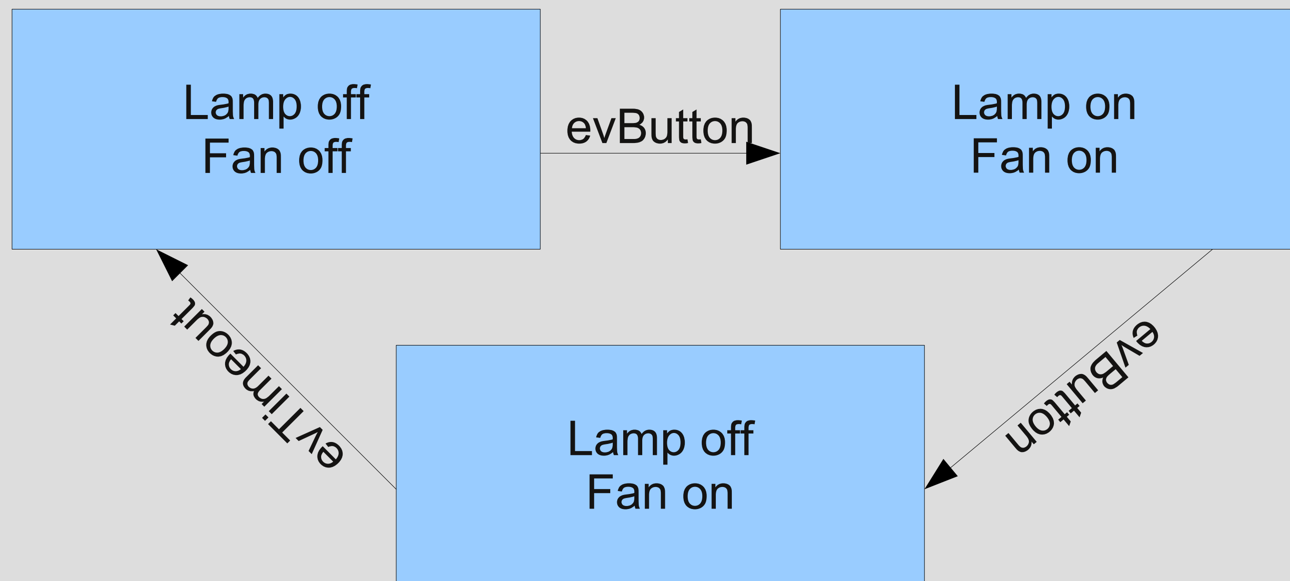
Cooperative Multitasking

Cooperative Multitasking

- let's make the controller for the lamp in an LCD projector
-



Code for LCD Projector

```
#define MSG_NONE 0
#define MSG_BUTTON 1
#define MSG_TICK 2
unsigned char buttonMsg=MSG_NONE, timerMsg=MSG_NONE;

void main(void) {
   // set up all the DDR's, the timer
   while (1) {
      if (buttonMsg) {
         projectorTask(buttonMsg);
         buttonMsg = MSG_NONE;
      }
      if (timerMsg) {
         projectorTask(timerMsg);
         timerMsg = MSG_NONE;
      }
   }
}
```

Code for LCD Projector

```
interrupt(TIMERB0_VECTOR) timer_isr(void) {
   timerMsg = MSG_TICK;
   // ack the interrupt
}

interrupt(PORT2_VECTOR) kbd_isr(void) {
   buttonMsg = MSG_BUTTON;
   // ack the interrupt
   P2IFG = 0;
}
```

Code for LCD Projector

```
#define STATE_OFF 0
#define STATE_ON 1
#define STATE_COOLING 2
unsigned char state = STATE_OFF;
int timer = 0;

void projectorTask(unsigned char msg) {
   switch (state) {
      case STATE_OFF:
         if (msg==MSG_BUTTON) {
            FAN = 1;
            LAMP = 1;
            state = STATE_ON;
         }
      break;
```

Code for LCD Projector

```
    case STATE_ON:
       if (msg==MSG_BUTTON) {
          LAMP = 0;
          timer = 5000;
          state = STATE_COOLING;
       }
       break;

    case STATE_COOLING:
       if (msg==MSG_TICK) {
          if (--timer==0) {
             FAN = 0;
             state = STATE_OFF;
          }
       }
       break;
    }
}
```

Projector code

- How would we add code that handles a button push while we're cooling?

Preemptive Multitasking

Preemptive Multitasking

- This technique allows us to write task code very simply, at the expense of a much more complicated O/S

Basic principles

- For each task, the O/S creates a PCB, and a stack
- The key to understanding Preemptive Multitasking is that there is one stack per task

As each task is created, the O/S finds a stack area (250 bytes? 500 bytes? 2k bytes?) and finds a PCB, and fills them in.

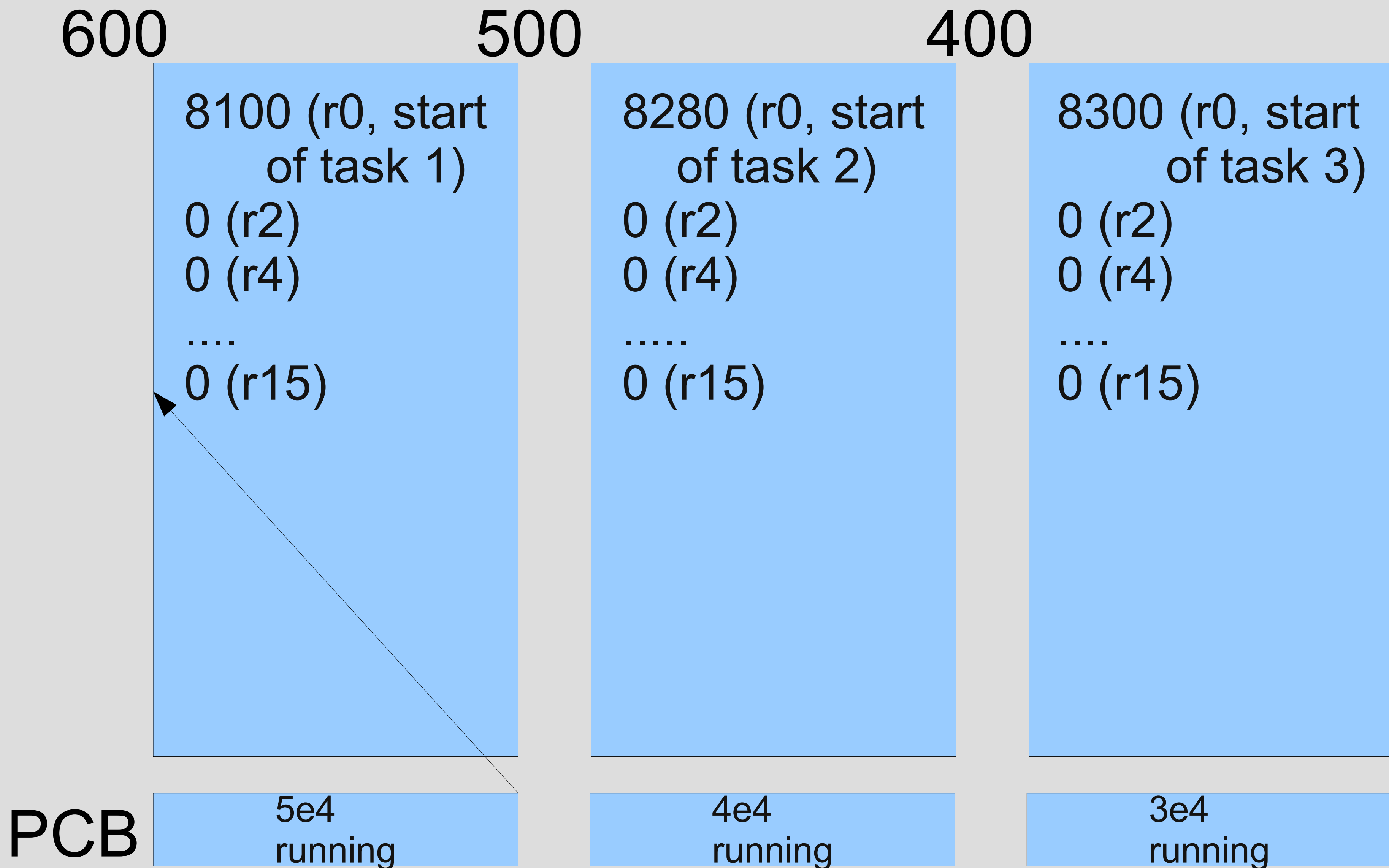
For each stack, it pushes the *start address* of the task and 13 zeros (sr/r2, r4-r15)

Process Control Block

- This can be a small structure, with as few as two items

```
struct PCB {
   unsigned int _SP;
   unsigned char _state;
};
#define PCB_STATE_RUNNING 1
#define PCB_STATE_BLOCKED 2
#define PCB_STATE_FINISHED 3
```

3 tasks ==> 3 stacks



O/S startup

- After creating the 3 stacks, the O/S goes into a simple cycle of checking all the PCB's, and for each one that is “running”, it sets the real SP to the _SP from the PCB, and then executes

  pop r15, pop r14.....pop r4
  RETI

cool, eh?

O/S activity

- Task 1 starts, and executes, until either
  - a timer tick, which causes an IRQ, jumps to the timer_isr, which pushes everything *back* on the stack
  - the timer_isr is owned by the O/S, which then goes back to the PCB list to see if there is another task to run
  - the O/S may select another PCB, and repeat
- OR
  - the task calls an O/S service
    - (next slide)

O/S services

- for example
  - waitForKey()
  - sleep()
- these are “blocking calls”

All these kinds of calls are implemented by

```
MOV #call_number,r15
CALL #softwareInterrupt
```

cool, eh?

pushes r2 and r4-r15 (all the registers) onto the stack, and goes back to the O/S

Task loading

- Consider: the O/S looks at all the PCB's and only runs the ones that are not "blocked"
  - 3 tasks, all "running"
  - each gets ~30% of the CPU (less the O/S overhead)

  - 3 tasks, 1 is "running"
  - it gets >90% of the CPU

  - 3 tasks, all are "blocked"
  - very common
  - the CPU can sleep, unless it needs to be polling I/O devices which don't have interrupts

Interrupts

- used to signal unexpected events
  - keystrokes
  - incoming packet
- a service complete
  - timer finished
  - disk read/write complete
  - packet sent
- to signal a time tick
  - to pre-empt a task (p.m.t.)
  - to create a tick message (c.m.t.)
- to request an OS service
  - a software interrupt

feature	brute force	cooperative multitasking	preemptive multitasking
code style	code in ISR, code in main	ISR's make messages; task code designed as subroutines	task code may “block”
timing accuracy	may be very accurate	may jitter by the time of the longest task	may jitter by nTasks * tickTime
O/S design	n/a	tiny: dispacher, messages	medium size; may be purchased
code size	smallest	a little bigger	requires an O/S
memory size	smallest	message queue	nTasks * stackSize *plus* PCB's and O/S variable
pitfalls	spaghetti code; critical sections; ISR's can get big	few	requires an O/S

LCD lamp code in pre-emptive

```
void projectorTask(void) {
   while (1) {
      waitForKey(); // a blocking call to the O/S
      LAMP = 1;
      FAN = 1;
      waitForKey(); // blocking again
      LAMP = 0;
      sleep(5000);  // blocks again, for 5 seconds
      FAN = 0;
   }
}
```

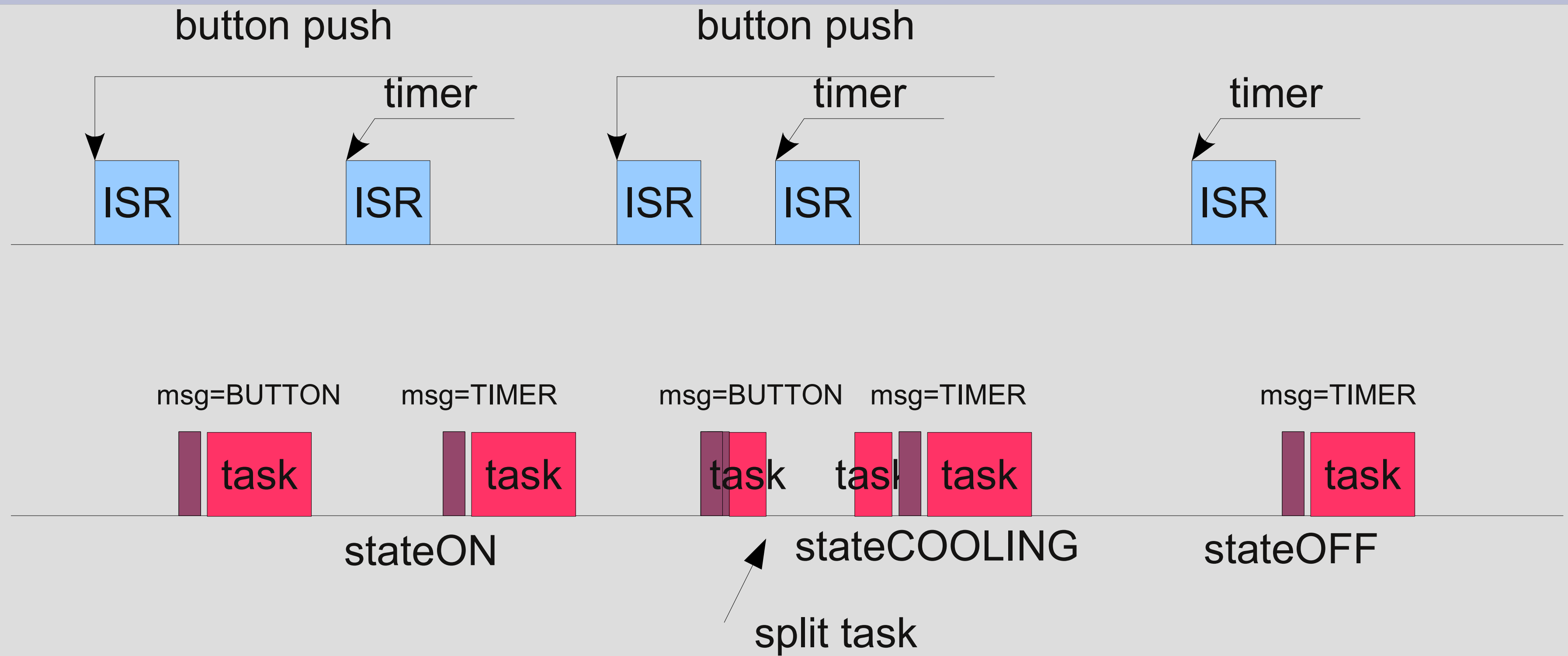
Much simpler code to write

The state machine goes implicitly from

ON->OFF->COOLING->ON

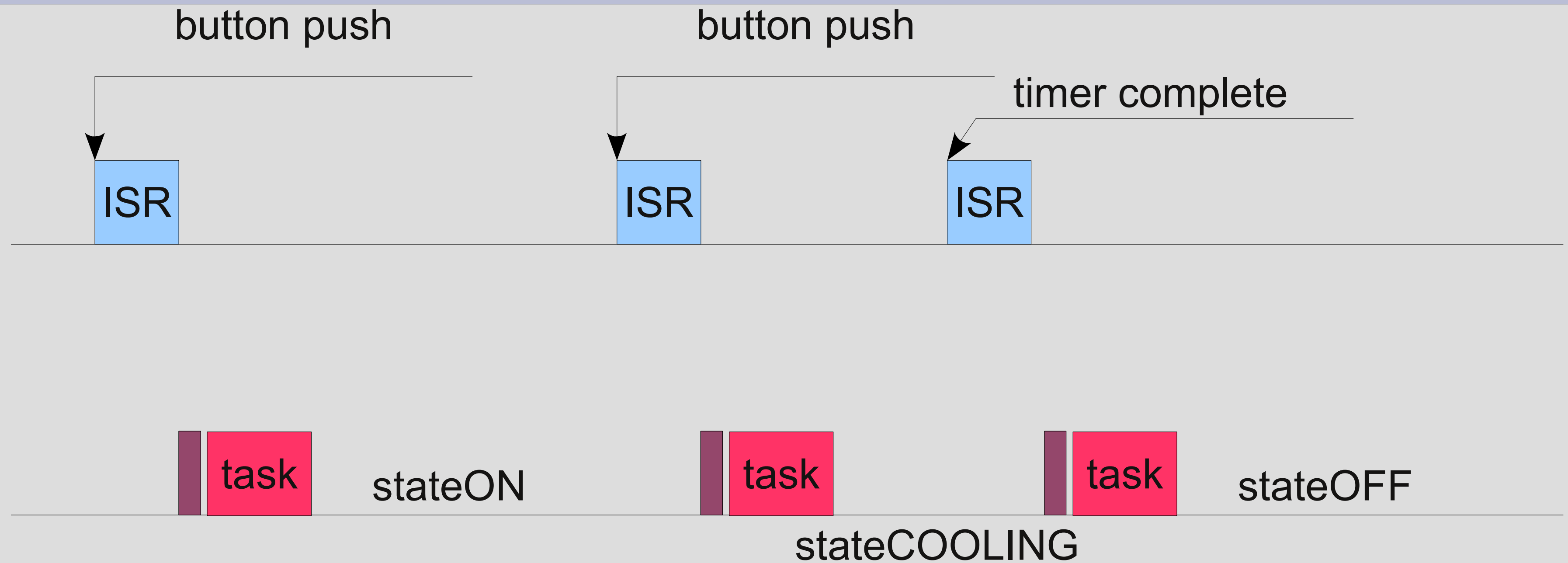
Some timing diagrams

The LCD fan timeline



This version has the timer ISR so that it generates a message on every time tick

The LCD fan timeline

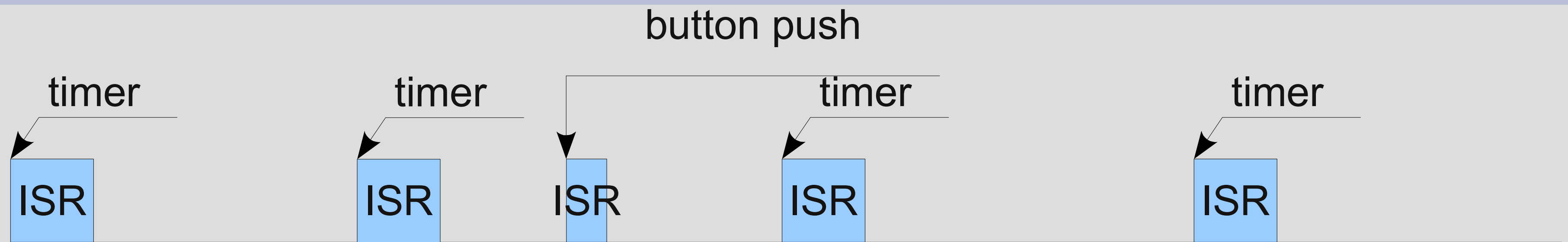


dispatcher

task

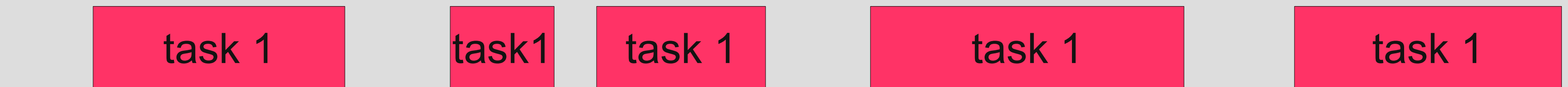
This version has the timer ISR so that it only generates a message when the timer finishes

The preemptive timeline for the car radio problem

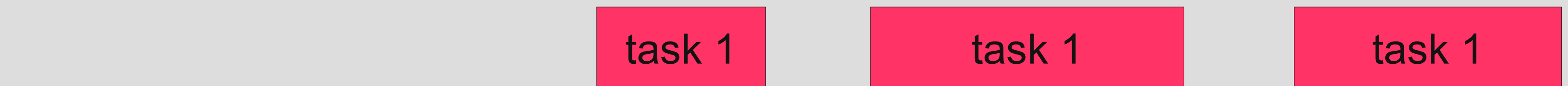


both tasks active

one task active

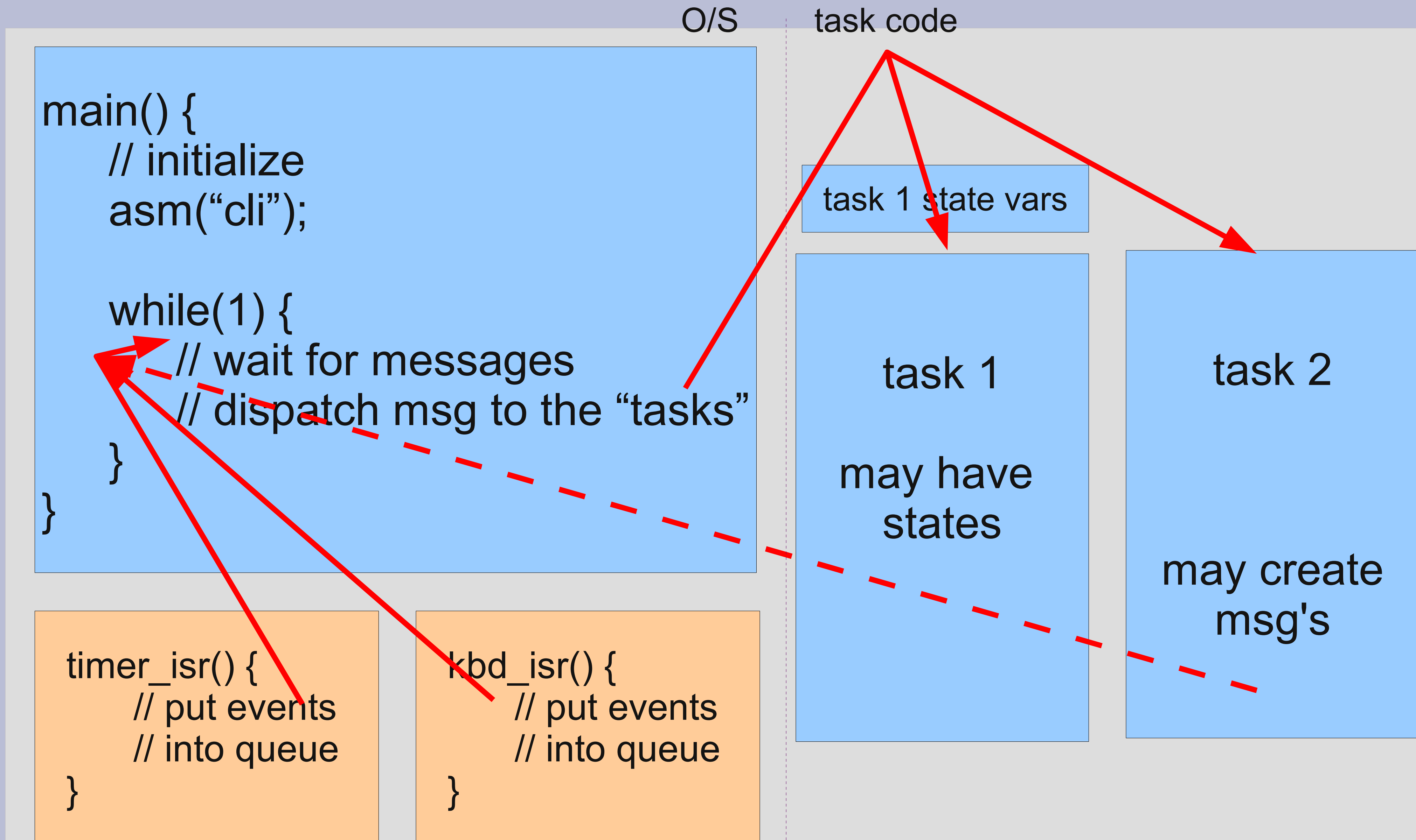


neither task active, one wakes up



Review

Cooperative Multitasking



Interrupts

- used to signal unexpected events
  - keystrokes
  - incoming packet
- a service complete
  - timer finished
  - disk read/write complete
  - packet sent
- to signal a time tick
  - to pre-empt a task (p.m.t.)
  - to create a tick message (c.m.t.)
- to request an OS service
  - a software interrupt

Sample Problem

- Create a packet monitor
  - watches and classifies incoming packets
    - into ICMP, ARP, TCP, UDP
    - counts the number of each packet
  - if the user presses '#', display the totals on the screen, and zero out the counts in memory
  - display the type and length on the LCD

```
packet type:    UDP TCP ARP ICMP
packet count: 1332 956   1   41
```

  - first, code up the problem *without cooperative multitasking*

Sample Problem

- Proposed solution
  - main() code watches the keyboard, continually checking for '#'
    - when the '#' arrives, copy the totals to the LCD display and zero out the globals
    - the user can total the LCD numbers for an accurate count
  - the ethernet interrupt code increments the packet counts in globals

```
// global variables to hold the packet data
#define PKT_UDP 0
#define PKT_TCP 1
#define PKT_ARP 2
#define PKT_ICMP 3
unsigned int eth_pkt_counts[4];
```

Sample Problem

- main()

```
int main(void) {
   initHardwareAndInterrupts();
   eint();

   while(1) {
      char k = checkKeyboard();
      if (k == '#') {
         displayCounts(eth_pkt_counts);
         eth_pkt_counts[0] = 0;
         eth_pkt_counts[1] = 0;
         eth_pkt_counts[2] = 0;
         eth_pkt_counts[3] = 0;
      }
   }
}
```

Sample Problem

- ethernet interrupt

```
interrupt(ETHERNET_VECTOR) packet_isr(void) {
    int r = ETH_DEVICE_TYPE;   // read the device

    eth_pkt_counts[r] += 1;

    // and ack the interrupt
    ETH_DEVICE_IFLG = 0;
}
```

Sample Problem

- What's wrong with the last 2 slides?

- What if a packet arrives after the
  `if (k == '#')`

Sample Problem Rewrite

- the problem is that we have *multi byte data* that is being accessed by two levels of program
  - if the interrupt level changes the data without the main level knowing about it, the data is spoilt
- there are several ways to fix this problem
  - we will examine two ways:
    - critical section
    - cooperative multitasking

Critical Section

- in the main(), there is an area of code that will only operate correctly *if interrupts are blocked*
  - this area is called a critical section

```
while(1) {
    char k = checkKeyboard();
    if (k == '#') {
       dint();
       displayCounts(eth_pkt_counts);
       eth_pkt_counts[0] = 0;
       eth_pkt_counts[1] = 0;
       eth_pkt_counts[2] = 0;
       eth_pkt_counts[3] = 0;
       eint();
    }
}
```

C.M.T. Solution

- another way to fix this is to *take the processing code **out** of the interrupt level*
  - and put it in as a task
  - task 1: increment the totals
  - task 2: check the keyboard for '#' and update display

```
// messages
#define MSG_PKT_UDP  1
#define MSG_PKT_TCP  2
#define MSG_PKT_ARP  3
#define MSG_PKT_ICMP 4
#define MSG_KEY_HASH 100
#define MSG_EMPTY    0
typedef char Message;
Message msgs[3] = {MSG_EMPTY, MSG_EMPTY, MSG_EMPTY};
```

C.M.T. Solution

- and we still keep the globals

```
// global variable to hold the packet data

unsigned int eth_pkt_counts[5];

// udp count goes into eth_pkt_counts[1]
// tcp count goes into eth_pkt_counts[2]
// etc, etc
// eth_pkt_counts[0] is not used
```

C.M.T. Solution

- main

```
int main(void) {
    int i;
    initializeHardwareAndInterrupts();

    eint();

    while(1) {
        for(i=0; i<3; i++) {
            while (msgs[i] == MSG_EMPTY)
                ;       // wait for message
            task_update_totals(msgs[i]);
            task_check_keyboard(msgs[i]);
            msgs[i] = MSG_EMPTY;
        }
    }
}
```

C.M.T. Solution

- ethernet interrupt (almost no processing)

```
interrupt(ETHERNET_VECTOR) packet_isr(void) {
   int r = ETH_DEVICE_TYPE;  // read the pkt type
   newMessage(r);

   // and ack the interrupt
   ETH_DEVICE_IFLG = 0;
}
```

- keyboard interrupt

```
interrupt(PORT2_VECTOR) kbd_isr(void) {
   int k = measureKeyboardToHex();
   if (k==11)
      newMessage(MSG_KEY_HASH);
   P2IFG = 0;
}
```

C.M.T. Solution

- task code

```
void task_update_totals(Message m){
   // we only care about message 1..4
   if (m >= MSG_PKT_UDP && m <= MSG_PKT_ICMP)
      eth_pkt_counts[m]++;
}

void task_check_keyboard(Message m) {
   if (m == MSG_KEY_HASH) {
      displayCounts(eth_pkt_counts);
      eth_pkt_counts[1] = 0;    // udp
      eth_pkt_counts[2] = 0;    // tcp
      eth_pkt_counts[3] = 0;    // arp
      eth_pkt_counts[4] = 0;    // icmp
   }
}
```

Terms

- as used in this course
- assembly language programming
  - *directive*: an assembly language instruction which does not generate code; instead, it directs the flow of code creation, creates data, inserts entries into the symbol table, etc

Terms

- interrupts
  - *ISR*: interrupt service routine; similar to a subroutine; called when an interrupt is serviced; ends with an 'reti'; preserves all registers
  - *vector table*: a reserved area at the end of flash (the .text section) which holds the addresses of interrupt and restart code

Terms

- interrupts
  - *pending:* an interrupt request which has not yet been serviced by the system (blocked by either the global or local interrupt gate)
  - *latency*: the time delay before the code can react to an interrupt
    - in c.m.t. the latency includes the time delay before the relevant task is executed

Terms

- C code
  - *static:* (applied to variable inside a subroutine) the variable exists *before* and *continues* to exist after the subroutine finishes
  - *typedef:* allows you to create a new *type*

Terms

- multitasking
  - *event:* some physical event which triggers interrupt code to insert an element into the event queue
  - *message*: pseudo-events which are a created by a task, in order to communicate with other *tasks*
  - *dispatcher*: a program which sends *events* to *task* code

Terms

- multitasking
  - *task:* a group of code which handles one independent operation
  - *state*: a small number with which the *task* code can keep track of what is should do, in between *events* & *messages*
  - *critical section*: a section of code which must not be interrupted (to preserve data integrity)