

- C03009 -

- Software Architectures -



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

```
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 measure ohms
                !! Format result
                break;
            case OHMS_10:
                !! Read hardware to measure ohms
                !! Format result
                break;
            .
            .
            .
            case VOLTS_100 :
                !! Read hardware to measure 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

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

```
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 (RTOS)

- 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.