EC7010/INTRODUCTION TO EMBEDDED CONTROLLERS

UNIT IV MULTITASKING AND THE REAL-TIME OPERATING SYSTEM

The challenge of multitasking and real time, multitasking with sequential programming, State machines, Real time operating system, RTOS services, synchronization and messaging tools, CCS PIC C Compiler RTOS. Design example: Voltmeter with RS232 serial output.

Textbooks

1. Tim Wilmshurst, “Designing Embedded Systems with PIC Microcontrollers-Principles and Applications”, Imprint of Elsevier, 2007.

www.sathieshkumar.com/tutorials

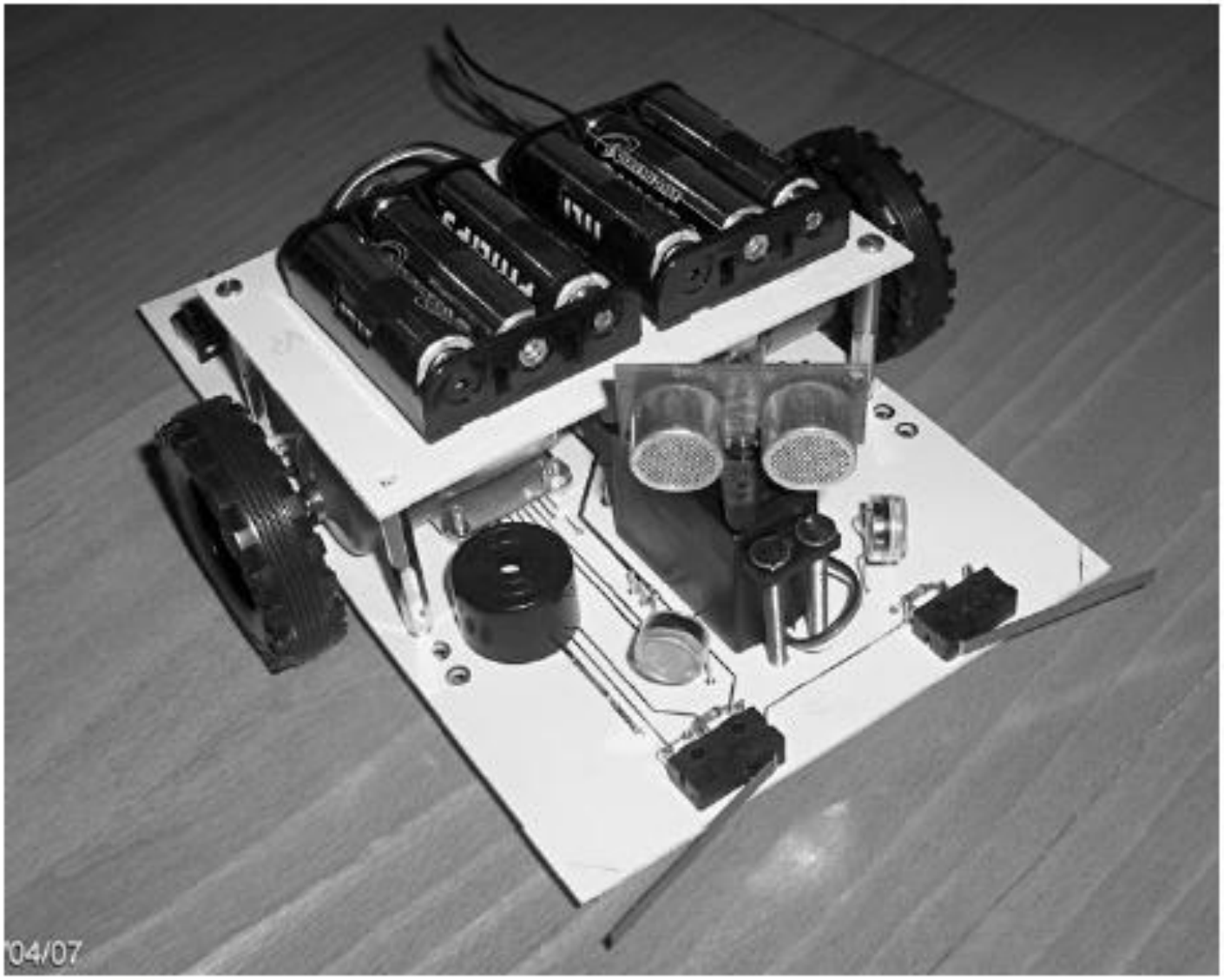
- Almost every embedded system has more than one activity that it needs to perform.
- As a system becomes more complicated, it becomes increasingly difficult to balance the needs of the different things it does.
- Each will compete for CPU time and may therefore cause delays in other areas of the system.
- The program needs a way of dividing its time ‘fairly’ between the different demands laid upon it.
- It’s no longer the program sequence which determines what happens next, but the operating system that is controlling it!
- It can be surrounded by many things, each demanding its attention. It will need to decide what to do first and what can be left till later.

The main ideas- the challenge of multitasking and real time:

Multitasking: The program is made up of a number of distinct activities. The practice of calling such activities **tasks**.

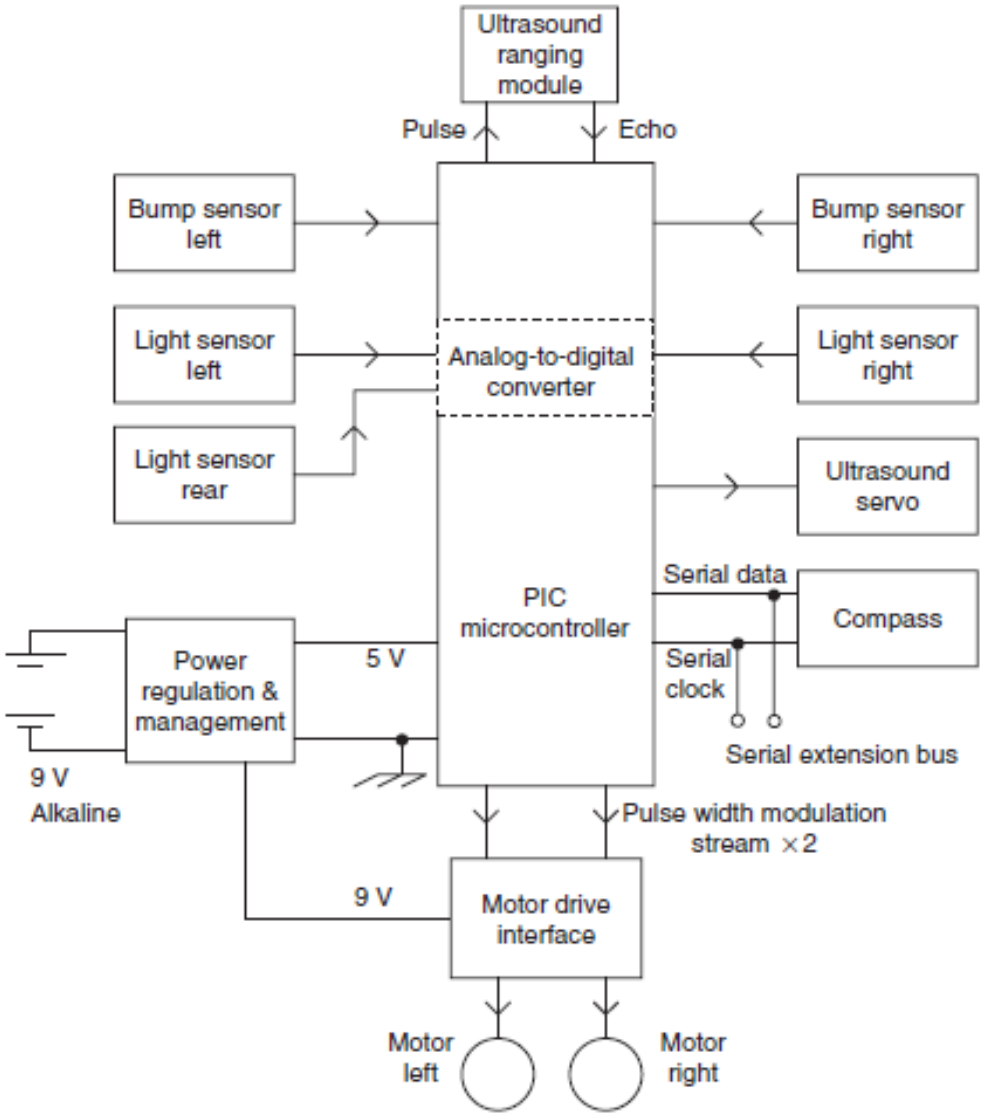
- A task is a program strand or section which has a clear and distinct purpose and outcome.
- Multi-tasking simply describes a situation where there are many tasks which need to be performed, ideally simultaneously. In reality, of course, the tasks are not all of equal importance. Therefore, we recognize that different tasks have different priorities.
- A high priority task should have the right to execute before a low priority task.
- Each task also has a deadline, or could have one estimated for it. The concept of priorities is linked to that of deadlines. Generally, a task with a tight deadline requires a high priority.

The main ideas- the challenge of multitasking and real time:



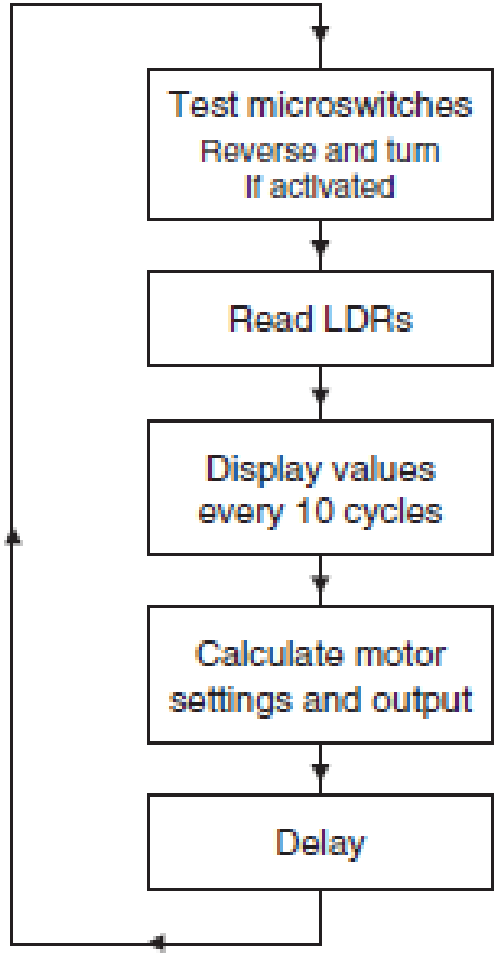
A Derbot AGV

The main ideas- the challenge of multitasking and real time:



The Derbot block diagram

The main ideas- the challenge of multitasking and real time:



Key: LDR = Light-dependent resistor

Simplified flow diagram of the Derbot light-seeking program

The main ideas- the challenge of multitasking and real time:

- A system operating in real time must be able to provide the correct results at the required time deadlines.
- This definition carries no implication that working in real time implies high speed, although this can often help.
- It simply states that what is needed must be ready at the time it is needed.

Table 18.1 Tasks in the Derbot light-seeking program

Task	Priority	Deadline (ms)
Respond to microswitches	1	20
Read LDRs	2	50
Calculate and set motor speed	2	50
Display	3	500

Achieving multitasking with sequential programming

- The type of programming we have engaged in to date, whether in Assembler or C, is sometimes called sequential programming.
- This simply implies that the program executes in the normal way, each instruction or statement following the one before, unless program branches, or subroutine or function calls, take place.

Achieving multitasking with sequential programming

Evaluating the super loop: Drawbacks

1. **Loop execution time is not constant**: The time it takes to run through the loop once is equal to the sum of the times taken by each task, plus the delay time.
2. **Tasks interfere with each other**: Once a task gets the chance to execute, it will keep the CPU busy until it has completed what it needs to do.
3. **High priority tasks don't get the attention they need**: In this continuous loop structure, every task has the same priority.

Achieving multitasking with sequential programming

Time-triggered and event-triggered tasks

- It is easy to recognize that some tasks are **time triggered** and others **event triggered.**
- A **time-triggered task** occurs on completion of a certain period of time and is usually periodic. An example of this is the reading of the LDRs in this program.
- **Event-triggered tasks** occur when a certain event takes place. In this case the pressing of a microswitch is a good example.

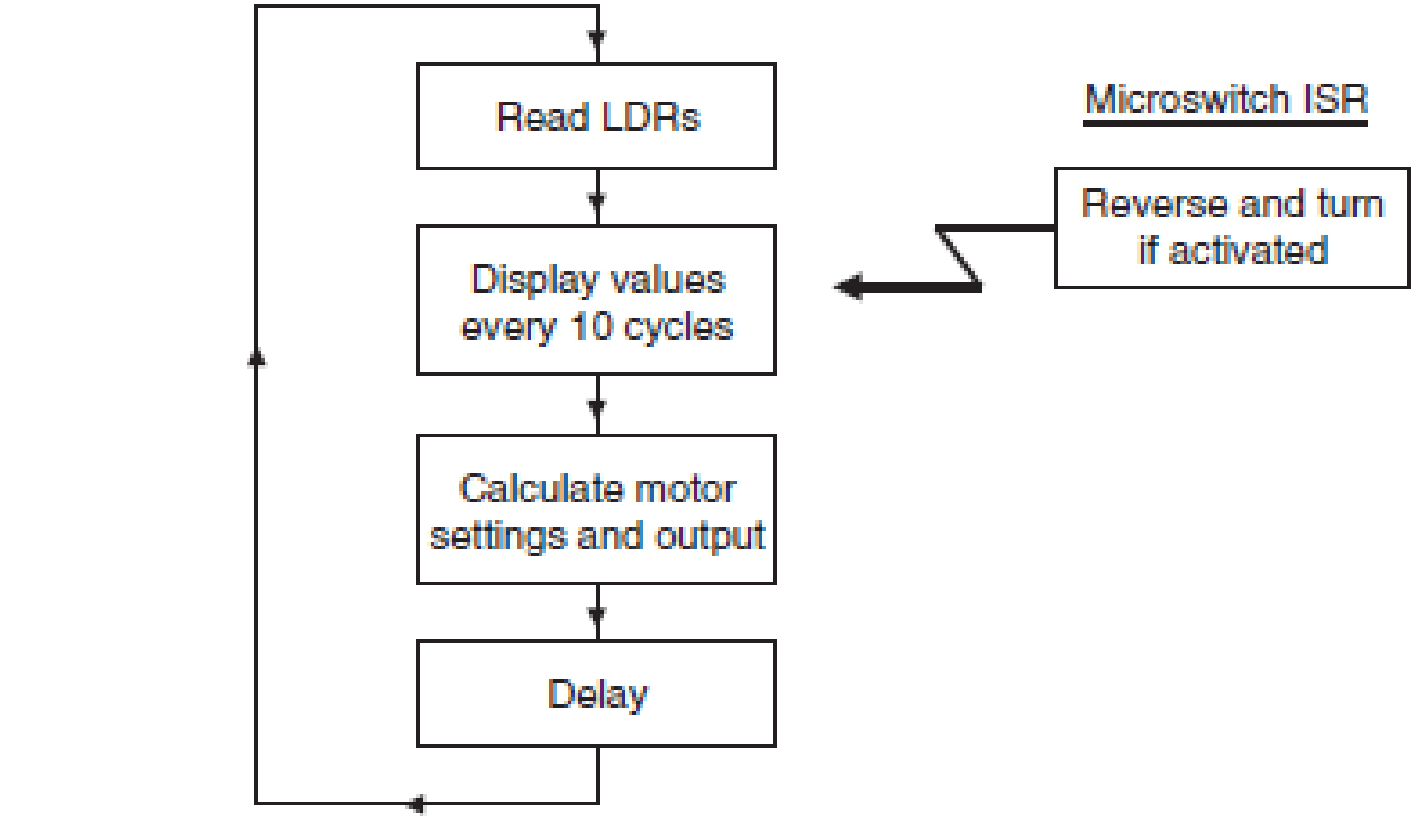
Achieving multitasking with sequential programming

Using interrupts for prioritization- the foreground/background structure

- To address the problem of lack of prioritization among the tasks in the light-seeking program, it would be possible to transfer high priority task(s) to interrupts.
- These would then gain CPU attention as soon as it was needed, particularly if only one interrupt was used.
- With this simple program structure given in the next slide, we have achieved a reliable repetition rate for tasks in the loop, and prioritization for tasks that need it. This is sometimes called a foreground/background program structure.
- Tasks with higher priority and driven by interrupts are in the foreground (when they need to be), while the lower priority tasks in the loop can run almost continuously in the background.

Achieving multitasking with sequential programming

Using interrupts for prioritization- the foreground/background structure



Using an interrupt for prioritisation in the Derbot light-seeking program

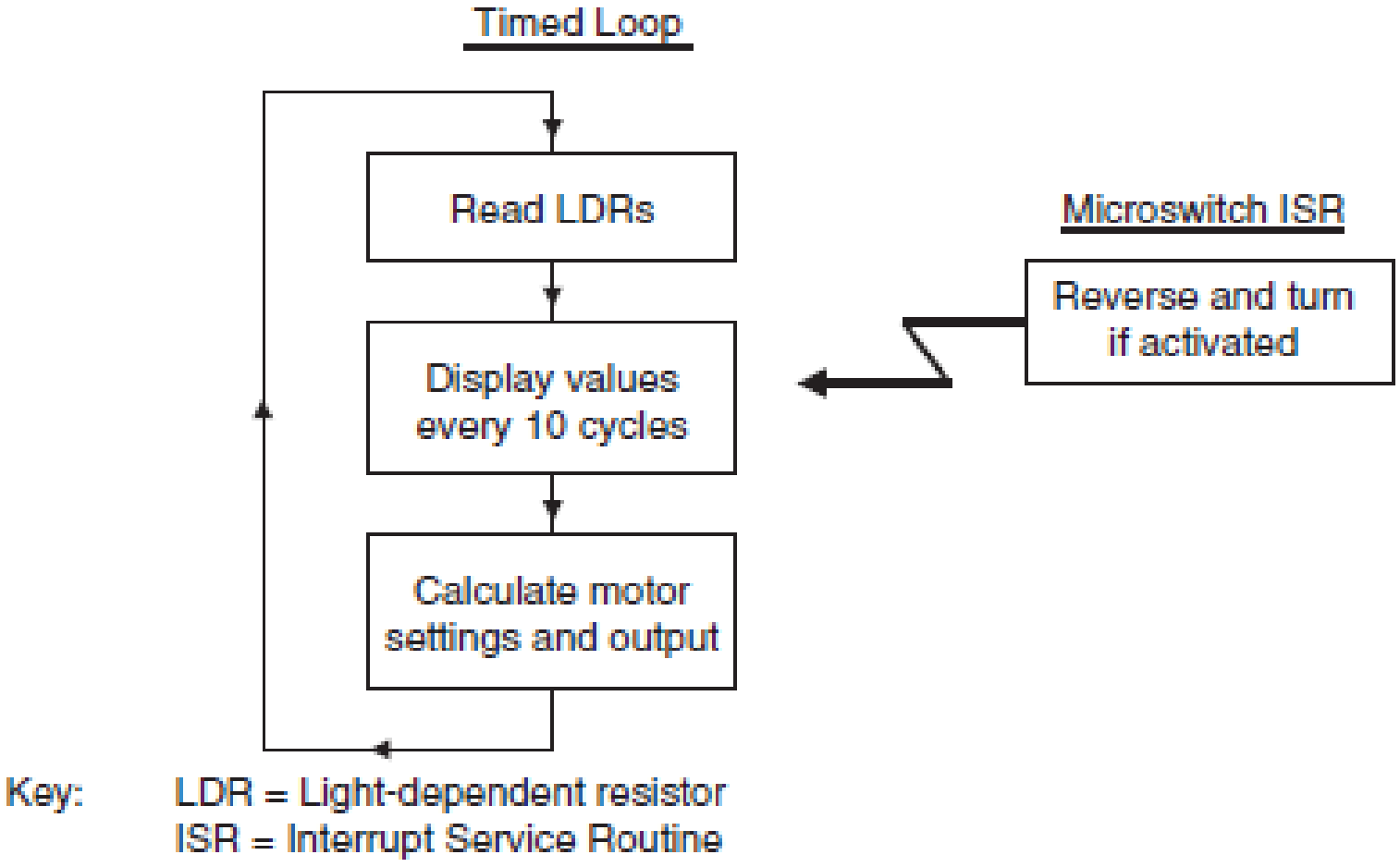
Achieving multitasking with sequential programming

Introducing a 'clock tick' to synchronize program activity

- To minimize the impact of the variable task execution times on the overall loop execution time, it is possible to trigger the whole loop from a timed interrupt, say from a **timer overflow**.
- The main loop is now triggered by a timer overflow, so occurs at a fixed and reliable rate.
- Time-triggered tasks can base their own activity on this repetition rate.
- Event-triggered tasks, through the interrupts, can occur when needed.
- Task timings, of course, have to be calculated and controlled, so that the loop has adequate time to execute within the time allowed and the event-triggered tasks do not disturb too much the repetitive nature of the loop timing.

Achieving multitasking with sequential programming

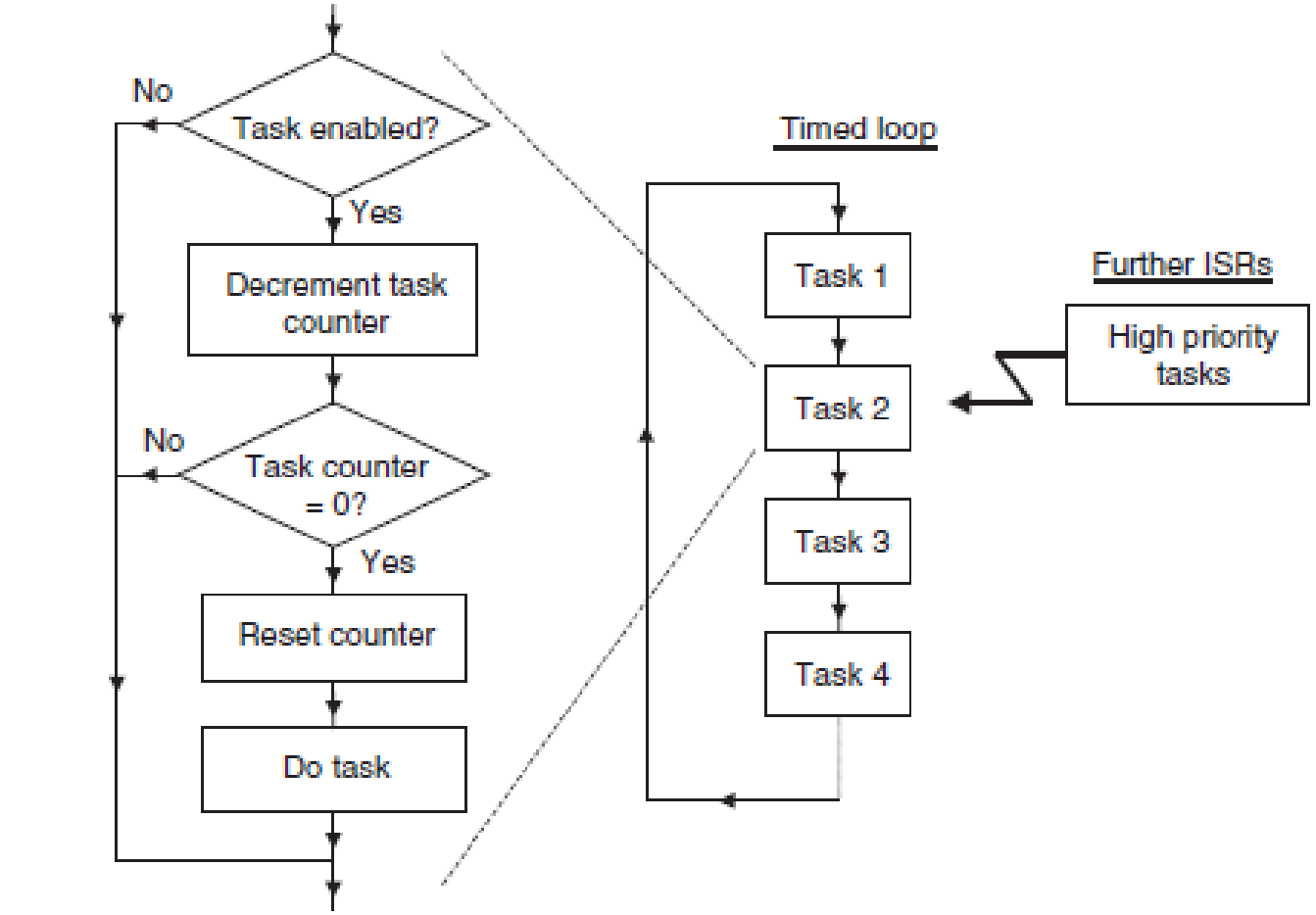
Introducing a 'clock tick' to synchronize program activity



Using a timed interrupt in the Derbot light-seeking program

Achieving multitasking with sequential programming

A general-purpose 'operating system'



A general-purpose 'operating system' structure, using sequential programming

Achieving multitasking with sequential programming

A general-purpose 'operating system'

- The main loop contains a series of low or medium priority tasks. It is driven by a '**clock tick**'.
- The general structure of each task is shown on the left. As needed, each task has an enable flag (a bit in a memory location) and each has a task counter.
- Tasks which need to execute every clock tick will do so. Many will only need to execute less frequently, at an interval set by the value of their task counter.
- Tasks can be enabled or disabled by the setting or clearing of their enable flag, by each other or by the ISRs.

Achieving multitasking with sequential programming

A general-purpose ‘operating system’

➢If several tasks are allocated to interrupts, then interrupt latency obviously suffers, as one ISR has to wait for another to complete. This must be analyzed carefully in a very time-sensitive system.

Achieving multitasking with sequential programming

The limits of sequential programming when multitasking

➢The approach to programming for multi-tasking just described will generally be acceptable, as long as:

1. There are not too many tasks
2. Task priorities can be accommodated in the structure
3. Tasks are moderately well behaved, for example their requirement for CPU time is always reasonable, and interrupt-driven tasks don't occur too often.

➢If these conditions are not met, it is necessary to consider a radically different programming strategy. The natural candidate is the **Real Time Operating System.**

The Real Time Operating System (RTOS)

- We hand over control of the CPU and all system resources to the operating system.
- It is the operating system which now determines which section of the program is to run and for how long, and how it accesses system resources.
- The application program itself is subservient to the operating system and is written in a way that recognizes the requirements of the operating system.
- Because we are concerned to meet real time needs, we make use of a particular type of operating system which meets this requirement, **the Real Time Operating System (RTOS).**

The Real Time Operating System (RTOS)

➢A program written for an RTOS is structured into tasks, usually (but not always) prioritized, which are controlled by the operating system.

➢The **RTOS** performs three main functions:

1. It decides which task should run and for how long
2. It provides communication and synchronization between tasks
3. It controls the use of resources shared between the tasks, for example memory and hardware peripherals.

The Real Time Operating System (RTOS)

- An **RTOS** itself is a general-purpose program utility.
- It is adapted for a particular application by writing tasks for it and by customizing it in other ways.
- While you can write your own RTOS, it is pretty much a specialist activity and generally done by specialists.
- There are a number of companies which develop and supply such operating systems, usually targeted towards one particular type of market and scale of processor.

Scheduling and the scheduler

- A central part of the RTOS is the scheduler. This determines which task is allowed to run at any particular moment.
- Among other things, the scheduler must be aware of what tasks are ready to run and their priorities (if any).
- There are a number of fundamentally different scheduling strategies, they are:

1. Cyclic scheduling
2. Round robin scheduling, context switching and Task states
3. Prioritized pre-emptive scheduling
4. Cooperative scheduling

- The role of interrupts in scheduling

Cyclic scheduling:

- Cyclic scheduling is simple.
- Each task is allowed to run to completion before it hands over to the next.
- A task cannot be discontinued as it runs. This is almost like the super loop operation.
- Cyclic scheduling carries the disadvantages of sequential programming in a loop.

Figure 18.5 Cyclic scheduling – Tasks 1, 2 and 3 execute in turn

Round robin scheduling and context switching:

- In round robin scheduling the operating system is driven by a regular interrupt (the 'clock tick').
- Tasks are selected in a fixed sequence for execution.
- On each clock tick, the current task is discontinued and the next is allowed to start execution.
- All tasks are treated as being of equal importance and wait in turn for their slot of CPU time.
- Tasks are not allowed to run to completion, but are pre-empted, i.e. their execution is discontinued mid-flight. This is an example of a **pre-emptive scheduler.**

Round robin scheduling and context switching:

➢The implications of this pre-emptive task switching, and its overheads, are not insignificant and must be taken into account.

➢When the task is allowed to run again, it must be able to pick up operation seamlessly, with no side-effect from the pre-emption. Therefore, complete **context saving (all flags, registers and other memory locations)** must be undertaken as the task switches.

➢ Time-critical program elements should not be interrupted, however, and this requirement will need to be written into the program.

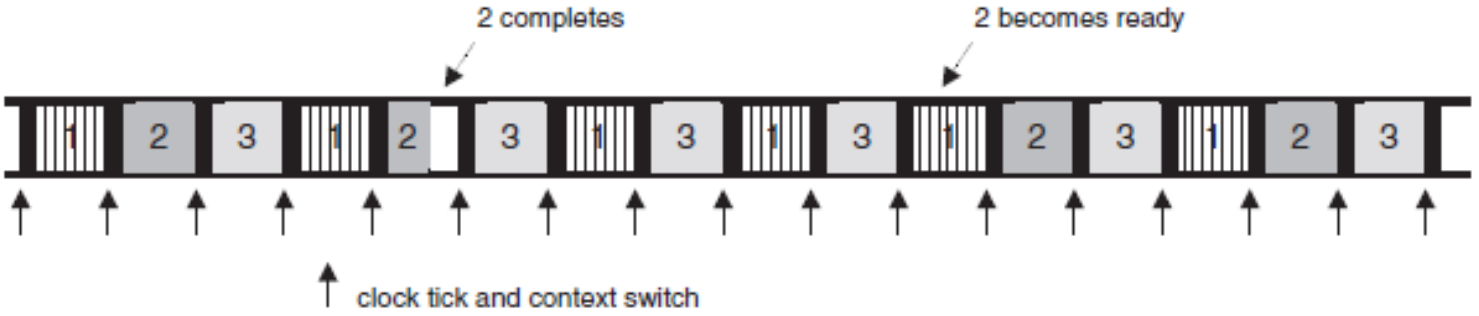
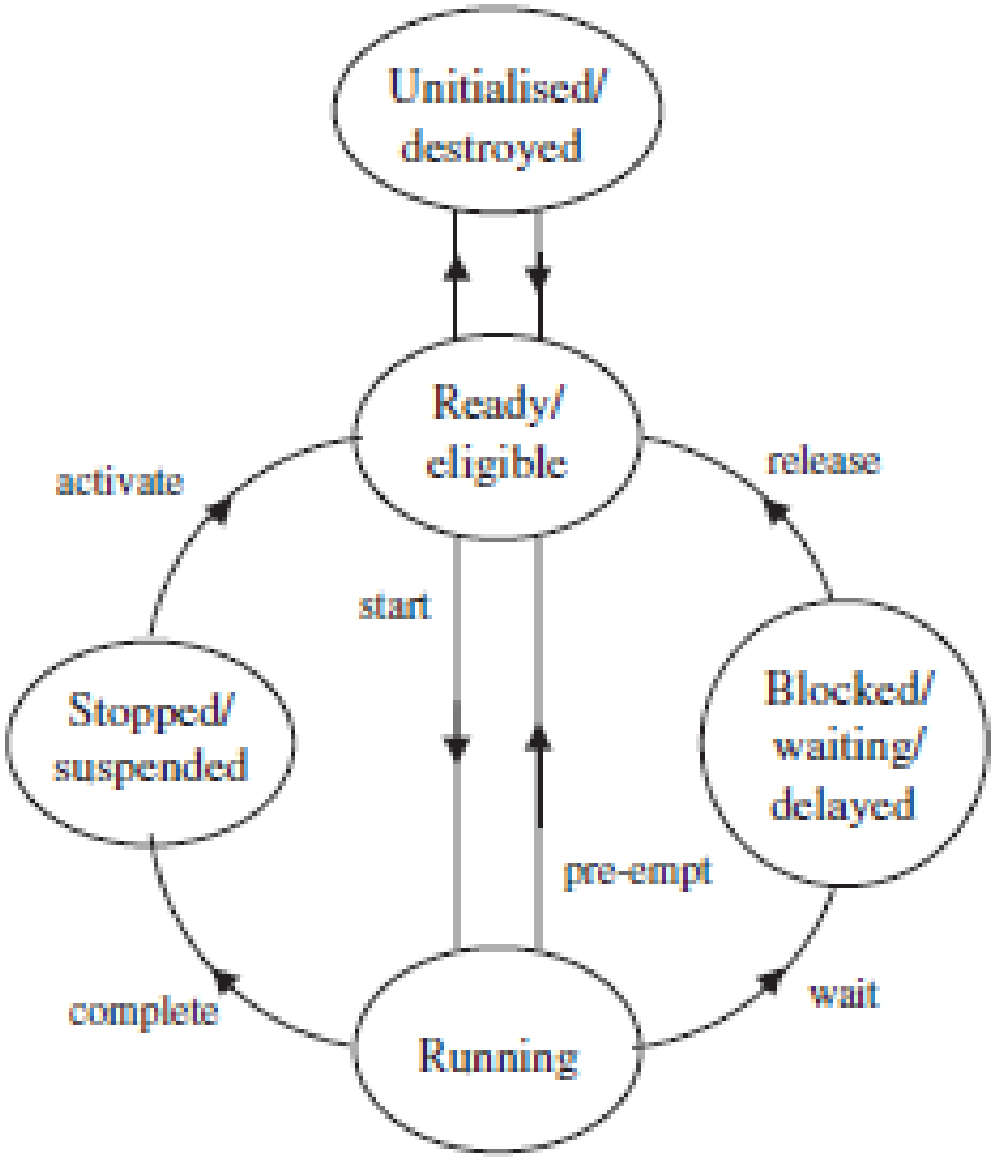


Figure 18.6 Round robin scheduling

Round robin scheduling and context switching: Task states

- Clearly, only one task is running at any one time. Others may need to run, but at any one instant do not have the chance.
- Others may just need to respond to a particular set of circumstances and hence only be active at certain times during program execution.
- It is important, therefore, to recognize that tasks can move between different states.

Round robin scheduling and context switching: Task states



Round robin scheduling and context switching: Task states

1. **Ready (or eligible):** The task is ready to run and will do so as soon as it is allocated CPU time. The task leaves this state and enters the active state when it is started by the scheduler.
2. **Running/Active:** The task has been allocated CPU time and is executing. A number of things can cause the task to leave this state. Maybe it simply completes and no longer needs CPU time. Alternatively, the scheduler may pre-empt it, so that another task can run. Finally, it may enter a blocked or waiting state for one of the reasons described below.

Round robin scheduling and context switching: Task states

3. **Blocked/waiting/delayed:** This state represents a task which is ready to run, but for one reason or another is not allowed to. The task could be waiting for some data to arrive or for a resource that it needs, which is currently being used by another task, or it could be waiting for a period of time to be up. The state is left when the task is released from the condition which is holding it there.

4. **Stopped/suspended/dormant**: The task does not at present need CPU time. A task leaves this state and enters the ready state when it is activated again, for whatever reason.

Round robin scheduling and context switching: Task states

5. **Initialized/destroyed:** In this state the task no longer exists as far as the RTOS is concerned. Removing unneeded tasks from the task list simplifies scheduler operation and reduces demands on memory.

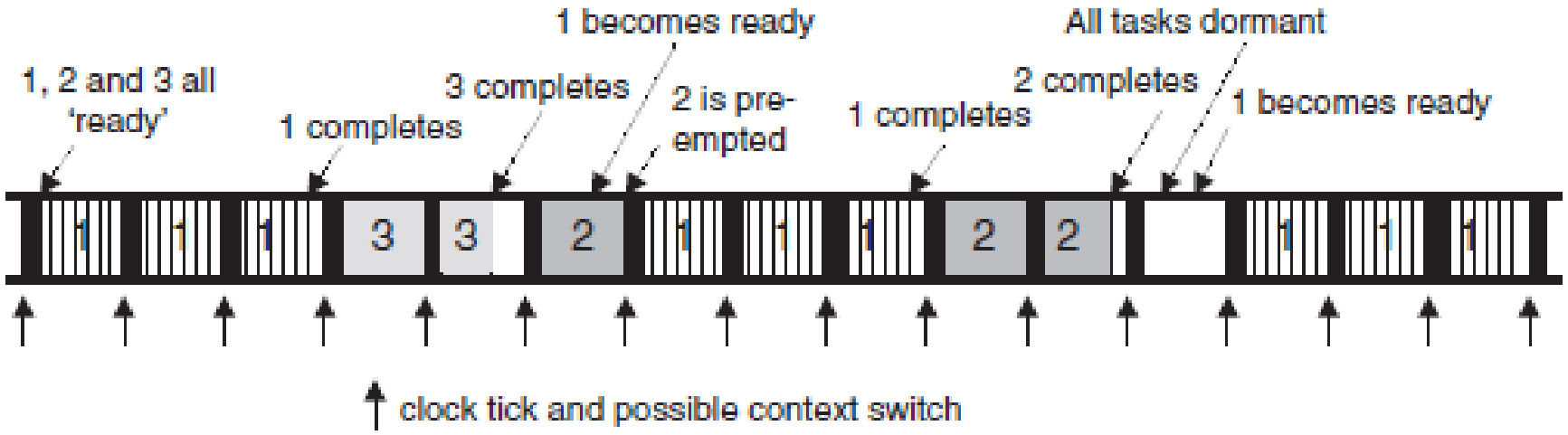
Prioritized pre-emptive scheduling

- In round robin scheduling tasks become subservient to a higher power – the operating system.
- Yet all tasks are of equal priority, so an unimportant task gets just as much access to the CPU as one of tip-top priority.
- We can change this by prioritizing tasks.
- In the prioritized pre-emptive scheduler, tasks are given priorities.
- High priority tasks are now allowed to complete before any time whatsoever is given to tasks of lower priority. The scheduler is still run by a clock tick.
- On every tick it checks which ready task has the highest priority. Whichever that is gets access to the CPU.

Prioritized pre-emptive scheduling

- An executing task which still needs CPU time, and is highest priority, keeps the CPU.
- A low priority task which is executing is replaced by one of higher priority, if that has become ready.

Prioritized pre-emptive scheduling



Task	Priority	Duration (in time slices)
1	1 (highest)	2.7
2	3	2.8
3	2	1.5

Figure 18.8 Prioritised pre-emptive scheduling

Prioritized pre-emptive scheduling: Drawbacks

- The scheduler must hold all context information for all tasks that it pre-empts.
- This is generally done in one stack per task and is memory intensive.
- The context switching can also be time-consuming.
- Moreover, tasks must be written in such a way that they can be switched at any time during their operation.
- An alternative to pre-emptive scheduling is cooperative scheduling.

Cooperative scheduling

- Now each task must relinquish, of its own accord, its CPU access at some time in its operation.
- This sounds like we're blocking out the operating system, but if each task is written correctly this need not be.
- The advantage is that the task relinquishes control at a moment of its choosing, so it can control its context saving and the central overhead is not required.
- Cooperative scheduling is unlikely to be quite as responsive to tight deadlines as pre-emptive scheduling.
- It does, however, need less memory and can switch tasks quicker. This is very important in the small system, such as one based on a PIC microcontroller.

The role of interrupts in scheduling

- The first use of interrupts is almost always to provide the clock tick, through a timer interrupt on overflow.
- Beyond this, ISRs are usually used to supply urgent information to the tasks or scheduler.
- The interrupt could, for example, be set to signal that a certain event has occurred, thereby releasing a task from a blocked state.
- The ISRs themselves are not normally used as tasks.

Developing tasks: Defining tasks

- The programmer has to actually choose which activities of the system will be defined as tasks.
- The number of tasks created should not be too many.
- More tasks generally imply more programming complexity, and for every task switch there is a time and memory overhead.
- A set of activities which are closely related in time are likely to serve a single deadline and should therefore be grouped together into a single task.
- A set of activities which are closely related in function and interchange a large amount of data should also be grouped into a single task.

Developing tasks: Writing tasks and setting priority

- Tasks should be written as if they are to run continuously, as self-contained and semi-autonomous programs, even though they may be discontinued by the scheduler.
- They cannot call on a section of another's code, but can access common code, for example C libraries.
- They may depend on services provided by each other and may need to be synchronized with each other.
- In either case, the RTOS will have special services to allow this to happen.
- In all cases but the most simple, the RTOS allows the programmer to set task priorities.

Developing tasks: Writing tasks and setting priority

- In the case of static priority, priorities are fixed.
- In the case of dynamic priority, priorities may be changed as the program runs.
- One way of looking at priority is to consider how important a task is to the operation and well-being of the system, its user and environment.
- Priority can then be allocated:

1. **Highest priority**: tasks essential for system survival
2. **Middle priority**: tasks essential for correct system operation
3. **Low priority**: tasks needed for adequate system operation – these tasks might occasionally be expendable or a delay in their completion might be acceptable.

Developing tasks: Writing tasks and setting priority

- Priorities can also be considered by evaluating the task deadlines.
- In this case high priority is given to tasks which have very tight time deadlines.
- If, however, a task has a demanding deadline, but just isn't very important in the overall scheme of things, then it may still end up with a low priority.

Data and resource protection- the semaphore

- Several tasks may need to access the same item of shared resource.
- This could be hardware (including memory or peripheral) or a common software module.
- This requires some care.
- A method for dealing with this is by **the semaphore**.
- A semaphore is allocated to each shared resource, which is used to indicate if it is in use.

Data and resource protection- the semaphore

- In a **binary semaphore**, the first task needing to use the resource will find the semaphore in a **GO** state and will change it to **WAIT** before starting to use the resource.
- Any other task in the mean time needing to use the resource will have to enter the blocked state.
- When the first task has completed accessing the resource, it changes the semaphore back to **GO**.
- This leads to the concept of mutual exclusion **(mutex)**; when one task is accessing the resource, all others are excluded.

Data and resource protection- the semaphore

- The **counting semaphore** is used for a set of identical resources, for example a group of printers.
- Now the semaphore is initially set to the number of units of resource.
- As any task uses one of the units, it decrements the semaphore by one, incrementing it again on completion of use.
- Thus, the counting semaphore holds the number of units that are available for use.

Data and resource protection- the semaphore

- As an effect of setting a semaphore to the **WAIT** state is that another task becomes blocked, they can be used as a means of providing time synchronization and signaling between different tasks.
- One task can block another by setting a semaphore and can release it at a time of its choosing by clearing the semaphore.
- **Remember the 'interrupt', the high priority task :** By using a semaphore, a low priority task can turn the tables on the interrupt!
- If a low priority task sets a semaphore for a resource that the high priority task needs, it can block that task. This leads to a dangerous condition known as **priority inversion**.

Summary

- Multitasking
- Drawbacks in sequential programming
- Real time operating system basis
- Scheduling strategies
- Defining and writing tasks
- Usage of Semaphores