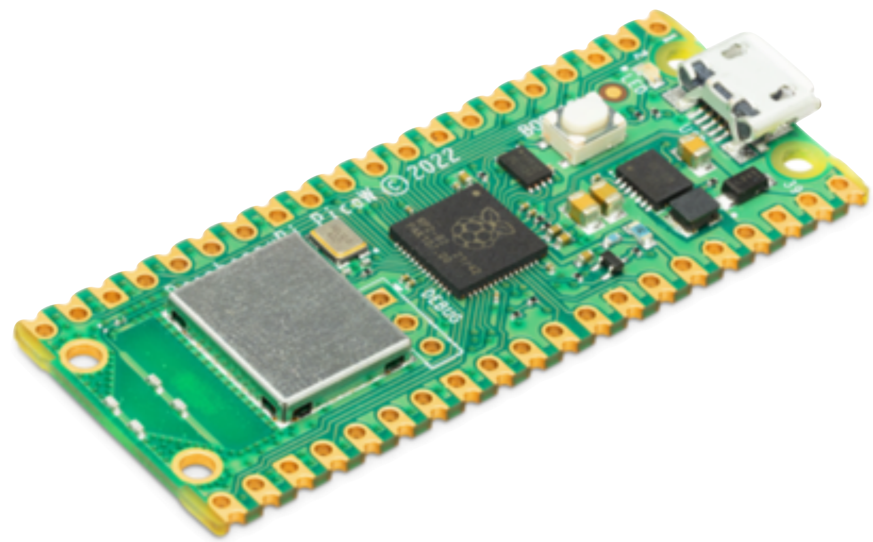
Jim Kring

Presented on May 23, 2023 at

ni CONNECT

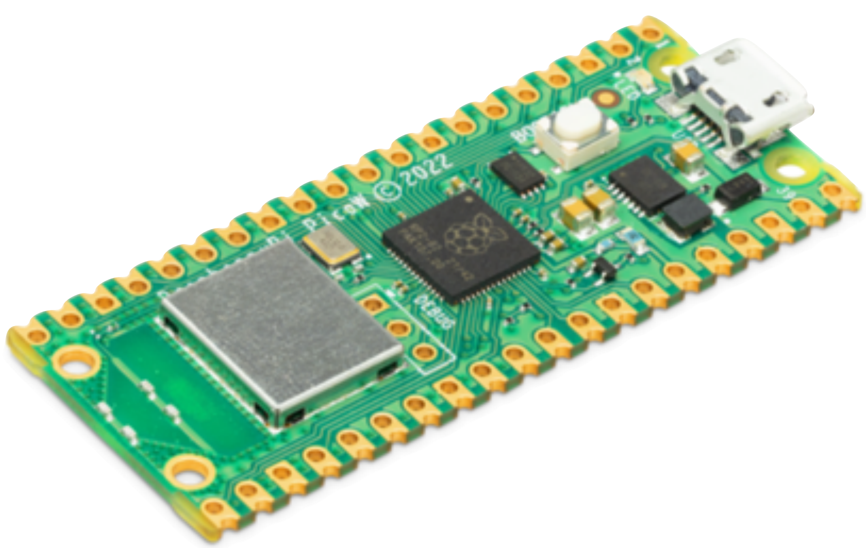


G Anywhere

Enhancing LabVIEW Development with a Cross-Platform Embedded Device Library

we’ll cover these topics

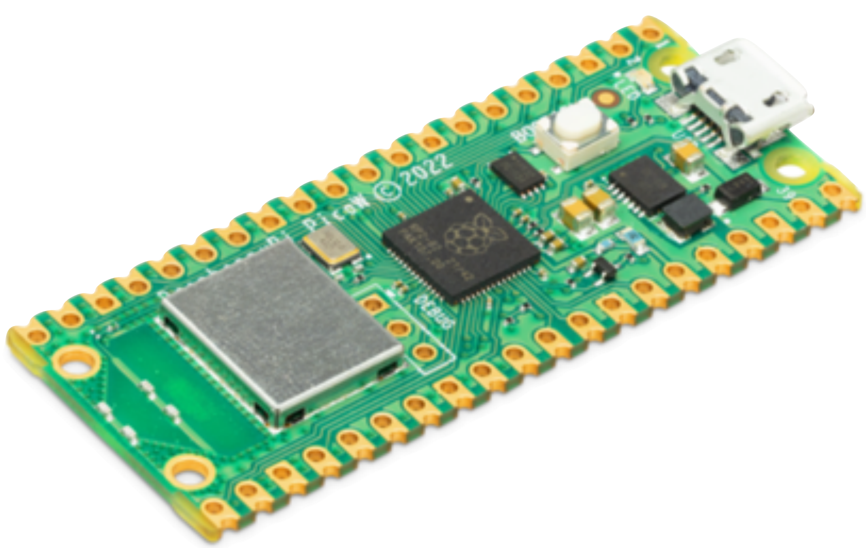
we’ll cover these topics



Raspberry Pi Pico

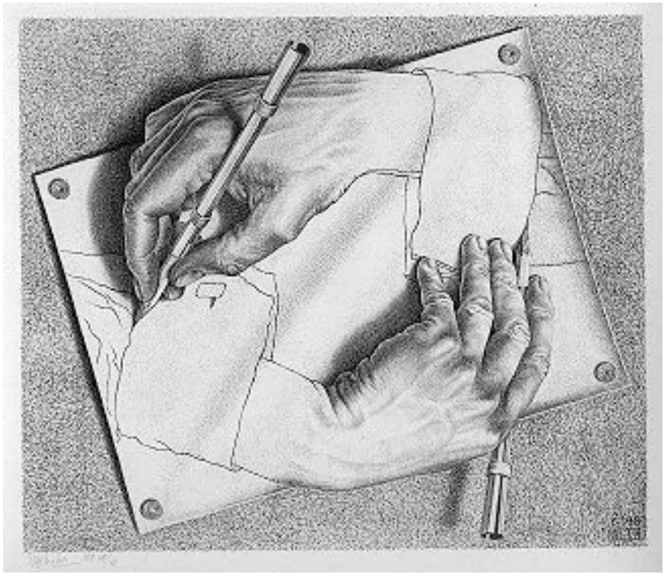
a great place to run LabVIEW code

we'll cover these topics



Raspberry Pi Pico

a great place to run LabVIEW code

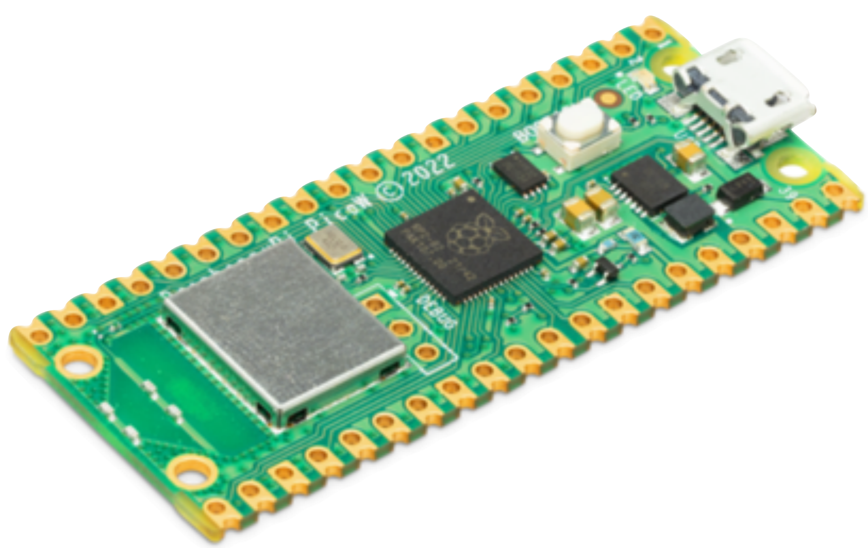


Compiling G in G

a way to run LabVIEW anywhere

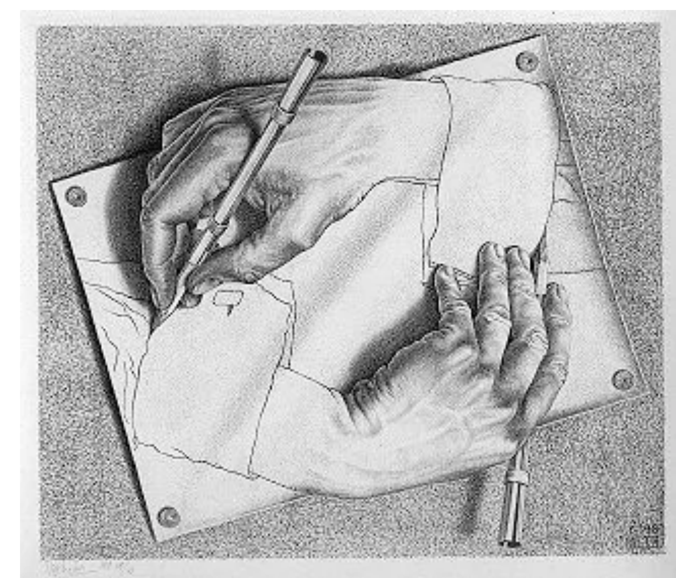
we'll cover these topics

and see some demos!



Raspberry Pi Pico

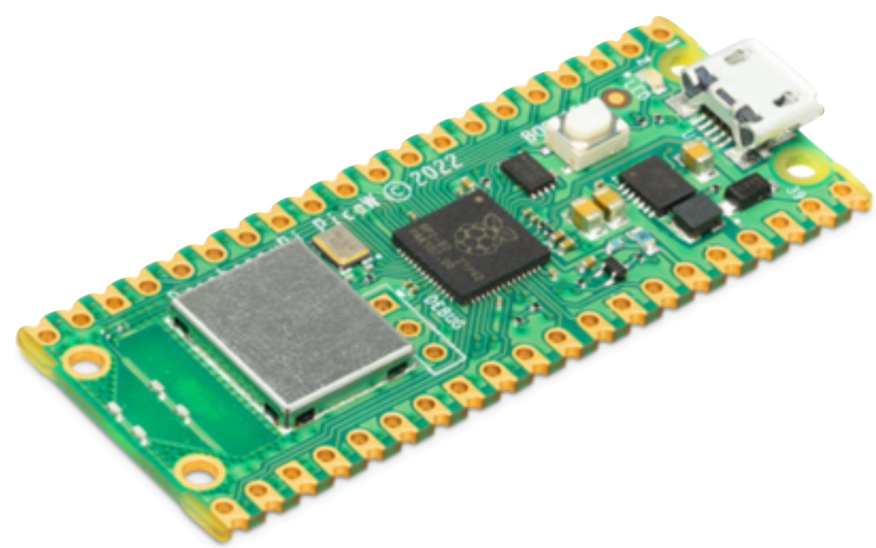
a great place to run LabVIEW code



Compiling G in G

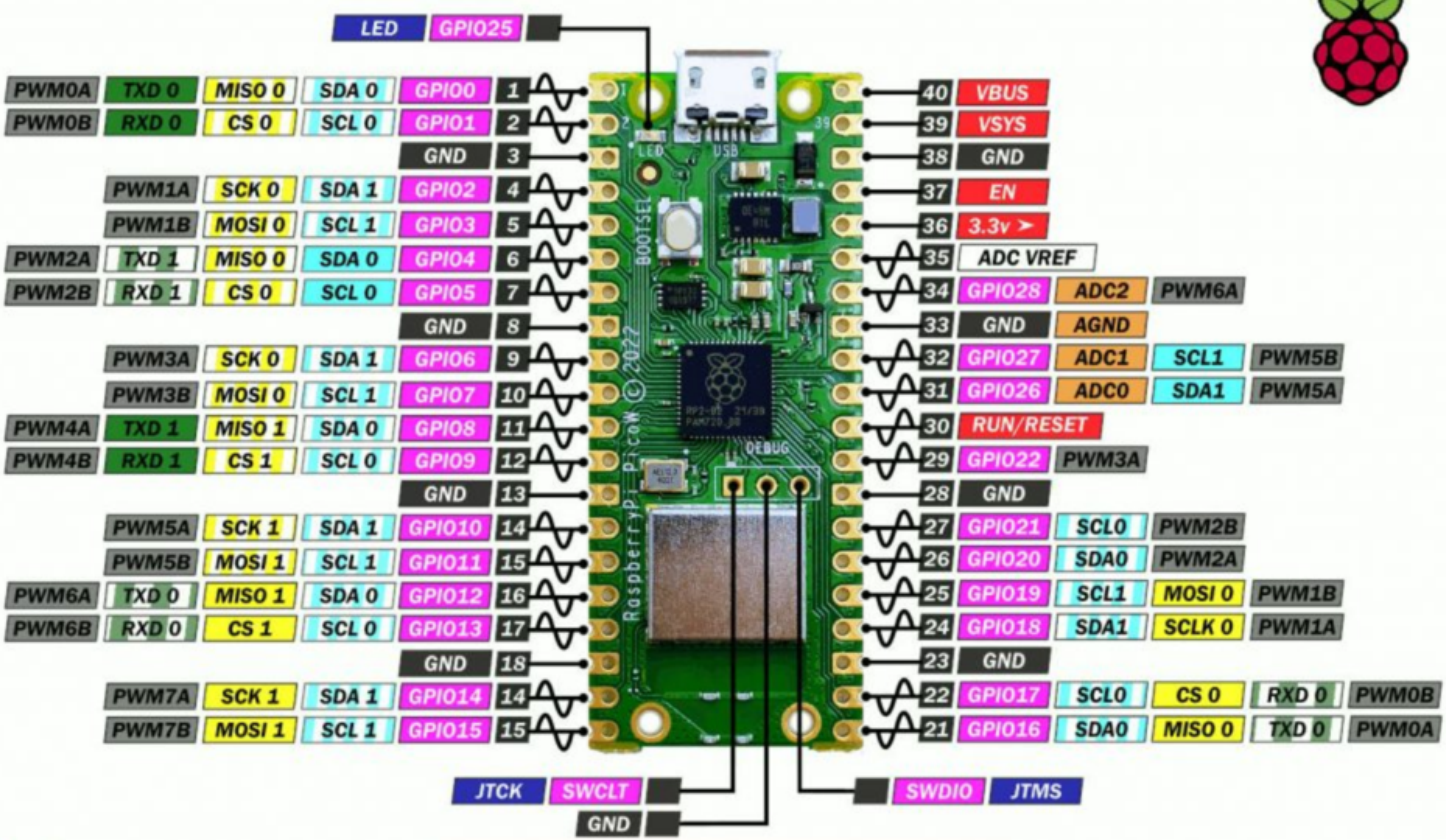
a way to run LabVIEW anywhere

Raspberry Pi Pico



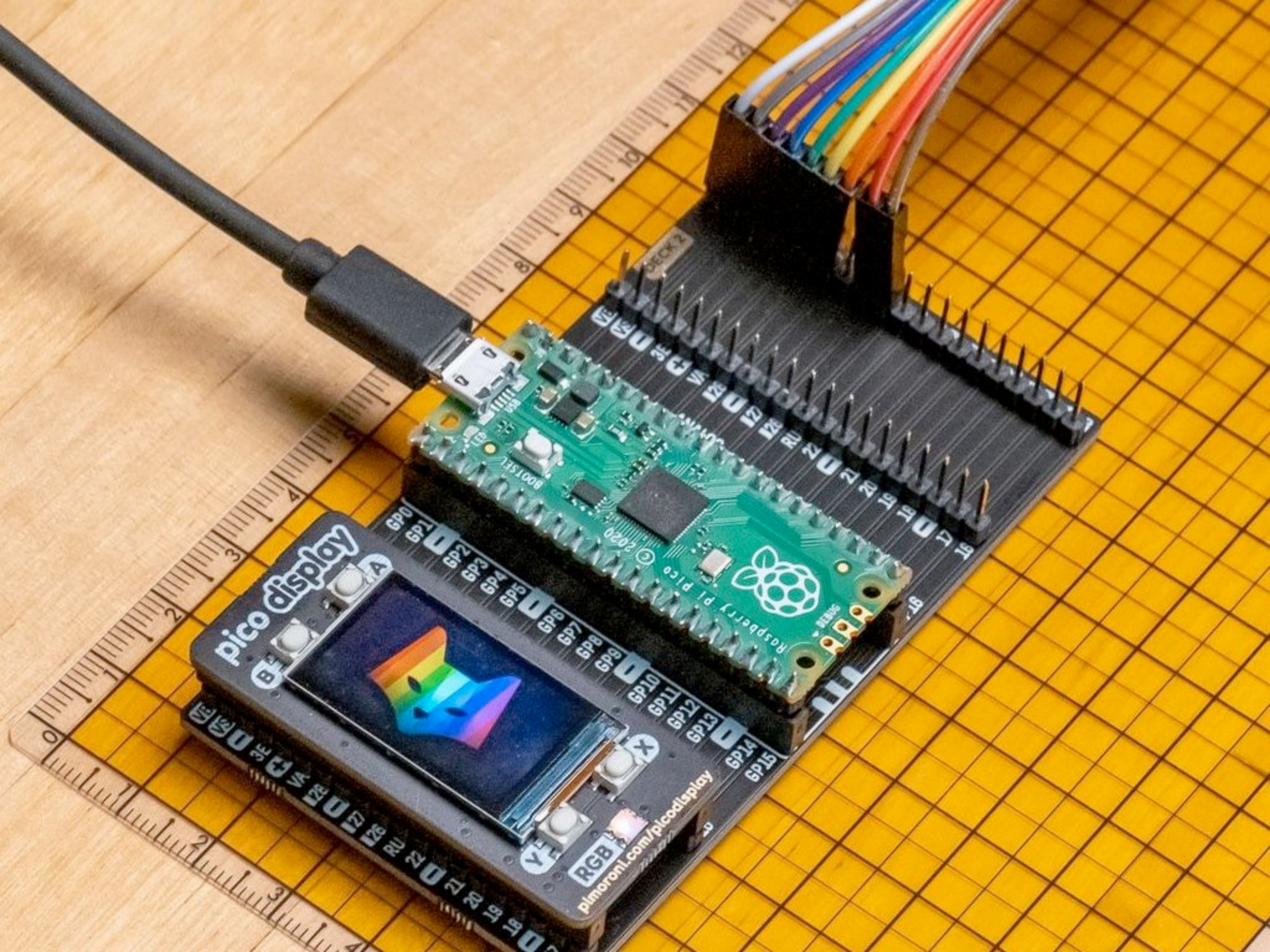
a great place to run LabVIEW code

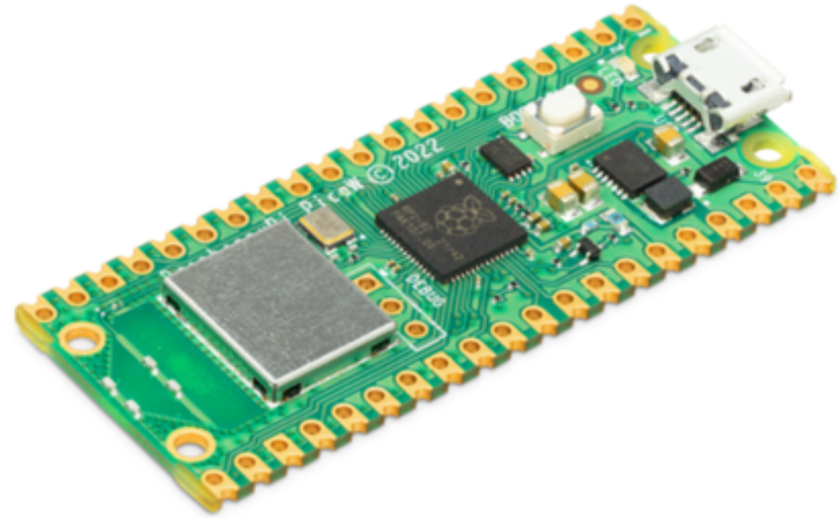
Raspberry Pi Pico rp2040 **PINOUT**



Internal board function pins	
IP Used in ADC mode (ADC3) to measure VSYS/3	GPIO29
OP Connected to user LED	GPIO25
IP VBUS sense - high if VBUS is present, else low	GPIO24
OP Controls the on-board SMPS Power Save pin	GPIO23

www.mischianti.org


pico display
A
B
X
Y
RGB
pimoroni.com/picodisplay
Raspberry Pi Pico
BOOTSEL
DEBUG
USB
LED
2020
GP0
GP1
GP2
GP3
GP4
GP5
GP6
GP7
GP8
GP9
GP10
GP11
GP12
GP13
GP14
GP15



Raspberry Pi Pico W

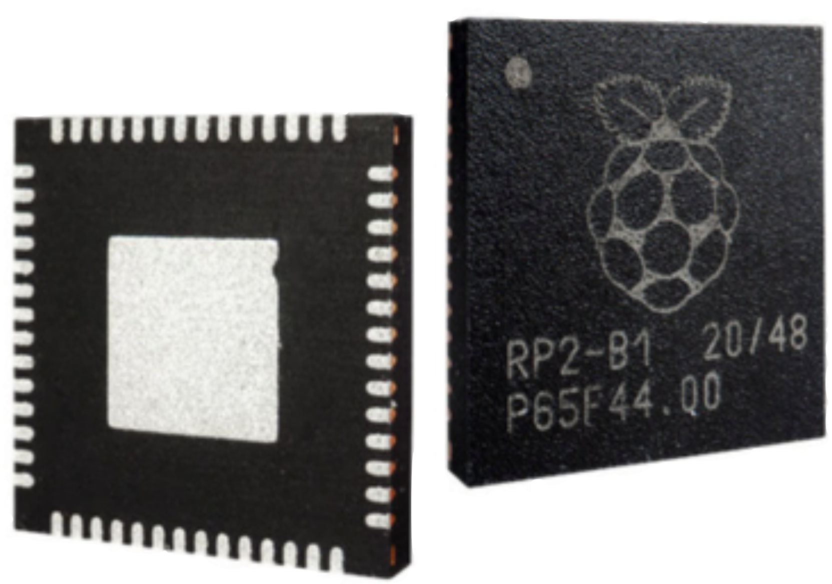
$6.00

2-Channel RS232 Module for Raspberry Pi Pico, SP3232EEN Transceiver, UART To RS232

$8.95

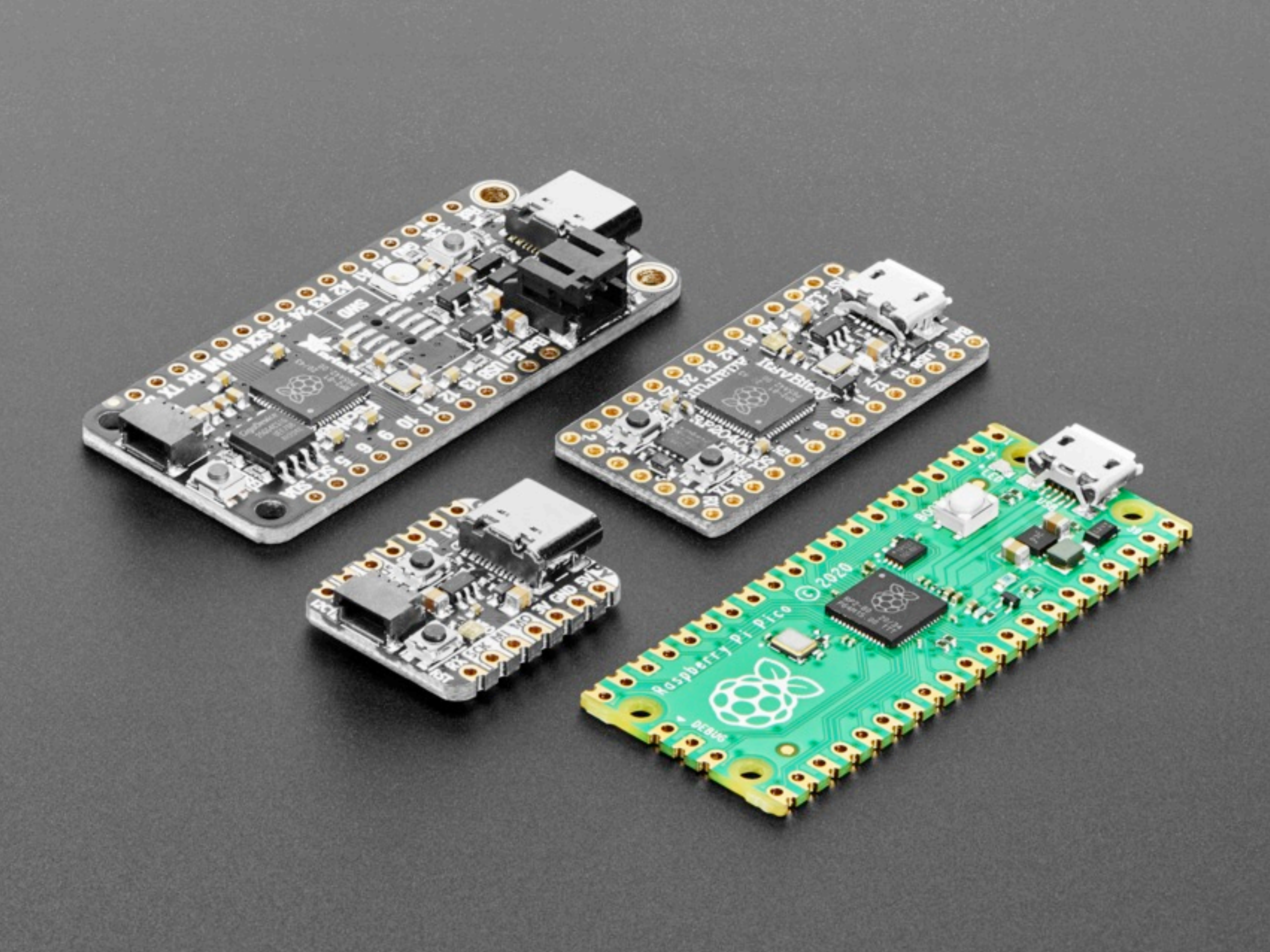
CAN Bus Module for Raspberry Pi Pico, UART to CAN conversion

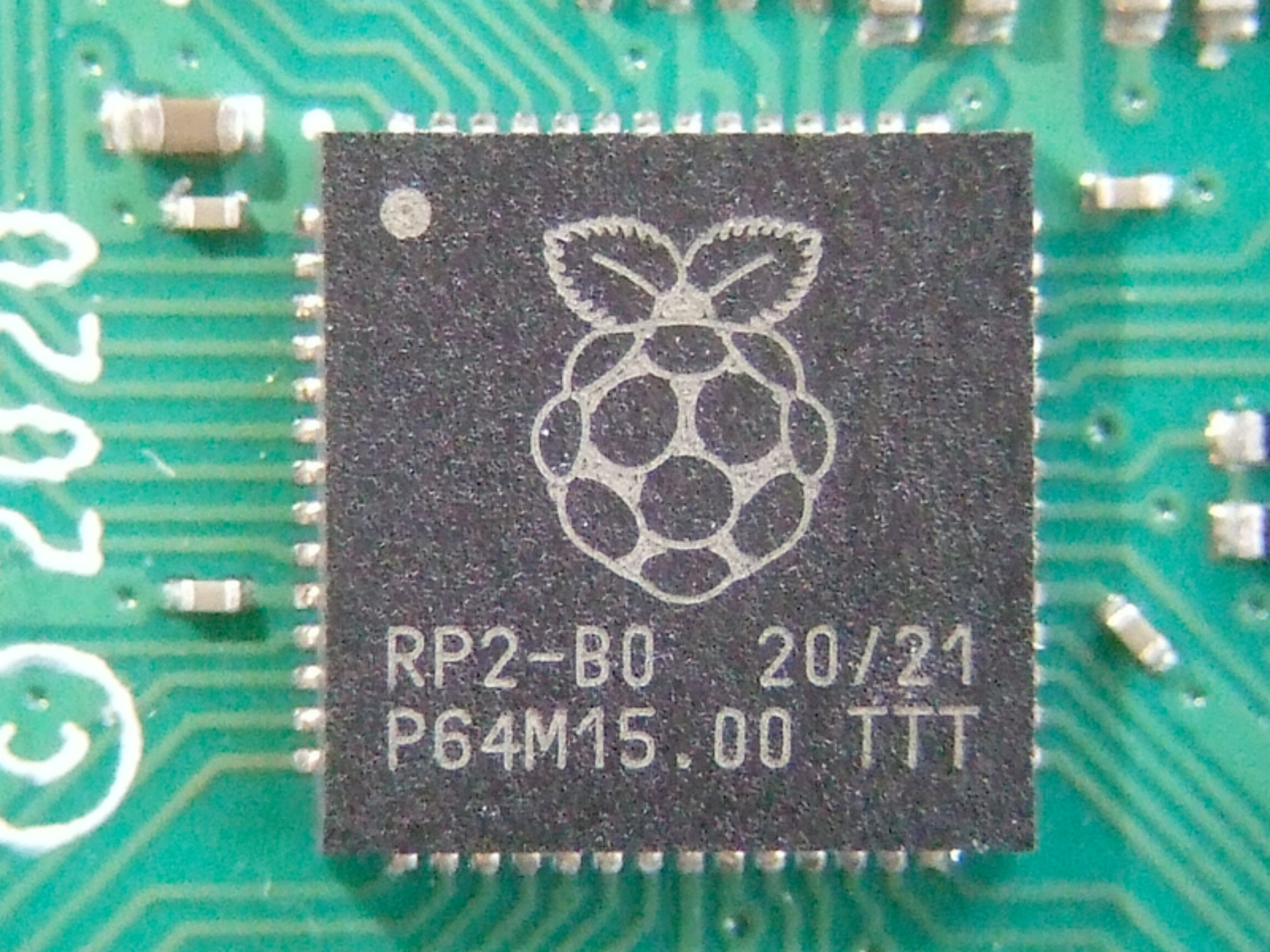
$24.95



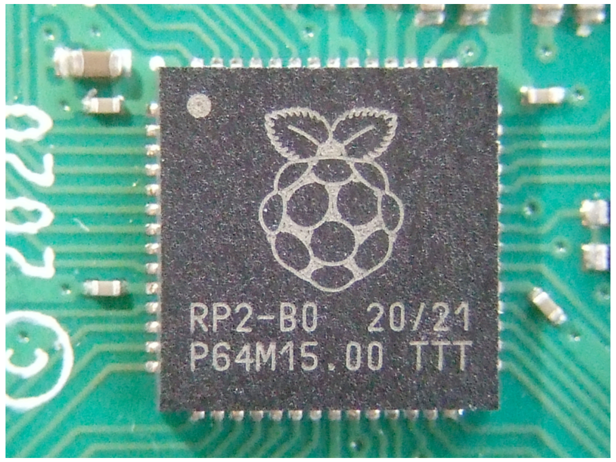
Raspberry Pi RP2040

$1.00


Raspberry Pi Pico
© 2020
DEBUG

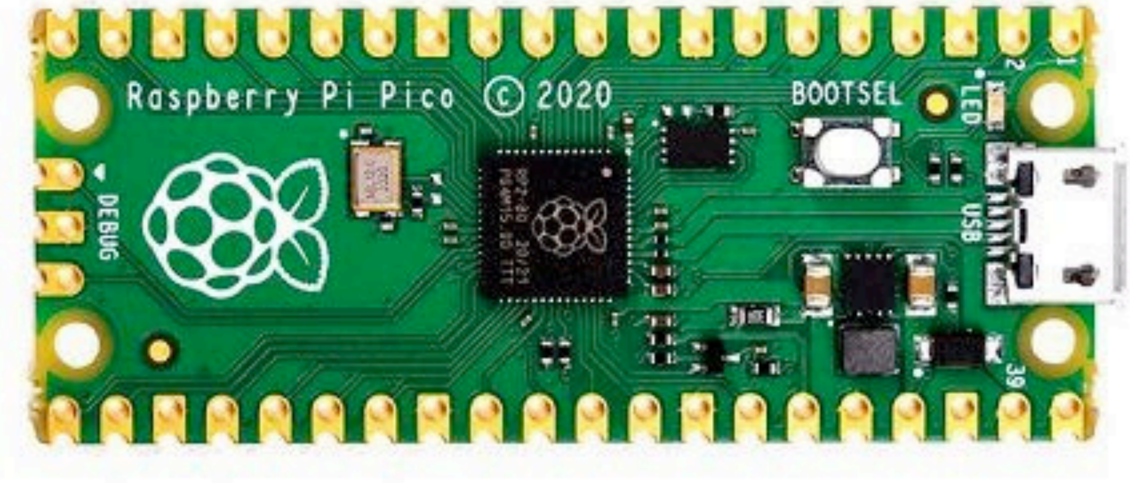

RP2-B0 20/21
P64M15.00 TTT

RP2040 µC



- Dual-core Arm Cortex-M0+ processor, flexible clock up to 133 MHz
- **264kB on-chip SRAM**
- 2 × UART, 2 × SPI controllers, 2 × I2C controllers, 16 × PWM channels
- 1 × USB 1.1 controller and PHY, with host and device support
- 8 × Programmable I/O (PIO) state machines for custom peripheral support
- Low-power sleep and dormant modes. Temperature sensor.
- Accelerated integer and floating-point libraries on-chip

embedded software
no operating system



embedded software
no operating system

264kB on-chip SRAM

embedded software
no operating system

264kB on-chip SRAM

MicroPython

embedded software
no operating system

264kB on-chip SRAM

MicroPython

embedded software
no operating system

264kB on-chip SRAM

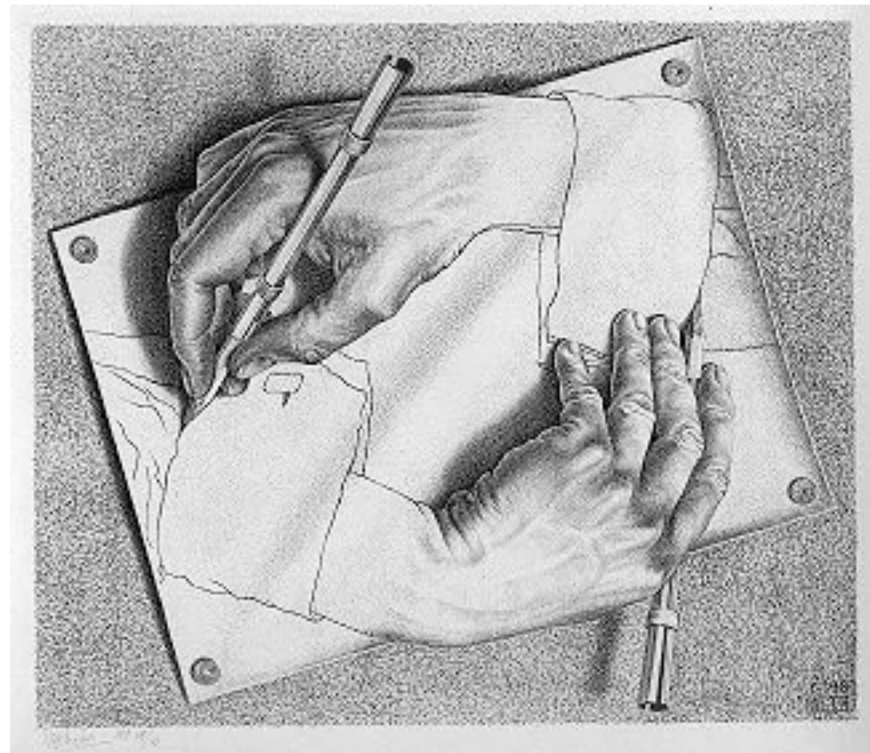
MicroPython

embedded software
no operating system

264kB on-chip SRAM

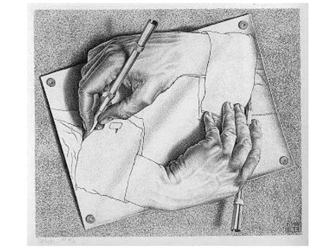
?

LabVIEW™

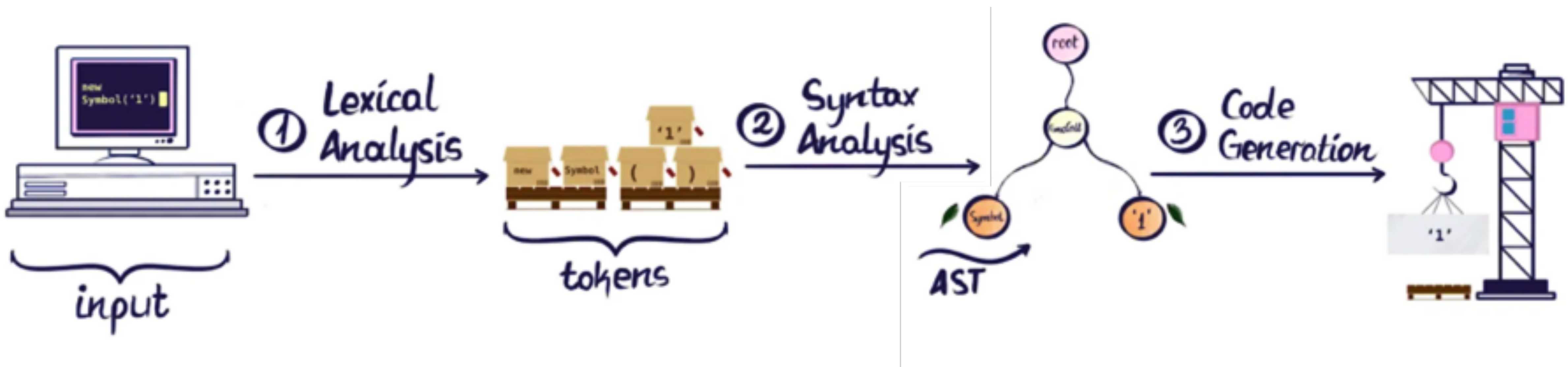


Compiling G with G

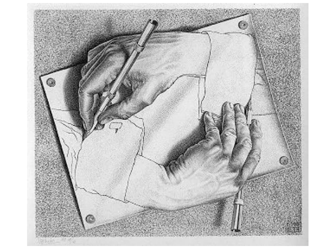
Proving to ourselves that G is a real programming language by writing a compiler in G that can compile itself



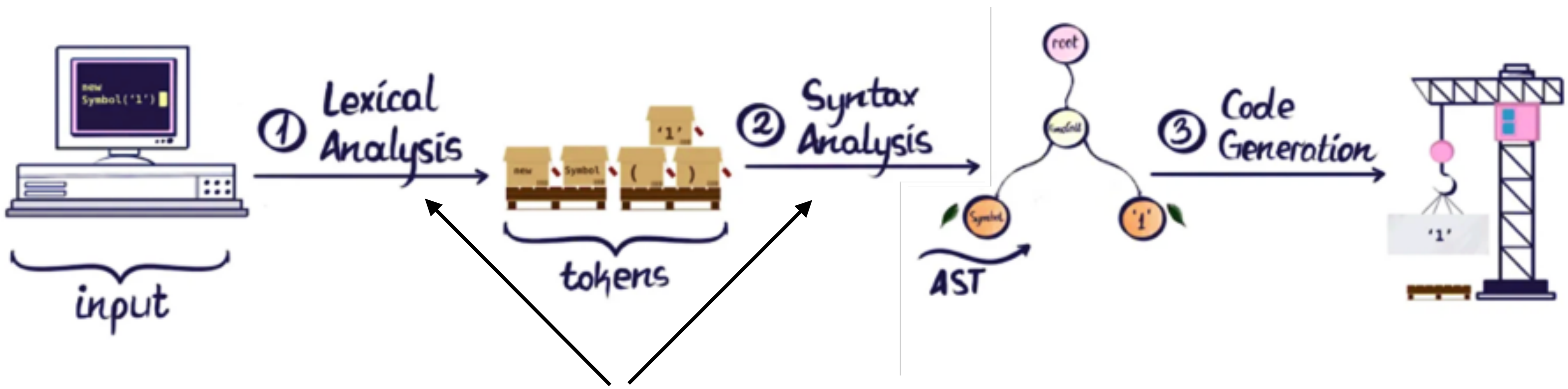
What a Compiler does



A compiler transforms instructions from a source format to some other target format, so it can be executed by a machine.

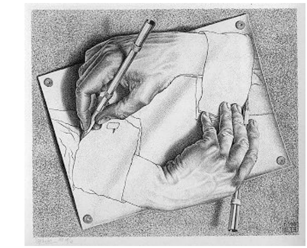


What a Compiler does

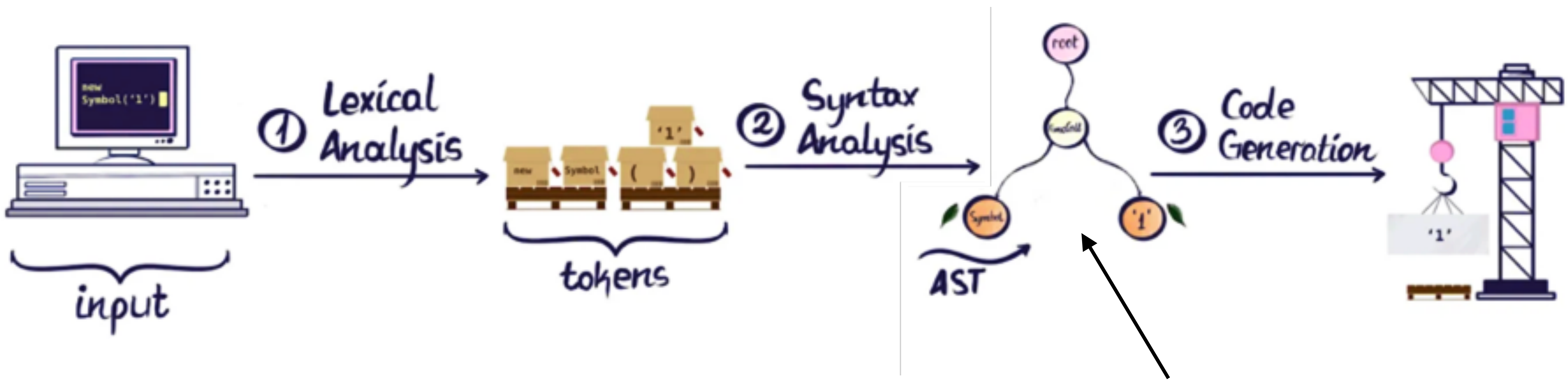


applicable for text-based languages

A compiler transforms instructions from a source format to some other target format, so it can be executed by a machine.

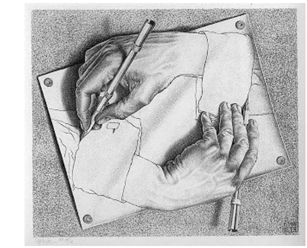


What a Compiler does

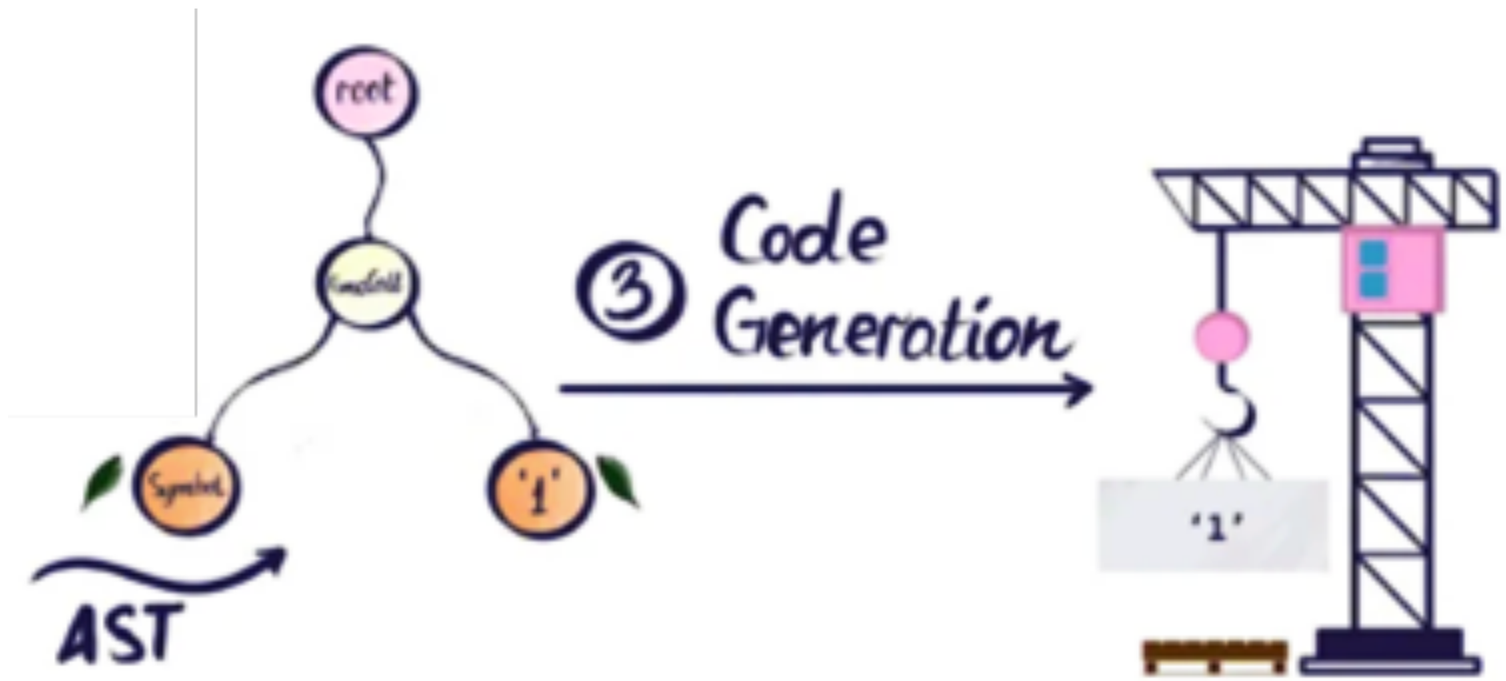


this looks a lot like a graph (nodes and edges)

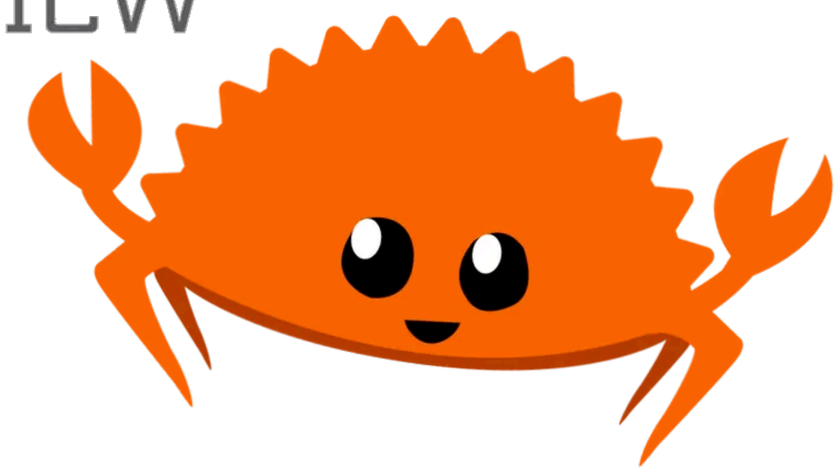
A compiler transforms instructions from a source format to some other target format, so it can be executed by a machine.



What a G Compiler needs to do

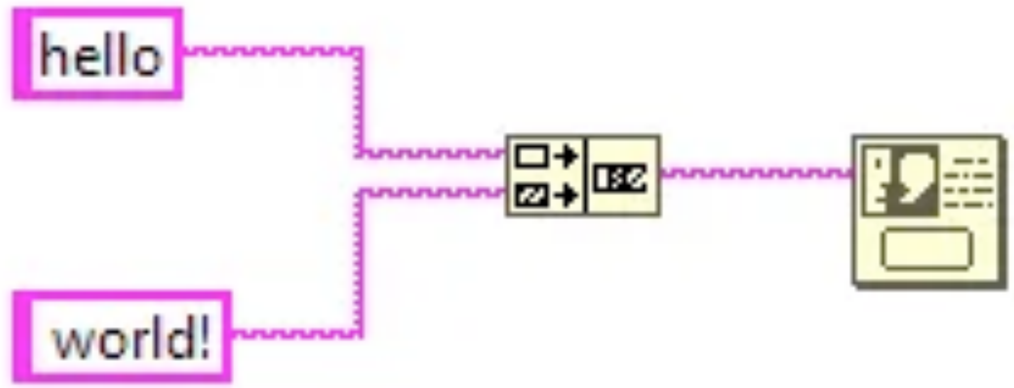


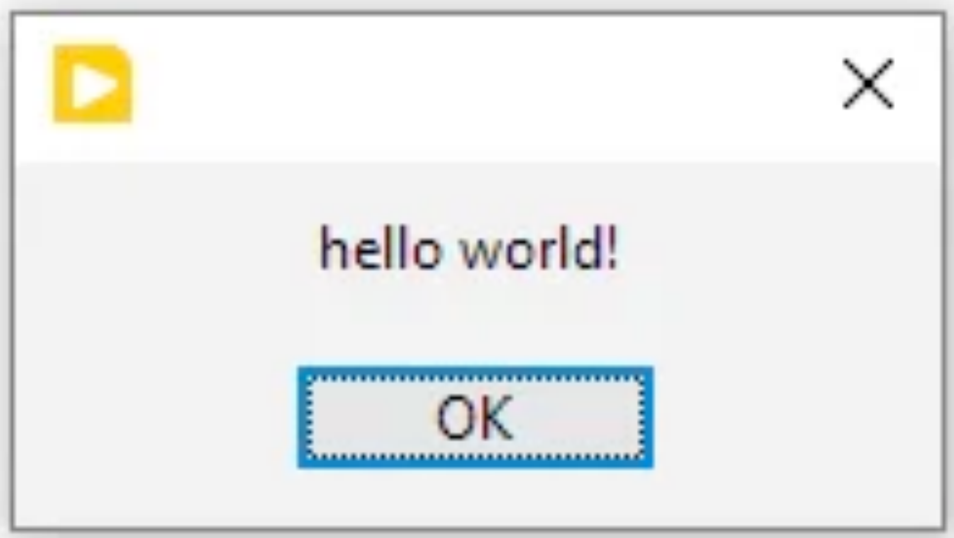
A G compiler needs to generate code from a program's data flow and control flow graphs (DFG and CFG).

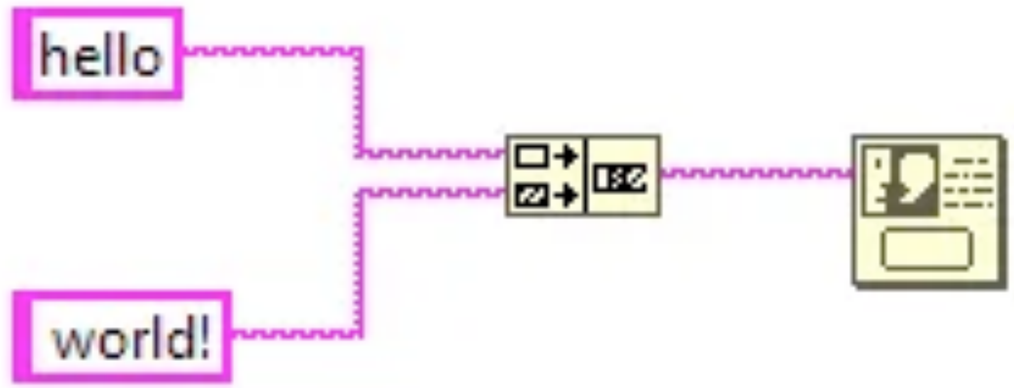


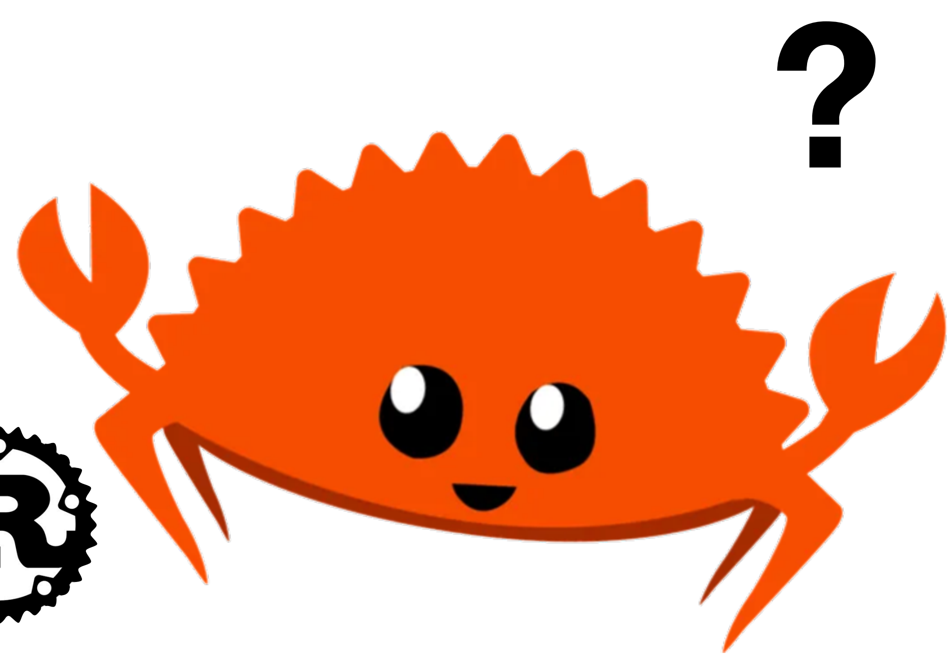
Translating G to Rust

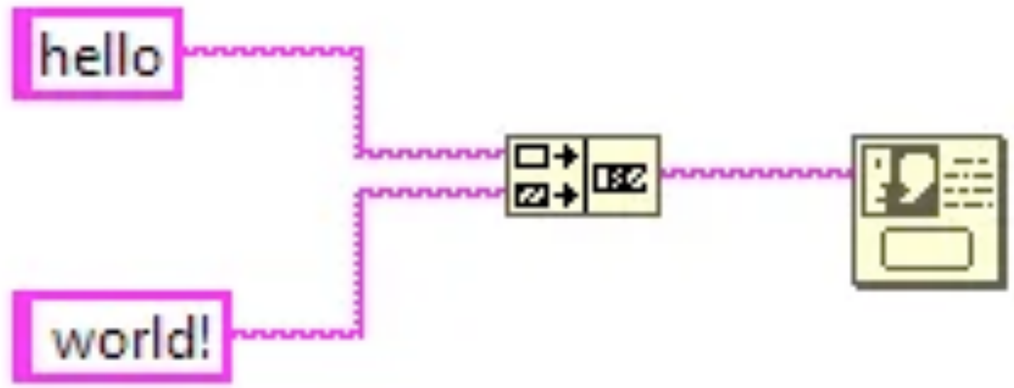
Mapping G’s parallelism to text-based concurrency as an intermediate representation.


hello
world!

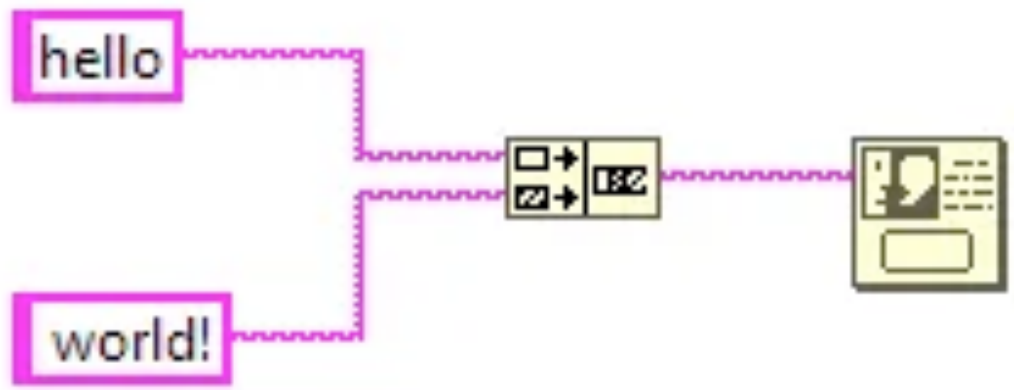

hello world!
OK


hello
world!



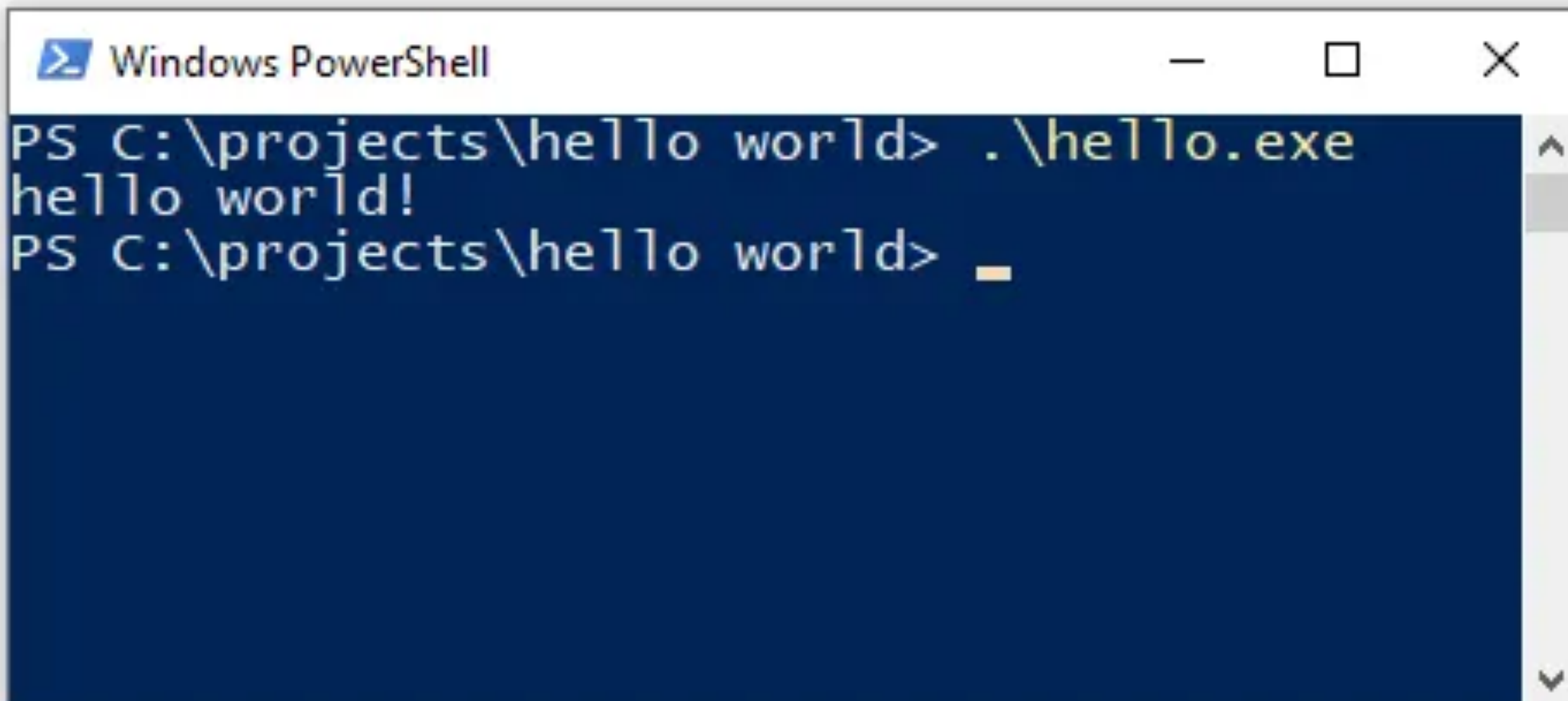



```
let my_string_01: String = "hello".to_string();

let my_string_02: String = " world!".to_string();

let my_string_03: String = my_concatenate_strings(
    my_string_01, // upper left input wire
    my_string_02  // lower left input wire
);

my_print(
   my_string_03
);
```

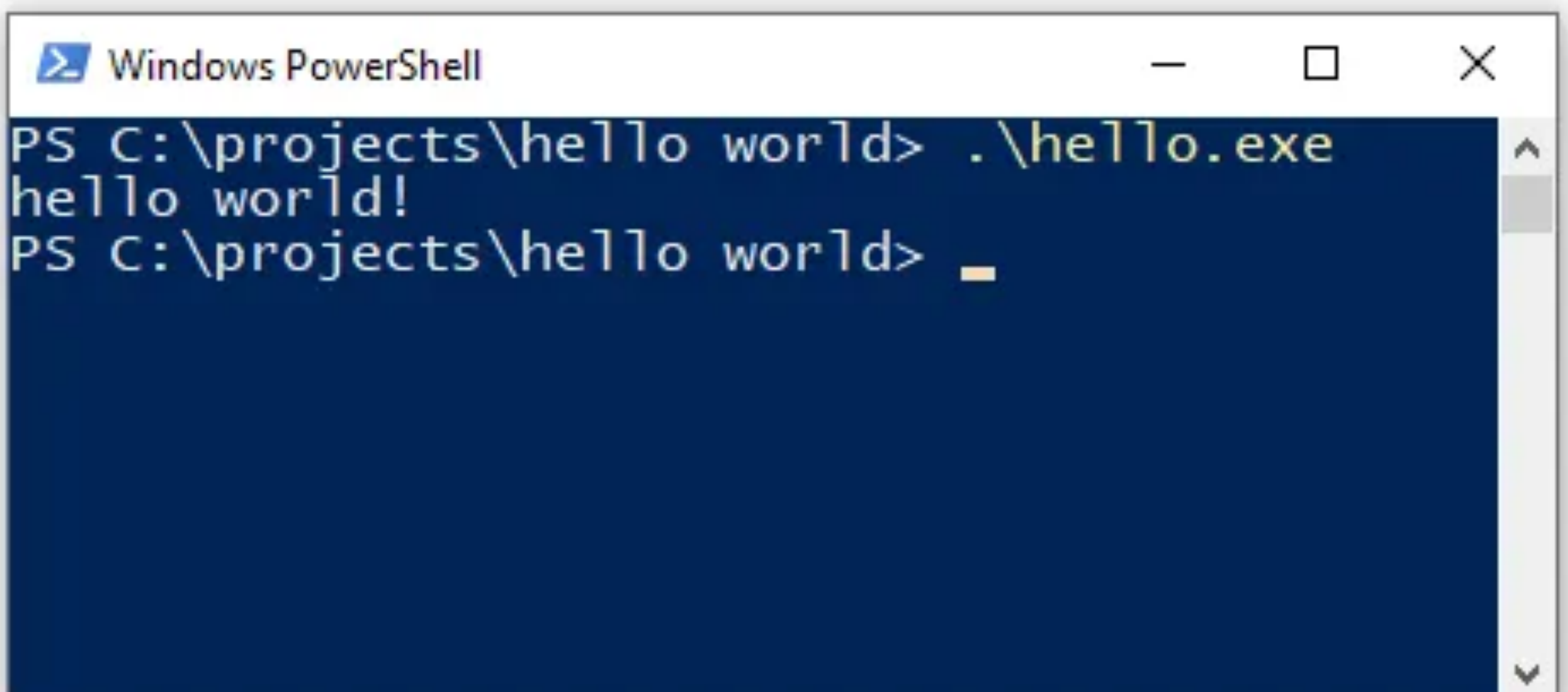


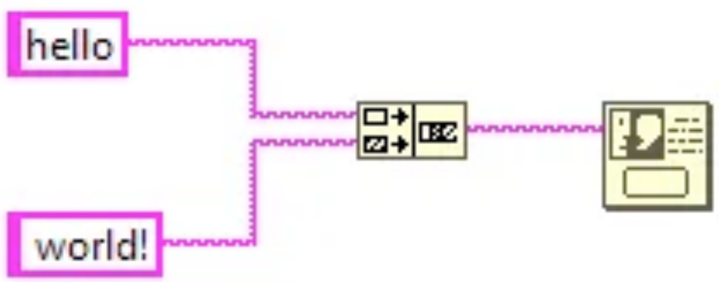

```
let my_string_01: String = "hello".to_string();

let my_string_03: String = my_concatenate_strings(
    my_string_01, // upper left input wire
    my_string_02  // lower left input wire
);

my_print(
  my_string_03
);

let my_string_02: String = " world!".to_string();
```

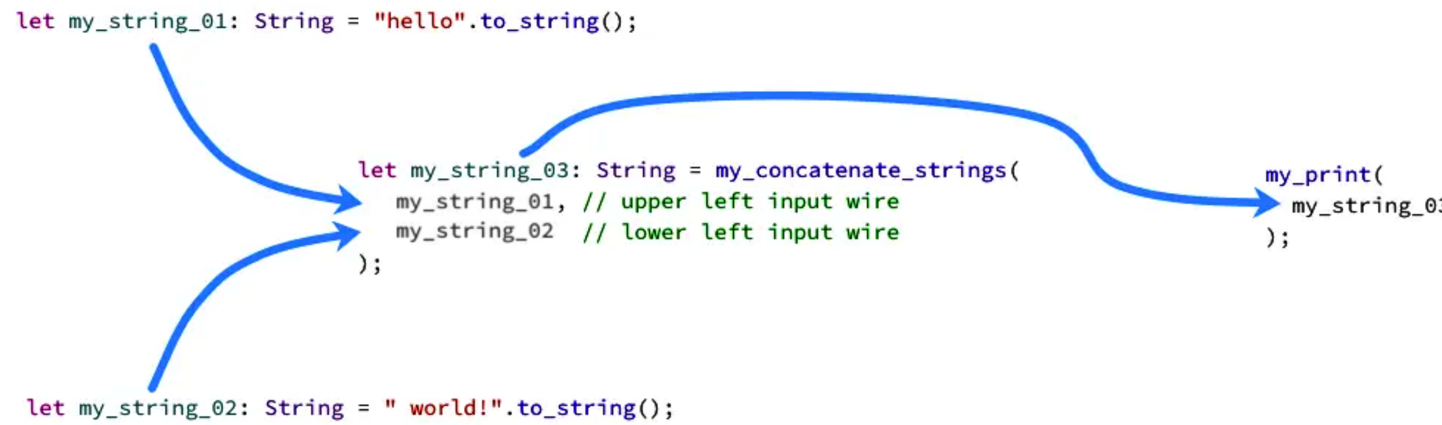
Demo

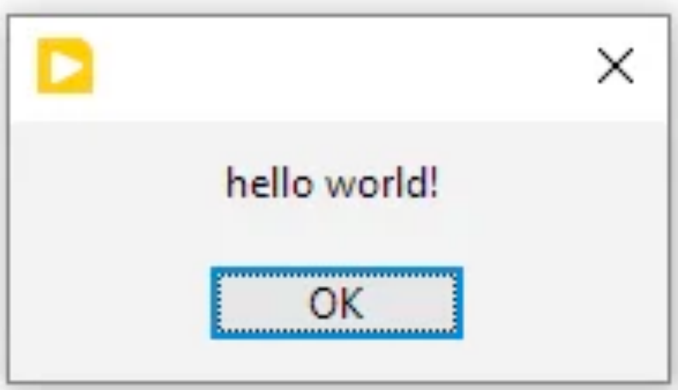

Windows PowerShell
PS C:\projects\hello world> .\hello.exe
hello world!
PS C:\projects\hello world>

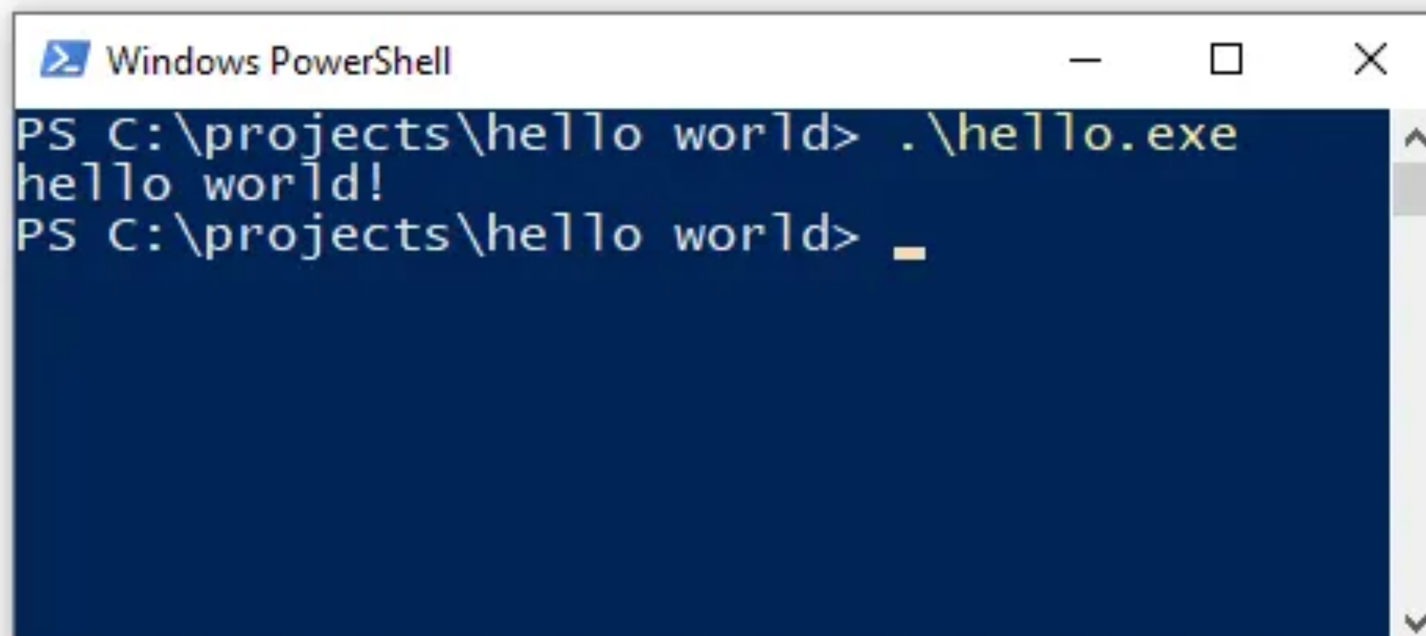
```
let my_string_01: String = "hello".to_string();

let my_string_03: String = my_concatenate_strings(
    my_string_01, // upper left input wire
    my_string_02  // lower left input wire
);

my_print(
  my_string_03
);

let my_string_02: String = " world!".to_string();
```




hello
world!

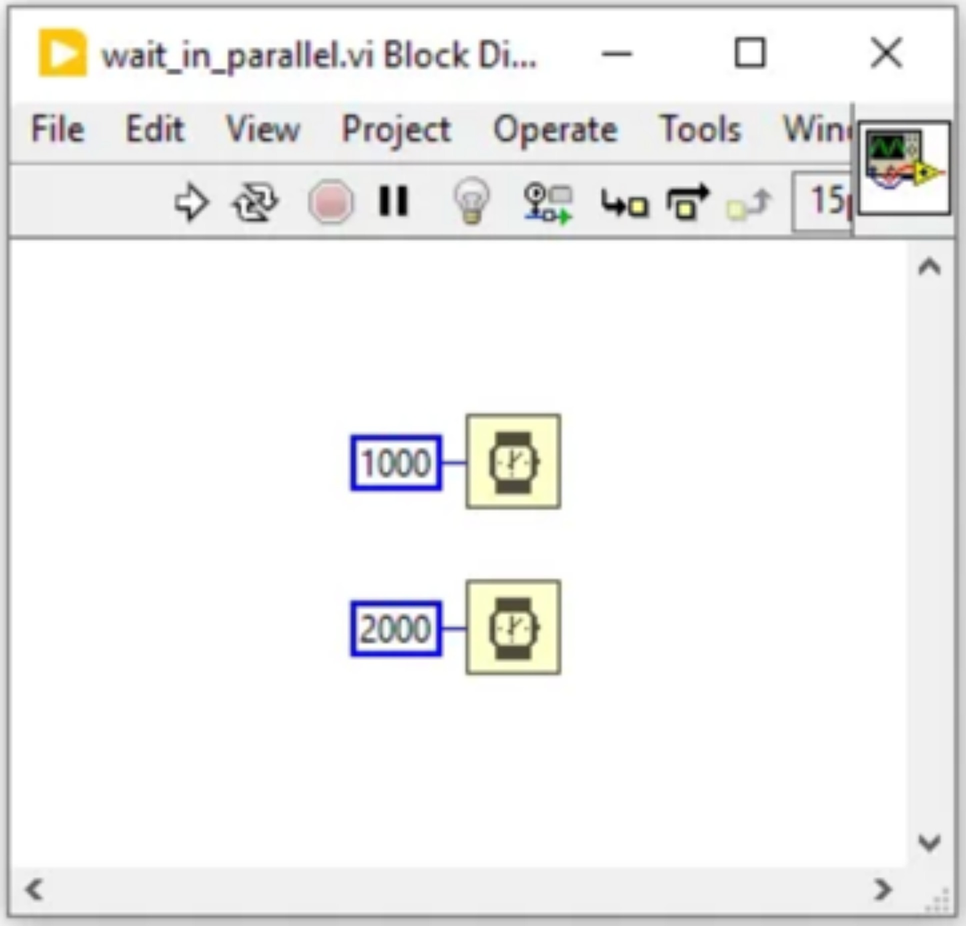

let my_string_01: String = "hello".to_string();
let my_string_03: String = my_concatenate_strings(
my_string_01, // upper left input wire
my_string_02 // lower left input wire
);
my_print(
my_string_03
);
let my_string_02: String = " world!".to_string();


hello world!
OK


Windows PowerShell
PS C:\projects\hello world> .\hello.exe
hello world!
PS C:\projects\hello world>

How about Parallelism?

We need a way to run code asynchronously,
according to the execution rules of sequential dataflow.




```
// run the first wait
wait_ms(1000, "Wait 1");

// run the second wait
wait_ms(2000);

// total wait time = 3000 milliseconds
```

Text-based Concurrency

We can leverage tools like threads and async features and frameworks of modern languages (like rust).

Threads

- `thread::spawn` runs a block of code asynchronously.
- `thread::spawn` returns a thread handle
- `join` waits until the thread completes and returns its data
- This is a lot like ACBRN in LabVIEW

```
// the main entry point of our program
fn main() {

    // let's start our stopwatch
    let start_time = Instant::now();

    // run the first wait in its own thread
    let thread_1 = thread::spawn(move || {
        wait_ms(1000, "Wait 1");
    });

    // run the second wait in its own thread
    let thread_2 = thread::spawn(move || {
        wait_ms(2000, "Wait 2");
    });

    // wait until both threads are finished
    thread_1.join().unwrap();
    thread_2.join().unwrap();

    // get the total ellapsed time of our program
    let total_duration = start_time.elapsed();

    // display the results
    println!(
        "Total execution time: {}.{:03} seconds",
        total_duration.as_secs(),
        total_duration.subsec_millis()
    );
}
```

Async & Await

Provides a framework to make “async” and act a little bit like `thread::spawn`.

Requires framework (runtime engine) to execute the async tasks to completion.

Jim Kring

Next Steps

Visit my blog at https://create.vi to stay up to date on progress.
Please feel free to message me if you are interested in contributing or learning more.
Find me on LinkedIn —> @jimkring