Real Time Operating Systems

- What is the basic thing we want the OS to do to help us improve worst case latency? Enable multithreading
- How? Define an OS time-slice (tick) at which highest priority 'runnable' task is continued. Priority function determines response behavior.
- Simplest Scheduling algorithm: each task gets at most 1 tick at a time to run. Round Robin Scheduling. Worst case task latency = #tasks*tick. Worst case run time = ticks/task * #tasks
- Some properties of such system: liveness, safety, fairness, latency, overhead.
- Other niceties: Device Drivers, Synchronization, Message passing, Memory Management

Features of an Embedded Operating System

- Interrupt latency
- System call overhead (Various functions…task switch, signal, create, delete)
- Memory overhead
- Tasks (threads)
- Scheduling Algorithms
- Communication and synchronization primitives (tools)
- Memory Management

Comparative Real Time OSes

	RTX51 Full	RTX51 Tiny
Maximum Number of Tasks	256	16
Maximum Active Tasks	19	16
CODE Space Required	6-8 Kbytes	900 Bytes
DATA Space Required	40-46 Bytes	7 Bytes
Stack (**IDATA**) Space Required	20-200 Bytes	3 Bytes for each task
XDATA Space Required	650 Bytes minimum	–
Timer Used	0, 1, or 2	0
System Clock Divisor	1,000-40,000 cycles	1,000-65,535 cycles
Interrupt Latency	< 50 cycles	< 20 cycles
Context Switch Time (Fast Task) (depends on stack load)	70-100 cycles	–
Context Switch Time (Standard Task) (depends on stack load)	180-700 cycles	100-700 cycles
Task Priority Levels	4	–
Semaphores	8 maximum	–
Mailboxes	8 maximum	–
Mailbox Size	8 entries	–
Memory Pools	16 maximum	–

Compare to uClinux at ~400Kbytes.

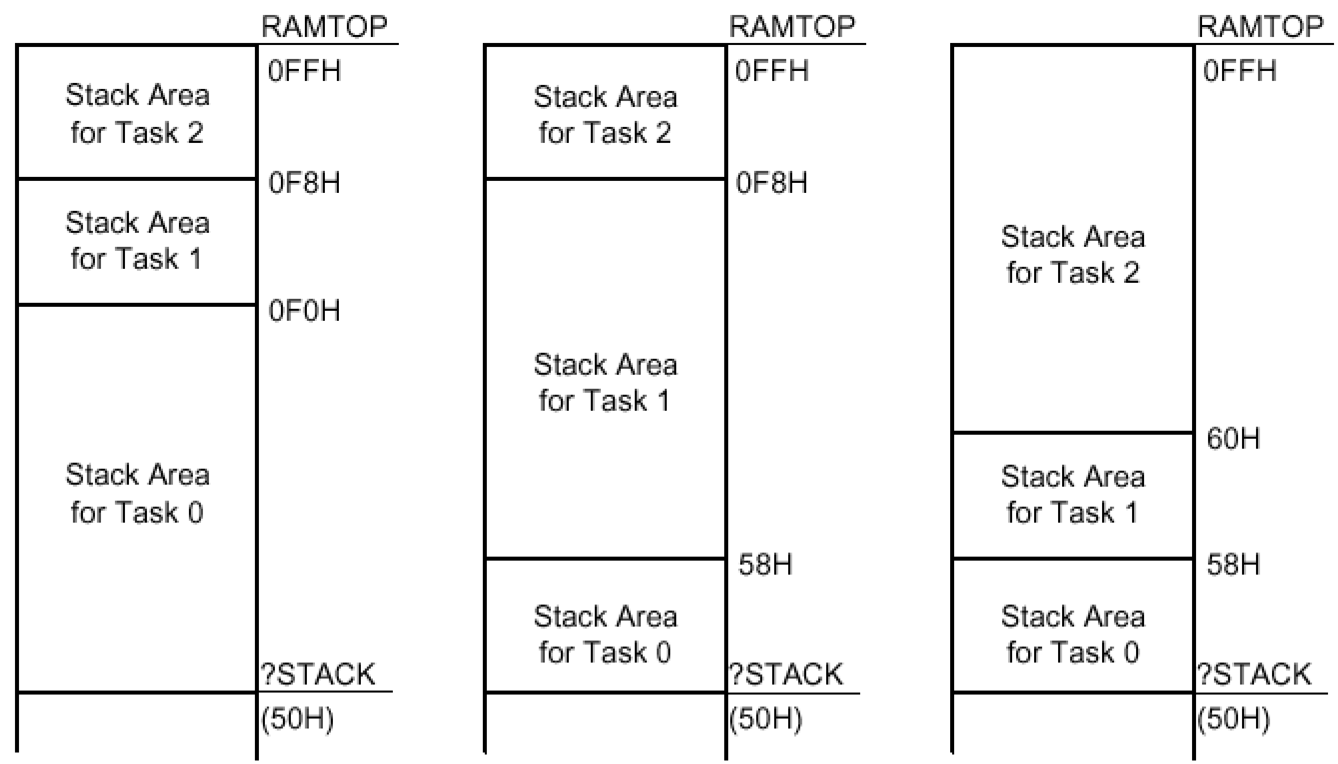
what for?

What is this?

38us – 280us why the variable?

actually 16 semaphores

Stack Management

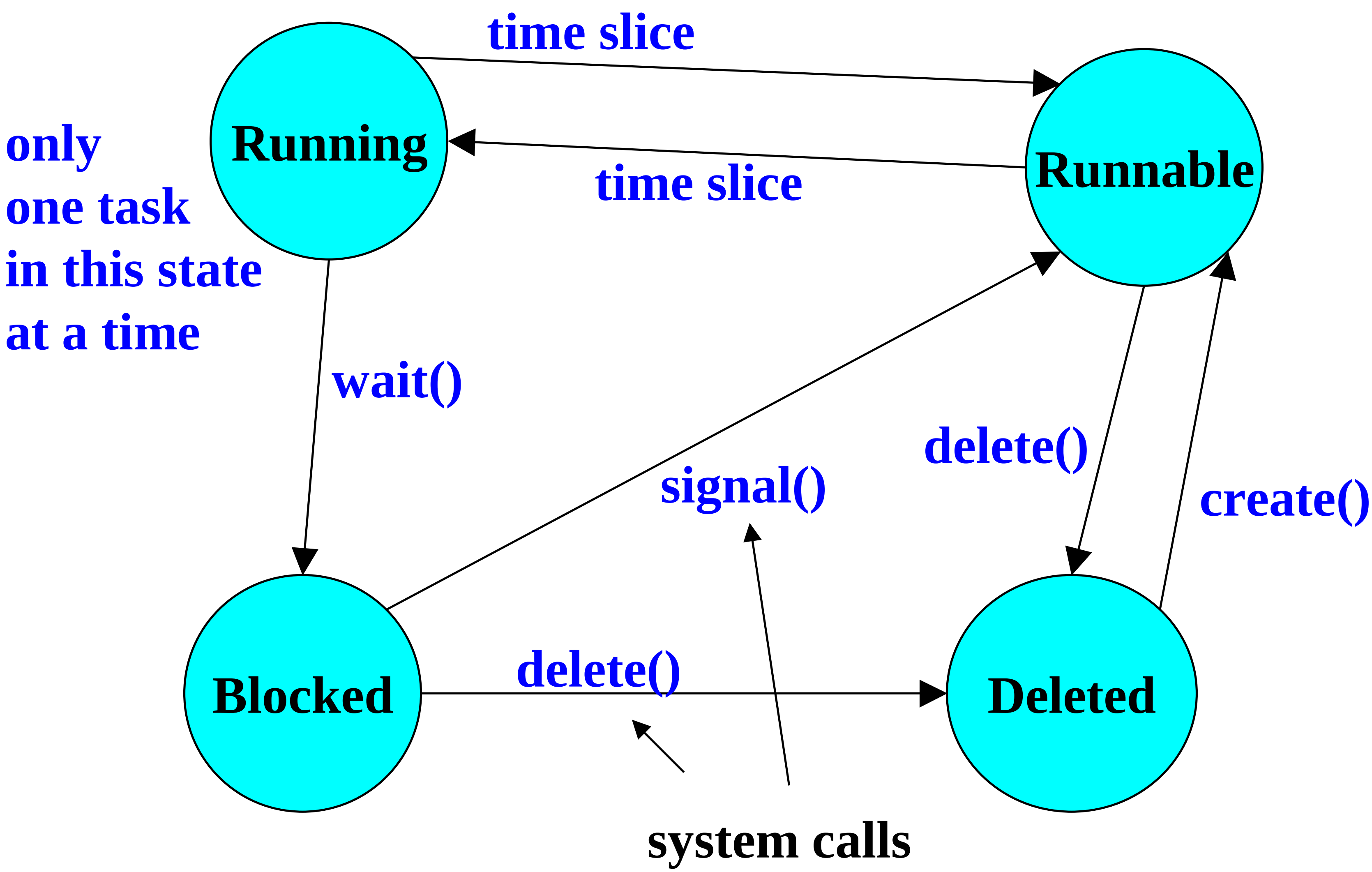


Stack Assignment for Task0 = Running Task

Stack Assignment for Task1 = Running Task

Stack Assignment for Task2 = Running Task

Multitasking – state maintained for each task



Programmers View of Tiny OS

```
void tone_isr(void) interrupt … {
        process_tones();
        if (!--sliceCount) {
                updateToneParameters();
                sliceCount = SliceSize
                isr_send_signal(MUSIC);
        }
}
void serial_isr(void) interrupt …{
        timeCritical();
        os_send_signal(SERIAL);
}
void play(void)  _task_ MUSIC {
        os_create(SERIAL);
        while (1) {os_wait();
                process_next_event();}
}
void serial(void) _task_ SERIAL {
        while (1) {os_wait();
                process_serial_data();} // os_create(MUSIC)?
}
```

Advantages:

•Deterministic response time even w/ non deterministic tasks lengths.

• Incremental development

Resources:

•Task switching overhead

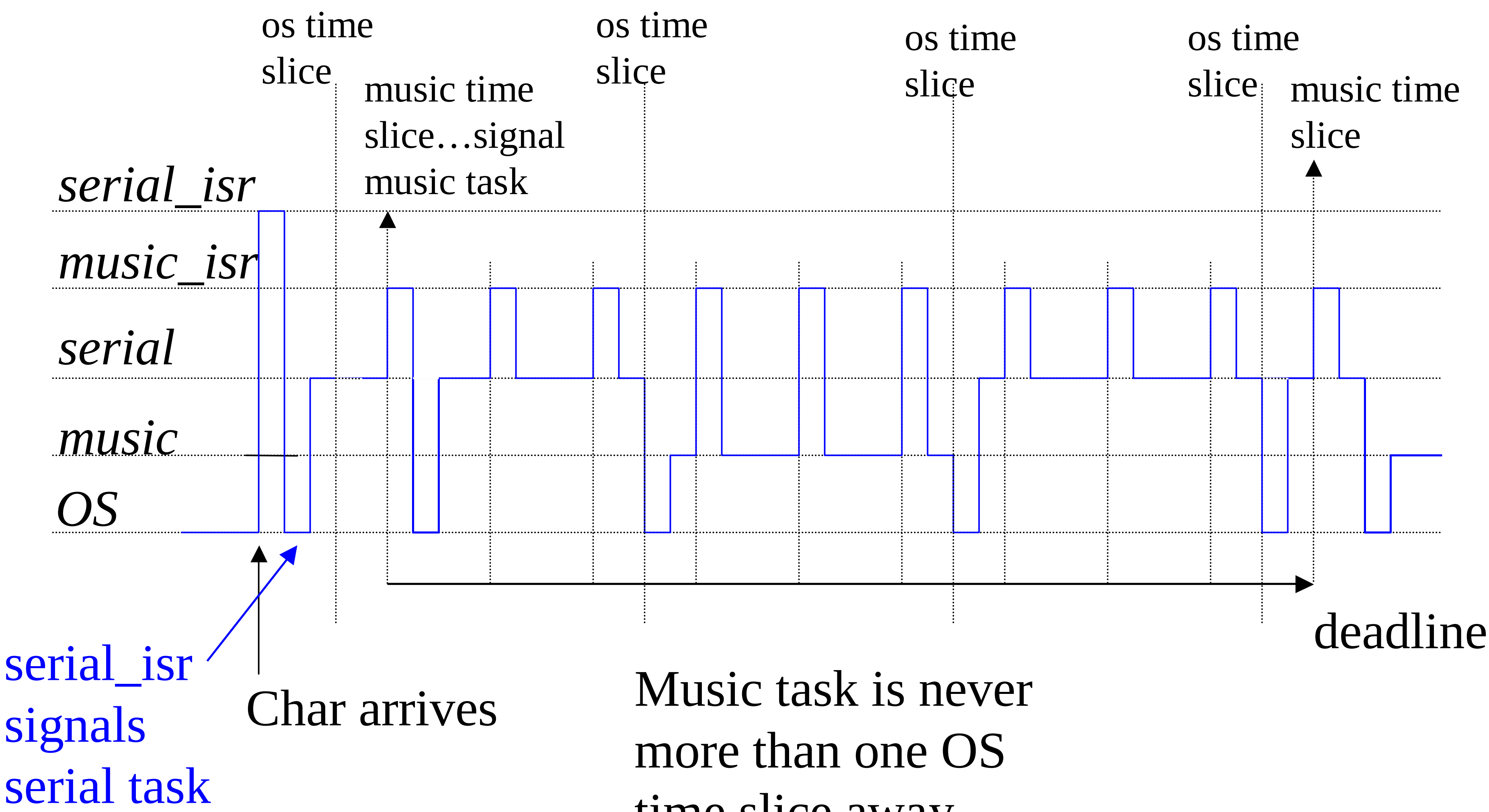
•Memory overhead

•Use of system timer

•Degrades best case response time.

Tasks are threads

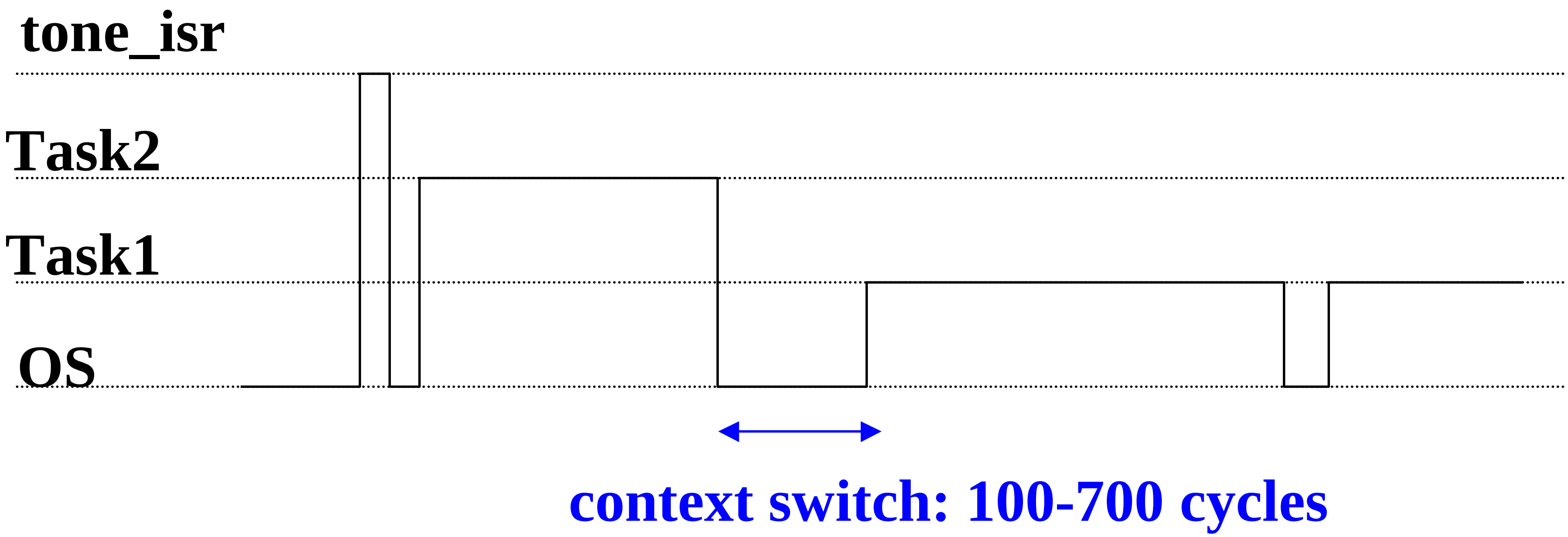
Task Diagram



Music task is never more than one OS time slice away

Interrupt Priorities

- Key question: Is there a bad time to get a tone gen interrupt?



Another Solution

- Multiprocessor: Dedicate one processor to each (or a few) tasks.
- Still need **synchronization** and **communication**.
- An M-BOX network could be an example of a multiprocessor system.
  A synthesizer w/ mutltiple notes and “voices”

Process v. Thread

- Process:
  Each process runs in a separate address space. Address 0x1 in process one is not the same memory location as address 0x1 in another process.
  Context switching is expensive:
  - need to reload memory management variables
  - may need to invalidate cache or do other cache coherency tricks
  - Anything address based needs to be saved and restored

  Threads: lightweight
  - All threads run in the same address space
  - Still have same basic state machine (ready, running, blocked, killed)
  - Still need context switching for registers, stack.

Reentrant functions...sharing code not data

- Are there shared functions that we would like to have?
  - deq?
  - enq?
  - next (same for head or tail)?
  - Q declaration and initialization?
- Can task switching clobber local variables (parameters and automatics)?
  What happens when this function is interrupted by the OS?

```
unsigned char next(unsigned char current, unsigned char size)  {
        if (current+1 == size) return 0;
        else return (current+1);
}
```

it depends on where the parameters, automatics, and spill registers are stored… this one is probably okay!

3 places for parameters

a. Registers

b. fixed locations

c. stack…but not the hardware stack!

How about these?

- Is this reentrant?
  void disable(void) { ET0 = 0;}
  test for reentrancy: no matter how instructions from separate threads are interleaved, the outcome for all threads will be the same as if there were no other thread.
- Is this reentrant? … note: we don't care about order
  void setPriority(bit sHi) {PS = sHi; PT = ~sHi;}

Thread 1 (sHi = 0)	Thread 2 (sHi = 1)
PS 0	
	PS 1
	PT 0
PT <- 1	

- When do we need reentrancy in non-multithreaded programming?
- How is this normally managed?

Examples of Reentrant functions

```
int sum(tree) {
        if (!tree) return 0;
        return sum(tree->left) + sum(tree->right) + tree->val;
}
```

reason for reentrancy: re-use code

The key to reentrancy: relative addressing

Other examples of reentrancy:

two tasks share a function, ISR and task share a function

Reentrancy in Keil C51

- In C51, most parameter passing is done through registers (up to three parameters). Then fixed memory locations are used. Register method is reentrant, the other isn't.
- Local (automatic) variables in functions are also mapped to fixed memory locations (w/ overlaying)…definitely not reentrant.
- How can we solve this: declare functions to be reentrant as in:

```
unsigned char next(unsigned char current, unsigned char size) reentrant {
    if (current+1 == size) return 0;
    else return (current+1);
}
```

- BUT…the stack used for reentrant functions is NOT the same as the hardware stack used for return address, and ISR/TASK context switching. There is a separate "reentrant" stack used for that, which is not protected by the TINY OS. It's a different region of memory, and a fixed memory location is used for the reentrant stack pointer. So this works for FULL and for recursion (no OS).
- Conclusion…you can have shared functions in TINY if you:
  - convince yourself that all parameters are passed through registers
  - convince yourself are no local variables that use fixed memory locations (compiler can allocate those to registers too)
  - be sure not not change HW settings non-atomically
  - or… you disable context switching in shared functions by disabling T0 interrupts
    - Think of shared functions as critical sections. Does this impact timing constraints or interrupt latency?

Implementation Example: Reentrant, Encapsulated Queue

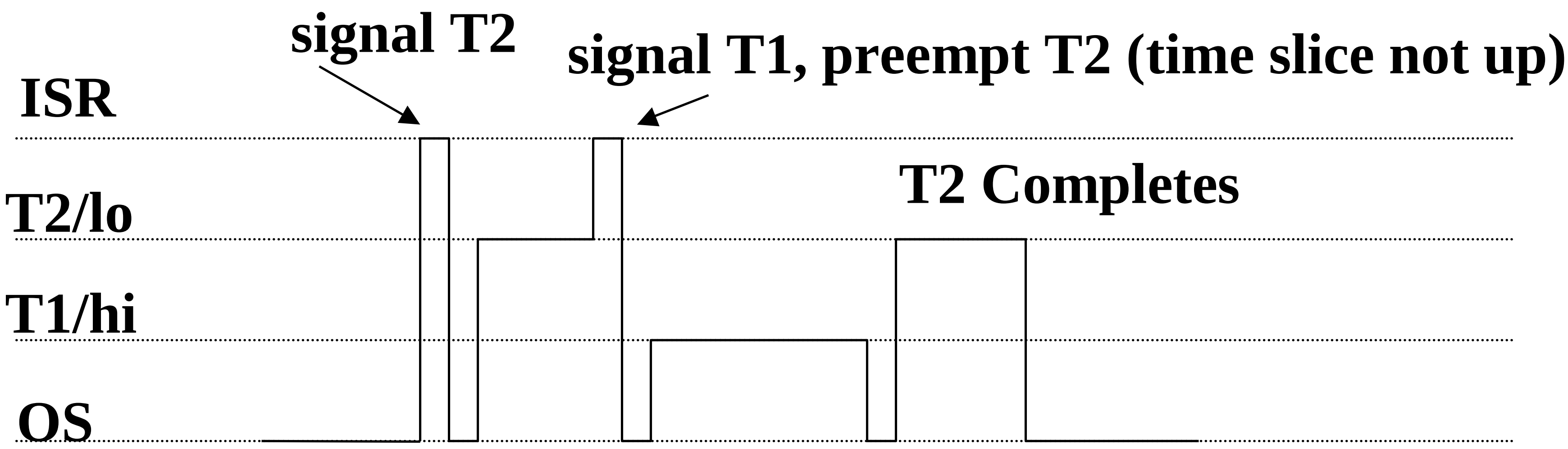
typedef struct qstruct {
unsigned char head;
unsigned char tail;
unsigned char *array;
unsigned char size;
} fifo;

Shared functions are okay if we disallow task switch during calls. why? re-entrant stack not protected by Tiny OS. But shared C libraries are okay. Why? not sure yet.

is this okay for timing if we don't use it in Tone Gen ISR (overhead)?

```
fifo Q;
unsigned char array[QSIZE];
void producer(void) _task_ 0 {
        unsigned char i;
        bit fail;
        initq(&Q, array, QSIZE);
        os_create_task(1);
        while (1) {
                do { disable();
                     fail = enq(&Q,i);
                     enable();
                }   while (fail);
                i++; // simulated data
        }
void consumer(void) _task_ 1 {
        bit fail;
        unsigned char i;
        while (1) {
            os_wait();
            disable();
             fail = deq(&Q,&i);
             enable();
             if (fail)…else use(I);
        }
}
```

Priority: Preemptive vs. Non preemptive



Pre-emptive: All tasks have a different priority…
- **hi priority task can preempt low priority task. Highest priority task always runs to completion (wait).**

Advantage: Lower latency, faster response for high priority tasks.

Disadvantage: Potential to starve a low priority task

Tiny: no priority, round robin only. No starvation.

<u>Priority Inversion:</u> when T2 disables interrupts

Coming Up

- A little more on OS
  - Real Time Scheduling Algorithms
  - Synchronization: Semaphores and Deadlock avoidance
  - Interprocess Communication
  - Concept of shared resources: Devices and Drivers
- Future
  - Linux and the Cerfboards
  - Networking
  - Product Safety
  - Java/Object Oriented Programming for Embedded Systems
- Design Meeting (Product Ideas…)