First time ChatGPT helped me with programming

R

Question
what is the best method to do somethin every x seconds on the Raspi Pico, in Micropython (no interrupts)?

Answer

In MicroPython on the Raspberry Pi Pico, if you want to perform a task every x seconds without using interrupts, you can achieve this by utilizing the `utime` module for timekeeping and the `while` loop for iteration. Here's a simple example of how you can do it:

```
import utime

# Define the interval in seconds
interval = 5

# Get the current time
last_time = utime.time()

while True:
    # Get the current time
    current_time = utime.time()

    # Check if the interval has elapsed
    if current_time - last_time >= interval:
        # Do something
        print("Performing task...")

        # Update the last time
        last_time = current_time

    # You may add other tasks or operations here

    # Add a small delay to reduce CPU usage
    utime.sleep_ms(100)
```

In this code:

- `utime.time()` is used to get the current time in seconds.
- We keep track of the last time the task was performed using the `last_time` variable.
- Inside the `while` loop, we continuously check if the elapsed time since the last task execution is greater than or equal to the specified interval.
- If the interval has elapsed, we perform the task and update the `last_time` variable.
- You can add other tasks or operations inside the loop as needed.
- Adding a small delay using `utime.sleep_ms(100)` reduces CPU usage and prevents the loop from executing too rapidly. Adjust the delay as needed based on the requirements of your application.

R

Question
Isn't it better to use the time.ticks_ms function as this operates with integers?

Yes, you're correct. Using `time.ticks_ms()` from the `machine` module in MicroPython provides a more efficient way to work with time, especially when dealing with integer operations. Here's how you can modify the previous example to use `time.ticks_ms()`:

```
from machine import time

# Define the interval in milliseconds
interval = 5000

# Get the current time in milliseconds
last_time = time.ticks_ms()

while True:
    # Get the current time in milliseconds
    current_time = time.ticks_ms()

    # Check if the interval has elapsed
    if time.ticks_diff(current_time, last_time) >= interval:
        # Do something
        print("Performing task...")

        # Update the last time
        last_time = current_time

    # You may add other tasks or operations here

    # Add a small delay to reduce CPU usage
    time.sleep_ms(100)
```

In this version:

- We use `time.ticks_ms()` to get the current time in milliseconds.
- The interval is defined in milliseconds (`5000` milliseconds for 5 seconds).
- `time.ticks_diff()` is used to calculate the difference between the current time and the last time the task was performed. This function handles integer overflow correctly, making it more efficient for these calculations.
- The rest of the code remains the same, including the loop structure and the addition of a small delay to reduce CPU usage.

R

Question

Fine! I don't understand the part with " # Add a small delay to reduce CPU usage time.sleep_ms(100)"

Answer

Adding a small delay using `time.sleep_ms(100)` in the loop is a common practice to reduce CPU usage and prevent the loop from executing too rapidly. This is especially important in embedded systems like the Raspberry Pi Pico, where you want to conserve power and prevent unnecessary CPU cycles.

In the provided example, after performing the task and before checking the time again, there's a short delay of 100 milliseconds. This means that the loop will sleep for 100 milliseconds before starting the next iteration. This effectively slows down the loop, reducing the number of times it checks the time and performs the task per second.

Here's why this is beneficial:

1. **Reduced CPU Usage**: Without a delay, the loop would continuously execute as fast as possible, consuming CPU resources unnecessarily, especially in cases where the interval is relatively long (e.g., several seconds).
2. **Prevention of Busy Waiting**: Without a delay, the loop would continuously check the time and perform the task, which is known as busy waiting. This consumes CPU cycles even when there's nothing to do, which is inefficient.
3. **Conservation of Power**: By introducing a delay, the CPU can enter a low-power state during the sleep period, conserving power and extending battery life in battery-operated applications.
4. **Stability and Predictability**: Adding a delay ensures that the loop operates at a predictable pace, making it easier to manage and debug.

The value of 100 milliseconds (`time.sleep_ms(100)`) is chosen somewhat arbitrarily and can be adjusted based on your specific requirements. You might need to fine-tune this value depending on how responsive you need your system to be and how much power you can afford to consume.