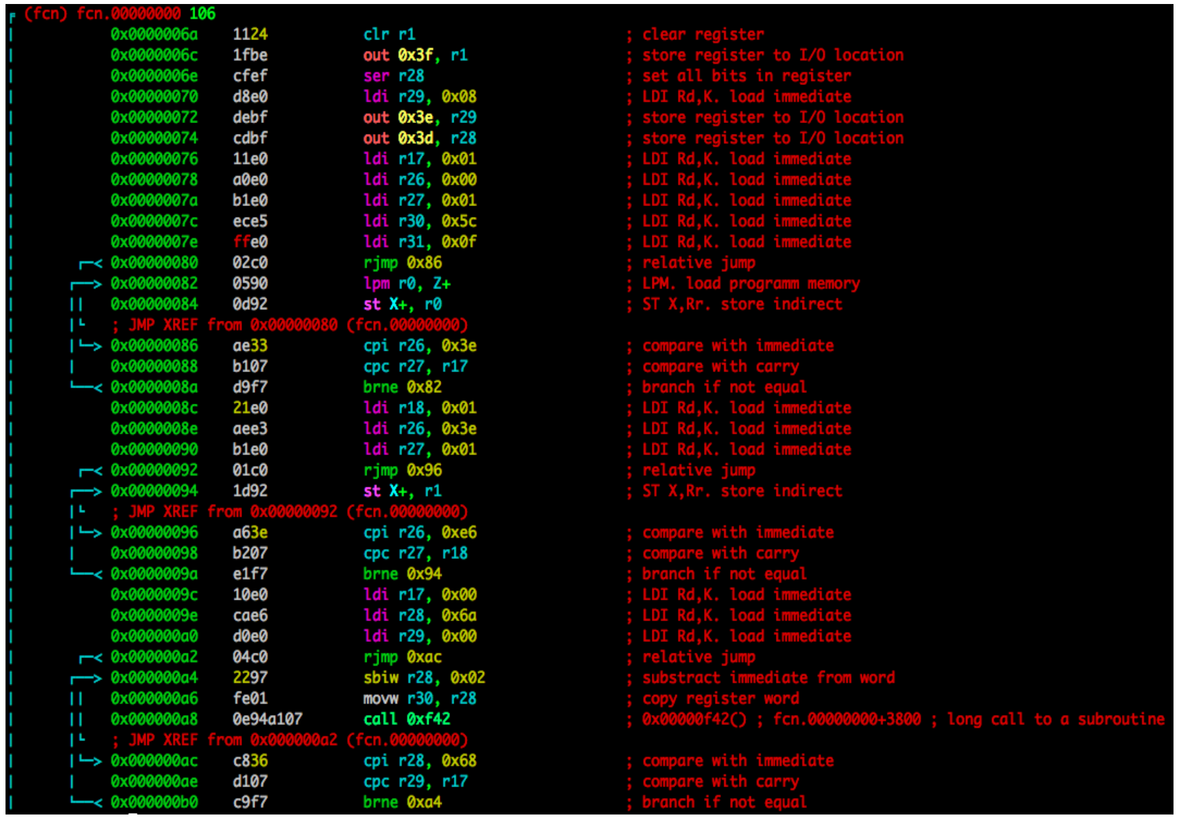
INT0_
INT1_
INT2_
INT3_
INT4_
INT6_
INT7_
TIMER2_COMP
TIMER2_OVF
TIMER1_CAPT
TIMER1_COMPA
TIMER1_COMPB
TIMER1_OVF
TIMER0_OVF
SPI_STC
USART0_UDRE
ADC_
EE_READY
ANALOG_COMP
TIMER1_COMPC
TIMER3_CAPT
TIMER3_COMPA
TIMER3_COMPB
TIMER3_COMPC
TIMER3_OVF
USART1_UDRE
TWI_
SPM_READY
__RESET

```
ROM:65EC 000 E050          ldi   r21, 0       ; Load Immediate
ROM:65ED 000 EF6F          ser   r22          ; Set Register
ROM:65EE 000 E070          ldi   r23, 0       ; Load Immediate
ROM:65EF 000 8908          ldd   r16, Y+0x10  ; Load Indirect wi
ROM:65F0 000 8919          ldd   r17, Y+0x11  ; Load Indirect wi
ROM:65F1 000 892A          ldd   r18, Y+0x12  ; Load Indirect wi
ROM:65F2 000 893B          ldd   r19, Y+0x13  ; Load Indirect wi
ROM:65F3 000 2304          and   r16, r20     ; Logical AND
ROM:65F4 000 2315          and   r17, r21     ; Logical AND
ROM:65F5 000 2326          and   r18, r22     ; Logical AND
ROM:65F6 000 2337          and   r19, r23     ; Logical AND
ROM:65F7 000 E088          ldi   r24, 8       ; Load Immediate
ROM:65F8 000 E090          ldi   r25, 0       ; Load Immediate
ROM:65F9 000 938A          st    -Y, r24      ; Store Indirect
ROM:65FA 000 940E A2B4     call  sub_A2B4     ; Call Subroutine
ROM:65FC 000 2A60          or    r6, r16      ; Logical OR
ROM:65FD 000 2A71          or    r7, r17      ; Logical OR
ROM:65FE 000 2A82          or    r8, r18      ; Logical OR
ROM:65FF 000 2A93          or    r9, r19      ; Logical OR
ROM:6600 000 E040          ldi   r20, 0       ; Load Immediate
ROM:6601 000 E050          ldi   r21, 0       ; Load Immediate
ROM:6602 000 E060          ldi   r22, 0       ; Load Immediate
ROM:6603 000 EF7F          ser   r23          ; Set Register
ROM:6604 000 8908          ldd   r16, Y+0x10  ; Load Indirect wi
ROM:6605 000 8919          ldd   r17, Y+0x11  ; Load Indirect wi
ROM:6606 000 892A          ldd   r18, Y+0x12  ; Load Indirect wi
ROM:6607 000 893B          ldd   r19, Y+0x13  ; Load Indirect wi
ROM:6608 000 2304          and   r16, r20     ; Logical AND
ROM:6609 000 2315          and   r17, r21     ; Logical AND
0000CBDE 00000000000065EF: sub_63B6+239 (Synchronized with Hex View-1)
```

```
Search completed
```

Radare2

- Opensource reverse engineering framework (RE, debugger, forensics)
- Crossplatform (Linux,Mac,Windows,QNX,Android,iOS, …)
- Scripting
- A lot of Architectures / file-formats
- …
- Without habitual GUI



```
┌ (fcn) fcn.00000000 106
|           0x0000006a    1124           clr r1                        ; clear register
|           0x0000006c    1fbe           out 0x3f, r1                  ; store register to I/O location
|           0x0000006e    cfef           ser r28                       ; set all bits in register
|           0x00000070    d8e0           ldi r29, 0x08                 ; LDI Rd,K. load immediate
|           0x00000072    debf           out 0x3e, r29                 ; store register to I/O location
|           0x00000074    cdbf           out 0x3d, r28                 ; store register to I/O location
|           0x00000076    11e0           ldi r17, 0x01                 ; LDI Rd,K. load immediate
|           0x00000078    a0e0           ldi r26, 0x00                 ; LDI Rd,K. load immediate
|           0x0000007a    b1e0           ldi r27, 0x01                 ; LDI Rd,K. load immediate
|           0x0000007c    ece5           ldi r30, 0x5c                 ; LDI Rd,K. load immediate
|           0x0000007e    ffe0           ldi r31, 0x0f                 ; LDI Rd,K. load immediate
|       ┌─< 0x00000080    02c0           rjmp 0x86                     ; relative jump
|      ┌──> 0x00000082    0590           lpm r0, Z+                    ; LPM. load programm memory
|      ||   0x00000084    0d92           st X+, r0                     ; ST X,Rr. store indirect
|      |└   ; JMP XREF from 0x00000080 (fcn.00000000)
|      |└─> 0x00000086    ae33           cpi r26, 0x3e                 ; compare with immediate
|      |    0x00000088    b107           cpc r27, r17                  ; compare with carry
|      └──< 0x0000008a    d9f7           brne 0x82                     ; branch if not equal
|           0x0000008c    21e0           ldi r18, 0x01                 ; LDI Rd,K. load immediate
|           0x0000008e    aee3           ldi r26, 0x3e                 ; LDI Rd,K. load immediate
|           0x00000090    b1e0           ldi r27, 0x01                 ; LDI Rd,K. load immediate
|       ┌─< 0x00000092    01c0           rjmp 0x96                     ; relative jump
|      ┌──> 0x00000094    1d92           st X+, r1                     ; ST X,Rr. store indirect
|      |└   ; JMP XREF from 0x00000092 (fcn.00000000)
|      |└─> 0x00000096    a63e           cpi r26, 0xe6                 ; compare with immediate
|      |    0x00000098    b207           cpc r27, r18                  ; compare with carry
|      └──< 0x0000009a    e1f7           brne 0x94                     ; branch if not equal
|           0x0000009c    10e0           ldi r17, 0x00                 ; LDI Rd,K. load immediate
|           0x0000009e    cae6           ldi r28, 0x6a                 ; LDI Rd,K. load immediate
|           0x000000a0    d0e0           ldi r29, 0x00                 ; LDI Rd,K. load immediate
|       ┌─< 0x000000a2    04c0           rjmp 0xac                     ; relative jump
|      ┌──> 0x000000a4    2297           sbiw r28, 0x02                ; substract immediate from word
|      ||   0x000000a6    fe01           movw r30, r28                 ; copy register word
|      ||   0x000000a8    0e94a107       call 0xf42                    ; 0x00000f42() ; fcn.00000000+3800 ; long call to a subroutine
|      |└   ; JMP XREF from 0x000000a2 (fcn.00000000)
|      |└─> 0x000000ac    c836           cpi r28, 0x68                 ; compare with immediate
|      |    0x000000ae    d107           cpc r29, r17                  ; compare with carry
|      └──< 0x000000b0    c9f7           brne 0xa4                     ; branch if not equal
```

Radare2. Tools

- radare2
- rabin2
- radiff2
- rafind2
- rasm2
- r2pm
- rarun2
- rax2
- r2agent
- ragg2
- rahash2
- rasign2

Radare2. Using

- Install from git

  ```
  # git clone https://github.com/radare/radare2
  # cd radare2
  # sys/install.sh
  ```

- Packages (yara, retdec / radeco decompilers, …):

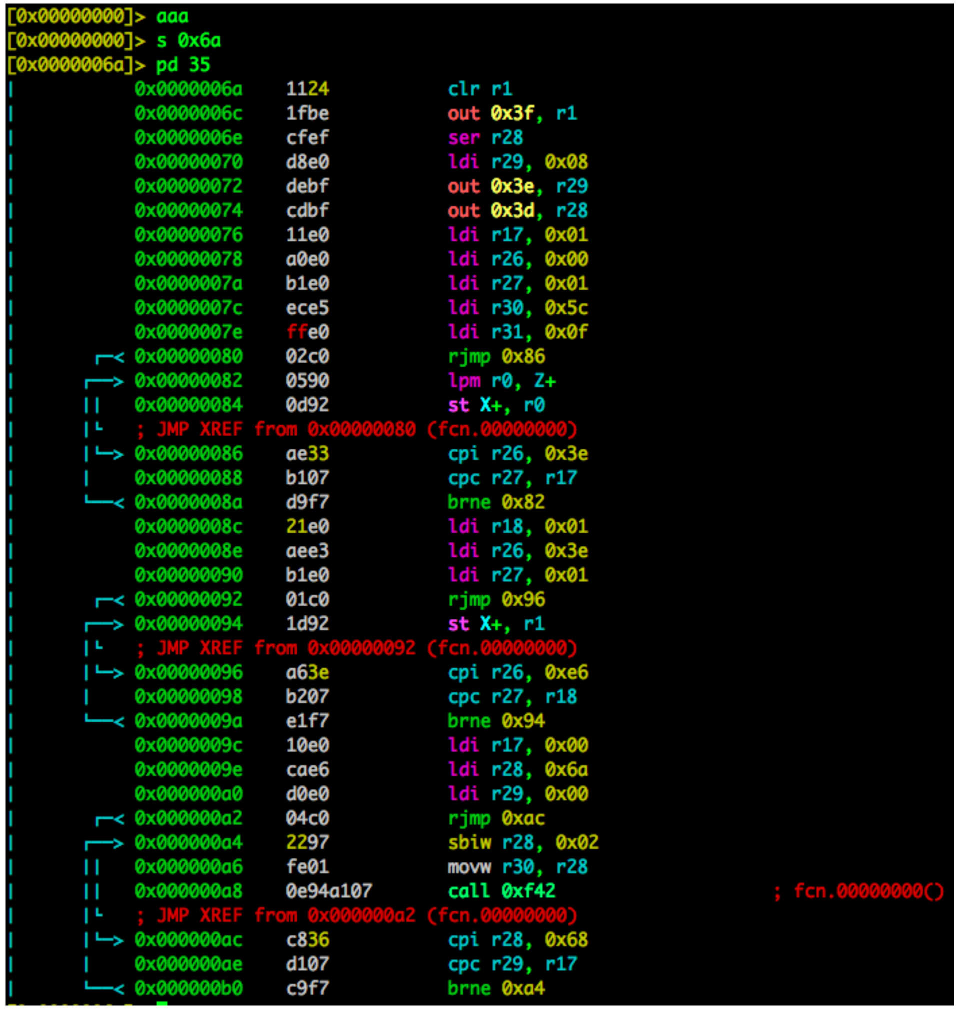
  ```
  # r2pm -i radare2
  ```

- Console commands

  ```
  # r2 -d /bin/ls – debugging
  # r2 –a avr sample.bin – architecture
  # r2 –b 16 sample.bin – specify register size in bits
  # r2 sample.bin –i script – include script
  ```

Radare2. Basic commands

- aaa – analyze
- axt – xrefs
- s – seek
- p – disassemble
- ~ - grep
- ! – run shell commands
- / – search
- /R – search ROP
- /c – search instruction
- ? – help

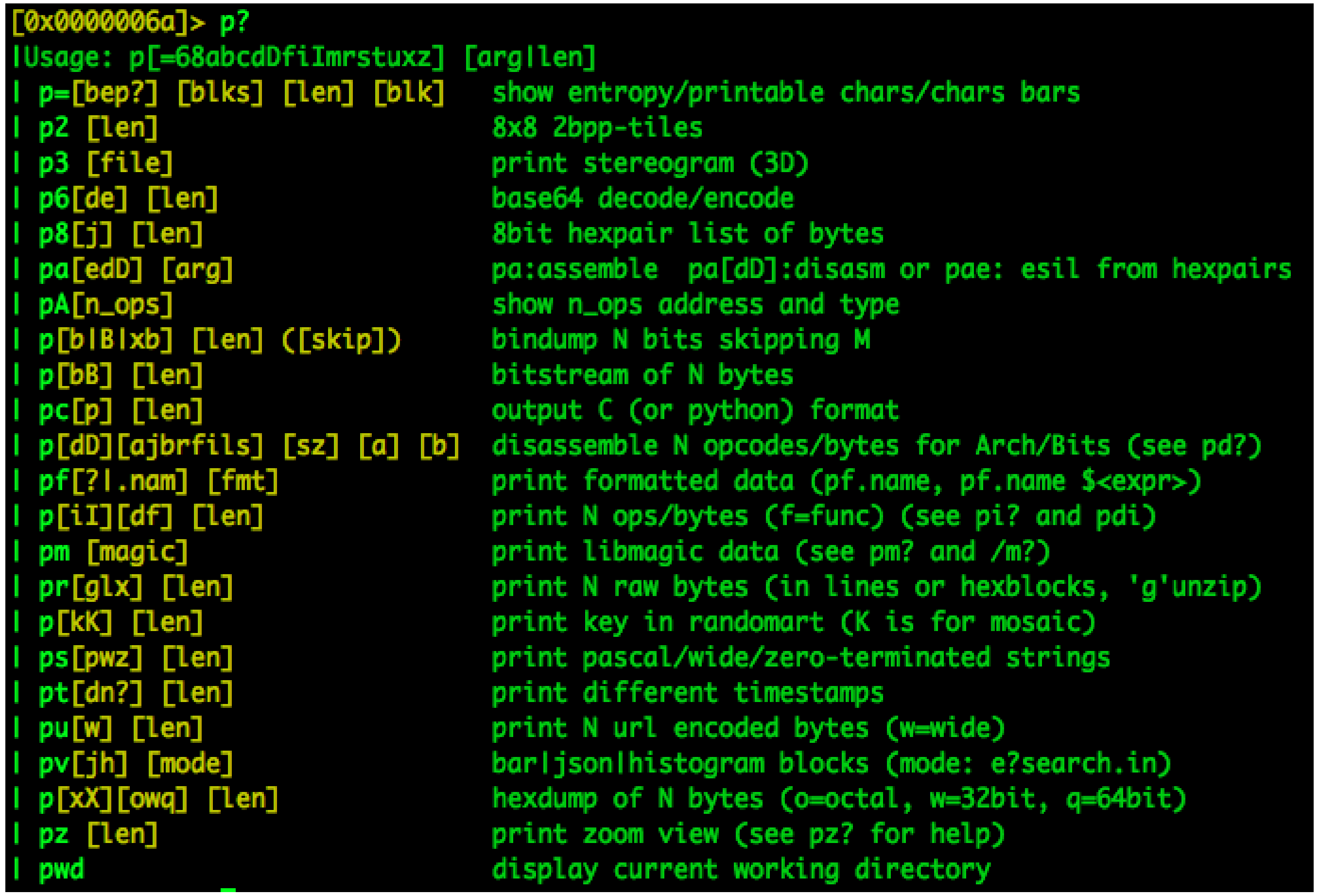


```
[0x00000000]> aaa
[0x00000000]> s 0x6a
[0x0000006a]> pd 35
|          0x0000006a    1124          clr r1
|          0x0000006c    1fbe          out 0x3f, r1
|          0x0000006e    cfef          ser r28
|          0x00000070    d8e0          ldi r29, 0x08
|          0x00000072    debf          out 0x3e, r29
|          0x00000074    cdbf          out 0x3d, r28
|          0x00000076    11e0          ldi r17, 0x01
|          0x00000078    a0e0          ldi r26, 0x00
|          0x0000007a    b1e0          ldi r27, 0x01
|          0x0000007c    ece5          ldi r30, 0x5c
|          0x0000007e    ffe0          ldi r31, 0x0f
|       ┌─< 0x00000080    02c0          rjmp 0x86
|      ┌──> 0x00000082    0590          lpm r0, Z+
|      ||   0x00000084    0d92          st X+, r0
|      |└   ; JMP XREF from 0x00000080 (fcn.00000000)
|      |└─> 0x00000086    ae33          cpi r26, 0x3e
|      |    0x00000088    b107          cpc r27, r17
|      └──< 0x0000008a    d9f7          brne 0x82
|           0x0000008c    21e0          ldi r18, 0x01
|           0x0000008e    aee3          ldi r26, 0x3e
|           0x00000090    b1e0          ldi r27, 0x01
|       ┌─< 0x00000092    01c0          rjmp 0x96
|      ┌──> 0x00000094    1d92          st X+, r1
|      |└   ; JMP XREF from 0x00000092 (fcn.00000000)
|      |└─> 0x00000096    a63e          cpi r26, 0xe6
|      |    0x00000098    b207          cpc r27, r18
|      └──< 0x0000009a    e1f7          brne 0x94
|           0x0000009c    10e0          ldi r17, 0x00
|           0x0000009e    cae6          ldi r28, 0x6a
|           0x000000a0    d0e0          ldi r29, 0x00
|       ┌─< 0x000000a2    04c0          rjmp 0xac
|      ┌──> 0x000000a4    2297          sbiw r28, 0x02
|      ||   0x000000a6    fe01          movw r30, r28
|      ||   0x000000a8    0e94a107      call 0xf42                            ; fcn.00000000()
|      |└   ; JMP XREF from 0x000000a2 (fcn.00000000)
|      |└─> 0x000000ac    c836          cpi r28, 0x68
|      |    0x000000ae    d107          cpc r29, r17
|      └──< 0x000000b0    c9f7          brne 0xa4
```

Radare2. Disassembling

- p?
- pd/pD - dissamble
- pi/pI – print instructions
- Examples:

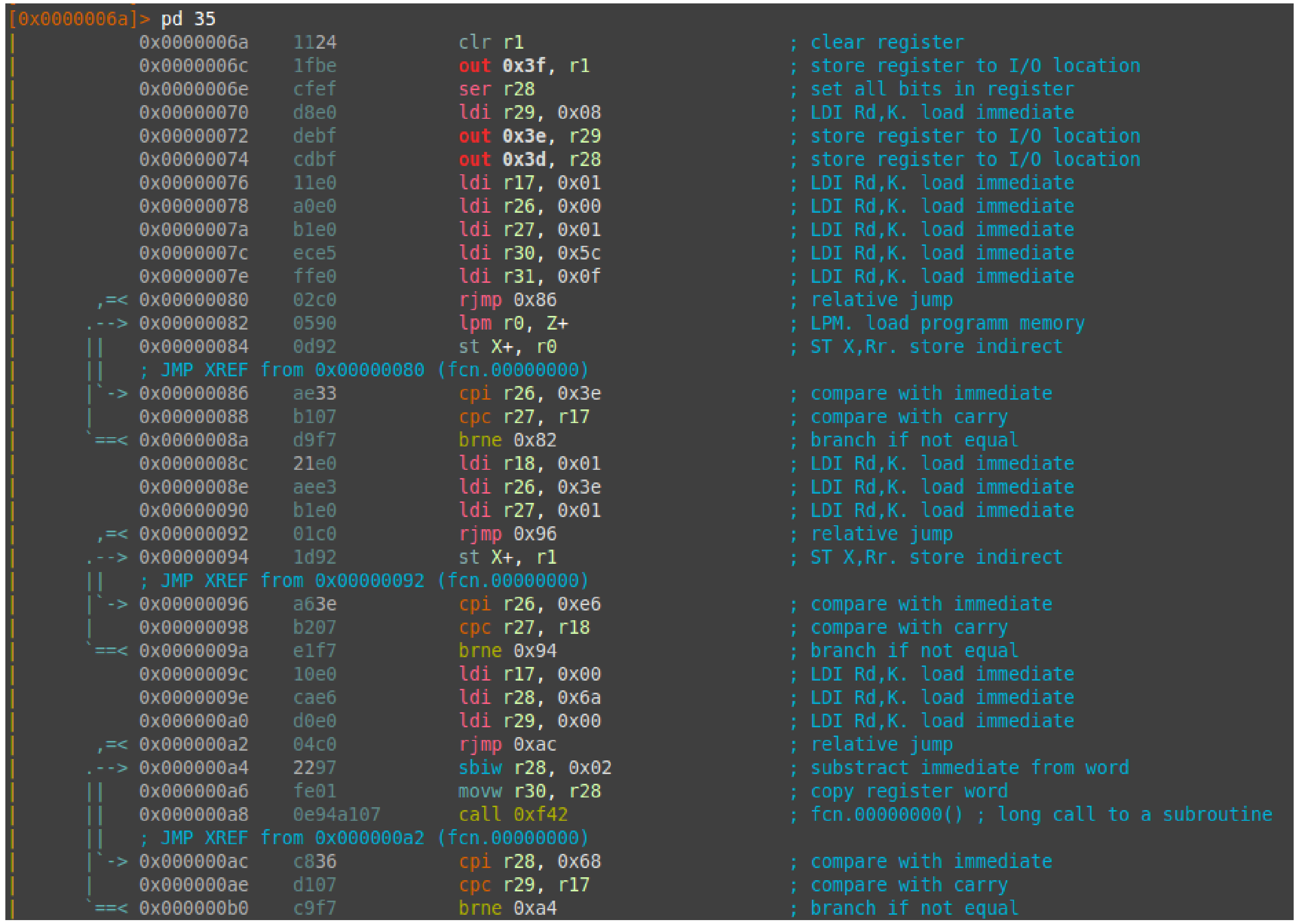
  > pd 35 @ function



```
[0x0000006a]> p?
|Usage: p[=68abcdDfiImrstuxz] [arg|len]
| p=[bep?] [blks] [len] [blk]   show entropy/printable chars/chars bars
| p2 [len]                      8x8 2bpp-tiles
| p3 [file]                     print stereogram (3D)
| p6[de] [len]                  base64 decode/encode
| p8[j] [len]                   8bit hexpair list of bytes
| pa[edD] [arg]                 pa:assemble  pa[dD]:disasm or pae: esil from hexpairs
| pA[n_ops]                     show n_ops address and type
| p[b|B|xb] [len] ([skip])      bindump N bits skipping M
| p[bB] [len]                   bitstream of N bytes
| pc[p] [len]                   output C (or python) format
| p[dD][ajbrfils] [sz] [a] [b]  disassemble N opcodes/bytes for Arch/Bits (see pd?)
| pf[?|.nam] [fmt]              print formatted data (pf.name, pf.name $<expr>)
| p[iI][df] [len]               print N ops/bytes (f=func) (see pi? and pdi)
| pm [magic]                    print libmagic data (see pm? and /m?)
| pr[glx] [len]                 print N raw bytes (in lines or hexblocks, 'g'unzip)
| p[kK] [len]                   print key in randomart (K is for mosaic)
| ps[pwz] [len]                 print pascal/wide/zero-terminated strings
| pt[dn?] [len]                 print different timestamps
| pu[w] [len]                   print N url encoded bytes (w=wide)
| pv[jh] [mode]                 bar|json|histogram blocks (mode: e?search.in)
| p[xX][owq] [len]              hexdump of N bytes (o=octal, w=32bit, q=64bit)
| pz [len]                      print zoom view (see pz? for help)
| pwd                           display current working directory
```

Radare2. Options

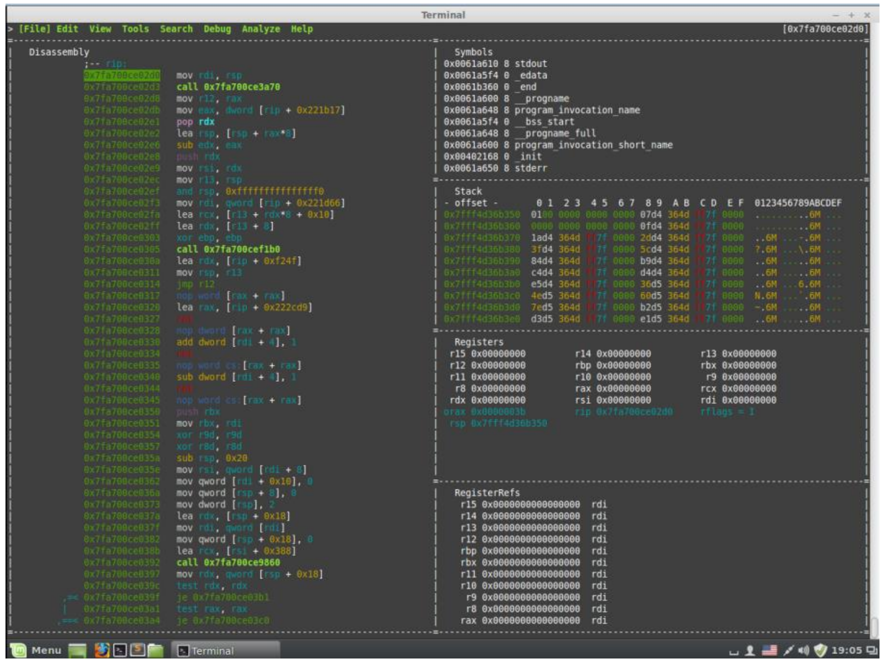
- ~/.radarerc
- e asm.describe=true
- e scr.utf8=true
- e asm.midflags=true
- e asm.emu=true
- eco solarized



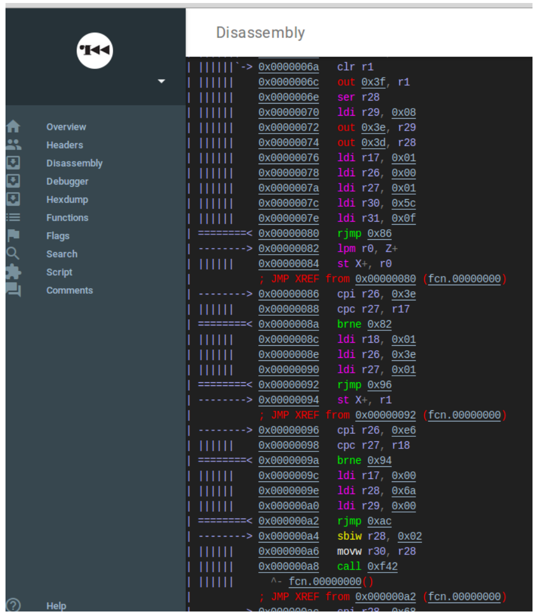
```
[0x0000006a]> pd 35
|          0x0000006a    1124          clr r1                   ; clear register
|          0x0000006c    1fbe          out 0x3f, r1             ; store register to I/O location
|          0x0000006e    cfef          ser r28                  ; set all bits in register
|          0x00000070    d8e0          ldi r29, 0x08            ; LDI Rd,K. load immediate
|          0x00000072    debf          out 0x3e, r29            ; store register to I/O location
|          0x00000074    cdbf          out 0x3d, r28            ; store register to I/O location
|          0x00000076    11e0          ldi r17, 0x01            ; LDI Rd,K. load immediate
|          0x00000078    a0e0          ldi r26, 0x00            ; LDI Rd,K. load immediate
|          0x0000007a    b1e0          ldi r27, 0x01            ; LDI Rd,K. load immediate
|          0x0000007c    ece5          ldi r30, 0x5c            ; LDI Rd,K. load immediate
|          0x0000007e    ffe0          ldi r31, 0x0f            ; LDI Rd,K. load immediate
|      ,=< 0x00000080    02c0          rjmp 0x86                ; relative jump
|     .--> 0x00000082    0590          lpm r0, Z+               ; LPM. load programm memory
|     ||   0x00000084    0d92          st X+, r0                ; ST X,Rr. store indirect
|     ||   ; JMP XREF from 0x00000080 (fcn.00000000)
|     |`-> 0x00000086    ae33          cpi r26, 0x3e            ; compare with immediate
|     |    0x00000088    b107          cpc r27, r17             ; compare with carry
|     `==< 0x0000008a    d9f7          brne 0x82                ; branch if not equal
|          0x0000008c    21e0          ldi r18, 0x01            ; LDI Rd,K. load immediate
|          0x0000008e    aee3          ldi r26, 0x3e            ; LDI Rd,K. load immediate
|          0x00000090    b1e0          ldi r27, 0x01            ; LDI Rd,K. load immediate
|      ,=< 0x00000092    01c0          rjmp 0x96                ; relative jump
|     .--> 0x00000094    1d92          st X+, r1                ; ST X,Rr. store indirect
|     ||   ; JMP XREF from 0x00000092 (fcn.00000000)
|     |`-> 0x00000096    a63e          cpi r26, 0xe6            ; compare with immediate
|     |    0x00000098    b207          cpc r27, r18             ; compare with carry
|     `==< 0x0000009a    e1f7          brne 0x94                ; branch if not equal
|          0x0000009c    10e0          ldi r17, 0x00            ; LDI Rd,K. load immediate
|          0x0000009e    cae6          ldi r28, 0x6a            ; LDI Rd,K. load immediate
|          0x000000a0    d0e0          ldi r29, 0x00            ; LDI Rd,K. load immediate
|      ,=< 0x000000a2    04c0          rjmp 0xac                ; relative jump
|     .--> 0x000000a4    2297          sbiw r28, 0x02           ; substract immediate from word
|     ||   0x000000a6    fe01          movw r30, r28            ; copy register word
|     ||   0x000000a8    0e94a107      call 0xf42               ; fcn.00000000() ; long call to a subroutine
|     ||   ; JMP XREF from 0x000000a2 (fcn.00000000)
|     |`-> 0x000000ac    c836          cpi r28, 0x68            ; compare with immediate
|     |    0x000000ae    d107          cpc r29, r17             ; compare with carry
|     `==< 0x000000b0    c9f7          brne 0xa4                ; branch if not equal
```

Radare2. Interfaces

- ASCII – VV
- Visual panels – V! (vim like controls)
- Web-server – r2 -c=H file
- Bokken

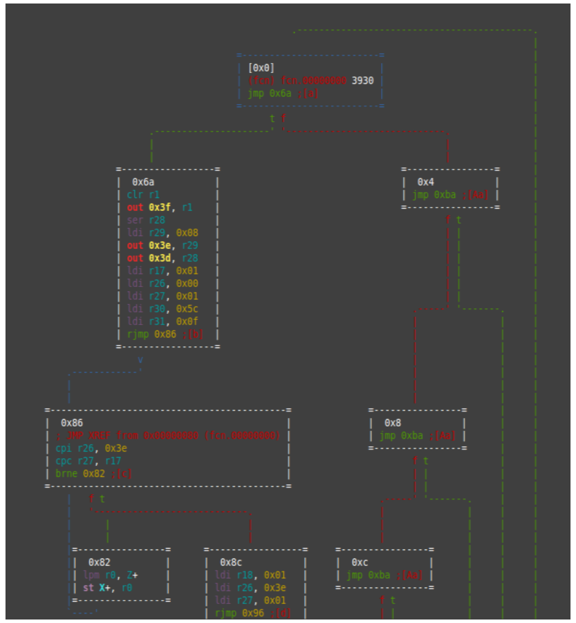


```
Terminal
> [File] Edit  View  Tools  Search  Debug  Analyze  Help                                [0x7fa700ce02d0]
 Disassembly                                                  Symbols
          ;-- rip:                                            0x0061a610 8 stdout
          0x7fa700ce02d0   mov rdi, rsp                       0x0061a5f4 0 _edata
          0x7fa700ce02d3   call 0x7fa700ce3a70                0x0061b360 0 _end
          0x7fa700ce02d8   mov r12, rax                       0x0061a600 8 __progname
          0x7fa700ce02db   mov eax, dword [rip + 0x221b17]    0x0061a648 8 program_invocation_name
          0x7fa700ce02e1   pop rdx                            0x0061a5f4 0 __bss_start
          0x7fa700ce02e2   lea rsp, [rsp + rax*8]             0x0061a648 8 __progname_full
          0x7fa700ce02e6   sub edx, eax                       0x0061a600 8 program_invocation_short_name
          0x7fa700ce02e8   push rdx                           0x00402168 0 _init
          0x7fa700ce02e9   mov rsi, rdx                       0x0061a650 8 stderr
          0x7fa700ce02ec   mov r13, rsp
          0x7fa700ce02ef   and rsp, 0xfffffffffffffff0        Stack
          0x7fa700ce02f3   mov rdi, qword [rip + 0x221d66]    - offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
          0x7fa700ce02fa   lea rcx, [r13 + rdx*8 + 0x10]      0x7fff4d36b350  0100 0000 0000 0000 07d4 364d ff7f 0000  ..........6M....
          0x7fa700ce02ff   lea rdx, [r13 + 8]                 0x7fff4d36b360  0000 0000 0000 0000 0fd4 364d ff7f 0000  ..........6M....
          0x7fa700ce0303   xor ebp, ebp                       0x7fff4d36b370  1ad4 364d ff7f 0000 2dd4 364d ff7f 0000  ..6M....-.6M....
          0x7fa700ce0305   call 0x7fa700cef1b0                0x7fff4d36b380  3fd4 364d ff7f 0000 5cd4 364d ff7f 0000  ?.6M....\.6M....
          0x7fa700ce030a   lea rdx, [rip + 0xf24f]            0x7fff4d36b390  84d4 364d ff7f 0000 b9d4 364d ff7f 0000  ..6M......6M....
          0x7fa700ce0311   mov rsp, r13                       0x7fff4d36b3a0  c4d4 364d ff7f 0000 d4d4 364d ff7f 0000  ..6M......6M....
          0x7fa700ce0314   jmp r12                            0x7fff4d36b3b0  e5d4 364d ff7f 0000 36d5 364d ff7f 0000  ..6M....6.6M....
          0x7fa700ce0317   nop word [rax + rax]               0x7fff4d36b3c0  4ed5 364d ff7f 0000 60d5 364d ff7f 0000  N.6M....`.6M....
          0x7fa700ce0320   lea rax, [rip + 0x222cd9]          0x7fff4d36b3d0  7ed5 364d ff7f 0000 b2d5 364d ff7f 0000  ~.6M......6M....
          0x7fa700ce0327   ret                                0x7fff4d36b3e0  d3d5 364d ff7f 0000 e1d5 364d ff7f 0000  ..6M......6M....
          0x7fa700ce0328   nop dword [rax + rax]
          0x7fa700ce0330   add dword [rdi + 4], 1             Registers
          0x7fa700ce0334   ret                                r15 0x00000000       r14 0x00000000       r13 0x00000000
          0x7fa700ce0335   nop word cs:[rax + rax]            r12 0x00000000       rbp 0x00000000       rbx 0x00000000
          0x7fa700ce0340   sub dword [rdi + 4], 1             r11 0x00000000       r10 0x00000000        r9 0x00000000
          0x7fa700ce0344   ret                                 r8 0x00000000       rax 0x00000000       rcx 0x00000000
          0x7fa700ce0345   nop word cs:[rax + rax]            rdx 0x00000000       rsi 0x00000000       rdi 0x00000000
          0x7fa700ce0350   push rbx                          orax 0x0000003b       rip 0x7fa700ce02d0   rflags = I
          0x7fa700ce0351   mov rbx, rdi                       rsp 0x7fff4d36b350
          0x7fa700ce0354   xor r9d, r9d
          0x7fa700ce0357   xor r8d, r8d
          0x7fa700ce035a   sub rsp, 0x20
          0x7fa700ce035e   mov rsi, qword [rdi + 8]           RegisterRefs
          0x7fa700ce0362   mov qword [rdi + 0x10], 0           r15 0x0000000000000000  rdi
          0x7fa700ce036a   mov qword [rsp + 8], 0              r14 0x0000000000000000  rdi
          0x7fa700ce0373   mov dword [rsp], 2                  r13 0x0000000000000000  rdi
          0x7fa700ce037a   lea rdx, [rsp + 0x18]               r12 0x0000000000000000  rdi
          0x7fa700ce037f   mov rdi, qword [rdi]                rbp 0x0000000000000000  rdi
          0x7fa700ce0382   mov qword [rsp + 0x18], 0           rbx 0x0000000000000000  rdi
          0x7fa700ce038b   lea rcx, [rsi + 0x388]              r11 0x0000000000000000  rdi
          0x7fa700ce0392   call 0x7fa700ce9860                 r10 0x0000000000000000  rdi
          0x7fa700ce0397   mov rdx, qword [rsp + 0x18]          r9 0x0000000000000000  rdi
          0x7fa700ce039c   test rdx, rdx                        r8 0x0000000000000000  rdi
      ,=< 0x7fa700ce039f   je 0x7fa700ce03b1                   rax 0x0000000000000000  rdi
      |   0x7fa700ce03a1   test rax, rax
    ,==< 0x7fa700ce03a4   je 0x7fa700ce03c0
Menu   Terminal                                                                                   19:05
```



Disassembly

```
| ||||||`-> 0x0000006a   clr r1
| ||||||    0x0000006c   out 0x3f, r1
| ||||||    0x0000006e   ser r28
| ||||||    0x00000070   ldi r29, 0x08
| ||||||    0x00000072   out 0x3e, r29
| ||||||    0x00000074   out 0x3d, r28
| ||||||    0x00000076   ldi r17, 0x01
| ||||||    0x00000078   ldi r26, 0x00
| ||||||    0x0000007a   ldi r27, 0x01
| ||||||    0x0000007c   ldi r30, 0x5c
| ||||||    0x0000007e   ldi r31, 0x0f
| ========< 0x00000080   rjmp 0x86
| --------> 0x00000082   lpm r0, Z+
| ||||||    0x00000084   st X+, r0
|           ; JMP XREF from 0x00000080 (fcn.00000000)
| --------> 0x00000086   cpi r26, 0x3e
| ||||||    0x00000088   cpc r27, r17
| ========< 0x0000008a   brne 0x82
| ||||||    0x0000008c   ldi r18, 0x01
| ||||||    0x0000008e   ldi r26, 0x3e
| ||||||    0x00000090   ldi r27, 0x01
| ========< 0x00000092   rjmp 0x96
| --------> 0x00000094   st X+, r1
|           ; JMP XREF from 0x00000092 (fcn.00000000)
| --------> 0x00000096   cpi r26, 0xe6
| ||||||    0x00000098   cpc r27, r18
| ========< 0x0000009a   brne 0x94
| ||||||    0x0000009c   ldi r17, 0x00
| ||||||    0x0000009e   ldi r28, 0x6a
| ||||||    0x000000a0   ldi r29, 0x00
| ========< 0x000000a2   rjmp 0xac
| --------> 0x000000a4   sbiw r28, 0x02
| ||||||    0x000000a6   movw r30, r28
| ||||||    0x000000a8   call 0xf42
| ||||||      ^- fcn.00000000()
|           ; JMP XREF from 0x000000a2 (fcn.00000000)
```

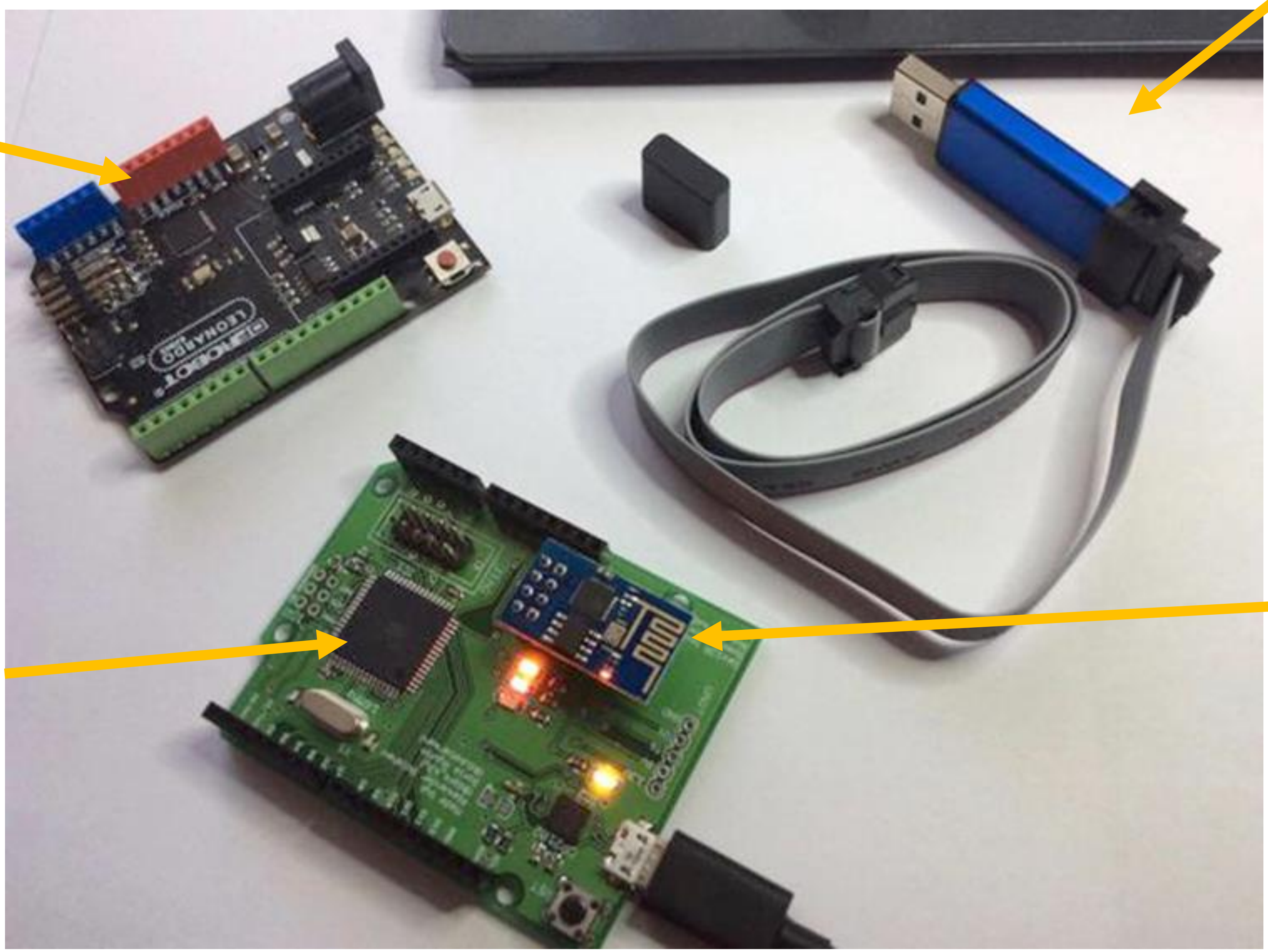


```
[0x0]
(fcn) fcn.00000000 3930
jmp 0x6a ;[a]

0x6a
clr r1
out 0x3f, r1
ser r28
ldi r29, 0x08
out 0x3e, r29
out 0x3d, r28
ldi r17, 0x01
ldi r26, 0x00
ldi r27, 0x01
ldi r30, 0x5c
ldi r31, 0x0f
rjmp 0x86 ;[b]

0x4
jmp 0xba ;[Aa]

0x86
; JMP XREF from 0x00000080 (fcn.00000000)
cpi r26, 0x3e
cpc r27, r17
brne 0x82 ;[c]

0x8
jmp 0xba ;[Aa]

0x82
lpm r0, Z+
st X+, r0

0x8c
ldi r18, 0x01
ldi r26, 0x3e
ldi r27, 0x01
rjmp 0x96 ;[d]

0xc
jmp 0xba ;[Aa]
```

Training kit content

AVR JTAG mkI

Arduino (not included)

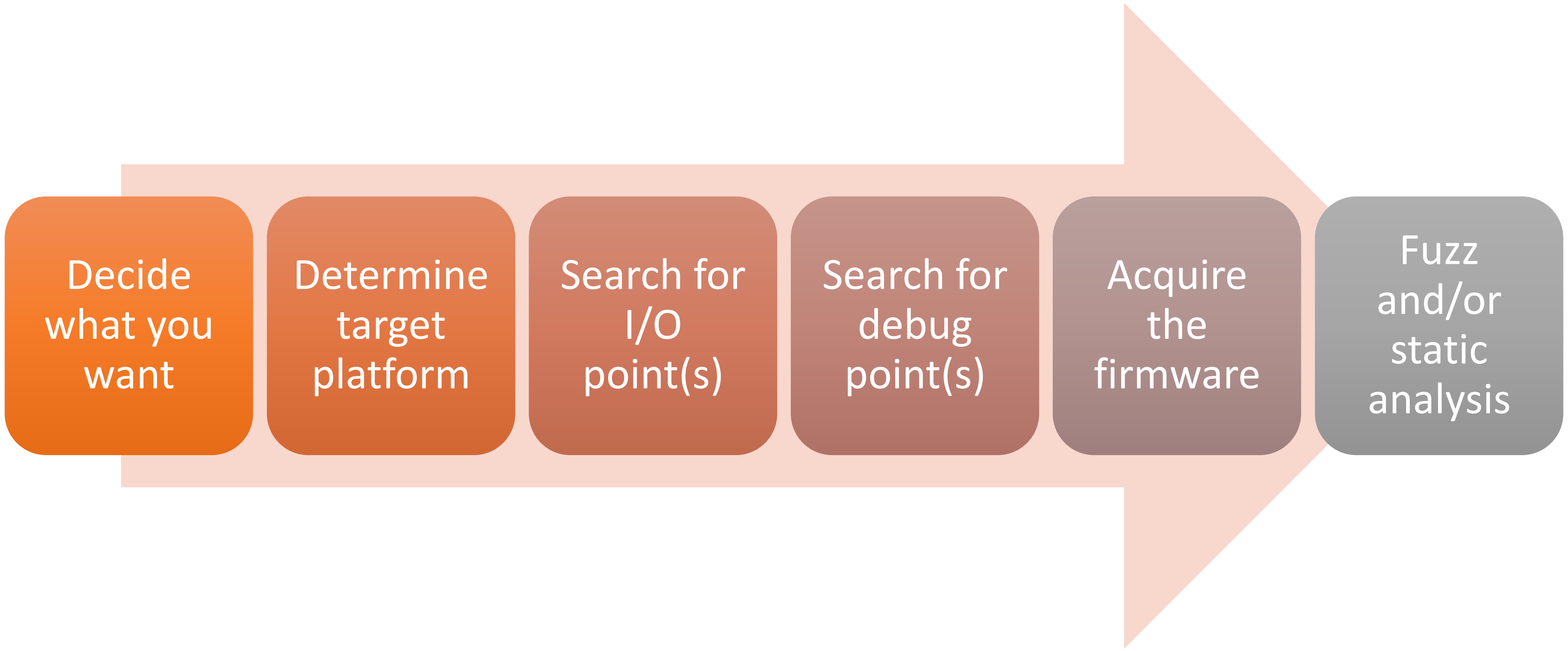


ESP8266 “WiFi to serial”

Atmega128 custom devboard

Part 2: Pre-exploitation

You have a device. First steps?


Decide what you want
Determine target platform
Search for I/O point(s)
Search for debug point(s)
Acquire the firmware
Fuzz and/or static analysis

Let’s start with a REAL example

- Let’s use training kit board as an example.
- Imagine that you know nothing about it
- We will go through all steps, one by one

What we want?

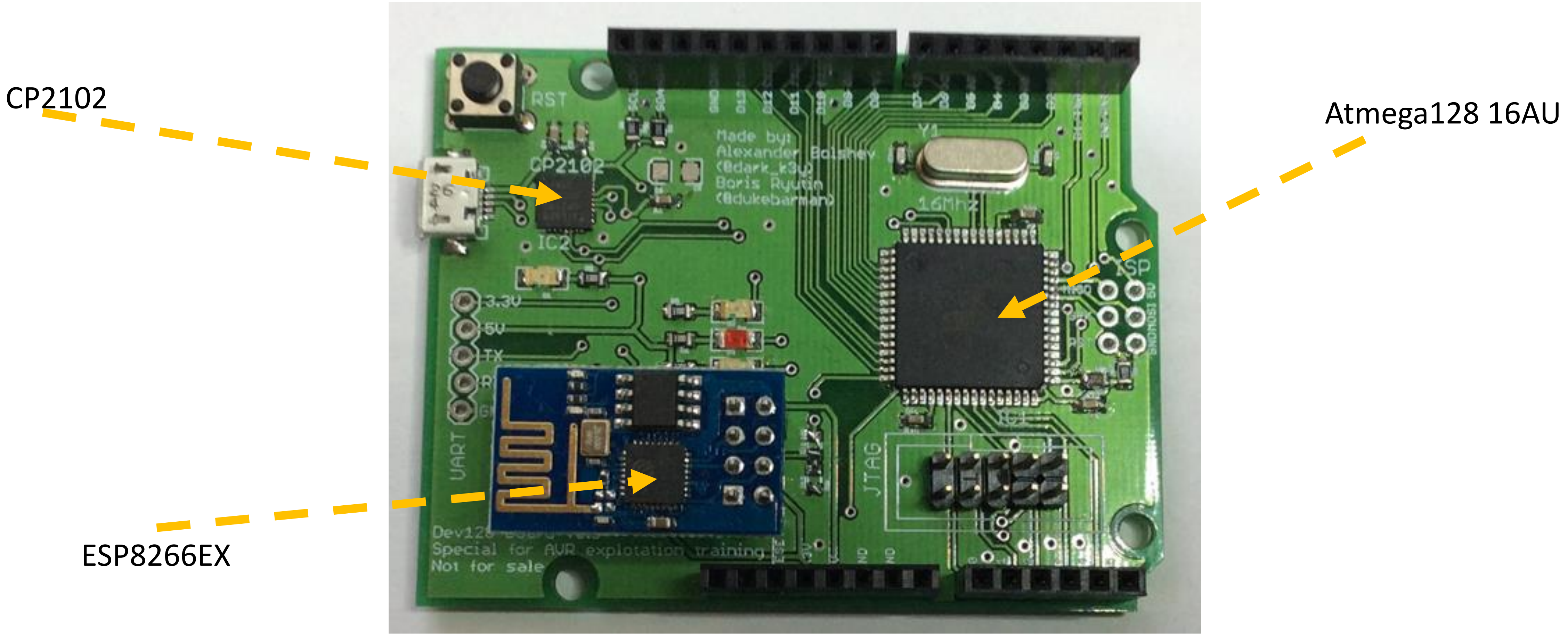
At first, decide what you want:

- Abuse functionality
- Read something from EEPROM/Flash/SRAM
- Stay persistant

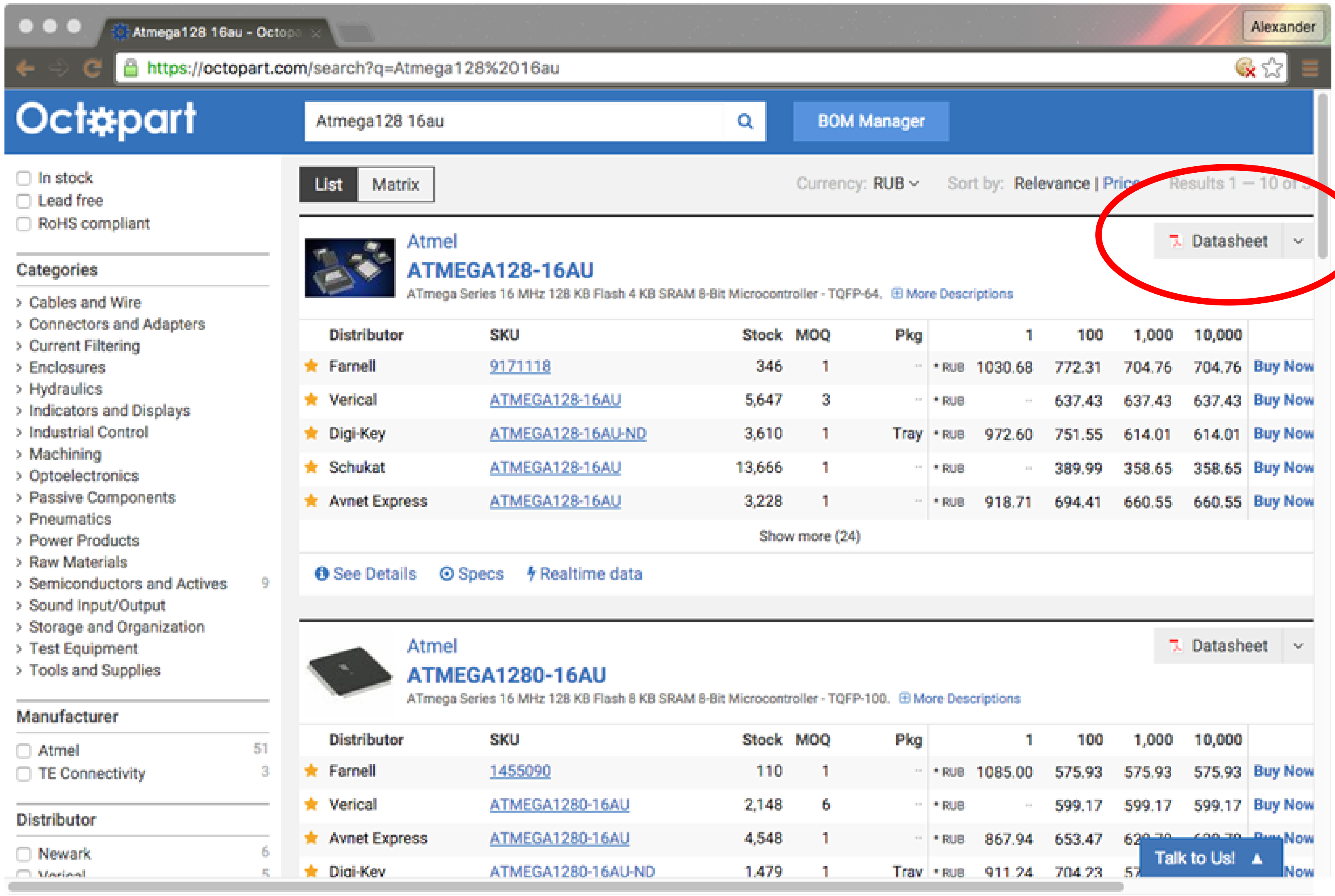
Complexity

Determine target platform

- Look at the board and search for all ICs…



Digikey/Octopart/Google...



Atmega128 16au - Octopa

Alexander

https://octopart.com/search?q=Atmega128%2016au

Octopart

Atmega128 16au

BOM Manager

- In stock
- Lead free
- RoHS compliant

Categories

- Cables and Wire
- Connectors and Adapters
- Current Filtering
- Enclosures
- Hydraulics
- Indicators and Displays
- Industrial Control
- Machining
- Optoelectronics
- Passive Components
- Pneumatics
- Power Products
- Raw Materials
- Semiconductors and Actives 9
- Sound Input/Output
- Storage and Organization
- Test Equipment
- Tools and Supplies

Manufacturer

- Atmel 51
- TE Connectivity 3

Distributor

- Newark 6
- Verical 5

List | Matrix

Currency: RUB Sort by: Relevance | Price Results 1 — 10 of

Atmel
ATMEGA128-16AU
ATmega Series 16 MHz 128 KB Flash 4 KB SRAM 8-Bit Microcontroller - TQFP-64. More Descriptions

Datasheet

Distributor	SKU	Stock	MOQ	Pkg		1	100	1,000	10,000	
Farnell	9171118	346	1	--	RUB	1030.68	772.31	704.76	704.76	Buy Now
Verical	ATMEGA128-16AU	5,647	3	--	RUB	--	637.43	637.43	637.43	Buy Now
Digi-Key	ATMEGA128-16AU-ND	3,610	1	Tray	RUB	972.60	751.55	614.01	614.01	Buy Now
Schukat	ATMEGA128-16AU	13,666	1	--	RUB	--	389.99	358.65	358.65	Buy Now
Avnet Express	ATMEGA128-16AU	3,228	1	--	RUB	918.71	694.41	660.55	660.55	Buy Now

Show more (24)

See Details Specs Realtime data

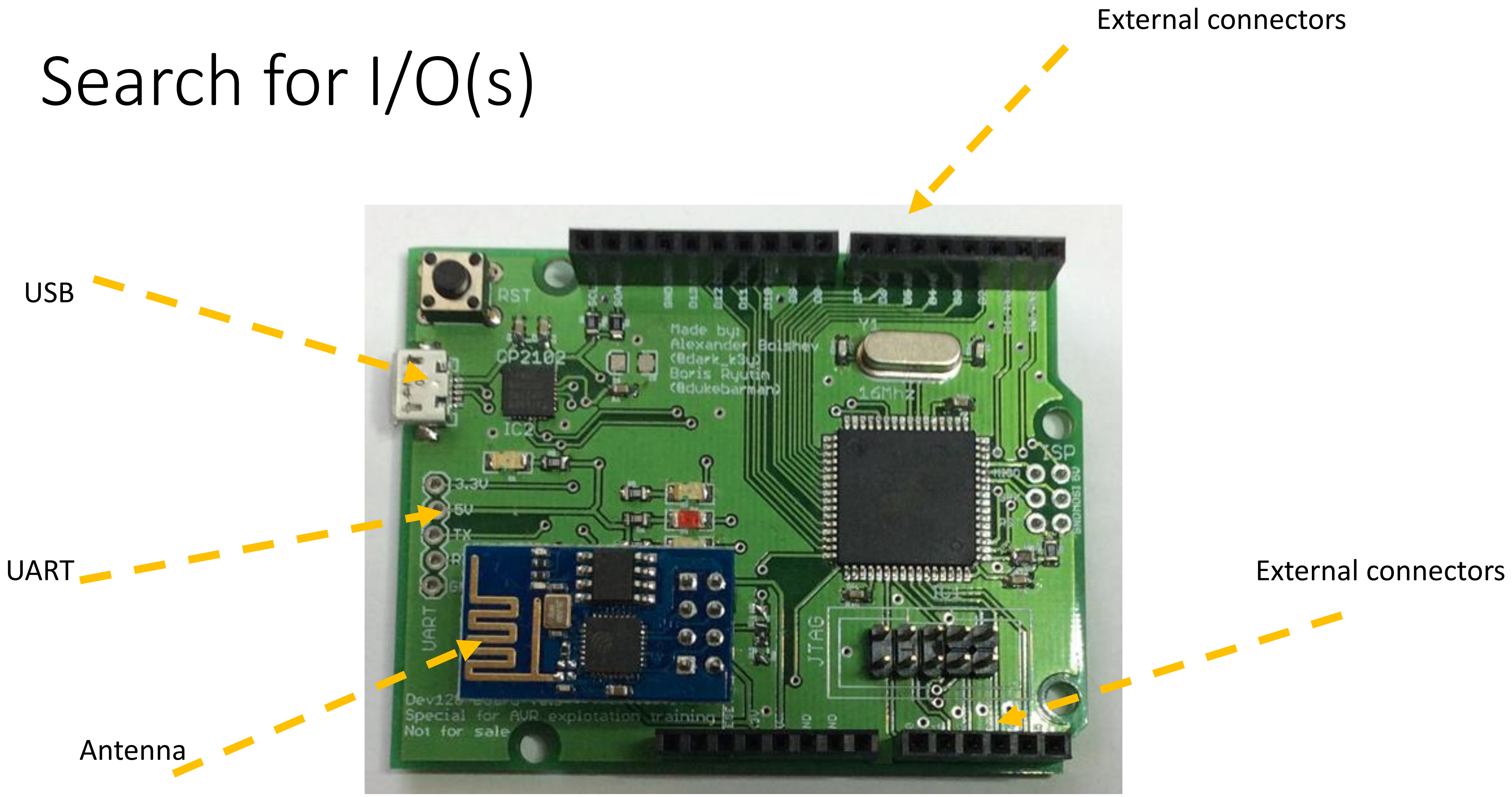
Atmel
ATMEGA1280-16AU
ATmega Series 16 MHz 128 KB Flash 8 KB SRAM 8-Bit Microcontroller - TQFP-100. More Descriptions

Datasheet

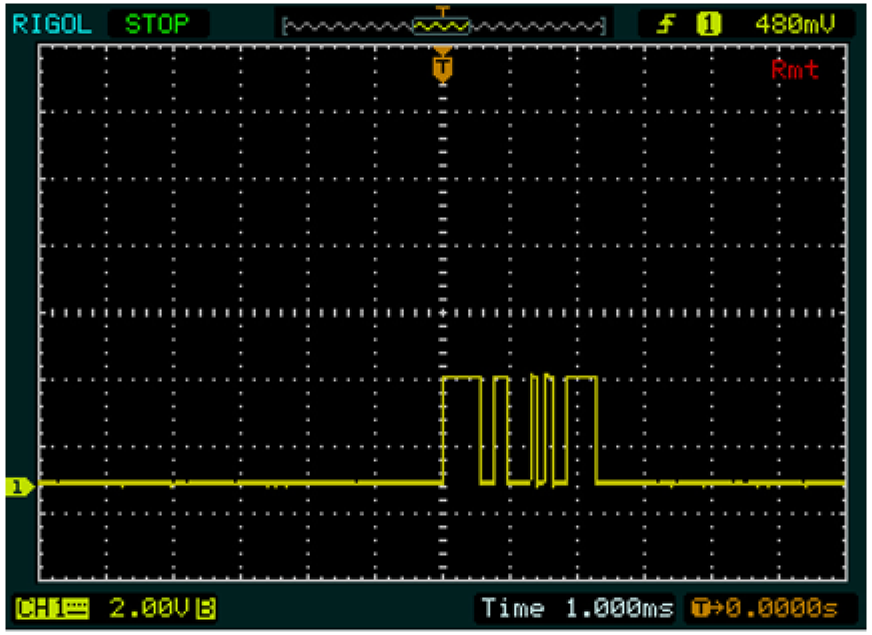
Distributor	SKU	Stock	MOQ	Pkg		1	100	1,000	10,000	
Farnell	1455090	110	1	--	RUB	1085.00	575.93	575.93	575.93	Buy Now
Verical	ATMEGA1280-16AU	2,148	6	--	RUB	--	599.17	599.17	599.17	Buy Now
Avnet Express	ATMEGA1280-16AU	4,548	1	--	RUB	867.94	653.47	62		Now
Digi-Key	ATMEGA1280-16AU-ND	1,479	1	Tray	RUB	911.24	704.23	57		Now

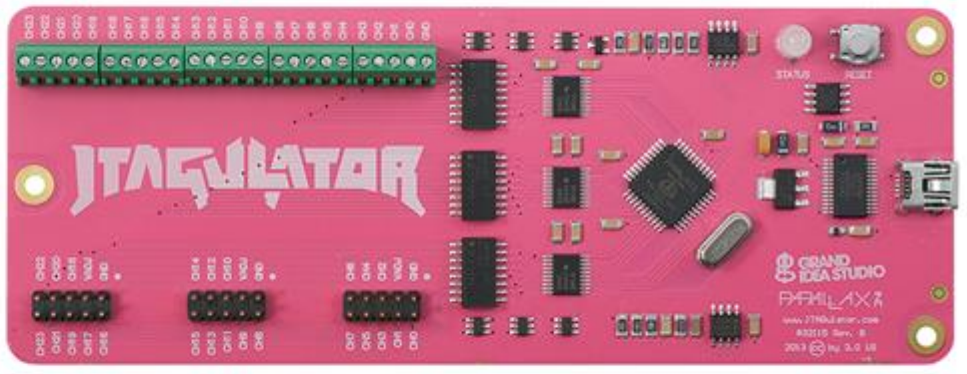
Talk to Us!

Search for I/O(s)

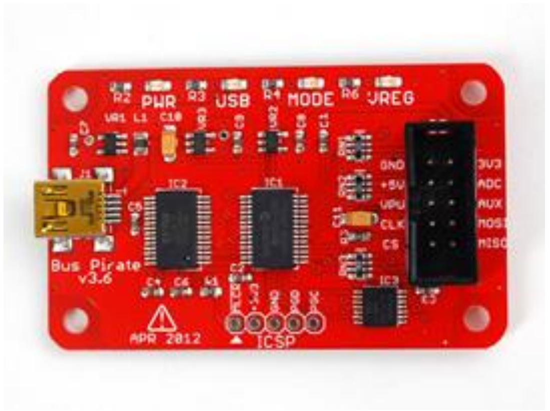


Search for I/O(s): tools





Jtagulator

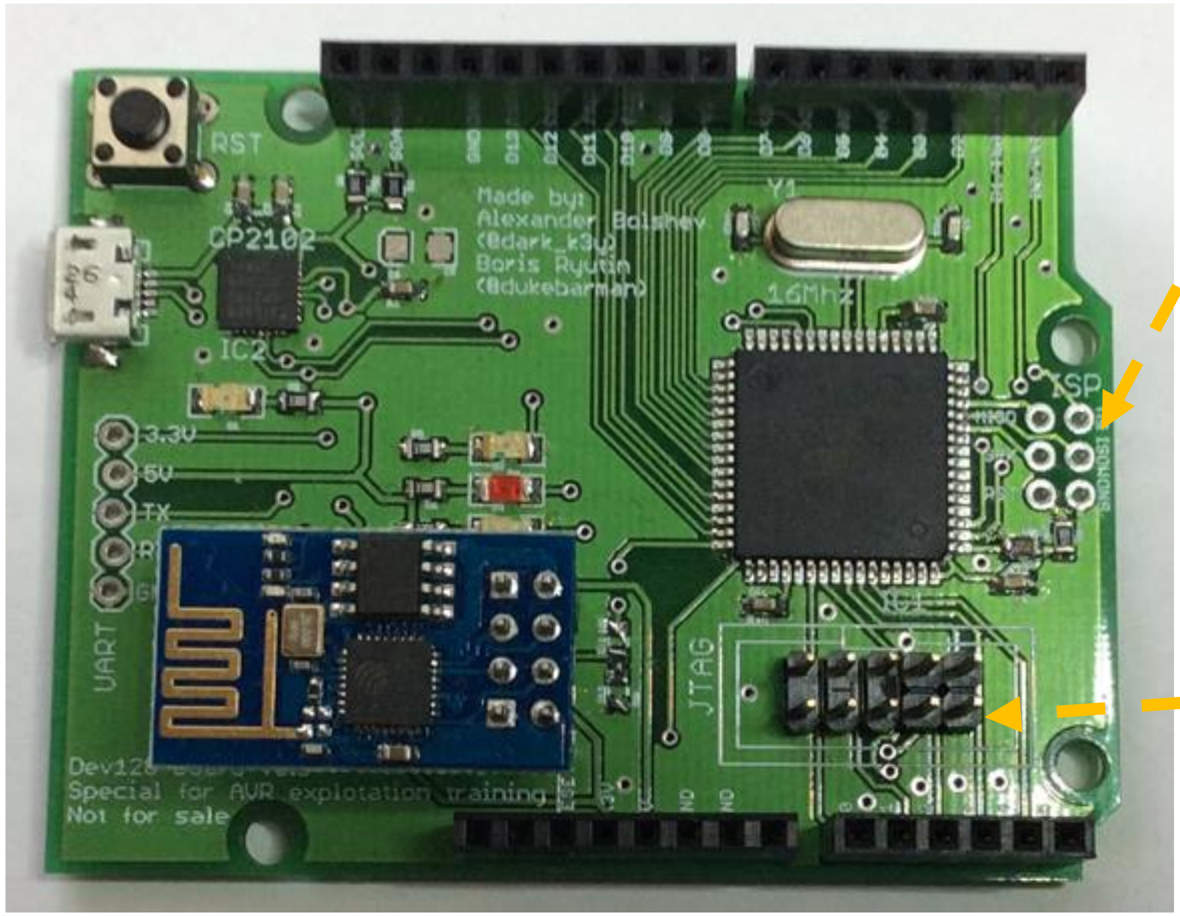


Bus pirate

Saleae logic analyzer

Arduino

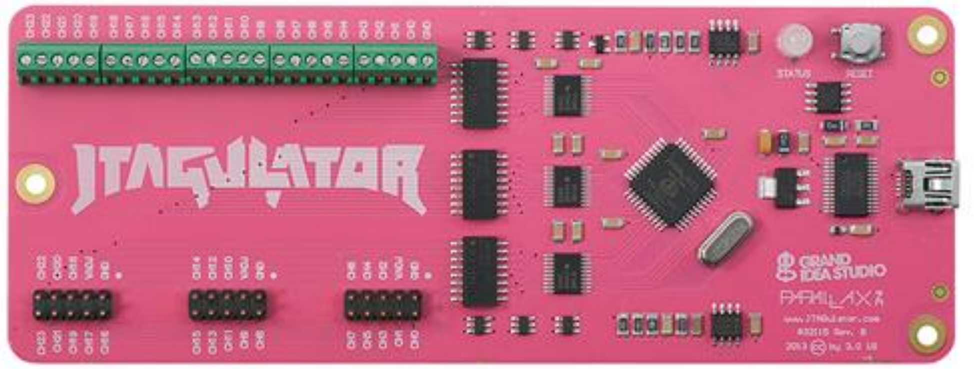
Search for debug interface(s)



ISP

JTAG

Search for debug interface(s): tools



Jtagulator

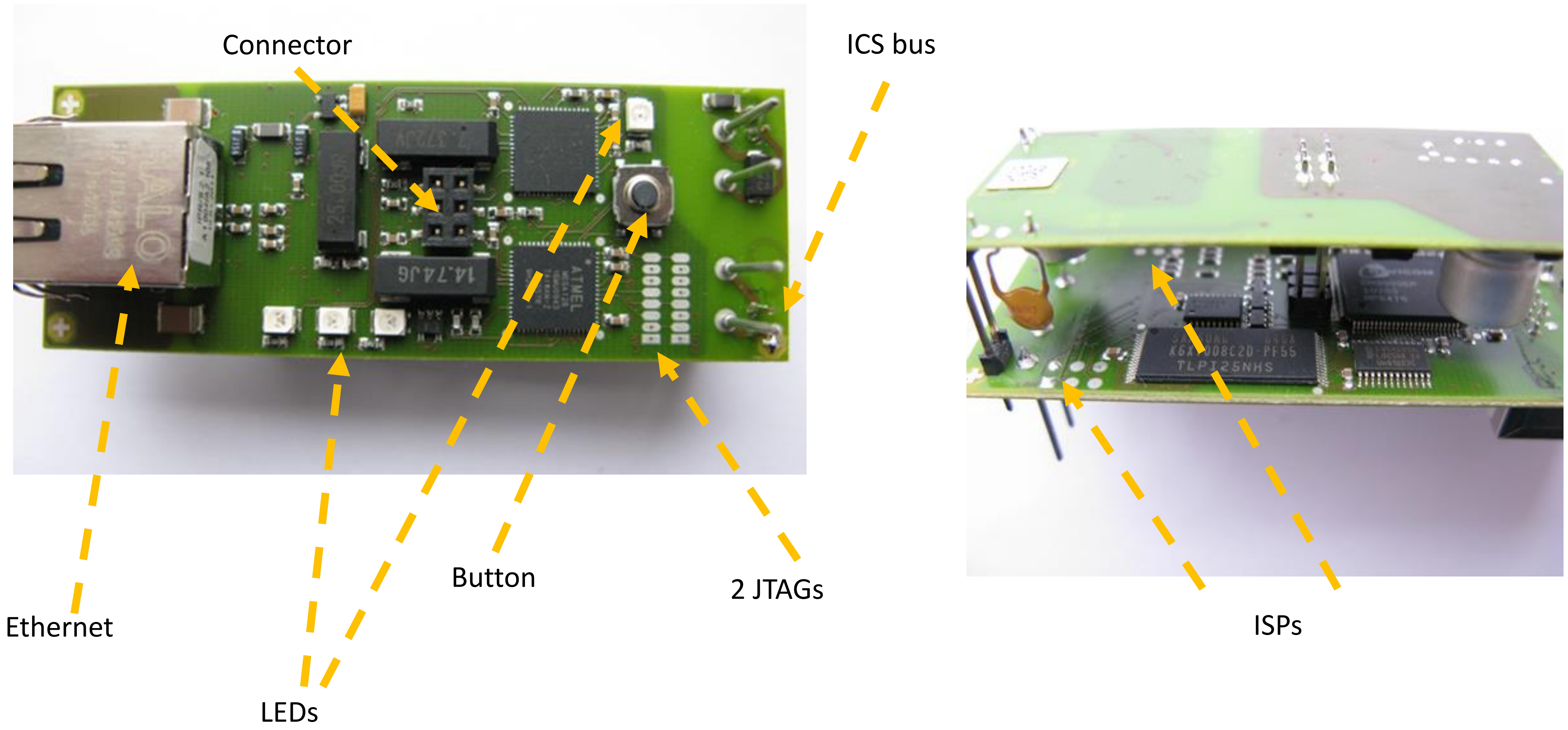
Or cheaper

Arduino + JTAGEnum

DEMO

JTAGEnum against Atmega128 demoboard

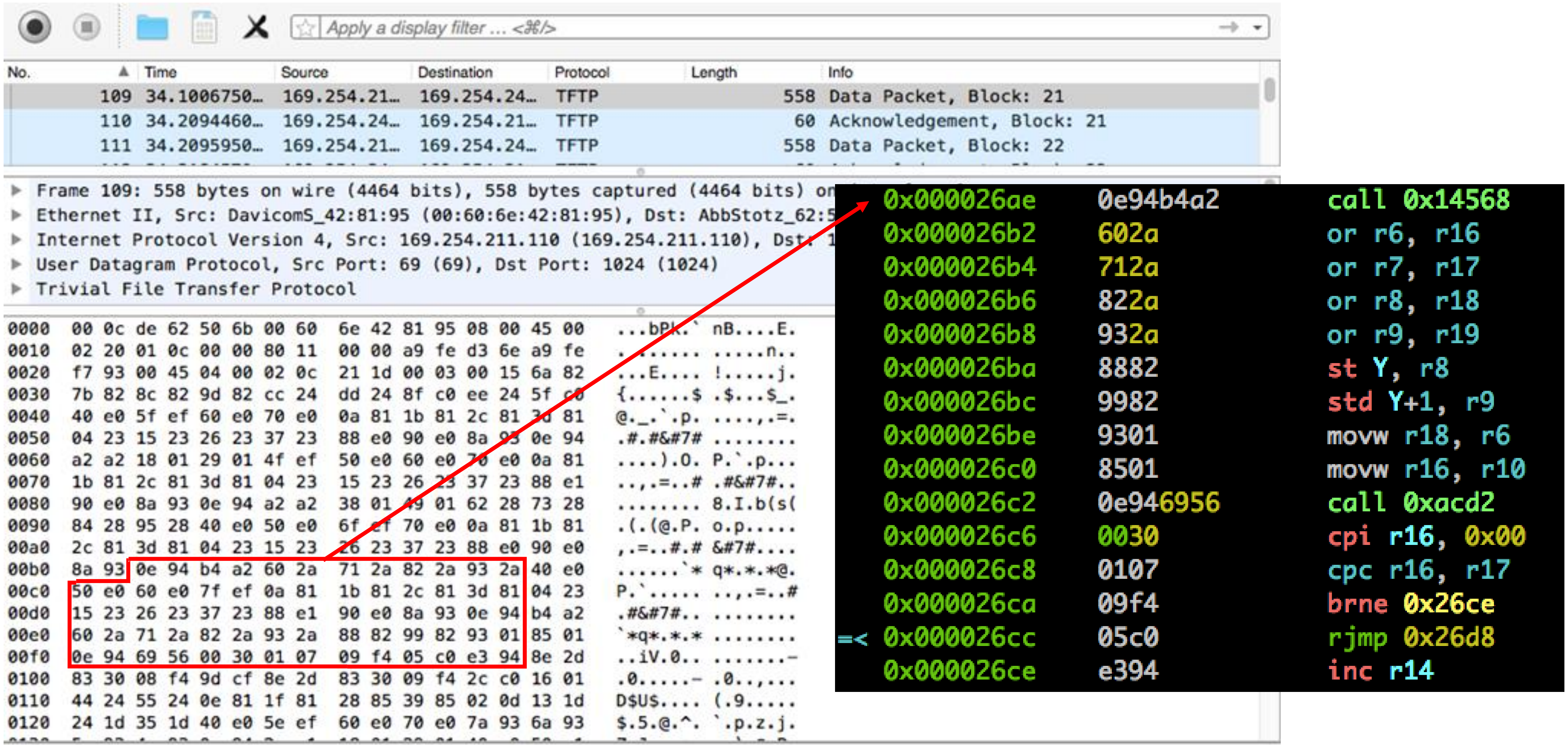
Search for debug & I/O: real device



Acquire the firmware

- From vendor web-site ☺
- Sniffing the update process
- From device

Acquire the firmware: sniff it!



```
No.    Time          Source          Destination     Protocol  Length  Info
  109  34.1006750…   169.254.21…     169.254.24…     TFTP         558  Data Packet, Block: 21
  110  34.2094460…   169.254.24…     169.254.21…     TFTP          60  Acknowledgement, Block: 21
  111  34.2095950…   169.254.21…     169.254.24…     TFTP         558  Data Packet, Block: 22
```

```
▶ Frame 109: 558 bytes on wire (4464 bits), 558 bytes captured (4464 bits) on
▶ Ethernet II, Src: DavicomS_42:81:95 (00:60:6e:42:81:95), Dst: AbbStotz_62:5
▶ Internet Protocol Version 4, Src: 169.254.211.110 (169.254.211.110), Dst: 1
▶ User Datagram Protocol, Src Port: 69 (69), Dst Port: 1024 (1024)
▶ Trivial File Transfer Protocol
```

```
0000  00 0c de 62 50 6b 00 60  6e 42 81 95 08 00 45 00   ...bPk.` nB....E.
0010  02 20 01 0c 00 00 80 11  00 00 a9 fe d3 6e a9 fe   ........ .....n..
0020  f7 93 00 45 04 00 02 0c  21 1d 00 03 00 15 6a 82   ...E.... !.....j.
0030  7b 82 8c 82 9d 82 cc 24  dd 24 8f c0 ee 24 5f c0   {......$ .$...$_.
0040  40 e0 5f ef 60 e0 70 e0  0a 81 1b 81 2c 81 3d 81   @._.`.p. ....,.=.
0050  04 23 15 23 26 23 37 23  88 e0 90 e0 8a 93 0e 94   .#.#&#7# ........
0060  a2 a2 18 01 29 01 4f ef  50 e0 60 e0 70 e0 0a 81   ....).O. P.`.p...
0070  1b 81 2c 81 3d 81 04 23  15 23 26 23 37 23 88 e1   ..,.=..# .#&#7#..
0080  90 e0 8a 93 0e 94 a2 a2  38 01 49 01 62 28 73 28   ........ 8.I.b(s(
0090  84 28 95 28 40 e0 50 e0  6f ef 70 e0 0a 81 1b 81   .(.(@.P. o.p.....
00a0  2c 81 3d 81 04 23 15 23  26 23 37 23 88 e0 90 e0   ,.=..#.# &#7#....
00b0  8a 93 0e 94 b4 a2 60 2a  71 2a 82 2a 93 2a 40 e0   ......`* q*.*.*@.
00c0  50 e0 60 e0 7f ef 0a 81  1b 81 2c 81 3d 81 04 23   P.`..... ..,.=..#
00d0  15 23 26 23 37 23 88 e1  90 e0 8a 93 0e 94 b4 a2   .#&#7#.. ........
00e0  60 2a 71 2a 82 2a 93 2a  88 82 99 82 93 01 85 01   `*q*.*.* ........
00f0  0e 94 69 56 00 30 01 07  09 f4 05 c0 e3 94 8e 2d   ..iV.0.. .......-
0100  83 30 08 f4 9d cf 8e 2d  83 30 09 f4 2c c0 16 01   .0.....- .0..,...
0110  44 24 55 24 0e 81 1f 81  28 85 39 85 02 0d 13 1d   D$U$.... (.9.....
0120  24 1d 35 1d 40 e0 5e ef  60 e0 70 e0 7a 93 6a 93   $.5.@.^. `.p.z.j.
```

```
   0x000026ae   0e94b4a2   call 0x14568
   0x000026b2   602a       or r6, r16
   0x000026b4   712a       or r7, r17
   0x000026b6   822a       or r8, r18
   0x000026b8   932a       or r9, r19
   0x000026ba   8882       st Y, r8
   0x000026bc   9982       std Y+1, r9
   0x000026be   9301       movw r18, r6
   0x000026c0   8501       movw r16, r10
   0x000026c2   0e946956   call 0xacd2
   0x000026c6   0030       cpi r16, 0x00
   0x000026c8   0107       cpc r16, r17
   0x000026ca   09f4       brne 0x26ce
=< 0x000026cc   05c0       rjmp 0x26d8
   0x000026ce   e394       inc r14
```

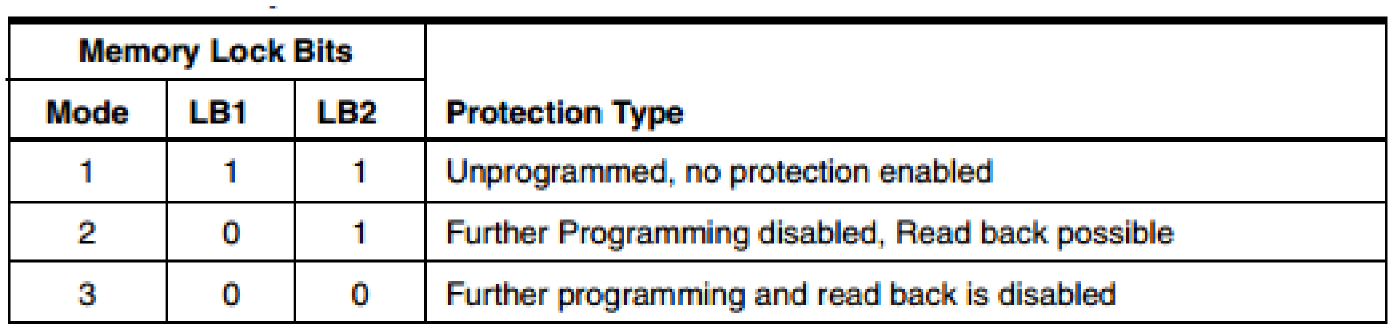
Acquire the firmware: JTAG or ISP

- Use JTAG or ISP programmer to connect to the board debug ports
- Use:
  - Atmel Studio
  - AVRDude
  - Programmer-specific software to read flash

```
$ avrdude -p m128 -c jtagmkI –P /dev/ttyUSB0 \
      -U flash:r:”/home/avr/flash.bin":r
```

Acquire the firmware: lock bits

- AVR has lock bits that protects device from extracting flash



Memory Lock Bits			
Mode	LB1	LB2	Protection Type
1	1	1	Unprogrammed, no protection enabled
2	0	1	Further Programming disabled, Read back possible
3	0	0	Further programming and read back is disabled

- Removing this lockbits will erase entire device
- If you have them set, you’re not lucky, try to get firmware from other sources
- However, if you have lock bits set, but JTAG is **enabled** you could try partial restoration of firmware with avarice –capture (rare case)

Exercise 2.0: Acquire!

EXAMPLE

Real hardware

- Read fuses and lock bits using `avarice -r`
- Acquire firmware using `avrdude`

Firmware reversing: formats

- Raw binary format
- ELF format for AVRs
- Intel HEX format (often used by programmers)

- Could be easily converted between with avr-objcopy, e.g.:

```
avr-objcopy -R .eeprom -O ihex test.elf “test.hex"
```

Ex 2.1: Hello! RE

EXAMPLE

Real hardware & Simulator

```
cd /home/radare/workshop/ex2.1

avr-objcopy -I ihex -O binary ex2.1.hex ex2.1.bin

r2 -a avr ex2.1.bin
```

Arithmetic instructions

```
add     r1,r2           ; r1  = r1 + r2
add     r28,r28         ; r28 = r28 + r28
and     r2,r3           ; r2  = r2 & r3
clr     r18             ; r18 = 0
inc     r0              ; r0  = r0 + 1
neg     r0              ; r0  = -r0
```

…

Bit manipulation instructions

```
lsl     r0       ; r0 << 2
lsr     r1       ; r1 >> 2
rol     r15      ; cyclic shift r16 bits to the
                   left
ror     r16      ; cyclic shift r16 bits to the
                   right
cbr     r18,1    ; clear bit 1 in r18
sbr     r16, 3   ; set bits 0 and 1 in r16
cbi     $16, 1   ; PORTB[1] = 0
```

Memory manipulation

```
mov       r1, r2          ; r1 = r2
ldi       r0, 10          ; r0 = 10
lds       r2,$FA00        ; r2 = *0xFA00
sts       $FA00,r0        ; *0xFA00 = r0
st        Z, r0           ; *Z(r31:r30) = r0
st        -Z, r1          ; *Z-- = r0
std       Z+5, r2         ; *(Z+5) = r2
in        r15, $16        ; r15 = PORTB
out       $16, r0         ; PORTB = r0
…
```

Same for LD*

Memory manipulation: stack

```
push      r14       ; save r14 on the Stack
```

SP = SP - 1

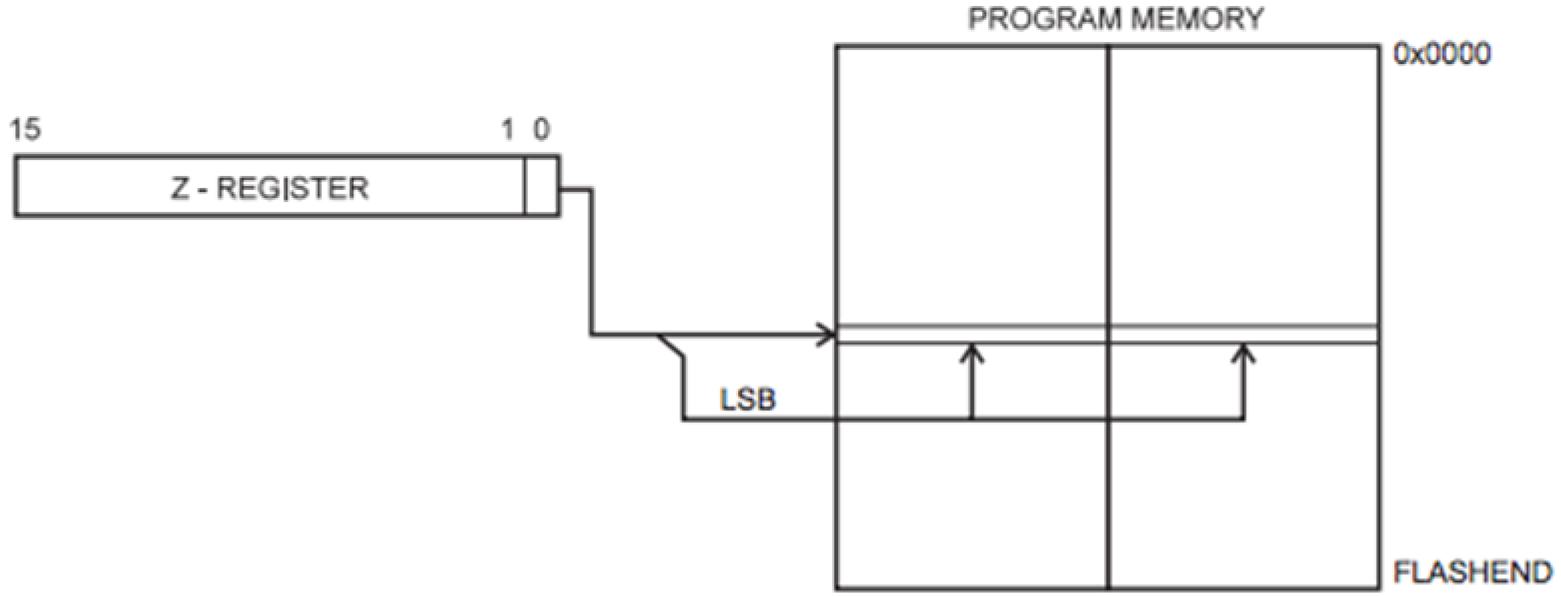
```
pop       r15       ; pop top of Stack to r15
```

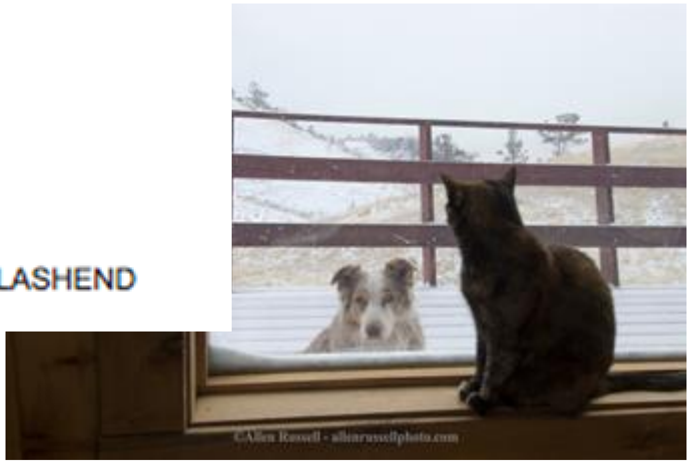
SP = SP + 1

Memory manipulation: flash

```
lpm r16, Z              ; r16 = *(r31:r30), but from flash
```

Figure 2-9. Program Memory Constant Addressing





Note: code is <u>separated</u> from data

Unconditional jump/call

```
jmp     $ABC1    ; PC = 0xABC1
rjmp    5        ; PC = PC + 5 + 1



call    $ABC1    ; “push PC+2”
                 ; jmp $ABC


ret              ; “pop PC”
```

Harvard architecture? But PC goes to **DATA** memory

STORE PC ON STACK?
ROP IS POSSIBLE!
memegenerator.net

SREG – 8-bit status register

C – **C**arry flag

Z – **Z**ero flag

N – **N**egative flag

V – two’s complement o**V**erflow indicator

S – **N** $\oplus$ **V**, for **S**igned tests

H – **H**alf carry flag

T – **T**ransfer bit (BLD/BST)

I – global **I**nterrupt enable/disable flag

Conditional jump

```
cpse      r1, r0   ; r1 == r2 ?
                        PC ← PC + 2 : PC ← PC + 3


breq      10       ; Z ? PC ← PC + 1 + 10
brne      11       ; !Z ? PC ← PC + 1 + 10
…
```

SREG manipulations

- sec/clc – set/clear carry
- sei/cli – set/clear global interruption flag
- se*/cl* – set/clear * flag in SRGE

Special

- break – debugger break
- nop – no operation
- sleep – enter sleep mode
- wdr – watchdog reset

Ex 2.2: **B**link! RE

EXERCISE

Real hardware & Simulator

```
cd /home/radare/workshop/ex2.1
avr-objcopy -I ihex -O binary blink.hex blink.bin
r2 -a avr ex2.1.bin
```

Questions:

1. Identify main() function and describe it using af
2. Find the LED switching command
3. What type of delay is used and why accurate frequency is required?
4. Locate interrupt vector and init code, explain what happens inside init code.

Reversing: function ~~s~~zignatures

- Most of firmwares contains zero or little strings.
- How to start?
- Use function signatures.
- However, in AVR world signatures may be to vary.
- Be prepared to predict target compiler/library/RTOS and options… or bruteforce it.
- In R2, signatures are called zignatures.

Embedded code priorities

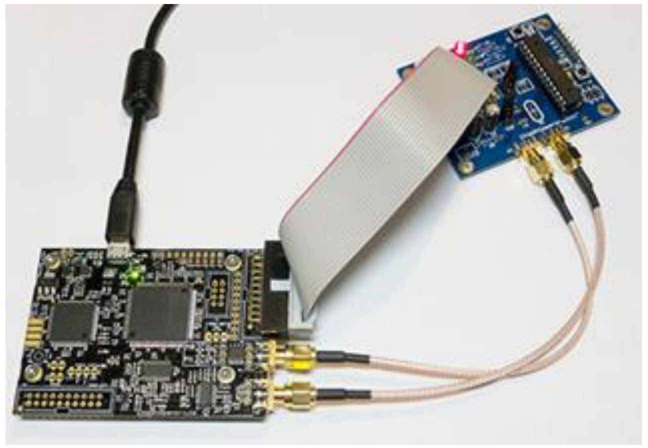
- Size
- Speed
- Hardware limits
- Redundancy
- …
- …
- …
- …
- Security

Fuzzing specifics

- Fuzzing is Fuzzing. Everywhere.
- But… we’re in embedded world.
- Sometimes you **could** detect crash through test/debug UART or pins
- In most cases, you could detect crash only by noticing, that device is no longer response
- Moreover, **watchdog timer** will could limit your detection capabilities, because it will reset device.
- So how to detect crash?

Fuzzing: ways to detect crash

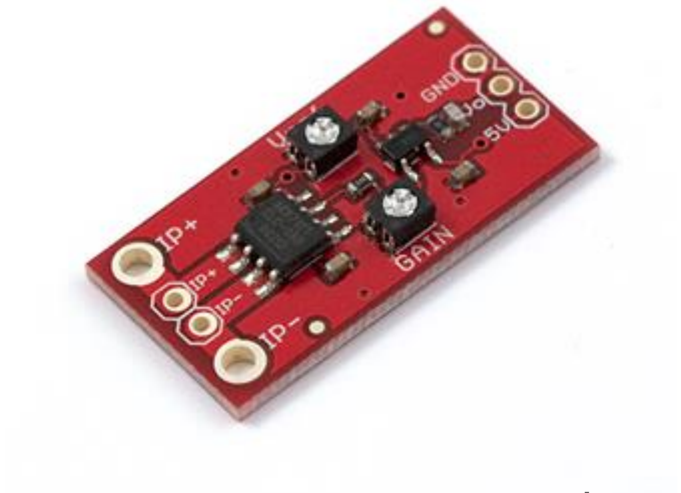
- JTAG debugger – break on RESET
- External analysis of functionality – detect execution pauses
- Detect bootloader/initialization code (e.g. for SRAM) behavior with logic analyzer and/or FPGA
- Detect power consumption change with oscilloscope/DAQ



Sometimes Arduino is enough to detect

- I^{2}C and SPI init sequencies could be captured by Arduino GPIOs
- If bootloader is slow and waits ~1 second, this power consumption reduction could be reliably detected with cheap current sensor, e.g.:

+



SparkFun Low Current Sensor Breakout - ACS712
https://www.sparkfun.com/products/8883

DEMO

Let’s proof it.

Part 3: Exploitation

Quick intro to ROP-chains

- Return Oriented Programming
- Series of function returns
- We’re searching for primitives (“gadgets”) ending with ‘ret’ that could be transformed into useful chain
- SP is our new PC

Notice: Arduino

- The next examples/exercises will be based upon Arduio ‘libc’ (in fact, Non-GNU AVR libc + Arduino wiring libs)
- We’re using Arduino because it’s complex, full of gadgets but free (against IAR or CV which are also complex and full of gadgets)
- Also, Arduino is fairly popular today, due to enormous number of libraries and “quick start” (e.g. quick bugs)

Ex 3.1 – 3.3

Real hardware

```
cd /home/radare/workshop/ex3.1
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex3.1.hex -g :4242
avr-gdb
```

Simulator

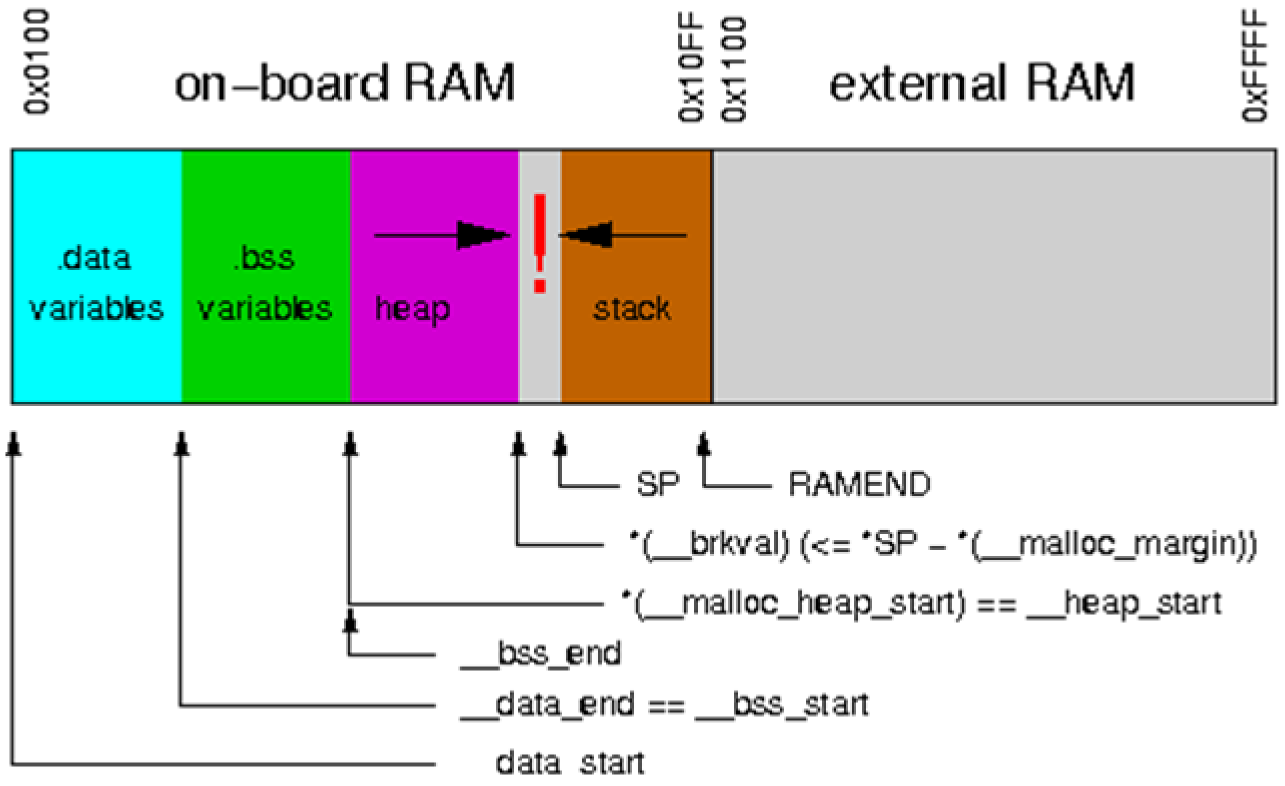
```
cd /home/radare/workshop/ex3.1_simulator
simulavr -P atmega128 -F 16000000 –f build-crumbuino128/ex3.1.elf
avr-gdb
```

Or: `node exploit.js`

EXAMPLE

Example 3.1: Abusing functionality: ret to function

Internal-SRAM only memory map



Overflowing the heap => Rewriting the stack!

http://www.atmel.com/webdoc/AVRLibcReferenceManual/malloc_1malloc_intro.html

How to connect data(string/binary) to code?

Standard model: with .data variables

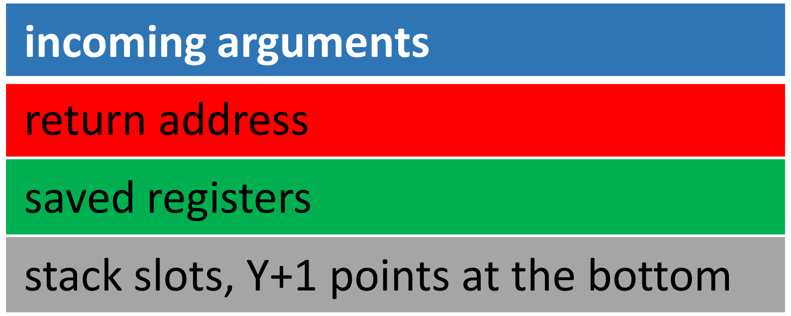
- Determine data offset in flash
- Find init code/firmware prologue where .data is copied to SRAM
- Using debugging or brain calculate offset of data in SRAM
- Search code for this address

Economy model: direct read with lpm/elpm

- Determine data offset in flash
- Search code with *lpm addressing to this offset

ABI, Types and frame layouts (GCC)

- Types: standard (short == int == 2, long == 4, except for double (4))
- Int could be 8bit if -mint8 option is enforced.
- Call-used: **R18–R27, R30, R31**
- Call-saved**: R2–R17, R28, R29**
- **R29:R28** used as frame pointer
- Frame layout after function prologue:



Calling convention: arguments

- An argument is passed either completely in registers or completely in memory.
- To find the register where a function argument is passed, initialize the register number R_n with R26 and follow this procedure:
    1. If the argument size is an odd number of bytes, round up the size to the next even number.
    2. Subtract the rounded size from the register number R_n.
    3. If the new R_n is at least R18 and the size of the object is non-zero, then the low-byte of the argument is passed in R_n. Other bytes will be passed in R_{n+1}, R_{n+2}, etc.
    4. If the new register number R_n is smaller than R18 or the size of the argument is zero, the argument will be passed in memory.
    5. If the current argument is passed in memory, stop the procedure: All subsequent arguments will also be passed in memory.
    6. If there are arguments left, goto 1. and proceed with the next argument.
- Varagrs are passed on the stack.

Calling conventions: returns

- Return values with a size of 1 byte up to and including a size of 8 bytes will be returned in registers.
- For example, an 8-bit value is returned in R24 and an 32-bit value is returned R22...R25.
- Return values whose size is outside that range will be returned in memory.

Example

For

```
int func (char a, long b);
```

- a will be passed in R24.
- b will be passed in R20, R21, R22 and R23 with the LSB in R20 and the MSB in R23.
- the result is returned in R24 (LSB) and R25 (MSB).

EXAMPLE

Example 3.2: Abusing functionality: simple ROP

ROP gadget sources

- User functions
- “Standard” or RTOS functions
- Data segment ☺
- Bootloader section

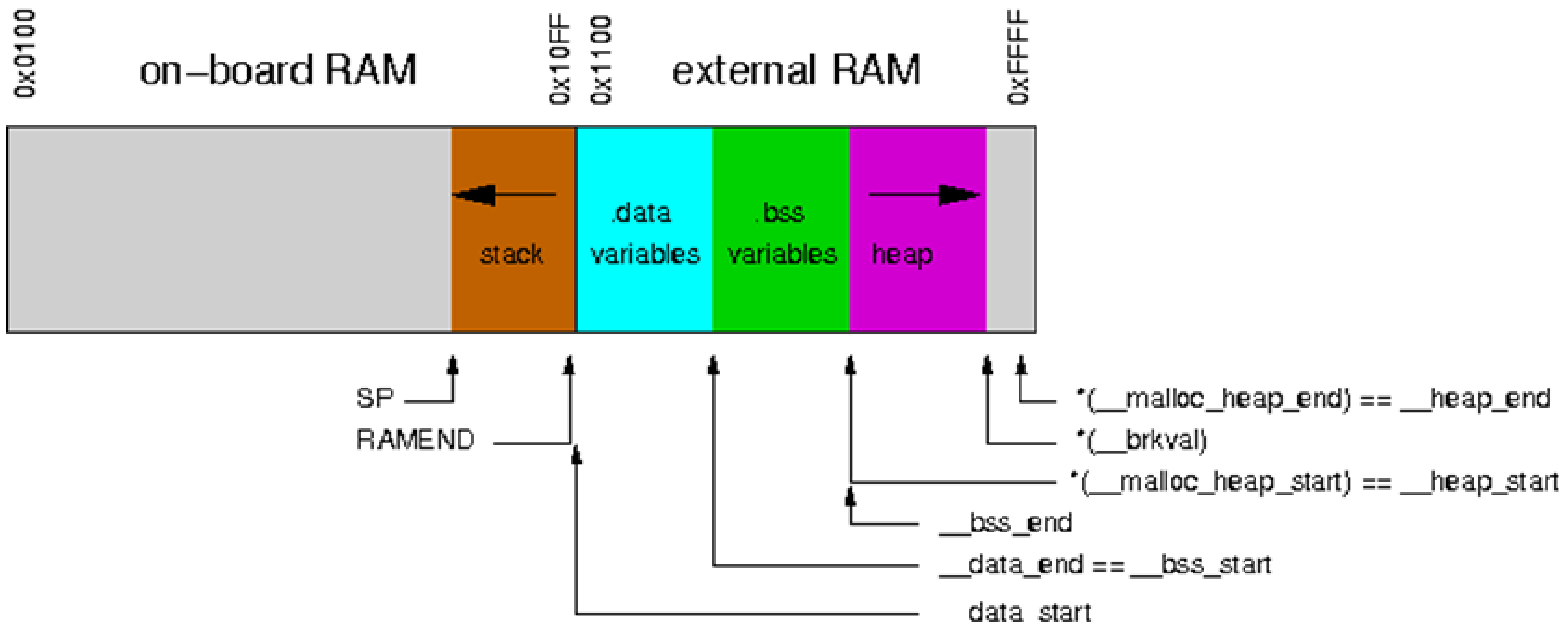
More code => more gadgets

ROP chain size

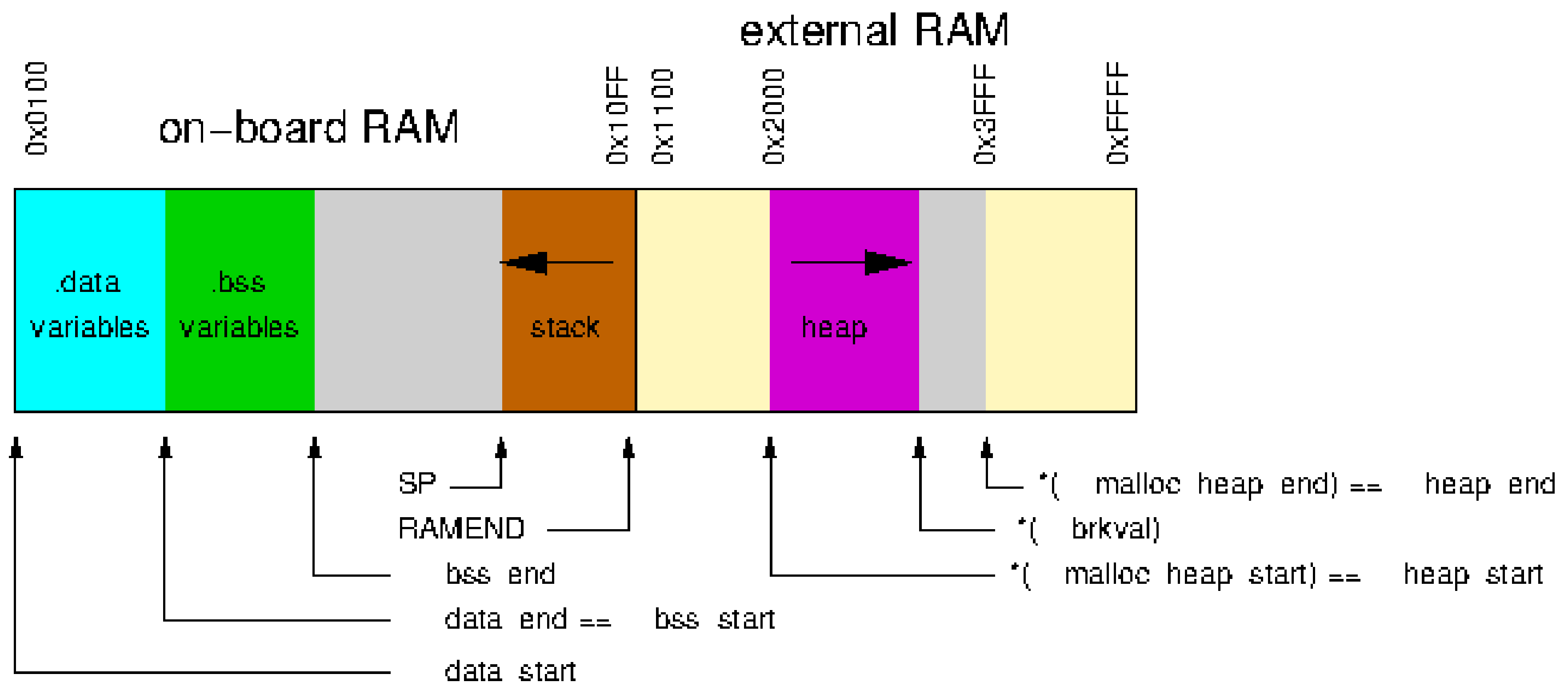
- It’s MCU
- SRAM is small
- SRAM is divided between register file, heap and stack
- Stack size is small
- We’re low on chain size
- Obviously, you will be limited with 20-40 bytes (~15-30 gadgets)
- However it all depends on compiler and **memory model**

http://www.atmel.com/webdoc/AVRLibcReferenceManual/malloc_1malloc_tunables.html

Memory maps – external SRAM/separated stack



Memory maps – external SRAM/mixed stack



Detecting “standard” functions

- In AVR we have bunch of compilers, libraries and even RToSes
- So, “standard” function could be vary.
- More bad news: memory model and optimization options could change function.
- The best approach is try to detect functions like malloc/str(n)cpy and then find the exact compiler/options that generates such code
- After it, use function signatures to restore the rest of the code
- In Radare2, you could use zignatures or Yara.

EXAMPLE

Example 3.3: more complex ROP

EXERCISE

Exercise 3.1: ret 2 function

build exploit that starts with ABC but calls switchgreen() function

EXERCISE

Exercise 3.3: print something else

3.3.1) build exploit that prints “a few seconds…”
3.3.2 (homework) build exploit that prints “blink a few seconds…”

Ex 3.4

Real hardware

```
cd /home/radare/workshop/ex3.1
```

in Blink.ino change APNAME constant from “esp_123” to “esp_your3digitnumber”

```
make
avr-objdump -I ihex -O binary build-crumbuino128/ex3.4.hex ex3.4.bin
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex3.4.hex -g :4242
avr-gdb
```

Connect to “esp_your3digitnumber” and type http://192.168.4.1 in your browser

Simulator

```
cd /home/radare/workshop/ex3.4_simulator
```

On 1st terminal: `node exploit.js`

On 2nd terminal: `tail -f serial1.txt`

In your browser: http://127.0.0.1:5000

EXAMPLE

Example 3.4: Blinking through HTTP GET

EXERCISE

Exercise 3.4: UARTing through HTTP query

EXERCISE

Exercise 3.5: Blinking through HTTP Post

It’s possible to construct ROP with debugger… …But if I don’t have some, how I could determine the overflow point?

- Reverse and use external analysis to find function that overflows
- Bruteforce it!

EXAMPLE

Arduino blink (ROP without debugger)

Part 4: Post-exploitation && Tricks

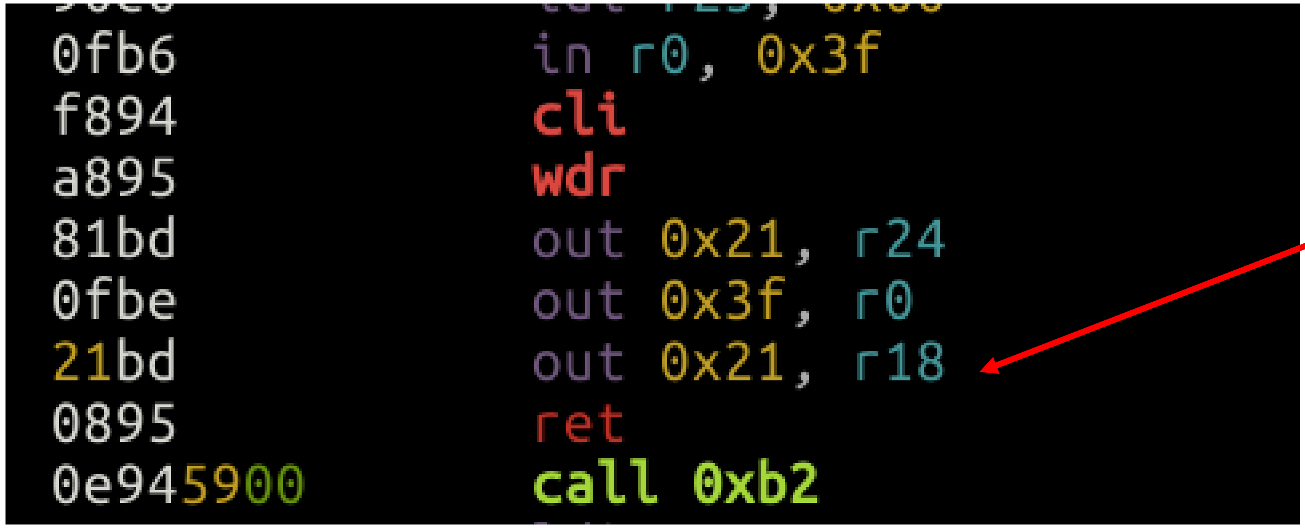
What do we want? (again)

- Evade watchdog
- Work with persistent memory (EEPROM and Flash)
- Stay persistent in device
- Control device for a long time

Evade the watchdog

In most cases, there three ways:

1. Find a ROP with **WDR** and periodically jump to it.
2. Find watchdog disable code and try to jump to it.
3. Construct watchdog disable code over watchdog enable code.

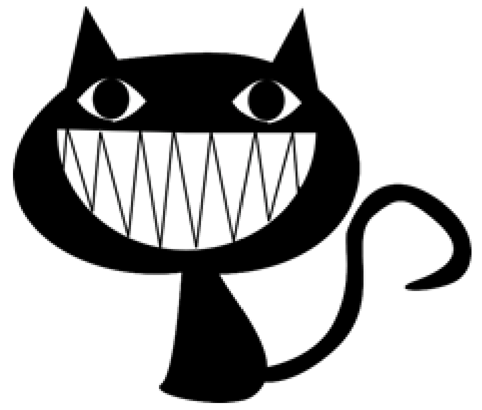


```
0fb6                    in r0, 0x3f
f894                    cli
a895                    wdr
81bd                    out 0x21, r24
0fbe                    out 0x3f, r0
21bd                    out 0x21, r18
0895                    ret
0e945900                call 0xb2
```

Set r18 to 0 and JMP here

Fun and scary things to do with memory…

- Read/write EEPROM (and extract cryptography keys)
- Read parts of flash (e.g., reading locked bootloader section) – could be more useful than it seems
- Staying persistent (writing flash)



Reading EEPROM/Flash

- Ok, in most cases it’s almost easy to find gadget(s) that reads byte from EEPROM or flash and stores it somewhere.
- We could send it back over UART or any external channel gadgets
- Not always possible, but there are good chances

Writing flash

- Writing flash is locked during normal program execution
- However, if you use “jump-to-bootloader” trick, you could write flash from bootloader sections.
- To do this, you need bootloader of that has enough gadgets.
- However, modern bootloaders are big and sometimes you could be lucky (e.g. Arduino bootloader)
- Remember to **disable interrupts** before jumping to bootloader.

“Infinite-ROP” trick*

1. Set array to some “upper” stack address (A1) and N to some value (128/256/etc) and JMP to read(..)
2. Output ROP-chain from UART to A1.
3. Set SPH/SPL to A1 (gadgets could be got from init code)
4. JMP to RET.
5. ???
6. Profit!

Don’t forget to include 1 and 3-4 gadgets in the ROP-chain that you are sending by UART.

*Possible on firmwares with read(array, N) from UART functions and complex init code

Mitigations

Mitigations (software)

- Safe code/Don’t trust external data (read 24 deadly sins of computer security)
- Reduce code size (less code -> less ROP gadgets)
- Use `rjmp/jmp` instead of `call/ret` (ofc, it won’t save you from ret2 function)
- Use “inconvenient” memory models with small stack
- Use stack canaries in your RTOS
- Limit external libraries
- Use watchdogs
- Periodically check stack limits (to avoid stack expansion tricks)

Mitigations (hardware)

- Disable JTAG/debuggers/etc, remove pins/wires of JTAG/ISP/UART
- Write lock bits to 0/0
- Use multilayered PCBs
- Use external/hardware watchdogs
- Use new ICs (more secure against various hardware attacks)
- Use external safety controls/processors

And last, but not least:

- Beware of Dmitry Nedospasov ;)

Part 4: Post-exploitation && Tricks

Conclusions

- RCE on embedded systems isn’t so hard as it seems.
- Abusing of functionality is the main consequence of such attacks
- However, more scary things like extracting cipherkeys or rewriting the flash is possible
- When developing embedded system remember that security also should be part of the Software DLC process.

Books/links

- Белов А.В. Разработка устройств на микроконтроллерах AVR
- Atmega128 disasm thread: http://www.avrfreaks.net/forum/disassembly-atmega128-bin-file
- Exploiting buffer overflows on arduino: http://electronics.stackexchange.com/questions/78880/exploiting-stack-buffer-overflows-on-an-arduino
- Code Injection Attacks on Harvard-Architecture Devices: http://arxiv.org/pdf/0901.3482.pdf
- Buffer overflow attack on an Atmega2560: http://www.avrfreaks.net/forum/buffer-overflow-attack-atmega2560?page=all
- Jump to bootloader: http://www.avrfreaks.net/forum/jump-bootloader-app-help-needed
- AVR Libc reference manual: http://www.atmel.com/webdoc/AVRLibcReferenceManual/overview_1overview_avr-libc.html
- AVR GCC calling conventions: https://gcc.gnu.org/wiki/avr-gcc
- Travis Goodspeed, Nifty Tricks and Sage Advice for Shellcode on Embedded Systems: https://conference.hitb.org/hitbsecconf2013ams/materials/D1T1%20-%20Travis%20Goodspeed%20-%20Nifty%20Tricks%20and%20Sage%20Advice%20for%20Shellcode%20on%20Embedded%20Systems.pdf
- Pandora’s Cash Box: The Ghost Under Your POS: https://recon.cx/2015/slides/recon2015-17-nitay-artenstein-shift-reduce-Pandora-s-Cash-Box-The-Ghost-Under-Your-POS.pdf

Radare2. Links

- http://radare.org
- https://github.com/pwntester/cheatsheets/blob/master/radare2.md
- https://www.gitbook.com/book/radare/radare2book/details
- https://github.com/radare/radare2ida

@dark_k3y

@dukeBarman

http://radare.org/r/

http://dsec.ru

http://eltech.ru

http://zorsecurity.ru

Now it’s CTF time! ☺