Embedded System Software

Lecture 12 : Peripheral Access in MicroPython (RP2) & Software Pattern in MCU

Nattapong Wattanasiri

Peripheral Access in MicroPython?

Python module for accessing the peripherals in RP2

- machine
- rp2

The `machine` module:

```
import machine

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

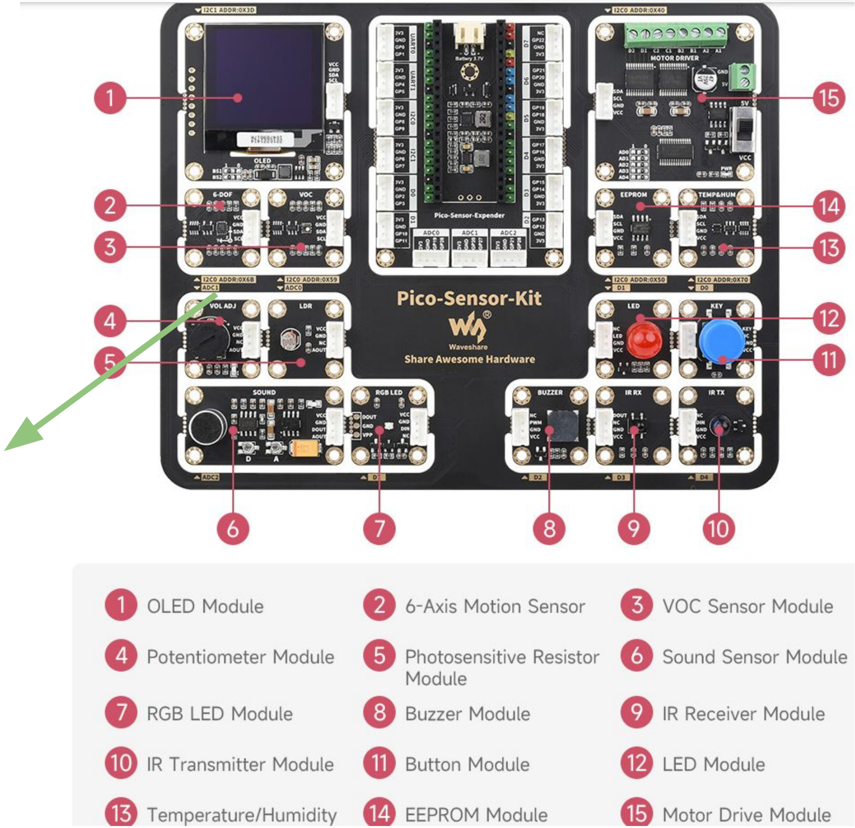
The `rp2` module:

```
import rp2
```

Pico-Sensor-Kit

Peripherals for this kit.

- GPIO
- PWM
- ADC
- I2C



GPIO

- For Pico-Sensor-Kit : **KEY**, **LED** modules
- Pin Connections

  KEY -> D0 : GPIO3

  LED -> D1 : GPIO10

GPIO

Use 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

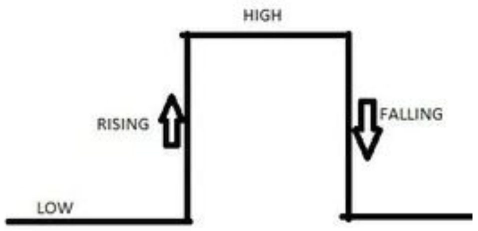
GPIO `mode` and `pull`:

- `pull` specifies if the pin has a (weak) pull resistor attached, and can be one of:
  - `None` - No pull up or down resistor.
  - `Pin.PULL_UP` - Pull up resistor enabled.
  - `Pin.PULL_DOWN` - Pull down resistor enabled.

- `mode` specifies the pin mode, which can be one of:
  - `Pin.IN` - Pin is configured for input. If viewed as an output the pin is in high-impedance state.
  - `Pin.OUT` - Pin is configured for (normal) output.
  - `Pin.OPEN_DRAIN` - Pin is configured for open-drain output. Open-drain output works in the following way: if the output value is set to 0 the pin is active at a low level; if the output value is 1 the pin is in a high-impedance state. Not all ports implement this mode, or some might only on certain pins.
  - `Pin.ALT` - Pin is configured to perform an alternative function, which is port specific. For a pin configured in such a way any other Pin methods (except `Pin.init()`) are not applicable (calling them will lead to undefined, or a hardware-specific, result). Not all ports implement this mode.
  - `Pin.ALT_OPEN_DRAIN` - The Same as `Pin.ALT`, but the pin is configured as open-drain. Not all ports implement this mode.
  - `Pin.ANALOG` - Pin is configured for analog input, see the `ADC` class.

GPIO

Interrupt setup for GPIO Input:




```python
from machine import Pin

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

def interruption_handler(pin):
    ...

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

while True:
    ...
```

GPIO

GPIO Example (Polling)

```
from machine import Pin
import time



# GPIO configuration for KEY module
# KEY -> GPIO3
gpio_num_KEY = 3
key = Pin(gpio_num_KEY, Pin.IN, Pin.PULL_UP) # Set GPIO as Input, Pull-up mode

# GPIO configuration for LED module
# LED -> GPIO10
# LED is in "current sink" configuration,
# Which means logic=1 -> turn off, logic=0 -> turn on
gpio_num_LED = 10
led = Pin(gpio_num_LED, Pin.OUT) # Set GPIO as Output

# GPIO configuration for Pico's onboard LED
led_pico = Pin('LED', Pin.OUT) # Set GPIO as Output



### Superloop
while True:
    # Task 1 : Polling button status
    button_status = key.value()

    # Task 2 : Blinking Pico's LED
    led_pico.toggle()
    time.sleep(1)

    # Task 3 : Toggle module's LED (GPIO10)
    if button_status == 0:
        while not key.value():
            time.sleep(0.01)
        led.toggle()
```

GPIO

GPIO Example (Interrupt)

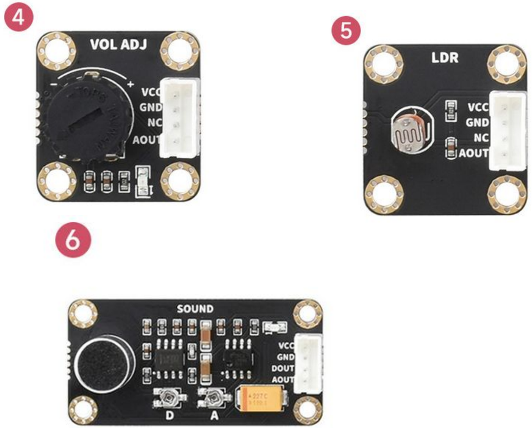
```python
from machine import Pin
import time


# global variable
task2_ready = False


def key_handler(pin):
    global task2_ready
    print("button at KEY module is pressed!")
    if task2_ready == False:
        task2_ready = True


# GPIO configuration for KEY module
# KEY -> GPIO3
# IO interrupt is used
gpio_num_KEY = 3
key = Pin(gpio_num_KEY, Pin.IN, Pin.PULL_UP) # Set GPIO as Input, Pull-up mode

# Set interrupt mode for GPIO
# Falling Edge type
key.irq(trigger=Pin.IRQ_FALLING,handler=key_handler)

# GPIO configuration for LED module
# LED -> GPIO10
# LED is in "current sink" configuration,
# Which means logic=1 -> turn off, logic=0 -> turn on
gpio_num_LED = 10
led = Pin(gpio_num_LED, Pin.OUT) # Set GPIO as Output

# GPIO configuration for Pico's onboard LED
led_pico = Pin('LED', Pin.OUT) # Set GPIO as Output


# Superloop
print("Wait for the key to be pressed")
while True:
    # Task 1 : Blinking Pico's LED
    led_pico.toggle()
    time.sleep(1)

    # Task 2 : Toggle module's LED (GPIO10)
    if task2_ready:
        print("task 2 running...")
        led.toggle()
        task2_ready = False
```

ADC

- RP2040 has five ADC channels in total, four of which are 12-bit SAR based ADCs: GP26, GP27, GP28 and GP29

- For Pico-Sensor-Kit : **VOL_ADJ**, **LDR**, **SOUND** modules
- Pin Connections

```
LDR     -> ADC0 : GPIO26
VOL_ADJ -> ADC1 : GPIO27
SOUND   -> ADC2 : GPIO28
```



ADC

Use the machine.ADC class:

```
from machine import ADC, Pin
adc = ADC(Pin(26))     # create ADC object on ADC pin
adc.read_u16()         # read value, 0-65535 across voltage range 0.0v - 3.3v
```

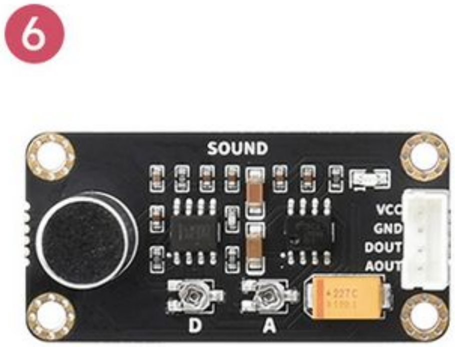
ADC

ADC Example

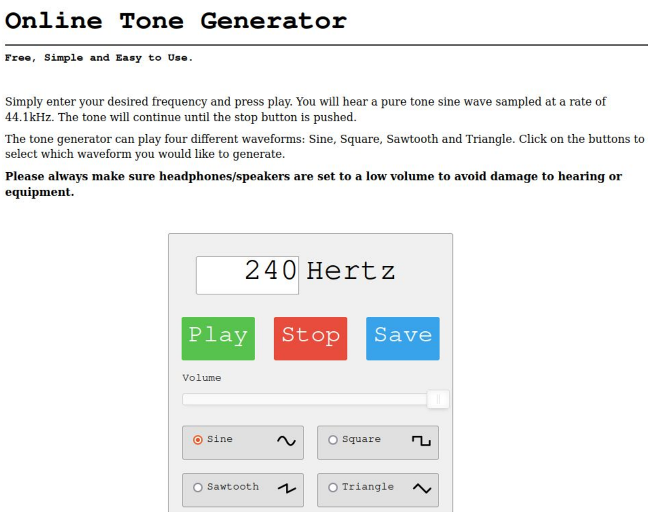
```python
from machine import Pin
from machine import ADC
import time



# Initialize ADC for light sensor (LDR)
gpio_ldr = 26
ldr = ADC(Pin(gpio_ldr))

# Initialize ADC for Knob (VOL ADJ)
gpio_knob = 27
knob = ADC(Pin(gpio_knob))



# Superloop
print("vol adj demo")
while True:
    # Task 1 : Read the analog voltage from volume's ADC
    voltage_ldr = ldr.read_u16()*3.3/65535

    # Task 2 : Read the analog voltage from volume's ADC
    voltage_knob = knob.read_u16()*3.3/65535

    # Task 3 : Display
    print("LDR voltage = {0:.2f}V, VOL ADJ voltage = {1:.2f}V ".format(voltage_ldr, voltage_knob))

    # Sleep
    time.sleep(1)
```

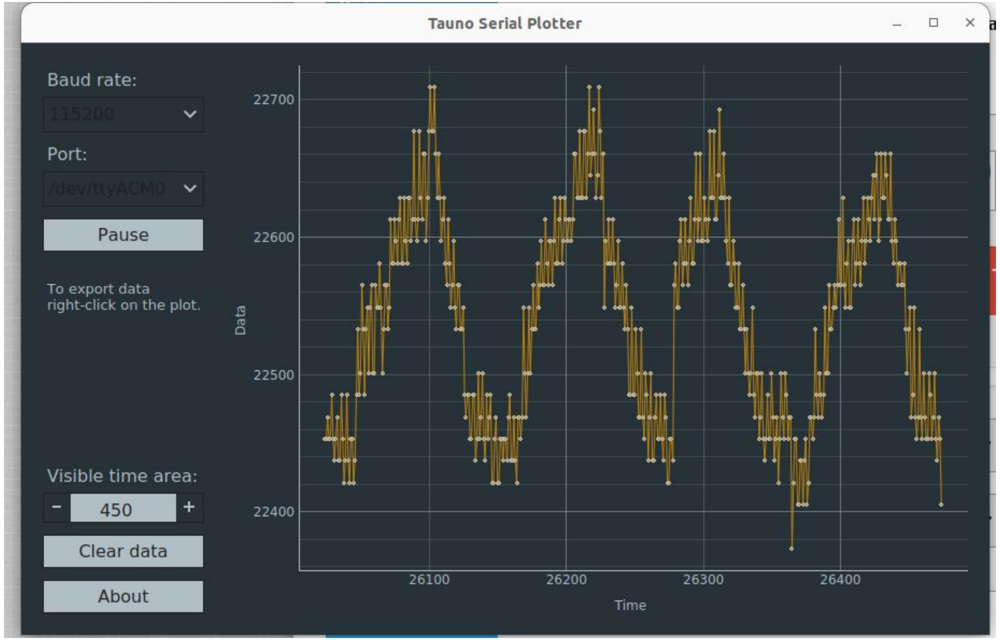
ADC

- About **SOUND** module, You can testrun and plot waveform of recorded sound.
- Save record in EEPROM, or sent it to Serial Plotter software.



ADC


Online Tone Generator
Free, Simple and Easy to Use.
Simply enter your desired frequency and press play. You will hear a pure tone sine wave sampled at a rate of 44.1kHz. The tone will continue until the stop button is pushed.
The tone generator can play four different waveforms: Sine, Square, Sawtooth and Triangle. Click on the buttons to select which waveform you would like to generate.
Please always make sure headphones/speakers are set to a low volume to avoid damage to hearing or equipment.
240 Hertz
Play
Stop
Save
Volume
Sine
Square
Sawtooth
Triangle


Tauno Serial Plotter
Baud rate:
Port:
Pause
To export data right-click on the plot.
Visible time area:
450
Clear data
About
22700
22600
22500
22400
Data
26100
26200
26300
26400
Time

PWM

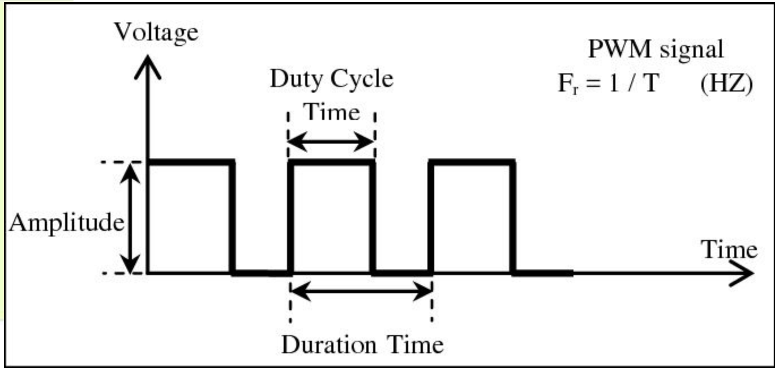
- RP2040 has 8 independent channels each of which have 2 outputs making it 16 PWM channels in total which can be clocked from 7Hz to 125Mhz.
- For Pico-Sensor-Kit : **BUZZER** modules
- Pin Connections

  BUZZER -> D2 : GPIO12

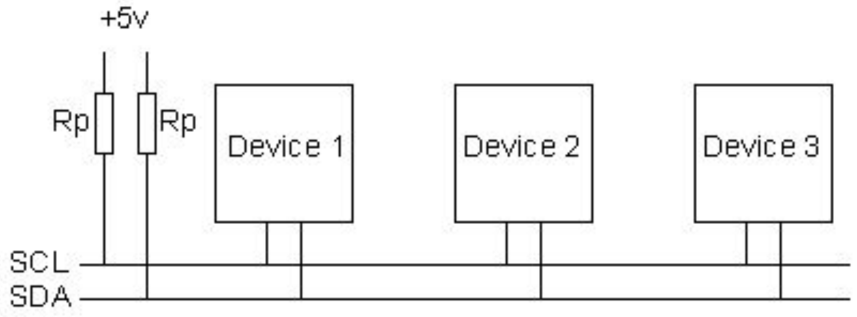
PWM

Use the machine.PWM class:

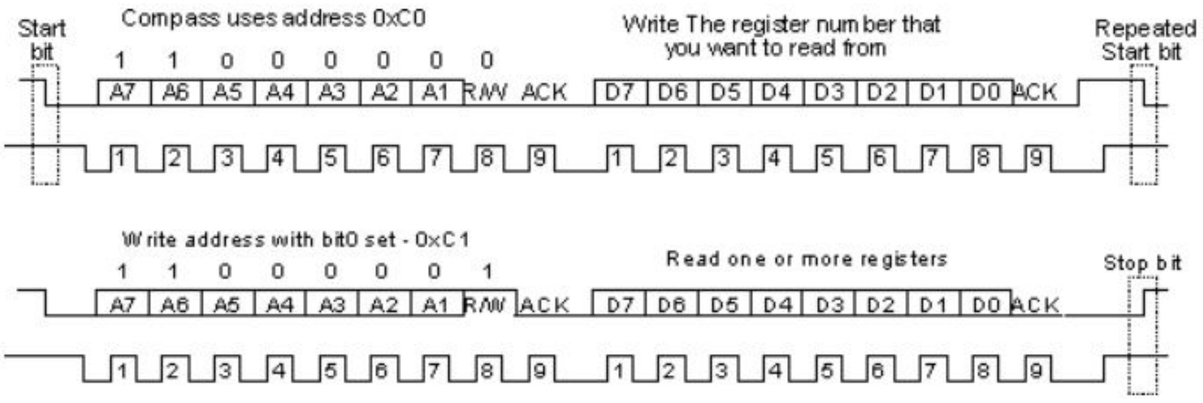
```
from machine import Pin, PWM

pwm0 = PWM(Pin(0))      # create PWM object from a pin
pwm0.freq()             # get current frequency
pwm0.freq(1000)         # set frequency
pwm0.duty_u16()         # get current duty cycle, range 0-65535
pwm0.duty_u16(200)      # set duty cycle, range 0-65535
pwm0.deinit()           # turn off PWM on the pin
```



I2C



The bit sequence will look like this:



I2C

- For Pico-Sensor-Kit : **6-DOF**, **VOC**, **TEMP&HUM**, **EEPROM**, **OLED** modules

I2C

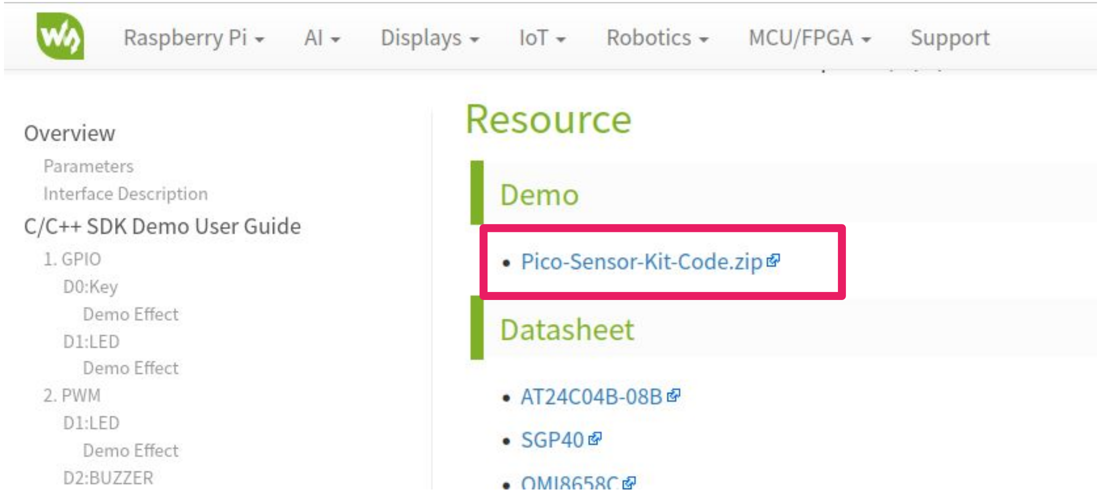
Hardware I2C bus

The driver is accessed via the machine.I2C class and has the same methods as software I2C above:

```
from machine import Pin, I2C

i2c = I2C(0)   # default assignment: scl=Pin(9), sda=Pin(8)
i2c = I2C(1, scl=Pin(3), sda=Pin(2), freq=400_000)
```

I2C

- Example of device library based on I2C (Show the demo code) https://www.waveshare.com/wiki/Pico-Sensor-Kit-B



Software Pattern for Embedded System

There are 4 Basic SW Pattern which can be seen alot in the real-world application

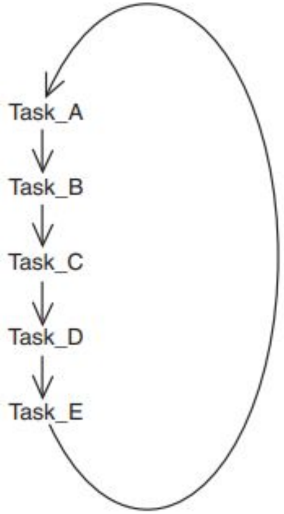
- Round-Robin
- Round-Robin + Interrupts
- Function-Queue Scheduling
- RTOS

For a basic micropython application

Round-Robin

- Extremely simple, Most used in simple system
- Aka. Superloop/Polling

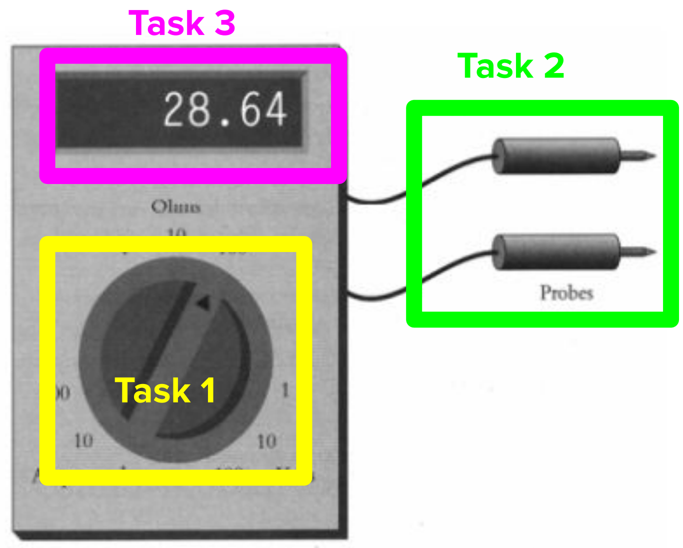
```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all devices have been serviced, then start over again
        }
    }
```



Round Robin Software Architecture

Round-Robin Example

```
switch_init()
measurement_init()
display_init()

void main(){

        while(True){
                // task1 : read switch position
                pos = readSwitchPosition()

                // task2 : measurement
                data = measurement(pos)

                // task3 : display
                display(data)
        }
}
```



Prioritize Task in Round-Robin

If one or more tasks have more strict deadlines than the others (they have higher priority), they may **simply be checked more often**:

```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B

        if (device_A requires service)
            service device_A
        if (device_C requires service)
            service device_C
        if (device_A requires service)
            service device_A
        ... and so on, making sure high-priority device_A is always serviced on time
        }
    }
```

Task A/ Device A has a high priority.
Check more often

Timer scheduling in Round-Robin

- Set period to execute tasks
- Minimize CPU usage for Low power mode

```
void main(){

        while(True){
                // task1 : read switch position
                if(task1_is_ready(sys_tick, period_task1){
                        pos = readSwitchPosition()
                }

                // task2 : measurement
                if(task2_is_ready(sys_tick, period_task2){
                        data = measurement(pos)
                }

                // task3 : display
                if(task2_is_ready(sys_tick, period_task3){
                        display(data)
                }
        }
}

void ISR_Timer()
{
        // Tick for some period. Example, for every 1 ms
        sys_tick++
}
```

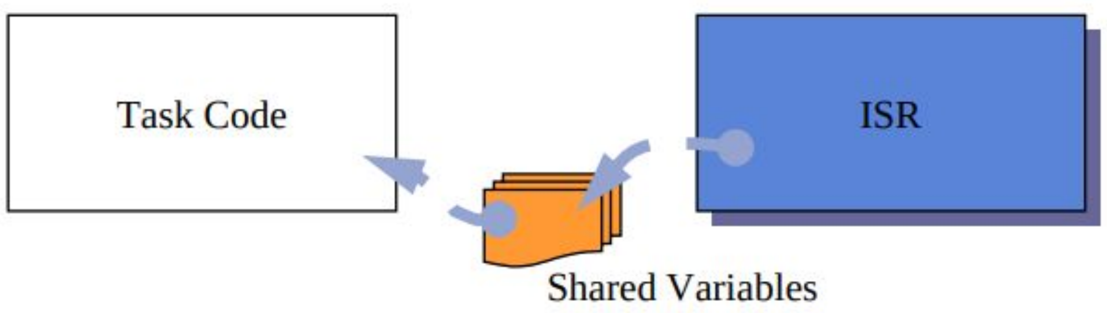
Round-Robin Execution Time

- Worst wait for highest-priority task code function = **Sum** length of all tasks

```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all devices have been serviced, then start over again
        }
    }
```

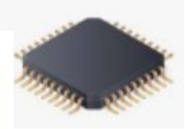
Round-Robin + Interrupt

- One step up on the performance scale is round robin with interrupts.
- **Urgent tasks** get handled in an interrupt service routine. Better response to urgent tasks.
- More event-driven software



Round-Robin + Interrupt

```
BOOL flag_A = FALSE; /* Flag for device_A follow-up processing */

/* Interrupt Service Routine for high priority device_A */

ISR_A(void) {
    ... handle urgent requirements for device_A in the ISR,
    then set flag for follow-up processing in the main loop ...
        flag_A = TRUE;
    }

void main(void) {
    while(TRUE) {
        if (flag_A)
            flag_A = FALSE
            ... do follow-up processing with data from device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all high and low priority devices have been serviced
        }
    }
```



Emergency stop button,
Urgent sensor etc.

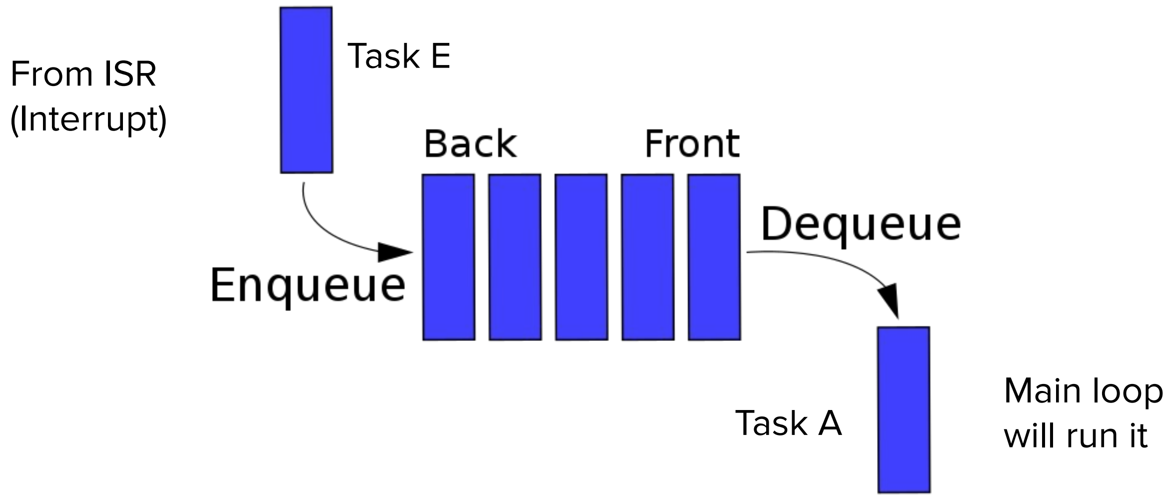
Round-Robin + Interrupt

- Racing Condition Problem :
- **Ex.** if the interrupted low priority function is in the middle of a calculation using data that are supplied or modified by the high priority interrupting function

```
void ISR_A()
{
        Disable_Other_ISR() // Disable Interrupt

        Flag_A = True
        Calculate_A(shared_data) // Calculation is needed immediately

        Enable_Other_ISR() // Enable Interrupt
}

void ISR_B()
{
        Flag_B = True
        shared_data = new_shared_data // shared data is updated
}
```

Function-Queue Scheduling

- Put a task's function after ISR is called into a **queue**
- The main loop simply checks the function queue
- Provides a method of assigning **priorities** to tasks



Function-Queue Scheduling

- Three parts

```
!! Queue of function pointers;

void interrupt vHandleDeviceA (void)
{
    !! Take care of I/O Device A
    !! Put function_A on queue of function pointers
}

void interrupt vHandleDeviceB (void)
{
    !! Take care of I/O Device B
    !! Put function_B on queue of function pointers
}
```

```
void main (void)
{
    while (TRUE)
    {
        while (!!Queue of function pointers is empty)
            ;

        !! Call first function on queue
    }
}
```

```
void function_A (void)
{
    !! Handle actions required by device A
}

void function_B (void)
{
    !! Handle actions required by device B
}
```

Function-Queue Scheduling Execution Time

- Worst wait for highest-priority task code function = length of longest task code function
- Better than RR, RR + Interrupt
- But, Response for lower-priority task may get worse due to **Starvation**

Starvation: lower-priority task code may never get executed!

Concurrency in Micropython?

- **_thread** module can be used for multi-threading. (pre-emptive multitasking)
- **uasyncio** module can be used. (co-operative multitasking)
- We will discuss on these in the next lecture.