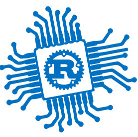
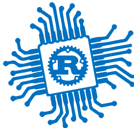


Introduction

Lecture 1

Copyright © Politehnica Bucharest and Wyliodrin SRL 2025, licensed under CC BY-SA 4.0.



Welcome

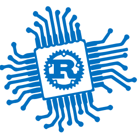
to the *Microprocessor Architecture* engineering class

You will learn

- how hardware works
- how to actually build your own hardware device
- the Rust programming Language

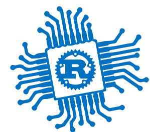
We expect

- to come to class
- ask a lot of questions



Team

Our team

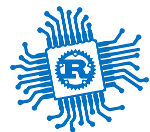


Lectures

- Alexandru Radovici

Labs

- Andrei Zamfir
- Dănuț Aldea
- Irina Bradu
- Genan Omer
- Cristiana Precup
- Eva Cosma
- Victor Lişman
- Roi Bachynskyi



Outline

Lectures

- 12 lectures
- 1 Q&A lecture for the project

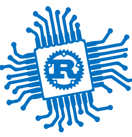
Labs + Project Support

- 12 labs

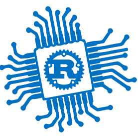
Project

- Build a hardware device running software written in Rust
- Presented at PM Fair during the last week of the semester

Grading

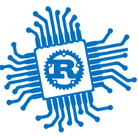


Part	Description	Points
Lecture tests	You will have a test at every class with subjects from the previous class.	0.2p
Final Lecture test	You will have a test during one of the lectures in May.	1.8p
Lab	Your **presence** at every lab will be graded.	1p
Lab Test	A final test at the lab - will scale your lab grade	1p
Project	You will have to design and implement a hardware device. Grading will be done for the documentation, hardware design and software development.	3p
Final Test	You will have to take an exam during the last week of the semester.	4p
Total	*You will need at least 4.5 points to pass the subject.*	**11p**



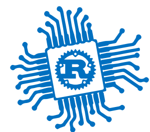
Subjects

Theory

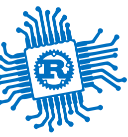


- How a microprocessor works
- How the ARM Cortex-M processor works
- Using digital signals to control devices
- Using analog signals to read data from sensors
- How interrupts work
- How asynchronous programming works (async/await)
- How embedded operating systems work

Practical

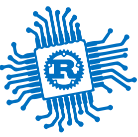


- How to use the STM32 Nucleo-U545RE-Q
  - Affordable
  - Powerful processor
  - Good documentation
- How to program in Rust
  - Memory Safe
  - *Java-like features, without Java's penalties*
  - Defines an embedded standard interface *embedded-hal*

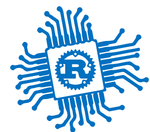


Why Rust

Let’s see some code



```c
#include <stdio.h>
#include <stdint.h>

void printBinary(uint32_t num) {
    for (int i = 31; i >= 0; i--) {
        printf("%d", (num >> i) & 1);
        if (i % 8 == 0) printf(" ");
    }
    printf("\n");
}

int main()
{
    uint8_t a;
    uint32_t b;

    a = 0x01;
    b = a << 24;

    printBinary(a);
    printBinary(b);

    return 0;
}
```



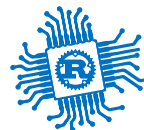
Variables in C

```c
#include <stdio.h>

int8_t, uint8_t
int16_t, uint16_t
int32_t, uint32_t
```

Variables in Rust

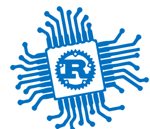
```rust
u8, u16, u32, u64, u128
i8, i16, i32, i64, i128
usize //word size (eg - 32b for 32b processor)
isize //word size (eg - 32b for 32b processor)

//NOTES:
char // 4 bytes != u8 //UTF-8 not ASCII like in C
b"str" //ASCII string
"str" UTF-8 string

's' // char
b's' // u8
```

Lets see how C++ is doing



Link	Memory-safety issue	Why Rust prevents it
ZDI-24-854	Unchecked AES-key length copied into fixed stack buffer => overflow enables network-adjacent remote code execution.	Safe Rust enforces bounds checks; unchecked stack copies aren’t possible.
Toyota: single bit flip	Stack overflow/memory corruption can kill RTOS task, bypass fail-safes => loss of throttle control (unintended acceleration).	Rust prevents overflows/races; hardware bit-flips and logic bugs remain possible.
CrowdStrike RCA (Channel File 291)	Template field-count mismatch caused out-of-bounds input-array read in sensor, triggering Windows system crash/BSOD.	Rust bounds-checked indexing prevents OOB reads; you get an error instead.

How do we keep C++ code functional in safety-critical applications?



Lots of tooling and processes

Coding standards / restricted subsets: MISRA C/C++, AUTOSAR C++14, SEI CERT C/C++, JSF AV C++ (plus documented deviations/waivers).

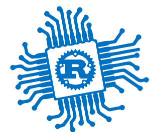
Static analysis & compliance checking: rule checkers + dataflow analyzers (e.g., Polyspace, Coverity/CodeSonar/Parasoft, clang-tidy) integrated in CI.

Compiler hardening & build gates: warnings-as-errors, stack protection, fortified libc checks / hardening bundles (e.g., _FORTIFY_SOURCE, -fstack-protector-strong, -fhardened).

Dynamic bug finding (test builds): sanitizers for memory/UB/races (ASan/UBSan/TSan), plus coverage-guided fuzzing (libFuzzer).

Safety evidence overhead: mandated reviews + traceability + V&V activities (ISO 26262 / DO-178C / IEC 61508-style workflows).

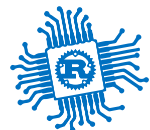
Why Rust - some tech insights



The tagline of Rust is No Undefined Behavior.

- no null reference; the Rust compiler explicitly asks developers to check this;
- no implicit cast, even adding a u32 to a u8 must be casted;
- safe access to shared data across threads verified at compile time;
- uses type states to move runtime checks to compile time and force developers to check;
- clearly defined data types, unlike i8 or u128;
- safe unions, that provide a safeguard to prevent wrong interpretation of data;
- clear code organization into crates and modules;
- backward compatibility at crate level.

Does Rust remove the need for tooling?



No, but it sure makes code safer and dev faster

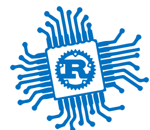
Rust’s advantage is biggest in safe Rust.

As unsafe/FFI grows, assurance shifts back to: unsafe policy & reviews, FFI boundary correctness, sanitizers/fuzzing on risky edges, dependency/toolchain governance; plus the same ISO/DO-178 traceability & V&V.

Rust in Android: move fast and fix things

- A 1000x reduction in memory safety vulnerability density compared to Android’s C and C++ code
- With Rust changes having a 4x lower rollback rate and spending 25% less time in code review, the safer path is now also the faster one

Who supports Rust-lang



Some links to read

the NSA: U.S. and International Partners Issue Recommendations to Secure Software Products Through Memory Safety

The White House: Back to the building blocks: A path toward secure and measurable software

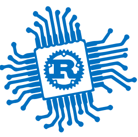
How Rust went from a side project to the world's most-loved programming language

On Rust-lang adoption based on git-hub adoption

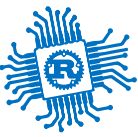
Rust developers at Google are twice as productive as C++ teams

Some companies that are building up Rust teams in embedded:

- Airbus, Ampere, Bae Systems, Boeing, Ford, General Dynamics, Hyundai, Northrop Grumman, NXP, Thales, Toyota, Volvo



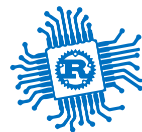
Apollo Guidance Computer



We choose to go to the moon

John F. Kennedy, Rice University, 1961

in this decade and do the other things, ***not because they are easy, but because they are hard****, because* ***that goal will serve to organize and measure the best of our energies and skills****, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win, and the others, too.*

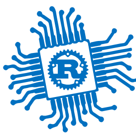


AGC

August 1966

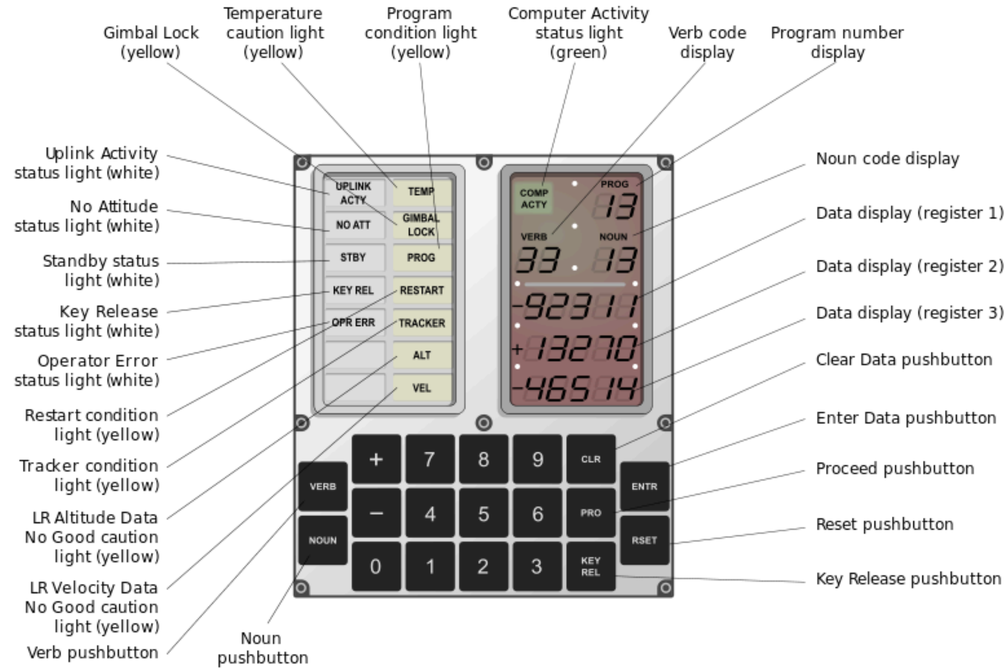
Frequency	2.048 MHz
World Length	15 + 1 bit
RAM	4096 B
Storage	72 KB
Software API	AGC Assembly Language

This landed the *moon eagle*.

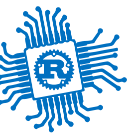


DSKY

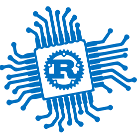
Display and keyboards



Simulator



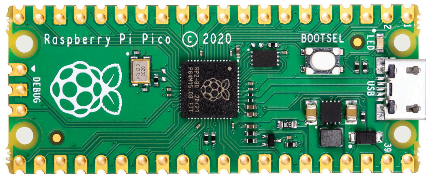
What is a microprocessor?



Microcontroller (MCU)

Integrated in embedded systems for certain tasks

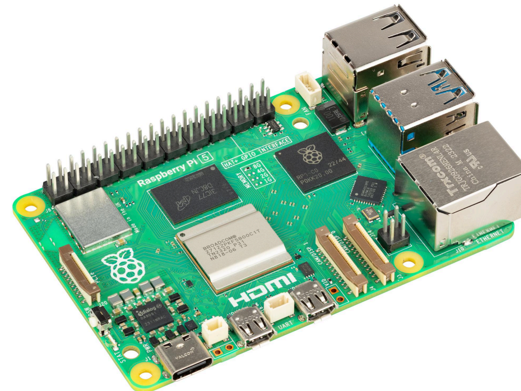
- low operating frequency (MHz)
- a lot of I/O ports
- controls hardware
- does not require an Operating System
- costs $0.1 - $25
- annual demand is billions

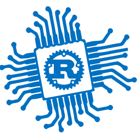


Microprocessor (CPU)

General purpose, for PC & workstations

- high operating frequency (GHz)
- limited number of I/O ports
- usually requires an Operating System
- costs $75 - $500
- annual demand is tens of millions





How a microprocessor works

This is a simple processor

Processor

Registers

r0

r1

r2

Processing

add

sub

and

Memory Read/Write

mov

load

store

Clock

Memory

word 0

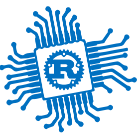
word 1

word 2

LED controller

8 bit processor

a simple 8 bit processor with a text display



Simple 8-bit Processor

Registers
A
B
C
D

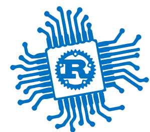
Processing
add
sub
and
⋮

Memory Read/Write
mov

Clock

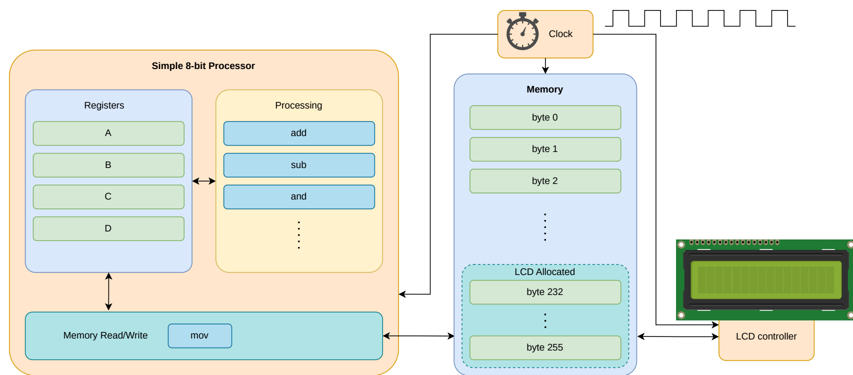
Memory
byte 0
byte 1
byte 2
⋮
LCD Allocated
byte 232
⋮
byte 255

LCD controller



Programming

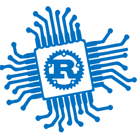
in Rust




```rust
use eight_bit_processor::print;

static hello: &str = "Hello World!";

#[start]
fn start() {
    print(hello);
}
```

Assembly

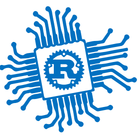
```
  JMP start
hello: DB "Hello World!" ; Variable
       DB 0  ; String terminator
start:
  MOV C, hello    ; Point to var
  MOV D, 232  ; Point to output
  CALL print
       HLT            ; Stop execution
print:       ; print(C:*from, D:*to)
  PUSH A
  PUSH B
  MOV B, 0
.loop:
  MOV A, [C]  ; Get char from var
  MOV [D], A  ; Write to output
  INC C
  INC D
  CMP B, [C]  ; Check if end
  JNZ .loop ; jump if not

  POP B
  POP A
  RET
```



Demo

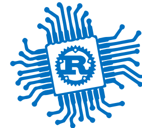
a working example for the previous code

Start



Real World Microcontrollers

Intel / AVR / PIC / TriCore / ARM Cortex-M / RISC-V rv32i(a)mc



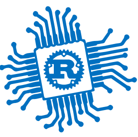
Bibliography

for this section

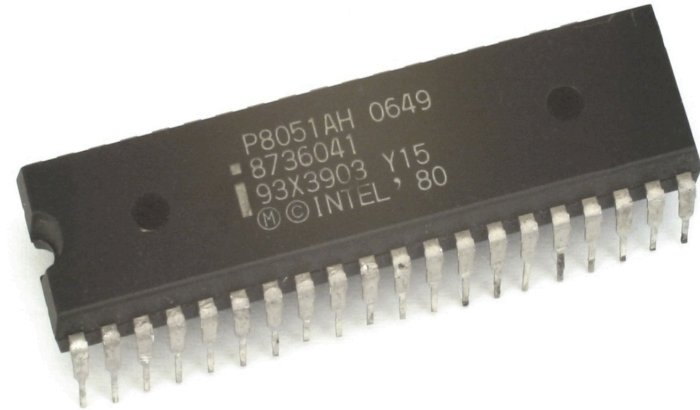
Joseph Yiu, *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors, 2nd Edition*

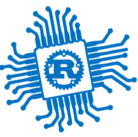
- Chapter 1 - *Introduction*
- Chapter 2 - *Technical Overview*

Intel



Vendor	Intel
ISA	8051, 8051
Word	8 bit
Frequency	a few MHz
Storage	?
Variants	*8048, 8051*

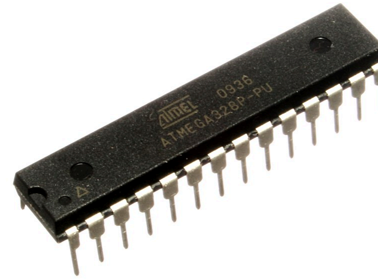




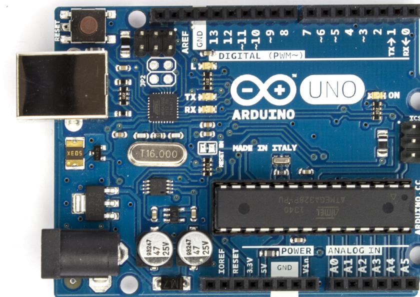
AVR

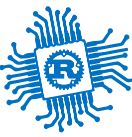
probably *Alf and Vegard's RISC processor*

Authors	Alf-Egil Bogen and Vegard Wollan
Vendor	Microchip (*Atmel*)
ISA	AVR
Word	8 bit
Frequency	1 - 20 MHz
Storage	4 - 256 KB
Variants	*ATmega*, *ATtiny*



Board

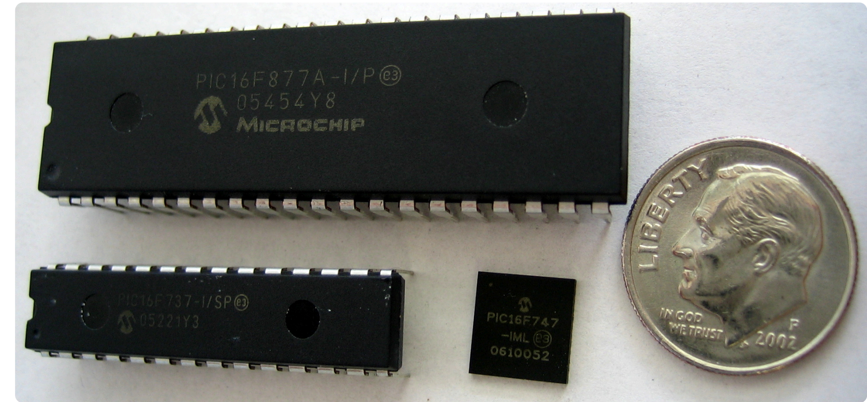


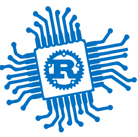


PIC

Peripheral Interface Controller / Programmable Intelligent Computer

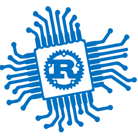
Vendor	Microchip
ISA	PIC
Word	8 - 32
Frequency	1 - 20 MHz
Storage	256 B - 64 KB
Variants	*PIC10, PIC12, PIC16, PIC18, PIC24, PIC32*





TriCore

Vendor	Infineon
ISA	AURIX32
Word	32 bit
Frequency	hundreds of MHz
Storage	a few MB
Variants	*TC2xx*, *TC3xx*, *TC4xx*



ARM Cortex-M

Advanced RISC Machine

Vendor	Qualcomm, NXP, Nordic Semiconductor, Broadcom, Raspberry Pi
ISA	ARMv6-M (Thumb and some Thumb-2) ARMv7-M (Thumb and Thumb-2) ARMv8-M (Thumb and Thumb-2)
Word	32
Frequency	1 - 900 MHz
Storage	up to a few MB
Variants	*M0, M0+, M3, M4, M7, M23, M33*

arm

ARM Cortex-M Instruction Set

what the MCU can do

Fun Facts

- M0/M0+ has no `div`
- M0 - M3 have no floating point
- M23 and M33 have security extensions

Cortex-M7
Cortex-M4

FPv4
VABS VADD VCMP VCMPE VCVT VCVTR VCVTB VCVTT
VDIV VFMA VFMS VFNMA VFNMS VLDM VLDR VMLA
VMLS VMOV VMRS VMSR VMUL VNEG VNMLA VNMLS
VNMUL VPOP VPUSH VSQRT VSTM VSTR VSUB

Added in FPv5
VCVTA VCVTN VCVTP VCVTM VMAXNM VMINNM VRINTA VRINTN VRINTP VRINTM VRINTX VRINTZ VRINTR VSEL

Floating point extension

PKHBT PKHTB QADD QADD16 QADD8 QASX QDADD QDSUB
QSAX QSUB QSUB16 QSUB8 SADD16 SADD8 SASZ SEL
SHADD16 SHADD8 SHASX SHSAX SHSUB16 SHSUB8 SMLABB SMLABT
SMLATB SMLATT SMLAD SMLADX SMLALBB SMLALBT SMLALTB SMLALTT
SMLALD SMLALDX SMLAWB SMLAWT SMLSD SMLSDX SMLSLD SMLSLDX
SMMLA SMMLAR SMMLS SMMLSR SMMUL SMMULR SMUAD SMUADX SMULBB SMULBT SMULTB SMULTT SMULWB SMULWT SMUSD SMUSDX SSAT16 SSAX SSUB16 SSUB8 SXTAB SXTAB16 SXTAH UADD16 UADD8 UASX UHADD16 UHSUB8 UMAAL UQADD16 UQADD8 UQASX UQSAX UQSUB16 UQSUB8 USAD8 USADA8 USAT16 USAX USUB16 USUB8 UXTAB UXTAB16 UXTAH UXTB16

DSP extension

Cortex-M3
ADC ADD ADR AND ASR BFC
BFI BIC CDP CDP2 CLZ CMN
CMP DBG EOR LDC LDC2 LDMIA
LDMDB LDR LDRB LDRBT LDRD LDRH
LDRHT LDRSB LDRSBT LDRSH LDRT LSL
LSR MCR MCR2 MCRR MCRR2 MLA
MLS MRC MRC2 MRRC MRRC2 MUL
MVN NOP ORN ORR PLD PLI
POP PUSH RBIT REV REV16 REVSH
ROR RRX RSB SBC SBFX SEV
SMLAL SMULL SSAT STC STC2 STMIA
B CBNZ CBZ CLREX LDREX LDREXB LDREXH MOV MOVT SDIV STREX STREXB STREXH UDIV
STMDB STR STRB STRBT STRD STRH STRHT STRT SUB SXTB SXTH TBB TBH TEQ TST UBFX UMLAL UMULL UDF USAT UXTB UXTH WFE WFI YIELD IT

Mainline extension

Cortex-M0/M0+
ADC ADD ADR AND ASR B
BIC BKPT BL BLX BX
CMN CMP CPS DMB EOR
DSB ISB LDMIA LDR
LDRB LDRH LDRSB LDRSH LSL LSR
MOV MRS MSR MUL
MVN NOP ORR POP PUSH REV
REV16 REVSH ROR RSB SBC SEV
STMIA STR STRB STRH SUB SVC
SXTB SXTH TST UDF UXTB UXTH
WFE WFI YIELD

LDA LDAB LDAH LDAEX LDAEXB LDAEXH STL STLB STLEX STLEXB STLEXH STLH

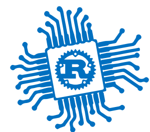
BLXNS BXNS SG TT
Security externsion

VLLDM VLSTM
Mainline Security extension

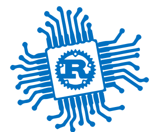
Armv8-M Mainline
Armv8-M Baseline
Cortex-M23
Cortex-M33

RISC-V rv32i(a)mc

Fifth generation of RISC ISA

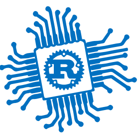


Authors	University of California, Berkeley
Vendor	Espressif System
ISA	rv32i(a)mc
Word	32 bit
Frequency	1 - 200 MHz
Storage	4 - 256 KB
Variants	*rv32imc*, *rv32iamc*



RP2040

ARM Cortex-M0+, built by Raspberry Pi

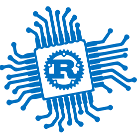


Bibliography

for this section

Raspberry Pi Ltd, *RP2040 Datasheet*

- Chapter 1 - *Introduction*
- Chapter 2 - *System Description*
  - Section 2.1 - *Bus Fabric*



RP2040

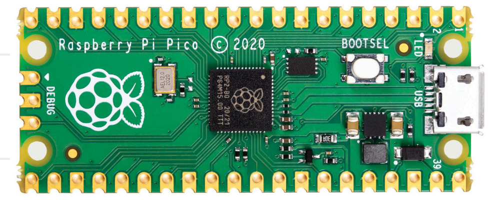
the MCU

Vendor	Raspberry Pi
Variant	ARM Cortex-M0+
ISA	ARMv6-M (Thumb and some Thumb-2)
Cores	2
Word	32 bit
Frequency	up to 133 MHz
RAM	264 KB

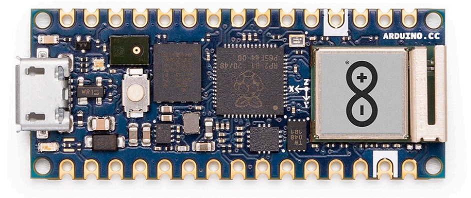
Boards

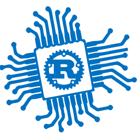
that use RP2040

Raspberry Pi Pico (W)

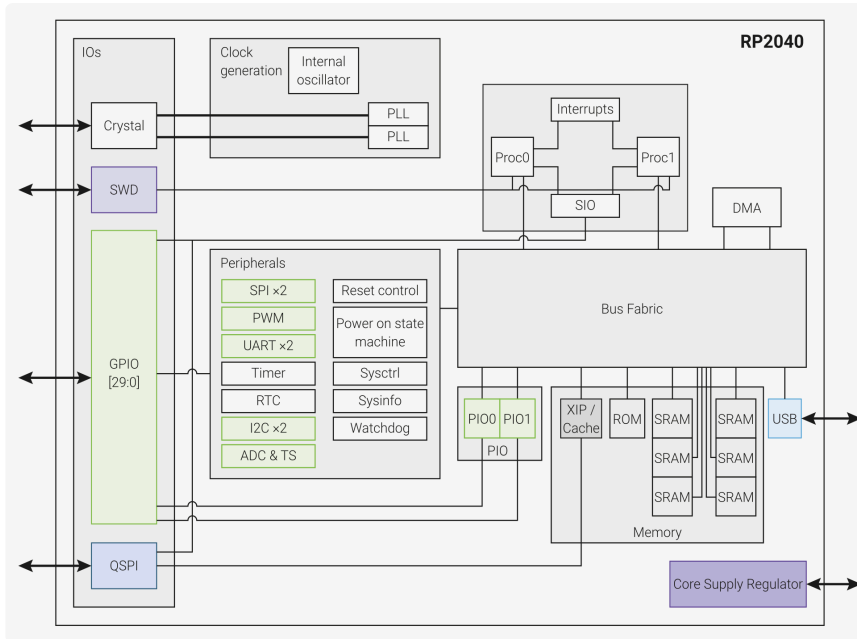


Arduino Nano RP2040 Connect





The Chip



GPIO: General Purpose Input/Output
SWD: Debug Protocol
DMA: Direct Memory Access

Peripherals

SIO	Single Cycle I/O (implements GPIO)
PWM	Pulse Width Modulation
ADC	Analog to Digital Converter
(Q)SPI	(Quad) Serial Peripheral Interface
UART	Universal Async. Receiver/Transmitter
RTC	Real Time Clock
I2C	Inter-Integrated Circuit
PIO	Programmable Input/Output

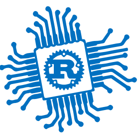
Pins

have multiple functions

Pin	Name
1	IOVDD
2	GPIO0
3	GPIO1
4	GPIO2
5	GPIO3
6	GPIO4
7	GPIO5
8	GPIO6
9	GPIO7
10	IOVDD
11	GPIO8
12	GPIO9
13	GPIO10
14	GPIO11
15	GPIO12
16	GPIO13
17	GPIO14
18	GPIO15
19	TESTEN
20	XIN
21	XOUT
22	IOVDD
23	DVDD
24	SWCLK
25	SWDIO
26	RUN
27	GPIO16
28	GPIO17
29	GPIO18
30	GPIO19
31	GPIO20
32	GPIO21
33	IOVDD
34	GPIO22
35	GPIO23
36	GPIO24
37	GPIO25
38	GPIO26/ADC0
39	GPIO27/ADC1
40	GPIO28/ADC2
41	GPIO29/ADC3
42	IOVDD
43	ADC_AVDD
44	VREG_VIN
45	VREG_VOUT
46	USB_DM
47	USB_DP
48	USB_VDD
49	IOVDD
50	DVDD
51	QSPI_SD3
52	QSPI_SCLK
53	QSPI_SD0
54	QSPI_SD2
55	QSPI_SD1
56	QSPI_SS_N

GND

TOP VIEW



The Bus

that interconnects the cores with the peripherals

Control

Cortex-M0+ Core 0

Cortex-M0+ Core 1

System DMA 1-Write 1-Read

AHB-Lite Crossbar 4:10

AHB-Lite Splitter

ROM 16 kB

SRAM0 64 kB

SRAM1 64 kB

SRAM2 64 kB

SRAM3 64 kB

SRAM4 4 kB

SRAM5 4 kB

APB Bridge

Flash XIP

PIO0

PIO1

USB

APB Splitter

...

UART0

UART1

SPI0

SPI1

I2C0

I2C1

ADC

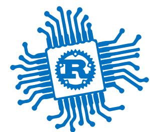
PWM

Timer

Watch-dog

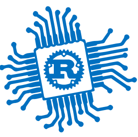
RTC

Other peripherals and system control registers



STM32U545RE

ARM Cortex-M33, built by STMicroelectronics



Bibliography

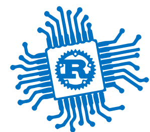
for this section

STMicroelectronics, *STM32U5 Reference Manual*

- Chapter 2 - *Memory and bus architecture*
  - Section 2.1 - *System architecture*

STMicroelectronics, *STM32U5 Datasheet*

- Chapter 2 - *"Description"*
- Chapter 4 - *"Pinout, pin description, and alternate function"*



STM32U545RE

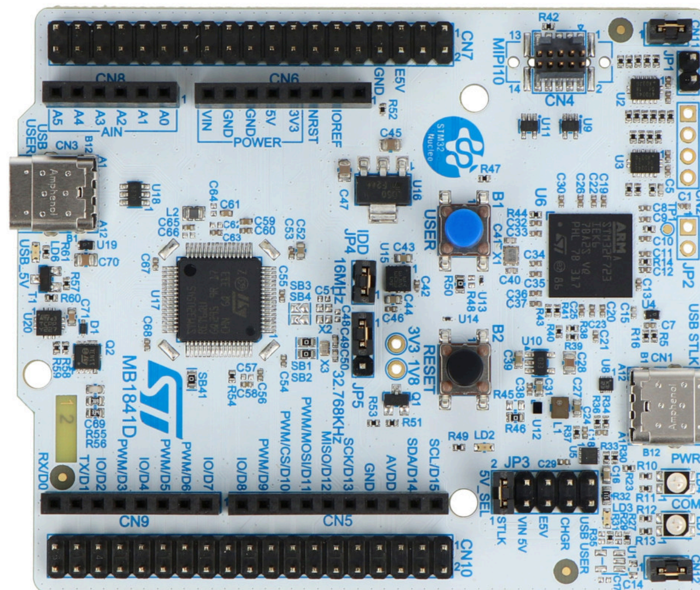
the MCU

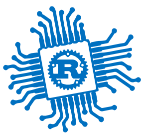
Vendor	STMicroelectronics
Variant	ARM Cortex-M33
ISA	ARMv8-M
Cores	1
Word	32 bit
Frequency	up to 160 MHz
RAM	272 KB

Board

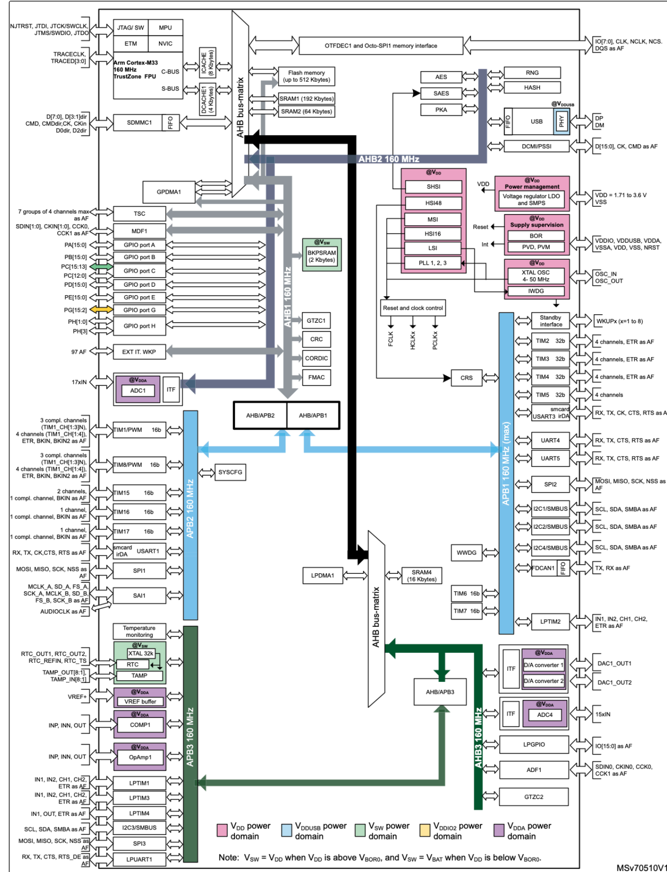
that use STM32U545RE

Nucleo U545RE-Q





The Chip



Peripherals

Parallel interface

FSMC 8-/16-bit (TFT-LCD, SRAM, NOR, NAND)

Timers

19 timers including:
2 x 16-bit advanced motor control timers
4 x ULP timers
5 x 16-bit timers
4 x 32-bit timers

I/Os

Touch-sensing controller
Camera Interface

Arm® Cortex®-M33 CPU

160 MHz
TrustZone®
FPU
MPU
ETM

LPDMA

ART Accelerator™

CORDIC

FMAC

Up to 512-Kbyte Flash memory Dual Bank

274-Kbyte RAM

Connectivity

USB Host/Device
1 x SD/SDIO/MMC, 3 x SPI,
4 x I²C, CAN FD,
1 x Octo-SPI,
4 x USART + 1 x LPUART

Digital

AES (256-bit),
SHA-1, SHA-256,
TRNG, PKA, 1 x SAI,
1 x MDF, 2 x ADF

Analog

1x 14-bit ADC 2 MSPS,
1x 12-bit ADC 2 MSPS,
2 x DAC, 2 x comparators,
1 x op amps
1 x temperature sensor

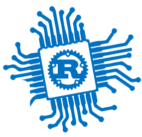
Datasheet STM32U545RE

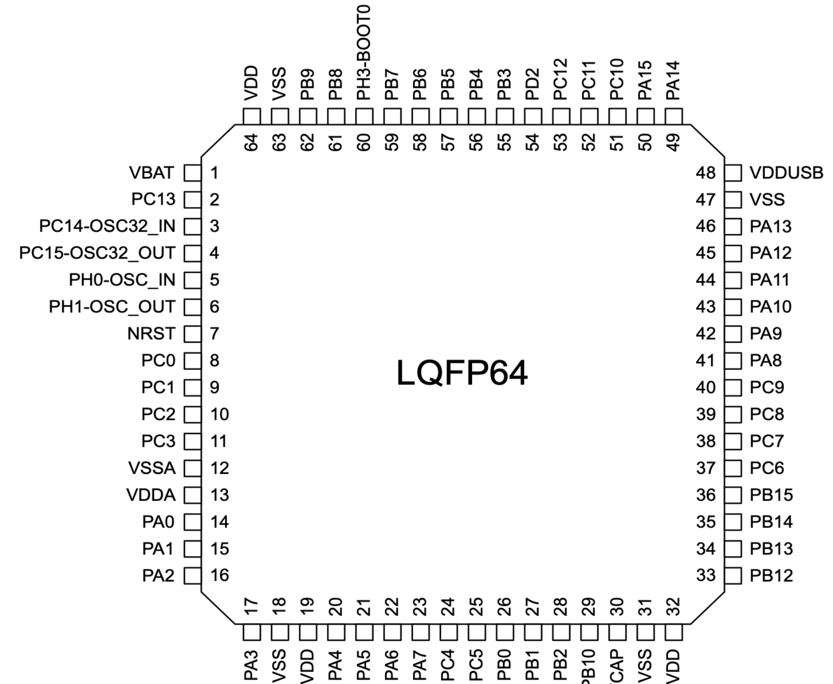
Pins

have multiple functions

GPIO	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11
0		SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB OVCUR DET	
1		SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2	TRACECLK	USB VBUS DET	
2		SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2	TRACEDATA0	USB VBUS EN	UART0 TX
3		SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	TRACEDATA1	USB OVCUR DET	UART0 RX
4		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	TRACEDATA2	USB VBUS DET	
5		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	TRACEDATA3	USB VBUS EN	
6		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART1 TX
7		SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 RX
8		SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB VBUS EN	
9		SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	
10		SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	UART1 TX
11		SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	UART1 RX
12	HSTX	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB OVCUR DET	
13	HSTX	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT0	USB VBUS DET	
14	HSTX	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS EN	UART0 TX
15	HSTX	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT1	USB OVCUR DET	UART0 RX
16	HSTX	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1	PIO2		USB VBUS DET	
17	HSTX	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1	PIO2		USB VBUS EN	
18	HSTX	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1	PIO2		USB OVCUR DET	UART0 TX
19	HSTX	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1	PIO2	QMI CS1n	USB VBUS DET	UART0 RX
20		SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN0	USB VBUS EN	
21		SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	PIO2	CLOCK GPOUT0	USB OVCUR DET	
22		SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	PIO2	CLOCK GPIN1	USB VBUS DET	UART1 TX

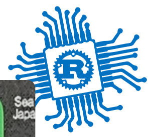
...





The Bus

that interconnects the core with the peripherals

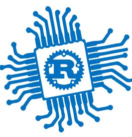


Lab Board

- Nucleo U545RE-Q Slot / Board
- 4 buttons
- 5 LEDs
- potentiometer
- buzzer
- photoresistor
- I2C EEPROM
- MPU-6500 accelerometer & Gyro
- BMP 390 Pressure sensor
- SPI LCD Display
- SD Card Reader
- servo connectors
- stepper motor

Project

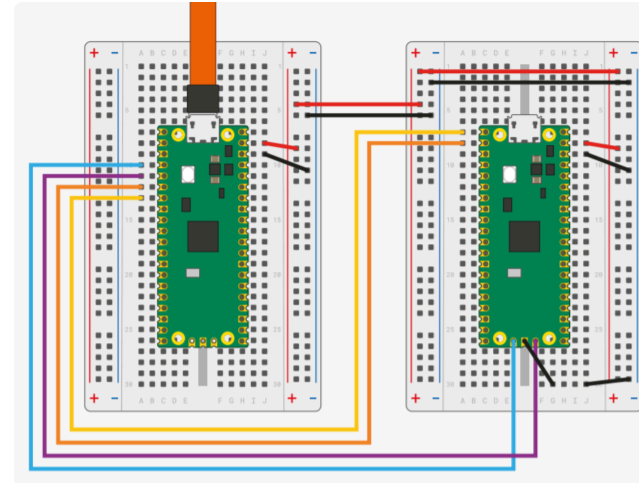
suggested hardware

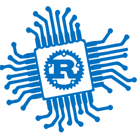


- the hardware should not cost more than 150 RON
- STM32 Nucleo F446RE or Nucleo U545RE-Q board (include debuggers)
- Raspberry Pi Pico with a debugger

Raspberry Pi Pico 2W + Debug Probe

Raspberry Pi Pico 2W + Raspberry Pi Pico 1

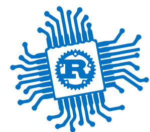




Bitwise Ops

How to set and clear bits

Set bit



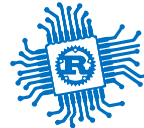
set the `1` on position `bit` of `register`

```rust
fn set_bit(register: usize, bit: u8) -> usize {
    // assume register is 0b1000, bit is 2
    //   1 << 2 is 0b0100
    //   0b1000 | 0b0100 is 0b1100
    register | 1 << bit
}
```

Set multiple bits

```rust
fn set_bits(register: usize, bits: usize) -> usize {
    // assume register is 0b1000, bits is 0b0111
    //   0b1000 | 0b0111 is 0b1111
    register | bits
}
```

Clear bit

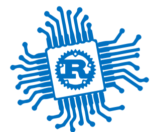


Set the `0` on position `bit` of `register`

```rust
fn clear_bit(register: usize, bit: u8) -> usize {
    // assume register is 0b1100, bit is 2
    //   1 << 2 is 0b0100
    //   !(1 << 2) is 0b1011
    //   0b1100 & 0b1011 is 0b1000
    register & !(1 << bit)
}
```

Clear multiple bits

```rust
fn clear_bits(register: usize, bits: usize) -> usize {
    // assume register is 0b1111, bits is 0b0111
    //   !bits = 0b1000
    //   0b1111 & 0b1000 is 0b1000
    register & !bits
}
```



Flip bit

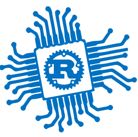
Flip the bit on position `bit` of `register`

```
fn flip_bit(register: usize, bit: u8) -> usize {
    // assume register is 0b1100, bit is 2
    //   1 << 2 is 0b0100
    //   0b1100 ^ 0b0100 is 0b1000
    register ^ 1 << bit
}
```

Flip multiple bits

```
fn flip_bits(register: usize, bits: usize) -> usize {
    // assume register is 0b1000, bits is 0b0111
    //   0b1000 ^ 0b0111 is 0b1111
    register ^ bits
}
```

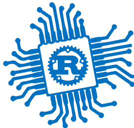
Let’s see a combined operation for value extraction



- We presume an 32 bits ID = `0b1100_1010_1111_1100_0000_1111_0110_1101`
- And want to extract a portion 0b**1100_1010_1111**_1100_0000_1111_0110_1101

```
const MASK: u32 = 0b0000_0000_0000_0000_0000_1111_1111_1111;

fn print_binary(label: &str, num: u32) {
    println!("{}: {:032b}", label, num);
}

fn main() {
    let large_id: u32 = 0b1100_1010_1111_1100_0000_1111_0110_1101;
    let extracted_bits = (large_id >> 20) & MASK;

   // Print values in binary
    print_binary("Original_", large_id);
    print_binary("Mask_____", MASK);
    print_binary("Extracted", extracted_bits);
}
/* RESULT
Original_: 11001010111111000000111101101101
Mask_____: 00000000000000000000111111111111
Extracted: 00000000000000000000110010101111 */
```

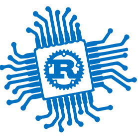
With nice formating



```rust
const MASK: u32 = 0b0000_0000_0000_0000_0000_1111_1111_1111;
fn format_binary(num: u32) -> String {
    (0..32).rev()
        .map(|i| {
            if i != 0 && i % 4 == 0 {
                format!("{}_", (num >> i) & 1)
            } else {
                format!("{}", (num >> i) & 1)
            }
        })
        .collect::<Vec<_>>()
        .join("")
}
fn print_binary(label: &str, num: u32) { println!("{}: {}", label, format_binary(num));}
fn main() {
    let large_id: u32 = 0b1100_1010_1111_1100_0000_1111_0110_1101;
    let extracted_bits = (large_id >> 20) & MASK;
    print_binary("Original_", large_id);
    print_binary("Extracted", extracted_bits);
}
/* RESULTS:
Original_: 1100_1010_1111_1100_0000_1111_0110_1101
Extracted: 0000_0000_0000_0000_0000_1100_1010_1111 */
```

Conclusion

we talked about



- How a processor functions
- Microcontrollers (MCU) / Microprocessors (CPU)
- Microcontroller architectures
- ARM Cortex-M
- RP2040 and STM32U545RE