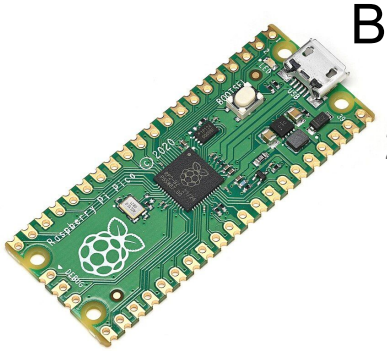
Everything You Always Wanted to Know About the

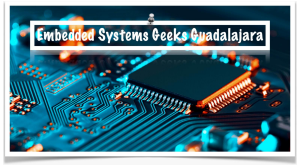
Raspberry Pi Pico

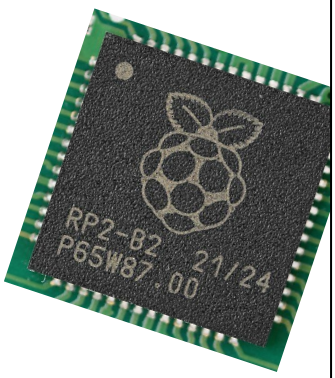
But Were Afraid to Ask

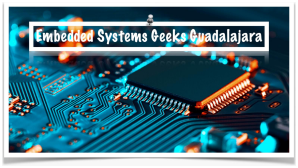
David Barnett

25 Apr 2024 @ Zoolatech







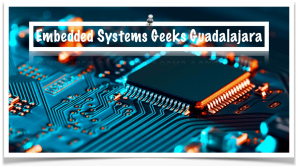


David Barnett 👋

- Software engineer & tinkerer
  - 11 years at the big G, meebo, startups
- Embedded systems experience
  - Smart lighting
  - Accessibility projects
  - Home automation

linkedin.com/in/david-e-barnett

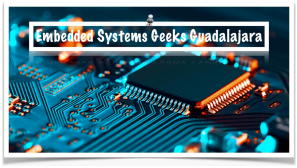
github.com/dbarnett



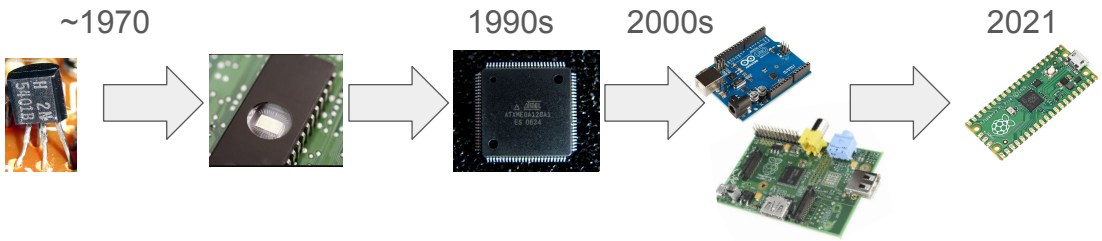
"Raspberry Pi is an engine for creativity, learning and innovation."

For hobbyists to build electronic stuff (**low-cost** and **easy to program**)

A bit of history…



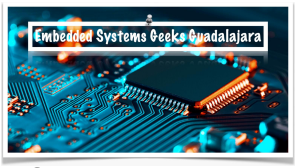
- First released in January 2021 at retail price of $4 USD
- Raspberry Pico is **not** a Raspberry Pi:
  - Raspberry Pi is a full-blown mini-computer (runs Linux and more …)
  - Pico is based on a microcontroller, the **RP2040** chip (runs bare-metal)
- Raspberry Pico W released in June 2022, supports WiFi and Bluetooth.



Timeline:

- Before ~1970: physical/solder
- ~1971: first programmable electronics
- 1990s: first MP/AVR MCUs
- 2005: Arduino
- 2014: Raspberry Pi
- 2021: Pico

Why “Pico”?



- The RP2040 microcontroller of the Pico combines a lot of nifty features.
- Not as fast as the ESP32 but PIOs and flexible DMA make up for it!
  - ESP32 can run at up to 240 MHz.
  - Pico runs at 133 MHz but can be overclocked (if not too aggressively).
- Easy to get started with due to support of *CircuitPython*.
- Datasheet is of exceptional high quality with lots of useful example.
- Well-design board SDK and easy to integrate libraries via **cmake**.

Note: performance is not always about CPU **clock speed**!

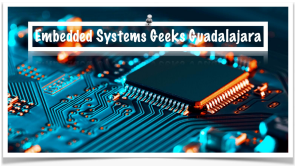
- ESP32 can generate VGA video signal with some I²S wizardry.
- RP2040 can generate VGA video signal natively with PIO (demo later on).

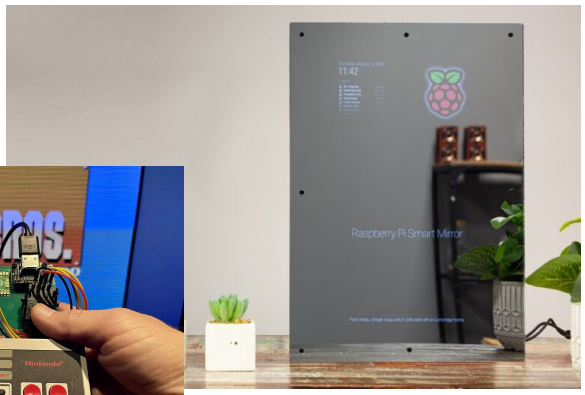
Why use a Pico vs. Pi?

- Price
- Fast boot
- Also… CHIP SHORTAGE 😅

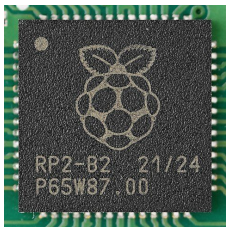
What can it do?

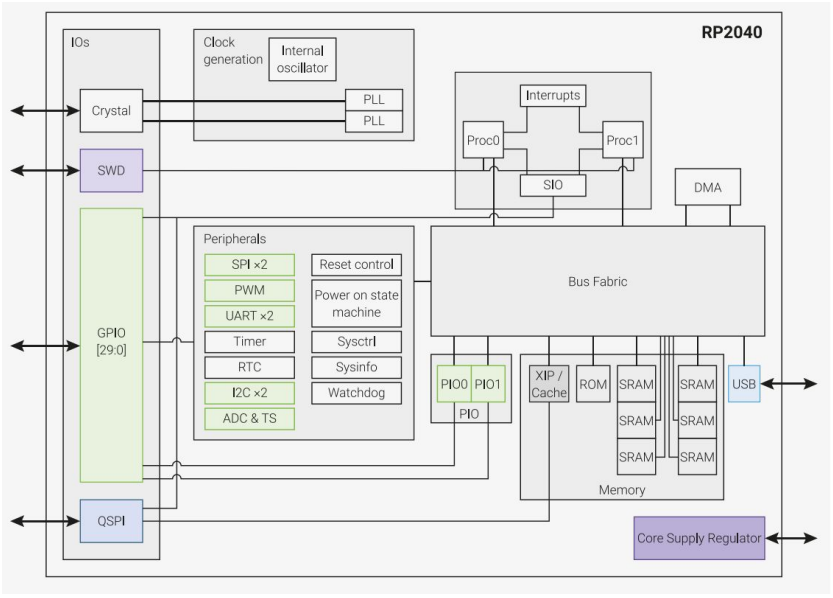
- Motion sensors
- Mini weather station
- Signal processing, audio/video
- Retro gaming
- Home automation
- Magic mirror





The hardware: RP2040

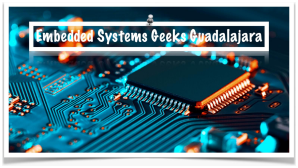




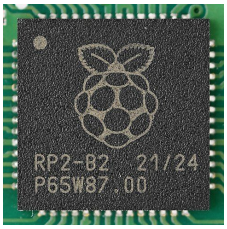
To point out:

- CPUs
- Memory
- Peripherals/GPIO

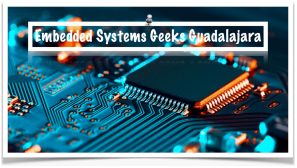
Key Features (from the RP2040 datasheet)



- Dual ARM Cortex-M0+ @ 133MHz
- 264kB on-chip SRAM in six independent banks
- Support for up to 16MB of off-chip Flash memory via dedicated QSPI bus
- DMA controller
- Fully-connected AHB crossbar
- Interpolator and integer divider peripherals
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, 4 of which can be used as analogue inputs

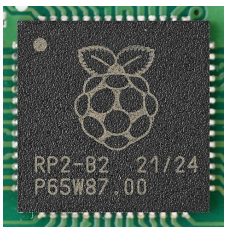


Key Features (from the RP2040 datasheet)

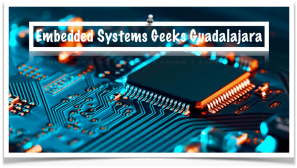


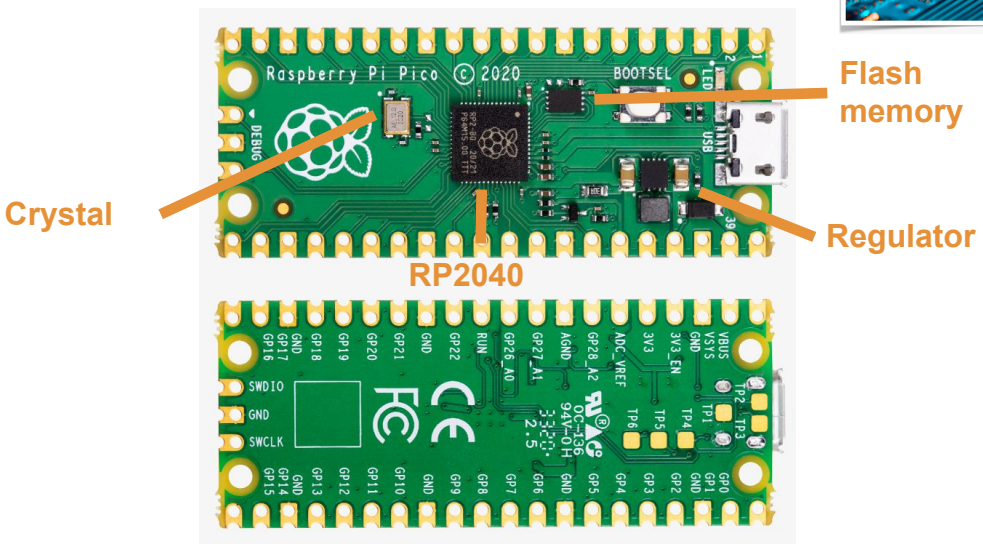
- Dual ARM Cortex-M0+ @ 133MHz
- 264kB on-chip SRAM i
- Support for up to 16M
- DMA controller
- Fully-connected AHB c
- Interpolator and intege
- On-chip programmable
- 2 on-chip PLLs to gene
- 30 GPIO pins, 4 of which can be used as analogue inputs

- Peripherals
  - 2 UARTs
  - 2 SPI controllers
  - 2 I2C controllers
  - 16 PWM channels
  - USB 1.1 controller and PHY, with host and device support
  - 8 PIO state machines



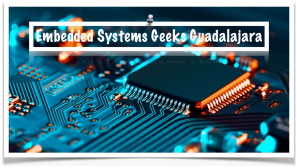
The hardware: Pico board



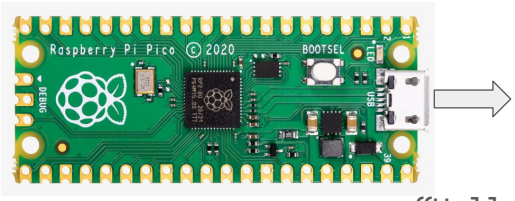


Not much to see! RP2040 is pretty self-contained.

Example: Hello, world!

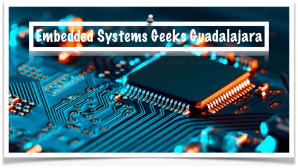



```c
#include <stdio.h>
#include "pico/stdlib.h"

int main() {
    stdio_init_all();
    while (true) {
        printf("Hello, world!\n");
        sleep_ms(1000);
    }
}
```



“Hello, world!
Hello, world!”

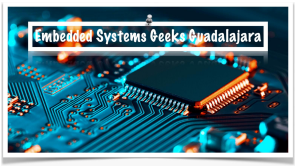
Note Pico SDK hides a lot of complexity (USB serial, etc).



Example: Blink LED

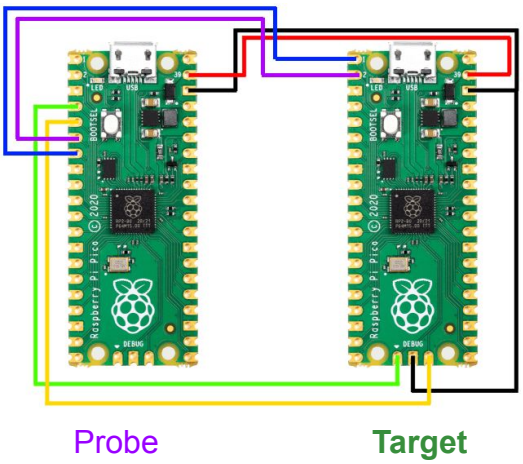
```
int LED_PIN = PICO_DEFAULT_LED_PIN;
gpio_init(LED_PIN);
gpio_set_dir(LED_PIN, GPIO_OUT);
while (true) {
    gpio_put(LED_PIN, 1);
    sleep_ms(250);
    gpio_put(LED_PIN, 0);
    sleep_ms(250);
}
```

Not working? Debug with SWD!

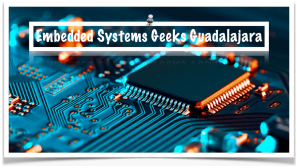


Drive a Pico using **Serial Wire Debug** and… another Pico!

Attach using GDB & OpenOCD, set breakpoints and single-step.

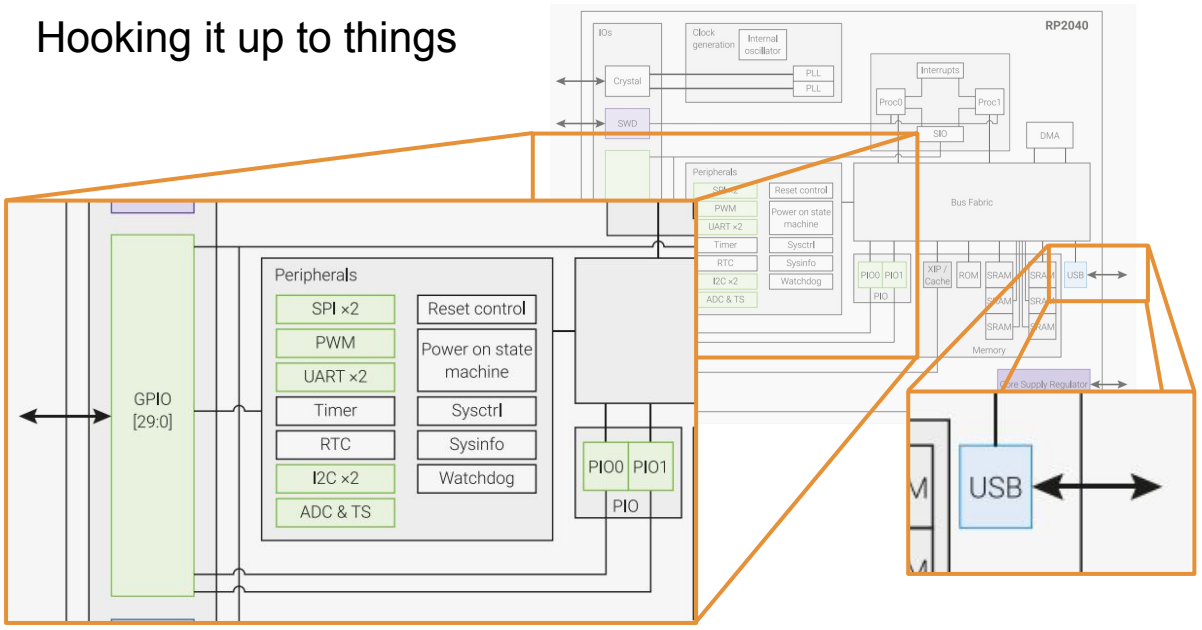


Other options for probe: a different Pi, ready-made debug probe.



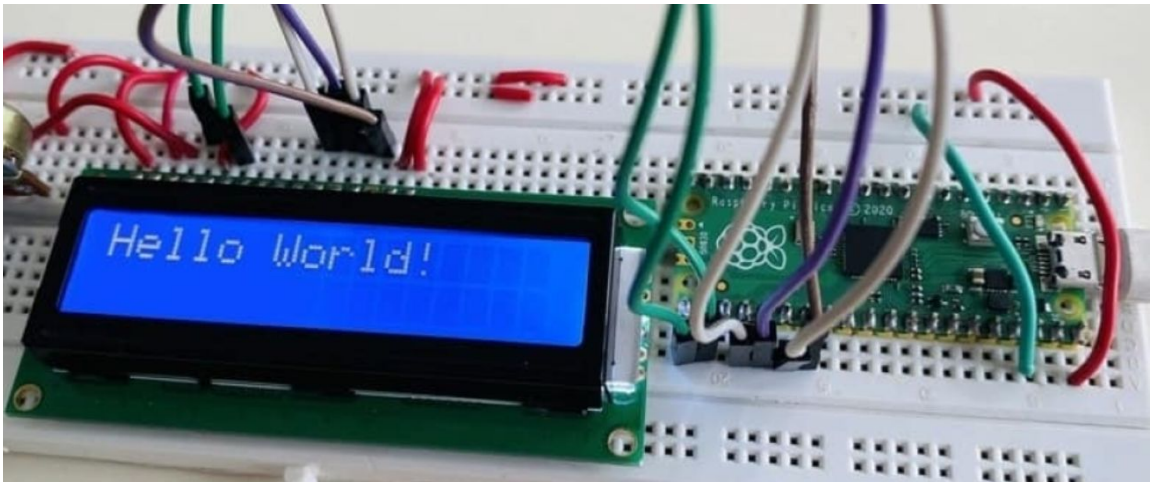
Is that all? 🤭

Hooking it up to things



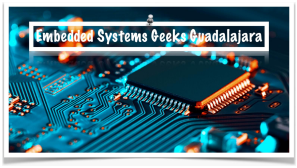
- Digital I/O: bit banging GPIO, PWM, PIO
- Protocols: USB serial, SPI, UART, I²C
- Analog I/O: ADC, temperature

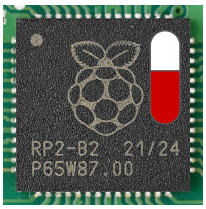
Digital I/O: LCD (SPI/I²C)



An example of digital I/O protocols: simple LCDs are commonly driven by SPI or I²C.

Analog I/O: Temperature sensor



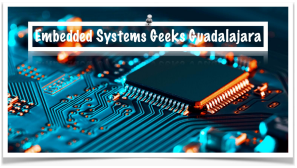


Temp (°C)	ADC value
0°	934
20°	891
50°	827

Example of analog I/O: approximate temp from internal sensor.

Note: extremely voltage-sensitive & measures hot.

HATs / accessories

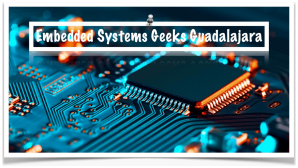


And when you're hooking it up to things…

Lots of HATs and other accessories available to simplify and augment what it can do! (Also: cases)

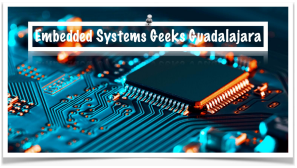
Simple connections (USB power), ports (VGA), and complex components (DAC).

Cases help keep it rugged and make it look nice.

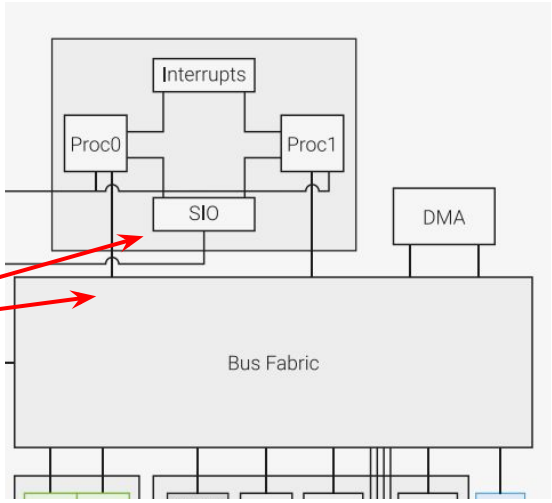


Let’s see the fancy stuff! 🧐

Dual processor cores

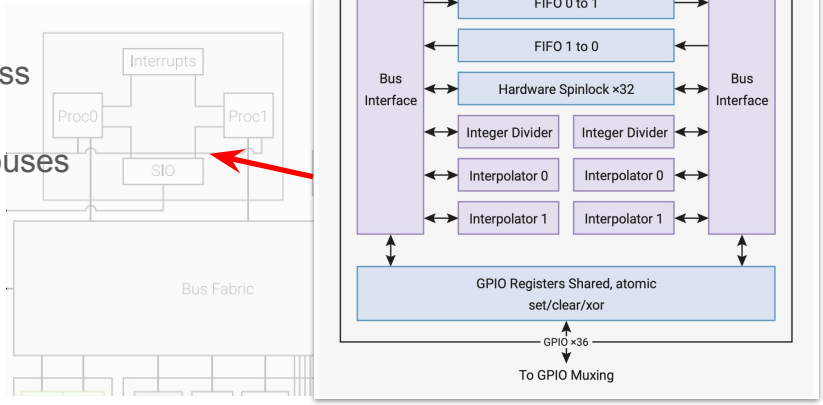


- 133 MHz clock, but **two cores**
- Proc1 waits in “deep sleep” at startup, started using multicore_launch_core1.
- Cores can perform tasks independently:
  - But they access the same ROM and SRAM; can result in contention.
  - Pico provides SIO and Bus systems to coordinate / parallelize accesses.

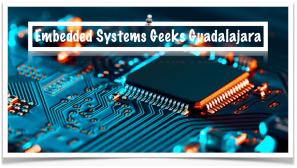


Processor subsystem

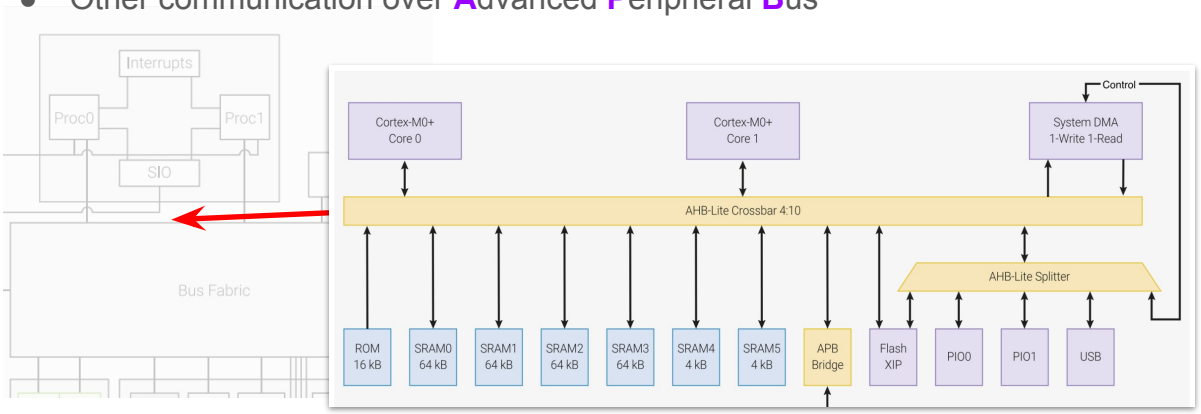
- Coordination using interrupts, events, and **S**ingle-cycle **IO** block
- High-speed, deterministic access
- Faster than other general-purpose buses



Processor Bus Fabric



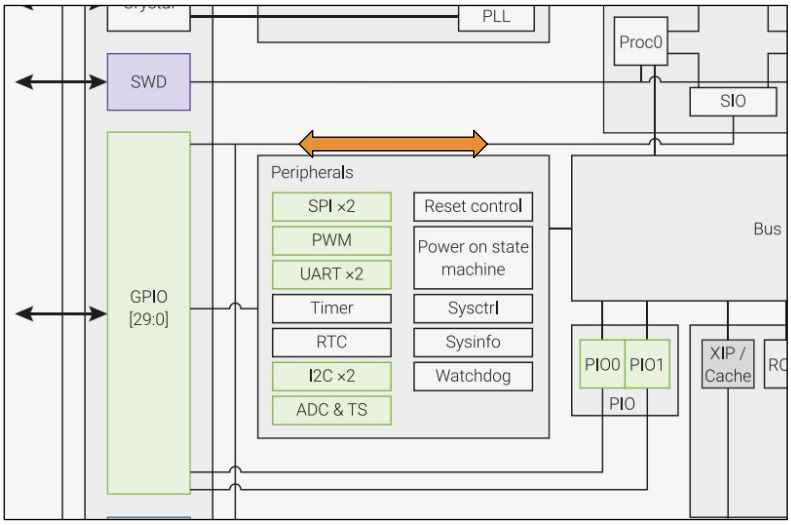
- Pico provides an **AHB-Lite Crossbar** to parallelize memory accesses.
  - AHB = **A**dvanced **H**igh-performance **B**us
- Other communication over **A**dvanced **P**eripheral **B**us



PIO: Programmable I/O

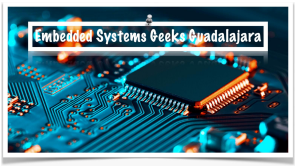
It's **possible** for the CPUs to drive GPIO…

…but has **disadvantages**.
Any ideas what they are?



Disadvantages of **bit-banging**: timing, CPU cycles, complexity

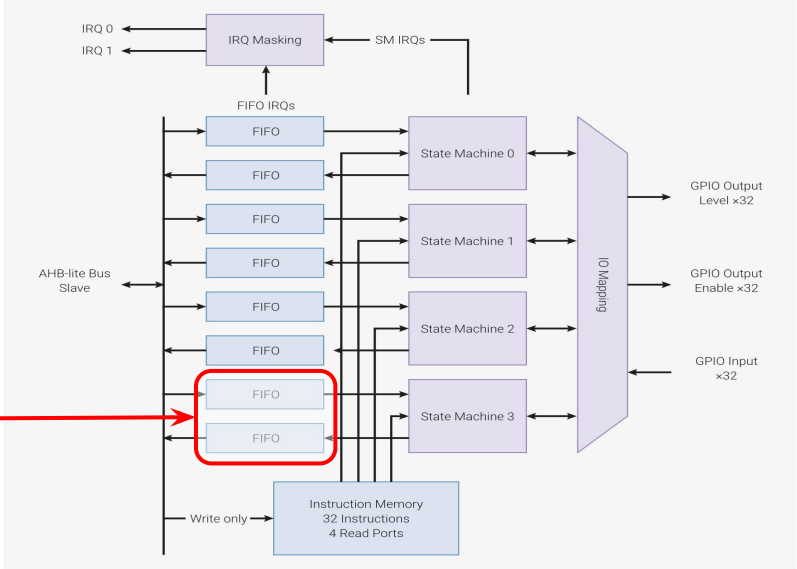
PIO: Programmable I/O



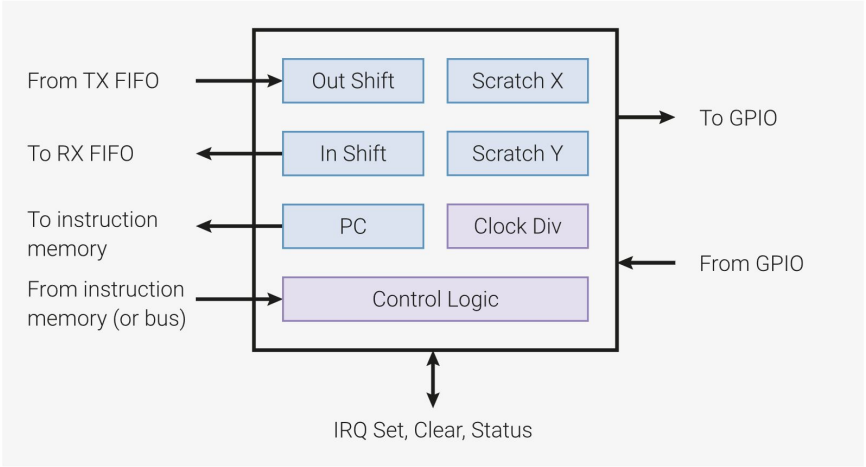
- Instead of **bit-banging**, use PIO: tiny coprocessors that can generate custom signals.
  - Called state machines in datasheet.
- PIOs can be programmed in a mini **assembly** language.
- There are **two** identical PIO blocks:
  - Each PIO block merely provides memory for 32 instructions.
  - Each PIO block comes with 8 FIFO buffers.
  - Each PIO block provides 4 independent state machines (processors).
  - Inputs and outputs of PIOs can be mapped freely to GPIOs.
  - PIOs can moreover generate interrupts!
- Despite the above limitations, they are extremely useful to generate nonstandard or bespoke signals at high frequency (even video signals).

PIO Architecture

- FIFO buffers: 4 x 32 bit words
- Decouple timing of PIO state machines and system bus.
- A pair of TX/RX FIFOs can be merged into a single 8 x 32 bit FIFO.

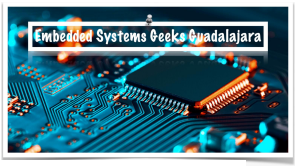


PIO State Machine



- Shift registers pull/push data from/to GPIOs
- X/Y Scratch registers hold intermediate values
- Clock division to fine-tune phase length

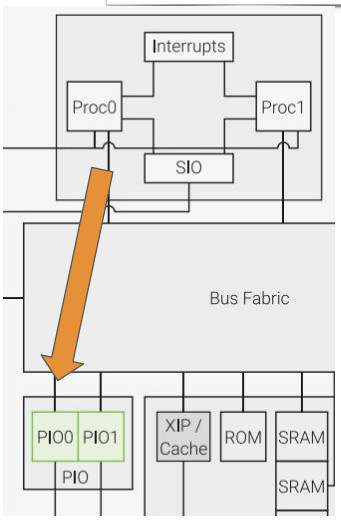
PIO Assembly Language



Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				

PIO Example Setup

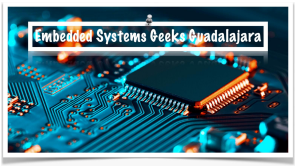
```c
#include "hardware/pio.h"

// Our assembled program (array of instrs):

#include "hello.pio.h"


int main() {

    PIO pio = pio0;

    for (int i = 0; i < count_of(hello_program_instructions); ++i)

        pio->instr_mem[i] = hello_program_instructions[i];

    // ... (other setup) ...

}
```



Program loads state machine instructions from array into PIO instruction memory.

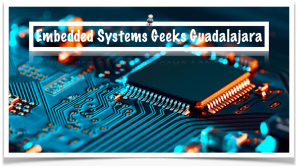
These can be any array of uint16s, but it’s simpler to use pioasm and include instruction data from its generated header files.

PIO Takeaways



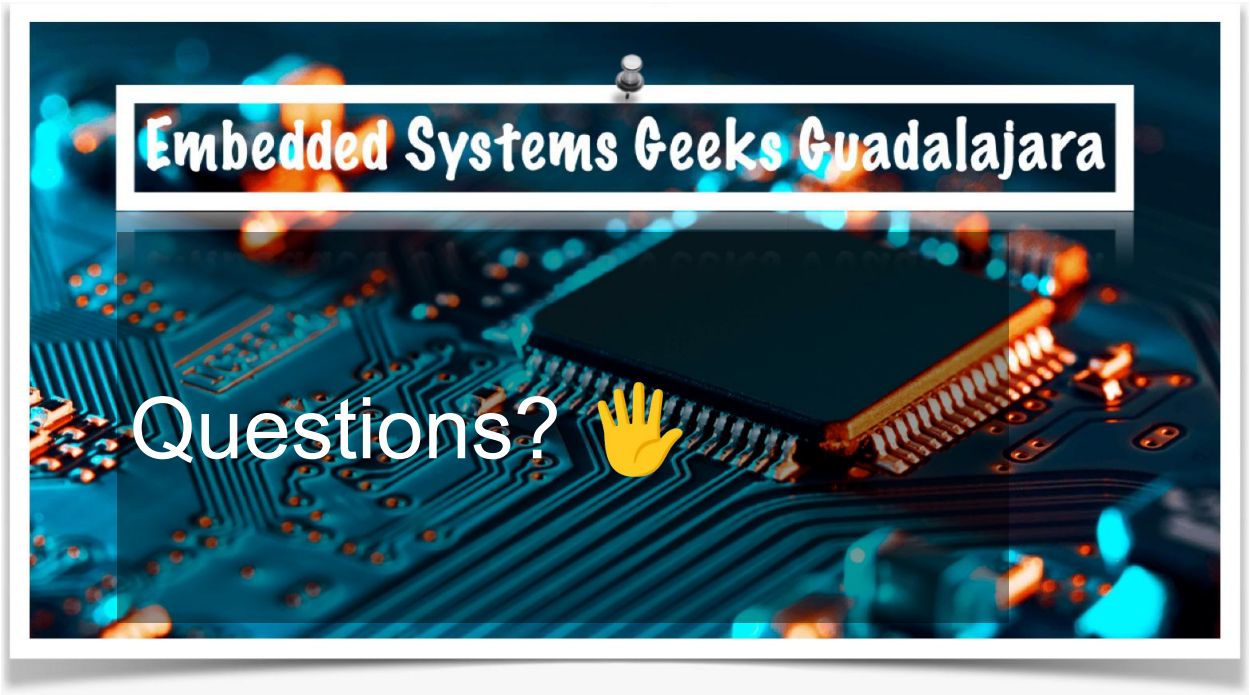
- PIOs let us avoid any **bit-banging** of nonstandard signals.
- Instead, we do something akin to word-banging:
  - Feed the FIFOs with 32 words.
  - This can be done at a 1/32 lower rate than outputting individual bits.
  - Even better: we can use **DMA channels** to feed the FIFOs.
- Rationale: produce complex signals without using up any CPU resources.
- Similar idea to GPUs on modern PCs and laptops.
  - The idea of coprocessors goes back to the 80s. E.g., the Amiga 500 already had several coprocessors with cute names, such as Copper, Blitter, Paule and Denise.
  - **DMA** was likewise used. May result in the **chip memory** bus becoming a bottleneck …
- PIOs *somewhat* make up for the lower clock speed of the Pico.

DMA

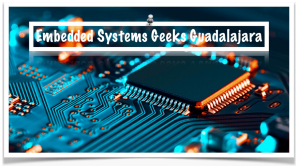


DMA = Direct Memory Access

- Not just the CPU cores but other components access the memory bus.
- DMA controllers typically copy/shift memory (faster than the CPU).
  - RP2040 DMA can perform one read&write access every clock cycle.
- Can be memory-to-memory or memory-to/from-peripheral (e.g, PIO FIFO)
- 12 programmable DMA channels are available.
  - Combining channels can implement more sophisticated behavior, with one channel programming another in a kind of ping-pong fashion.
  - Channels can generate interrupts too.
- Additional features: **Pacing Timers** and **CRC** checksum calculation.


Embedded Systems Geeks Guadalajara
Questions? 🖐

Helpful resources



- https://github.com/raspberrypi/pico-examples
- https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf
- https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf
- https://github.com/raspberrypi/pico-examples
- Cortex-M0+ reference manual: https://developer.arm.com/documentation/ddi0486/


Embedded Systems Geeks Guadalajara
Thank you!