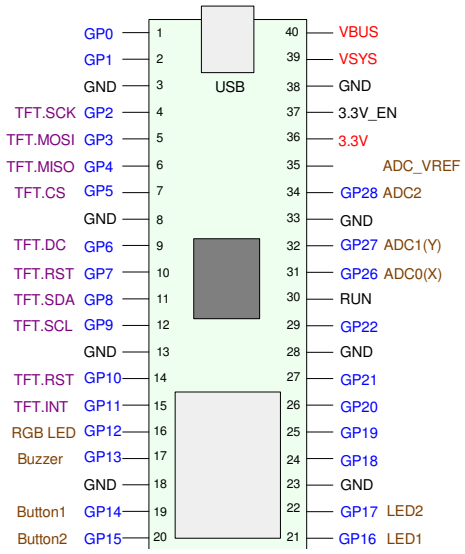
5. Binary Outputs

Machine & Time Library - Parallel Outputs
Morse Code



Introduction:

Similar to the PIC processor we covered in ECE 376, the Raspberry Pi Pico has 25 I/O pins (termed GPIO or General Purpose Input / Output pins). Unlike may other microcontrollers, these pins are not grouped in to clusters of eight: instead they are simply stand-alone pins.

Each of the I/O pins can be used for binary inputs or binary outputs. Many pins have can also be used for other things, such as UART or SPI communications (coming up later). The logic levels are

- 0V - 0.8V: logic level 0
- 2.0V - 3.3V: logic level 1

This lectures looks at

- Driving an LED
- Driving a Buzzer
  - Beep Five Times
  - Morse Code
- More Power
  - BJT (speaker, solenoid)
  - H-Bridge
- Parallel Outputs - LED Array
- Driving Multiple Outputs (PortA_Write)
- Display Routine (send to terminal pin values)
- Timing with Binary Outputs
  - Counter
  - Morse Code
- Frequency Out

Making a Light Blink

A simple program which makes the LED on pin 16 blink ten times is:

```
1  from machine import Pin
2  from time import sleep
3
4  LED = Pin(16, Pin.OUT)
5
6  for i in range(0,10):
7      LED.toggle()
8      sleep(0.1)
9  LED.value(0)
```

The way this program works is as follows:

- Lines 1 & 2: These import routines used later on in the program
- Line 4: This sets up GPIO pin 16 to be an output pin
- Line 7: This uses the routine *toggle()* from routine *machine* to toggle the LED
- Line 8: This calls the routine *sleep()* to pause 0.1 second
- Line 9: The LED is then turned off at the end of the program

You can also make the on-board beeper chirp five times with a simple change:

```
1  from machine import Pin
2  from time import sleep
3
4  Beeper = Pin(13, Pin.OUT)
5
6  for i in range(0,10):
7      Beeper.toggle()
8      sleep(0.1)
9  Beeper.value(0)
```

More details on how this work follows...

Background - Modules

One nice feature of Python is you can build upon subroutines that you wrote or that other people wrote. Some of the more commonly used and useful subroutines have been standardized and placed into files termed *modules*.

This is similar to C programmed where you incorporate previously-written subroutines using *include* statements. For example, a C program usually starts with something like this:

```
// Start of a C program
#include <stdio.h>
#include <math.h>
```

Once you include these files, you have access to and can call the subroutines defined in the module.

You do the same thing in Python. In Python, the start of your program usually starts with

```
#Start of a Python program
import machine
import time
import math
```

Once you import the module, you have access to and can use the routines (definitions in Python-speak) defined in these routines.

One slight difference with Python is in the syntax for calling routines within a module. To access routines within *math* and *time*, for example, you would use the following code:

```
import math
import time

x = 2 * math.cos( 1.74 * math.pi )
time.sleep(0.1)
```

When you call a routine from the *math* module, you need to tell Python

- The module you are using, and
- The routine you want to use

In the above example, we're using the routines *cos* and *pi* from module *math.* This has it's good points: if two modules have routines with the same name, there's no conflict: *math.pi* is different than *MyFunction.pi*. It also has it's bad points: the code can get kind of clunky to write and read.

So, Python also allows you to just import certain functions, allowing you to access them just by using their names. Repeating the previous example, you could also write

```
from math import
sin, cos, pi
from time import sleep

x = 2 * cos( 1.74 * pi )
sleep(0.1)
```

You just have to make sure that the function names (sin, cos, pi, sleep) don't conflict with other functions or variable names in your program.

If you want to know what modules are available to use, in the shell window type:

```
>>> help('modules')
random    machine     math      time    ...
```

(a complete list modules is in the appendix).

If you want to see what's inside a given module, such as *machine* or *time*, type

```
>>> import machine
>>> dir(machine)
['PWM', 'Pin', time_pulse_us', ...]

>>> import time
>>> dir(time)
['sleep', 'sleep_ms', 'sleep_us', ...]
```

To get some help on a specific function within a module, use the help function:

```
>>> help(machine.PWM)
object <class 'PWM'> is of type type
  init -- <function>
  deinit -- <function>
  freq -- <function>
  duty_u16 -- <function>
  duty_ns -- <function>
```

A complete list of modules and functions in the appendix[1].

Binary Outputs (Software)

Starting out, let's turn on and off an LED. With the RPi-Pico, the GPIO pins are binary signals:

- Logic 0 = 0V, capable of sinking up to 30mA
- Logic 1 = 3.3V, capable of sourcing up to 30mA

Each GPIO pin can be set up for either output (this lecture) or input (next lecture). To do this, a low-level routine *Pin* is used, which is in the module *machine*.

To set up pin #16 to output, use the code:

```
red = Pin(16,Pin.OUT)
```

To write to this pin or check its value, type in:

```
red.toggle()      toggle the pin on/off
red.value(1)      set the pin to ON
red.value(0)      set the pin to OFF
red.value()       return 1 if on, 0 if off
red.low()         set the pin to OFF
red.high()        set the pin to ON
```

To control the timing of a light turning on and off, routines from the module *time* are used

```
time.sleep(1.234)              sleep for 1.234 seconds
time.sleep_ms(1.234)           sleep for 1.234 milli-seconds
time.sleep_us(1.234)           sleep for 1.234 microseconds
```

[1] note: The appendix is a place to put stuff which would kill the flow of your document. A useful tool in technical documents.

For example, the following program

- Sets up pin 16 for output,
- Turns on pin 16 for one second,
- Turns off pin 16 for one second, and then
- Toggles pin 16 ten times every 100ms

```
from machine import Pin
from time import sleep

LED = Pin(16, Pin.OUT)

print('Light On')
LED.value(1)
sleep(1)

LED.value(0)
sleep(1)

for i in range(0,10):
    LED.toggle()
    sleep(0.1)
```

Sidelight: Using Arduino Syntax

Python on an Arduino and Raspberry Pi uses slightly different syntax for output pins. For these, the syntax for writing to an I/O pin is

```
GPIO(pin, value)
```

where *pin* is the pin number and *value* is 1 or 0 for on/off.

If you add a subroutine, GPIO, you can mimic this functionality:

- LED defines the I/O pins being used
- Each pin is set up for binary output

```
from machine import Pin
from time import sleep_ms

LED = [16,17,18,19,20,21,22,26]
for i in range(0, len(LED)):
    LED[i] = Pin(LED[i],Pin.OUT)

def GPIO(X, Value):
    if((Value > 1) | (Value < 0)):
        LCD[X].toggle()
    else:
        LED[X].value(Value)
```

You can then write to the I/O pins using Arduino syntax:

```
GPIO(4,1)              turn on output #4 (pin 19)
GPIO(4,0)              turn off output #4 (pin 19)
GPIO(4,-1)             toggle output #4 (pin 19)
GPIO(4,2)              another way to toggle pin #4
```

Binary Outputs (Hardware)

A RPi-Pico can drive more than just the LEDs on your development board. If you want to drive external devices, some simple electronic circuits work

Loads: < 3.3V and < 12mA:

If you're driving something that requires less than 30mA and less than 3.3V, you can drive the device directly from the GPIO pin. For example, suppose you want to drive a red LED with the following specifications at 10mA.

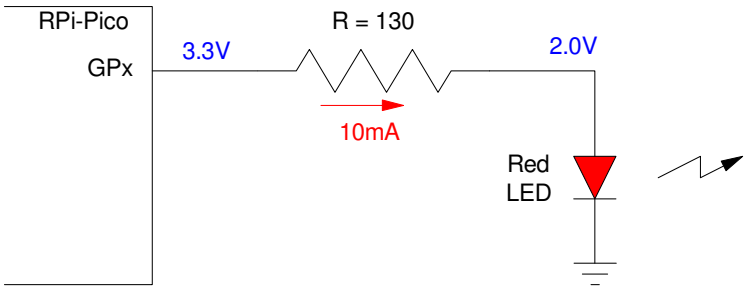
Digikey	color	wavelength	Vf @ 20mA	mcd @ 20mA	price
732-5013-ND	red	628nm	2.0V	2600mcd	$0.18

Vf tells you the voltage drop across the LED when turned on. Since the GPIO pin outputs 3.3V, to set the current to 10mA, you need a 130 Ohm resistor to limit the current:

$$R = \left(\frac{3.3V - 2.0V}{10mA}\right) = 130\Omega$$

The brightness of the LED will then be proportional to the current:

$$\left(\frac{10mA}{20mA}\right) 2600mcd = 1300mcd$$



If driving a load that needs less than 3.3V and less than 35mA,
you can connect it directly to the RPi-Pico with just a resistor (to limit the current)

Loads: >3.3V or >12mA:

If your load needs more than 3.3V or more than 30mA, the GPIO pin can't drive that device directly. If you add a BJT transistor or a MOSFET as a buffer, however, it can.

For example, design a circuit which allow the Pi-Pico to drive a 3W white LED:

ebay	color	Vf	Output	price
Lighthouse LEDs	warm white	3.6V @ 750mA	200lm @ 750mA	$2.06

In this case, a Pi-Pico can't drive the LED directly: it needs too much voltage and too much current. A Pi-Pico *can* drive it using a BJT transistor, however.

If you use a 6411 NPN transistor,

Digikey	Vce (sat)	hfe (min)	Ic (max)	hfe
2SC6144SG	360mV	200	10A	$0.85

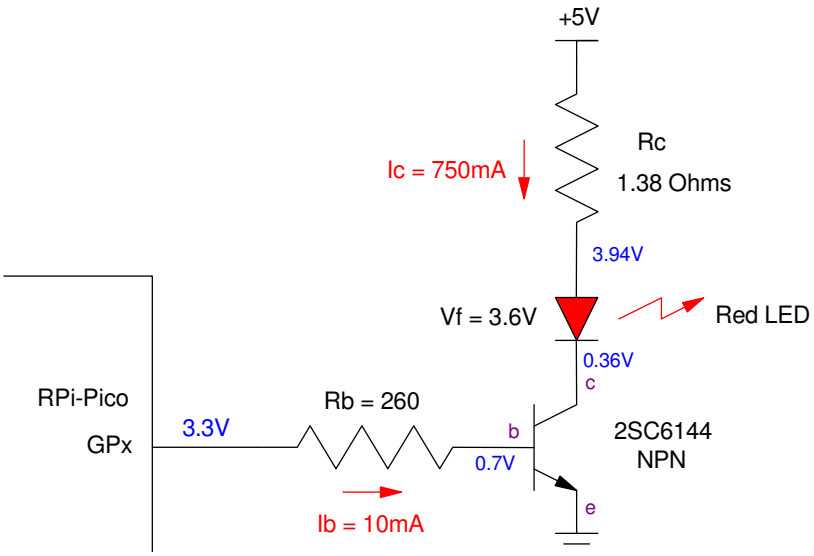
you can set the current to 750mA with the following circuit. Assuming a 5V source, the calculations are:

$$R_c = \left(\frac{5V - 3.6V - 0.36V}{750mA}\right) = 1.38\Omega$$

$$I_b > \frac{I_c}{h_{fe}} = \frac{750mA}{200} = 3.75mA$$

Let Ib = 10mA

$$R_b = \left(\frac{3.3V - 0.7V}{10mA}\right) = 260\Omega$$



If your load needs more than 3.3V or more than 35mA, you can use a BJT transistor as a switch

Note that using a BJT transistor as a switch works for just about any load that you want to turn on and off. The transistor doesn't really care about what the load is - as long as it needs less than 3A, it can turn the load on (GPx = 3.3V) or off (GPx = 0V).

$$\max(I_c) = h_{fe} \cdot I_b$$

$$= 300 \cdot 10mA = 3A$$

This makes a BJT switch very versatile and very common. With it, you can turn on and off

- LED lights
- DC motors
- Heaters
- Speakers,
- etc

providing they need less than 3A when on.

Note on Inductive Loads: As a slight caveat, if your load is inductive in nature:

- Solenoids
- DC motors

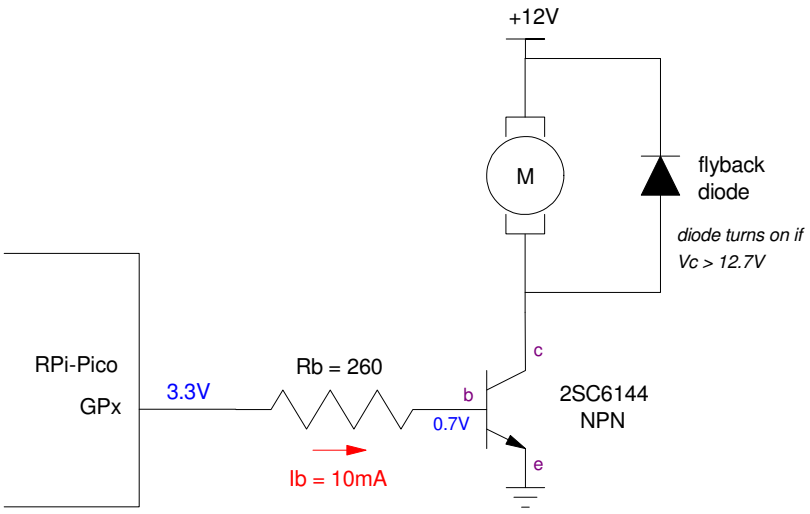
you need to include a flyback diode. This diode limits the voltage at Vc to +12.7V in the example below. This is important since

- For inductors, $v = L\frac{di}{dt}$
- When the transistor turns off, the current suddenly goes to zero
- This sudden drop in current can cause the voltage to shoot to infinity, burning out your transistor.

What's happening is

- Energy is stored in the inductor as $E = \frac{1}{2}Li^2$
- When the transistor turns off, the stored energy *must* go somewhere.

To bleed off the stored energy, the inductor will raise its voltage until it finds a path to ground. With the flyback diode, this voltage is limited to 12.7V



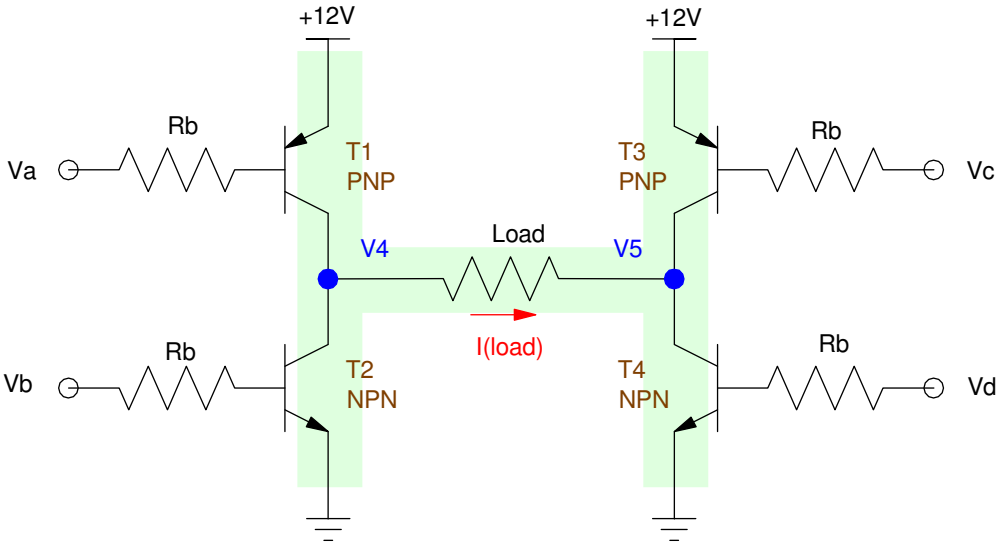
If you are turning on and off an inductive load (DC motor, solenoid),
add a flyback diode to limit the voltage at Vc

Forward & Reverse: H-Bridge

Finally, if you want to apply a positive and negative voltage to a load while using just a single power supply, a H-bridge can be used.

An H-bridge gets its name from the shape - it looks like the letter 'H'. The way it works is

- If transistors T1 and T4 are on,
  - V4 is pulled high (11.8V),
  - V5 is pulled low (0.2V), and
  - Current flows left to right (forward)
- If transistor T3 and T2 are on,
  - V5 is pulled high (11.8V),
  - V4 is pulled low (0.2V), and
  - Current flows right to left (reverse)
- If all transistors are off
  - Current is zero



H-Bridge. This allows you to apply positive and negative voltages to a load with a single power supply

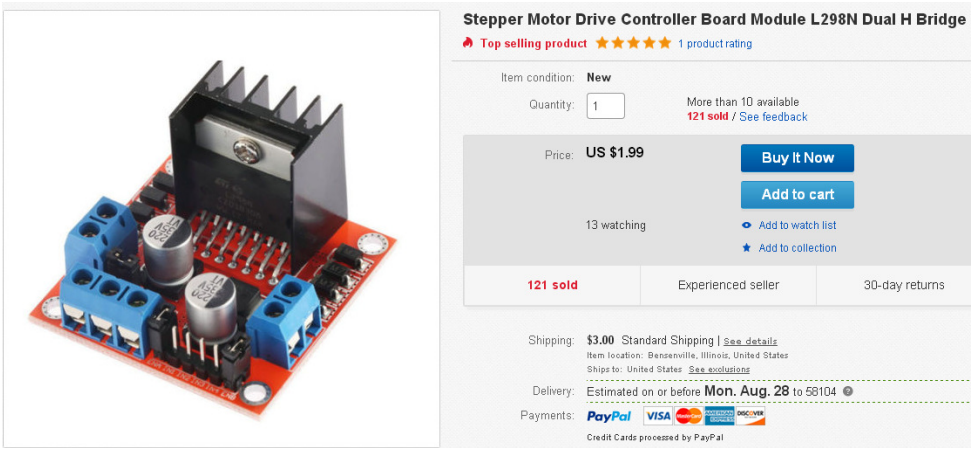
The net results, is with an H-bridge and a since +12V power supply, you can apply

- +11.6V to the load (T1 and T4 on),
- -11.6V to the load (T2 and T3 on), and
- 0.0V to the load (all transistors off)

This allows you to

- Drive a speaker forward (+) and back (-)
- Drive a motor forward (+) and in reverse (-)
- etc

In terms of how to do this with a RPi-Pico, probably the easiest way to use an H-bridge it to get one off of ebay:



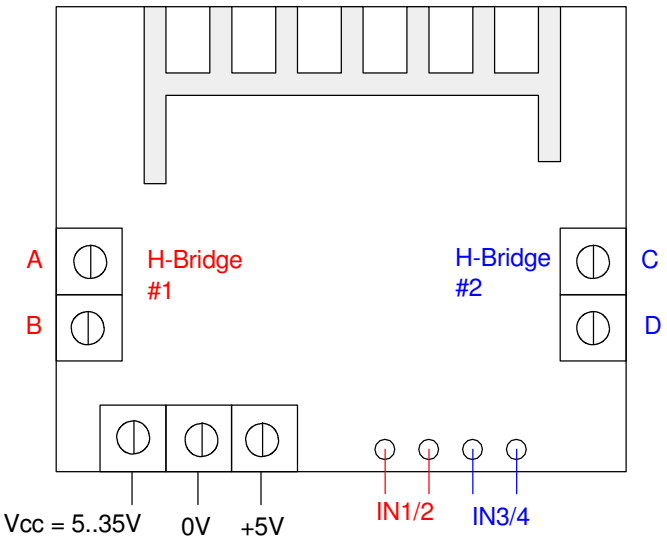
L298N Dual H-Bridge from ebay (search: Arduino H Bridge)

The L298N is actually has two H-bridges - for when you want to drive two loads (such as two speakers, two DC servo motors, or a single stepper motor. More on this later). The wiring for the H-bridge is as follows.

- Connect +5V and ground to the left screw terminals on the bottom. This provides the +5V needed by the electronics on this PCB.
- Connect your + power to the leftmost pin. This can be anything in the range of 5V..35V DC
- Connect your load to the screw terminal on the left (H-bridge #1) or right (H-bridge #2)

Finally, connect two GP output pins on your RPi-Pico board to

- IN1/2 to control H-bridge #1 or
- IN3/4 to control H-bridge #2



Wiring for a 298N Dual H-Bridge

In terms of software, you can control the voltage to the load using two GP output pins:

IN-1	IN-2	Vab
0V	0V	0 V
0V	3.3V	+ Vcc
3.3V	0V	- Vcc
3.3V	3.3V	0 V

IN-3	IN-4	Vcd
0V	0V	0 V
0V	3.3V	+ Vcc
3.3V	0V	- Vcc
3.3V	3.3V	0 V

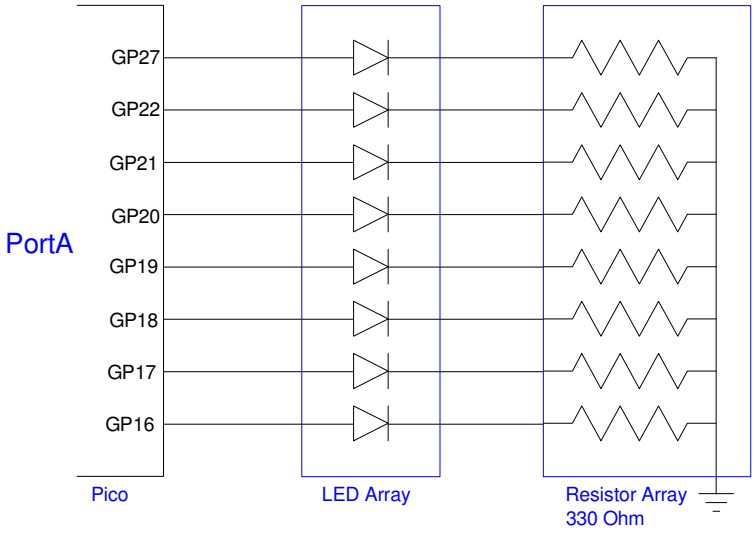
Sidelight: The 298N is really designed to operate off of 5V rather than 3.3V from the RPi-Pico. The transition from logic 0 to logic 1 usually happens right around 1.6V for TTL logic gates, however - meaning that the 298N will recognize 3.3V as logic 1. You don't need to add any buffer circuits between the Pico and the 298N.

Binary Outputs: Ports

In the previous examples, the Pi-Pico was driving a single pin high and low. What if you want to drive multiple pins high and low together? Or, to put it another way, can you group sets of pins together and call the grouping something like Port A?

Many microcontrollers, including PICs, group the I/O pins into clusters of 8, termed *ports*. With the RPi-Pico, no such groupings are made: each I/O pin is a stand-alone binary pin. So, the question arises

- *Can you group IO pins together to create a port?*
- *Can I set up the Pi-Pico so that when I write to PortA, I'm writing to eight LEDs at once?*



Problem: Can you group GPIO pins together to create a port?

The answer, of course is yes: you can do almost anything in software. The trick is to write a subroutine which allows you to treat a group of pins like it's a single port.

With the Pi-Pico, all GPIO pins are treated as stand-alone pins. This has it's good and bad features:

- good: you can assign pins wherever you like
- bad: you can't write to 8 bits at a time by writing to a port.

With software, you can mimic a port, however. In this example

- Pins 16..26 are assigned to PORTA (16 = LSB, 26 = MSB)
- By writing to PORTA with BinaryOut(X), you write to all eight bits

Note that this solution has it's good and bad features:

- good: you can assign pins wherever you like
- bad: you can't write to 8 bits at a time by writing to a port.
- bad: there is a slight time delay between when the first pin is set and cleared (pin 16) and the last pin (pin 26)

```
from machine import Pin
from time import sleep_ms

PORTA = [16,17,18,19,20,21,22,26]
for i in range(0, len(PORTA)):
    PORTA[i] = Pin(PORTA[i],Pin.OUT)

def display():
    X = ''
    n = len(PORTA)
    for i in range(0, n):
        X += str(PORTA[n-i-1].value)
    print(X)

def BinaryOut(X):
    for i in range(0, len(PORTA)):
        if(X & (1 << i)):
           PORTA[i].value(1)
        else:
           PORTA[i].value(0)

for i in range(0, 65535):
    BinaryOut(i)
    display()
    sleep_ms(50)

BinaryOut(0)
```

Fun With Binary Outputs: Blinking Light

Input a number from the keyboard

Blink the light N times

```
from machine import Pin
from time import sleep_ms

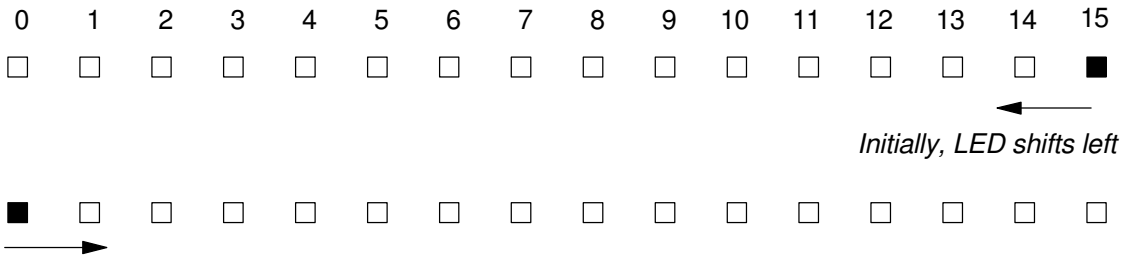
LED = Pin(16,Pin.OUT)

while(1):
    N = int(input('Number of Blinks: '))
    for i in range(0,2*N)
    LED.toggle()
    sleep_ms(100)
```

Fun with Binary Outputs: Night Rider

A second example, turn on a single LED and have it bounce back and forth.

- On power up, GP15 is on and the rest of the LEDs are off
- Every 100ms, the LED shifts left until it reaches GP0
- Once that happens, the LED starts to shift right until it reaches GP15
- At that point the LED starts to shift left again, etc.



When it reaches GP0, the LED shifts right

To make this work,

- Assign 16 pins to PortA with GP0 being the MSB, GP15 being the LSB
- Make all 16 bits output (routine Init())
- Set up a routine where when you pass a 16-bit number to PortA_Write, the value of each GP port is set or cleared based upon the value of bit i

Finally, in the main routine

- Starting with x = 0x0001
- Start shifting left every 100ms, writing x to PortA each loop
- When X = 0x8000, start shifting right

and so on.

```
# Blink the lights on PORTA up and down

from machine import Pin
from time import sleep_ms

PortA = [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0]

def Init():
    for i in range(0, len(PortA)):
        PortA[i] = Pin(PortA[i], Pin.OUT)

def PortA_Write(X):
    for i in range(0, len(PortA)):
        if(X & (1<<i)):
            PortA[i].value(1)
        else:
            PortA[i].value(0)

def PortA_Read():
    X = 0
    for i in range(0, len(PortA)):
        X += (1<<i) * PortA[i].value()
    return(X)

dir = 1
x = 1
Init()
while(1):
    PortA_Write(x)
    if(x & 0x8000):
        dir = -1
    if(x & 1):
        dir = 1
    if(dir == 1):
        x = x << 1
    else:
        x = x >> 1
    sleep_ms(100)
```

Fun with Binary Outputs: Morse Code

Finally, lets come up with a routine which outputs NDSU in Morse Code using bottom-up programming.

- Start with routines that output a dit (100ms on, 100ms off) and a dah (300ms on, 100ms off)
- Once that works, write routines to output N, D, S, and U in Morse code
- Once that works, string them all together to play NDSU every second.

```
# Morse Code

from machine import Pin
from time import sleep_ms

Beeper = Pin(13, Pin.OUT)

def Dit():
    Beeper.value(1)
    sleep_ms(100)
    Beeper.value(0)
    sleep_ms(100)

def Dah():
    Beeper.value(1)
    sleep_ms(300)
    Beeper.value(0)
    sleep_ms(100)

def Pause():
    sleep_ms(300)

def _N():
    Dah()
    Dit()
    Pause()

def _D():
    Dah()
    Dit()
    Dit()
    Pause()

def _S():
    Dit()
    Dit()
    Dit()
    Pause()

def _U():
    Dit()
    Dit()
    Dah()
    Pause()



while(1):
    _N()
    _D()
    _S()
    _U()
    sleep_ms(1000)
```

Summary:

With the Pi-Pico, you can turn on and off devices using the general purpose pins.

- If the load needs less than 3.3V and less than 12mA, the Pi-Pico can drive that device directly, using only a resistor to limit the current,
- If the load needs more voltage or current, the Pi-Pico can drive the device using a BJT transistor as a switch or an H-bridge as a buffer.
- With software, you can also cluster GPIO pins together to create ports. These allow you to drive multiple devices with a single Pi-Pico board.

Appendix:

PWM Outputs

The following program sets up pin 16 for

- PWM output
- 1000 Hz
- Duty Cycle varies from 0 to 100%

note:

- duty_u16(x) sets the duty cycle from 0 (x = 0x0000) to 100% (x = 0xFFFF)
- duty_ns(x) sets the on-time as x nanosecondns02

```
from machine import Pin
from time import sleep

red = Pin(16, Pin.OUT)
red16 = PWM(Pin(16))
red16.freq(1000)
x = 0
while(1):
    red16.duty_u16(x)
    x = (x+1) & 0xFFFF
    sleep_us(10)
```

Pulse With (ns)

- Set the frequency to 50Hz (period = 20ms)
- Set the pulse width from 0.5ms (500,000ns) to 3.0ms (3,000,000ns)

Typical for servo-motor controls

```
from machine import Pin
from time import sleep

red = Pin(16, Pin.OUT)
red16 = PWM(Pin(16))
red16.freq(50)
x = 500_000
dx = 1000
while(1):
    red16.duty_ns(x)
    x += dx
    if(x > 3_000_000):
        dx = -dx
    if(x < 500_000):
        dx = abs(dx)
    sleep_us(10)
```

Standard Modules Available

```
>>> help('modules')
__main__          array             framebuf          random
_asyncio          asyncio/__init__  gc                re
_boot             asyncio/core      hashlib           requests/__init__
_boot_fat         asyncio/event     heapq             rp2
_onewire          asyncio/funcs     io                select
_rp2              asyncio/lock      json              socket
_thread           asyncio/stream    lwip              ssl
_webrepl          binascii          machine           struct
aioble/__init__   bluetooth         math              sys
aioble/central    builtins          micropython       time
aioble/client     cmath             mip/__init__      uasyncio
aioble/core       collections       neopixel          uctypes
aioble/device     cryptolib         network           urequests
aioble/l2cap      deflate           ntptime           webrepl
aioble/peripheral dht               onewire           webrepl_setup
aioble/security   ds18x20           os                websocket
aioble/server     errno             platform
Plus any modules on the filesystem
```

Functions within machine

```
>>> import machine
>>> dir(machine)
['__class__', '__name__', 'ADC', 'I2C', 'I2S', 'PWM', 'PWRON_RESET', 'Pin',
'RTC', 'SPI', 'Signal', 'SoftI2C', 'SoftSPI', 'Timer', 'UART', 'WDT',
'WDT_RESET', '__dict__', 'bitstream', 'bootloader', 'deepsleep',
'dht_readinto', 'disable_irq', 'enable_irq', 'freq', 'idle', 'lightsleep',
'mem16', 'mem32', 'mem8', 'reset', 'reset_cause', 'soft_reset',
'time_pulse_us', 'unique_id']
```

Functions within time

```
>>> import time
>>> dir(time)
['__class__', '__name__', '__dict__', 'gmtime', 'localtime', 'mktime',
'sleep', 'sleep_ms', 'sleep_us', 'ticks_add', 'ticks_cpu', 'ticks_diff',
'ticks_ms', 'ticks_us', 'time', 'time_ns']
```

References

Pi-Pico and MicroPython

- https://github.com/geeekpi/pico_breakboard_kit
- https://micropython.org/download/RPI_PICO/
- https://learn.pimoroni.com/article/getting-started-with-pico
- https://www.w3schools.com/python/default.asp
- https://docs.micropython.org/en/latest/pyboard/tutorial/index.html
- https://docs.micropython.org/en/latest/library/index.html
- https://www.fredscave.com/02-about.html

Pi-Pico Breadboard Kit

- https://wiki.52pi.com/index.php?title=EP-0172

Other

- https://docs.sunfounder.com/projects/sensorkit-v2-pi/en/latest/
- https://electrocredible.com/raspberry-pi-pico-external-interrupts-button-micropython/
- https://peppe8o.com/adding-external-modules-to-micropython-with-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-raspberry-pi-pico/
- https://randomnerdtutorials.com/projects-esp32-esp8266-micropython/