CSE466 Blimp Project – Phase One
AUTUMN 2012

OBJECTIVES

In this lab you will **DO** the following:

- Develop a task control block framework for robotic blimp development, and begin to implement tasks.
- Specify and implement an interface for wireless control of the blimp
- Implement a System Monitor task
- Develop a PC-side application for wireless control of the blimp

In this lab you will **LEARN** the following:

- Use of task control blocks for encapsulation and data sharing
- Details of the MSP430F5510
- The inner workings of the various blimp subsystems

NOTE: As you begin to implement portions of the blimp functionality, look for areas where you can encapsulate peripheral control functions into a low-level peripheral module driver and provide an API. For instance, replacing:

```c
P1DIR |= BIT0;
P1OUT |= BIT0;
```

With a call to something like:

```c
GPIO_Set(GPIO_1_0, HIGH);
```

DELIVERABLES

At the beginning of the next lab period you will demo your wireless controller, and describe/present the organizational structure of your firmware. **This demo will not be graded; the purpose of the demo will be to check in with the TA and keep your project on track.**

RESOURCES

These documents and web resources will be useful in completion of the lab and/or in answering the questions posed. Additional resources are described in the procedures below.

- BlimpBot schematics and component datasheets on course website
- Task control block example from lecture
- MSP430F5510 Product Page
IMPORTANT NOTES

There are several very important rules to follow when using the blimp hardware. If you do not follow these rules, there’s a good chance you’ll release the magic smoke and have to wait to have your hardware repaired.

1. **ALWAYS POWER ON THE BLIMP BATTERY SWITCH BEFORE CONNECTING THE DEBUGGER.** Never connect the debugger when the battery switch is OFF or when the battery is disconnected or dead.

2. **NEVER short the motor terminals.** The motor driver ICs and battery are very good at sourcing HUGE currents, and things will be damaged.

3. **ALWAYS plug in the battery with the proper polarity.** There is no protection against a reversed battery!

4. **ALWAYS keep the battery voltage above 3.4V.** When you notice the regulated voltage on VDD starting to drop below 3.3 V, it is time to charge the battery (Perhaps your firmware should watch for this “drop out” condition...)

5. **NEVER attempt to do anything to your LiPO battery without talking to the TA.** LiPOs are extremely dangerous. If your battery appears to be having problems, email the TA. Do not try to measure or modify your battery or this could happen.

PART 1: TASK CONTROL BLOCK FRAMEWORK

In this section you will set up the framework for task deployment and global data sharing. The resulting code should look something like the TCB example discussed in lecture. Try to make all data relevant to a particular task accessible via the task’s data struct.

The following tasks should be outlined. You will implement some of these now and some in future weeks, but you should start by getting a skeleton outline of all the TCB structs, data structs, and task functions set up so that implementation is just a matter of adding data and filling in the task function.

**Tasks:**

- **Initialization (Run once)**
  - Configures all peripherals and GPIO ports, system clock(s).

- **System Monitor**
  - Handles indicator lights based on system state
  - Implements a global RUN/STOP controller for the sensing and motor control tasks, which can be triggered from the wireless interface. This will be useful for development purposes.
  - Keeps the watchdog timer happy by clearing its counter
  - Forces system reset when instructed by wireless controller
  - Implements a fail-safe mode when communication is lost
  - Optional: Monitors MSP430 internal temperature sensor

- **Communication**
  - Receives, parses and buffers incoming messages
  - Formats and transmits outgoing messages
  - Interprets and acts on queries, control messages, etc
  - Optional: Implements ACK/NACK for incoming messages

- **Motor Control**
  - Interprets low-level motor control commands
  - Updates PWM channels for motor control

- **Sensing (Details TBD)**
  - Handles sensor polling
  - Processes/interprets sensor results

- **Motion Controller (Details TBD)**
  - Given high-level commands, implements a control algorithm for some form of motion control

For the purposes of Phase One, you should implement the **Initialization, System Monitor, and Communication** tasks.
PART 2: INITIALIZATION AND SYSTEM MONITOR TASKS

In this section you will implement the Initialization and System Monitor tasks.

The **Initialization task** responsibilities are:

- Set up the clocking module
  - Use 12MHz FLL setting
- Initialize peripheral devices
  - Timer modules
  - USCI
  - ADC
- Initialize GPIO ports
  - Pin direction
  - Pin function selection, port mapping
- Put sensor and motor hardware into safe state
- Initialize any data/control flow flags or signaling used by your tasks
- Configure the watchdog timer
  - Set up the timeout interval for the watchdog timer. Clear the timer and start it.

The **System Monitor task** responsibilities are:

- Keep track of system status:
  - Running / Stopped
    - The state of the RUN/STOP controller
  - Connected / Not connected
    - Whether or not the PC-side application is connected. Should make use of keepalive/ping signal.
- Use indicator lights to indicate system status
  - Light and/or flash the indicators based on the Running/Stopped and Connected/Not Connected status
- Implements a global RUN/STOP controller for the sensing and motor control tasks, which can be triggered from the wireless interface.
  - Watches for a RUN/STOP command from communication task
  - Puts motors and sensors into a safe power-down state when STOP command arrives and prevents further operation until RUN command received.
- Keeps the watchdog timer happy by clearing its counter
  - Make sure the WDT timeout interval is longer than the time required to cycle through the task queue in normal operation.
- Forces system reset when instructed by wireless controller
  - You can force a reset by attempting to write to the WDTCTL register without the proper password. This is not a hack, this is how TI says to do it!
- Implements a fail-safe mode when communication is lost
  - When not connected to the PC-side application, puts the motors and sensors in some type of fail-safe state.
    - Note: Turning them off may not always be the safest solution...
- Optional: Monitors MSP430 internal temperature sensor
  - Optional: Periodically read and store the value from the internal temp sensor

PART 3: WIRELESS CONTROL/TELEMETRY INTERFACE

In this section you will develop the wireless control and data telemetry interface for the blimp. On one side of the communication link will be a PC-side application that you will develop in Part 4. On the other side will be the blimp microcontroller.
The wireless interface will use the wireless UART relay developed in Lab 5, which you will need to interface with the USCI module on the blimp microcontroller.

The low level details of the communication protocol are up to you, but you must implement at least the following command types. Some commands include extra data in the payload field:

a. **Command category**
   i. **Command** (Optional payload)

**Command types (PC -> Blimp):**

b. **RUN/STOP controller signals**
   i. **RUN** (enable sensors and motors)
   ii. **STOP** (disable sensors and motors)

c. **Low-level motor control commands**
   i. **Set** {"Motor ID [0,1,2]", "Direction [Fwd,Rev]", "Duty Cycle [0-255]"}

d. **High-level motion controller commands**
   i. **Set** {One byte payload, meaning TBD}
   ii. **Tune** {Three byte payload, meaning TBD}
   iii. **Query motion controller status**

e. **Sensing**
   i. **Poll sensor** {One byte payload, meaning TBD}

f. **System Monitor**
   i. **Keepalive/ping**
   ii. **Query system status**
   iii. **Force reset**
   iv. Optional: Query core temperature

You will also need to implement a protocol for transmitting data back to the PC-side application controlling the blimp. The following telemetry data types are required:

**Telemetry data types (Blimp -> PC):**

a. **Motion Controller**
   v. **Response to motion controller status query** {One byte payload, meaning TBD}

b. **Sensing**
   vi. **Sensor result** {10 byte payload, meaning TBD}

c. **System Monitor**
   vii. **Keepalive/ping response**
   viii. **Response to system status query** {One byte state ID}
   ix. Optional: **Response to core temperature query** {Two byte ADC result}
In this section, and throughout the rest of the lab course, you will develop a PC-side application with accompanying GUI for control of your blimp. Buttons, sliders, or other input controls on your GUI will be able to initiate each of the control commands listed in Part 3, and indicators will show the current status of the blimp as well as sensor readings, etc.

Eventually you will add sensor polling and high-level commands to your application, but for this week you can start by implementing at least the following set of functionality:

- The Blimp <-> PC Communication framework
- Connected/not connected indicator (using keepalive/ping command and reply)
- System status indicator (displays result of system status query)
- RUN/STOP control

As in prior labs, you may use the language and development environment of your choice for this PC-side application. If you are lacking in ideas, visit the “Final Project Websites” page of the Autumn 2011 course website and look through the applications that previous students have built. Note that the project goals were slightly different, and so your GUI will not look exactly like theirs.