Programming using States

Raspberry Pico Club – April 2025

Agenda

1. The theory of ‘Programming using States’
    - Python class example (state.py)
2. Three implementations
    - if...elif...else (main.v1.py)
    - Switch
    - Functions and routing table
3. Leveraging the 2 Pico Cores
    - Arcade game (main.game.py)
4. Let’s bring it together!
    - Sous Vide application

1. The theory of ‘Programming using States’

Software design pattern is a general, reusable solution to a commonly occurring problem.

- https://en.wikipedia.org/wiki/Software_design_pattern

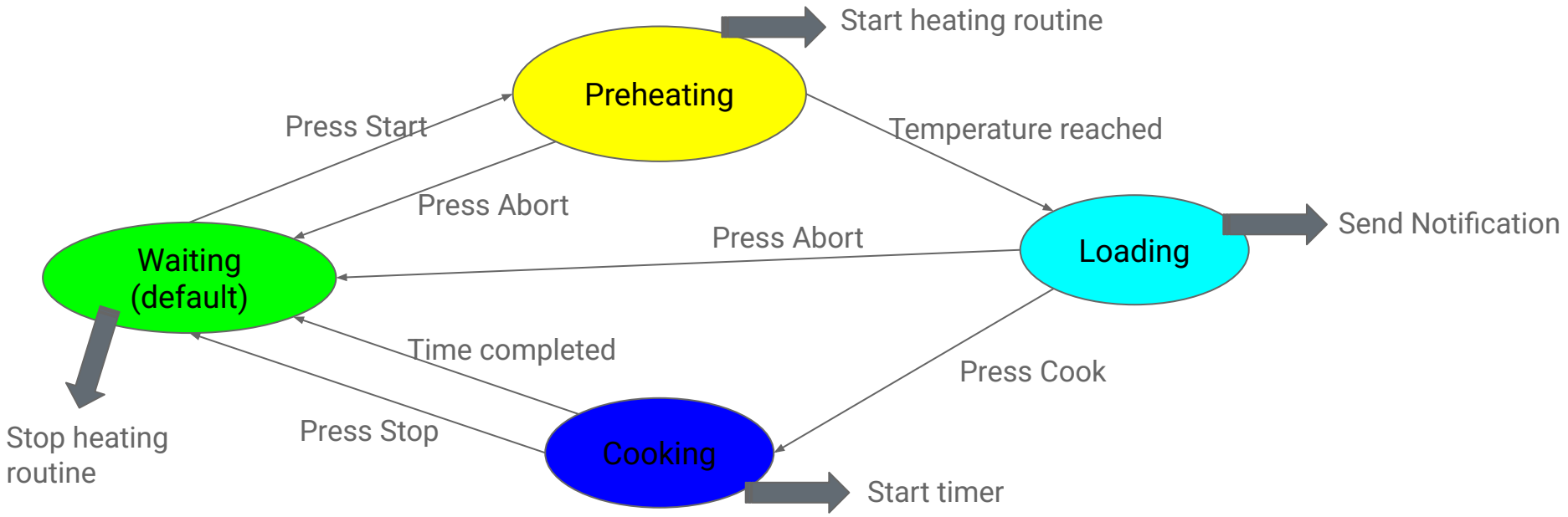
Famous ones:

- Event driven programming: modern GUI (desktop app)
- Functional programming: everything is a function
- Scheduler/time driven programming: the clock is the maestro
- State programming: based on graph and triggers (works well with Object-Oriented Languages)

At any given time, the program is in a state, waiting for a trigger to happen. This trigger might:

- Cause an action within the current state (or not)
- Might provoke a change of state (or not)

1.1 Graph to combine States, Triggers and Action



1.2 Look at code “state.py”

A state is a simple object that:

- tracks the current state
- allows current state easy checking
- provides state change method
- keeps the previous state for ‘undo’

Extra:

- has a default state (optional)
- has a ‘first time’ indicator to trigger action upon entry in the state

2.1 if…elif…else implementation

```
mystate = State(‘default’)
while True:
    trigger = GetOrCheckTriggers   # check if a button is pressed or a timer is completed or temperature is reached
    if mystate == ‘default’:
        if mystate.firstTime:
            <upon entry action>
            mystate.firstTiem = False
        elif trigger == TRIGGER1:
            <action for trigger>
        elif trigger == TRIGGER2:
            mystate.changeTo(‘next’)
    elif mystate == ‘next’:
        if trigger == TRIGGER4:
            <action for trigger>
            mystate.changeToDefault()
    else:
        print(“Error: unknown state”, mystate)
        mystate.changeTo(mystate.lastState)  # undo
```

2.2 switch implementation

```
mystate = State('default')
while True:
        trigger = GetOrCheckTriggers
        switch(mystate):
          case 'default':
                if mystate.firstTime:
                        <upon entry action>
                        mystate.firstTiem = False
                elif trigger == TRIGGER1:
                        <action for trigger>
                elif trigger == TRIGGER2:
                        mystate.changeTo('next')
         case 'next':
                if trigger == TRIGGER4:
                        <action for trigger>
                        mystate.changeToDefault()
          else:
                print("Error: unknown state", mystate)
                mystate.changeTo(mystate.lastState)  # undo
```

2.3 Functions and routing table

```
mystate = State('default')
def default(trigger=None):
        if mystate.firstTime:
                <upon entry action>
                mystate.firstTiem = False
        elif trigger == TRIGGER1:
                <action for trigger>
        elif trigger == TRIGGER2:
                mystate.changeTo('next')

def next(trigger=None):
        if trigger==TRIGGER4:
                <action for trigger>
                mystate.changeToDefault()

ROUTING = {
        'default': default
        'next': next
}

while True:
        trigger = GetOrCheckTriggers
        try:
                ROUTING[mystate](trigger)
        except IndexError:
                print("Error: unknown state", mystate)
                mystate.changeTo(mystate.lastState)  # undo
```

3. Leveraging the 2 Pico Cores

- The RP2040 chip has two independent cores, enabling parallel processing of tasks
  - in Micropython, Core 0 is used by default
  - Core 1 is accessible using specific module `_thread`
  - Shared memory - highly recommended to have one Core in Read only for a given variable
- Stack management is buggy: must call the garbage collector at regular interval to prevent crash

```
import _thread, gc    #  call at regular interval gc.collect()

def core1_thread():
      """
      This code will run on Core 1
      """
      <here is the code>
```

start this code by calling:

```
      _thread.start_new_thread(core1_thread, ())
```

3.1 simple arcade game

This simple arcade game demonstrated how to leverage both cores:

- Core 0 manages user's interactions by scanning the button pressed
  - navigation through menus using state
  - during 'Play' state:
    - calculate racket movements
    - calculate balls generation and falls
    - verify if a ball has reached bottom or got caught by the racket
    - call regularly the garbage collector

- Core 1 manages the screen display
  - only active when in 'Play' mode
  - display the score, level, lifes
  - draw the racket and the falling balls (if any)
  - "read only" the global variables

4. Let’s bring it together: *Sous vide* application

- Core 0 main routine
  - Two state objects: cooking stage and menu navigation
    - User can set the water temperature and timer
    - Interaction like ‘Start’, ‘Stop’, ‘Cook’ → actions on menu triggers cooking stage changes
  - Notifications are sent (optional) to Whatsapp using Callmebot service

- Core 1 routine to control the water temperature and thus switch ON/OFF the electric resistance (heater)
  - reads the temperature sensors and average their readings
  - controls the water resistance and water mixing motor (uniform water temperature)
  - loop is subjected to a global boolean ‘`controllerIsRunning`’ activated by Core 0 main routine