

# NET3001 Fall 09

---

## Assignment 6

**Due:** Dec 3, noon

**Submitting:** Use the submit.exe program. You may chose the names of the files that you submit, but the recommended names are:

assign6.c	motors.c	motors.h
lcd.c	lcd.h	
radiolib.h	radiolib.c	

In addition to the code files, you must submit drawings for *all* the tasks. Clearly mark each transition, label and describe each state, and describe what happens on the entry and exit from each state; label the diagram with the name of the state machine and your student number. [You will be expected to create drawings like this on the exam.] The transitions should be annotated with message *names* (not message *numbers*). Submit these drawings in .PDF format. Even if a task is extremely simple and does not use a state machine, submit a diagram.

Several requirements in this document are marked “optional”. These requirements can be omitted, and you will still be able to get full marks on the assignment. If you wish to implement them, only implement them *after* you have the base code functioning.

This assignment will implement a model of an electric wheelchair. You must write your program in the style of *cooperative multitasking*.

The wheelchair has the following hardware:

- a keyboard, with 12 keys, with functions assigned as shown below
- a 7 segment display, which shows the speed, and indicates charging
- a green LED, which is lit when a nearby wheelchair is detected
- a red LED, which flashes once a second, to indicate the passage of time, or to show time set
- a yellow LED which lights at low battery, and flashes at *very low* battery charge
- a 2-line LCD display, which shows the current status of the system, and warnings
- a main wheel drive motor (use the DC motor)
- a steering motor (use the stepper motor)
- a beeper, which is used to confirm keystrokes, warn of low battery, and warn of backing up
- a digital thermometer
- a serial link for maintenance
- a wireless link, to locate nearby chairs



The keypad on the MTS board will be used with this overlay:



actual size; cut it out if you wish

These keys are referenced in this assignment as “charge”, “forward”, “showCharge”, “left”, “stop”, “right”, “n/a”, “reverse”, “align”, “timeSet”, “time+” and “time-”

1	2	3
4	5	6
7	8	9
*	0	#

I suggest that you use the ASCII characters to code the keyboard. These codes match up with the codes issued by task 8 below, so if you start using them from the beginning, task 8 will be easier.

Write the software to make the system operate according to the following requirements:

---

### **Forward and Reverse Motion: (task 1)**

The user can press the “forward” (2), “reverse” (8) and “stop” (5) keys to operate the main drive motor. The chair has 3 forward speeds, and one reverse speed. Each press of the forward button increases the forward speed; if the user is already in reverse, the forward button changes the speed to 0, stopping the chair. When the chair is stationary, the reverse button causes the chair to back up; when the chair is in forward motion, each press of the reverse button reduces the speed one level.

In other words, there is a line of speeds like this:

reverse ↔ stopped ↔ forward1 ↔ forward2 ↔ forward3

The “forward” button moves to the right, the “reverse” button moves to the left.

If the user presses the “stop” button at *any time*, the chair comes to an immediate halt (jumps to the stopped state)

The motor can also be “locked”. This state prevents the keyboard from operating, and keeps the motor turned off.

Forward1 and reverse move the chair at 2km/h. Forward2 moves at 6km/h and forward3 moves at 10km/h.

Several other processes are interested in what state the chair is in, so this task should present global information to the system for:

- the current speed

This task should also issue messages for:

- chair is stopped
- chair is moving forward
- chair is backing up (reverse)

When the user begins to charge the battery (task 3 below), the charging task will issue a “lock” message. This task should listen for a “lock” and “unlock” message, and change to a state that ignores the forward and reverse buttons. When the battery capacity is reduced to 0, the charging task will also issue a “lock” message, and the main motor should shut off.

To model the wheelchair, operate the DC motor in a manner that matches the current speed. [Unfortunately, when the DC motor is operating, it interferes with the keyboard measurements. So I recommend that you simply operate the motor for a half second at the beginning of each state, and then turn it off.]

This task should also set the 7 segment display to the following:

- 0 when stopped
- 1,2,3 to indicate that the chair is in forward1, forward2 or forward3 movement
- 'b' to indicate reverse (backing up)

Other tasks may manipulate the display, but this should only happen when the chair is stationary. The other displays are:

- 'A' to show that the steering is being aligned
- 'C' to show that the chair is being charged

---

## ***Steering Task (task 2)***

The wheelchair has electronic steering, controlled by the left (4) and right (6) buttons. We model the steering by moving the arrowhead on the stepper motor, from left (pointing west) to right (pointing east); when the chair is traveling straight ahead, the arrow should be pointing directly up (north). You should move the stepper one step for each keypress, and limit the movement of the stepper to +/- 5 steps from the straight up position.

When the user presses the “stop” button, task 1 above will stop the main drive motor. This task (the steering task) should reset the steering back to straight-ahead. You will need a state for this, because the stepper motor may take some time to be moved back to vertical.

As in previous assignments, we need to set the stepper motor pointing in the correct direction. If the user presses the “align” key (9) while the chair is not moving, enter a state in which the user can press the left and right arrows (4 and 6 key) to spin the motor to the vertical position (without affecting the “direction” global). A second press of the “align” key should return the chair to normal operation.

During the alignment operation, put a 'A' on the 7 segment display.

This task should offer a global to the rest of the tasks:

- direction

---

## ***Battery Manager (task 3)***

This wheelchair has a large battery that can be filled and emptied. You can model the battery charge by using an integer between 0 and 10,000, where the large number indicates a battery that's fully charged.

The battery can be charged if the chair is not moving. The user should press the “charge” key (1) to start the charging process. The battery charges at 500 units/sec, to a maximum of 10,000 units. The user can then press the “charge” button again to stop the charging process. Display a 'C' on the 7 segment display during charging. Issue the “lock” and “unlock” messages (see below).

While the chair is operating, it drains the battery, according to the following rules:

- stationary, just the electronics operating: drains at 1 unit/sec
- moving forward, drains at 10, 30 and 50 units/sec
- moving in reverse, drains at 10 units/sec

This task should present some global information to the other tasks:

- charge in the battery

This task may issue some messages to the other tasks:

- battery is full (issued during charging)
- battery is usable (>20%, which is 2000 units, issued during charging)
- battery is low (<5%, which is 500 units, issued during discharge)
- battery is *very* low (<2%, which is 200 units, issued during discharge)
- lock the drive motor, charging is starting
- unlock the drive motor, charging has finished

---

## ***Display (task 4)***

The LCD display shows several kinds of data, and needs to be managed, so it has its own task.

The usual mode of operation is that the LCD displays:

- time and temperature
- battery state
- movement information

in a rotating cycle, 2 seconds for each display.

If the user pressed the “showCharge” key (3), the display locks into only displaying the battery stage. The user can release the display hold by pressing the same button again.

At any time, a task may signal that it wants a warning on the display, by sending a “display warning” message. The warning displays for 5 seconds and then the display returns to what it was doing, cycling through system information.

For some tricky operations, a task may request complete control of the LCD, by sending a “display grab” message. When it is finished, the other task will send a “display release” message.

### Details

Here are the details of what should go on the display:

Time and Temperature	“Time: 14:12” “Temp: 27°C”
Battery state	“Battery in use” or “Battery charge” “[      ]54%”
Movement information	“Speed: 2km/h” “Direction: -2”

When another task wishes to put a warning on the display, it should copy its message into a display buffer, and then post the “display warning” message.

In order to display the “degree” symbol on the LCD screen, you may use this format string in your code:

```
lcdSprintf(displayBuffer, "Temp: %d\xdf" "C", temperature);
```

The battery state display has a bar graph which illustrates the charge state *and* a percentage of the battery. You can calculate these number as follows:

```
percent = batteryCharge/100;  
numberOfBars = batteryCharge/1000;
```

See the appendix of this assignment for the code you should use for divide-by-ten. The calculations should actually be coded as :

```
percent = div10(div10(batteryCharge));  
numberOfBars = div10(percent);
```

You will have to create the bars in the string by writing code. For this display, the bar symbol is '\xFF'. The display for movement should indicate when the motor is locked.

This task presents one global to the other tasks:

- a display buffer, which is an array of bytes.

This task does not need to issue any message. It *does* need to respond to messages.

---

## **Sound Manager: (task 5)**

There are several requirements for sound in this projects, so we will need a sound manager.

The most common task for the sound manager is to beep a short tone (30msec) for each key that is pressed, to let the user know that the key was detected.

When the wheelchair is reversing, the sound manager must create a long slow backup tone.

When the system battery is low, the sound manager plays a slow chirp, and when it is *very* low, the chirp becomes more rapid.

When a task wishes to create a warning beep, the sound manager will play a long single tone.

Here is an outline of the tones:

key press	30msec single tone
reverse motion	700msec tone on, 300msec off, repeat
low battery	50msec on, 2.95sec off
very low battery	50msec on, 950msec off
warning beep	500msec tone

I suggest that you check for the reverse message *outside of any state code*. Yes, this is unusual, but the reverse message is the highest priority, and must be detected and handled no matter what state you are currently in.

The messages that the sound manager will watch for are:

- keyboard events
- reverse, stop
- low charge, very low charge, battery ok
- warning

The sound manager does not need to present any globals to the system. The sound manager does not need to issue any messages.

---

## Temperature (task 6)

Create a small task which wakes up once a second and performs an A/D conversion process. It sets a global variable which contains the temperature. The task then goes dormant again until one second has passed.

---

## Wireless Communication (task 7)

The wheelchair has a wireless module which will allow the user to detect when another wheelchair approaches.

The wireless task should send out a beacon every second, with a 8 character string with the name of the chair. Go ahead and use your first or last name, up to 8 characters long, filling it with spaces to make exactly 8 characters. Send this message out every second on channel 12345.

This task should listen for wireless signals, and when it gets one it should turn on the green light for 5 seconds, and send a message to the display task, asking it to display something like:

“Chair detected”

“Name: Scott”

Details of how to use the wireless library are in the appendix below. You can ignore messages back from send requests, all you need to act on is messages when a wireless signal is received.

---

## Maintenance Manager (task 8)

The serial port will be connected to a test jig, which allows the maintenance person to collect stats about the system, and inject keystrokes into the system.

Task 8 must listen to the serial port, and may receive one of the following ASCII characters.

'1'	charge	'2'	forward	'3'	showCharge	'R'	resetStats
'4'	left	'5'	stop	'6'	right		
'7'		'8'	reverse	'9'	align		
'*'	timeSet	'0'	time+	'#'	time-		

Each time a character is received, this task should respond with a single “.” to confirm.

Every 15 seconds, this task should send the current status, in the form of a string like this:

“S:10 D:3 T:25 C:8432”

followed by a carriage return (speed==10, direction==3, temperature==25, charge remaining=8432). This line is followed immediately by a line like this:

“K:141 M:31 m:26 U:340”

where these are the stats collected (Keycount, Maxtemp, mintemp, UmotorUsage).

## System Stats

This task simply keeps track of the usage of the system. It should keep a count of:

- the total number of keystrokes that the system received

- highest and lowest temperature that the system has encountered
- the total operating time of the main drive motor (in seconds)

Store these data in global variables called: tKeys, tMaxTemp, tMinTemp and tMotorUsage. You should be easily able to pick up messages which will allow you to collect this data.

When you receive a “resetStats” command on the serial port, set the key count and motor usage to 0, and the max temp to a small number, and the min temp to a big number.

## Serial Communication

The communications interface is designed to match the keyboard. When this task receives a character on the serial port, it should simply create a message and give it to the dispatcher. You will need extra messages to deal with the “resetStats” message, which is destined for task 8.

This task should check the serial port one or more times a second. If a character is available, you can call the serialGetch() function to fetch it. Here is some code you can use:

```
if (SCOSR1 & 0x20) {
    k = serialGetch();
    // ... do something with k
}
```

This task *does not* require a state diagram.

---

## Clock (task 9)

The clock task must keep track of the seconds, minutes and hours. For assignment 5, just start with hours=3 and minutes=23. Increment these as the system runs.

The clock task also flashes the red LED for 10 msec out of every 1 second.

For assignment 6, the user can set the real time in this system by pressing the “clock” key (\*). The clock task should then issue a “display grab” message to take over the LCD. Light the red LED steady while the user is setting the time.

The user is given a display of the current time. By pressing the “time+”/”time-” buttons, the user can set the hours. When the user presses the “clock” button, the system changes to setting the minutes. Again, the “time+” and “time-” buttons allow the user to scroll through the possible minutes. When the users presses “clock” a third time, the system returns to normal operation with the time changed to the new time. The red LED returns to a one second flash.

---

# Resources

## Event/Message Design

For timing, I recommend an EVT\_TICK, which happens 100 times per second. This is useful for moving the stepper motor, and timing the beeper.

You may want to consider a EVT\_ONE\_SEC, which happens only once per second. This makes it easy to count out the timer for the updating the clock, the display and a few other things.

## Serial Port

You will need to connect to the serial port. You can use this code to get started:

```
void initSerial(void) {
    UCA0CTL0 = 0;    // uart no prity, LSB first, 1 stop async
    UCA0CTL1 = 0xC0 + UCSWRST; // SMCLK, 1MHz, reset
    UCA0BR0 = 3;    // 19.2kbps
    UCA0BR1 = 0;
    UCA0MCTL = 0x41; // mod pattern 4, enable high-rate clock
    P3SEL |= 0x30;   // turn on uart pins
    P3REN |= 0x20;  // and pullup on the rx
    UCA0CTL1 &= ~UCSWRST; // release the reset, enable uart
}

/* get a char from the uart
 * or you may chose to use an interrupt for this
 */
int serialGetch(void) {
    if (IFG2 & UCA0RXIFG)
        return UCA0RXBUF;
    return 0; // for no char
}

/* send a char */
void serialPutch(char ch) {
    while (!(IFG2 & UCA0TXIFG))
        ;
    UCA0TXBUF = ch;
}

#define serialIsCharReady() (SC0SR1 & 0x20)
```

## Exact Timing

The boards you are using run at approximately 1 million cycles per second, but are accurate to only about +/- 25%. The boards have a calibration facility, which allows you to tune them to within 1% of absolute. Simply copy the two bytes shown into the clock control register:

```
BCSCTL1 = CALBC1_1MHZ;
DCOCTL = CALDCO_1MHZ;
```

## Measuring Temperature

You can measure the temperature with the following code:

```
dint();
ADC10CTL1 = INCH_3 | CONSEQ_0; // set to sample the temp probe
ADC10CTL0 |= ENC + ADC10SC;
while (ADC10CTL1 & ADC10BUSY)//
;
ADC10CTL0 &= ~ENC; // turn it back off
t = ADC10MEM;
eint();
```

Because the keyboard is running at the interrupt level, it may grab the ADC (analog to digital converter) at any time, and interfere a temperature measurement. To prevent this, you can wrap the temperature measurement in a `dint()/eint()` pair, which locks out the keyboard and the timer for a very short period of time.

This technique is called *critical section*.

Once you have a measurement in the variable 't', simply subtract 288 from this value. This is 10 times the real temperature in degrees C.

## Divide by Ten

The msp430 microcontroller does not have a divide operation, so you can use this subroutine instead:

```
/** divide by 10
 * based on multiplying the argument by binary 0.000110011001100...
 * @param n
 */
unsigned int div10(unsigned int n) {
    int a=0;
    while (n>999) {
        n-=1000;
        a+=100;
    }
    while (n>99) {
        n-=100;
        a+=10;
    }
    while (n>9) {
        n-=10;
        a++;
    }
    return a;
}
```

## Sound

All the sound in this project can be based on a 1KHz tone, which you can get by setting:

```
TACTL = TASSEL_2 + MC_1; // use SMCLK (same as main clock)
TACCR0 = 1000; // continuous count up to 1000
```

To start the tone, simply set

```
TACCTL0 = OUTMOD_4; // [0x80] tone on
```

And to stop the tone, set

```
TACCTL0 = OUTMOD_0; // [0] tone off
```

To make a click sound, you can

```
TACCTL0 ^= OUT; // [4] click the beeper
```

## Wireless Library

To include wireless function, include “radiolib.h”. To start the wireless functions, you must first call `radioInit()`.

Then you must call

```
radioRegisterEvents(newEvent, EVT_RADIO_SENT, EVT_RADIO_SENT_FAILED,
EVT_RADIO_RECV);
```

where the parameters are:

- the name of the subroutine which adds events to the dispatcher
- the event that you want to get when a radio signal is successfully sent
- the event that you want to get when a radio signal is not successfully sent
- the event that you want to get when a radio signal is received

After that, you can call

```
radioSetAddress(address);
```

to set the channel number for the radio.

You may call

```
radioSend(data, len);
```

to send a radio signal. `Data` is an array of chars and `len` is the size of it.

You may call

```
radioRecv(data, len);
```

to prepare your radio to receive a message. The `data` variable is an empty array, which will be filled in when the message is detected. You'll get a message through the dispatcher to tell you when a radio signal has been received, and the data will be waiting for you in the data array. The `len` variable *must* match the length of the sender, so for this assignment we'll always use 8 for the `len`.

Finally, when anything interesting happens in the radio, it will cause an interrupt on `PortB`. This is the same port as the keyboard, so the keyboard interrupt service routine that you used in assignment 5 will get called. You will need to modify your `kby_isr()` so that early in the `isr`, you check to see whether the interrupt came from the keyboard or the radio. If it came from the radio, you will need to call `radioInterrupt()` [in the `radiolib.c`] to handle the interrupt.

---

## General Notes

This is a complicated assignment.

1. Use the operating system framework that you built in lab 9.
2. Plan out what events & messages you will need.

3. Then design each task, starting with the state diagram. You have to hand this in anyway, you might as well use it.
4. Code up each task, and add it to the framework. Debug each task as it is added. If you need to create fake messages for testing, you can use one of the keys to generate the fake message, just until you get your system debugged.
5. If there are state transitions that are not described in the text above, add them in a way that you think is sensible.