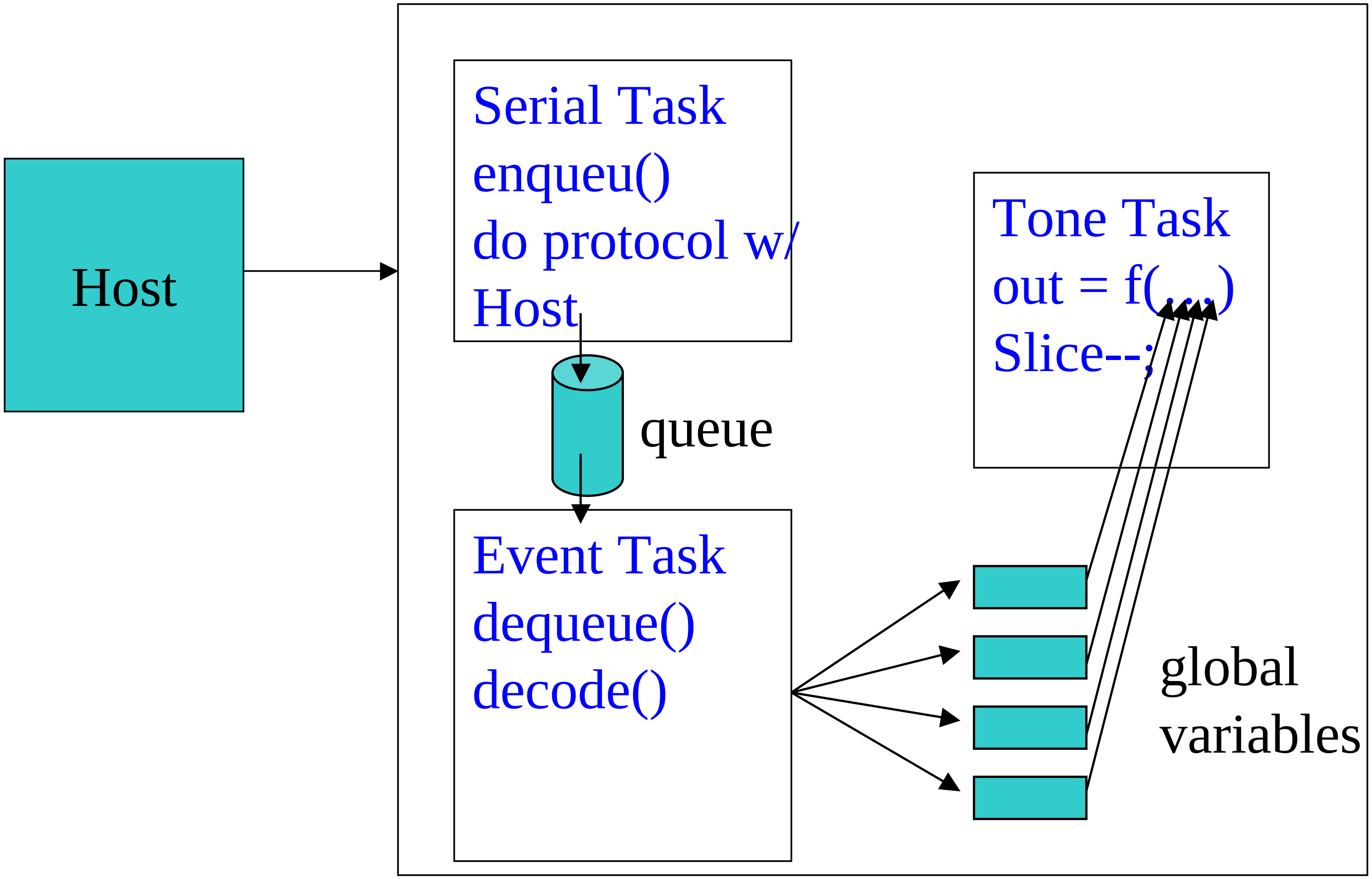
Embedded Software Architectures

- No Operating System
    - Round robin: sequential polling for events
    - Round robin w/ interrupts
    - Function Queue Scheduling
- Real Time Operating Systems

Overall Architecture



Host

Serial Task
enqueu()
do protocol w/
Host

queue

Event Task
dequeue()
decode()

Tone Task
out = f(….)
Slice--;

global
variables

FIFO

- Empty Condition

  Tail = Head

  - Tail is place to get next byte if != head

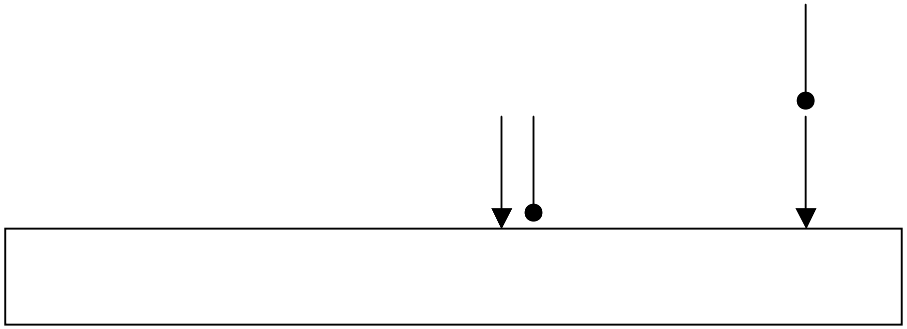
- Initial Condition

  Same as empty condition

- Full Condition

  Head = (Tail –1)

  - Head is place to put next byte if not right behind tail



FIFO Queue

- Event Task

  If (not empty) get item, move “tail” pointer
  else what?

- Serial Task

  If (not full) put item, move “head” pointer,
    and acknowledge received byte
  else what?

- When writing this code:

  Think of Event and Serial as parallel processes running on separate computers using shared memory to communicate: why?

Example

- Queue size is SIZE

  In Event Task

  ```
  void Event () {
      while (head == tail); // wait until queue not empty
      data = queue[tail];
      tail = tail + 1
      if (tail == SIZE) tail = 0;
      process(data);
  }
  ```

- Do we have a problem?

  If serial gets control after tail is incrememted pasted the end of the queue, the serial process could fail to detect when the queue if full, causing data to be lost due

- Solution?

  Disable serial interrupts during critical section

- Note, we assume that Event is interruptable, so we make blocking on empty queue.
  and we assume that event is run only when the time slice runs out.

Safe Queue?

```
void Event() {
        while (head == tail); // wait until queue not empty
        data = queue[tail];
        ES = 0;                      // disable serial interrupts
        tail = next(tail);
        ES = 1                       // enable serial interrupts
        process(data);
}
unsigned char next(unsigned char ptr) {
        if (++ptr == SIZE) ptr = 0;
        return(ptr);
}
```

Consider the compiler

C code for blocking queue

```
void Event() {
    while (head == tail);
    data = queue[tail];
    ES = 0;
    tail = next(tail);
    ES = 1
    process(data);
}
unsigned char next(unsigned char ptr) {
    if (++ptr == SIZE) ptr = 0;
    return(ptr);
}
```

Compiler output?

```
         MOV R0,TAIL
         MOV R1,HEAD
LOOP:    MOV A,R1
         SUBB A,R0
         JZ      LOOP
         MOV  R3,#QUEUE
         ADD   R3,R0
         MOV   R4, @R3
         CLR   ET0
         ACALL NEXT
         MOV TAIL,A
         SETB ET0
```

Assume next() operates on R0, returns in A

I think we’re safe now

```
volatile data unsigned char head, tail;
…
viod Event() {
      while (head == tail);
      data = queue[tail];
      ES = 0;
      tail = next(tail);
      ES = 1
      process(data);
}
unsigned char next(unsigned char ptr) {
      if (++ptr == SIZE) ptr = 0;
      return(ptr);
}
```

```
         MOV R0,TAIL
LOOP:    MOV R1,HEAD
         MOV A,R1
         SUBB A,R0
         JZ     LOOP
         MOV  R3,#QUEUE
         ADD  R3,R0
         MOV   R4, @R3
         CLR  ET0
         ACALL NEXT
         MOV TAIL,A
         SETB ET0
```

***compiler knows that sfr’s are volatile (ports, flags)**

M-BOX in Round Robin Arch.

```
volatile bit fTNEexpired;
void main (void) {
        if (TF0) tone();            // process timer, set expired bit
        if (R1) serial();           // process serial input
        if (fTNEexpired) event();
}
```

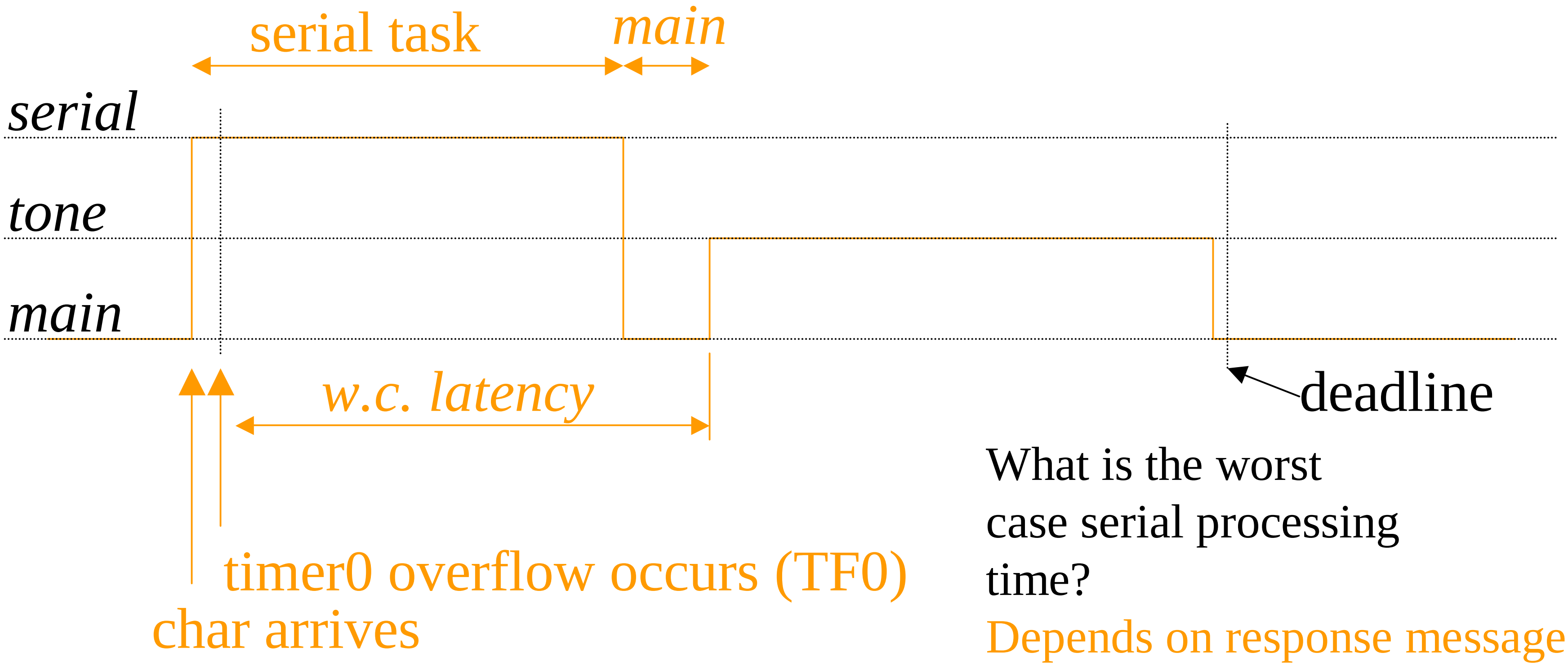
Would this work for the M-BOX?

How do we know?

Task Diagram

Worst case: character arrives one cycle before TF0

Worst Case Latency = Σmax run time of all other tasks



What is the worst case serial processing time?

Depends on response message length--could be bad!

How much latency can we tolerate? Practically none

Round Robin w/ Interrupts

```
volatile bit fEvent;
void timer_isr(void) {
    time_critical_processing();
    if (…) fTNEexpired = TRUE;
}
void main (void) {
    if (R1) serial_input_task();
    if (fTNEexpired) {
            Event();
            fEvent = FALSE;
    }
}
```

Why not put Event() into the ISR too?

Then our worst case latency to a other time critical processing would be poor

Would this work for the M-BOX? See next slide

M-BOX in RR+INT

```
volatile bit fEndOfSlice, fSerial;
void tone_isr(void) interrupt … {
        process_tones();
        if (!--sliceCount) {
                changeTones();
                sliceCount = SliceSize
                fEndOfSlice = TRUE;
        }
}
void serial_isr(void) interrupt …{
        timeCritical();
        fSerial = TRUE;
}

main () {
        if (fSerial) {process_serial_data(); fSerial = FALSE;}
        if (fEndOfSlice) {
                if (--TNE==0)
                                process_next_event();
                fEndOfSlice = FALSE;
        }
}
```

What are the time critical functions?

compute output, timeslice countdown

What are the event functions?

TNE count down event decoding (TNE, Tones)

Do these have hard time constraints too?

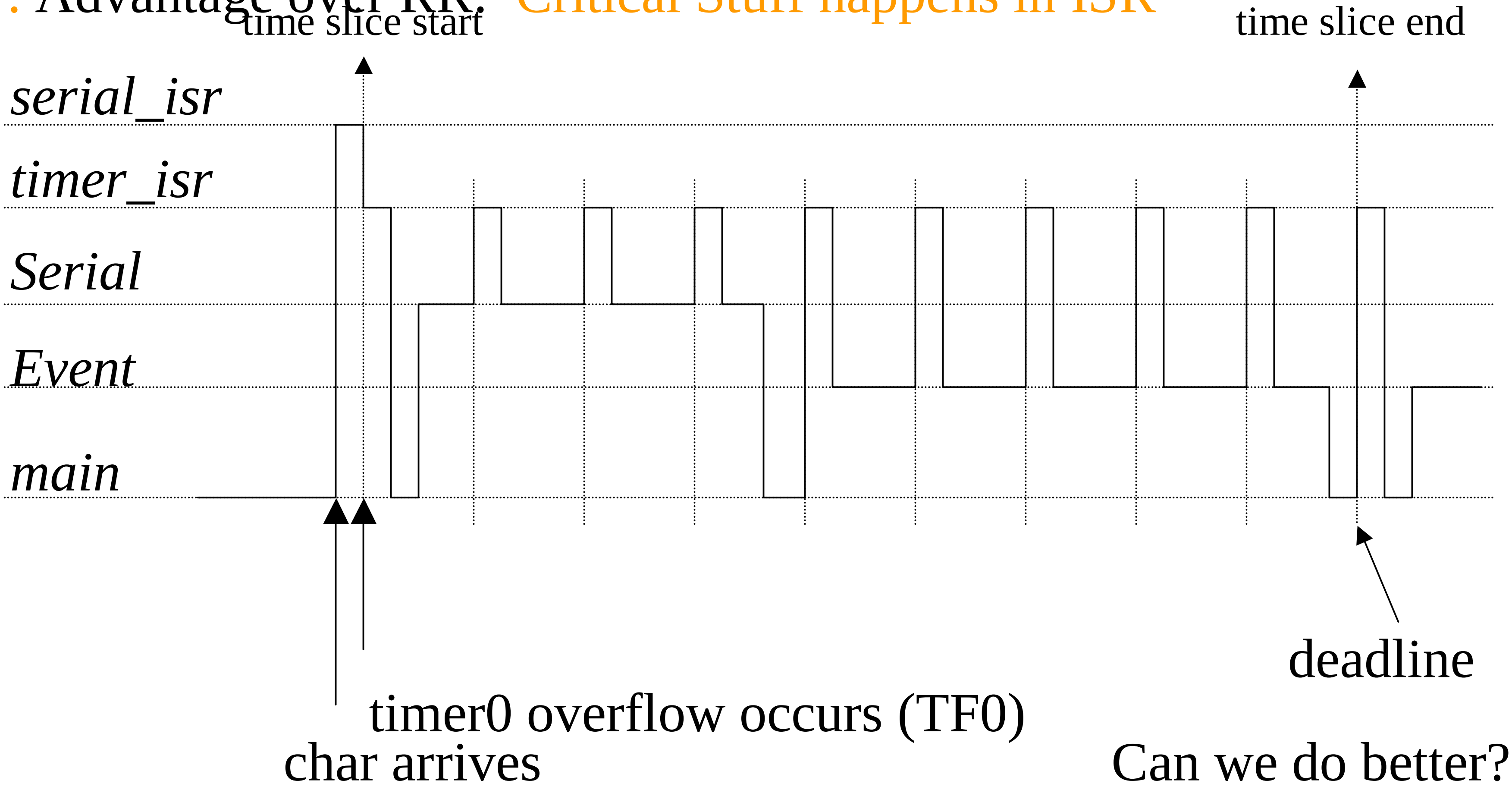
Yes, one time slice, is it good enough? Not necessarily…serial is undefined!

Task Diagram

Worst case analysis: character comes one cycle before TF0

Worst case latency: for ISR: sum of all same or higher priority ISR's. For tasks: Sum of all tasks

. Advantage over RR: Critical Stuff happens in ISR



Can we do better?

Function Queue

```
void isr(void) interrupt … {
        process_tones();
        if (!--sliceCount) {
                changeTones();
                sliceCount = SliceSize;
                enq(Event);
        }
}
void serial(void) interrupt …{
        SerialTimeCritical();
        enq(Serial);
}

void main(void) {
        while (1) if (f = deq()) { *f());
}
```

You get a scheduling opportunity every time a task completes.

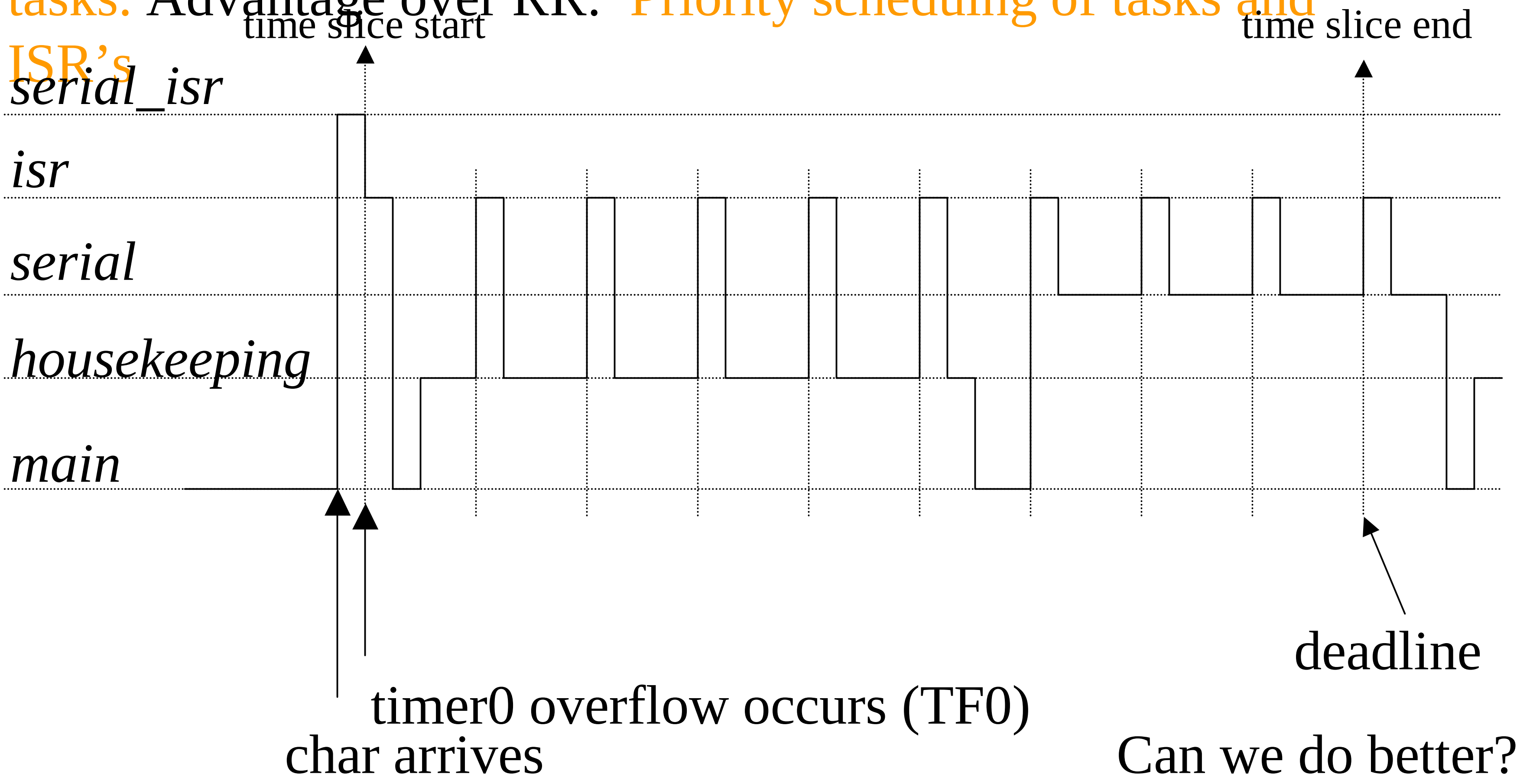
What is the advantage of this? Programmer can set priority for task functions.

Worst case latency for priority *n* task function? Sum of max execution time for all task functions of priority > *n* + max current task

Task Diagram

Worst case analysis: character comes one cycle before TF0

Worst case latency: for ISR: sum of all higher or equal priority ISR’s, for Task: Max Task + Sum of all higher or equal priority tasks. Advantage over RR: Priority scheduing of tasks and ISR’s



Can we do better?

Comparison Non OS Architectures

- See Chapter 5, table 5.1 Simon