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
enqueue()
do protocol with Host

Event Task
decode()
dequeue()

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

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

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

q 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

```c
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 incremented past the end of the queue, the serial process could fail to detect when the queue is 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.
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

```c
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?

```assembly
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
volatile data unsigned char head, tail;

... 
vioid 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 knows that sfr’s are volatile (ports, flags)
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 = $\sum_{\text{max run time of all other tasks}}$

What is the worst case serial processing time?
Depends on response message length--could be bad!
How much latency can we tolerate? Practically none
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
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;
    }
}
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

serial_isr

timer_isr
time slice start

time slice end

Serial
time slice start
time slice end

Event
time slice start
time slice end

main
time slice start
time slice end

timer0 overflow occurs (TF0)
char arrives
deadline
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();
}

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 $> n + \text{max current task}$

You get a scheduling opportunity every time a task completes.
**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 scheduling of tasks and ISR’s

char arrives

---

time slice start

time slice end

deadline

Can we do better?
Comparison Non OS Architectures

See Chapter 5, table 5.1 Simon