Today

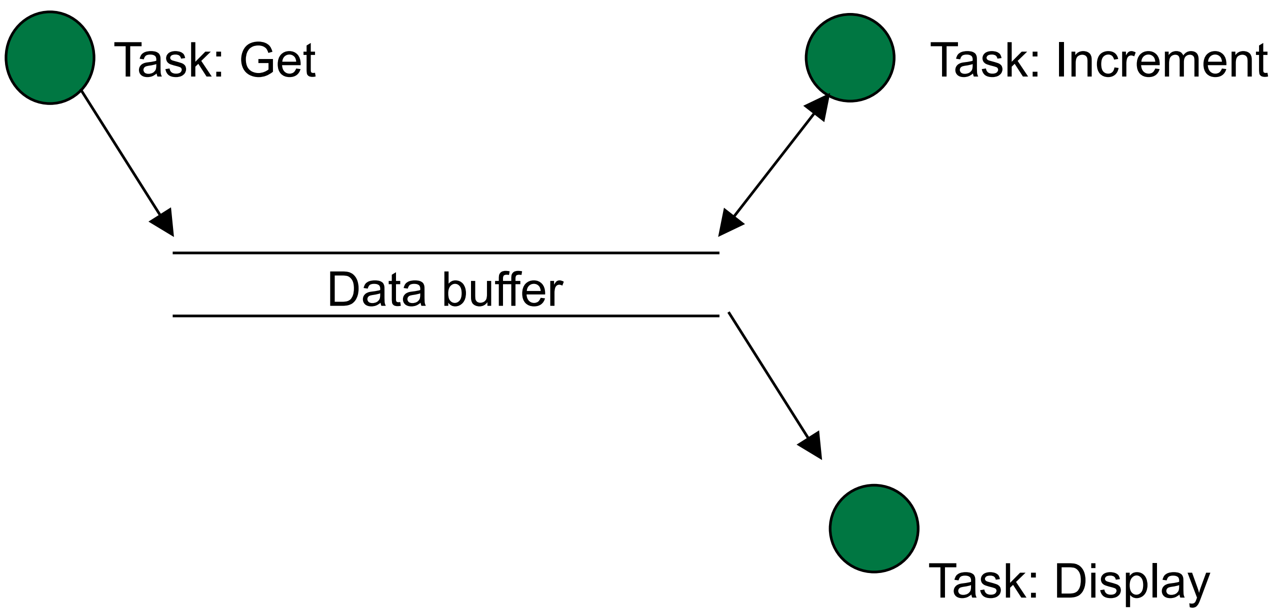
- Discuss final project
- Task Control Blocks / Simple kernel
- FSM Implementation
- Wednesday
  - Lecture: Safety
  - Course Evaluation

Final Project – Blimps!

- Groups of 4 – Choose wisely
- Tasks
  - Wireless radio link
    - Control
    - Telemetry
    - PC-side GUI
  - Motor Control
  - Sensing and control
  - Inter-blimp communication?
- Report format - Multimedia

Task Control Blocks:
Motivation – Organizing multiple tasks

- Simple “kernel” will be described
- Example application that simple kernel runs:
  3 tasks sharing a common data buffer



Based on content by James Peckol

Example 1: Simple “kernel” without TCBs

Header

```
// From James Peckol, Embedded Systems
// A simple OS kernel - step 1
#include <stdio.h>

// Prototypes for the tasks
void get (void* aNumber); // input task
void increment (void* aNumber); // computation task
void display (void* aNumber); // display task
```

Example 1: Simple "kernel" without TCBs

main() loop

```
void main(void) {
  int i=0;                  // queue index
  int data;                 // declare a shared var, data
  int* aPtr = &data;        // point to it

  void(*queue[3])(void*); // declare queue as an array of pointers to
                            // fns taking an arg of type void*
  queue[0]=get;             // enter the tasks into the queue
  queue[1]=increment;
  queue[2]=display;

  while(1) {
    queue[i]((void*)aPtr); // dispatch each task in turn
    i=(i+1)%3;
  }
  return;
}
```

Example 1: Simple “kernel” without TCBs

Task executables

```
void get(void* aNumber) {        // perform input operation
  printf("Enter a number, 0..9 ");
  *(int*) aNumber = getchar();
  getchar(); // discard CR
  *(int*) aNumber -= '0';        // convert to decimal from ASCII
  return;
}

void increment(void* aNumber) {// perform computation
  int* aPtr = (int*) aNumber;
  (*aPtr)++;
  return;
}

void display (void* aNumber) { // perform output operation
  printf("The result is: %d\n",*(int*)aNumber);
  return;
}
```

How can we make this more general?

- Organize tasks into Task Control Blocks (TCBs, a.k.a. Process Control Blocks)
- In an OS, a TCB/PCB defines a process
  - “The manifestation of a process in an operating system”
- Typically organized in a linked list in a full-blown OS scheduler
- Can be dynamically or statically allocated

Task Control Block - Contents

May contain:

- Process ID, priority info, process state
- Program counter, stack pointer, CPU registers
- Memory management info (main and virtual mem, shared data access)
- Scheduling info (timing requirements, time allocation)
- I/O status info (open files, resources)
- Processor status info
  - Clock frequency?
  - Power mode?

Why use TCB/PCBs?

- Necessary for OS-based embedded applications.
- Encourages good practices, even if not using full OS
  - Encapsulation of functionality
  - Well defined global data access
  - Easier to upgrade to OS

Task control block example

- Non preemptive
  - No management of SP or PC needed, no state save, etc.
  - Task functions run to completion and return
  - Task executables should be non-blocking
  - Resource access should be non-blocking
  - Shouldn’t fully utilize CPU

Task control block example

- Contains:
  - Structs with pointers to global data
  - Pointer to executable
- Simple “Scheduler”
  - Static task queue (typical for embedded)
  - Non-preemptive round robin
    - Simply run all tasks in-order, and repeat!

Example 2: Simple "kernel" with TCBs

Header

```
// From James Peckol, Embedded Systems
// A simple OS kernel - step 2
#include <stdio.h>

// Prototypes for the tasks
void get (void* aNumber);               // input task
void increment (void* aNumber);         // computation task
void display (void* aNumber);           // display task

// Declare a TCB structure
typedef struct {
  void* taskDataPtr;
  void (*taskPtr)(void*);
}
TCB;
```

Example 2: Simple “kernel” with TCBs

main() loop, part 1

```
void main(void) {
  int i=0;                    // queue index
  int data;                   // declare a shared var, data
  int* aPtr = &data;          // point to it
  TCB* queue[3];  // declare queue as an array of ptrs to TCBs

  // Declare some TCBs
  TCB inTask;
  TCB compTask;
  TCB outTask;
  TCB* aTCBPtr;

  // Initialize the TCBs
  inTask.taskDataPtr=(void*)&data;
  inTask.taskPtr=get;

  compTask.taskDataPtr=(void*)&data;
  compTask.taskPtr=increment;

  outTask.taskDataPtr=(void*)&data;
  outTask.taskPtr=display;
```

Example 2: Simple "kernel" with TCBs

main() loop, part 2

```
  // initialize the task queue
  queue[0]= &inTask;
  queue[1]= &compTask;
  queue[2]= &outTask;

  // schedule and dispatch the tasks
  while(1) {
    aTCBPtr=queue[i];
    aTCBPtr->taskPtr( (aTCBPtr->taskDataPtr) );
    i=(i+1)%3;
  }
  return;
}
```

Example 2: Simple "kernel" with TCBs

Task executables [unchanged from step 1]

```
void get(void* aNumber) {      // perform input operation
  printf("Enter a number, 0..9 ");
  *(int*) aNumber = getchar();
  getchar();                               // discard CR
  *(int*) aNumber -= '0';                  // convert to decimal from ASCII
  return;
}

void increment(void* aNumber) {           // perform computation
  int* aPtr = (int*) aNumber;
  (*aPtr)++;
  return;
}

void display (void* aNumber) {            // perform output operation
  printf("The result is: %d\n",*(int*)aNumber);
  return;
}
```

Realistic example

- Use task-specific data structs containing pointers to shared global data.
- Prototype on desktop with GCC or Visual Studio.
- Example: Defining TCB and task data structs

Other ideas

- Add CPU configuration info to TCB definition
  - Clock frequency
  - Power mode
- Use more complex non-preemptive scheduler, like Earliest Deadline First
- Use preemptive scheduler
  - However, context switching is expensive relative to complexity of typical embedded task… is it worth it?

Finite State Machines
Implementation Methods

- Switch-case
  - Simple, efficient implementation for small FSMs
  - Inherently mistake prone (when adding events or states)
  - Gets very ugly for larger FSMs
- State object representation
  - Easy to scale, even dynamically
  - Probably good for mid-size FSMs
- Tabular
  - Very scalable, great for larger FSMs
  - Efficient w.r.t. memory usage
  - A bit cumbersome to maintain

Finite State Machines

- State-centric switch-case
  - Switch(State) … { Switch(Event) … }
  - State transitions and actions occur in inner cases
- Event-centric switch-case
  - Switch(Event) … { Switch(State) … }
  - State transitions and actions occur in inner cases
- State object representation
  - State object/struct contains transition and action table
- Tabular
  - Action table, indexed by State and Event (or just State)
  - Transition table, indexed by State and Event