UNIT-II

PART B

Basic Design Using a Real-Time Operating System Contents

Contents:

- Principles,
- Encapsulating Semaphores and queues,
- Hard real-time scheduling considerations,
- Saving memory and power,
- An example RTOS like µC-OS (open source)

I. PRINCIPLES:

- A very normal embedded system design technique is to have each of the RTOS tasks spend most of the time blocked, waiting for an interrupt routine or another task to send a message or cause an event or free a semaphore to tell the task that there is something for it to do.
- When an interrupt occurs, the interrupt routine uses the RTOS services to signal one or more of the task, each of which then does its work and each of which may then signal yet other task. In this way, each interrupt can create a cascade of signals and task activity.
- The following figure1 shows a very simplified version of some of what happens inside the telegraph system. In that figure the curvy arrows indicate message passed through the RTOS.
- When the system receives a network frame, the hardware interrupts. The interrupts routine reset the hardware and then passes a message containing the received frame to the DDP (datagram delivery protocol) protocol task.
- The DDP protocol task was blocked waiting for a message. When this message arrives, the task wakes up and among many of other things, determines if the frame was intended for telegraph or received by telegraph by mistake.
- If the frame was intended for telegraph, the DDP protocol task sends a message containing the received frame to the ADSP (Apple Talk data Stream Protocol) protocol task.
- This message unblocks the ADSP protocol task, if the frame contains print data; the ADSP protocol task sends a message containing the data to the serial-port task, which sends a data to the serial port hardware and through it to the printer.
- Similarly, when the system receives serial data from the printer, the interrupt routine resets the hardware and forwards the data in a message to the serial port task.

- If the data contains printer status, the serial port task forward the status to the ADSP, the ADSP uses it when respond to later status request from the network.

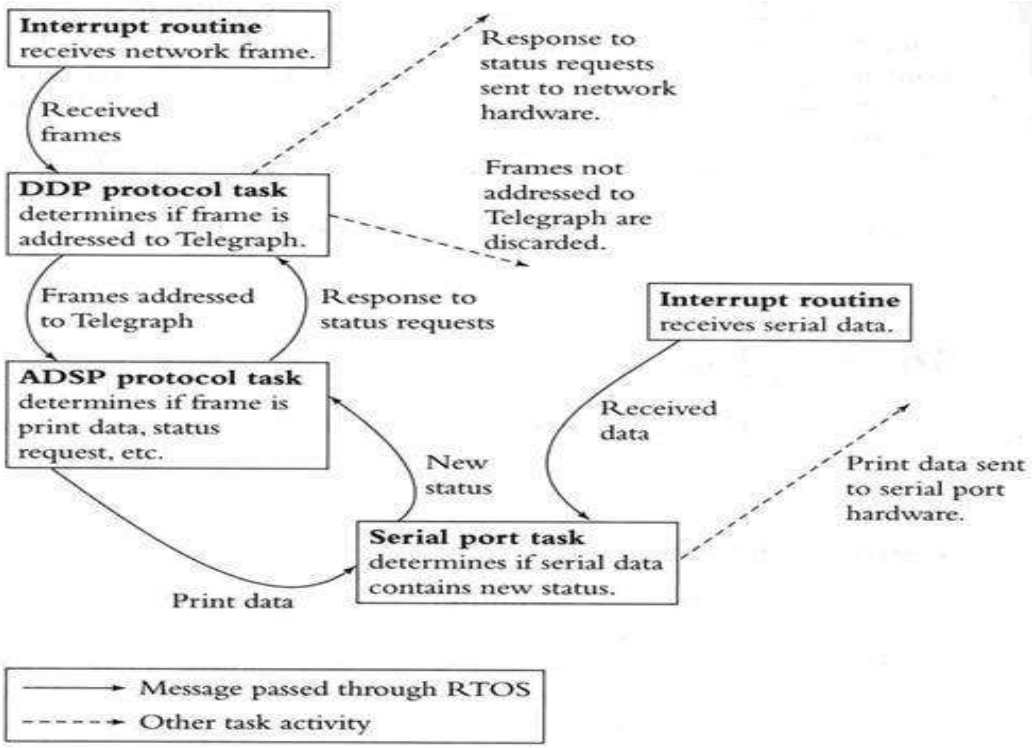


Figure 1: Telegraph Operation showing principles of design of RTOS

Best principles in design of an RTOS:

1. Write short interrupt routines
2. Decomposing a problem into tasks - How many tasks
3. Assigning task priorities
4. Encapsulation of tasks
5. Use of Task structure
6. Avoid creating and destroying tasks
7. Turning off the time slicing

1. Write short interrupt routines:

- Short interrupt routines are preferable than long interrupt routines. Interrupt routine code requiring immediate /quick response.
- There are two reasons for this.

1. Even lowest priority IR's are handled before the highest priority task code (minimize task code response time), writing a longer interrupt routines translates directly into slower task code response.

2. IR's are error prone and hard to debug than task code (due to hardware-dependent software parts)

Suppose that we are writing the software for the system with the following characteristics:

- A system must responds to commands coming from a serial port
- All commands end with a carriage-return (CR)
- Commands arrive one at a time; the next command will not arrive until the system responds to the previous one.
- Serial port hardware can only store one received character at a time, and characters arrive quickly.
- The system can respond to commands relatively slow.

  - ✓ One wretched way to write this system is to do all of the work in the interrupt routine that receives characters.
  - ✓ That interrupt routine will be long and complex and difficult to debug and it will be slow response for every operation.
  - ✓ At the opposite extreme you could write this system with an entirely brainless interrupt routine that simply forwards every character in an RTOS as message to command parsing task.
  - ✓ One possible compromise design uses an interrupt routine that saves the received characters in a buffer and watches for the carriage return that ends each command.
  - ✓ When the carriage return arrives, the interrupt routine sends a single message to the command parsing task, which reads the characters out of the buffer.

2. Decomposition Problem into Tasks – How many tasks:

- One of the first problem in an embedded system design is to divide your system's work into the RTOS tasks. This will make to have a large number of tasks.
- There are some advantages and disadvantages of using a large number of tasks.

Advantages:

- More tasks offer better control of overall response time.
- With more tasks your system can be somewhat more modular.
- With more tasks you can sometimes encapsulate the data more effectively.

Disadvantages:

- With more tasks you are likely to have more data shared among two or more tasks. This will make the system to have more semaphores, and hence into more microprocessors time to handle semaphores and into more semaphore related bugs.

- With more tasks you are likely to have more requirements to inter task communication through pipes, mailboxes queues and so on. This will also translate more microprocessor time and more chances for bugs.
- Each task requires a stack; therefore more tasks probably need more memory, at least for stack space.
- Each time the RTOS switches tasks. It takes more microprocessor time for saving the context of the task that is stopping and restoring the context of the task about to run.

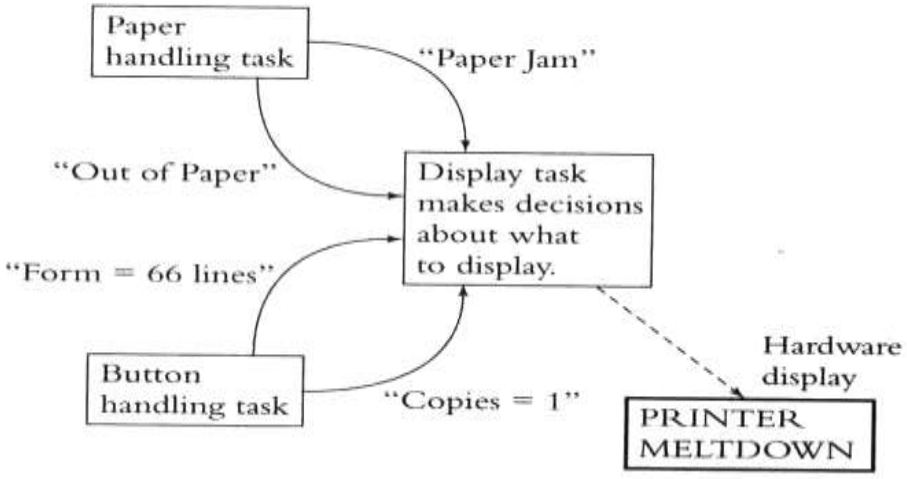
3. Task Priorities:

- Decomposing based on 'functionality' and 'time criticality,' separates ES components into tasks (naturally), task prioritization will give the quicker response – high priority for time-critical ones, and low priority for others.

4. Tasks for encapsulation:

- It makes sense to have a separate task to deal with hardware shared by different parts of the system.
- A single task that controls the hardware display can solve these problems.
- When other task in the system has information to display, they send messages to the display task.
- The RTOS will ensure that messages sent to the display task are queued properly; Simple logic in the display task can then decide which message should be placed on the display.
- The below figure shows that.

Figure 8.3 A Separate Task Helps Control Shared Hardware



5. Recommended Task Structure:

- The task structure will give the overall information about private data and its life time. And how many number of times that will execute, and from which task it reads data and to which any number of other tasks it write.
- A task declares its own private data.

Figure 8.5 Recommended Task Structure

```
vtaska.c

// Private static data is declared here

void vTaskA (void)
{
    // More private data declared here, either static
    // or on the stack

    // Initialization code, if needed.

    while (FOREVER)
    {
        // Wait for a system signal (event, queue message, etc.)

        switch (//type of signal)
        {
            case // signal type 1:
                :
                break;

            case // signal type 2:

                :
                break;
            :
        }
    }
}
```

Advantages of the task structure:

- The task blocks in only one place. When another task puts a request on its task queue; this task is not in the off waiting for some other event that may or may not happen in a timely fashion.
- When there is nothing to for this task to do, its input queue will be empty, and the task will block and use up on microprocessor time.
- This task does not have public data that other tasks can share, so there is no shared data, and there are no semaphores and semaphore problems.

6. Avoid creation and destroying tasks:

- Every RTOS allows you to create tasks as the system is starting.
- Most RTOSs also allow us to create and destroy tasks while system is running.
- There are two good reasons to avoid this:
- The functions that create and destroy tasks are typically the most time consuming functions in the RTOS, than getting a semaphore.
- Creating a task is relatively reliable operation, it is difficult to destroy a task without leaving little pieces lying around to cause bugs

7. Turning time- slicing off:

- The situation in which, if two or more ready tasks have the same priority and no other ready task has a high priority.
- One option that most other task offer in this situation is to **time slice** among those tasks.

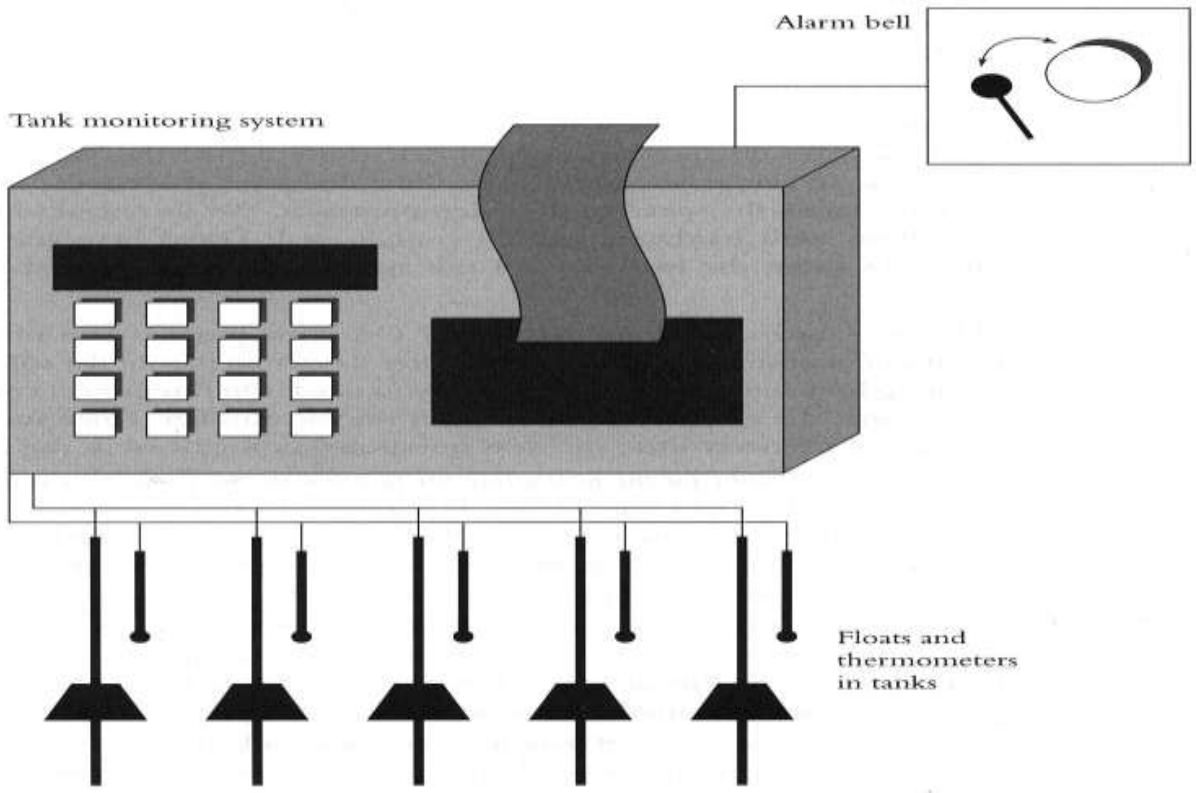
- Giving the microprocessor to one of the tasks for a short period of time, and then switching the microprocessor to the other task for a small period of time and so on.
- One might want to consider whether they want to have two tasks with the priority or whether they could just as well be one task. RTOS allows turning off this option.

An Example: Underground Tank monitoring System

Summary of Problem Specification:

- System of 8 underground tanks
- Measures read:
    1. Temperature of gas (thermometer) read at any time
    2. Float levels (float hardware) interrupted periodically by the microprocessor
- Calculate the number of gallons per tank using both measures
- Set an alarm on leaking tank (when level slowly and consistently falls over time)
- Set an alarm on overflow (level rising slowly close to full-level)
- User interface: a) 16-button control panel, LCD, thermal printer
- System can override user display options and show warning messages

Figure 8.7 Tank Monitoring System



System Decomposition for Tasks:

- One **low priority task** that handles all gallons calculations and detects leaks as well (for all tanks – 1 at a time)
- A high priority **overflow-detection task** (higher than a leak-detection task)
- A high priority float-hardware task, using semaphores to make the level-calc and overflow-detection task wait on it for reading (semaphores will be simpler, faster than queuing requests to read levels)
- A high priority **button handling tasks** – need a state-machine model (an IR? with internal static data structures, a simple wait on button signal, and an action which is predicated on sequence of button signals) since semaphores won't work.
- A high priority display task – to handle contention for LCD use
- [Turning the alarm bell on/off by the level-calc, overflow, and user button is typically non-contentious – an atomic op – hence do not need a separate alarm-bell task] However, need a module with BellOn (), BellOff() functions to encapsulate the alarm hardware.
- Low priority task to handle report formatting (one line at a time), and handle report queue
- (See below table Table 8.2)

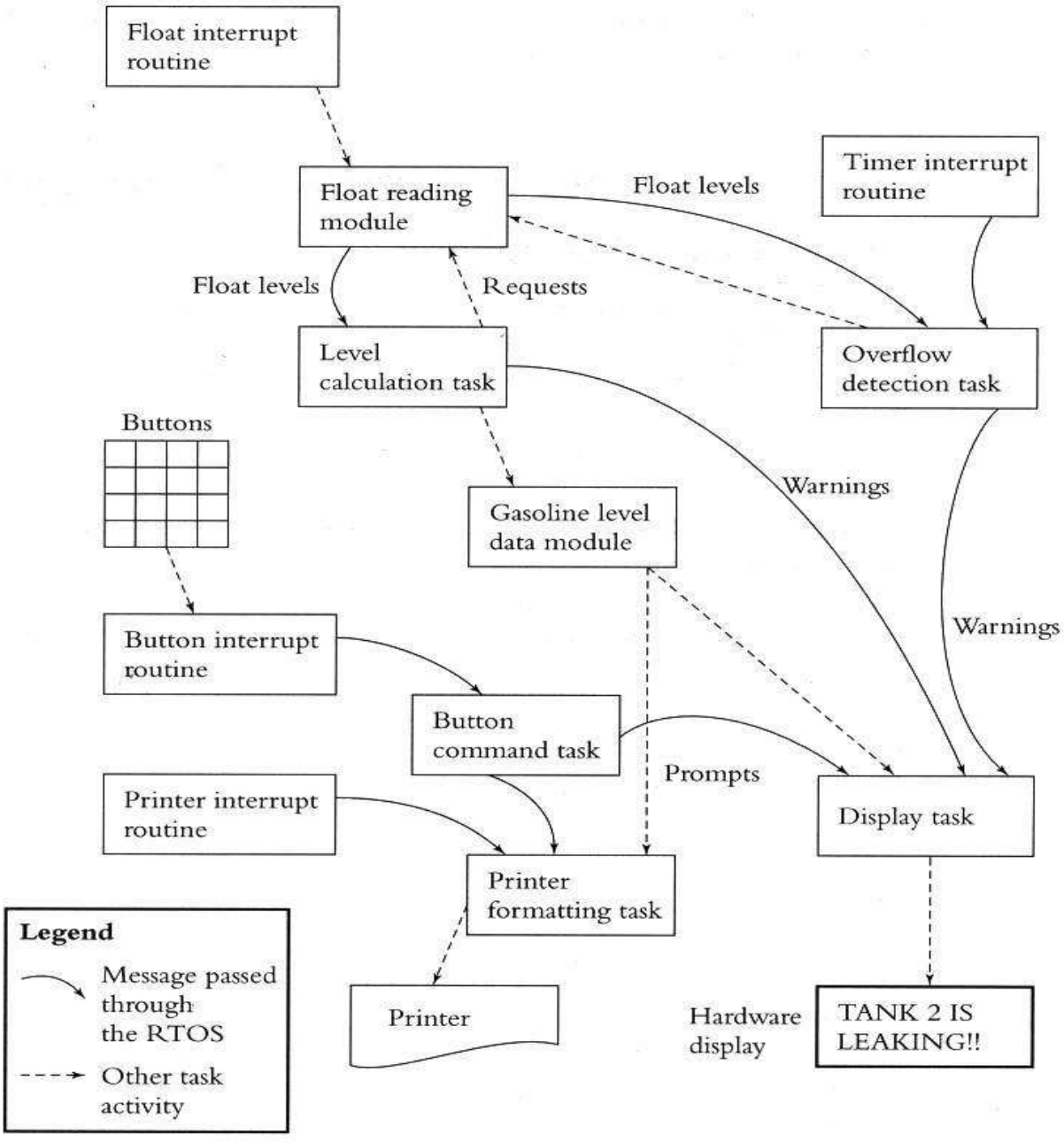
Table 8.2 Tasks in the Underground Tank System

Task	Priority	Reason for Creating This Task
Level calculation task	Low	Other processing is much higher priority than this calculation, and this calculation is a microprocessor hog.
Overflow detection task	High	This task determines whether there is an overflow; it is important that this task operate quickly.
Button handling task	High	This task controls the state machine that operates the user interface, relieving the button interrupt routine of that complication, but still responding quickly.
Display task	High	Since various other tasks use the display, this task makes sure that they do not fight over it.
Print formatting task	Medium	Print formatting might take long enough that it interferes with the required response to the buttons. Also, it may be simpler to handle the print queue in a separate task.

- Moving System Forward – Putting it together as Scenarios
    - System is interrupt driven via interrupt routines responding to signals, activating tasks to their work

- User presses button, button hardware interrupts the microprocessor, the button IR sends message to button-handling task to interpret command, which activates display task or printer task
- Timer interrupts, timer IR -> signal to Overflow-Detection task
- User presses printer button, print IR signals print-formatting task -> which sends first line to printer; printer interrupts for print IR to send next line to printer; when all lines (for report) are done, print IR signals print-formatting task for next report
- A level task need to read, it interrupts the level-read-hardware routine; the level is read by the hardware and the IR interrupts the task to read the new float level

Figure 8.9 Tank Monitoring Design



II. Encapsulating semaphores and queues:

Encapsulating Semaphores:

At least some of those bugs stem from undisciplined use. You can squash these bugs before they get crawling simply by hiding the semaphore and the data that it protects inside of a module.

```
void vTimerTask (void)
{
GetSemaphore <SEMAPHORE_TIME_OF_DAY);
++l SecondsToday;
if (lSecondsToday == 60 * 60 * 24)
l SecondsToday = 0L;
GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
}

long lSecondsSinceMidnight (void)
{
long lReturnValue;
GetSemaphore (SEMAPHORE_TIME_OF_DAY);
lReturnValue =lSecondsToday;
GiveSemaphore (SEMAPHORE_TIME_OF_DAY);
return (lReturnValue);
}

long lSecondsSinceMidnight (void);
void vHackerTask (void)
{
lDeadline = lSecondsSinceMidnight ()+ 1800L:
if (lSecondsSinceMidnight () > 3600*12)
}

long lSecondsSinceMidnight Cvoid;)

void vJuniorProgrammerTask (void)
{
long lTemp;
lTemp= lSecondsSinceMi dnight ();
for (l= lTemp; 1 < lTemp + 10; ++1)
}
```

- In the above code , rather than letting just any code that wants the value of the *lSecondsToday* variable read it directly and hopping for the best, this construction forces any code that wants to know the value of *lSecondsToday* to call *lSecondSinceMidnight* to get it. *lSecondSinceMidnight* uses the semaphore correctly. This semaphore will cause no more bugs.

Encapsulating queues:

- Similarly, you should consider encapsulating queues that task use to receive messages from other tasks. We should write code to handle a shared flash memory. That code deals correctly with synchronizing the requests for reading from and writing to flash memory. However, it would probably be a bad idea to ship that code in a real system.
- Consider this list of potential bugs.
- Even if everyone uses the correct structure, some body may assign a value to flash other than one of the two legal values.
- Anybody accidentally write a message intended for the flash task to the wrong queue.
- Any task might destroy the flash task input queue by mistake.
- The flash task sends data it read from the flash back through another queue. Someone might send an invalid queue ID.

III. HARD REAL TIME SCHEDULING CONSIDERATIONS

Considerations:

- A hard real time system is hard or software that must operate within the deadline, which comes from fast code. A hard RTOS is one which has predictable performance with no deadline miss.

Hard real time systems provide the following:

- Some critical code in assembly to meet real time fast.
- When hard real time tasks are running, all other interrupts of lower priority are disabled.
- Predictions of interrupt latencies and context switching latencies of the tasks.
- Pre-emption of higher priority task by lower priority task.

Characterizing real-time systems:

- Made of task n that execute periodically every T_n units of time
- Each task worst case execution time, C_n units of time and deadline of D_n
- Assume task switching time is 0 and non-blocking on semaphore
- Each task has priority P_n
- Question: $\mathbf{SC_n = S(D_n + J_n) < T_n}$,
  Where Jn is some variability in task's time
- Predicting C_n is very important, and depends on avoiding 'variability' in execution times for tasks, functions, access time of data structures/buffers, semaphore blocking – any operation that can't be done in the same time units on each execution/access

IV. SAVING MEMORY SPACE

- Here are a few ways to save code space some of these techniques have disadvantages. We should apply those only if they are needed to squeeze our code into our ROM.
- Make sure that two functions do not use same thing or does same operation.
- Check that our development tools aren't taking much time for development.
- Configure our RTOS to contain only those functions that are needed.
- We should always look at the assembly language listing created by our cross-compiler to see if certain of our C statement translate into huge number of instructions.
- Use 'static' variable instead of relying on stack variables.
- For an 8-bit processor, use **char** instead of **int** variable
- We can save a lot of space by writing our code in assembly language.

Saving Power:

- Most embedded system microprocessors have at least one power-saving mode.
- Software can typically put the microprocessor into one of these modes with a special instruction

The modes have names such as:

- ✓ Sleep mode
- ✓ Low-power mode
- ✓ Ideal mode
- ✓ Standby mode

- A very common power saving mode is one in which the microprocessor stops executing instructions, stop any built in peripherals, and stops its clock circuit. This saves a lot of power, but the drawback is that the only way to start the microprocessor is to restart it.
- Another typical power saving mode is one in which the microprocessor stops executing instructions but the on board peripheral continue to operate. Any interrupt starts the microprocessor up again, and the microprocessor will execute the corresponding interrupt routine and then resume the task code from the instruction that follows the one that put the microprocessor to sleep. This method saves less power than the above method.
- Another common method for saving power is to turn off the entire system and user turn it back on when it is needed.
- If we want to put our microprocessor in power saving mode, we should plan to write fast software. The faster your software finishes its work, the sooner it can put the microprocessor back into a power saving mode and stop using up the battery.

V. AN EXAMPLE µC-OS RTOS:

- µC-OS is a microcontroller operating system.
- µC-OS is a simple, pre-emptive, multitasking RTOS in combination with microprocessor and microcontroller.
- µC-OS provides basic support for
    - Semaphore
    - Queues
    - Mailboxes
- Other services that a µC-OS provides are:
    - Mutual exclusion semaphore semaphores(to reduce priority inversions)
    - Event flags
    - Task management
        - Create a task
        - Delete a task
        - Change priority of a task.
        - Suspend or resume a task
    - Invariable size memory blocks management.
    - Time and time management.