

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

Alexander @dark_k3y Bolshev

Boris @dukeBarman Ryutin

Agenda

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

Part 2: Pre-exploitation

Part 3: Exploitation and ROP-chains building

Part 4: Post-exploitation and tricks



Thus: If you have
a question, please
interrupt and ask
immediately

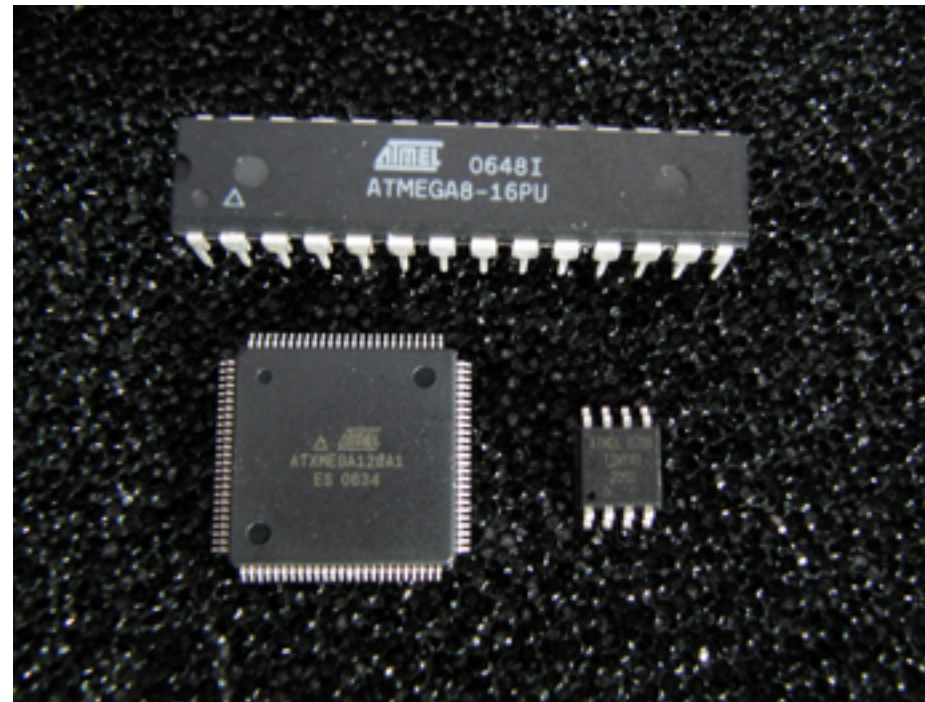
Disclaimer:

- 1) Training is **VERY** fast-paced
- 2) Training is highly-practical
- 3) You may encounter information overflow
- 4) My English is far from perfect

Part 1: What is AVR?

AVR

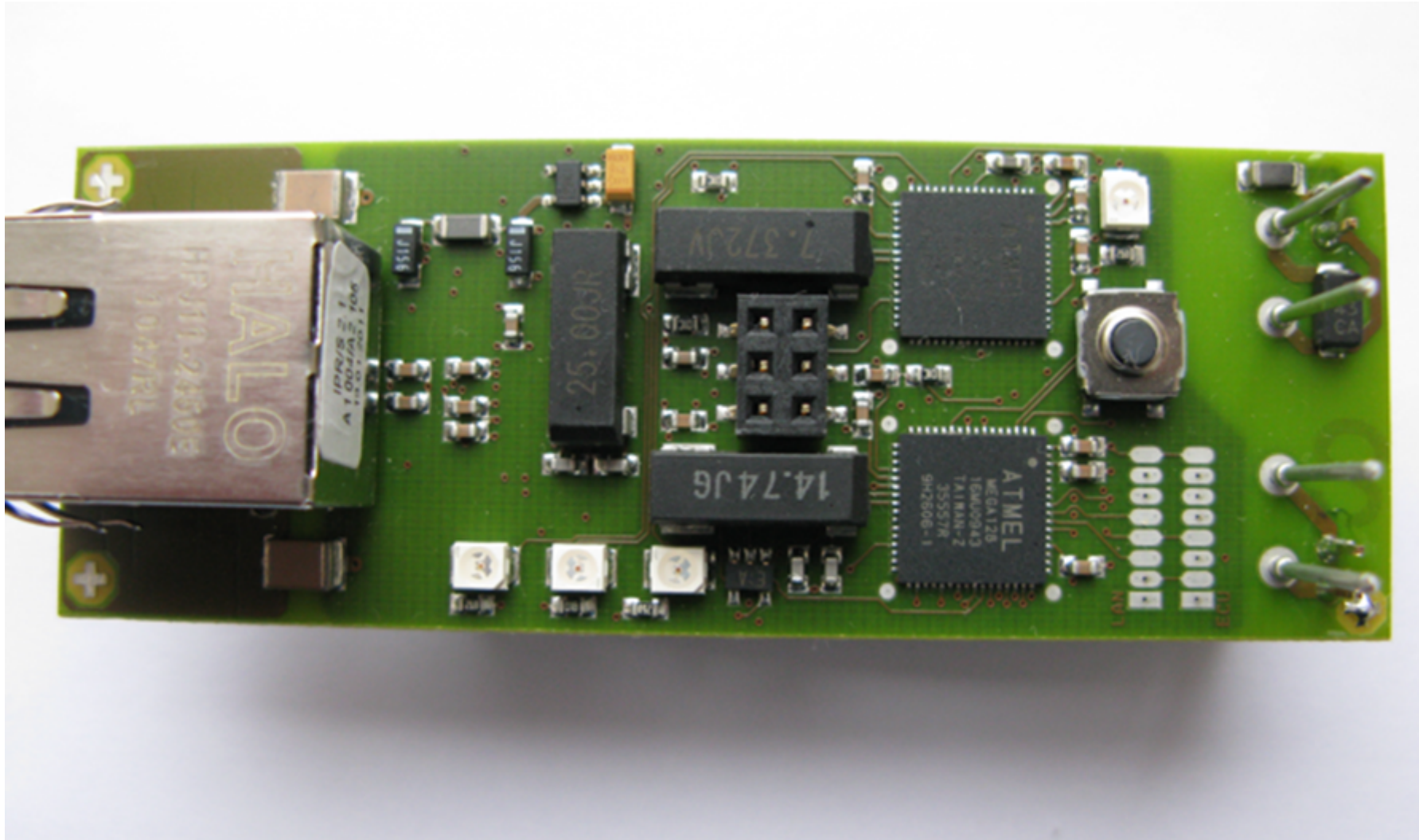
- **A**lf (Egil Bogen) and **V**egard (Wollan)'s RISC processor
- Modified Harvard architecture 8-bit RISC single-chip microcontroller
- Developed by Atmel in 1996 (now Dialog/Atmel)



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
- WirelessHART transmitters and sensors
- Your new shiny IoE fridge ;-)

AVR inside industrial gateway



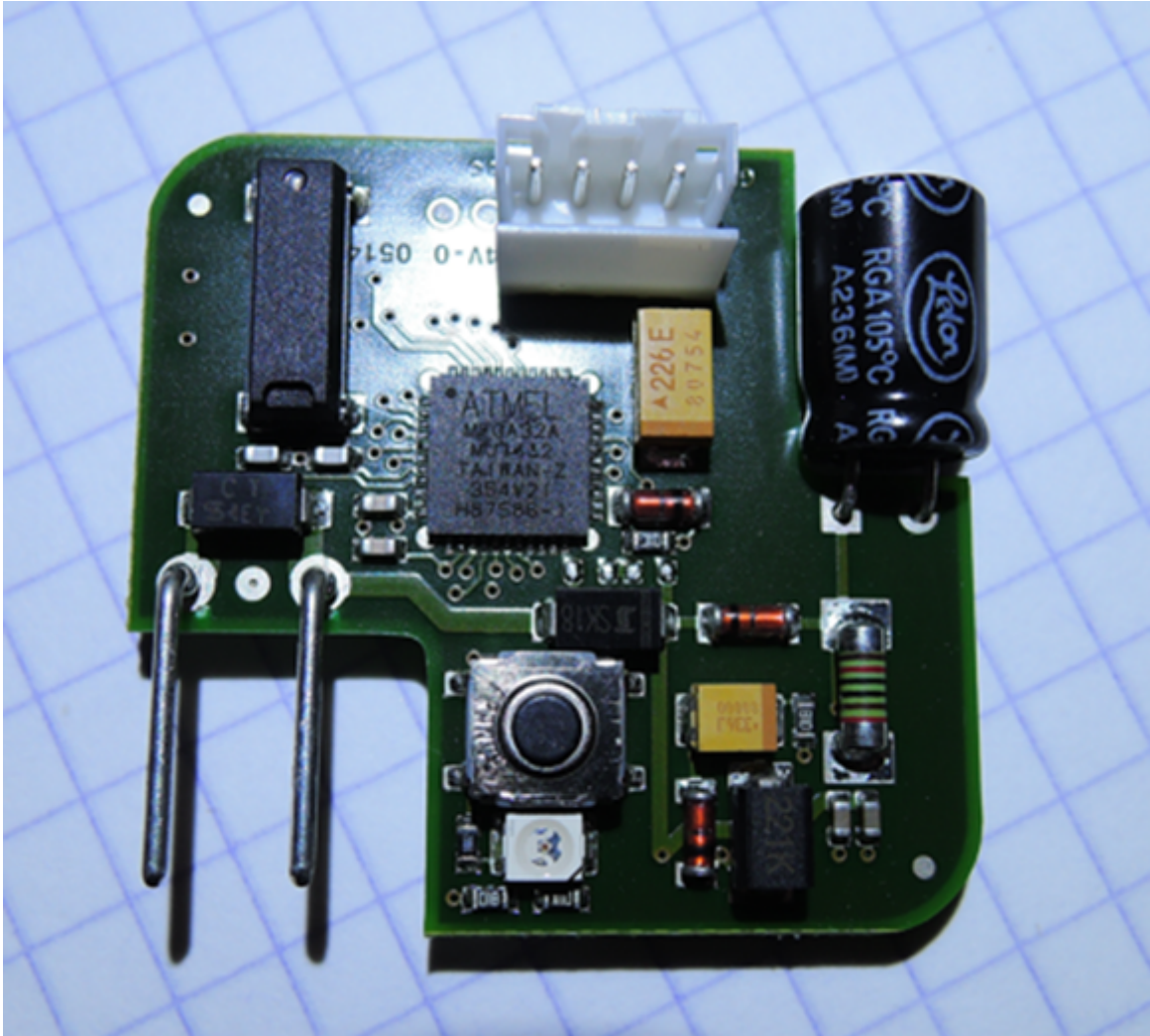
Synapse IoT module with Atmega128RFA1 inside



Philips Hue Bulb



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



Modified Harvard architecture...

...allows the contents of the instruction memory to be accessed as if it were data¹

¹but 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 and Flash
- GPIO (discrete I/O pins)
- UART(s)
- I²C, SPI, CAN, ...
- ADC
- PWM or DAC
- Timers
- Watchdog
- Clock generator and divider(s)
- Comparator(s)
- In-circuit programming and debugging support

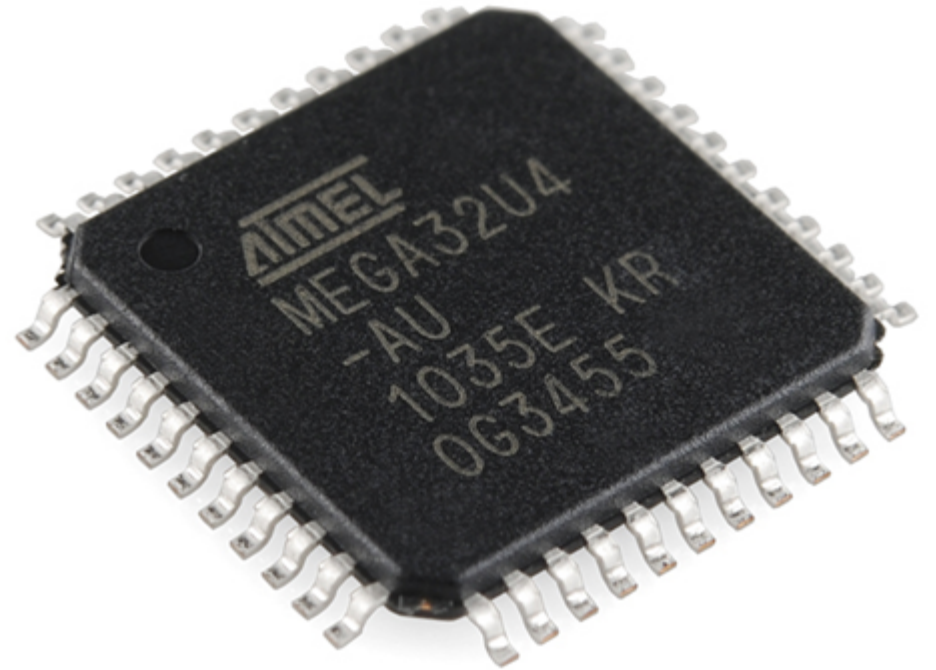
AVRs are very different

- AtTiny13
- Up to 20 MIPS Throughput 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, I²C, ... => tons of stuff



AVRs are very different

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

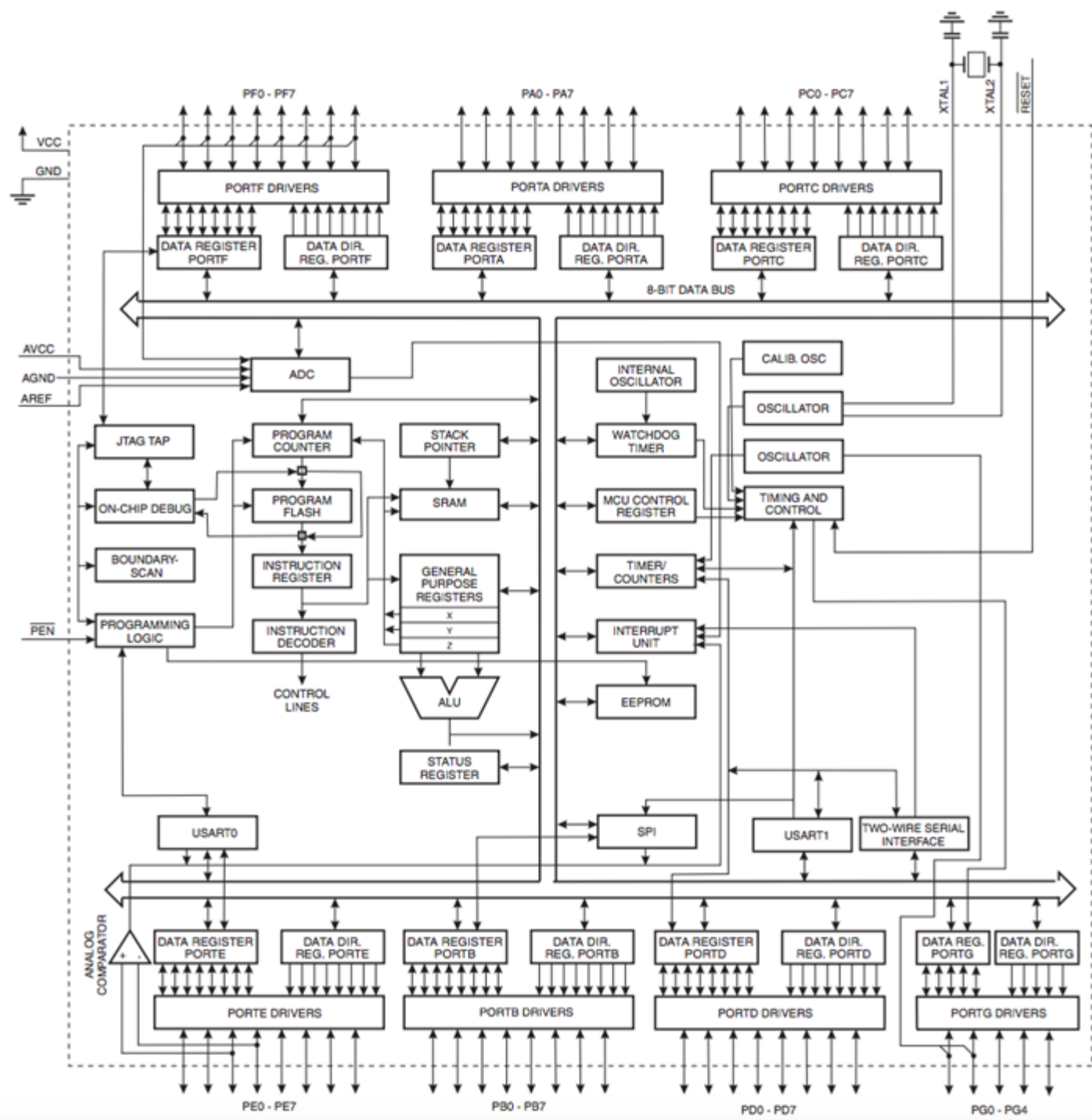


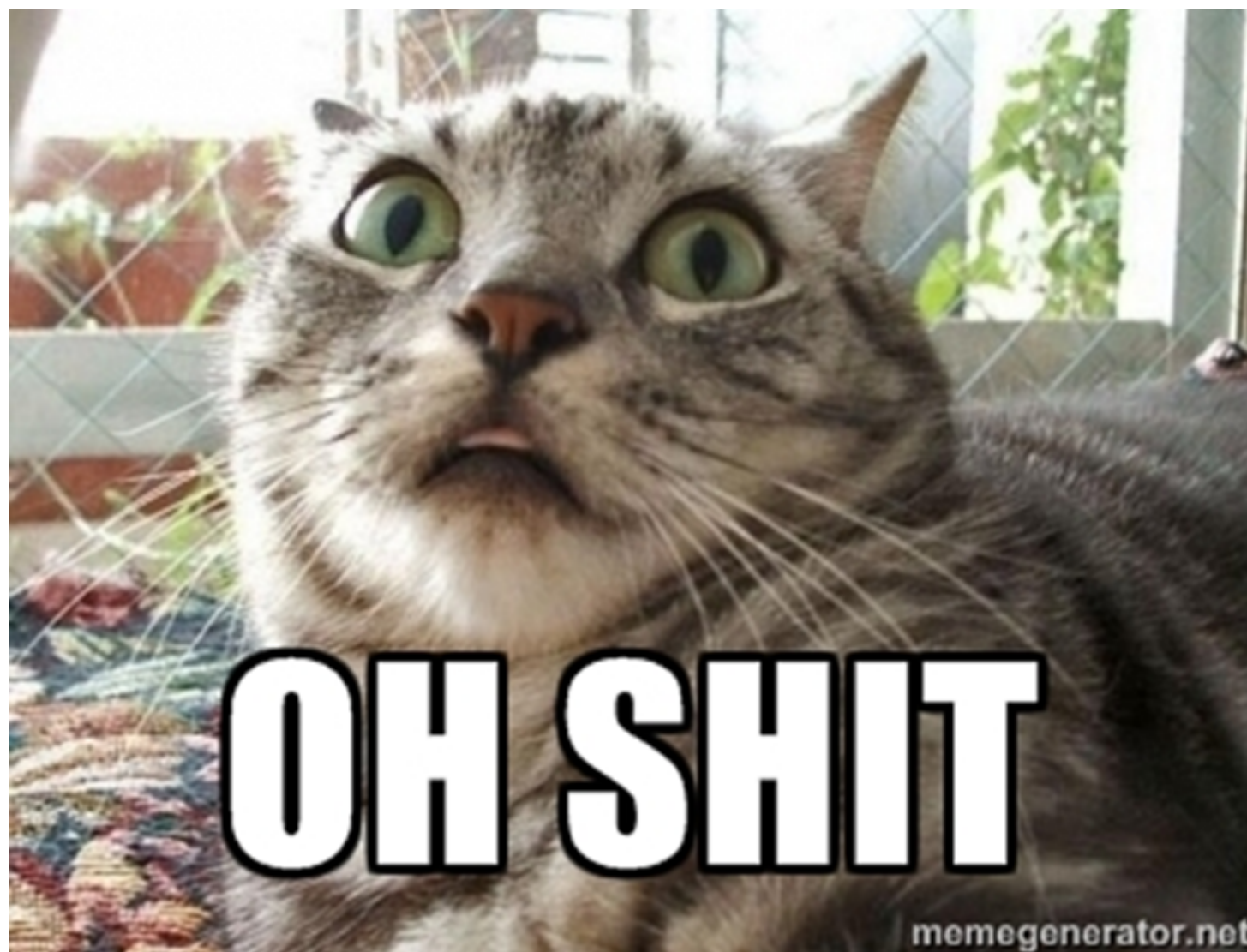
The workshop focuses on this chip

Why Atmega128?

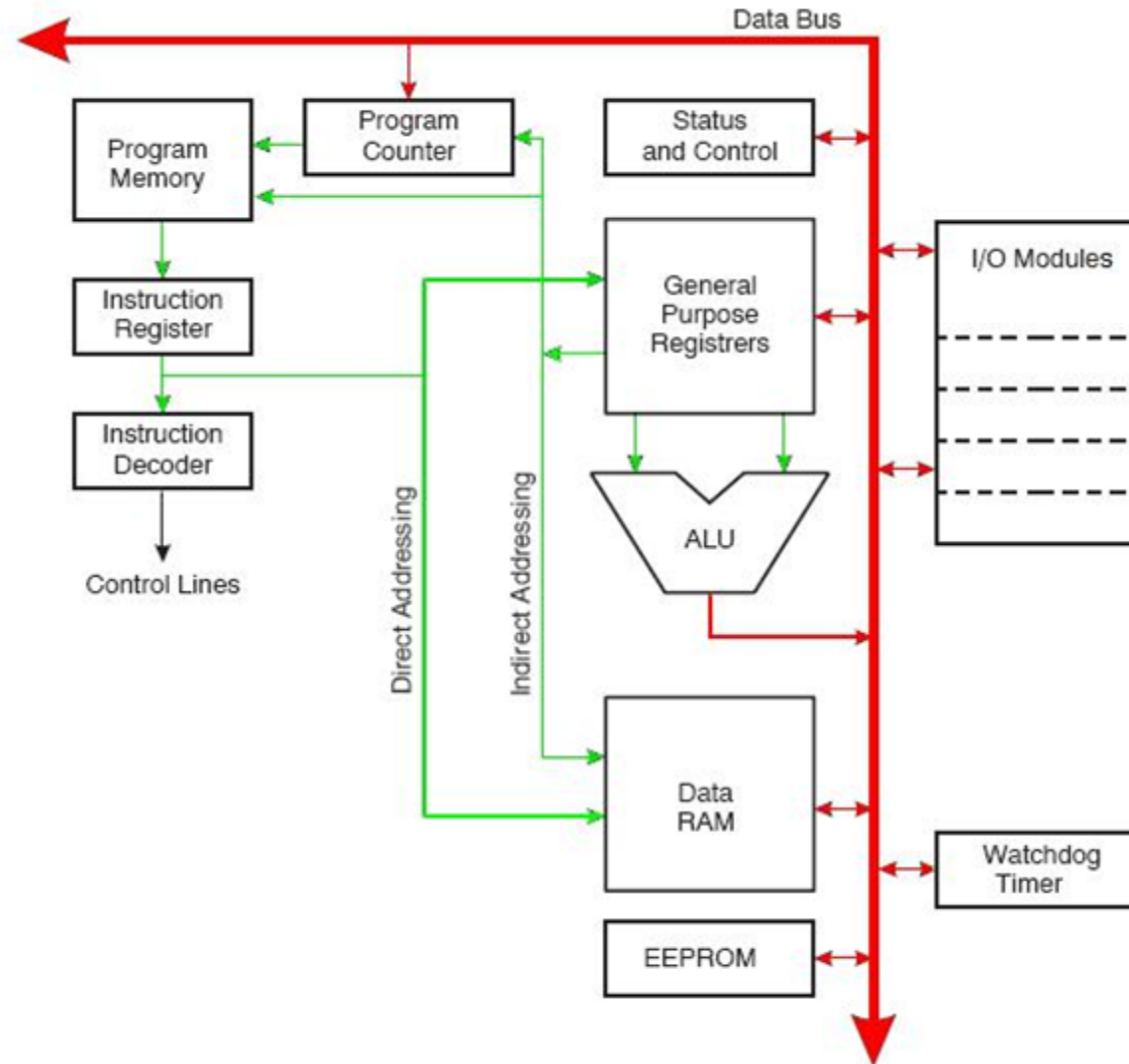
- Old, but very widespread chip
- At90can128 – popular analogue for CAN buses in automotive application
- **Cheap JTAG programmer**
- Much SRAM == ideal for ROP-chain construction training

Let's look to the architecture of Atmega128...





Ok, ok, let's simplify a bit 😊

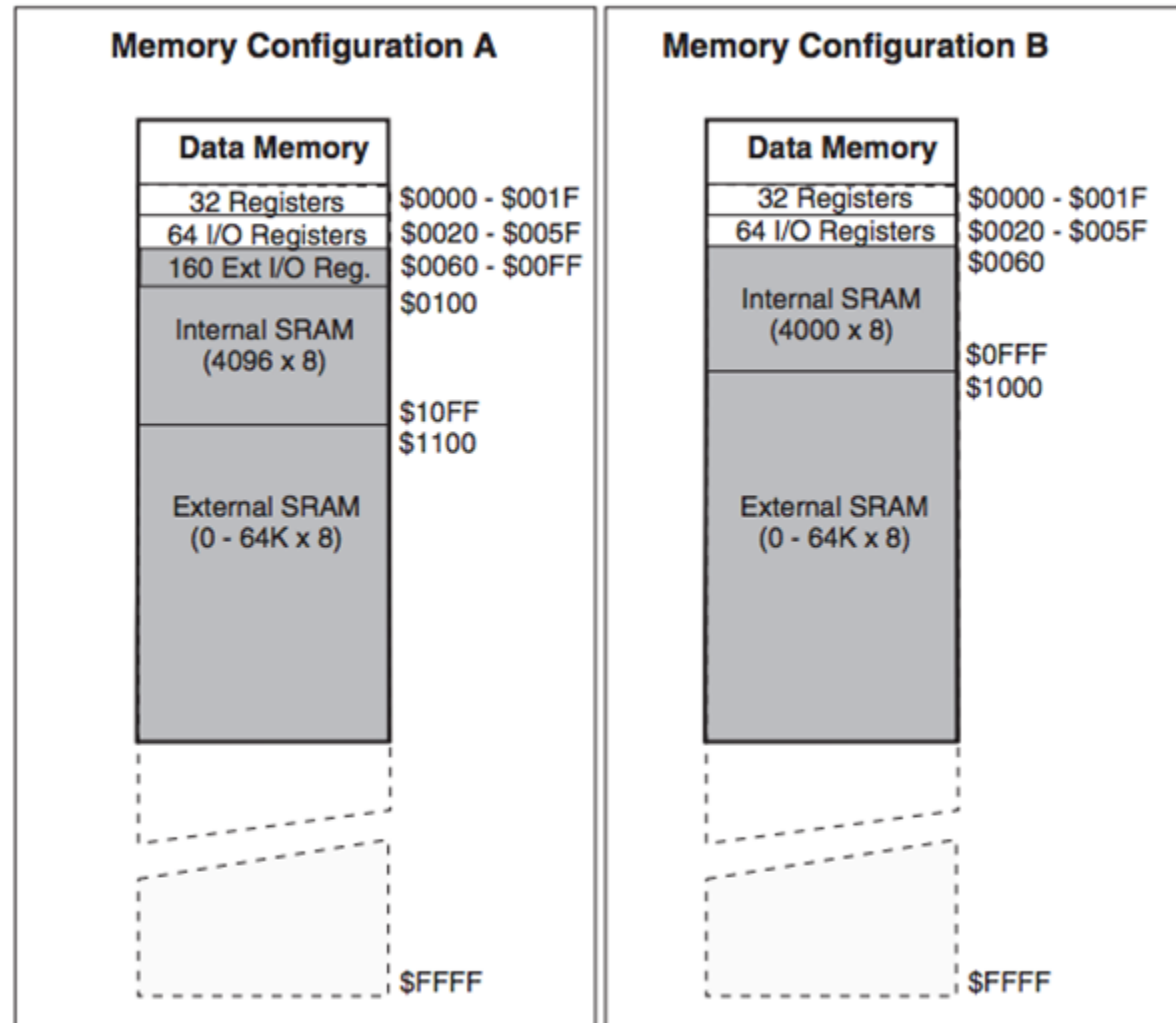


Note: code is separated from data



Memory map

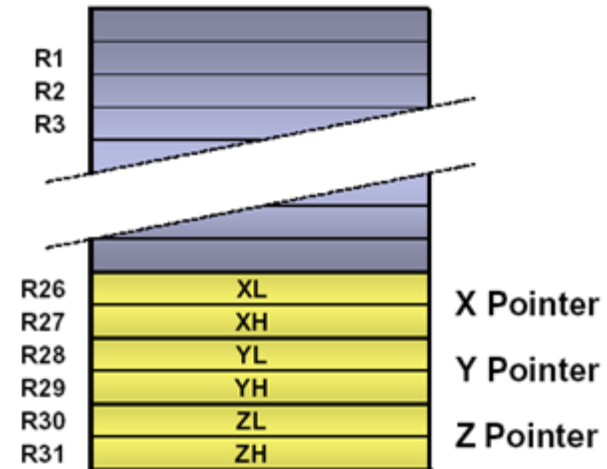
Figure 9. Data Memory Map



Memory: registers

- R0-R25 – GPR
- X,Y,Z – pair “working” registers, e.g. for memory addressing operations
- I/O registers – for accessing different “hardware”

AVR Register File



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

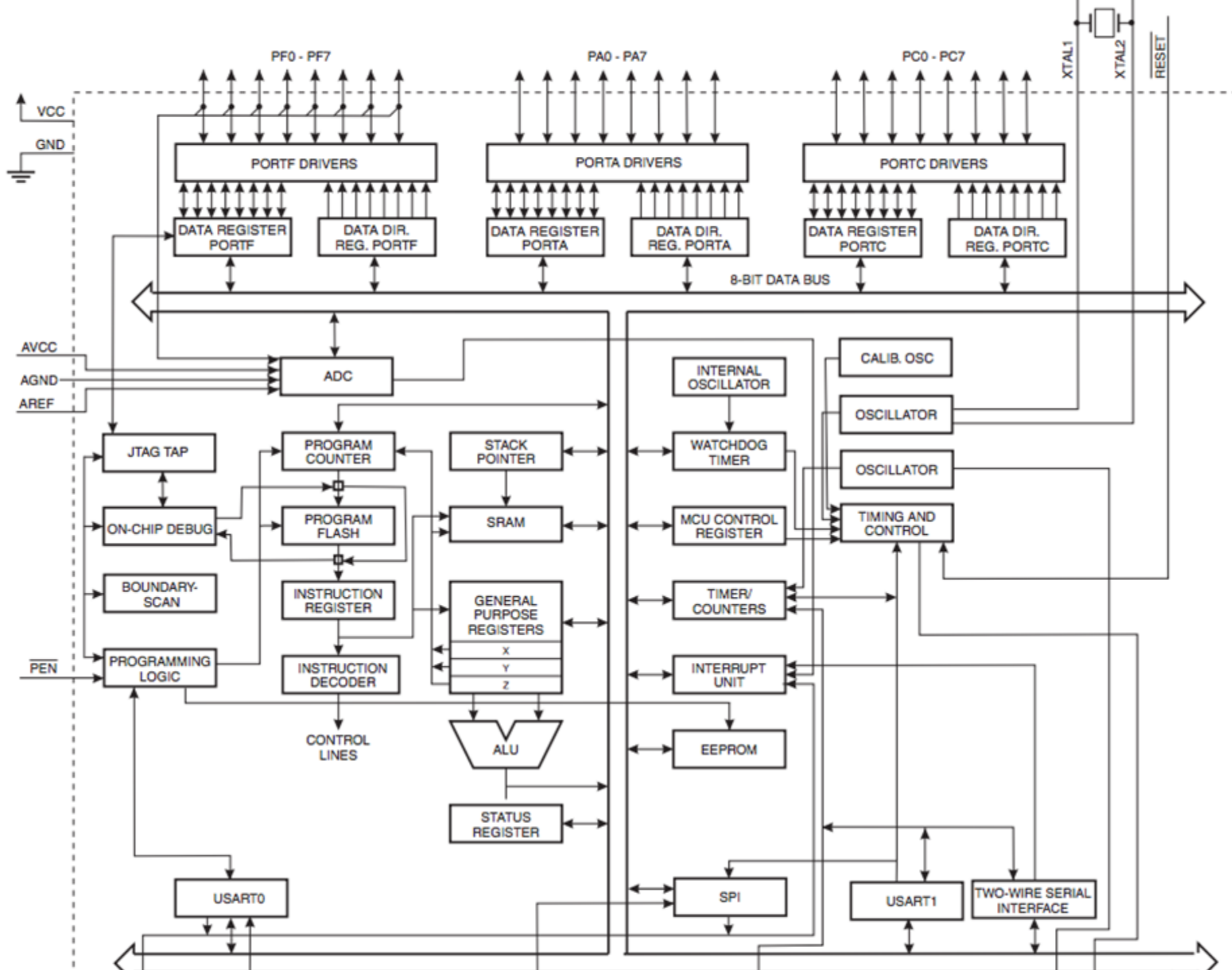


Memory addressing directions

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

Datasheets are your best friends!





Interrupts

- Interrupts interrupt normal process of code execution for handling something or reacting to some event
- Interrupt handler is a 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

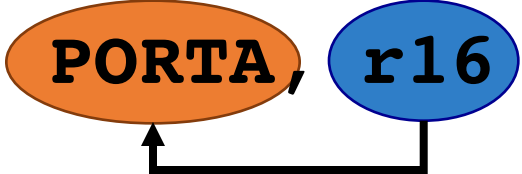


Instruction types

- 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, pins, ability to debug/program, etc.
- 2 lock bits controls programming protection.

The screenshot shows the 'Atmel-ICE (J41800035627) @ - Device Programming' window. The 'Fuses' tab is selected in the left sidebar. The main area displays a list of fuse names and their values, along with checkboxes for various settings. Below this is a table for Fuse Registers (EXTENDED, HIGH, LOW) and their values. At the bottom, there are checkboxes for 'Auto read' and 'Verify after programming', and buttons for 'Program', 'Verify', 'Read', and 'Close'. A status bar at the bottom shows the progress of reading registers.

Fuse Name	Value
M103C	<input type="checkbox"/>
WDTON	<input type="checkbox"/>
OCDEN	<input checked="" type="checkbox"/>
JTAGEN	<input checked="" type="checkbox"/>
SPIEN	<input checked="" type="checkbox"/>
EESAVE	<input type="checkbox"/>
BOOTSZ	4096W_F000
BOOTRST	<input type="checkbox"/>
CKOPT	<input type="checkbox"/>
BODLEVEL	2V7
BODEN	<input type="checkbox"/>
SUT_CKSEL	EXTHIFXTALRES_16KCK_64MS

Fuse Register	Value
EXTENDED	0xFF
HIGH	0x19
LOW	0xFF

Auto read
 Verify after programming

Starting operation read registers
Reading register EXTENDED...OK
Reading register HIGH...OK
Reading register LOW...OK
Read registers...OK

Read registers...OK

AVR bootloader – what is it?

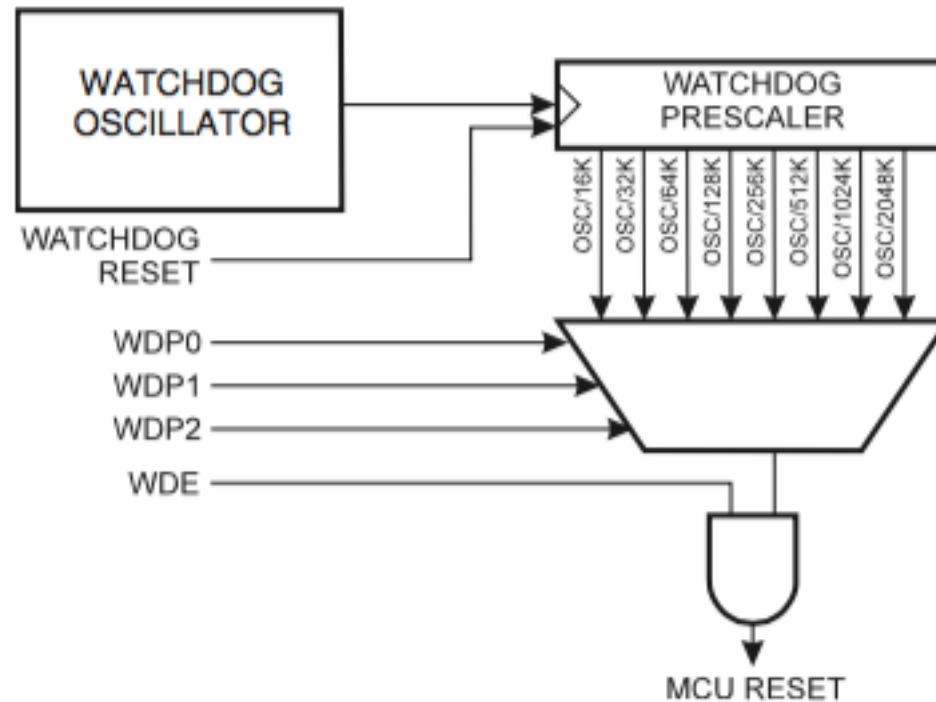
- Part of code that starts **BEFORE** RESET interrupt.
- Could be used for self-programmable (i.e. without external device) systems, in case you need update the firmware of your IoT device.
- Bootloader address and behavior configured via FUSES.
- BLB lock bits controls bootloader ability to update application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



Development for AVR

Atmel studio

The screenshot displays the Atmel Studio IDE interface. The main window shows a C source file named "Blinking LED.c" with the following code:

```
#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
    }
}
```

The Solution Explorer on the right shows the project structure for "Blinking LED", including "Dependencies", "Output Files", and "Blinking LED.c". The Output window at the bottom shows the successful compilation process:

```
Show output from: Build

"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"

Build succeeded
```

The status bar at the bottom indicates "Build succeeded" and shows the current line and column numbers: Ln 19, Col 42, Ch 36, INS.

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

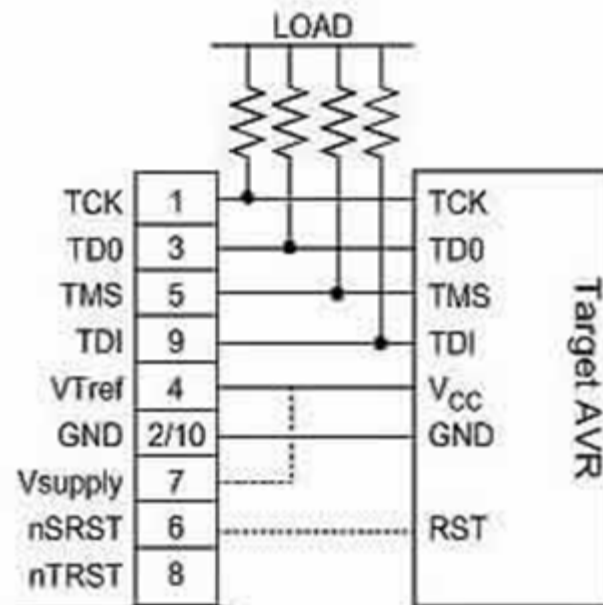
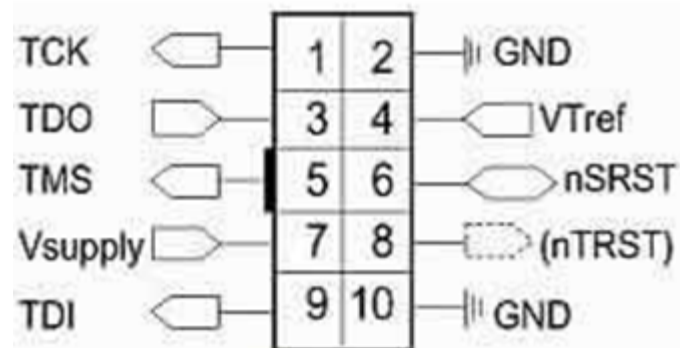
Other tools

- 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 AVR

AVR JTAG mkII



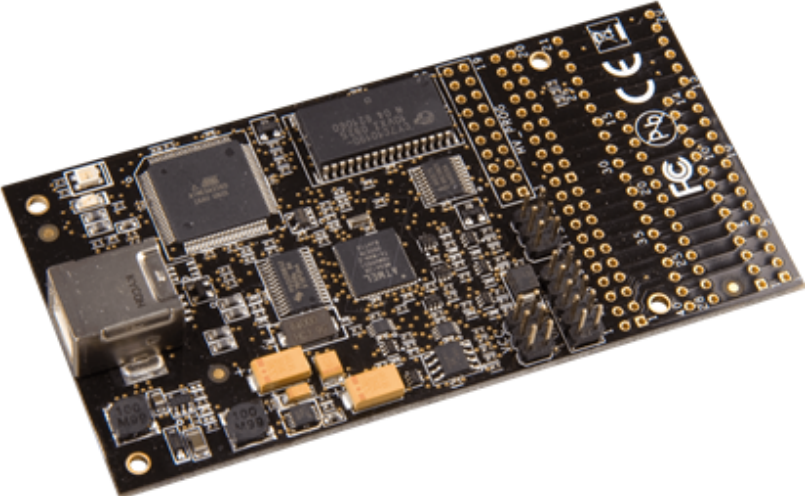
AVR JTAGIce3



AVR JTAG mkl



AVR Dragon



Atmel ICE3



Avarice

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

```
avarice --program --file test.elf --part atmega128 --jtag /dev/ttyUSB0 -d :4242
```

AVR-GDB

- Part of “nongnu” AVR gcc kit.
- Roughly ported standard gdb to AVR platform
- Doesn't understand Harvard architecture
 - You will need to resolve memory address by reference of \$pc to read the flash

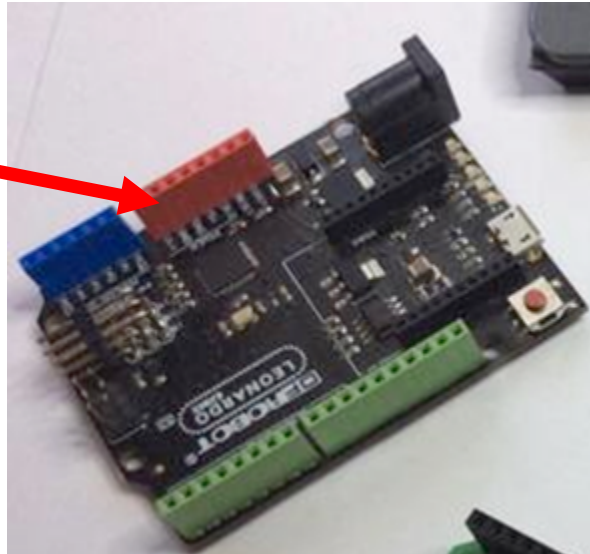
```
(gdb) x/10b $pc + 100
```

Simulators

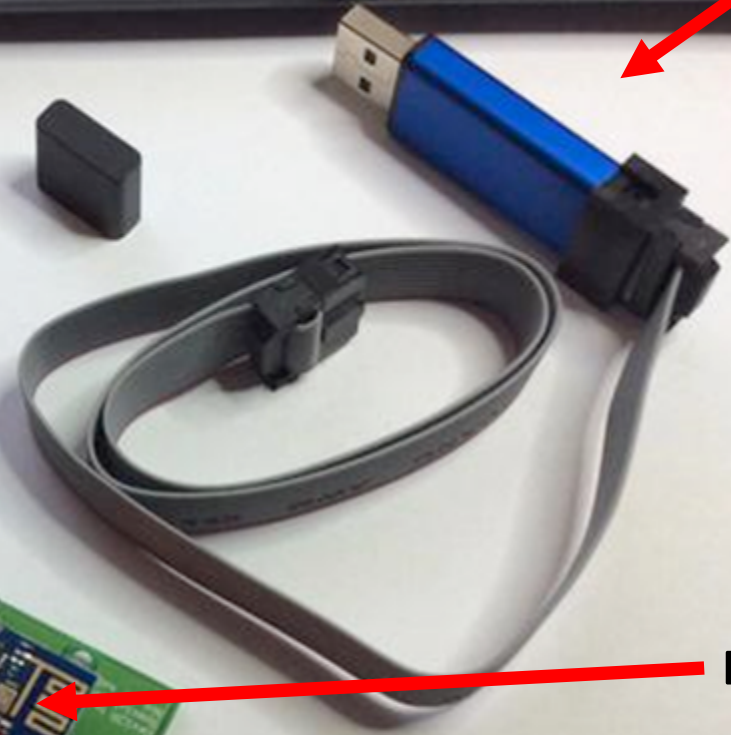
- Atmel Studio simulator
- Proteus simulator
- Simavr
- Simulavr

Training kit content

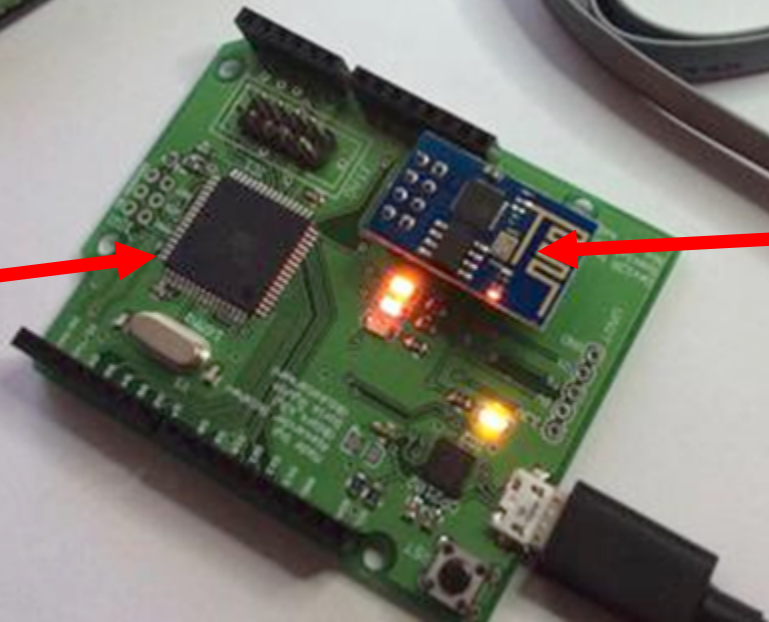
Arduino



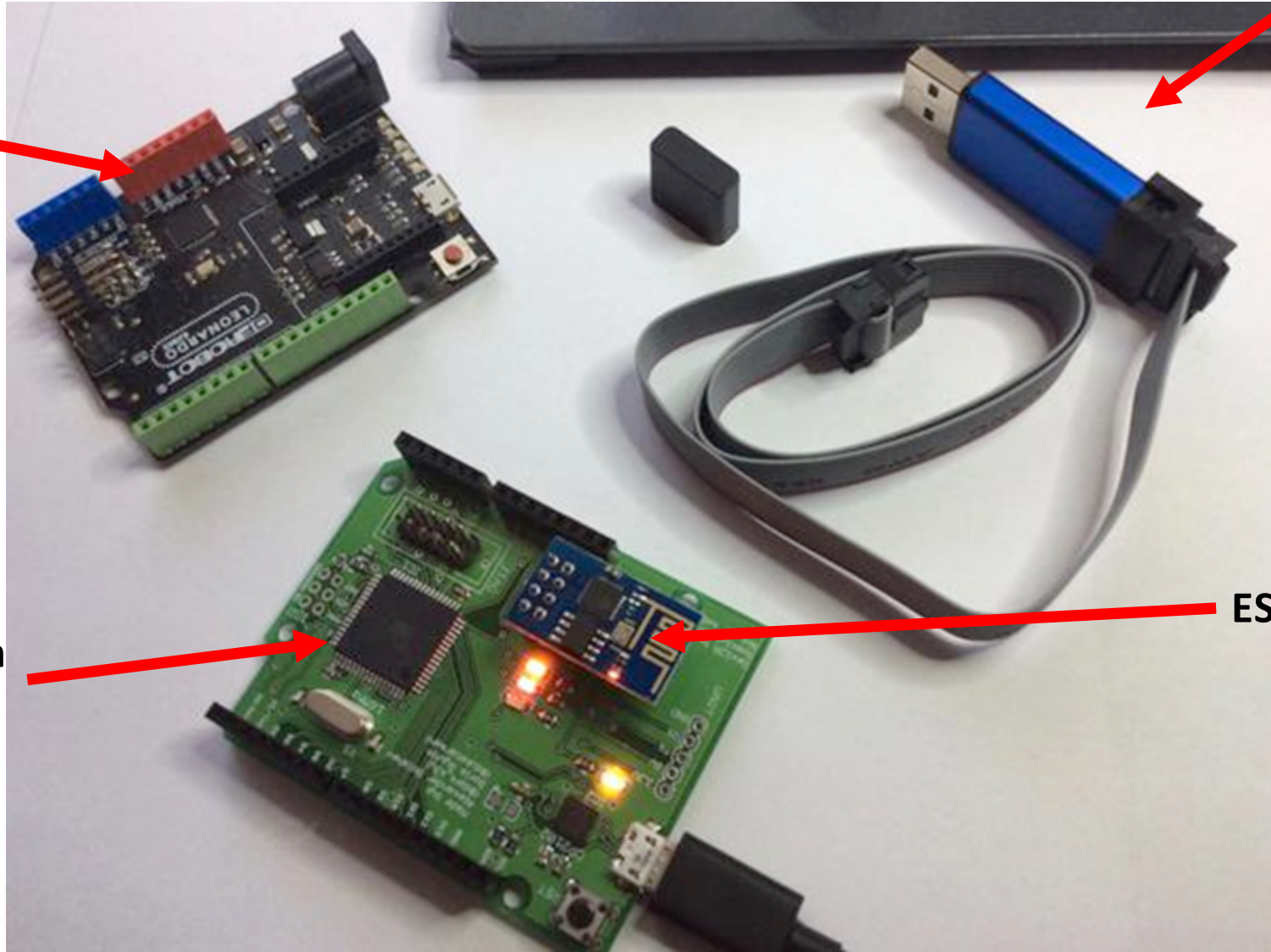
AVR JTAG mkl



Atmega128 custom devboard



ESP8266 "WiFi to serial"



VM access:

Login: radare

Password: radare

Ex 1.1: Hello world!

EXAMPLE

```
cd /home/radare/workshop/ex1.1  
avtarice --mkI --jtag /dev/ttyUSB0 -p -e --file hello.hex
```

Communication: CuteCom **or** Cscreen /dev/ttyUSB1 9600

For debugging:

```
avtarice --mkI --jtag /dev/ttyUSB0 -p -e --file hello.hex -d :4242
```

In new terminal window:

```
avr-gdb  
(gdb) target remote :4242
```

Ex 1.1_simulator: Hello world!

EXAMPLE

Simulator

```
cd /home/radare/workshop/ex1.1_simulator
simulavr -d atmega128 -f hello.elf -F 16000000 -x -,E1,9600 -y -,E0,9600
```

For debugging:

```
simulavr -d atmega128 -f hello.elf -F 16000000 -x -,E1,9600 -y -,E0,9600 -g
avr-gdb
(gdb) target remote :1212
```

Ex 1.2: Blink!

EXAMPLE

```
cd /home/radare/workshop/ex1.2
```

```
avrisp --mkI --jtag /dev/ttyUSB0 -p -e --file blink.hex
```

For debugging:

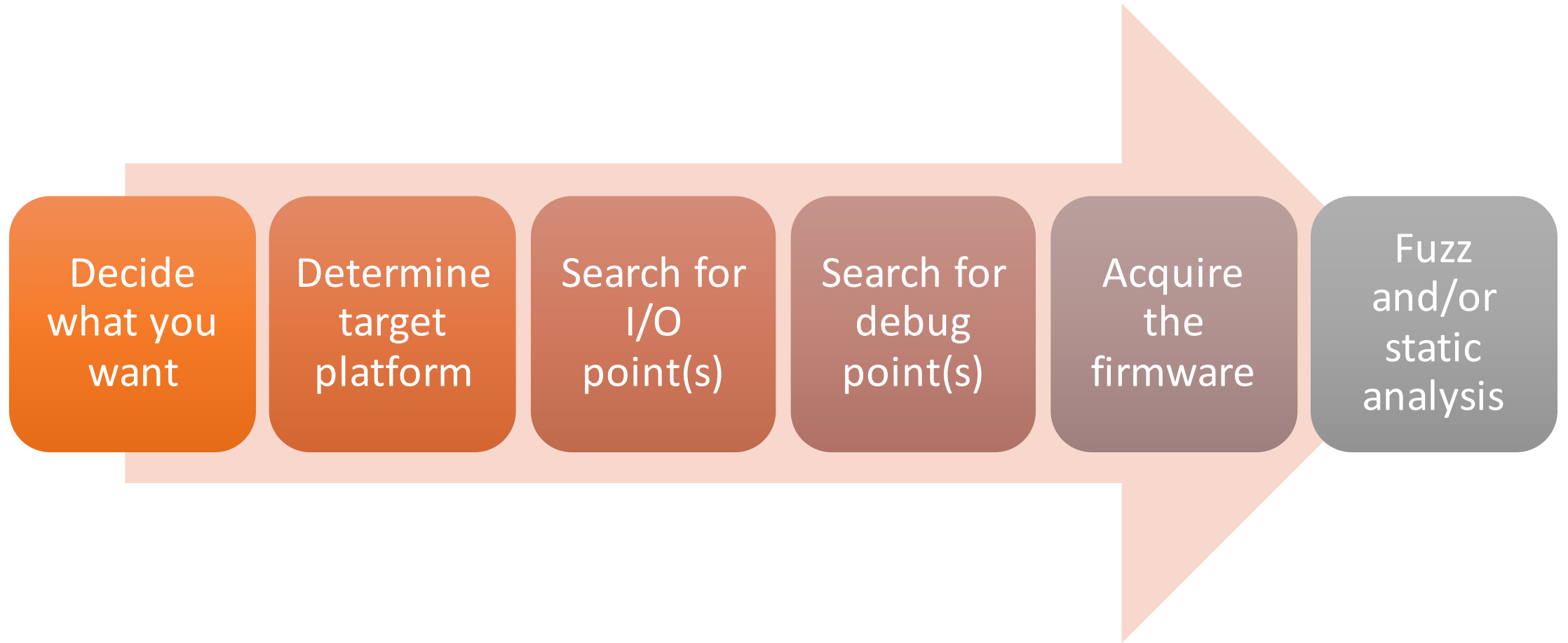
```
avrisp --mkI --jtag /dev/ttyUSB0 -p -e --file blink.hex -d :4242
```

```
avr-gdb
```

```
(gdb) target remote :4242
```


Part 2: Pre-exploitation

You have a device. First steps?



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?

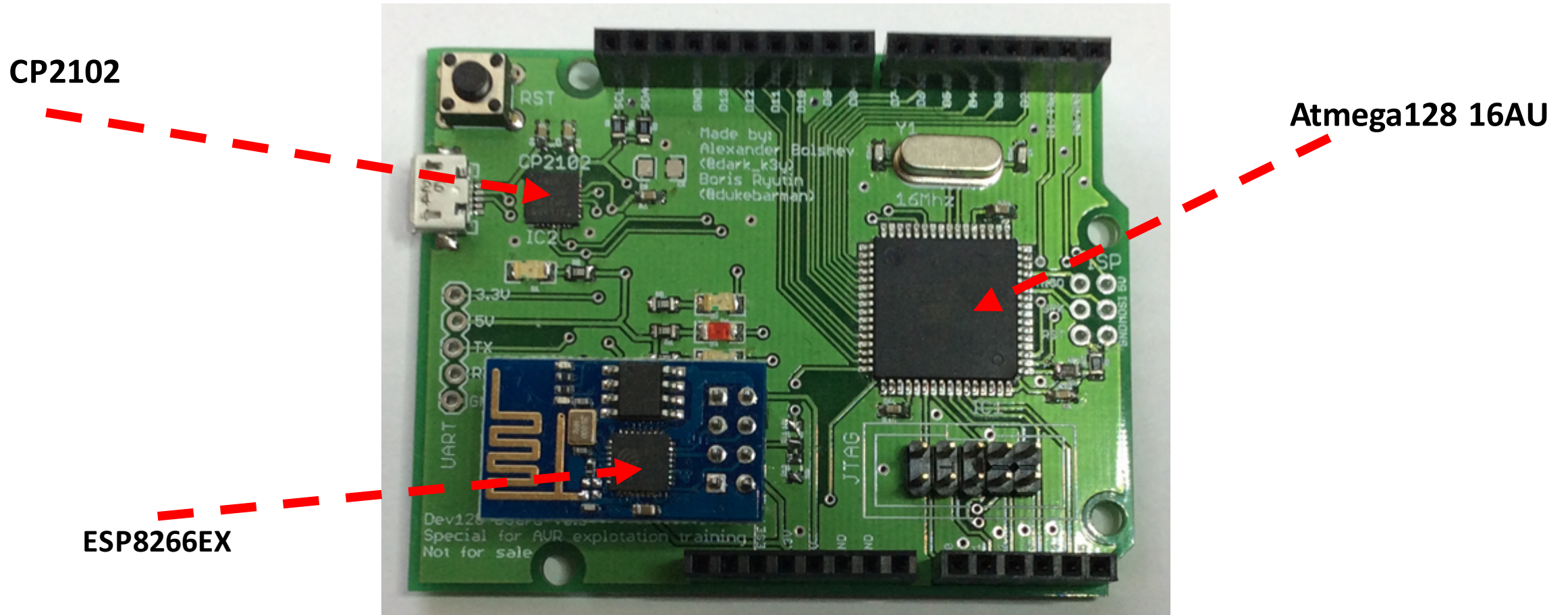
To start with, decide what you want:

- Abuse of functionality
- Read something from EEPROM/Flash/SRAM
- Stay persistent



Determine target platform

- Look at the board and search for all ICs...



Digikey/Octopart/Google...

The screenshot shows the Octopart website interface. The search bar contains 'Atmega128 16au'. The page displays two product listings for ATMEGA128-16AU. The first listing is for the ATMEGA128-16AU (TQFP-64) and the second is for the ATMEGA1280-16AU (TQFP-100). Both listings include a table of distributors and their prices for various quantities (1, 100, 1,000, 10,000). A red circle highlights the 'Datasheet' link in the first listing.

Product 1: ATMEGA128-16AU
ATMega Series 16 MHz 128 KB Flash 4 KB SRAM 8-Bit Microcontroller - TQFP-64.

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

[See Details](#) [Specs](#) [Realtime data](#)

Product 2: ATMEGA1280-16AU
ATMega Series 16 MHz 128 KB Flash 8 KB SRAM 8-Bit Microcontroller - TQFP-100.

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	620.70	620.70	Buy Now
★ Digi-Key	ATMEGA1280-16AU-ND	1,479	1	Tray	* RUB 911.24	704.23	570.00	570.00	Buy Now

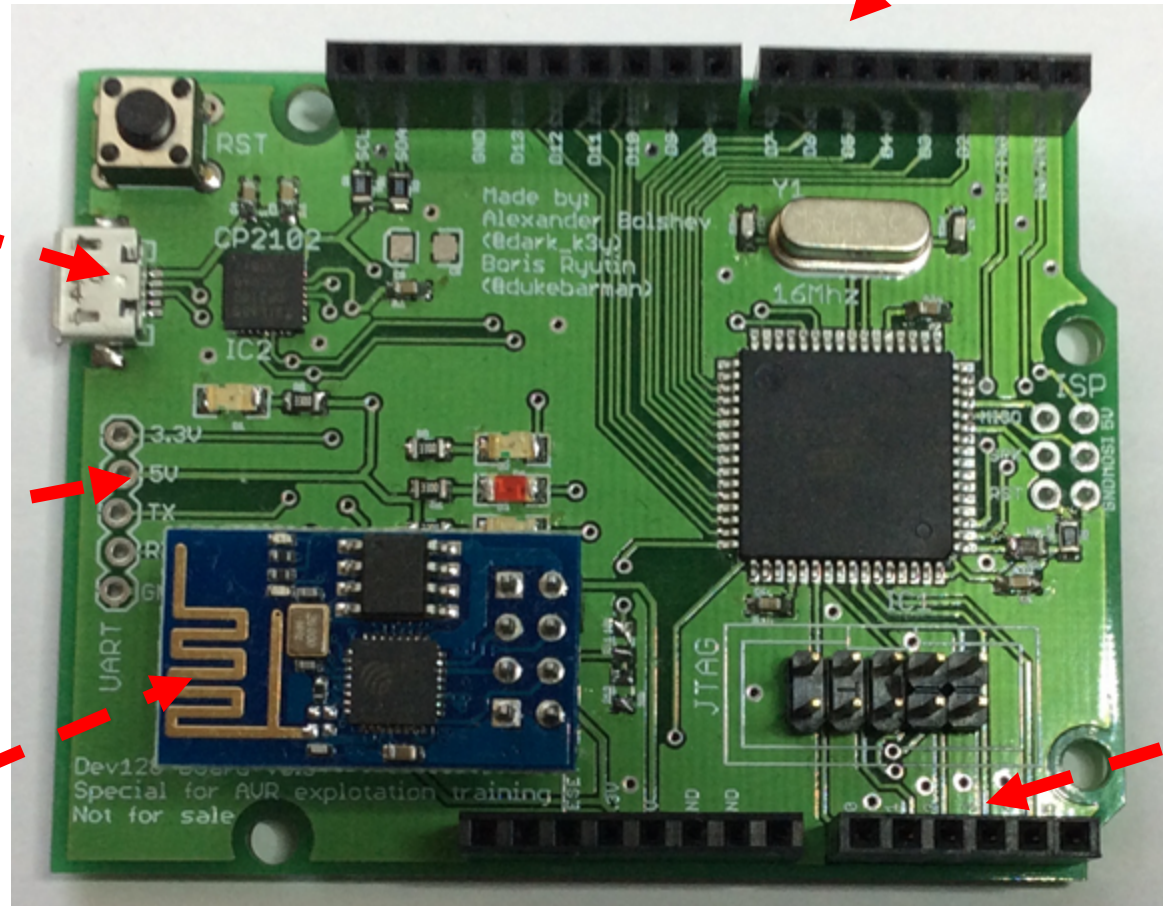
Search for I/O(s)

External connectors

USB

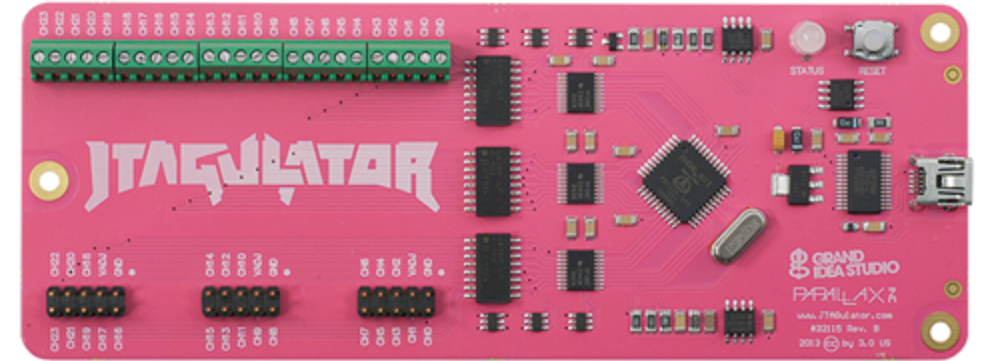
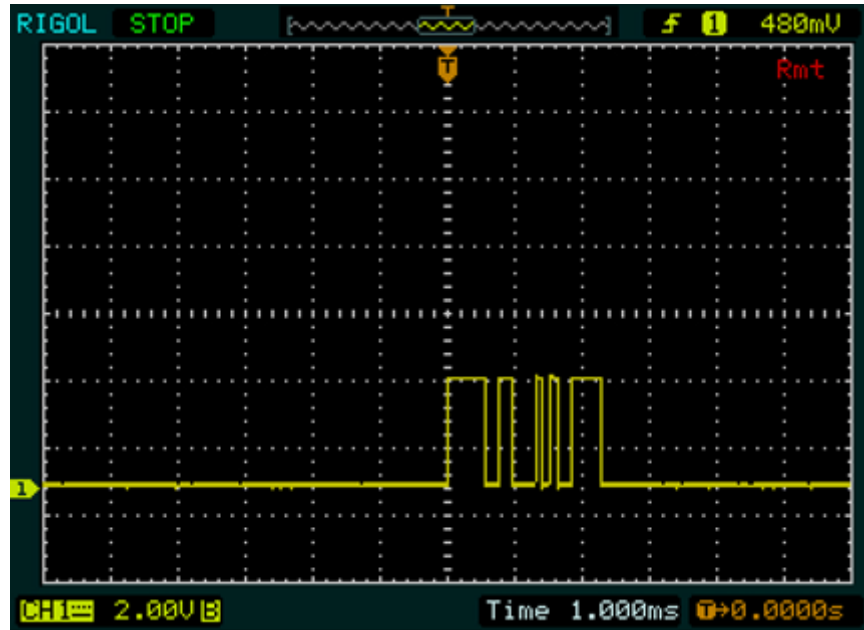
UART

Antenna

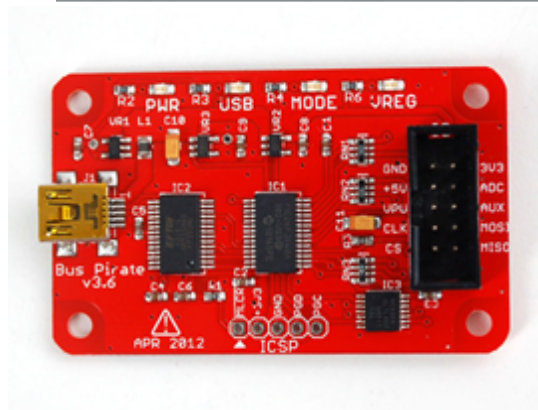


External connectors

Search for I/O(s): tools



Jtagulator



Bus pirate

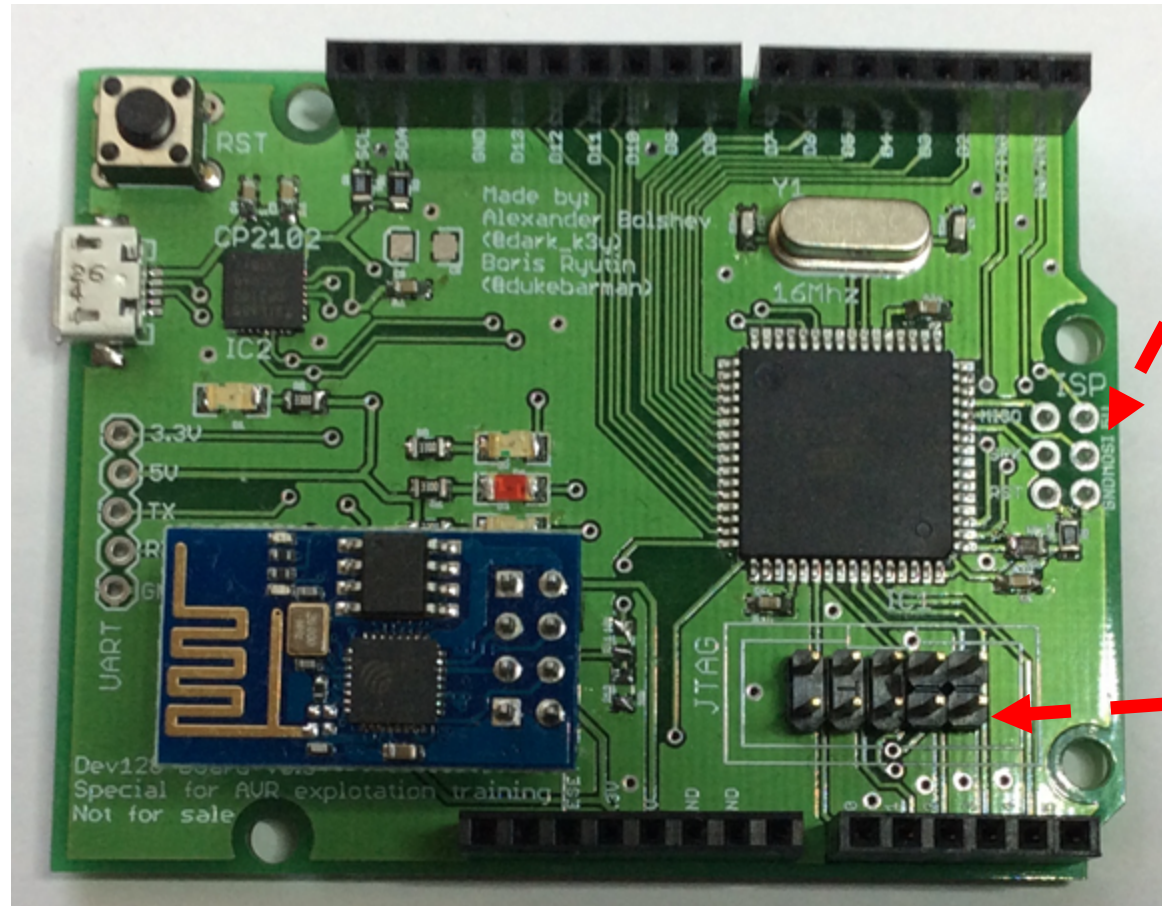


Saleae logic analyzer



Arduino

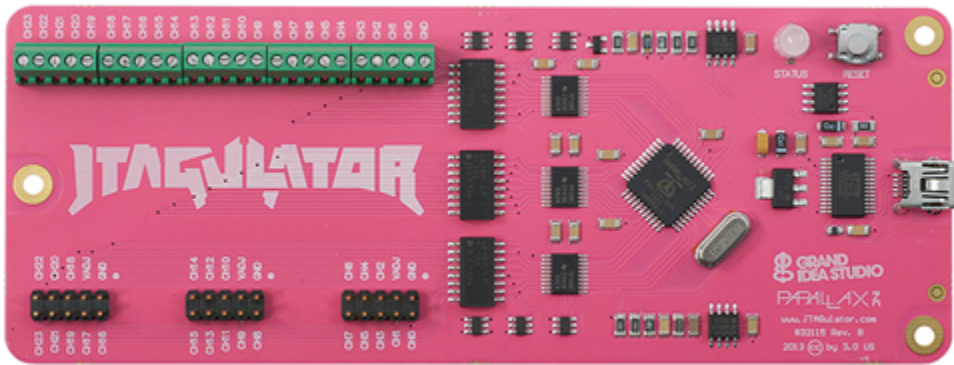
Search for debug interface(s)



ISP

JTAG

Search for debug interface(s): tools



Jtagulator

Or cheaper



Arduino + JTAGEnum

EXAMPLE

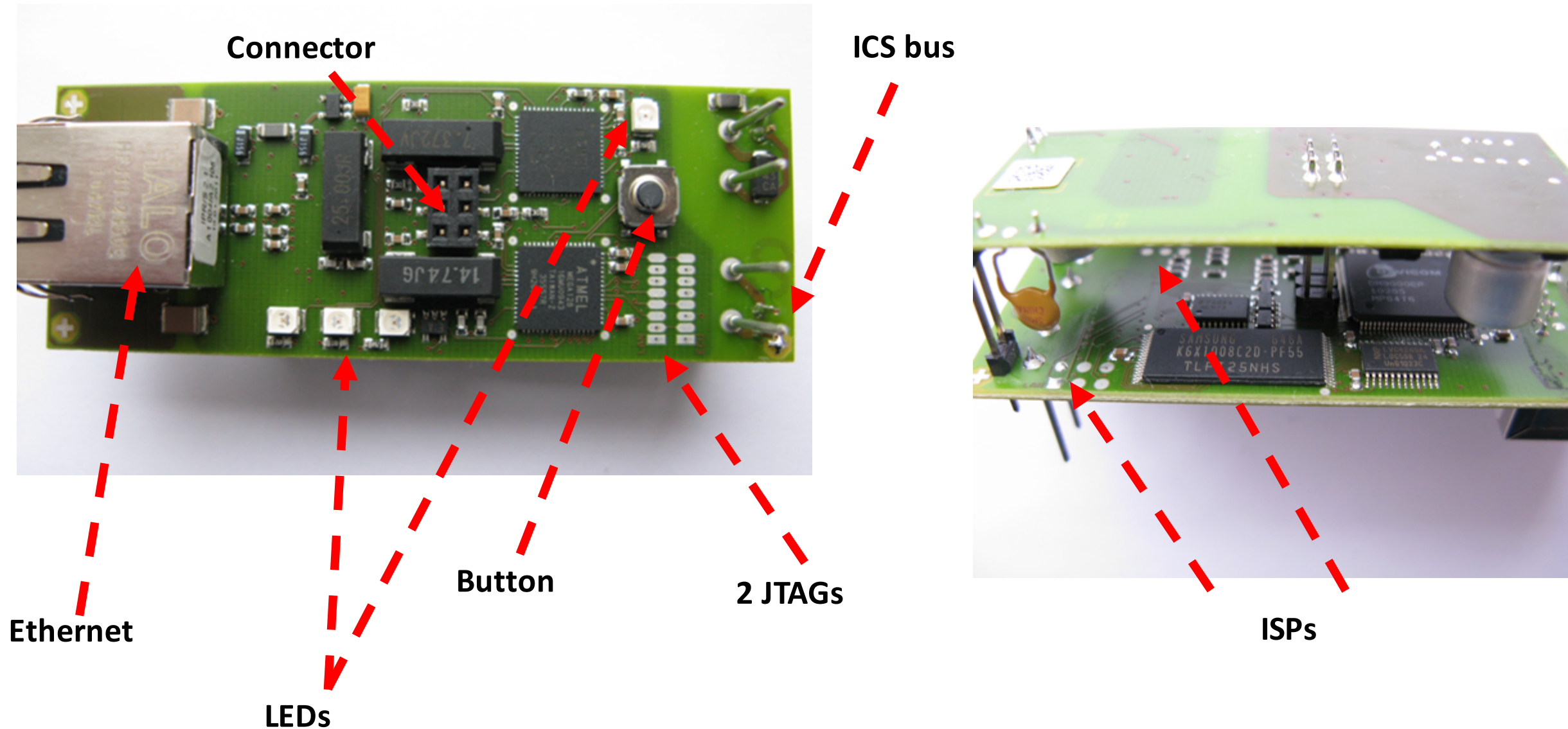
JTAGEnum against Atmega128 demoboard

- Connect Arduino to Atmega128 demoboard
- Connect Arduino to PC with USB cable

```
cd ~/workshop/JTAGenum  
make upload (click reset on arduino just before it)  
screen /dev/ttyACM0 115200
```

- Press “s”

Search for debug & I/O: real device



Acquire the firmware

- From vendor web-site 😊
- Sniffing the firmware update session
- From device itself

Acquiring the firmware: sniff it!

The image shows a Wireshark packet capture interface. The top pane displays a list of packets, with packet 109 selected. The middle pane shows the details of packet 109, including Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and Trivial File Transfer Protocol. The bottom pane shows the raw packet data in hexadecimal and ASCII. A red box highlights a specific region of the raw data, and a red arrow points from this region to a disassembled assembly code block on the right.

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 95 0e 94 .#.#&#7# .....  
0060 a2 a2 18 01 29 01 4f ef 50 e0 60 e0 70 e0 0a 81 .....).0. 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 4f 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
```

Acquiring 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 jtag1 -P /dev/ttyUSB0 \  
-U flash:r:"/home/avr/flash.bin":r
```

Acquiring the firmware: lock bits

- AVR has lock bits that protect device from extracting flash

Memory Lock Bits			Protection Type
Mode	LB1	LB2	
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

- Clearing these lockbits will erase the 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: Fuses

EXAMPLE

Read fuses and lock bits using

```
avarice --mkI --jtag /dev/ttyUSB0 -r -l
```

Firmware reversing: formats

- Raw binary format
- ELF format for AVR
- Intel HEX format (often used by programmers)
- Could be easily converted between with `avr-objcopy`, e.g.:

```
avr-objcopy -I ihex -O binary blink.hex blink.bin
```

AVR RE

Reverse engineering AVR binaries

Pure disassemblers:

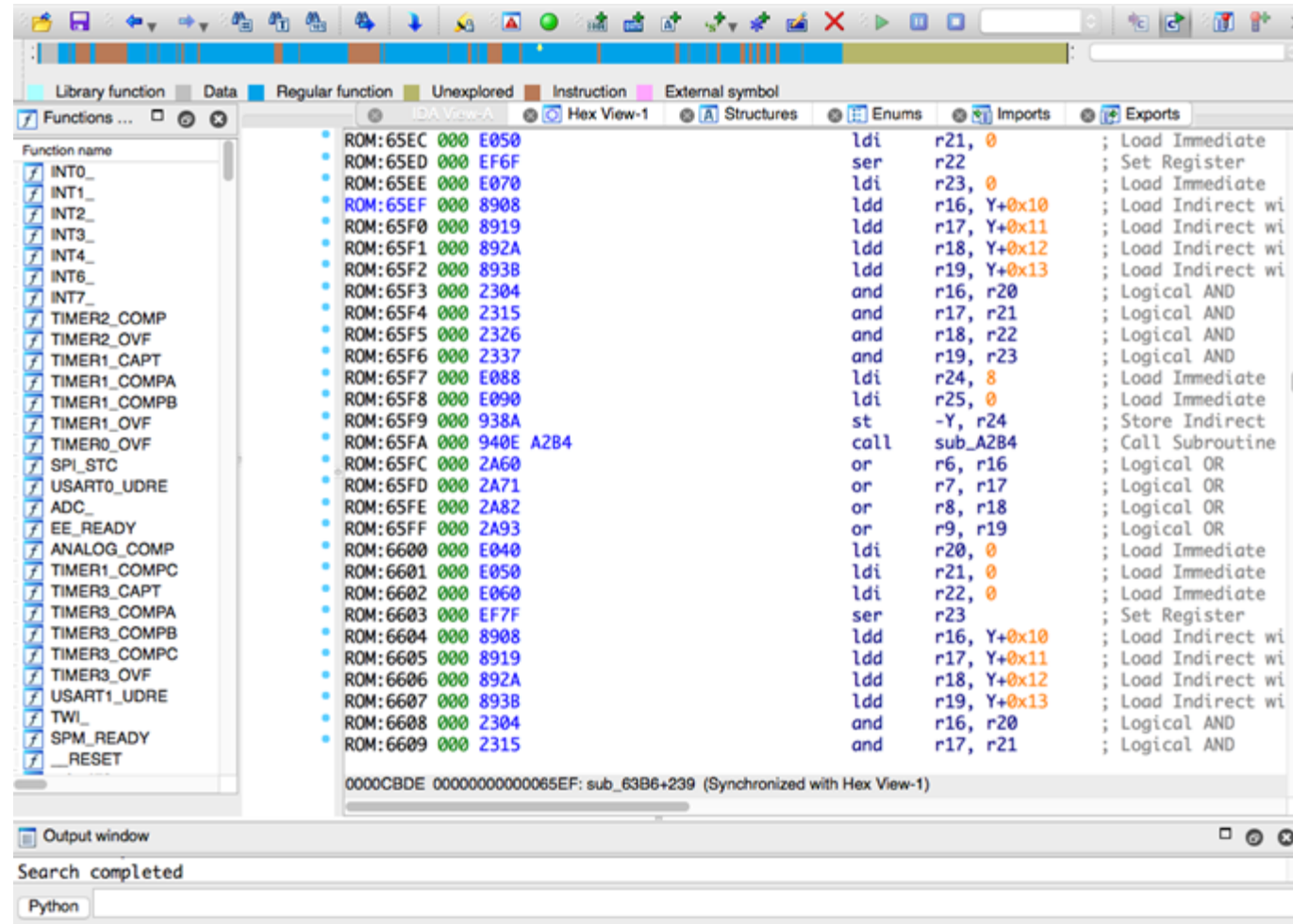
- avr-objdump – gcc kit standard tool
- Vavrdisasm -- <https://github.com/vsergeev/vavrdisasm>
- ODAweb -- <https://www.onlinedisassembler.com/odaweb/>

“Normal” disassemblers:

- IDA Pro
- Radare

IDA PRO: AVR specifics

- Incorrect AVR elf-handling
- Incorrect LPM command behavior
- Addressing issues
- Sometimes strange output
- ...
- Still usable, but “with care”



The screenshot displays the IDA Pro interface for an AVR assembly file. The main window shows a list of instructions with their addresses and mnemonics. The instructions are as follows:

Address	Hex	Mnemonic	Operands	Comment
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

At the bottom of the window, the output window shows the message: "Search completed" and "Python".

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 (c) pancake

```
(Fcn) fcn.00000000 106
0x0000006a 1124      clr r1                ; clear register
0x0000006c 1fbfe     out 0x3f, r1          ; store register to I/O location
0x0000006e cfe       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 program 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        ; subtract immediate from word
0x000000a6 fe01     movw r30, r28         ; copy register word
0x000000a8 0e94a107 call 0xf42            ; 0x00000f42(C) ; 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
- rassign2

Radare2: Usage

- 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/lsc - 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 eca5      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 - disassemble
- pi/pl – 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[bI|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

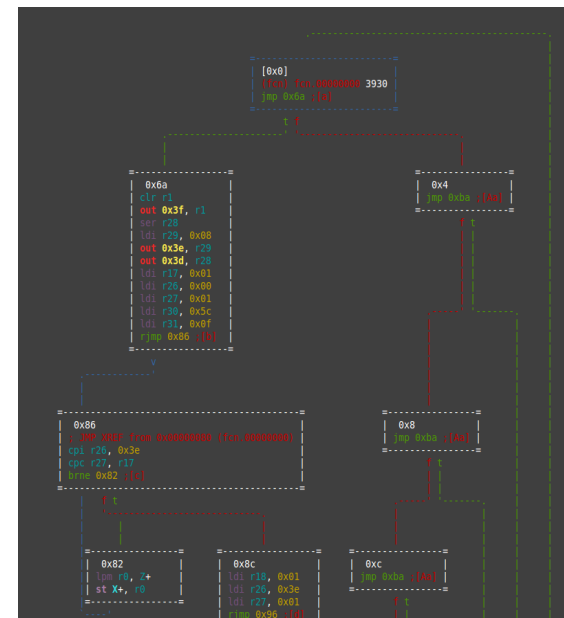
```
[0x0000006a]> pd 35
0x0000006a 1124      clr r1                ; clear register
0x0000006c 1fbc      out 0x3f, r1         ; store register to I/O location
0x0000006e cfef      ser r28              ; set all bits in register
0x00000070 d8e0      ldi r29, 0x08       ; LDI Rd,K. load immediate
0x00000072 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
;K 0x00000080 02c0      rjmp 0x86           ; relative jump
--> 0x00000082 0590      lpm r0, Z+         ; LPM. load programm memory
;K 0x00000084 0d92      st X+, r0          ; ST X,Rr. store indirect
;K ; JMP XREF from 0x00000080 (fcn.00000000)
--> 0x00000086 ae33      cpi r26, 0x3e      ; compare with immediate
0x00000088 b107      cpc r27, r17       ; compare with carry
;K 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
;K 0x00000092 01c0      rjmp 0x96          ; relative jump
--> 0x00000094 1d92      st X+, r1          ; ST X,Rr. store indirect
;K ; JMP XREF from 0x00000092 (fcn.00000000)
--> 0x00000096 a63e      cpi r26, 0xe6      ; compare with immediate
0x00000098 b207      cpc r27, r18       ; compare with carry
;K 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
;K 0x000000a2 04c0      rjmp 0xac          ; relative jump
--> 0x000000a4 2297      sbiw r28, 0x02     ; subtract immediate from word
0x000000a6 fe01      movw r30, r28      ; copy register word
0x000000a8 0e94a107  call 0xf42         ; fcn.00000000(); long call to a subroutine
;K ; JMP XREF from 0x000000a2 (fcn.00000000)
--> 0x000000ac c836      cpi r28, 0x68      ; compare with immediate
0x000000ae d107      cpc r29, r17       ; compare with carry
;K 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

The screenshot shows the Radare2 interface with the disassembly view on the left and the symbols view on the right. The disassembly view shows assembly instructions such as `mov rdi, rsp`, `call 0x7fa70ce3a70`, and `mov r12, rax`. The symbols view lists symbols like `0x0061a610 8 stdout`, `0x0061a5f4 0 _edata`, and `0x0061a640 0 _end`. The stack view shows memory addresses and their contents, and the registers view shows the current state of registers like `r15 0x00000000`, `r14 0x00000000`, and `r13 0x00000000`.

The screenshot shows the Radare2 disassembly view with a list of assembly instructions. The instructions include `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, 0x6f`, `rjmp 0x86`, `lpm r0, Z+`, `st X+, r0`, `JMP XREF from 0x00000080 (fcn.00000080)`, `cpl r26, 0x3e`, `cpc r27, r17`, `brne 0x82`, `ldi r18, 0x01`, `ldi r26, 0x3e`, `ldi r27, 0x01`, `rjmp 0x96`, `st X+, r1`, `JMP XREF from 0x00000092 (fcn.00000090)`, `cpl r26, 0x66`, `cpc r27, r18`, `brne 0x94`, `ldi r17, 0x00`, `ldi r28, 0x0a`, `ldi r29, 0x00`, `rjmp 0x9a`, `sbllw r28, 0x02`, `movw r30, r28`, `call 0xf42`, and `JMP XREF from 0x000000a2 (fcn.00000090)`.



Best combinations for AVR RE

- Both Radare2 and IDA Pro have pitfalls when working with AVR
- That's why I am using the following combination

IDA Pro **6.6+** + Radare2 + GDB + avr-objdump

Here we will focus on Radare2 + GDB, because not everyone can afford latest IDA Pro ☹️

Ex 2.1: Hello! RE

EXAMPLE

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

Now we will scrutinize **every**
line of disassembled code.
Boring, but is required for further
understanding



Interrupts vector && init section

```
==< 0x00000000 0c944600 jmp 0x8c ;[1] ; jump
==< 0x00000004 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000008 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000000c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000010 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000014 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000018 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000001c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000020 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000024 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000028 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000002c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000030 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000034 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000038 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000003c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000040 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000044 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000048 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000004c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000050 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000054 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000058 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000005c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000060 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000064 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000068 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000006c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000070 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000074 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000078 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x0000007c 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000080 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000084 0c945000 jmp 0xa0 ;[2] ; jump
==< 0x00000088 0c945000 jmp 0xa0 ;[2] ; jump

0x0000008c 1124
0x0000008e 1fbe
0x00000090 cfef
0x00000092 d0e1
0x00000094 debf
0x00000096 cdbf
0x00000098 0e945900
0x0000009c 0c947500
0x000000a0 0c940000

;[4] ; jump
;[2] ; jump
;[2] ; jump
;[2] ; jump
;[2] ; jump
;[2] ; jump

0x000000ea f894 cli
-> 0x000000ec ffcf rjmp 0xec
```

Interrupts vector

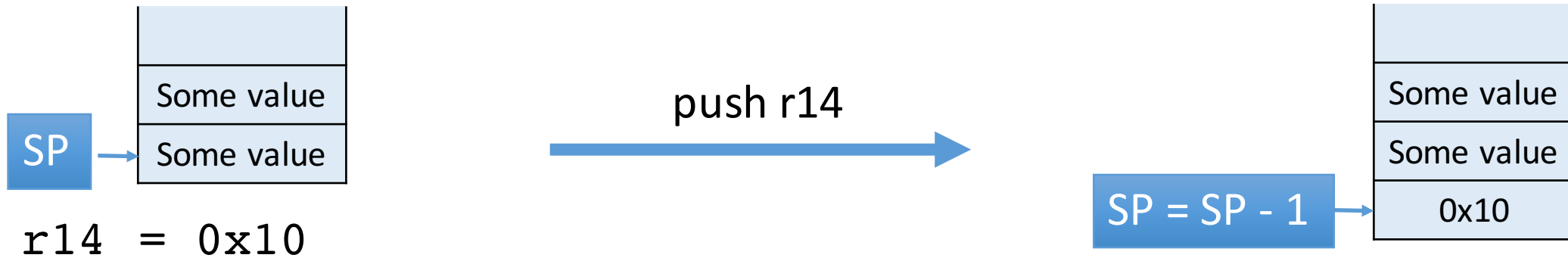
Init section

`call 0xb2 main()`

Program halt

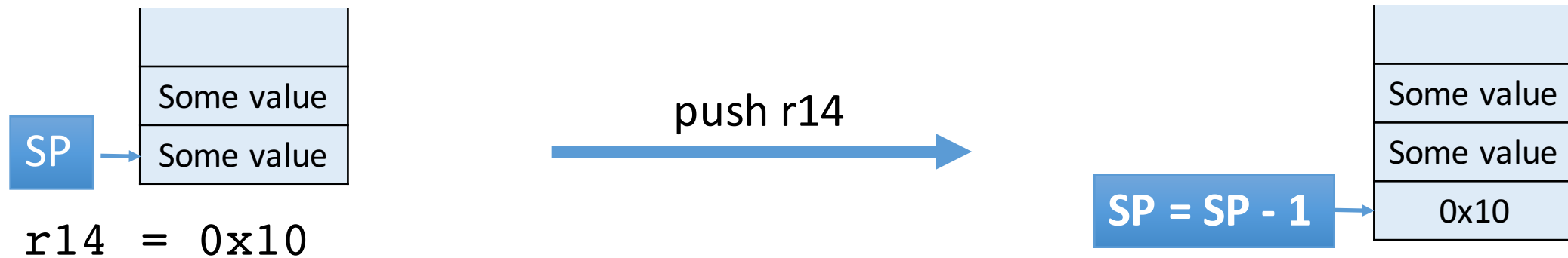
Memory manipulation: stack push

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

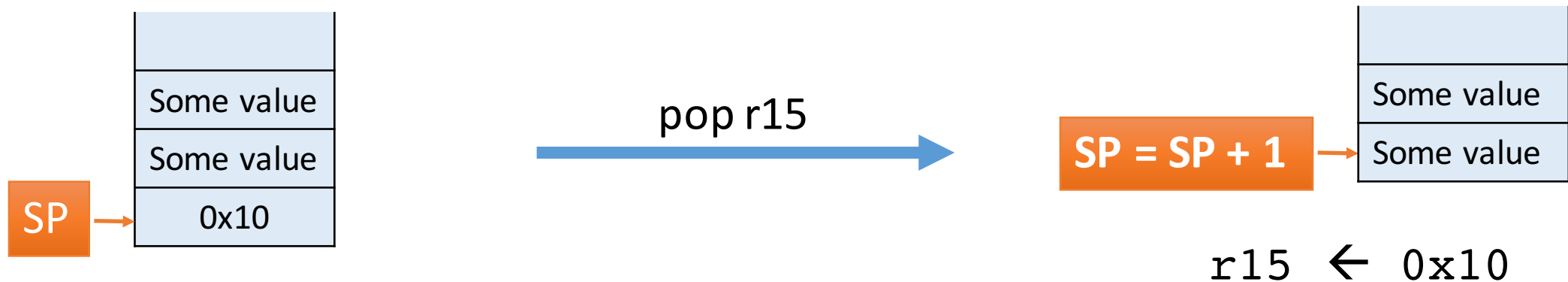


Memory manipulation: stack pop

`push r14 ; save r14 on the Stack`



`pop r15 ; pop top of Stack to r15`



Unconditional jump/call

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

call    0xABC1    ; "push PC+2"
                    ; jmp 0xABC1

ret      ; "pop PC"
```

```
0x00000098  0e945900  call 0xb2
0x0000009c  0c947500  jmp 0xea
0x000000a0  0c940000  jmp 0x0
```

```
0x000000ea  f894  cli
-> 0x000000ec  ffcf  rjmp 0xec
```

Harvard architecture? But PC goes to DATA memory



Arithmetic instructions

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

...

```
0x0000008c  1124  clr r1
0x0000008e  1fbe  out 0x3f, r1
0x00000090  cfef  ser r28
0x00000092  d0e1  ldi r29, 0x10
0x00000094  debf  out 0x3e, r29
0x00000096  cdbf  out 0x3d, r28
0x00000098  0e945900 call 0xb2
```

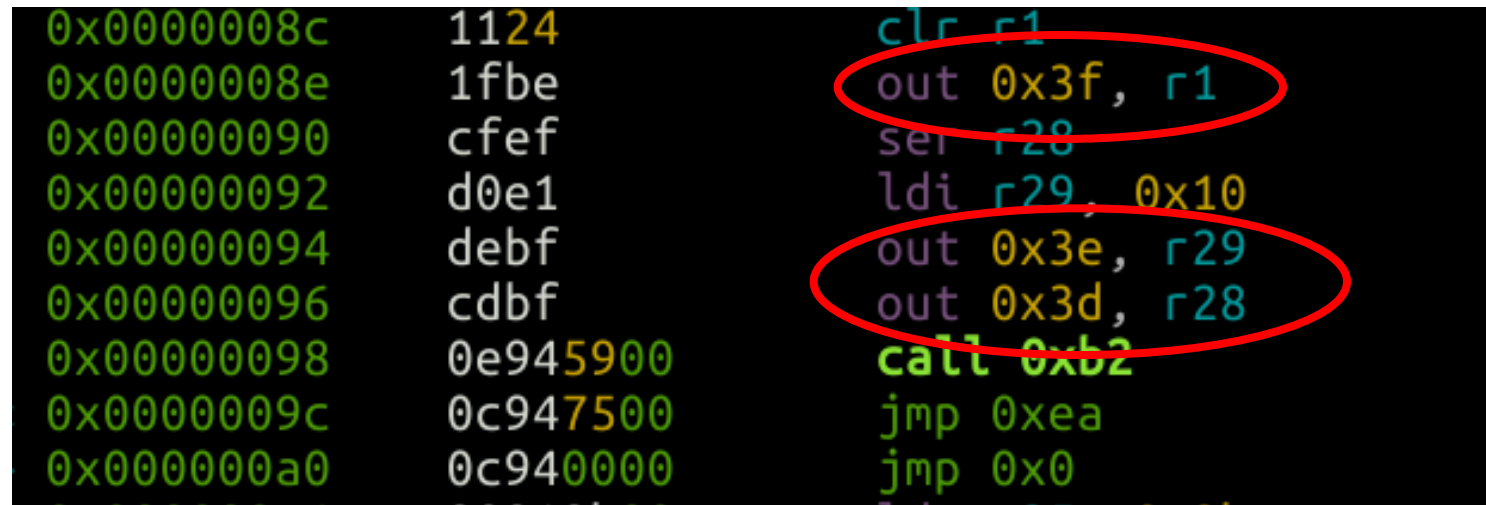
Memory manipulation: immediate values

```
ldi    r29, 0x10    ; r29 = 0x10
```

```
0x0000008c  1124  clr r1
0x0000008e  1fbe  out 0x3f, r1
0x00000090  cfef  ser r28
0x00000092  d0e1  ldi r29, 0x10
0x00000094  debf  out 0x3e, r29
0x00000096  cdbf  out 0x3d, r28
0x00000098  0e945900  call 0xb2
0x0000009c  0c947500  jmp 0xea
0x000000a0  0c940000  jmp 0x0
```

Memory manipulation: ports

```
in      r15, $16      ; r15 = PORTB
out     $16, r0       ; PORTB = r0
```



```
0x0000008c 1124      clr r1
0x0000008e 1fbe      out 0x3f, r1
0x00000090 cfef      ser r28
0x00000092 d0e1      ldi r29, 0x10
0x00000094 debf      out 0x3e, r29
0x00000096 cdbf      out 0x3d, r28
0x00000098 0e945900 call 0xb2
0x0000009c 0c947500 jmp 0xea
0x000000a0 0c940000 jmp 0x0
```

The screenshot shows assembly code with three instructions circled in red: `out 0x3f, r1`, `out 0x3e, r29`, and `out 0x3d, r28`. The code also includes `clr r1`, `ser r28`, `ldi r29, 0x10`, `call 0xb2`, `jmp 0xea`, and `jmp 0x0`.

What is the 0x3f, 0x3e, 0x3d and where to find them?

Datasheets are your best friends! (2)

P. 366 of Atmega128L datasheet



Register Summary (Continued)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(\$61)	DDRF	DDF7	DDF6	DDF5	DDF4	DDF3	DDF2	DDF1	DDF0	88
(\$60)	Reserved	-	-	-	-	-	-	-	-	
\$3F (\$5F)	SREG	I	T	H	S	V	N	Z	C	11
\$3E (\$5E)	SPH	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	14
\$3D (\$5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	14
\$3C (\$5C)	XDIV	XDIVEN	XDIV6	XDIV5	XDIV4	XDIV3	XDIV2	XDIV1	XDIV0	43
\$3B (\$5B)	RAMPZ	-	-	-	-	-	-	-	RAMPZ0	14
\$3A (\$5A)	EICRB	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	90
\$39 (\$59)	EIMSK	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	91
\$38 (\$58)	EIFR	INTF7	INTF6	INTF5	INTF4	INTF3	INTF	INTF1	INTF0	91
\$37 (\$57)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	108, 140, 160
\$36 (\$56)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	108, 141, 161
\$35 (\$55)	MCUCR	SRE	SRW10	SE	SM1	SM0	SM2	IVSEL	IVCE	31, 44, 63
\$34 (\$54)	MCUCSR	JTD	-	-	JTRF	WDRF	BORF	EXTRF	PORF	53, 257
\$33 (\$53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	103
\$32 (\$52)	TCNT0	Timer/Counter0 (8 Bit)								105

So, what's going on here?

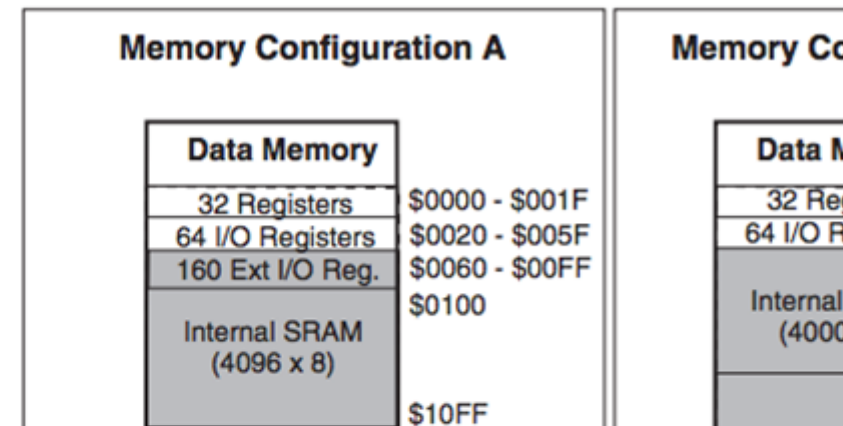
```
0x0000008c    1124    clr r1
0x0000008e    1fbe    out 0x3f, r1 ①
0x00000090    cfef    ser r28
0x00000092    d0e1    ldi r29, 0x10
0x00000094    debf    out 0x3e, r29 ②
0x00000096    cdbf    out 0x3d, r28 ③
0x00000098    0e945900 call 0xb2
0x0000009c    0c947500 jmp 0xea
0x000000a0    0c940000 jmp 0x0
```

1. SREG (Status REGISTER) is cleared (set to r1 value, which is 0x00)
2. SPL \leftarrow r28 (0xFF)
3. SPH \leftarrow r29 (0x10)

After init:

SREG = 0, SP = 0x10FF = 4351(SRAM limit)

Figure 9. Data Memory Map



Going further

```
0x000000b2    10929800    sts 0x98, r1
0x000000b6    87e6        ldi r24, 0x67
0x000000b8    80939900    sts 0x99, r24
0x000000bc    80919a00    lds r24, 0x9a
0x000000c0    8869        ori r24, 0x98
0x000000c2    80939a00    sts 0x9a, r24
0x000000c6    80919d00    lds r24, 0x9d
0x000000ca    8e60        ori r24, 0x0e
0x000000cc    80939d00    sts 0x9d, r24
.-> 0x000000d0    88e4        ldi r24, 0x48
| 0x000000d2    0e945200    call 0xa4
| 0x000000d6    2fef        ser r18
| 0x000000d8    80e7        ldi r24, 0x70
| 0x000000da    92e0        ldi r25, 0x02
.--> 0x000000dc    2150        subi r18, 0x01
|| 0x000000de    8040        sbci r24, 0x00
|| 0x000000e0    9040        sbci r25, 0x00
`==< 0x000000e2    e1f7        brne 0xdc
,==< 0x000000e4    00c0        rjmp 0xe6
`--> 0x000000e6    0000        nop
`= < 0x000000e8    f3cf        rjmp 0xd0
```

Memory manipulation: lds/sts

```
lds      r2,0xFA00      ; r2 = *0xFA00
sts      0xFA00,r0      ; *0xFA00 = r0
```

```
0x000000b2  10929800  sts 0x98, r1
0x000000b6  87e6      ldi r24, 0x67
0x000000b8  80939900  sts 0x99, r24
```

Here:

1. $*0x98 \leftarrow r1$ (0x00)
2. $r24 \leftarrow 0x67$
3. $*0x99 \leftarrow r24$ (0x67)

Datasheets are your best friends! (3)

P. 365 of Atmega128L datasheet



Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(\$FF)	Reserved	-	-	-	-	-	-	-	-	
..	Reserved	-	-	-	-	-	-	-	-	
(\$9E)	Reserved	-	-	-	-	-	-	-	-	
(\$9D)	UCSR1C	-	UMSEL1	UPM11	UPM10	USBS1	UCSZ11	UCSZ10	UCPOL1	192
(\$9C)	UDR1	USART1 I/O Data Register								190
(\$9B)	UCSR1A	RXC1	TXC1	UDRE1	FE1	DOR1	UPE1	U2X1	MPCM1	190
(\$9A)	UCSR1B	RXCIE1	TXCIE1	UDRIE1	RXEN1	TXEN1	UCSZ12	RXB81	TXB81	191
(\$99)	UBRR1L	USART1 Baud Rate Register Low								194
(\$98)	UBRR1H	-	-	-	-	USART1 Baud Rate Register High				194

What is it all about and why `sts` and not `out`?

```
UBRR1H = ( BAUD_PRESCALE >> 8 );
```

```
    10929800      sts      0x0098, r1
```

```
UBRR1L = ( BAUD_PRESCALE );
```

```
    87e6         ldi      r24, 0x67
```

```
    80939900      sts      0x0099, r24
```

- Registers are also part of the RAM
- Common rule:
 - Every IO/RAM address is reachable with `sts/lds` while `in/out` are used for (0x00 - 0x3F range)

Why 0x0067?

- USART is clocking from internal generator (16MHz in our case)
- We selected baud speed of 9600
- The common formula of USART frequency divider for AVR (see datasheet for USART section, p.194+):

$$\begin{aligned} \text{BAUD_PRESCALE} &= (\text{F_CPU} / (\text{USART_BAUDRATE} * 16)) - 1 = \\ &= 16\,000\,000 / (9600 * 16) - 1 = 104.1666666(6) - 1 \approx 103 = 0x0067 \end{aligned}$$

More arithmetic instructions

```
andi          r2, 0x10          ; r2 = r2 & 0x10
ori           r24, 0x98         ; r24 = r24 | 0x98
```

...

```
UCSR1B |= (1 << RXEN1) | (1 << TXEN1) | (1 << RXCIE1);
```

```
0x000000bc 80919a00 lds r24, 0x9a
0x000000c0 8869 ori r24, 0x98
0x000000c2 80939a00 sts 0x9a, r24
```

Enables RX and TX lines, enable RX interrupt.

```
UCSR1C |= (1 << USBS1) | (3 << UCSZ10);
```

```
0x000000c6 80919d00 lds r24, 0x9d
0x000000ca 8e60 ori r24, 0x0e
0x000000cc 80939d00 sts 0x9d, r24
```

Set stop bit and character size

Functions & Calling conventions

```
send_byte('H');
```

```
0x000000d0      88e4      ldi r24, 0x48  
0x000000d2      0e945200   call 0xa4
```

Typical AVR calling convention for arguments

- Call-used: **R18–R27, R30, R31**
- Call-saved: **R2–R17, R28, R29**
- **R29:R28** used as frame pointer

We will discuss it in more details later.


```
_delay_ms(1000);
```

This “function” is inlined as:

```
0x000000d6  2fef  ser r18
0x000000d8  80e7  ldi r24, 0x70
0x000000da  92e0  ldi r25, 0x02
0x000000dc  2150  subi r18, 0x01
0x000000de  8040  sbci r24, 0x00
0x000000e0  9040  sbci r25, 0x00
0x000000e2  e1f7  brne 0xdc
0x000000e4  00c0  rjmp 0xe6
0x000000e6  0000  nop
```

```
subi    r18, 0x01    ; r18 = r18 - 1
```

```
sbci    r24, 0x00    ; r24 = r24 - 0 - C
```

```
; C – Carry flag from arithmetic operations (SREG)
```

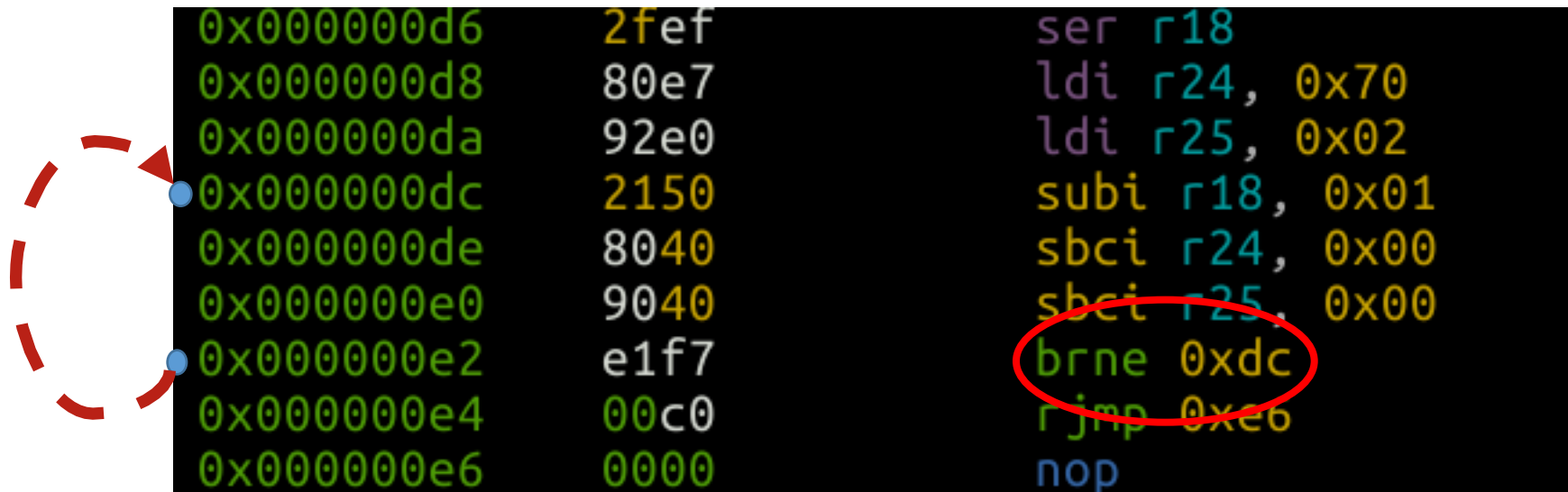
Conditional jump

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

```
breq    10        ; Z ? PC ← PC + 1 + 10
```

```
brne    -4        ; !Z ? PC ← PC + 1 - 4
```

...



The screenshot shows a list of assembly instructions with their addresses and hex values. The instruction `brne 0xdc` at address `0x000000e2` is circled in red. A dashed red arrow on the left indicates a jump from the `brne` instruction back to the `subi r18, 0x01` instruction at address `0x000000dc`.

```
0x000000d6  2fef  ser r18  
0x000000d8  80e7  ldi r24, 0x70  
0x000000da  92e0  ldi r25, 0x02  
0x000000dc  2150  subi r18, 0x01  
0x000000de  8040  sbci r24, 0x00  
0x000000e0  9040  sbci r25, 0x00  
0x000000e2  e1f7  brne 0xdc  
0x000000e4  00c0  rjmp 0xe6  
0x000000e6  0000  nop
```

Why -4?



Operation:

- (i) If $R_d \neq R_r$ ($Z = 0$) then $PC \leftarrow PC + k + 1$, else $PC \leftarrow PC + 1$

Syntax:

- (i) **BRNE** k

Operands:

$$-64 \leq k \leq +63$$

Program Counter:

$$PC \leftarrow PC + k + 1$$

$$PC \leftarrow PC + 1, \text{ if condition is false}$$

16-bit Opcode:

1111	01kk	kkkk	k001
------	------	------	------

$$f7e1 = 1111 \ 0111 \ 1110 \ 0001$$

11 1110 0 in two's complement form == -4

Special

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

```
void send_byte(uint8_t byte)
```

```
while((UCSR1A &(1<<UDRE1)) == 0);
```

```
.-> 0x000000a4    90919b00    lds r25, 0x9b  
|   0x000000a8    95ff       sbrs r25, 5  
'=< 0x000000aa    fccf       rjmp 0xa4
```

```
UDR1 = byte;
```

```
0x000000ac    80939c00    sts 0x9c, r24  
0x000000b0    0895       ret
```

Conditional “skip”

```
sbrc r0, 7    ; skip if bit 7 in r0 cleared
```

```
cpse r4,r0    ; skip if r4 == r0
```

```
sbrs r25,5    ; skip if bit 5 in r25 set
```

```
.-> 0x000000a4    90919b00    lds r25, 0x9b
|      0x000000a8    95ff        sbrs r25, 5
`=< 0x000000aa    fccf        rjmp 0xa4
```

More things to know

Comparison

```
cp      r4,r19    ; Compare r4 with r19  
brne   label1    ; jump if r19 != r4
```

```
; Compare r3:r2 with r1:r0  
cp     r2,r0      ; Compare low byte  
cpc   r3,r1      ; Compare high byte  
brne   label2    ; jump if r3:r2 != r1:r0
```

```
cpi    r19,3     ; Compare r19 with 3  
brne   label3    ; jump if r19 != 3
```


SREG – 8-bit status register

C – **C**arry flag

Z – **Z**ero flag

N – **N**egative flag

V – two's complement **o**Verflow indicator

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

H – **H**alf carry flag

T – **T**ransfer bit (BLD/BST)

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

SREG manipulations

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

More memory manipulation

```
mov      r1, r2      ; r1 = r2
st       Z, r0       ; *Z(r31:r30) = r0
st       -Z, r1      ; *Z-- = r1
std      Z+5, r2     ; *(Z+5) = r2
...
```

Direct

Indirect

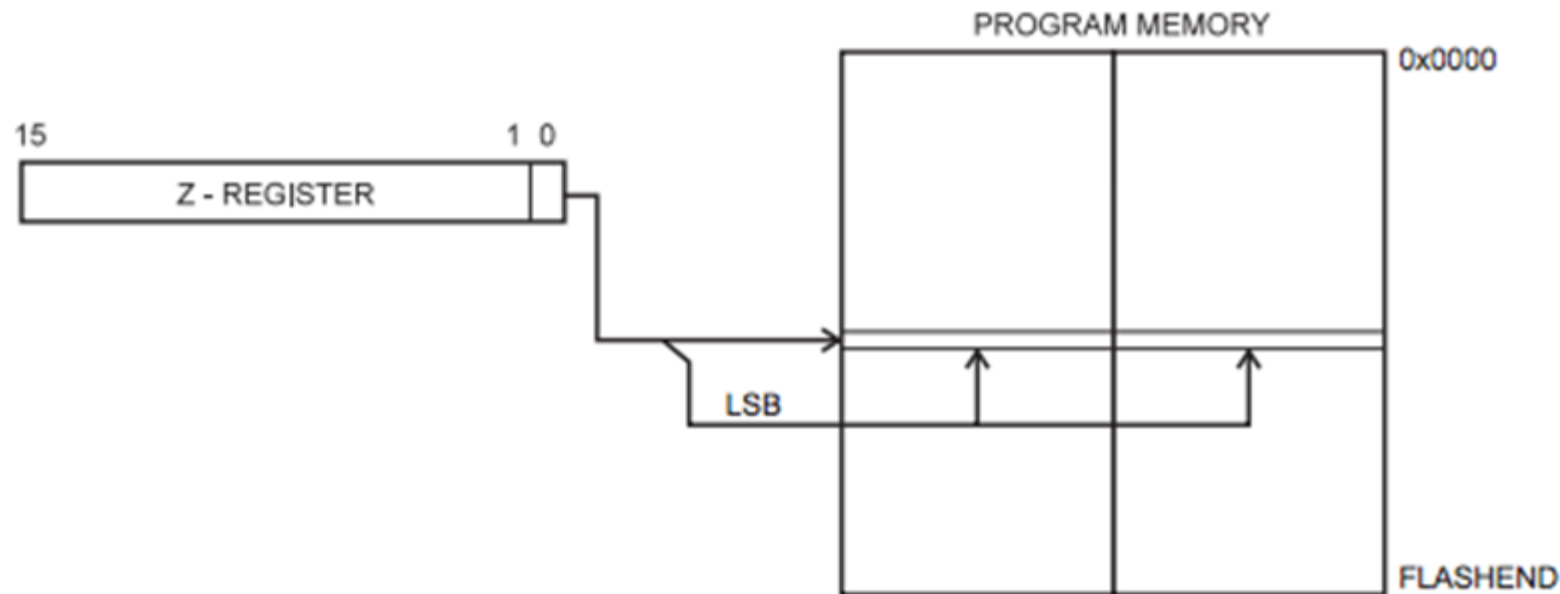


Same
for LD*

Memory manipulation: flash

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

Figure 2-9. Program Memory Constant Addressing



Note: code is separated from data

Bit manipulation instructions

```
sbr    r16, 3    ; set bits 0 and 1 in r16
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
cbi    $16, 1    ; PORTB[1] = 0
```

Ex 2.2: Blink! RE

EXERCISE

```
cd /home/radare/workshop/ex2.2
avr-objcopy -I ihex -O binary blink.hex blink.bin
r2 -a avr blink.bin
```

Questions:

1. Identify main() function, define and rename it
2. Find the LED switching command
3. What type of delay is used and why accuracy of MCU frequency important?
4. Locate interrupt vector and init code, explain what happens inside init code

Reversing: function **s**ignatures

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

DEMO

Working with zignatures

Embedded code priorities

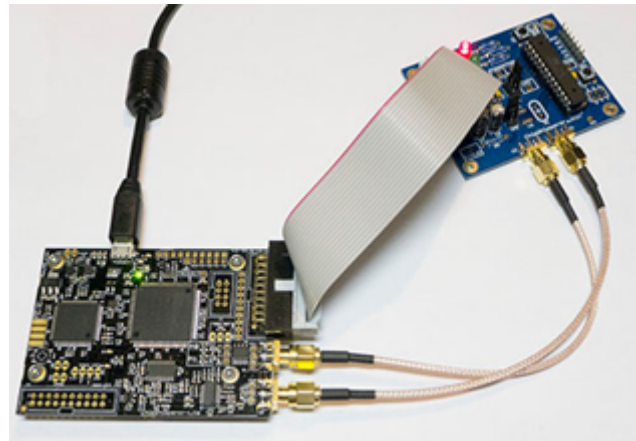
- Size
- Speed
- Hardware limits
- Redundancy
- ...
- ...
- ...
- ...
- Security

Fuzzing specifics

- Fuzzing is a fuzzing. Everywhere.
- But... we're in embedded world
- Sometimes you **can** detect crash through test/debug UART or pins
- In most cases, you can detect crash only by noticing that device is no longer response
- Moreover, **watchdog timer** can limit your detection capabilities by resetting the 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

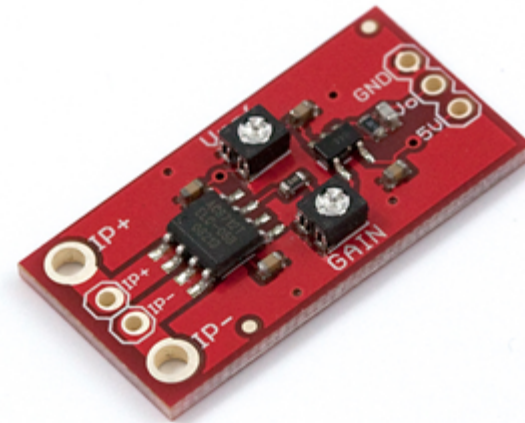


Sometimes Arduino is enough to detect

- I²C and SPI init sequences could be captured by Arduino GPIOs
- In case bootloader is slow and has ~1 second loading delay, 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

- **R**eturn **O**riented **P**rogramming
- Series of function returns
- We are searching for primitives (“gadgets”) ending with **‘ret’** that could be chained into a useful code sequence
- SP is our new PC

Notice: Arduino

- The next examples/exercises will be based upon Arduino 'libc' (in fact, Non-GNU AVR libc + Arduino wiring libs)
- We're using Arduino because it is sufficiently complex, full of gadgets and free (vs. 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" (and quick bugs)



Ex 3.1 – 3.3

```
cd /home/radare/workshop/ex3.1  
avrice --mkI --jtag /dev/ttyUSB0 -p -e --file build-  
crumbuino128/ex3.1.hex -d :4242
```

In the new terminal window:

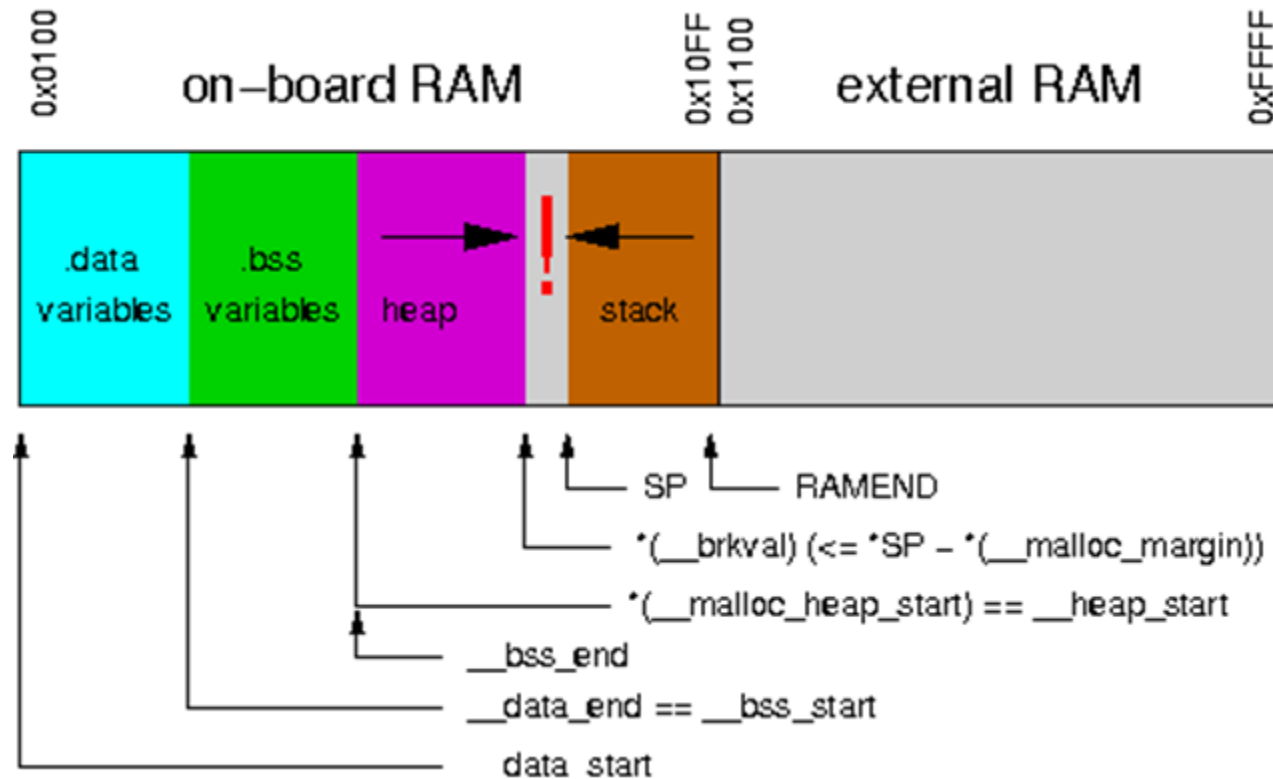
```
avr-gdb  
(gdb) target remote :4242
```

EXAMPLE

Example 3.1

Abusing functionality: ret to
function

Internal-SRAM only memory map



Overflowing the heap => Rewriting the stack!

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

Standard model: with .data variables

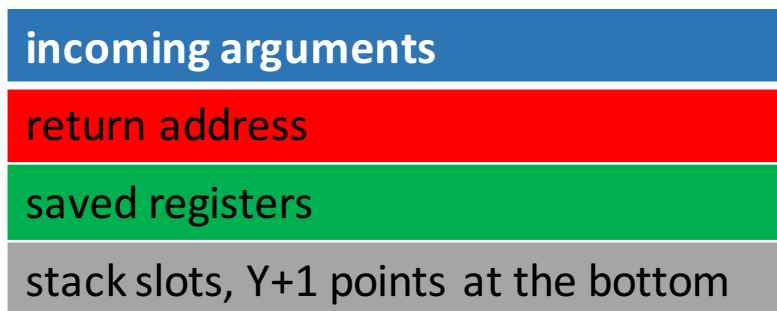
- Determine data offset in flash
- Find init code/firmware prologue where .data is copied to SRAM
- Using debugging or own brain calculate offset of data in SRAM
- Search code for this address

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 the 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 number 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.
- Varargs are passed on the stack

Calling conventions: returns

- Return values of size 1 byte up to 8 bytes (including) 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

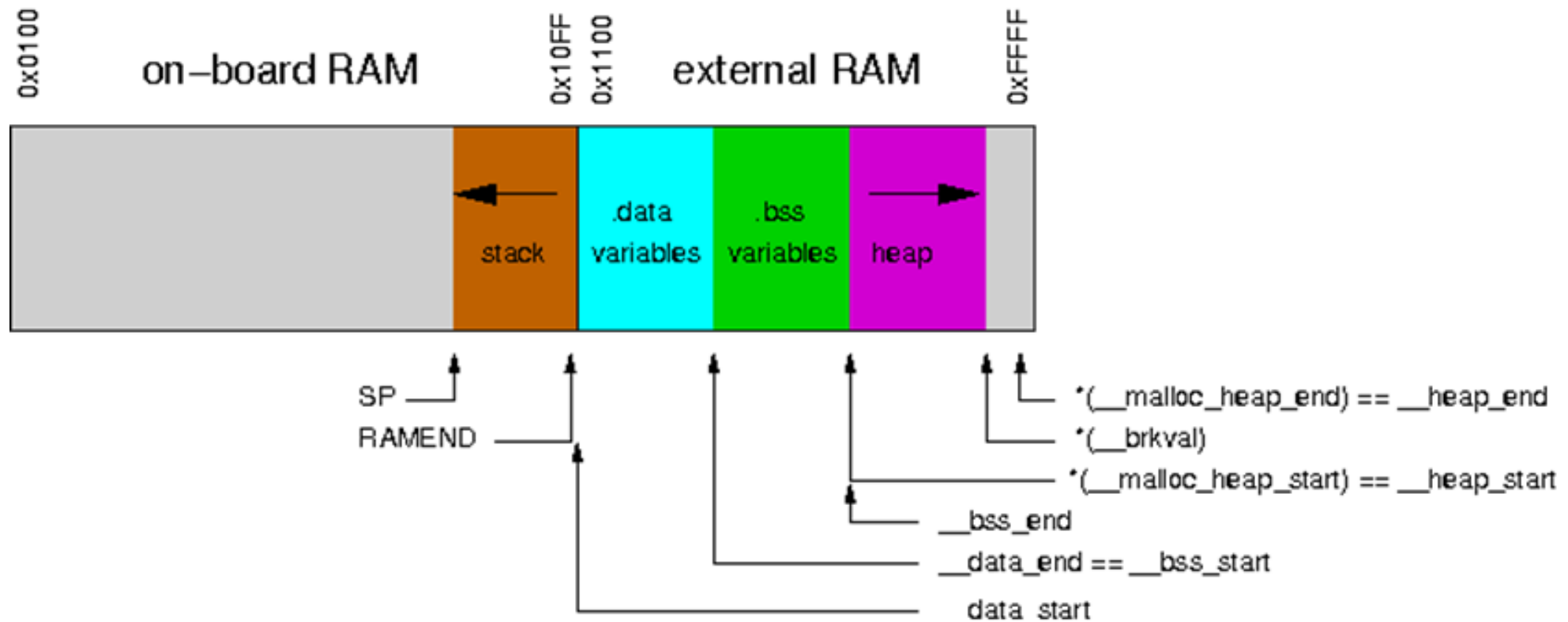
- User functions
- “Standard” or RTOS functions
- Data segment 😊
- Bootloader section

More code => more gadgets

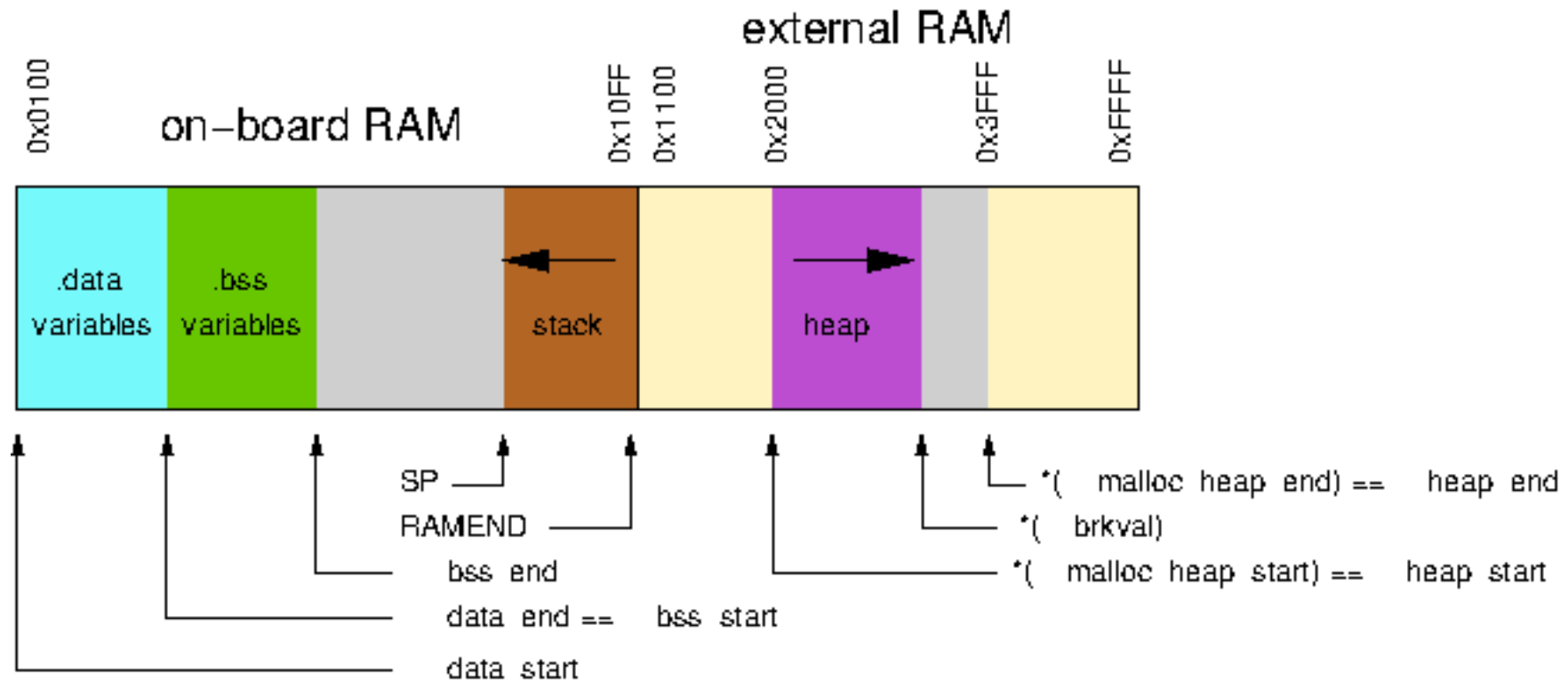
ROP chain size

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

Memory maps – external SRAM/separated stack



Memory maps – external SRAM/mixed stack



Detecting “standard” functions

- In AVR world there are a lot of different compilers, libraries and even RToSes
- Thus, “standard” function could vary
- More bad news: memory model and optimization options can change function
- The best approach is to try to detect functions like malloc/str(n)cpy and then find the exact compiler/options that generates this code
- After that, 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

```
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
```

```
avrice --mkI --jtag /dev/ttyUSB0 -p -e --file build-  
crumbuino128/ex3.4.hex -g :4242
```

```
avr-gdb
```

- Connect to WiFi “esp_<your3digitnumber>” (password: 1234567890) and type <http://192.168.4.1> in your browser

EXAMPLE

Example 3.4
Blink using HTTP GET

EXERCISE

Exercise 3.4

UARTing using HTTP query

EXERCISE

Exercise 3.5

Blink using HTTP Post

(homework)

It is possible to construct ROP with a debugger...
...But if you don't have one, how could you determine the overflow point?

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

EXAMPLE

Arduino blink (ROP without debugger)

- Connect Arduino board using MicroUSB cable

```
cd /home/radare/workshop/ex_arduino  
make upload (click reset on arduino just before it)
```

- Run cutecom and connect to /dev/ttyACM0 using speed 9600

EXERCISE

Arduino blink (ROP without debugger)

Modify ROP chain to generate another blinking pattern

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 over long time

Evade the watchdog

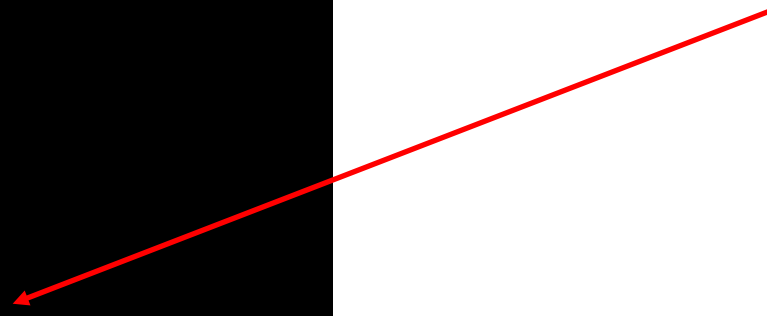
In most cases, there are three ways:

1. Find a ROP with **WDR** and periodically jump on it
2. Find watchdog disabling code and try to jump on it
3. Construct watchdog disabling code using watchdog enabling code



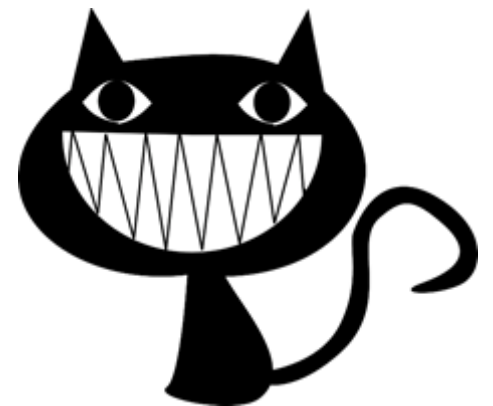
```
0fb6      in r0, 0x3f
f894      cli
a895      wdr
81bd      out 0x21, r24
0fbe      out 0x3f, r0
21bd      out 0x21, r18
0895      ret
0e945900  call 0xb2
```

Set r18 to 0 and JMP here



Fun and scary things to do with memory...

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



Reading EEPROM/Flash

- In most cases it is easy to find gadget(s) that reads byte from EEPROM or flash and stores it somewhere
- We could send this byte 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 which has enough gadgets
- Modern bootloaders are large and you may be lucky quite often (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 coding/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` (but 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 modern MCUs (more secure against various hardware attacks)
- Use external safety controls/processors

And last, but not least:

- Beware of Dmitry Nedospasov ;-)

Conclusions

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

Books/links

- 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/>

