Microprocessors

Raspberry Pi Pico, Micropython

Stanislav Vítek
Department of Radioelectronics
Czech Technical University in Prague

General-purpose Processors

- Programmable devices
  - Microprocessor
  - Microcontroller
- Main components
  - Program and data memory
  - General data path
  - Register set
  - General ALU
- Application-specific processors (ASIC)
  - Optimized data path
  - Special functional blocks

Dedicated Processors

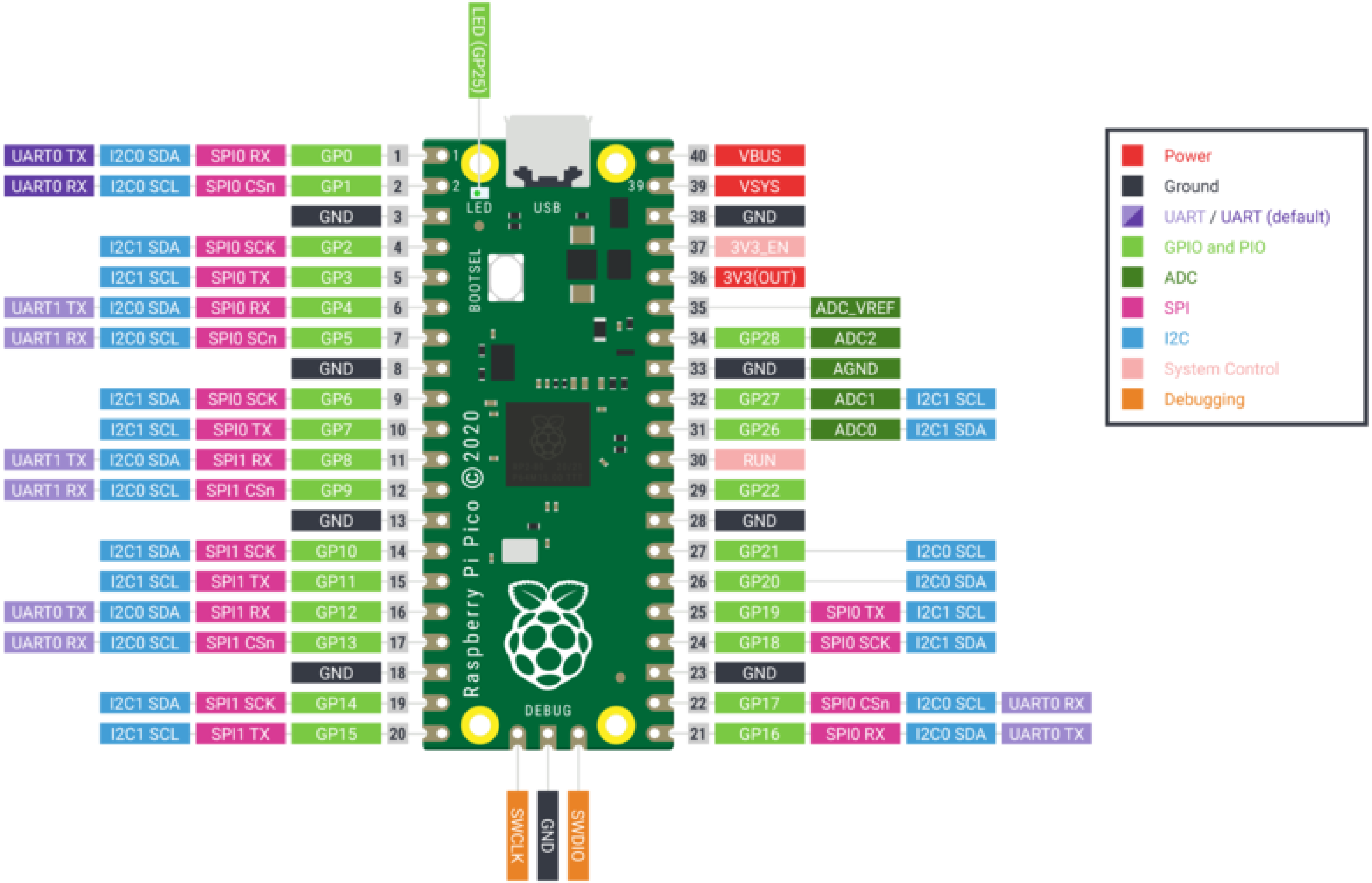
- Application-specific digital circuit
- Main components
  - Components necessary to perform a single program
  - No program memory
- Advantages
  - Small
  - Fast
  - Low power consumption

Embedded Systems

- Dedicated functionality
- Real-time operation
- Small size and low weight
- Low power consumption
- Harsh environments
- Operation critical in terms of security
- Cost-effective

Raspberry Pi Pico

- All previous Raspberry Pi boards, such as Raspberry Pi 3 Model B+, Raspberry Pi 4 Model B, or smaller Raspberry Pi Zero, were equipped with Broadcom processors (BCM2835, BCM2836, BCM2711, etc.).
- Raspberry Pi Pico is equipped with RP2040, a microcontroller designed by Raspberry Pi, the first proprietary processor from Raspberry Pi Foundation.
- RP2040 is based on two ARM Cortex-M0+ cores with a clock frequency of up to 133 MHz and is manufactured using 40 nm technology.
- The RP2040 MCU also has MicroPython support and a UF2 bootloader in ROM for easy program loading.

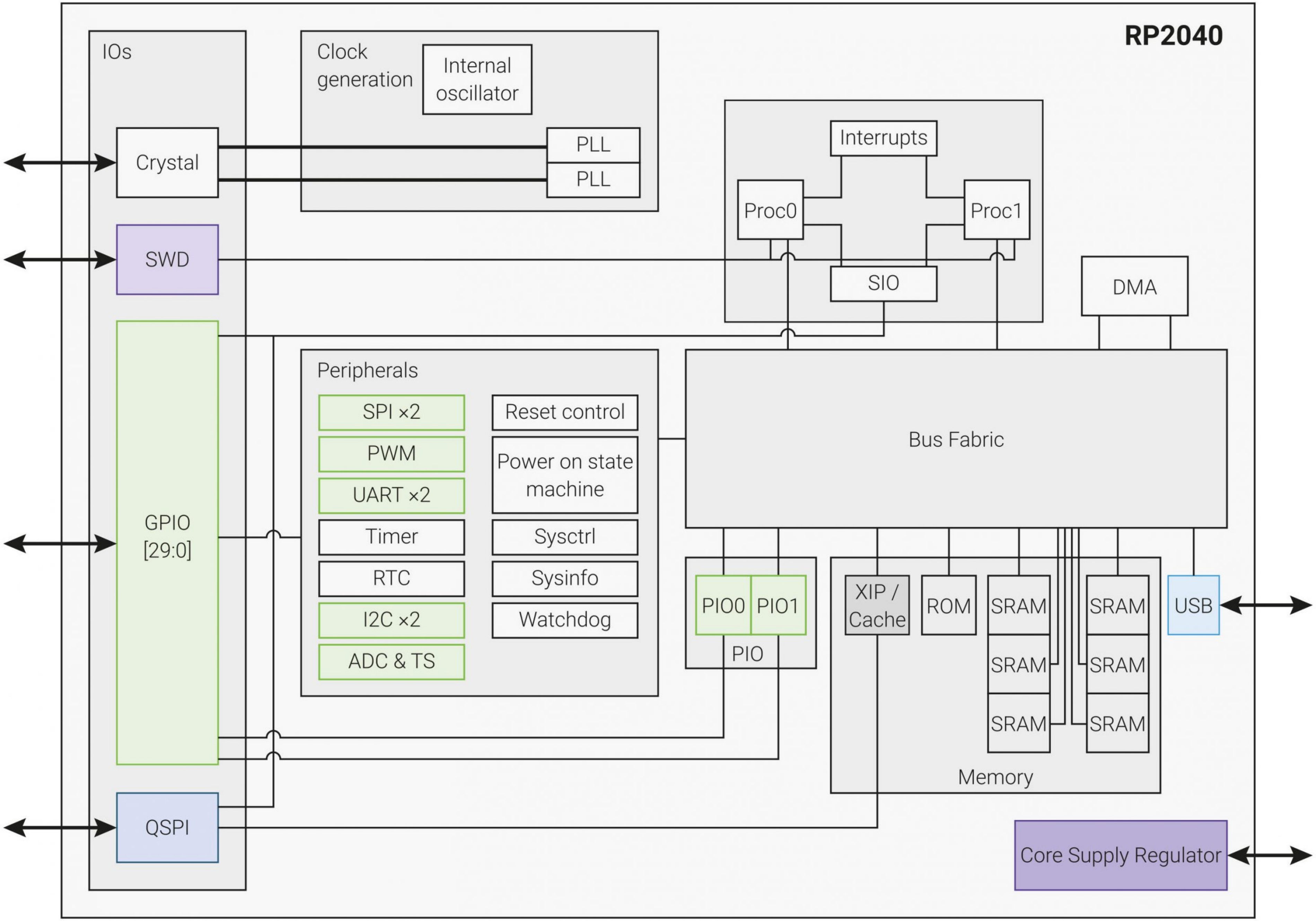

LED (GP25)
USB
LED
BOOTSEL
DEBUG
Raspberry Pi Pico © 2020
UART0 TX | I2C0 SDA | SPI0 RX | GP0 | 1
UART0 RX | I2C0 SCL | SPI0 CSn | GP1 | 2
GND | 3
I2C1 SDA | SPI0 SCK | GP2 | 4
I2C1 SCL | SPI0 TX | GP3 | 5
UART1 TX | I2C0 SDA | SPI0 RX | GP4 | 6
UART1 RX | I2C0 SCL | SPI0 SCn | GP5 | 7
GND | 8
I2C1 SDA | SPI0 SCK | GP6 | 9
I2C1 SCL | SPI0 TX | GP7 | 10
UART1 TX | I2C0 SDA | SPI1 RX | GP8 | 11
UART1 RX | I2C0 SCL | SPI1 CSn | GP9 | 12
GND | 13
I2C1 SDA | SPI1 SCK | GP10 | 14
I2C1 SCL | SPI1 TX | GP11 | 15
UART0 TX | I2C0 SDA | SPI1 RX | GP12 | 16
UART0 RX | I2C0 SCL | SPI1 CSn | GP13 | 17
GND | 18
I2C1 SDA | SPI1 SCK | GP14 | 19
I2C1 SCL | SPI1 TX | GP15 | 20
40 | VBUS
39 | VSYS
38 | GND
37 | 3V3_EN
36 | 3V3(OUT)
35 | ADC_VREF
34 | GP28 | ADC2
33 | GND | AGND
32 | GP27 | ADC1 | I2C1 SCL
31 | GP26 | ADC0 | I2C1 SDA
30 | RUN
29 | GP22
28 | GND
27 | GP21 | I2C0 SCL
26 | GP20 | I2C0 SDA
25 | GP19 | SPI0 TX | I2C1 SCL
24 | GP18 | SPI0 SCK | I2C1 SDA
23 | GND
22 | GP17 | SPI0 CSn | I2C0 SCL | UART0 RX
21 | GP16 | SPI0 RX | I2C0 SDA | UART0 TX
SWCLK
GND
SWDIO
Power
Ground
UART / UART (default)
GPIO and PIO
ADC
SPI
I2C
System Control
Debugging

Raspberry Pi Pico

- RP2040 Microcontroller
- 2 MB SPI Flash memory
- Micro-USB type B port for power and programming
- 40 DIP-type input and output pins with edge soldering
- 3-pin ARM serial debugging interface (SWD)
- 12 MHz crystal oscillator
- Boot select button
- One user LED (connected to GPIO 25, on the **W** model with a Wi-Fi controller)
- 3.3V Buck-Boost SMPS converter

RP2040 Microcontroller

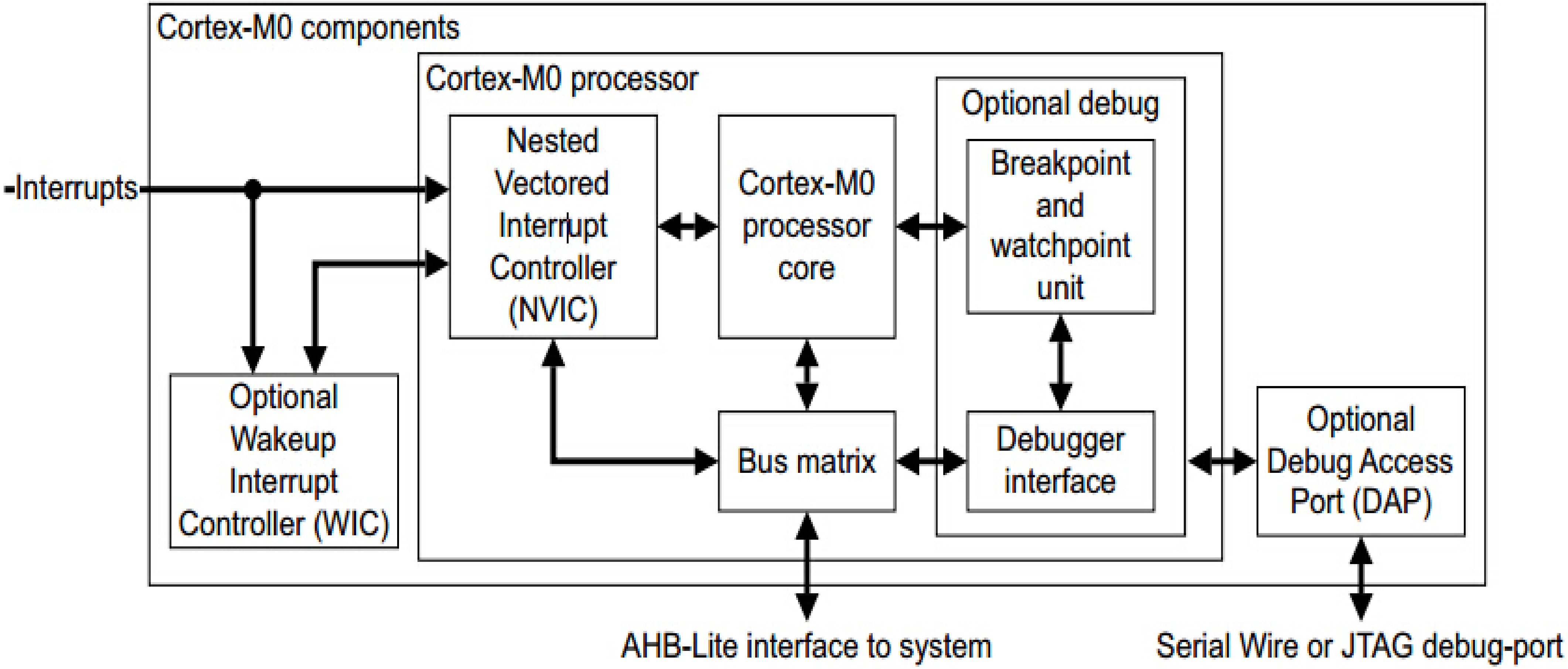
- Two ARM Cortex-M0+ cores
- Clock frequency up to 133 MHz
- 264 KB embedded SRAM memory
- 30 GPIO pins
- Up to 16 MB Flash memory external to the chip
- Four-channel ADC with 12-bit resolution
- Programmable I/O (PIO)
- Additional peripherals - 2x UART, 2x SPI controller, 2x I2C controller, 16 PWM channels, USB 1.1 controller


RP2040
IOs
Crystal
SWD
GPIO
[29:0]
QSPI
Clock
generation
Internal
oscillator
PLL
PLL
Interrupts
Proc0
Proc1
SIO
DMA
Peripherals
SPI ×2
PWM
UART ×2
Timer
RTC
I2C ×2
ADC & TS
Reset control
Power on state
machine
Sysctrl
Sysinfo
Watchdog
Bus Fabric
PIO0
PIO1
PIO
XIP /
Cache
ROM
SRAM
SRAM
SRAM
SRAM
SRAM
SRAM
Memory
USB
Core Supply Regulator

ARM Cortex-M0+

- Core communication interface
  - External AHB-Lite interface -> bus fabric
  - Debug Access Port (DAP)
  - Single-cycle I/O Port -> SIO peripherals
- Core configuration
  - 32-bit, Little Endian, 8 MPU (Memory Protection Unit)
  - Debug support (2-wire SWD interface)
  - 26 ext. interrupts, 34 WIC (Wake-up Interrupt Controller)
  - All registers reset upon restart

ARM Cortex-M0+ Architecture



Clock Sources

clk_ref

- Internal Ring Oscillator (ROSC), can be switched to external crystal oscillator (XOSC)
- 6-12 MHz

clk_sys

- Initially powered from clk_ref, then typically switched to PLL
- 125 MHz

clk_peri

- Typically powered from clk_sys, but allows peripherals to be independent (clk_sys can be changed in software)
- 12-125 MHz

Clock Sources

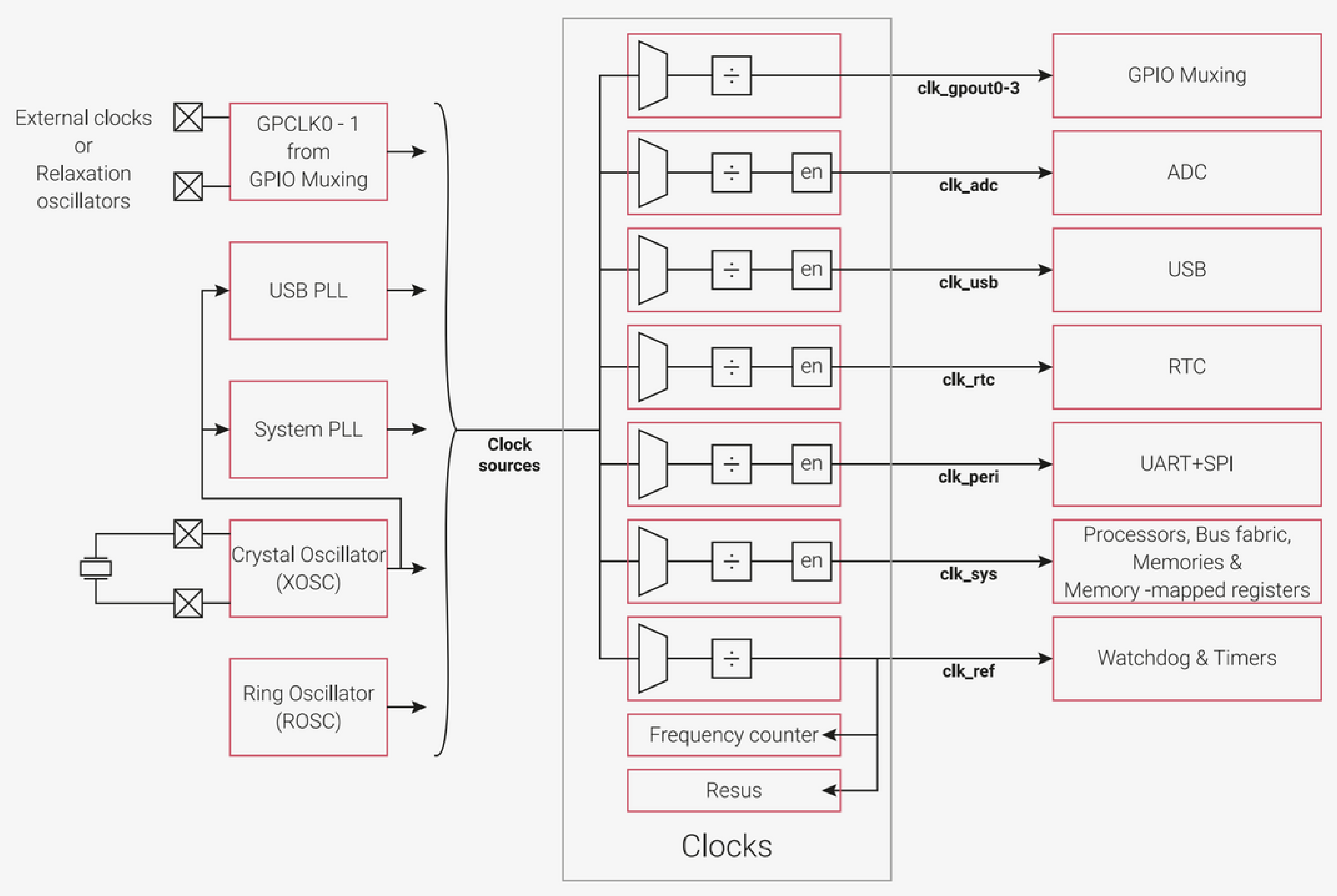
clk_usb, clk_adc

- Reference clocks for USB and ADC
- 48 MHz

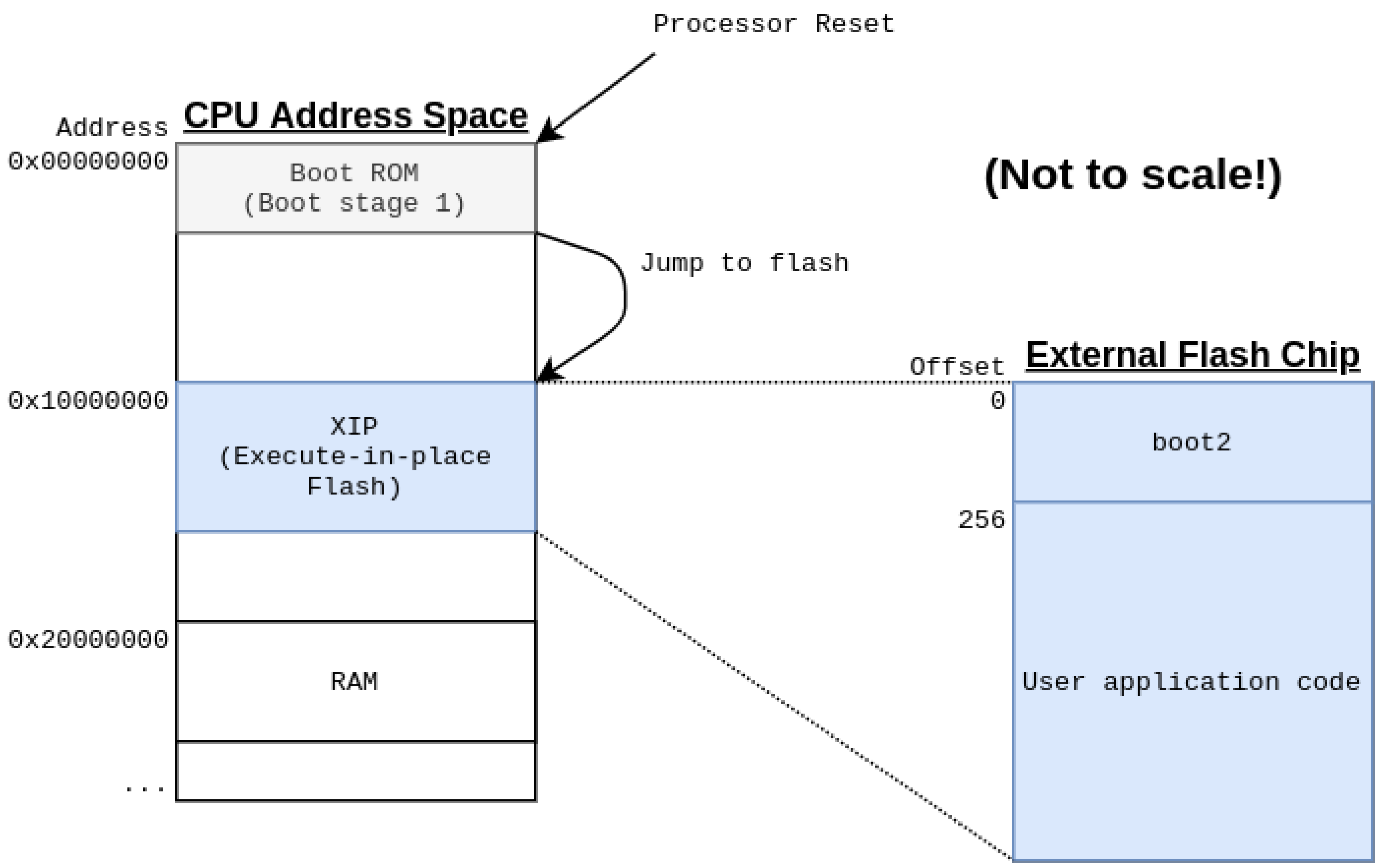
clk_rtc

- RTC divides the clock to get a 1s reference
- 46875 Hz

For more details, refer to the documentation.


External clocks
or
Relaxation
oscillators
GPCLK0 - 1
from
GPIO Muxing
USB PLL
System PLL
Crystal Oscillator
(XOSC)
Ring Oscillator
(ROSC)
Clock
sources
÷
en
Frequency counter
Resus
Clocks
clk_gpout0-3
clk_adc
clk_usb
clk_rtc
clk_peri
clk_sys
clk_ref
GPIO Muxing
ADC
USB
RTC
UART+SPI
Processors, Bus fabric,
Memories &
Memory -mapped registers
Watchdog & Timers

Memory



Processor Reset
Address
CPU Address Space
0x00000000
Boot ROM
(Boot stage 1)
Jump to flash
(Not to scale!)
Offset
External Flash Chip
0x10000000
XIP
(Execute-in-place
Flash)
0
boot2
256
User application code
0x20000000
RAM
...

Peripherals

- GPIO pins
- UART (x2)
- SPI (x2)
- I2C (x2)
- 16 PWM channels
- 8 PIO state machines
- USB 1.1 controller
- AHB DMA controller
- Timer
- Real-Time Counter (RTC)
- PIO (x2)

GPIO

- 30 programmable input/output pins
- 26×2 external interrupt handlers
- 30 input/output programmable I/O pins
- 26 external interrupt handlers
- Pin layout is similar to most other RP boards

GPIO

Some pins are better suited for special purposes, such as:

- GP0-3 are **Quad-SPI** pins
- GP4-11 can function as an **I2C** controller
- GP14-15 are the **UART0** controller
- GP16-17 are the **UART1** controller
- GP25 and GP28 have an **onboard LED**
- GP26 and GP27 are the two 5 V-tolerant pins
- Each GPIO pin has an associated **6-bit input pad and 12-bit ADC**. They are grouped into banks and each bank can be independently configured.

GPIO Pin Characteristics

- 3.3 V I/O
- 5 V-tolerant inputs
- Software-configurable input pull-ups and pull-downs
- Programmable rise/fall input edges
- Separate drive enables for each pin (only one pin per bank supports UART functionality)
- Input sensing, selectable between 3 levels and a glitch filter

Voltage Levels

- GPIO pins operate at 3.3 V, so it's safe to use 3.3 V devices.
- **5 V-tolerant** inputs allow direct connection of **5 V signals**.
- **Voltage input threshold** for logic 1: typically 2.0 V, minimum 1.65 V.
- **Voltage input threshold** for logic 0: typically 0.8 V, maximum 0.99 V.

UART

- Two UARTs, UART0 and UART1, available for serial communication.
- UART0: **GP0 (TX), GP1 (RX)**
- UART1: **GP4 (TX), GP5 (RX)**
- UART baud rate up to 2 Mbps.
- Each UART module contains a 16-byte hardware FIFO for transmit and receive.

SPI

- Two SPI controllers, SPI0 and SPI1, support Serial Peripheral Interface communication.
- SPI0: **GP16 (CE0), GP17 (CE1), GP18 (MOSI), GP19 (MISO), GP20 (SCK)**
- SPI1: **GP10 (CE0), GP11 (CE1), GP12 (MOSI), GP13 (MISO), GP14 (SCK)**
- Data rates up to 50 Mbps.

I2C

- Two I2C controllers support I2C communication.
- I2C0: **GP0 (SDA), GP1 (SCL)**
- I2C1: **GP2 (SDA), GP3 (SCL)**
- Data rates up to 400 kHz.

PWM

- 16 PWM channels.
- **GP0-GP15** and **GP26-GP27** can act as PWM pins.
- Independent counters and clocks for each channel.

USB 1.1 Controller

- Implements **Low-speed (1.5 Mbps) USB 1.1 protocol**.
- Requires an external crystal (12 MHz) connected to XOSC.
- Includes a data FIFO for endpoints.
- Supports one USB output (D-).
- Optionally allows USB voltage to be provided from a GPIO.

AHB DMA Controller

- Provides **DMA (Direct Memory Access) capability**.
- Has **8 DMA channels**, which can be used to move data between peripherals.
- The AHB DMA controller can perform burst data transfers.
- **AHB-Lite interface** connects to main bus fabric and can transfer data between memory and peripherals.

Timer

- One timer module available.
- Suitable for basic timer and delay functions.
- Each timer has a 16-bit counter and can generate an interrupt when the counter reaches a specified value.
- The timer can run from the reference clock, sys clock, or any general-purpose clock.

Real-Time Counter (RTC)

- A **1 Hz counter** that provides a **reference clock source**.
- The RTC counts from 0 to 31,768 and then wraps back to 0.
- It generates a **tick interrupt** at 1 Hz.
- The RTC provides **calibration registers** for fine-tuning its frequency.

PIO (Programmable Input/Output)

- 8 PIO state machines are available.
- Each PIO state machine can operate independently.
- The PIO is used to provide deterministic and tightly controlled I/O operations.
- It's highly flexible and can be used for various custom serial protocols, high-speed interfacing, and generating complex waveforms.
- Provides direct, low-level, hardware-timed control of I/O signals.

PIO (Programmable Input/Output)

- PIO in the RP2040 can be programmed at a lower level even within Micropython.
- PIO assembler instructions: JMP, WAIT, IN, OUT, PUSH, PULL, MOV, IRQ, and SET
    - Instructions are focused on bit manipulation
    - Each instruction takes exactly one clock cycle
    - They do not implement any arithmetic operations
    - The only logical operation is bitwise complement
- The rp2 module provides a wrapper for assembler instructions.
    - For example, the **set()** function creates a wrapper for the **SET** instruction, which toggles the state of a GPIO pin independently of the main processor.
- Examples can be found on GitHub.

RP2040 Datasheet

For more detailed information on RP2040 and Raspberry Pi Pico, you can refer to the official RP2040 Datasheet.

RP2040 Software Ecosystem 1/2

The Raspberry Pi Pico and RP2040 microcontroller are supported by a rich software ecosystem, which includes the following:

1. **C/C++ SDK**: Official SDK for low-level access and high-performance applications.
2. **MicroPython**: Popular among developers who prefer Python.
3. **Thonny**: A beginner-friendly integrated development environment (IDE).
4. **CircuitPython**: A variant of MicroPython that simplifies hardware programming.

RP2040 Software Ecosystem 2/2

5. **Visual Studio Code**: An advanced development environment for those who prefer a full-featured IDE.
6. **PlatformIO**: An open-source ecosystem for IoT development.
7. **Rust**: For developers interested in a systems programming language.
8. **Pico Explorer**: A display, input, and audio add-on for Raspberry Pi Pico.
9. **Pico Display**: A MicroPython library for controlling a Pico Display Pack.

MicroPython

- Raspberry Pi Pico supports MicroPython, which is a lightweight Python implementation optimized for microcontrollers.
- You can easily program the Pico using MicroPython to create your IoT applications.
- MicroPython provides access to all the RP2040's features, including GPIO, UART, SPI, I2C, PWM, PIO, and more.
- It's an efficient way to get started with IoT development on Raspberry Pi Pico.

Micropython

- Micropython Documentation
- Micropython Introduction Tutorial

Micropython is not the only Python implementation for microcontrollers.

- CircuitPython, a derivative of Micropython, maintained by Adafruit, with differences
- MicroPython for BBC micro:bit

General RPi Control

machine Module

- Abstract layer for hardware communication (common across multiple controllers)

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(240000000) # set the CPU frequency to 240 MHz
```

Module rp2

- RP2040 specific functions

```
import rp2
```

Machine Module

- The module contains specific functions related to hardware on a particular board.
- Most functions of the module allow direct and unrestricted access to hardware blocks of the system (such as the processor, timers, buses, etc.) and their control.
- Incorrect usage can lead to malfunctions, locking up, and in extreme cases, hardware damage.
- On suitable hardware, MicroPython offers the possibility to write interrupt handlers in the Python language. Interrupt handlers, also known as Interrupt Service Routines (ISRs), are defined as callback functions. These are executed in response to events such as timer triggers or changes in voltage on a pin.

Memory Access

- The module defines three objects for direct memory access.

machine.mem8

- Write/read 8 bits of memory.

machine.mem16

- Write/read 16 bits of memory.

machine.mem32

- Write/read 32 bits of memory.

Memory Access Example

- Example specific to the STM32 platform

```
import machine
from micropython import const

GPIOA = const(0x48000000)
GPIO_BSRR = const(0x18)
GPIO_IDR = const(0x10)

# Set PA2 high
machine.mem32[GPIOA + GPIO_BSRR] = 1 << 2

# Read PA3
value = (machine.mem32[GPIOA + GPIO_IDR] >> 3) & 1
```

Device Reset 1/2

machine.reset()

- Resets the device with the same effect as an external Reset signal.

machine.soft_reset()

- Soft reset of the interpreter, removes all Python objects and resets the Python heap.
- Tries to preserve the way the user is connected to the MicroPython REPL (e.g., serial, USB, Wi-Fi).

machine.reset_cause()

- Returns the cause of the reset.
- Causes are described by constants.

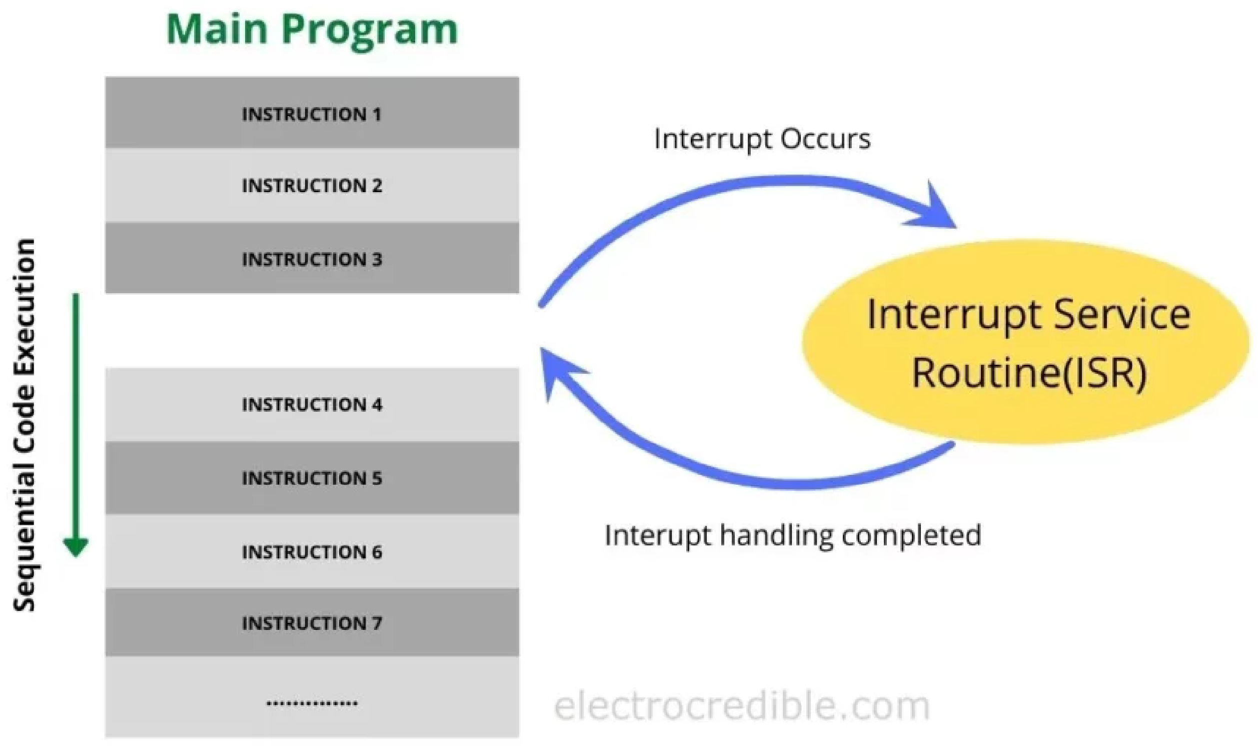
Device Reset 2/2

machine.bootloader([value])

- Resets the device and enters its bootloader.
- Typically used to put the device in a state where new firmware can be programmed.
- Some ports support passing an optional argument, `value`, which can control which bootloader to enter or what to pass to it.

Interrupts

- Interrupts are handled by software components called Interrupt Service Routines (ISRs).
- When an interrupt occurs, the processor starts executing code within this routine.
- After completing the task in the routine, the processor resumes executing code from where it left off.



Interrupts

- Interrupts can be disabled (turned off) and re-enabled.
- Some subsystems require interrupts for proper operation, so disabling them for an extended period can jeopardize the core's functionality (e.g., watchdog).
- Interrupts should only be disabled for a minimal duration and then re-enabled to their previous state.

```
import machine

# Disable interrupts
state = machine.disable_irq()

# Do a small amount of time-critical work here

# Enable interrupts
machine.enable_irq(state)
```

Interrupts

machine.disable_irq()

- Disables interrupt requests.
- Returns the previous IRQ state, which should be considered as an opaque value.
- This return value should be passed to the `enable_irq()` function to restore interrupts to their previous state before calling `disable_irq()`.

machine.enable_irq(state)

- Re-enables interrupt requests.
- The `state` parameter should be the value returned from the last call to the `disable_irq()` function.

Power

machine.freq([Hz])

- Returns the processor frequency in Hz. On some ports, this function can also be used to set the processor frequency by providing the Hz value.

machine.idle()

- Halts the processor clock, which is useful for reducing power consumption at any time during short or long periods.
- Peripherals continue to work, and execution resumes as soon as any interrupt is triggered (on many ports, this includes a system timer interrupt occurring at regular intervals in milliseconds).

Power

machine.lightsleep([time_ms])

machine.deepsleep([time_ms])

- Halts program execution and attempts to enter a low-power state.
- If **time_ms** is provided, it's the maximum time in milliseconds for which the sleep will last. Otherwise, sleep may last indefinitely.
- Program execution can be resumed at any time with or without a time limit if events requiring processing occur. Such wakeup events or sources should be configured before sleeping, such as a pin change or RTC timeout.

Power 3/4

The exact behavior and power-saving capabilities of **lightsleep** and **deepsleep** modes are highly hardware-dependent, but some general features are as follows:

- **lightsleep** preserves RAM and state. After waking up, execution continues from the point where sleep was requested, and all subsystems are functional.
- **deepsleep** must not preserve RAM or any other system state (e.g., peripherals or network interfaces). After waking up, execution is resumed from the main script, similar to a hard or power-on reset. The `reset_cause()` function returns the value `machine.DEEPSLEEP`, which can be used to distinguish a deep sleep wake-up from other resets.

Power 4/4

machine.wake_reason()

- Returns the reason for waking up from sleep.
- Wakeup reasons are described by constants.

Additional Functions

Other useful functions are summarized on this page.

Timer

- The RP2040 system timer peripheral provides a global microsecond time base and generates interrupts for it.
- Simultaneously, a software timer is available in unlimited quantity (if memory allows).
- The timer is described by the machine.Timer class.

```
from machine import Timer

tim = Timer(period=5000, mode=Timer.ONE_SHOT, callback=lambda t:print(1))
tim.init(period=2000, mode=Timer.PERIODIC, callback=lambda t:print(2))
```

GPIO

- GPIO is described by the machine.Pin class.

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

GPIO with interrupt

```
import time
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(16, mode=Pin.OUT)


def button_isr(pin):
  pin_led.value(not pin_led.value())

pin_button.irq(trigger=Pin.IRQ_FALLING, handler=button_isr)

while True:
  ...
```

ADC

- ADC is described by machine.ADC

```
from machine import ADC
import utime

sensor_temp = ADC(4)
conversion_factor = 3.3 / (65535)

while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    print(temperature)
    utime.sleep(2)
```

UART

- UART is described by the machine.UART class.
- RP2040 has two UART peripherals (UART0 and UART1).
  - Programmable data length (5-8 bits) and the number of stop bits (1 or 2).
  - FIFO in both directions up to 32 bytes.
- Interrupts can be used to monitor data arrival or departure, device status, communication error, or data reception timeout.
- Both devices can be configured on various pairs of TX and RX pins.
  - UART0: GP0-GP1, GP12-GP13, GP16-GP17
  - UART1: GP4-GP5, GP8-GP9

UART

```
from machine import Pin, UART
import time

uart = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart.init(bits=8, parity=None, stop=2)

led = Pin("LED", Pin.OUT)

while True:
    uart.write('t')
    if uart.any():
        data = uart.read()
        if data== b'm':
            led.toggle()
    time.sleep(1)
```