Got Python?

- python.org and the “cheese shop”
  - www.python.org
  - http://pypi.python.org/pypi
- Norm Matloff (UC-Davis)
  - heather.cs.ucdavis.edu/~matloff/python.html
- Dive into Python
  - diveintopython.net
- Lutz's “Programming Python”
  - 3/e uses Python 2.x   4/e uses Python 3.x
  - available from O'Reilly

Other built-in Python modules to investigate (1)

- math/cmath
  - higher-order math functions
- argparse
  - powerful command line option parser
- csv
  - “Comma-Separated Value” file reading and writing
- gzip/zipfile/tarfile/bz2
  - front-door to reading/writing compressed files
- thread/threading
  - low-level and high-level thread control/mgmt.
- os and os.*xxxx*
  - platform-independent functions (***VERY USEFUL!***)
- popen2
  - subprocess control/access/mgmt.

Other built-in Python modules to investigate (2)

- unittest
  - unit testing framework and test automation
- re
  - regular expression support (***VERY USEFUL!***)
- socket
  - module access to BSD sockets
- time
  - time-related functions and manipulations
- pdb
  - Python debugger (can be called programmatically)

3rd party Python resources

- ipython
  - feature-rich interactive Python shell
  - supports “parallel” python computing
  - I use this instead of “python”
- wxPython/pyQT
  - Python bindings for GUI widgets libraries
- Stani's Python editor (SPE) -- or -- Spyder
  - SPE: not-too-big IDE for Python – Spyder: everything IDE
- scipy
  - scientific numerical routines for Python
- others?
  - visit *http://www.scipy.org/Topical_Software*

ECE 4723/6723
Embedded Systems

Lecture

Embedded Systems Operating System
(ESOS)

Reading:

Why do we need an OS?

- Embedded systems are usually reactive
  - Lots of flow control, handshaking, and conditional execution
- Embedded systems usually do more than one thing “at a time”
  - Most practical embedded system “reactions” have more than one causal condition
    - read sensors and buttons, process HMI, control motors, flash LEDs, send telemetry over comm. links, etc.

You can write your own application execution framework.
But <u>WHY</u> would you <u>WANT</u> to ????

Why do we need an OS?

- Lots of choices for embedded OS designer
  - Real-time: soft- vs. hard- vs. non-?
  - Multitasking? Multiprocessing?
  - Threaded vs event-driven?
  - Memory protection? Memory mgmt? Virtual mem?
  - Provided services, libraries, etc.
- Many commericial and free OS-es available
  - Wind River/VxWorks, LynxOS, uCOS-ii/-iii, linux, android
  - Windows CE/Mobile/Phone/Phone7/Embedded/RT*
    - **or whatever Microsoft is calling it today

FACT: Any embedded system you will work with that has <u>any complexity to it whatsoever</u> will use an OS!

Multitasking contexts

Roll-your-own

```
while (1) {
    do audio functions 1;
    do GUI functions 1;
    do I/O functions 1;
    do audio functions 2;
    do GUI functions 2;
    do I/O functions 2;
        •
        •
        •
    do audio functions N;
    do GUI functions N;
    do I/O functions N;
}
```

OS

```
while (1) {
    call tasks periodically;
}
```

audio task

```
while (1) {
    do audio functions 1;
    do audio functions 2;
        •
        •
        •
    do audio functions Y;
}
```

I/O task

```
while (1) {
    do I/O functions 1;
    do I/O functions 2;
        •
        •
        •
    do I/O functions X;
}
```

GUI task

```
while (1) {
    do GUI functions 1;
    do GUI functions 2;
        •
        •
        •
    do GUI functions Z;
}
```

Multitasking

- Multitask programs, threads, or tasks
- Cooperative multitasking
  - task must choose to give up focus/control
  - Dangerous???
    - one task can kill everything
- Preemptive multitasking
  - some "higher" force can wrestle control away from your task
  - Safer??
    - theoretically, a watchdog <u>could</u> shut down errant task

Preemptive Multitasking

Preemptive Multitasking OS

```
OS task timer ISR(void) {
   save current task's state;
   determine "best" task to run next;
   restore new task state;
   resume execution in new task;
}
```

audio task

```
while (1) {
   copy compressed data from MP3 file;
   uncompress music data to a buffer;
   copy buffer to D/A converter;
   compute elapsed time values;
   while (D/A converter is busy);
}
```

I/O task

```
while (1) {
   read button/scroll wheel states;
   update status LEDs;
   delay (50ms);
}
```

GUI task

```
while (1) {
   update song title on screen;
   update artist/album on screen;
   while (elapsed time data is stale);
   update song's elapsed time;
}
```

Cooperative Multitasking

Cooperative Multitasking OS

```
while(1) {
   determine “best” task to run next;
   call that task;
}
```

audio task

```
while (1) {
   copy compressed data from MP3 file;
   uncompress music data to a buffer;
   copy buffer to D/A converter;
   compute elapsed time values;
   yield until D/A converter is ready for
         more data;
}
```

I/O task

```
while (1) {
   read button/scroll wheel states;
   update status LEDs;
   yield;
}
```

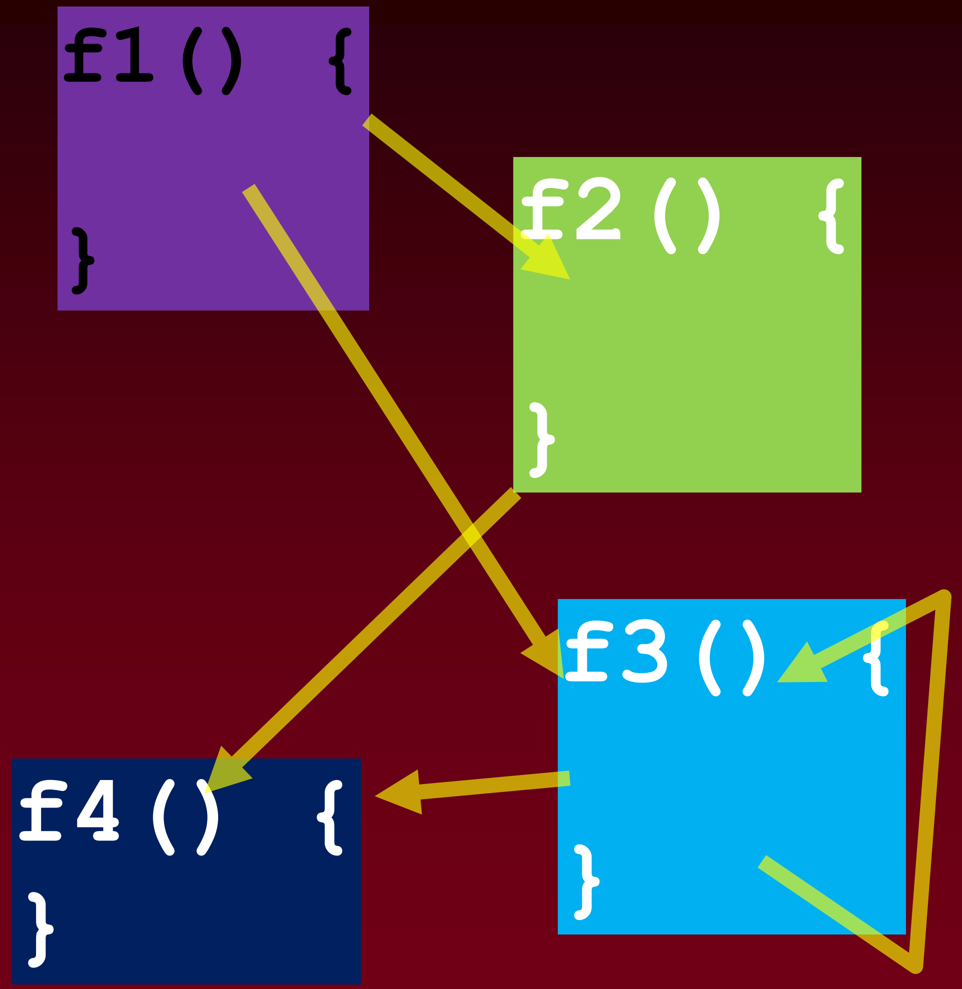
GUI task

```
while (1) {
   update song title on screen;
   update artist/album on screen;
   yield until elapsed time is updated;
   update song’s elapsed time;
}
```

Tasks vs. event-driven function

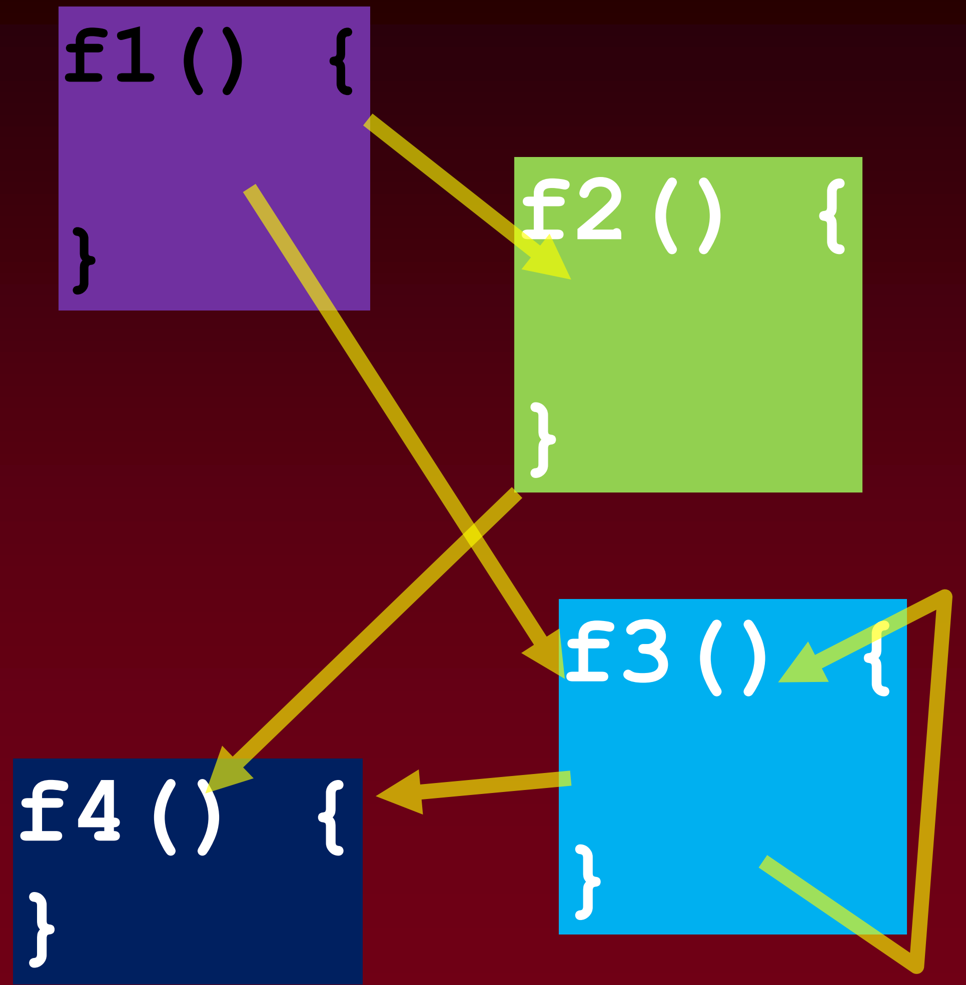
```
TaskX() {

  if() {
    YIELD();
  } else {
    YIELD();
    while() {
      YIELD();
    }
  }
}
```

where X is 1, 2, 3, 4



Event-driven: Problem with explicit state machine flow control

- State machine keeps track of flow control
- Control flow is difficult to follow (at best)
  - Flow is not evident from reading code
- Code is hard to write, understand, debug, and maintain



Event-driven programs

- Programs are not not sequential
    - code is often logically “muddled”
- Programs are event-handlers
    - An event-handler is a C function
    - “calling” function is not clear
    - An event-handler must **always** return;
    - An event-handler cannot block/wait for something to happen
        - Handler functions must be “atomic”

Background: Event-driven programming

- Why use event-driven programming?
  - Often used in memory-constrained systems
  - Threading libraries require too much memory and overhead
- “Backwards” to “normal” programming
  - Code is executed only when events occur
    - Incoming packets, sensor input, mouse clicks, time-outs, HMI
  - Nothing happens without an “event”
    - Programs have no “idle loop”
    - Events are called/triggered by some other entity (OS, GUI, etc.)

Example of event-driven code

```
void handle_sensor_input(int data) {
    if(check_sensor_data(data)) {
        send_radio_packet(data);
    }
    return;
}

void handle_incoming_packet(char *packet) {
    if(packet_should_be_forwarded(packet)) {
        send_radio_packet(packet);
    }
    return;
}
```

Threads/Tasks: No explicit state machine needed

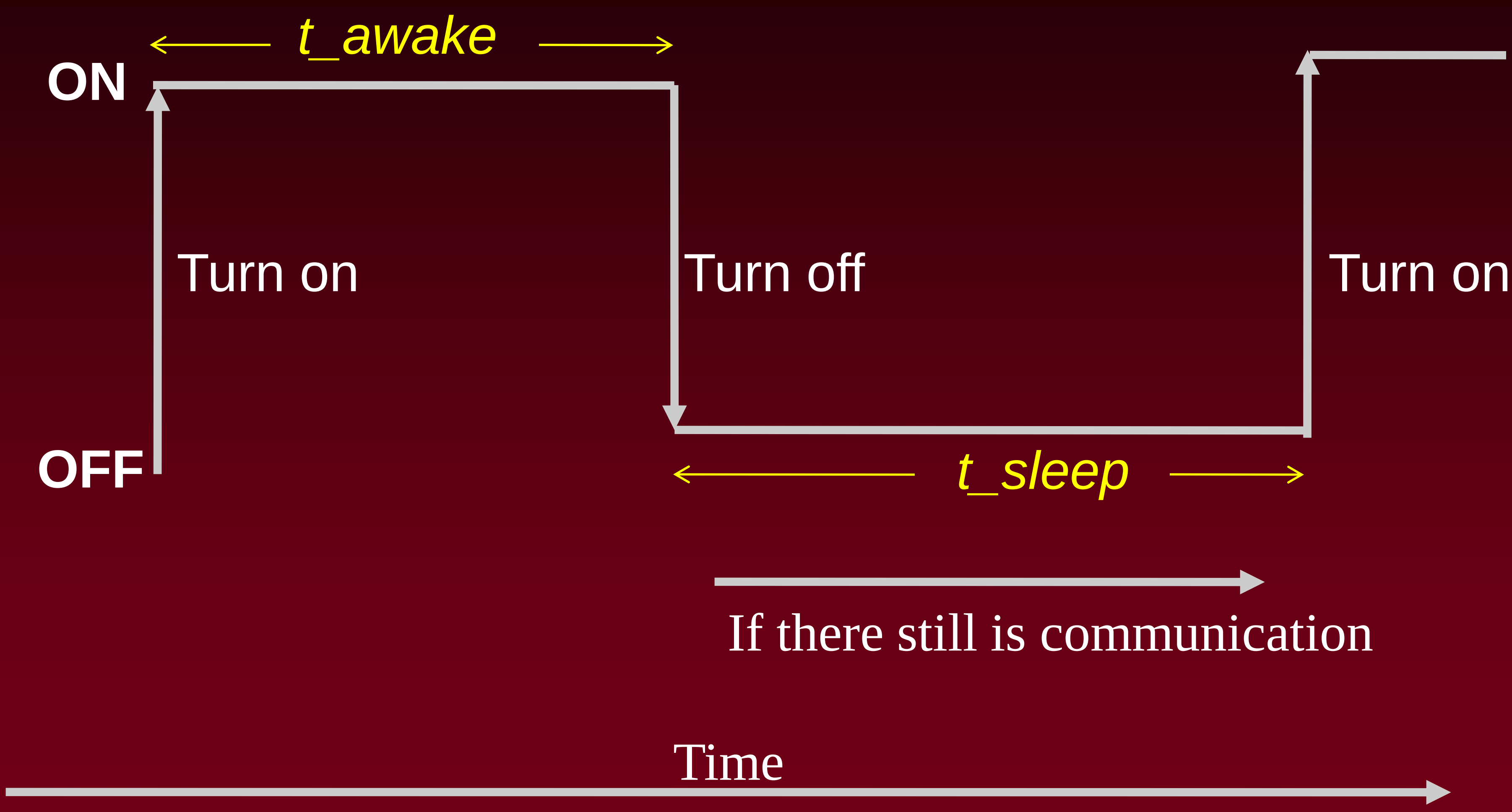
```
THREAD(f()) {

  if() {
    THREAD_WAIT_UNTIL();
  } else {
    THREAD_WAIT_UNTIL();
    while() {
      THREAD_WAIT_UNTIL();
    }
  }
}
```

- With threads, code ***looks*** "sequential"
- Flow control is apparent
- Code is easier to understand, debug, and maintain
- Easy to write once you start thinking "threaded"

The STACK!

- Threading
  - every thread requires a stack
  - stacks contain dead space
  - may use large amounts of the available memory
  - thread "manager" required
- Event-driven
  - only one stack is needed.
    - more efficient use of memory
  - Stack rewound on each event
  - Event-handlers are normal functions that explicitly return

Example: radio sleep cycle



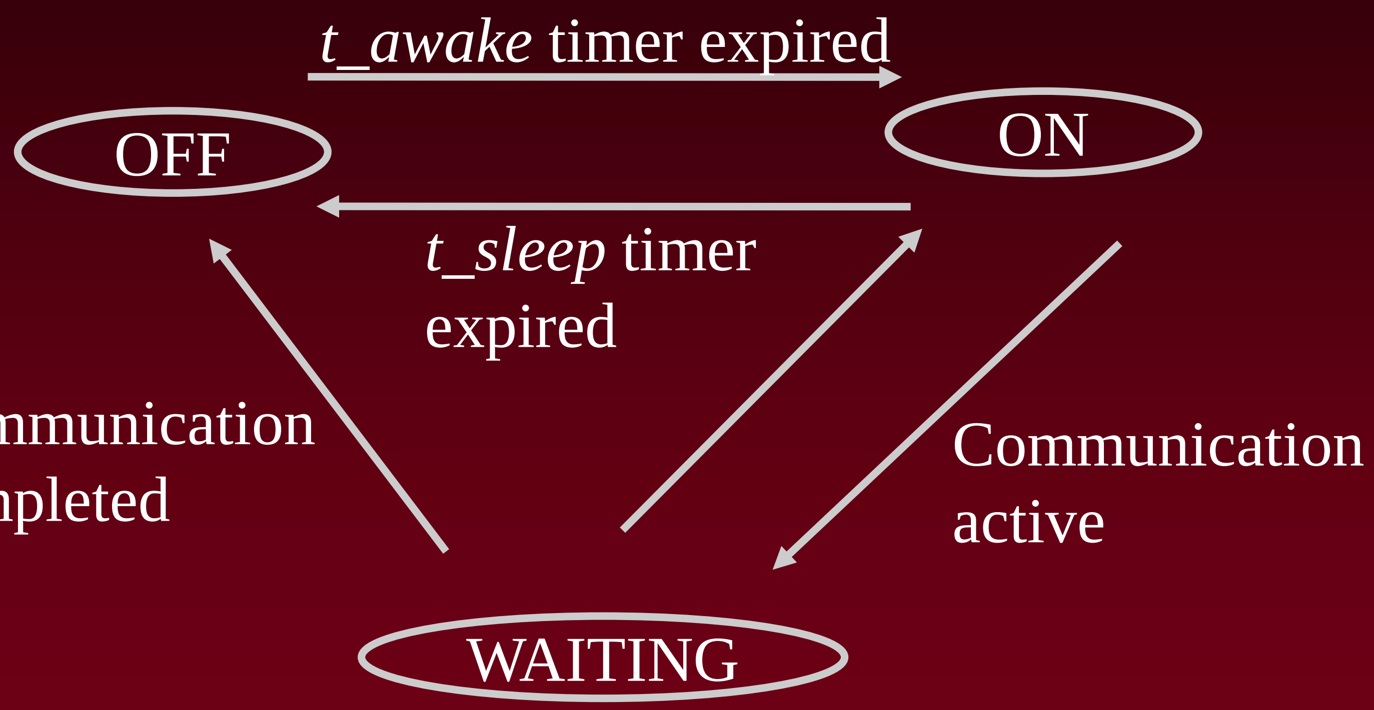
Example: radio sleep cycle
6-step informal specification

1. Turn radio on.
2. Wait for *t_awake* milliseconds.
3. Turn radio off, but only if all communication has completed.
4. If communication has not completed, wait until it has completed. Then turn off the radio.
5. Wait for *t_sleep* milliseconds. If the radio could not be turned off before *t_sleep* milliseconds because of remaining communication, do not turn the radio off at all.
6. Repeat from step 1.

Problem with events:
We can't write this as a 6-step program!

State machine implementation

With events, we must use an explicit state machine!



Our state machine coded in C

```
enum {
  ON,
  WAITING,
  OFF
} state;
```

Called by somebody...

```
void radio_wake_eventhandler() {
  switch(state) {
  case OFF:
    if(timer_expired(&timer)) {
       radio_on();
       state = ON;
       timer_set(&timer, T_AWAKE);
    }
    break;
  case ON:
    if(timer_expired(&timer)) {
       timer_set(&timer, T_SLEEP);
       if(!communication_complete()) {
          state = WAITING;
       } else {
          radio_off();
          state = OFF;
       }
    }
    break;
  case WAITING:
    if(communication_complete() ||  \
              timer_expired(&timer)) {
       state = ON;
       timer_set(&timer, T_AWAKE);
    } else {
       radio_off();
       state = OFF;
    }
    break;
  }
}
```

Quite complex!

And not even correct!

Have our cake and eat it too!

We want the memory efficiency of events

AND

the code clarity and maintainability of isolated threads/tasks!

Our cake: MSU's *Embedded Systems Operating System* (ESOS)

- ESOS "threading" is based on protothreads
  - ANSI C code from Adam Dunkels (*www.sics.se*)
- Protothreads are a mixture of the event-driven and "true" threads
  - stackless, non-preemptive threading
  - *can* be driven by an event-handler
  - With protothreads, we can write blocking waits, inside an event-handler
    - `ESOS_TASK_WAIT_UNTIL()` – conditional blocking
    - `ESOS_TASK_WAIT_WHILE()` – conditional blocking

ESOS implementation

```
ESOS_USER_TASK( radio_wake_thread ) {
  ESOS_TASK_BEGIN();
  while( TRUE ) {
 1  radio_on();
    timer_set(&timer, T_AWAKE);
 2  ESOS_TASK_WAIT_UNTIL(timer_expired(&timer));
    timer_set(&timer, T_SLEEP);
    if(!communication_complete()) {
3, 4  ESOS_TASK_WAIT_UNTIL(
        communication_complete() || timer_expired(&timer));
    }
    if(!timer_expired(&timer)) {
 5    radio_off();
      ESOS_TASK_WAIT_UNTIL( timer_expired(&timer) );
    }
 6}
  ESOS_TASK_END();
}
```

ESOS implementation

- Provides sequential code flow
  - The 6-step informal specification visible in the code
- Possible to use C control structures
  - if(), while(), for(), etc.
- Debug stack retained
  - Possible to follow calls
- Implicit blocking calls evident

Limitations in ESOS

- "Automatic" variables (stack variables) are **NOT** saved across blocking waits
  - Programmer must manually save automatic variables
  - However, **static** local variables still work as expected
- Standard C compiler may issue some warnings
- C language "switch" statements can not be used across "waits"
  - Not a big deal really
  - use *if-else if-else* construct instead (for short switch-es)
  - Refactor (for large switch statements)

ESOS thread functions (1)

- ESOS_TASK_WAIT_UNTIL(cond)
  - blocks current task *until* condition is TRUE
- ESOS_TASK_WAIT_WHILE(cond)
  - blocks current task *while* condition is TRUE
- ESOS_TASK_YIELD()
  - blocks current task until its next execution opportunity
- ESOS_TASK_RESTART()
  - blocks and re-inits the current task. Task will start executing at its beginning.
- ESOS_TASK_EXIT()
  - causes the current task to exit/end. If the ESOS task was spawned by a parent task, the parent task will become unblocked and resume execution

ESOS thread functions (2)

- ESOS_TASK_SLEEP()
  - current task goes to sleep and will be blocked until some other task wakes it up
- ESOS_TASK_WAKE(pstTask)
  - wakes up task pstTask. Task pstTask executes at next opportunity
- ESOS_TASK_SEM_INIT(s, v)
  - creates a semaphore s with initial value v
- ESOS_TASK_WAIT_SEMAPHORE(s, val)
  - Task will <u>*wait*</u> until semaphore s signaled val times
- ESOS_SIGNAL_SEMAPHORE(s, val)
  - signal (increment) the s semaphore val times

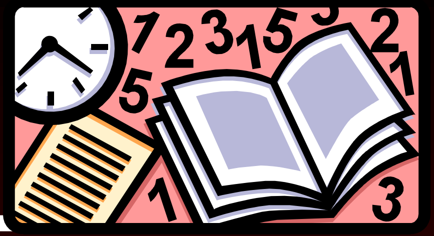
ESOS thread functions (3)

- ESOS_ALLOCATE_CHILD_TASK(thChild)
  - Allocates/inits the task handle thChild
- ESOS_TASK_SPAWN_AND_WAIT(thChild, pfnChild)
  - Current task waits on the completion (exit/death) of child task thChild that runs function pfnChild.

Next time, we will look at the

- ***structure of an ESOS application***
- ***other ESOS services***
  - ***communications***
  - ***interrupt control***
  - ***timer services***

References



New PIC24/dsPIC33 users:

- Read Chapters 1-9 in R/B/J

Read Chapter 14 in R/B/J

- *You may want to build a few of the examples from Chapter 14.*

Little Book of Semaphores

- *Reference 74 in R/B/J bibliography*

Adam Dunkel’s Protothreads

- *Reference 75 in R/B/J bibliography*

Read ECE4723 C language coding conventions

Ganssle Chapts. 1-3

- *If you bought this nice little book*