Microprocessor – Microcontroller

201 W02 - **Embedded Software Architectures**

Nguyen Tran Huu Nguyen

D: Computer Engineering

E: nthnguyen@hcmut.edu.vn

Embedded Software Architectures

- Round robin
- Round robin with Interrupts
- Function Queue Scheduling
- Real-time operating system

Round robin

- No interrupts
- Main loop checks each of the I/O device in turn and services any that need service
- No priorities

Round robin

```
void main(void){
        while (TRUE){
                if (!! I/O Device A needs service) {
                        !! Take care of I/O Device A
                        !! Handle data to or from I/O Device A
                }
                if(!! I/O Device B needs service) {
                        !! Take care of I/O Device B
                        !! Handle data to or from I/O Device B
                }
                etc.
                etc.
                if(!! I/O Device Z needs service) {
                        !! Take care of I/O Device Z
                        !! Handle data to or from I/O Device Z
                }

        }
```

Round robin – Digital multimeter


```
void vDigitalMultiMeterMain(void){
        enum {OHMS_1, OHMS_10, …, VOLTS_100} eSwitchPosition;
        while (TRUE){
                eSwitchPosition  = !! Read the position of the switch;
                switch (eSwitchPosition) {
                        case OHMS_1:
                                !! Read hardware to mearsure ohms
                                !! Format result
                        break;
                        case OHMS_10:
                                !! Read hardware to mearsure ohms
                                !! Format result
                        break;
                        .
                        .
                        .
                        case VOLTS_100 :
                                !! Read hardware to mearsure ohms
                                !! Format result
                        break;
                }
                !! Write result to display

        }
}
```

Round robin

- If any one device needs response in less time that it takes the MCU to get around the main loop in the worst-case scenario, then the system won't work
- For example, if device Z can wait no longer than 7 milliseconds for service and if the pieces of code that service devices A and B take 5 milliseconds each, then the processor won't always get to device Z quickly enough

Round robin

- Even if none of the required response times are absolute deadlines, the system may not work well if there is any lengthy processing to do.
- For example, if one of the cases were to take, say, 3 seconds, then the system's response to the rotary switch may get as bad as 3 seconds. This may not quite meet the definition of "not working", but it would probably not be a system that anyone would be proud to ship

Round robin

- This architecture is fragile.
- Even if you manage to tune it up so that the MCU gets around the loop quickly enough so satisfy all the requirements, a single additional device or requirement may break everything.
- This architecture is probably suitable only for very simple devices.

Round robin with Interrupts

- Interrupt routines deal with the very urgent needs of the hardware and the set flags
- The main loop polls the flags and does any follow-up processing required by the interrupts.

Round robin with Interrupts

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
…
BOOL fDeviceZ = FALSE;
void INTERRUPT vHandleDeviceA (void){
        !! Take care of I/O Device A
        fDeviceA = TRUE
}
void INTERRUPT vHandleDeviceB (void){
        !! Take care of I/O Device B
        fDeviceB = TRUE
}
…
void INTERRUPT vHandleDeviceZ (void){
        !! Take care of I/O Device Z
        fDeviceZ = TRUE
}
```

```
void main (void) {
        while(TRUE){
                if (fDeviceA) {
                        fDeviceA = FALSE;
                        !! Handle data to or from device A
                }
                if (fDeviceB) {
                        fDeviceB = FALSE;
                        !! Handle data to or from device B
                }
                ….
                if (fDeviceZ) {
                        fDeviceZ = FALSE;
                        !! Handle data to or from device Z
                }

        }
}
```

Round robin with Interrupts

- This architecture gives you a bit more control over priorities.
- The interrupt routines can get good response, because the hardware interrupt signal causes the microcontroller to stop whatever it is doing in the main function and execute the interrupt routine instead.
- All of the processing that you put into the interrupt routines has a higher priority than the task code in the main routine.
- Since you can usually assign priorities to the various interrupts in your system, you can control the priorities among the interrupt routines as well

Function-Queue Scheduling

- The interrupt routines add function pointers to a queue of function pointers for the main function to call.
- The main routine just reads pointers from the queue and calls the functions.

Function-Queue Scheduling

```
!! Queue of function pointers;
void interrupt vHandleDeviceA (void){
        !! Take care of I/O Device A
        !! Put function_A on queue of function poitners
}
void interrupt vHandleDeviceB (void){
        !! Take care of I/O Device B
        !! Put function_B on queue of function poitners
}
void function_A (void){
       !! Handle actions required by device A
}
void function_B (void){
      !! Handle actions required by device B
}
```

```
void main (void){
     while(TRUE){
          while (!! Queue of function pointer is empty);
          !! Call first function on queue
     }
}
```

Function-Queue Scheduling

- No rule says main has to call the functions in the order that the interrupt routines occurred.
- It can call them based on any priority scheme that suits your purposes.

Real-time operating systems

```
void interrupt vHandleDeviceA (void) {
        !! Take care of I/O Device A
        !! Set signal X
}
void interrupt vHandleDeviceB (void) {
        !! Take care of I/O Device B
        !! Set signal B
}
void Task1 (void){
        while(TRUE){
                !! Wait for signal X
                !! Handle data to or from I/O device A
        }
}
void Task2 (void){
        while(TRUE){
                !! Wait for signal Y
                !! Handle data to or from I/O device B
        }
```

Real-time operating systems

- The interrupt routines take care of the most urgent operations.
- They signal that there is work for the task code to do.

Real-time operating systems

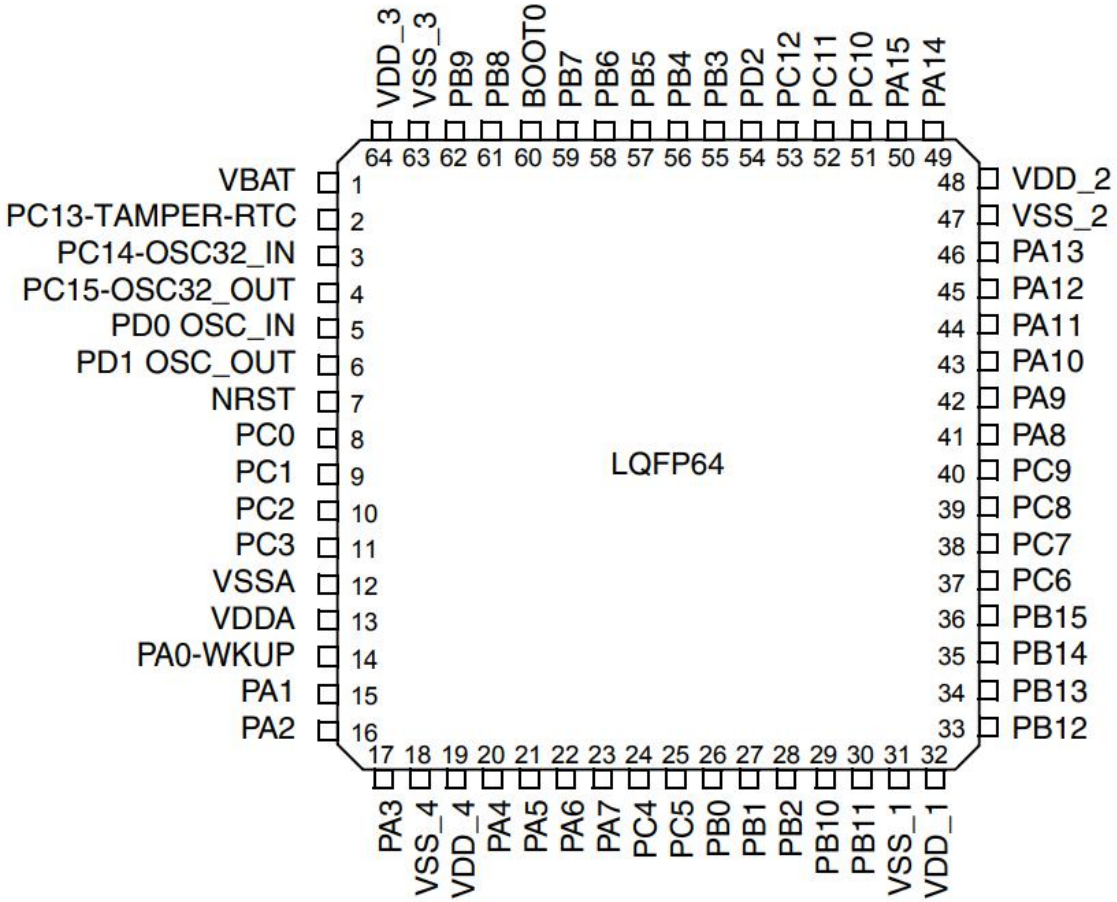
- The differences between RTOS and the previous architectures are that:
  - The necessary signaling between the interrupt routines and the task code is handled by the RTOS.
  - No loop in our code decides what needs to be done next. Code inside the RTOS decides which of the task code function should run.
  - The RTOS knows about the various task-code subroutines and will run whichever of them is more urgent at any given time.
  - The RTOS can suspend one task code subroutines in middle of its processing in order to run another.

Summary

- Response requirements most often drive the choice of architecture.
- Generally, you will be better off choosing a simpler architecture.
- Hybrid architecture can make sense for some systems.

Summary

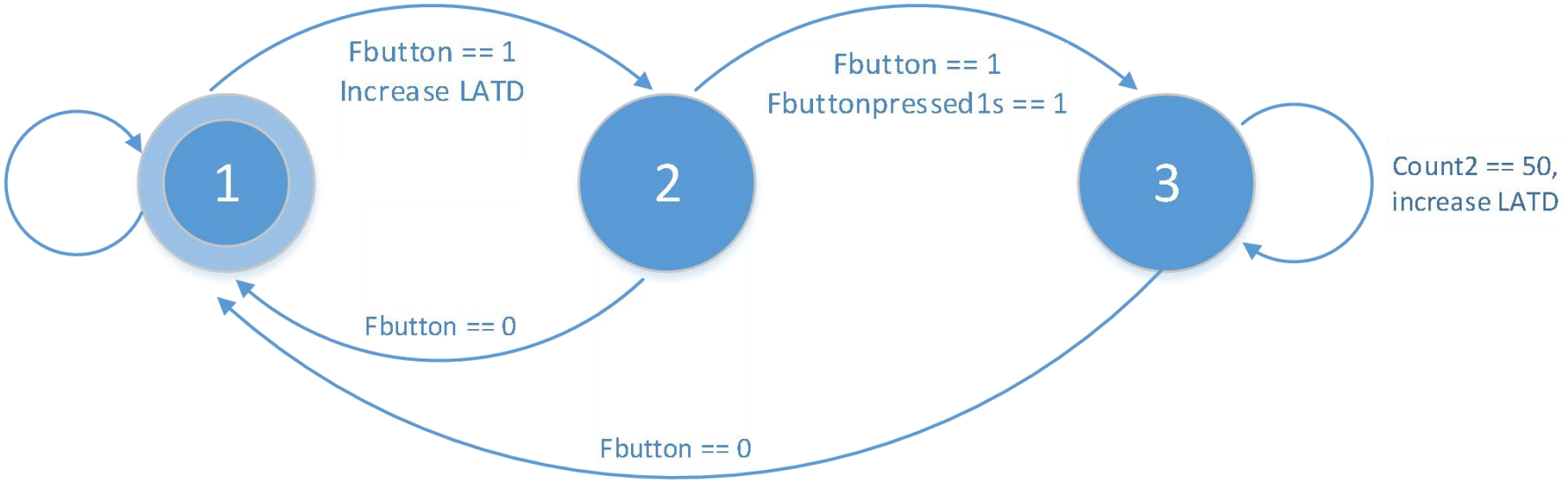
	Priorities Available	Worst Response Time for Task code	Stability of Response when the code changes	Simplicity
Round robin	None	Sum of all task code	Poor	Very simple
Round robin with interrupts	Interrupt routines in priority order, then all task code at the same priority	Total of execution time for all task code (plus execution time for interrupt routines)	Good for interrupt routines, poor for task code.	Must deal with data shared between interrupt routings and task code.
Function-Queue scheduling	Interrupt routines in priority order, then task code in priority order	Execution time for the longest function (plus execution timer for interrupt routines)	Relatively good	Must deal with shared data and must write function queue code
RTOS	Interrupt routines in priority order, then task code in priority order	Zero (plus execution time for interrupt routines)	Very good	Most complex (although much of the complexity is inside the OS itself.


VDD_3
VSS_3
PB9
PB8
BOOT0
PB7
PB6
PB5
PB4
PB3
PD2
PC12
PC11
PC10
PA15
PA14
64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49
VBAT 1
PC13-TAMPER-RTC 2
PC14-OSC32_IN 3
PC15-OSC32_OUT 4
PD0 OSC_IN 5
PD1 OSC_OUT 6
NRST 7
PC0 8
PC1 9
PC2 10
PC3 11
VSSA 12
VDDA 13
PA0-WKUP 14
PA1 15
PA2 16
LQFP64
48 VDD_2
47 VSS_2
46 PA13
45 PA12
44 PA11
43 PA10
42 PA9
41 PA8
40 PC9
39 PC8
38 PC7
37 PC6
36 PB15
35 PB14
34 PB13
33 PB12
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
PA3
VSS_4
VDD_4
PA4
PA5
PA6
PA7
PC4
PC5
PB0
PB1
PB2
PB10
PB11
VSS_1
VDD_1

Round robin with timer interrupts example

- **W**rite a program that
  - Has a timer which has an interrupt in every 10 milliseconds.
  - Reads values of button P**A5** every 10 milliseconds.
  - Increases the value of LEDs connected to **PB[0-7] (or LATD in PIC)** when the button P**A5** is pressed.
  - Increases the value of **PB[0-7] (or LATD in PIC)** automatically in every 0.5 second, if the button RA5 is pressed in more than 1 second.

Finite State Machine



Example code

```
void main (void) {
    enum {STATE1, STATE2, STATE3} eState;
    while (TRUE) {
        switch (eState){
            case STATE1:
                if (fbutton) {
                    increase PB[7-0];
                    eState = STATE2;
                }
            break;
            case STATE2:
                if (fbutton == 0) eState = STATE1;
                else if (fbuttonpress1s) eState = STATE3;
            break;
            case STATE3:
                if (count2 == 50) increase  PB[7-0]
                if (fbutton == 0) eState = STATE1;
            break;
            }
        }
}
```

```
void Readbutton(void) {
    firstReadPA5 = secondReadPA5;
    secondReadPA5 = PA5;
    if (secondReadPA5 == firstReadPA5) {
        if (firstReadPA5) {
            fbutton = 1;
            count1++
            if (count1 >= 100){
                fbuttonpressed1s = 1;
                count2 ++;
                if (count2 >= 50){
                    count2 = 0;
                }
            }
        } else {
            fbutton = 0;
            fbuttonpressed1s = 0
            count1 = 0;
            count2 = 0;
        }
    }
}
```