Survey Of Embedded Software Architectures

Survey Of Embedded Software Architectures

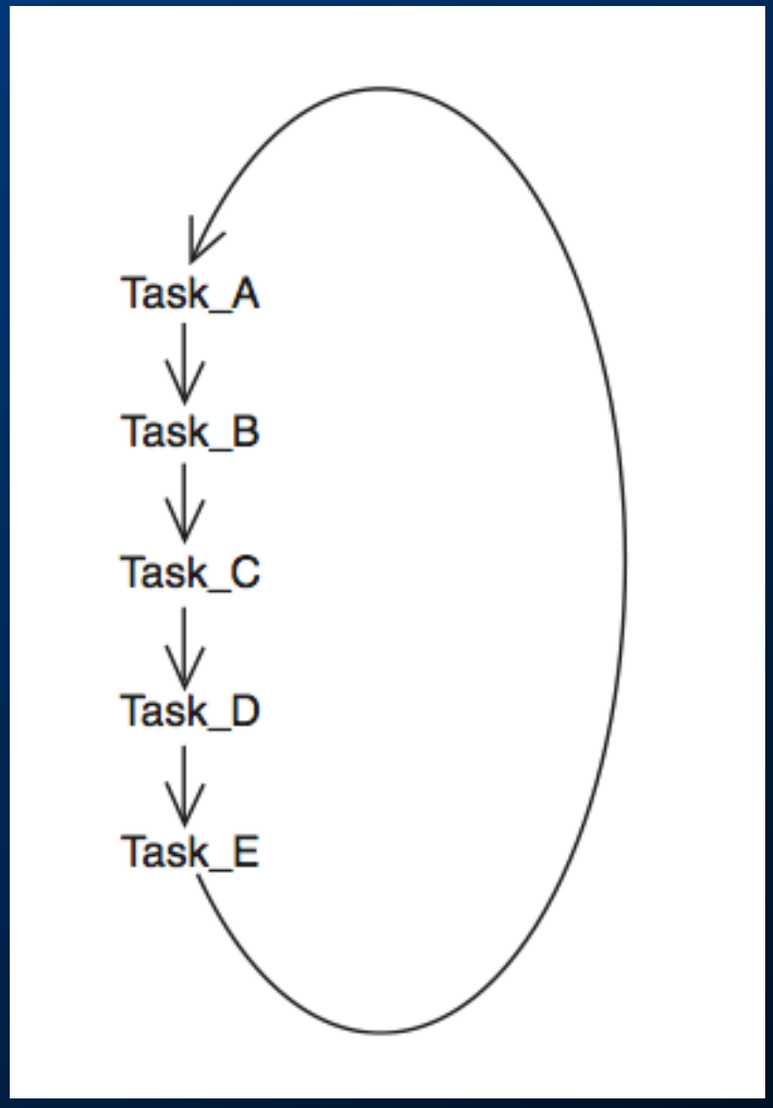
- Round Robin
- State Machine
- Round Robin with Interrupts
- Just interrupts
- Function Queue Scheduling
- Real time operating systems (RTOS)

Round Robin

Round Robin / Control Loop

Everything is a function call from the main loop

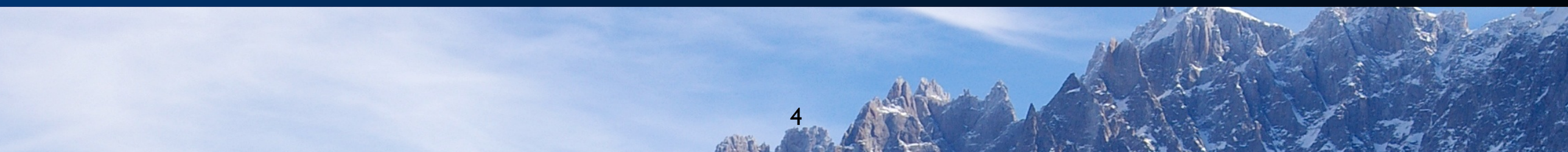
```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all devices have been serviced,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



Round Robin

- Low priority tasks need to be slowed down

```
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
    }
}
```

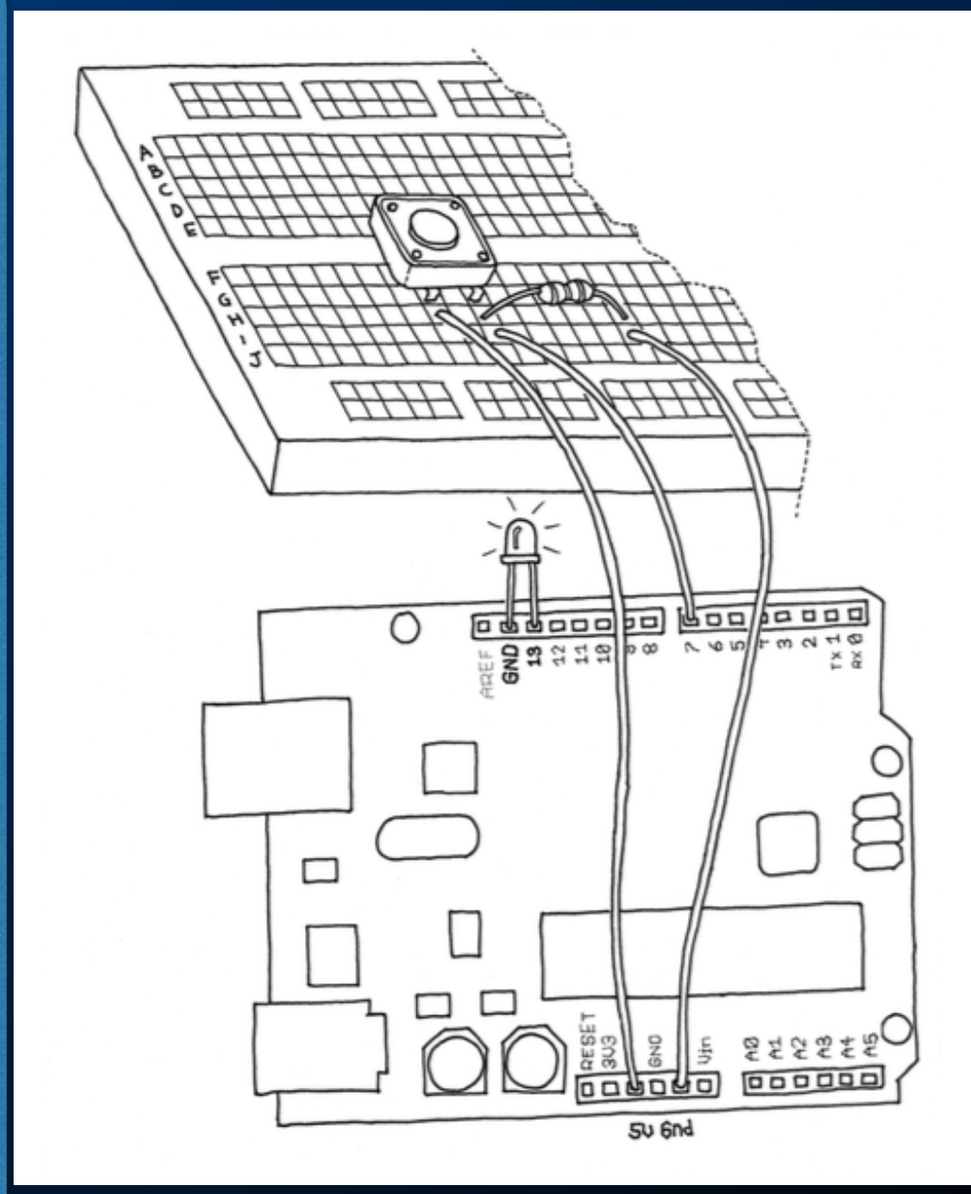


Round Robin

- Priority – None, everything runs in sequence.
- Response time – The sum of all tasks.
- Impact of changes – Significant. Changing the execution time of tasks or adding tasks impacts all other tasks.
- Simplicity, no shared data problems.

Arduino – Turn on LED while the button is pressed

```
const int LED = 13; // the pin for the LED
const int BUTTON = 7; // the input pin where the
                      // pushbutton is connected
int val = 0; // val will be used to store the state
             // of the input pin
void setup() {
  pinMode(LED, OUTPUT); // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
  }
void loop(){
  val = digitalRead(BUTTON); // read input value and store
it
  // check whether the input is HIGH (button pressed)
  if (val == HIGH) {
    digitalWrite(LED, HIGH); // turn LED ON
  }else{
    digitalWrite(LED, LOW);
  }
}
```

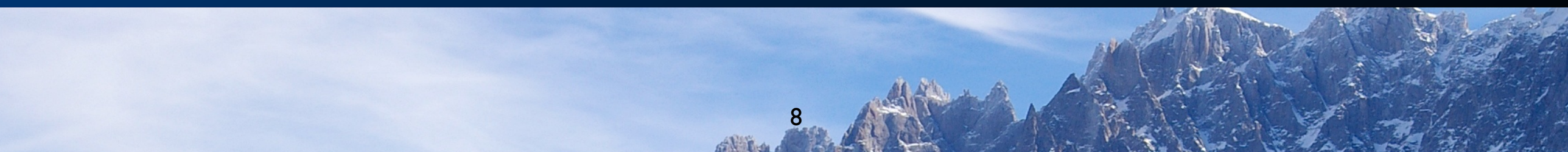


Arduino – Watchdog

```
#include <avr/wdt.h>
#define TIMEOUT WDTO_8S      // predefine time, refer avr/wdt.h
const int ledPin =  13;      // the number of the LED pin

void setup(){
    // disable the watchdog
    //wdt_disable();
    pinMode(ledPin,OUTPUT);
    // LED light once after start or if timeout
    digitalWrite(ledPin,HIGH);
    delay(1000);
    // enable the watchdog
    wdt_enable(TIMEOUT);
}

 void loop(){
    // process runing
    digitalWrite(ledPin,LOW);
    delay(9000); //if timeout trig the reset
    //feed dog
    wdt_reset();
}
```

State Machine

```
while(1) {
        switch(state) {
                case IDLE:
                        check_buttons();
                        LEDisplay_hex(NUM1);
                        if (BUTTON1 | BUTTON2 | BUTTON3)
                                state=SHOW;
                        break;
                case SHOW:
                        NUM1=0;
                        if (BUTTON1) NUM1 += 0x0001;
                        if (BUTTON2) NUM1 += 0x0010;
                        if (BUTTON3) NUM1 += 0x0100;
                        state=IDLE;
                        break;
        }
}
```



State Machine

- Similar to round robin, but only the current state gets executed.
- Each state determines the next state (non-sequential execution).

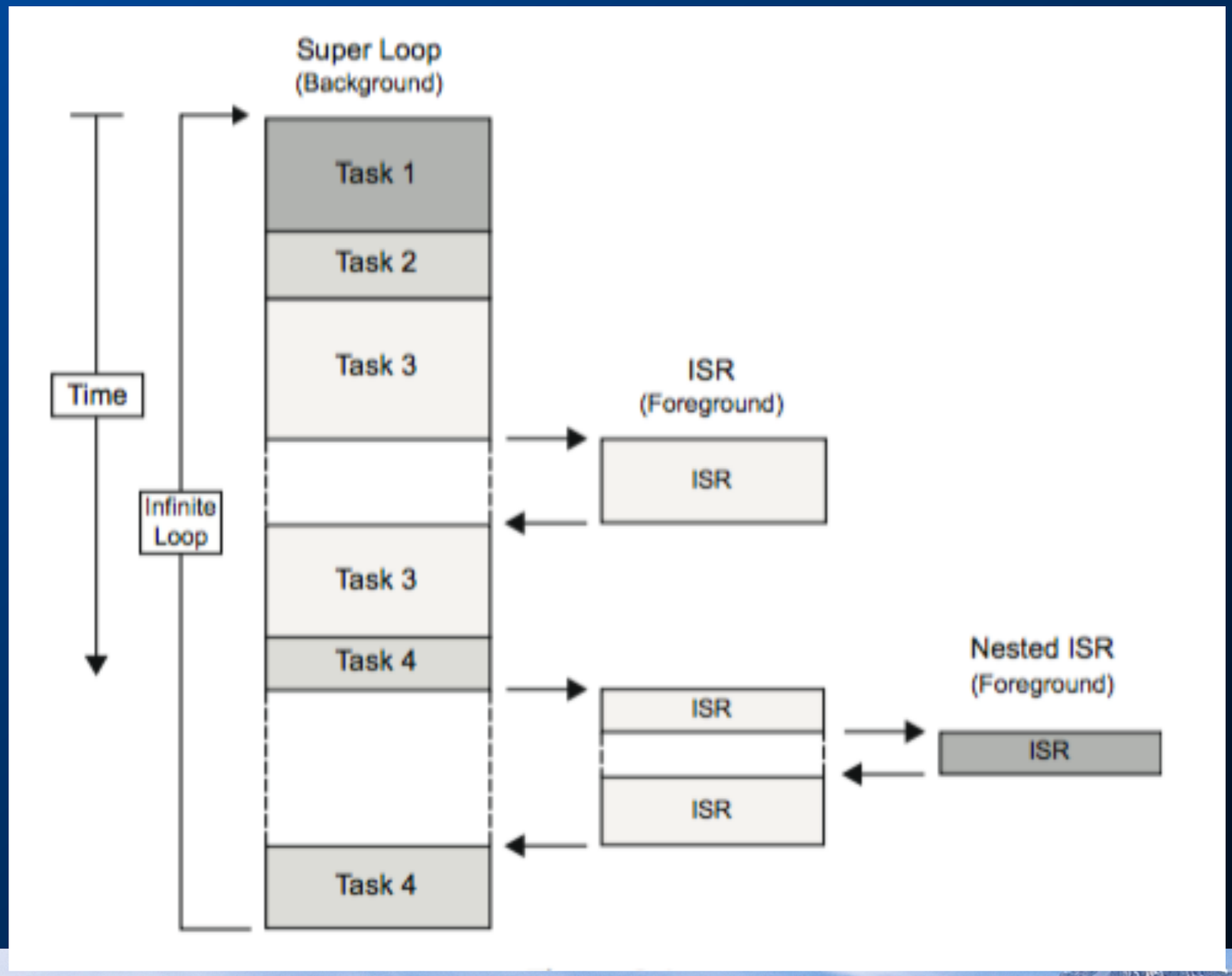
State Machine

- Priority – Each state determines the priority of the next state.
- Response time – The sum of all tasks.
- Impact of changes – Significant.
  Changing the execution time of tasks or adding tasks impacts all other tasks.
- Simplicity - No shared data problems.

Round Robin with Interrupts

```
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
        }
}
```

i.e. Foreground/background systems



https://doc.micrium.com/display/osiiidoc/Foreground-Background+Systems

Round Robin with Interrupts

- Priority – Interrupts get priority over main loop
  - Priority of interrupts as well
- Response time –
  - The sum of all tasks or
  - Interrupt execution time
- Impact of changes – Less significant for interrupt service routines. Same as Round Robin as main loop.
- Shared data – must deal with data shared with interrupt service routines

Just Interrupts

```
SET_VECTOR(P3AD, button_isr);
SET_VECTOR(TIMER1, display_isr);
while(1) {
;
}
```

Just interrupts

- Can have problems if too many ISRs
- If a high priority interrupt takes longer to execute than lower priority interrupts, then some will get missed.
  - Or you need to deal with nested interrupts.

Just Interrupts

- Priority – Interrupts get priority
- Response time –
  - Interrupt execution time
- Impact of changes – Little significant for interrupt service routines.
- Shared data – Must deal with data shared with interrupt service routines

Function Queue Scheduling

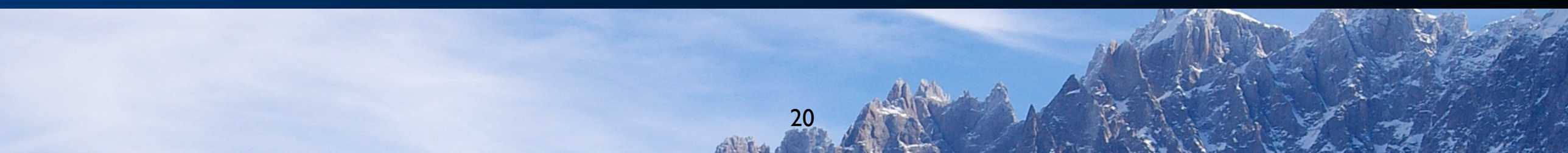
- Function pointers are added to a queue.
- The main loop cycles through the queue and executes tasks.
- Tasks or interrupts add new tasks to the function queue.
- The worst case timing

Function Queue Scheduling

```
#define MAX_TASKS 20
typedef int(*FuncPtr)();
FuncPtr tasks[MAX_TASKS]
int current_task = 0;

void add_task(FuncPtr func) {
        int n;
        for(n=current_task+1;n<MAX_TASKS-1;n++) {
            if(tasks[n]==NULL) {
                tasks[n]=func;
                return;
            }
        }
        for(n=0;n<current_task;n++) {
            if(tasks[n]==NULL) {
                tasks[n]=func;
                return;
            }
    }
```

```
id display_task() {
    LEDisplay_hex(NUM1);
    add_task(button_task);
}

void button_task() {
      check_buttons();
      NUM1=0;
      if (BUTTON1)     NUM1 += 0x0001;
      if (BUTTON2)     NUM1 += 0x0010;
      if (BUTTON3)     NUM1 += 0x0100;

      add_task(display_task);
}
```

```
main()  {
        LEDisplay_init();
        LEDisplay_clear();
        init_buttons();

        add_task(button_task);

        while(1) {
                if(tasks[current_task]==NULL) {
                ;
                }
                else {
                    (*tasks[current_task])();
                    tasks[current_task]=NULL;
                }
                current_task++;
                if(current_task>=MAX_TASKS) current_task=0;
        }
}
```



Function Queue Scheduling

- Priority – Interrupts have priority. Tasks execute in sequence
- Response time – Execution time of the longest task
- Impact of changes – Low. Interrupts manage priority functions. Queue manages lower priority.
- Shared data – must deal with data shared with interrupt service routines

Function Queue Improvements

- Include time scheduling

```
typedef int(*FuncPtr);

typedef struct {
    long timer;
    int status;
    FuncPtr;
} Task;
Task task_list[MAX_TASKS];
```

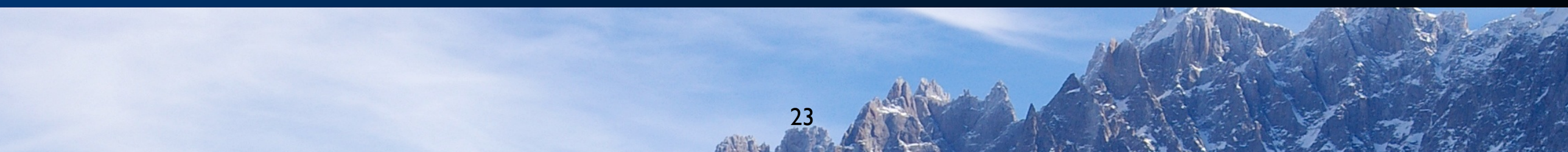
An interrupt decrements all task timers. When it reaches 0 its available for execution

Function Queue Improvements

Include task priority

```
typedef int(*FuncPtr);

typedef struct {
    int priority;
    FuncPtr;
} Task;
Task task_list[MAX_TASKS];
```

Highest priority tasks get moved to the head of the queue.



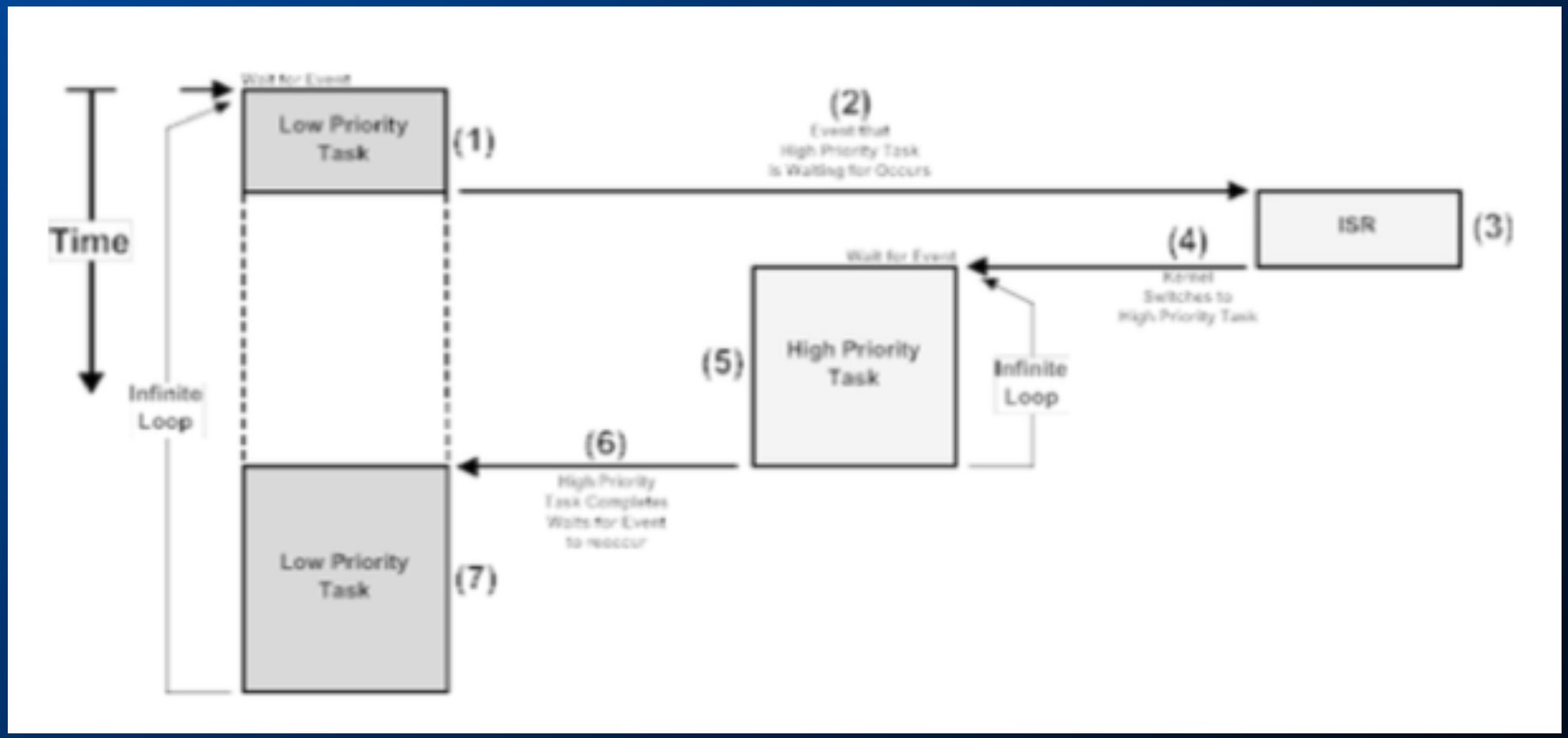
Function Pointers

- The Function Pointer Tutorial
  - http://www.newty.de/zip/e_fpt.pdf

Preemptive multitasking or multi-threading

- In this type of system, a low-level piece of code switches between tasks or threads based on a timer (connected to an interrupt). This is the level at which the system is generally considered to have an "operating system" kernel. Depending on how much functionality is required, it introduces more or less of the complexities of managing multiple tasks running conceptually in parallel.
- To access to shared data must be controlled by some synchronization strategy, such as message queues, semaphores or a non-blocking synchronization scheme.
- Because of these complexities, it is common for organizations to use a real-time operating system (RTOS), allowing the application programmers to concentrate on device functionality rather than operating system service.

Real-time kernels



Microkernels

- A microkernel is a logical step up from a real-time OS. The usual arrangement is that the operating system kernel allocates memory and switches the CPU to different threads of execution. User mode processes implement major functions such as file systems, network interfaces, etc.
- In general, microkernels succeed when the task switching and intertask communication is fast and fail when they are slow.

Monolithic kernels

- a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment.
  - examples: embedded Linux and Windows CE
- Despite the increased cost in hardware, this type of embedded system is increasing in popularity, especially on the more powerful embedded devices such as wireless routers and GPS navigation systems. Here are some of the reasons:
  - Ports to common embedded chip sets are available.
  - They permit re-use of publicly available code for device drivers, web servers, firewalls, and other code.
  - Development systems can start out with broad feature-sets, and then the distribution can be configured to exclude unneeded functionality, and save the expense of the memory that it would consume.
  - Many engineers believe that running application code in user mode is more reliable and easier to debug, thus making the development process easier and the code more portable.
  - Features requiring faster response that can be guaranteed can often be placed in hardware.

Thank you!