电子科技大学
University of Electronic Science and Technology of China

University of Glasgow

UESTC2004: Embedded Processors

Developing and Debugging Embedded System Firmware

Lecture 9

Dr. Lina Mohjazi
Lecturer
Glasgow College, UESTC
James Watt School of Engineering

➢Objectives - To understand:

- ❖ Quality attributes of embedded systems
- ❖ Design and development of embedded systems
- ❖ Types of embedded systems
- ❖ Time management and scheduling in embedded systems
- ❖ Debugging tools used for embedded systems

Quality Attributes in Embedded Systems

Quality Attributes of Embedded System:

Quality attributes are the non-functional requirements that need to be documented properly in any system design

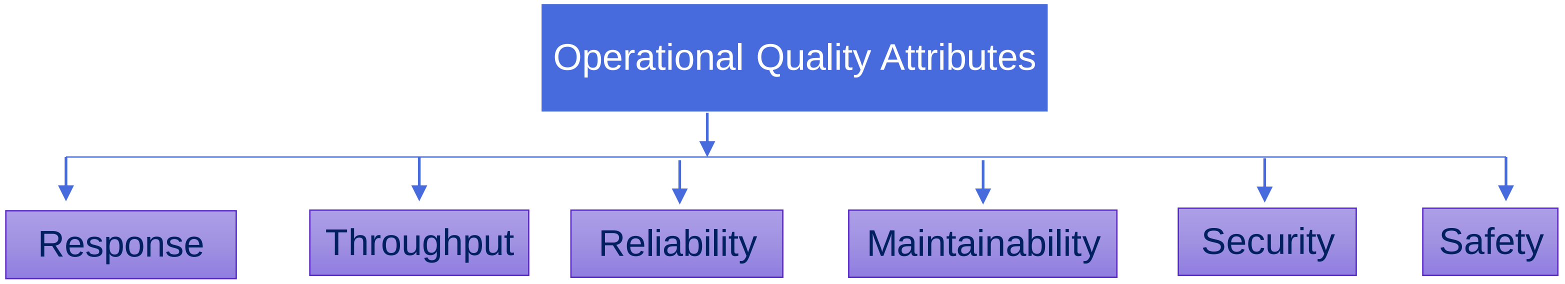
➢ Quality attributes can be classified as

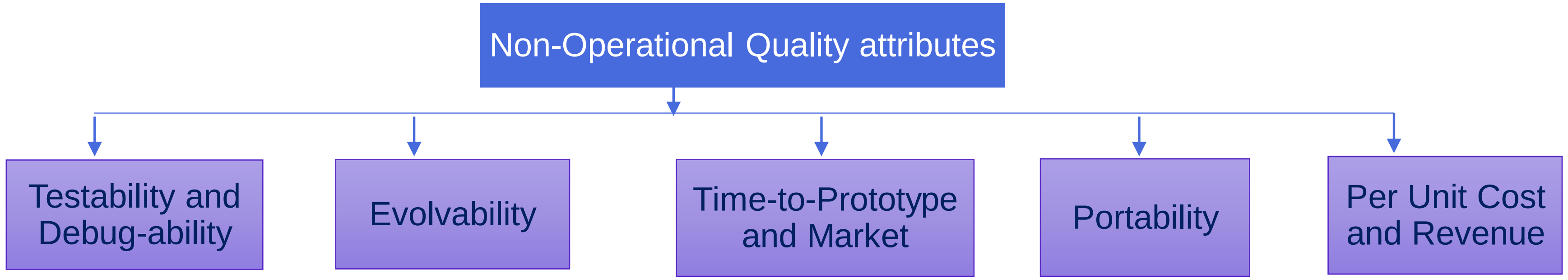
1. Operational quality attributes
2. Non-operational quality attributes

➢ *Operational Quality Attributes:* The operational quality attributes represent the relevant quality attributes related to the embedded system when it is in the operational mode or online mode.

➢ *Non-Operational Quality Attributes*: The quality attributes that needs to be addressed for the product not on the basis of operational aspects are grouped under this category.

Quality Attributes in Embedded Systems




Non-Operational Quality attributes
Testability and Debug-ability
Evolvability
Time-to-Prototype and Market
Portability
Per Unit Cost and Revenue

Embedded Firmware

Introduction:

- **Definition:** The control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system.
- It is an un-avoidable part of an embedded system.
- The embedded firmware can be developed in various methods like:
  - Write the program in high level languages like **Embedded C/C++** using an Integrated Development Environment (IDE):
    - *The IDE will contain an editor, compiler, linker, debugger, simulator etc.*
    - *IDEs are different for different family of processors/controllers*.
  - Write the program in **Assembly Language** using the instructions supported by your application's target processor/controller

Embedded Firmware – Design and Development

- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements of the product.
- The embedded firmware is the master brain of the embedded system.
- The embedded firmware imparts intelligence to an Embedded system.
- It is a one-time process and it can happen at any stage.
- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.
- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.
- In case of hardware breakdown, the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.

Embedded Firmware – Design and Development

- The embedded firmware is usually stored in a permanent memory (ROM) and it is non-alterable by end users.

- Designing Embedded firmware requires understanding of:

  - embedded product **hardware**:
    - interfacing components
    - memory map details
    - I/O port details
    - configuration and register details of various hardware chips used.

  - some programming language
    - Low level Assembly Language
    - High level language like C/C++
    - a combination of the two.

Embedded Firmware – Design and Development

- There exist two basic approaches for the design and implementation of embedded firmware, namely:

  1. The Super loop based approach

  2. The Embedded Operating System based approach

- **The decision on which approach needs to be adopted for firmware development is purely dependent on the <u>complexity</u> and <u>system requirements</u>**

Embedded Firmware – Super Loop Approach

- **The Super loop:** The Super loop based firmware development approach is suitable for applications that are *not time critical* and where the *response time is not so important* (Ex. Embedded systems where missing deadlines are acceptable).

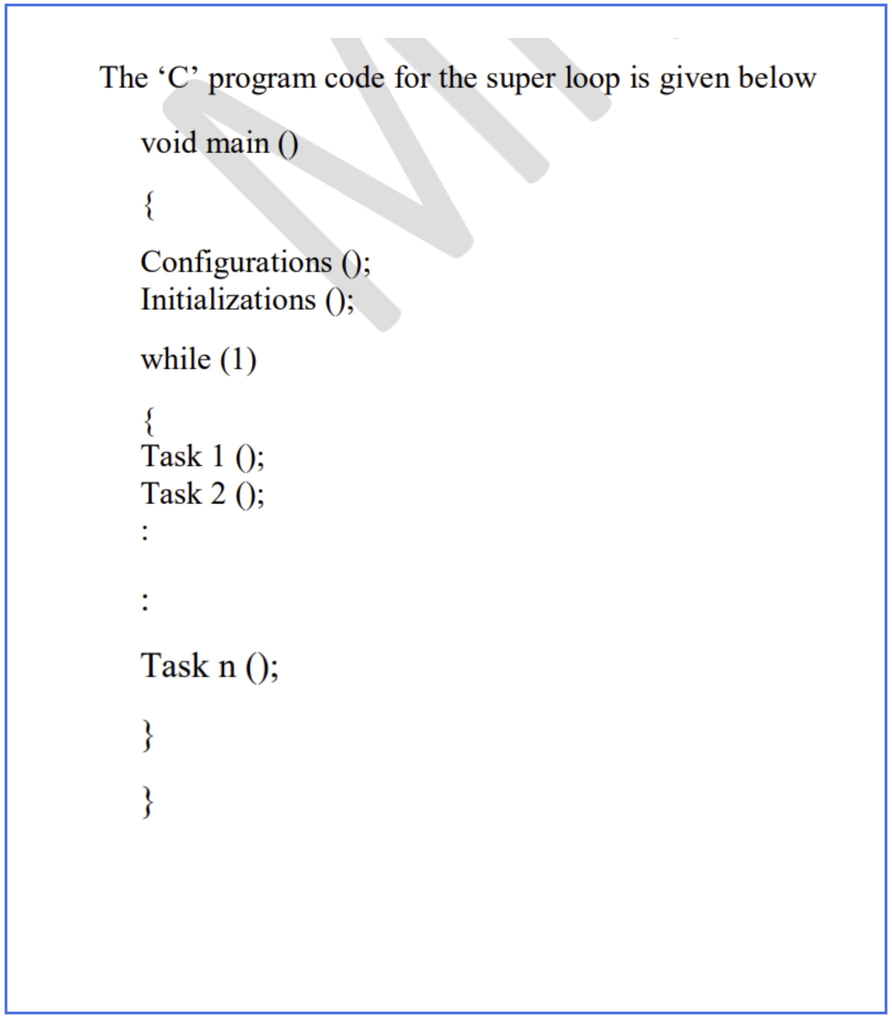
  - It is very similar to a conventional procedural programming where the code is executed task by task.

  - The tasks are executed in a never ending loop.

  - The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task.

Embedded Firmware – Super Loop Approach

- A typical super loop implementation will look like:
  1. Configure the common parameters and perform initialization for various hardware components memory, registers etc.
  2. Start the first task and execute it
  3. Execute the second task
  4. Execute the next task
  5. :
  6. :
  7. Execute the last defined task
  8. Jump back to the first task and follow the same flow.



The 'C' program code for the super loop is given below

```
void main ()
{
Configurations ();
Initializations ();
while (1)
{
Task 1 ();
Task 2 ();
:

:

Task n ();
}
}
```

University of Glasgow

Embedded Firmware – Super Loop Approach

- **Pros:**
  - Doesn't require an Operating System for task scheduling and monitoring and free from OS related overheads.
  - Simple and straight forward design.
  - Reduced memory footprint.

- **Cons:**
  - Non-Real time in execution behavior (*As the number of tasks increases the frequency at which a task gets CPU time for execution also increases*).
  - Any issues in any task execution may affect the functioning of the product (This can be effectively tackled by using Watch Dog Timers for task execution monitoring).

Enhancements:

- Combine Super loop based technique with interrupts.
- Execute the tasks (like keyboard handling) which require Real time attention as Interrupt Service routines.

mbedded Firmware – Embedded OS based Approach

- **Embedded OS based Approach**: The embedded device contains an Embedded Operating System which can be one of:

  1. A Real Time Operating System (RTOS).
  2. A Customized General Purpose Operating System (GPOS)

- Categorized based on: **1) type of kernel and kernel services, 2) purpose and type of computing systems where the OS is deployed, 3) the responsiveness to applications.**

- The Embedded OS is responsible for **scheduling the execution** of user tasks and the **allocation of system resources** among multiple tasks.

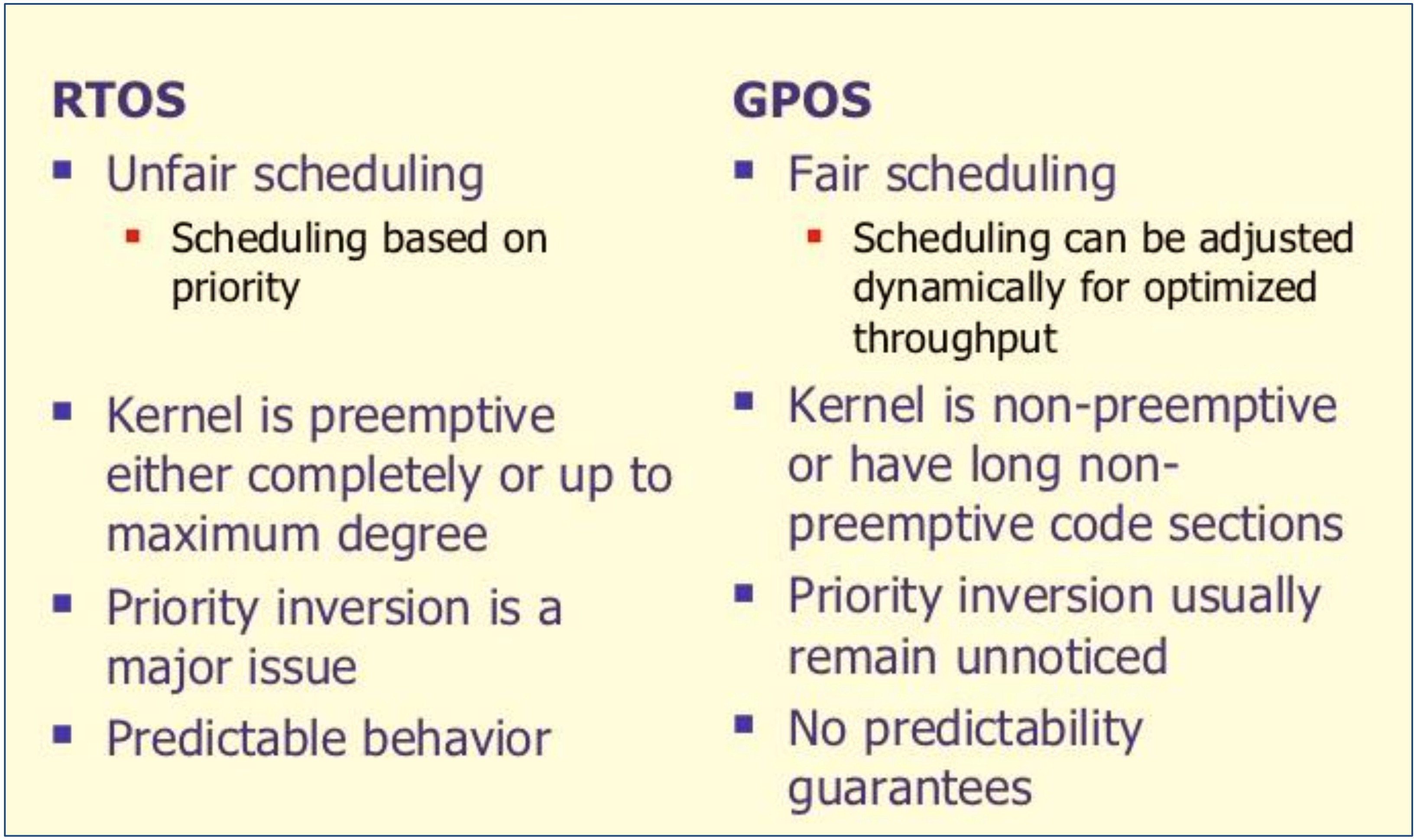
- It Involves high OS related overheads apart from managing and executing user defined tasks.

- Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs etc are examples of embedded devices running on embedded **GPOS**s.

- ‘Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on **RTOS**s

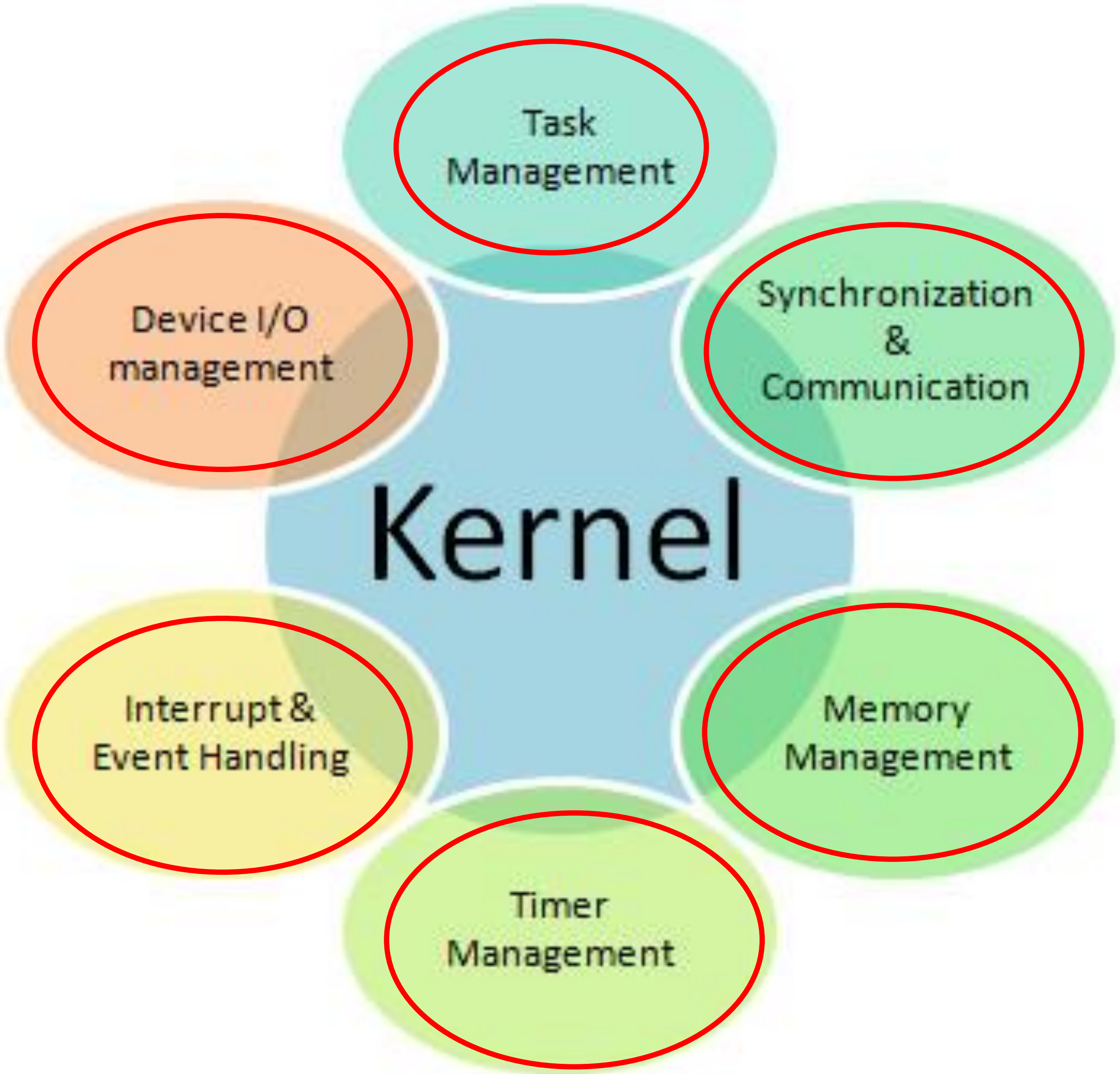
University of Glasgow

Embedded Firmware – RTOS vs. GPOS



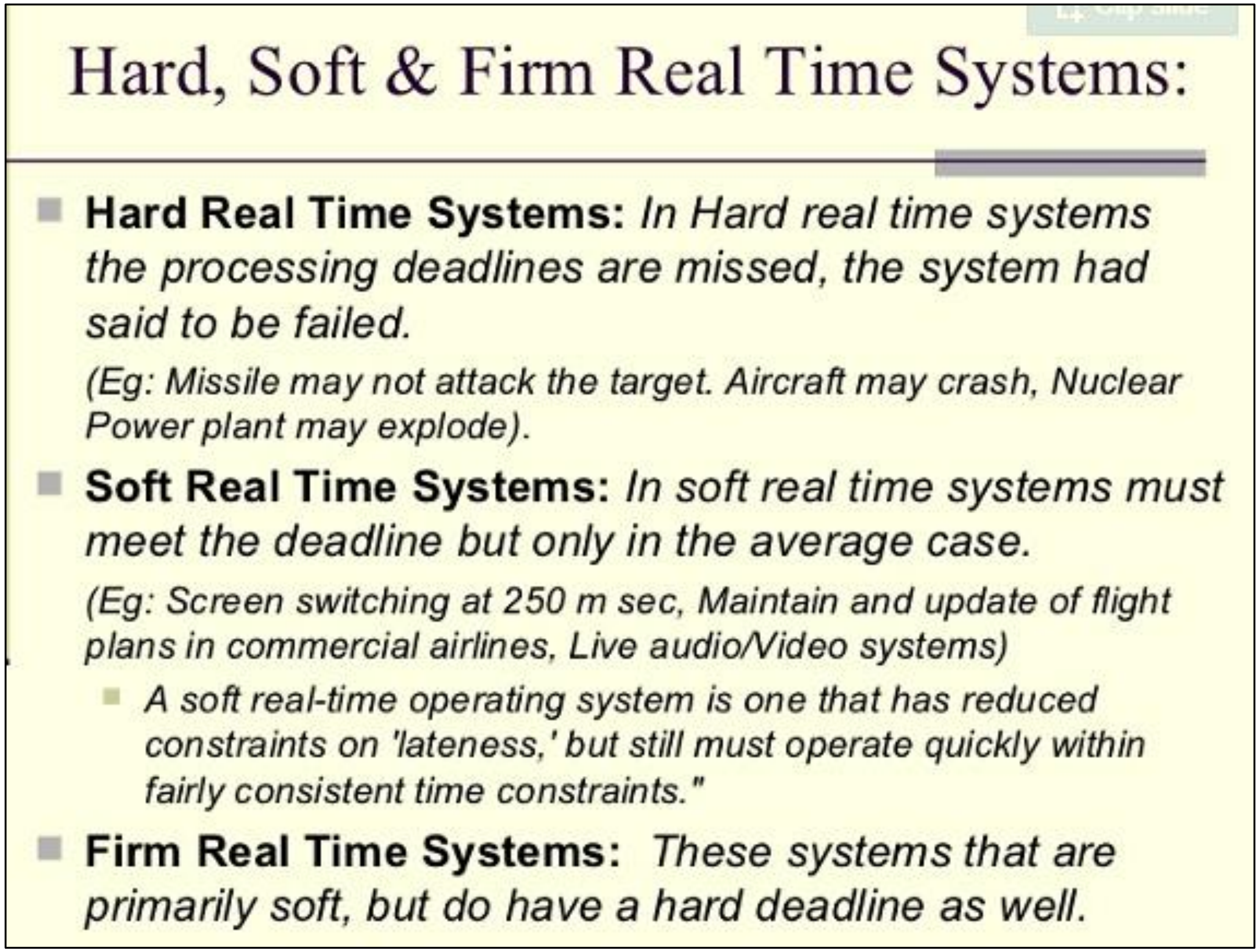
RTOS	GPOS
▪ Unfair scheduling ▪ Scheduling based on priority	▪ Fair scheduling ▪ Scheduling can be adjusted dynamically for optimized throughput
▪ Kernel is preemptive either completely or up to maximum degree	▪ Kernel is non-preemptive or have long non-preemptive code sections
▪ Priority inversion is a major issue	▪ Priority inversion usually remain unnoticed
▪ Predictable behavior	▪ No predictability guarantees

Embedded Firmware – Kernel of RTOS



Embedded Firmware – Time Management

Based on the degree of tolerance in the timing constraints, RTOSs are classified into three types:



Hard, Soft & Firm Real Time Systems:

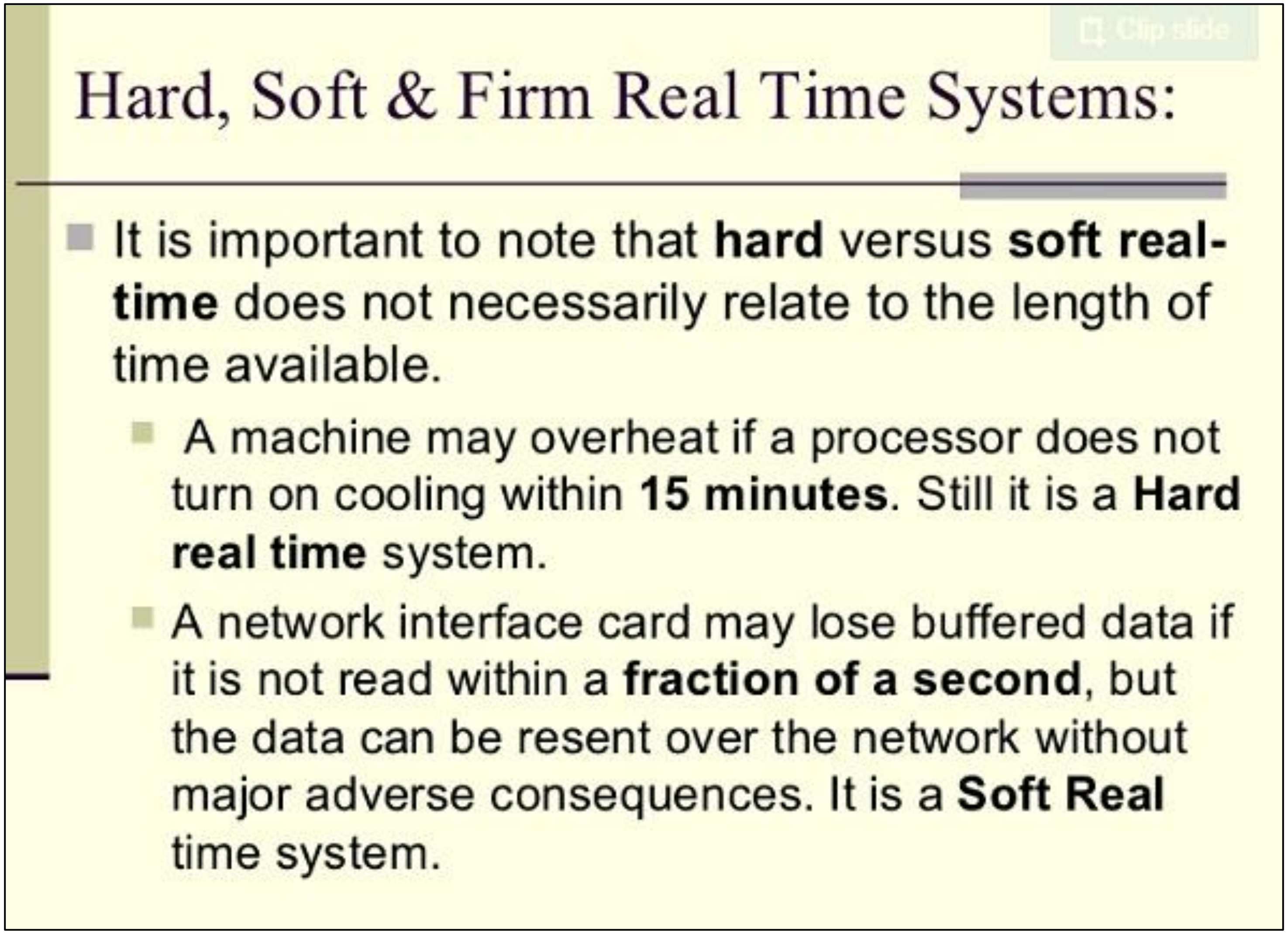
- **Hard Real Time Systems:** *In Hard real time systems the processing deadlines are missed, the system had said to be failed.*

  (Eg: Missile may not attack the target. Aircraft may crash, Nuclear Power plant may explode).
- **Soft Real Time Systems:** *In soft real time systems must meet the deadline but only in the average case.*

  (Eg: Screen switching at 250 m sec, Maintain and update of flight plans in commercial airlines, Live audio/Video systems)
  - *A soft real-time operating system is one that has reduced constraints on 'lateness,' but still must operate quickly within fairly consistent time constraints."*
- **Firm Real Time Systems:** *These systems that are primarily soft, but do have a hard deadline as well.*

University of Glasgow

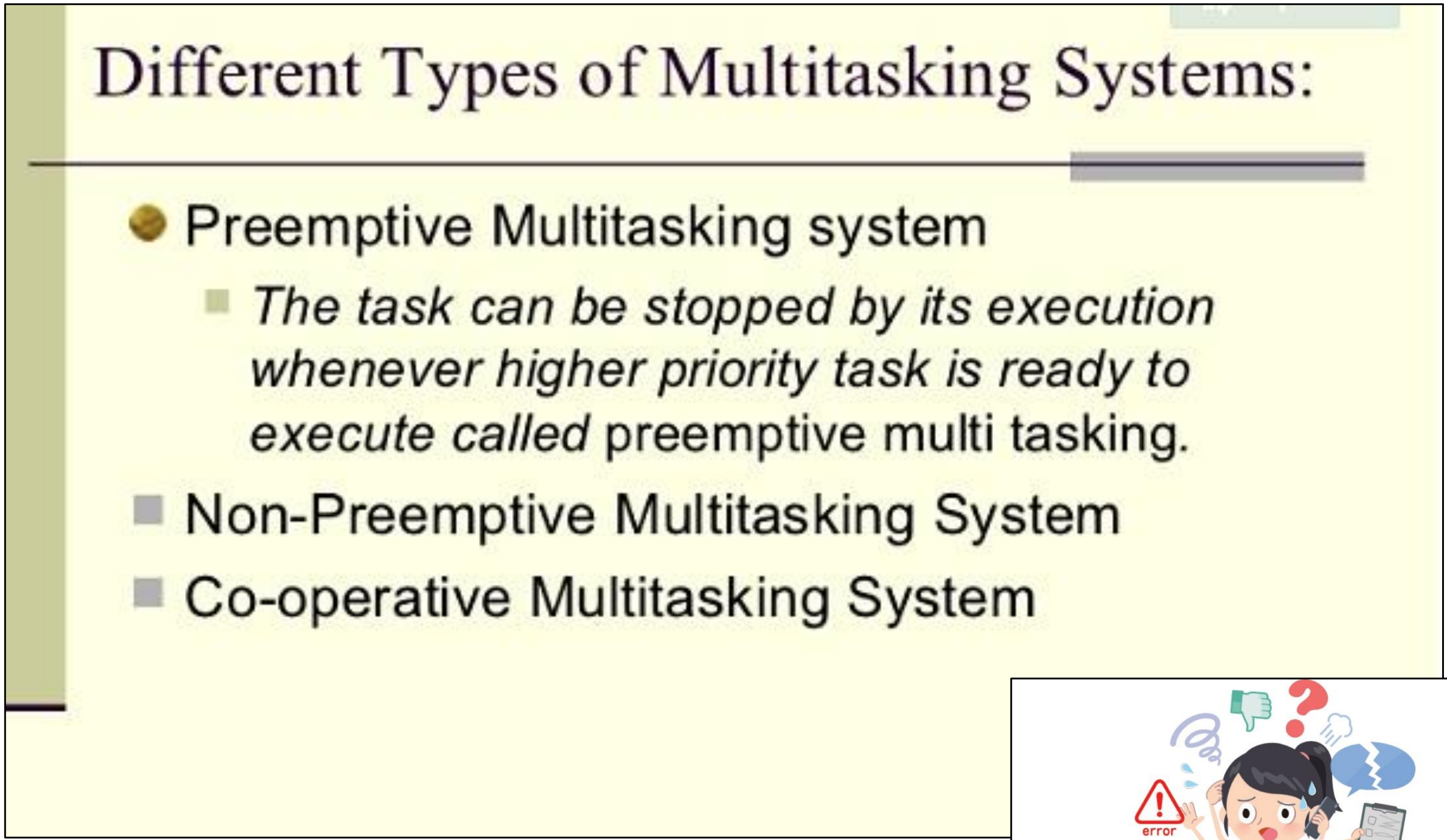
Embedded Firmware – Time Management



Hard, Soft & Firm Real Time Systems:

- It is important to note that **hard** versus **soft real-time** does not necessarily relate to the length of time available.
  - A machine may overheat if a processor does not turn on cooling within **15 minutes**. Still it is a **Hard real time** system.
  - A network interface card may lose buffered data if it is not read within a **fraction of a second**, but the data can be resent over the network without major adverse consequences. It is a **Soft Real** time system.

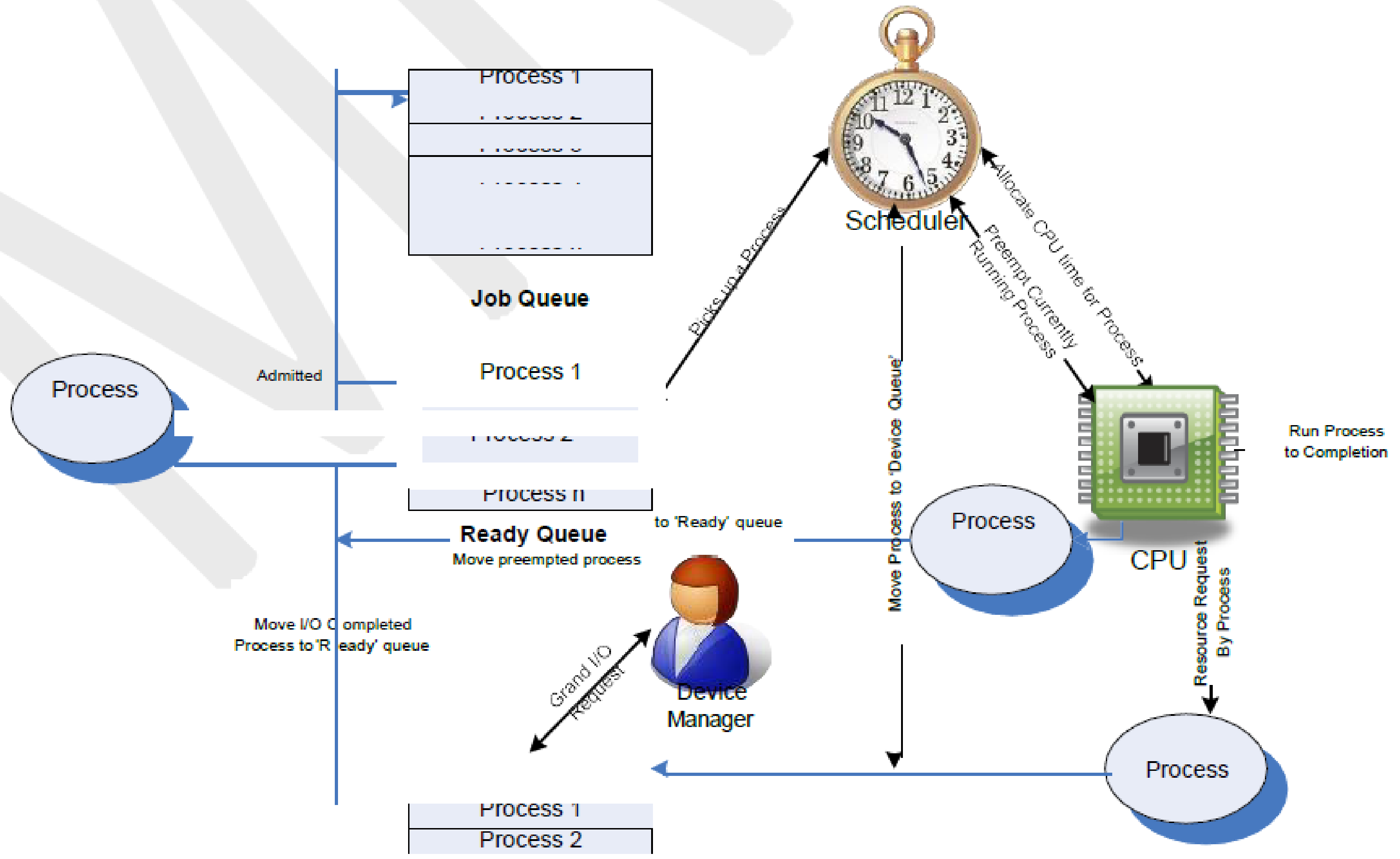
Embedded Firmware – Multitasking



Different Types of Multitasking Systems:

- Preemptive Multitasking system
  - *The task can be stopped by its execution whenever higher priority task is ready to execute called* preemptive multi tasking.
- Non-Preemptive Multitasking System
- Co-operative Multitasking System

Embedded Firmware – Task Scheduling

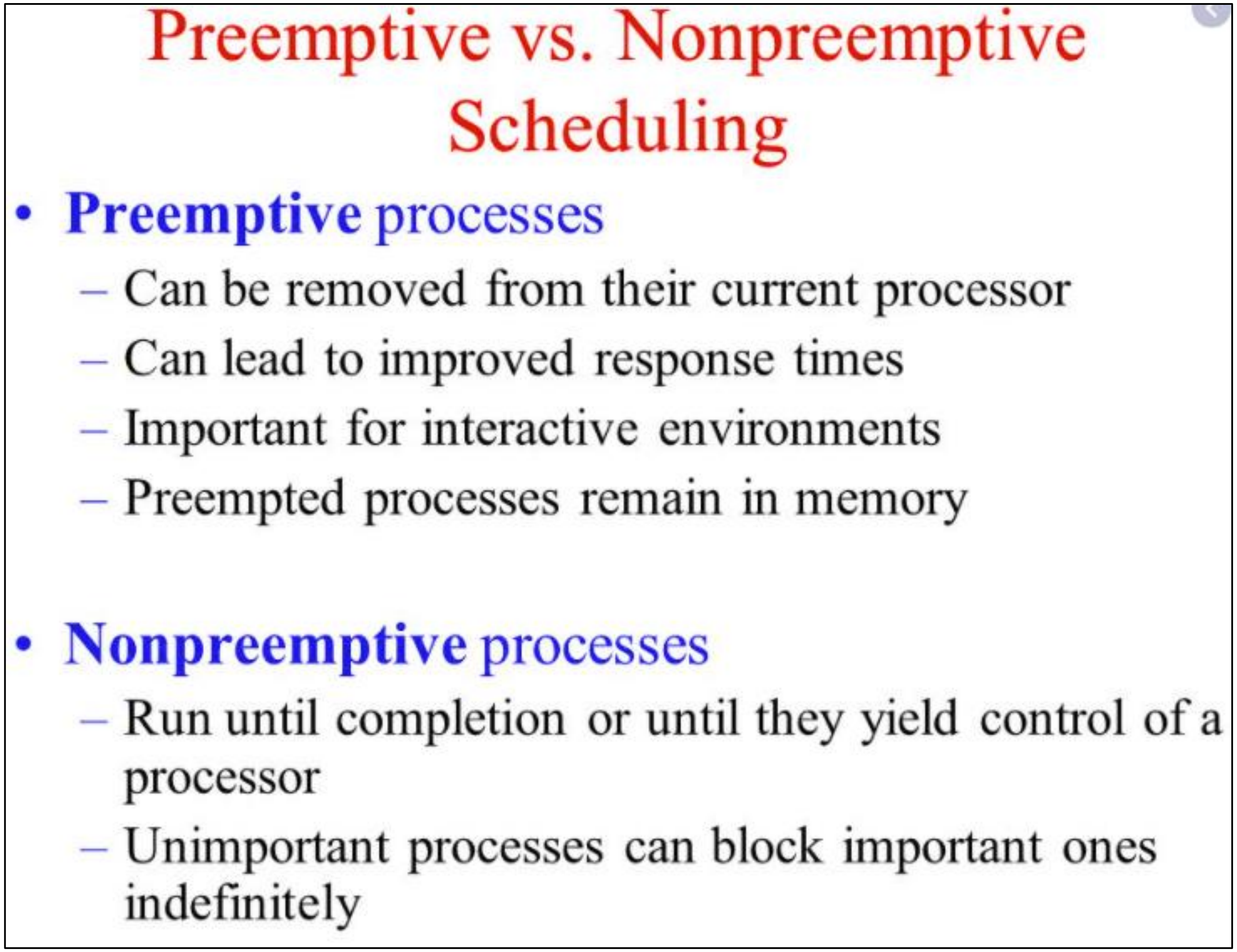


Task Scheduling - Scheduler Selection:

- CPU Utilization
- Throughput
- Turnaround Time
- Waiting Time
- Response Time

To summarize, a good scheduling algorithm has high CPU utilization, minimum Turn Around Time (TAT), maximum throughput and least response time.

Embedded Firmware – Scheduling



Preemptive vs. Nonpreemptive Scheduling

- **Preemptive** processes
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory
- **Nonpreemptive** processes
  - Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely

Embedded Firmware – Non-preemptive

Non-preemptive scheduling – First Come First Served (FCFS)/First In First Out (FIFO) Scheduling:

- Allocates CPU time to the processes based on the order in which they enters the ‘*Ready*’ queue
- The first entered process is serviced first
- It is same as any real world application where queue systems are used; E.g. Ticketing

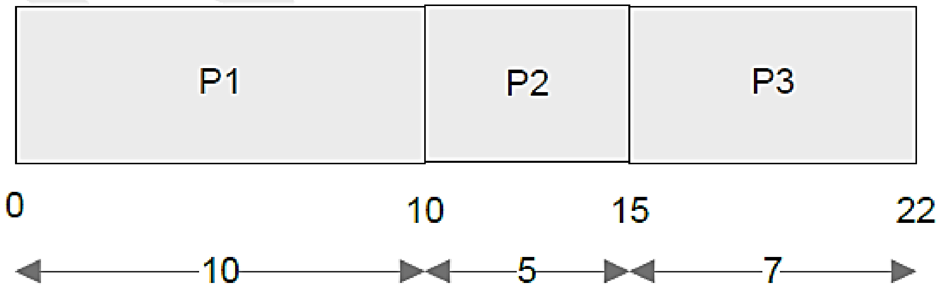
Drawbacks:

- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- In general, FCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- The average waiting time is not minimal for FCFS scheduling algorithm

Embedded Firmware – Non-preemptive

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

Solution: The sequence of execution of the processes by the CPU is represented as



Embedded Firmware – Non-preemptive

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)

Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P2+P3)) / 3

= (0+10+15)/3 = 25/3 = 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 15 ms (-Do-)

Turn Around Time (TAT) for P3 = 22 ms (-Do-)

Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (10+15+22)/3 = 47/3

= 15.66 milliseconds

Embedded Firmware – Non-preemptive

Non-preemptive scheduling – Last Come First Served (LCFS)/Last In First Out (LIFO) Scheduling:

- Allocates CPU time to the processes based on the order in which they are entered in the '*Ready*' queue
- The last entered process is serviced first

Drawbacks:

- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- The average waiting time is not minimal for LCFS scheduling algorithm

University of Glasgow

Embedded Firmware – Non-preemptive

Non-preemptive scheduling – Shortest Job First (SJF) Scheduling.

- Allocates CPU time to the processes based on the execution completion time for tasks
- The average waiting time for a given set of processes is minimal in SJF scheduling
- Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

- A process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the ‘*Ready*’ queue before the process with longest estimated execution time starts its execution
- May lead to the ‘Starvation’ of processes with high estimated completion time
- Difficult to know in advance the next shortest process in the ‘*Ready*’ queue for scheduling since new processes with different estimated execution time keep entering the ‘*Ready*’ queue at any point of time.

Embedded Firmware – Non-preemptive

Non-preemptive scheduling – Priority based Scheduling

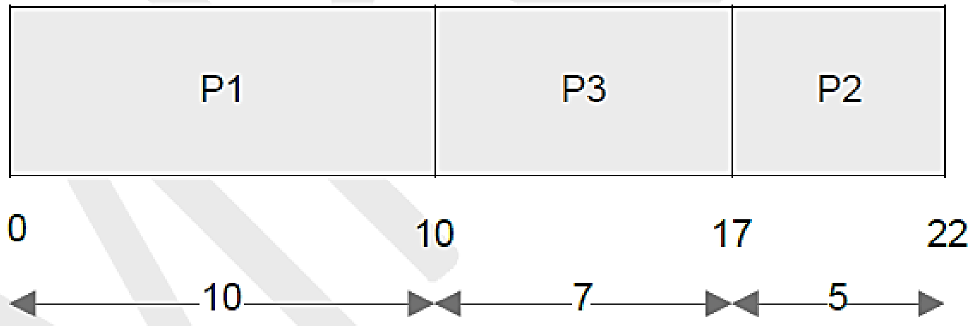
- A priority, which is unique or same is associated with each task
- The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)

- The non-preemptive priority based scheduler sorts the 'Ready' queue based on the priority and picks the process with the highest level of priority for execution

Embedded Firmware – Non-preemptive

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Solution: The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second and so on. The order in which the processes are scheduled for execution is represented as



Embedded Firmware – Non-preemptive

The waiting time for all the processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P3 = 10 ms (P3 starts executing after completing P1)

Waiting Time for P2 = 17 ms (P2 starts executing after completing P1 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P3+P2)) / 3

= (0+10+17)/3 = 27/3

= 9 milliseconds

Embedded Firmware – Non-preemptive

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P3 = 17 ms (-Do-)

Turn Around Time (TAT) for P2 = 22 ms (-Do-)

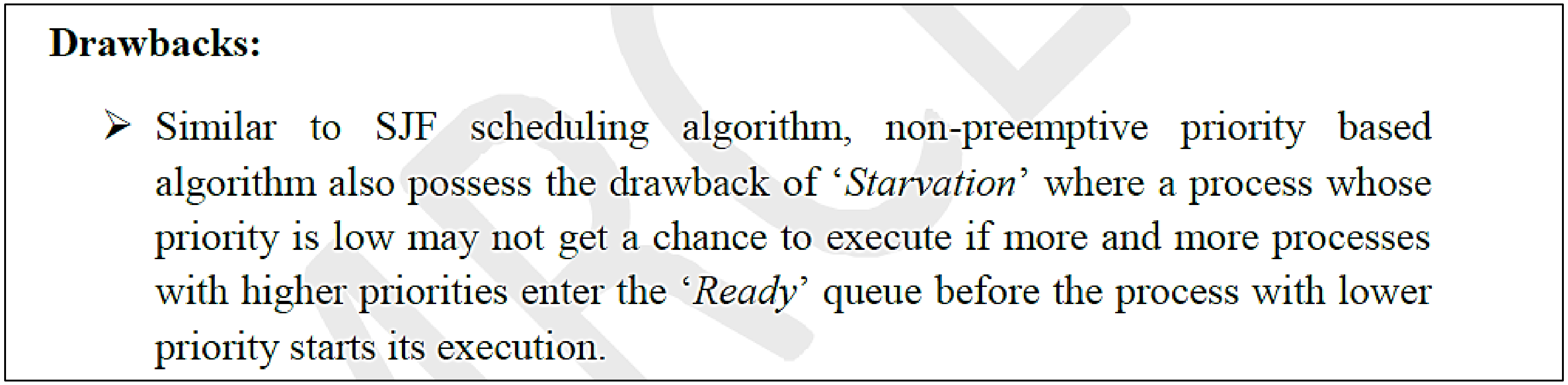
Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P3+P2)) / 3

= (10+17+22)/3 = 49/3

= 16.33 milliseconds

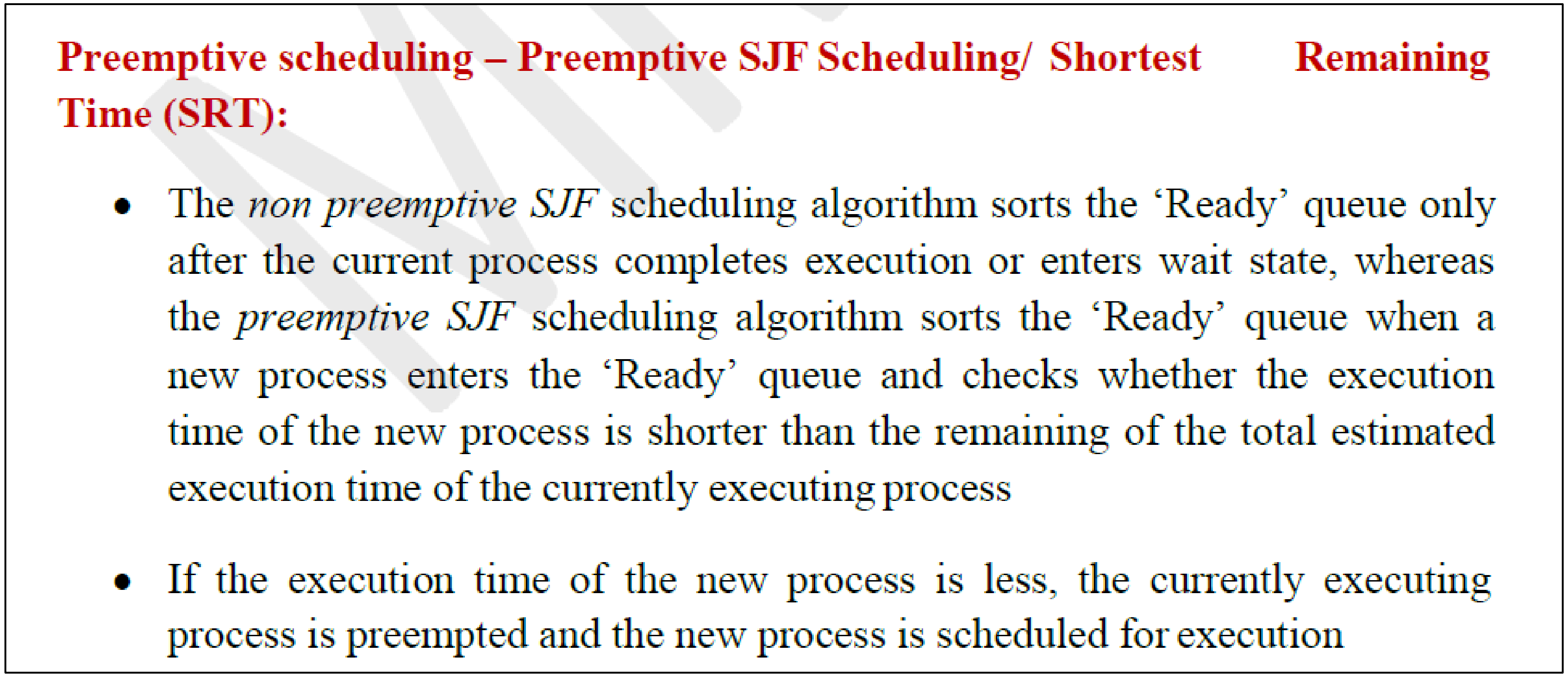
Embedded Firmware – Non-preemptive



Drawbacks:

- Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority starts its execution.

Embedded Firmware – Preemptive Scheduling



Preemptive scheduling – Preemptive SJF Scheduling/ Shortest Remaining Time (SRT):

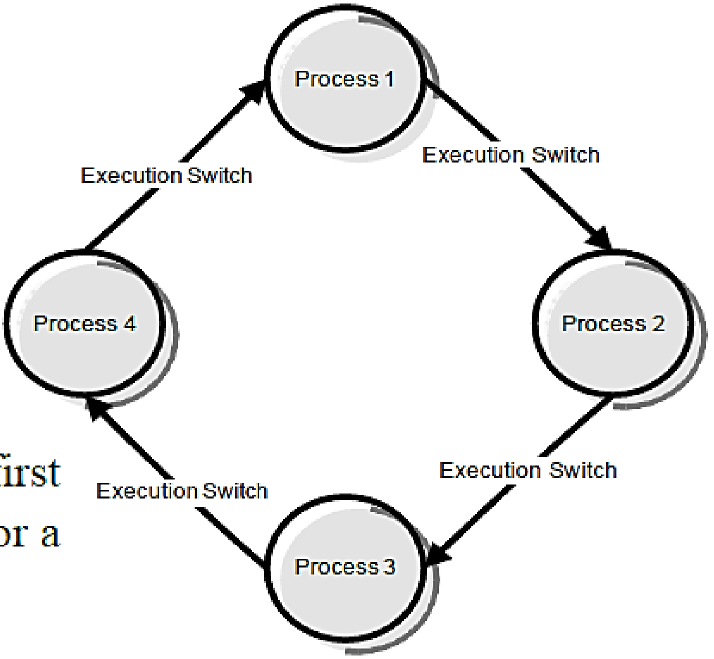
- The *non preemptive SJF* scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas the *preemptive SJF* scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated execution time of the currently executing process

- If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution

Embedded Firmware – Preemptive Scheduling

Preemptive scheduling – Round Robin (RR) Scheduling:

- Each process in the 'Ready' queue is executed for a pre-defined time slot.
- The execution starts with picking up the first process in the 'Ready' queue. It is executed for a pre-defined time



Embedded Firmware – Preemptive Scheduling

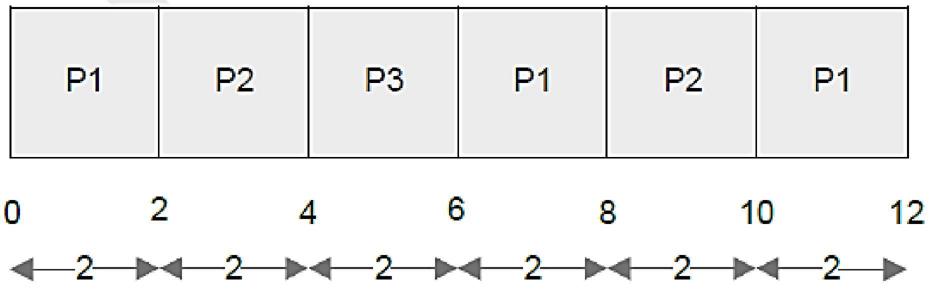
- When the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution.
- This is repeated for all the processes in the 'Ready' queue
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue

University of Glasgow

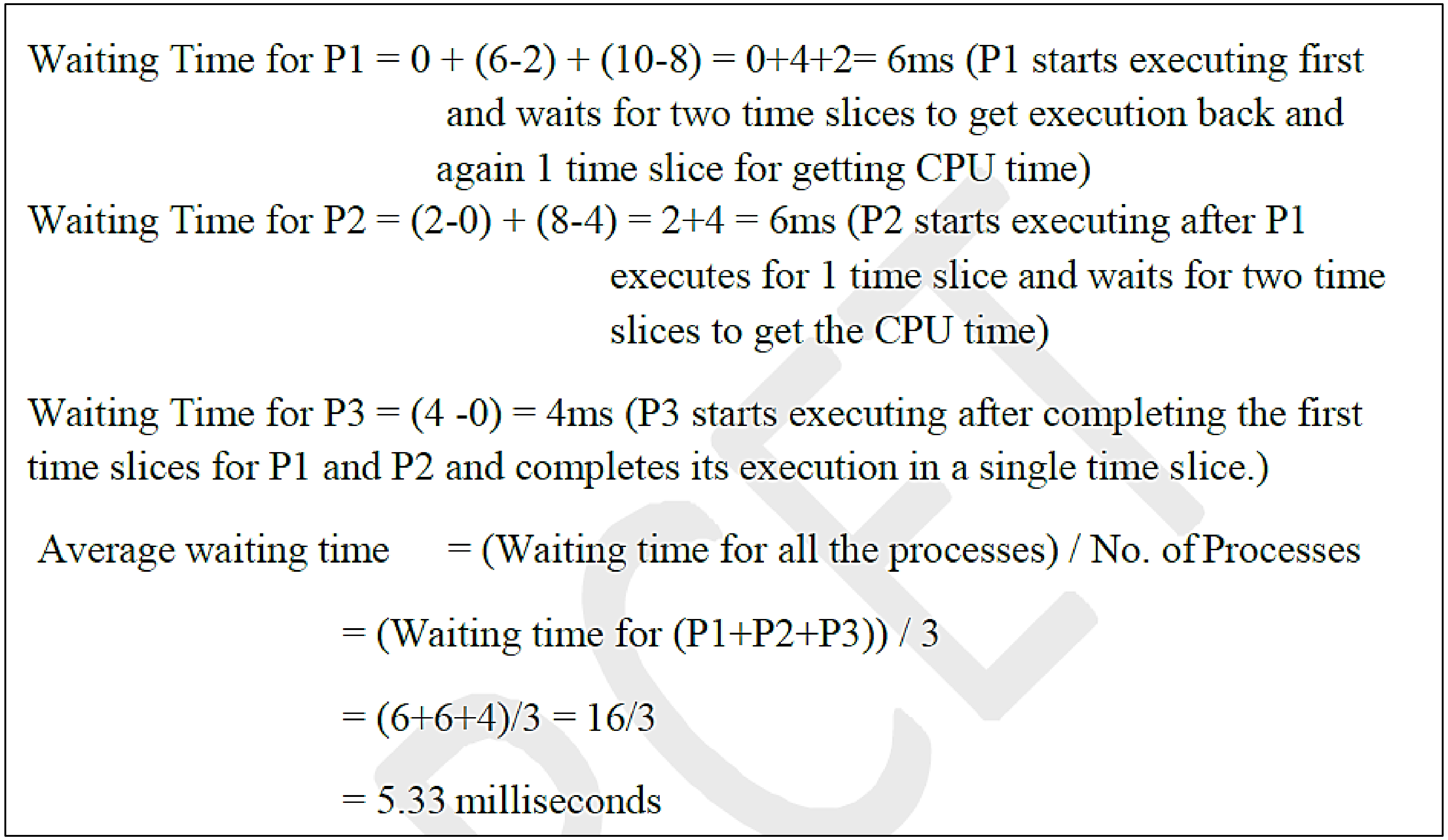
Embedded Firmware – Preemptive Scheduling

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice= 2ms.

Solution: The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



Embedded Firmware – Preemptive Scheduling



Waiting Time for P1 = 0 + (6-2) + (10-8) = 0+4+2= 6ms (P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting Time for P2 = (2-0) + (8-4) = 2+4 = 6ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)

Waiting Time for P3 = (4 -0) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)

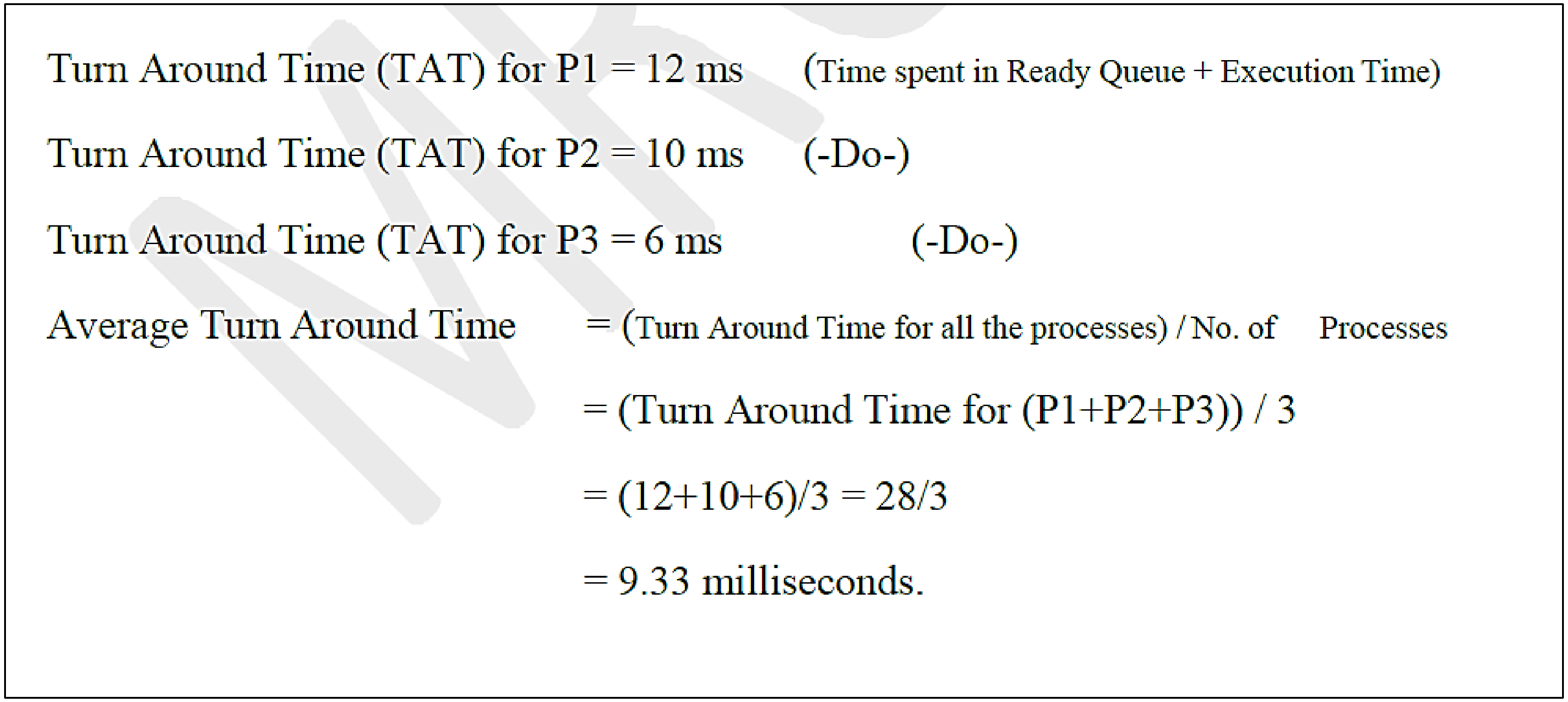
Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P1+P2+P3)) / 3

= (6+6+4)/3 = 16/3

= 5.33 milliseconds

Embedded Firmware – Preemptive Scheduling



Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 10 ms (-Do-)

Turn Around Time (TAT) for P3 = 6 ms (-Do-)

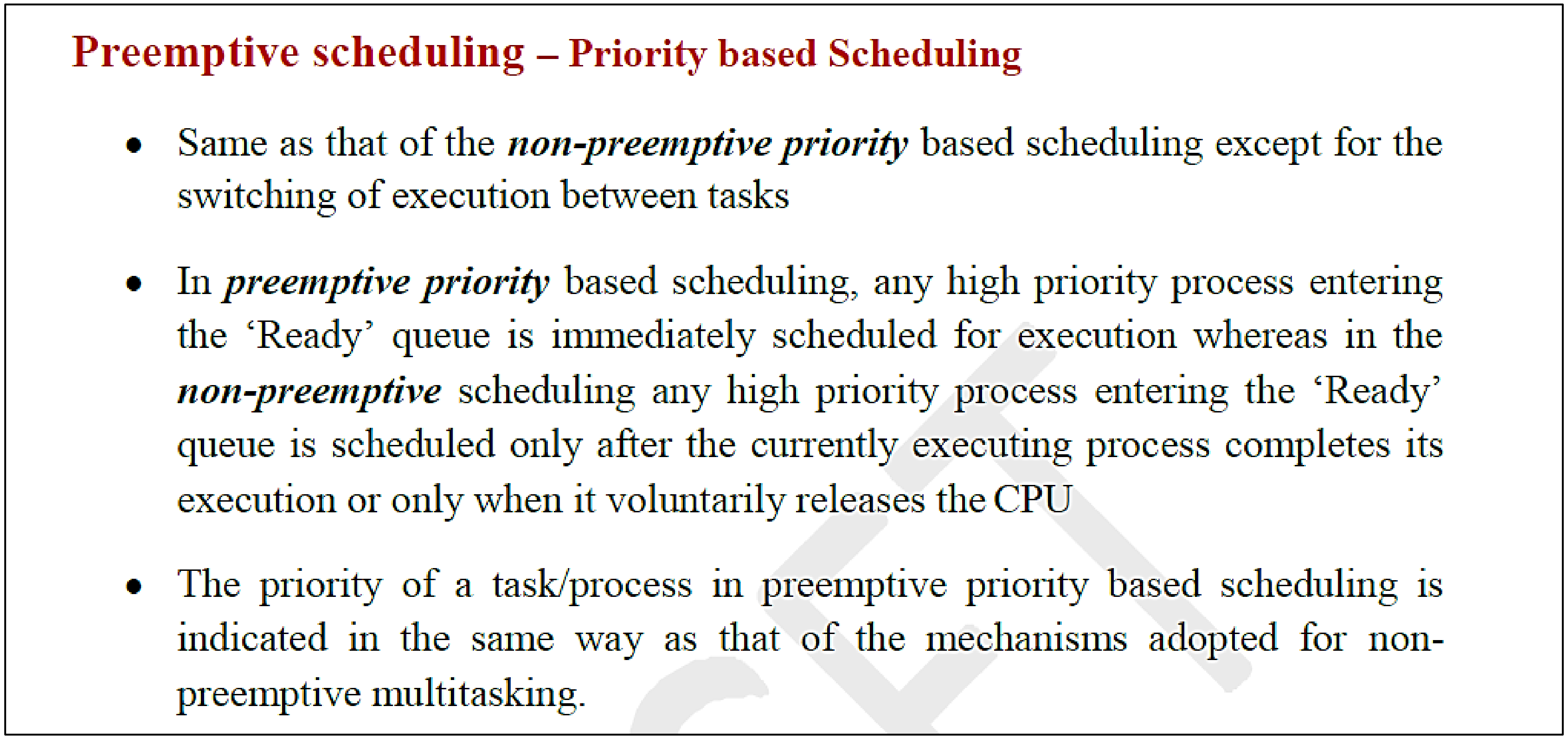
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (12+10+6)/3 = 28/3

= 9.33 milliseconds.

Embedded Firmware – Preemptive Scheduling

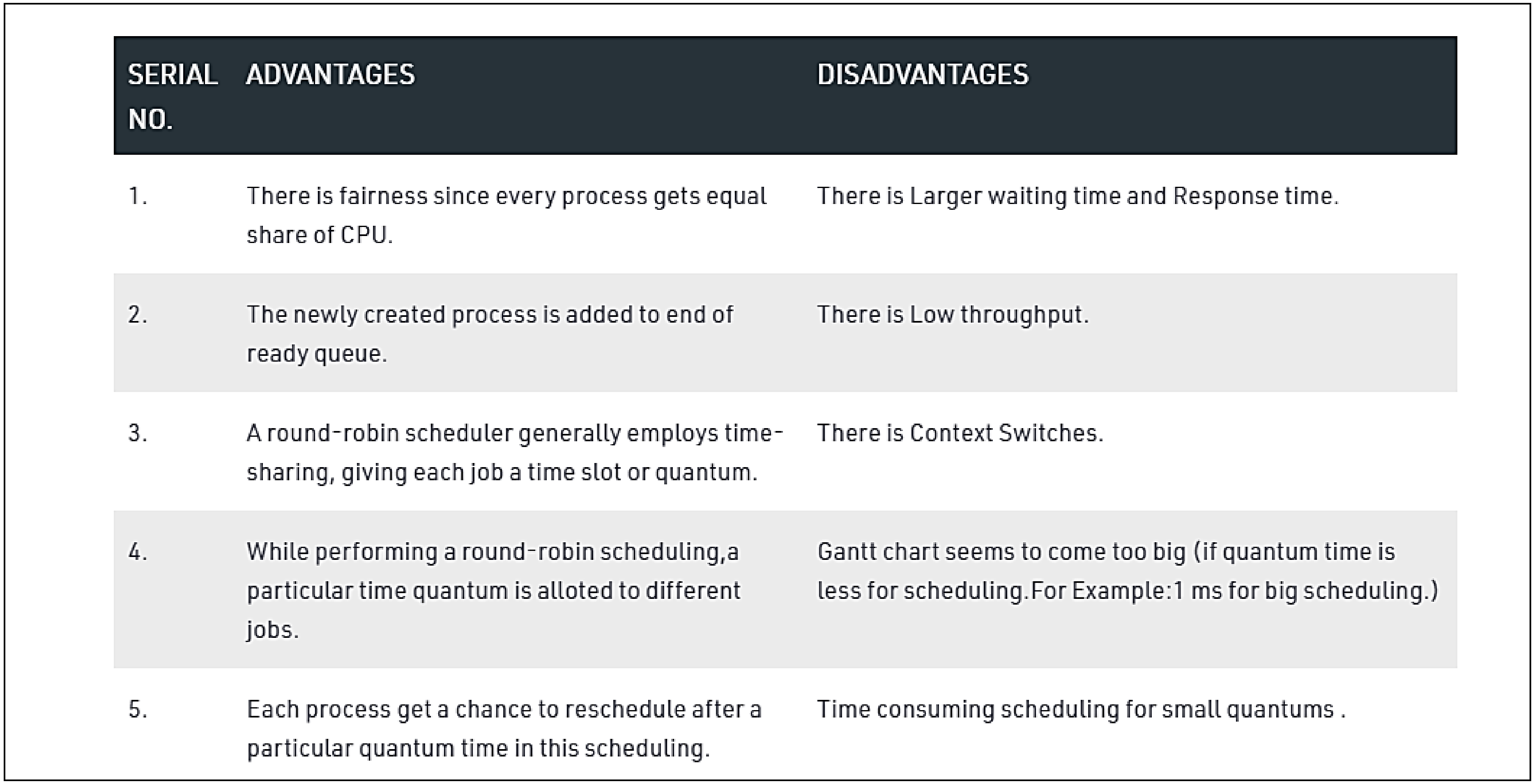


Preemptive scheduling – Priority based Scheduling

- Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks
- In ***preemptive priority*** based scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ***non-preemptive*** scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily releases the CPU
- The priority of a task/process in preemptive priority based scheduling is indicated in the same way as that of the mechanisms adopted for non-preemptive multitasking.

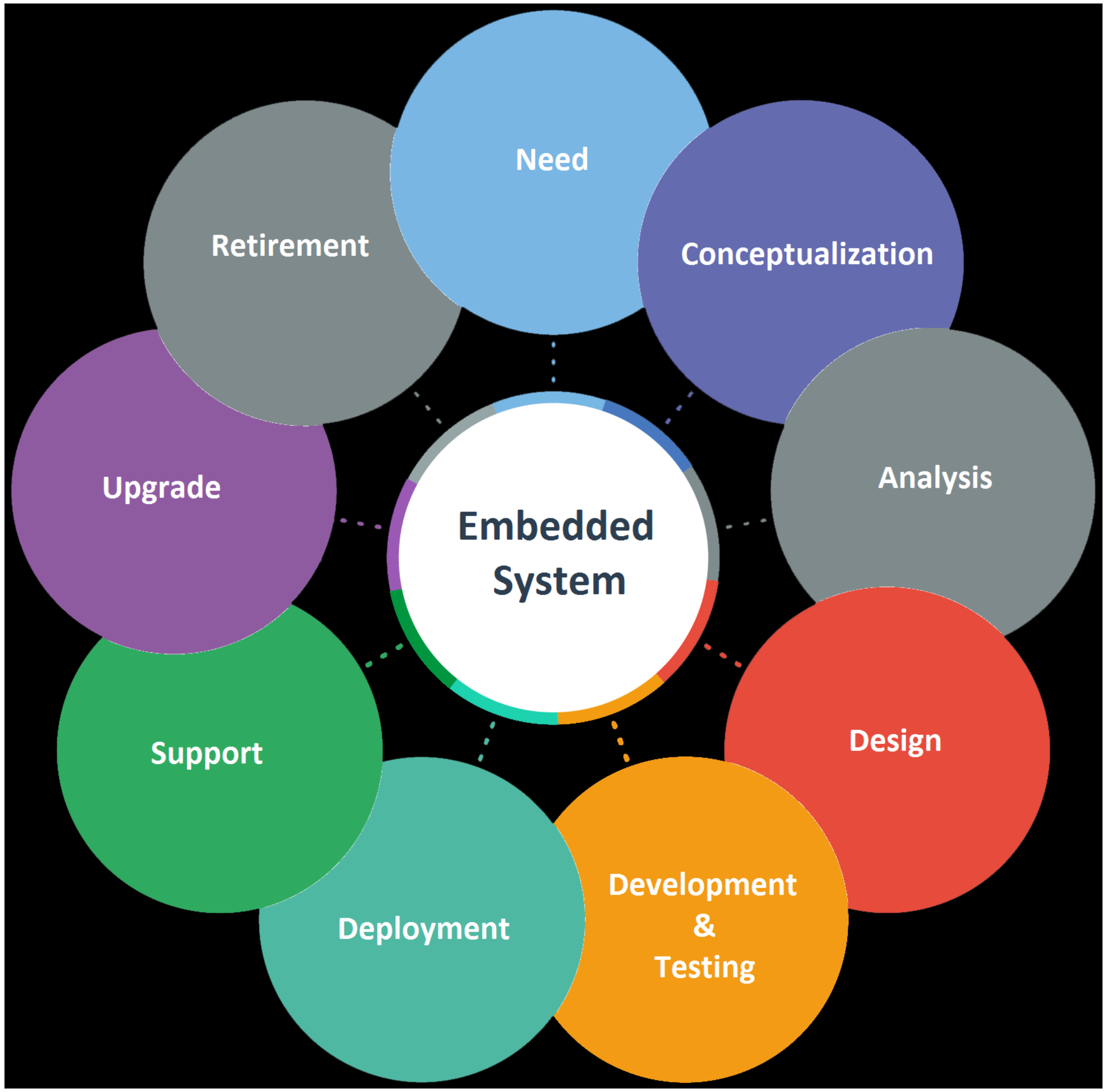
Embedded Firmware –

Round Robin Preemptive Scheduling



SERIAL NO.	ADVANTAGES	DISADVANTAGES
1.	There is fairness since every process gets equal share of CPU.	There is Larger waiting time and Response time.
2.	The newly created process is added to end of ready queue.	There is Low throughput.
3.	A round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum.	There is Context Switches.
4.	While performing a round-robin scheduling,a particular time quantum is alloted to different jobs.	Gantt chart seems to come too big (if quantum time is less for scheduling.For Example:1 ms for big scheduling.)
5.	Each process get a chance to reschedule after a particular quantum time in this scheduling.	Time consuming scheduling for small quantums .

Embedded Systems Development Cycle



Embedded Firmware – Debugging

BORIS BEIZER

"Testing proves a programmer's failure.
Debugging is the programmer's vindication."

Testing != Debugging

Two related goals: Are there any software defects? (if so, where)
We think the software is good to go –are we confident no defects remain?

Testing: Executing a program to see if it performs as expected

Debugging: Given a symptom of software failure, locate and correct the defect

Unpleasant truths: Testing tells you how buggy your software is, ***NOT*** where ***ALL*** the bugs are

It's still useful know how to remove bugs

Embedded Firmware – Debugging

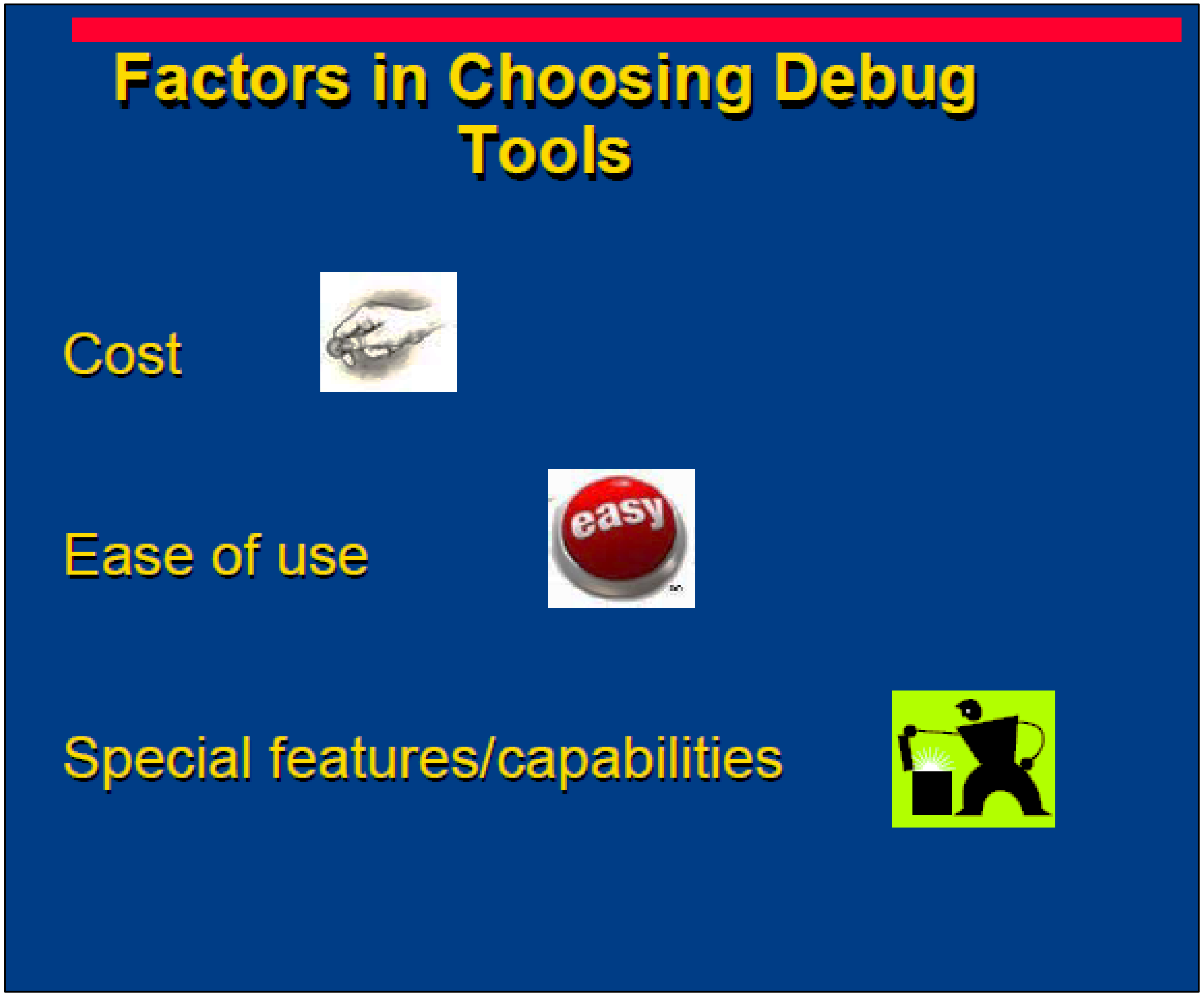
- Developing an embedded application requires hardware design, software coding and programming a system that is subject to real-world interactions.

- Hardware and software component must work together in an effective design.

- A debug tool can:
    1. help bring a prototype system up.

    2. help identify hardware and software problems both in the prototype and final application.

    3. assist in fine-tuning the system.



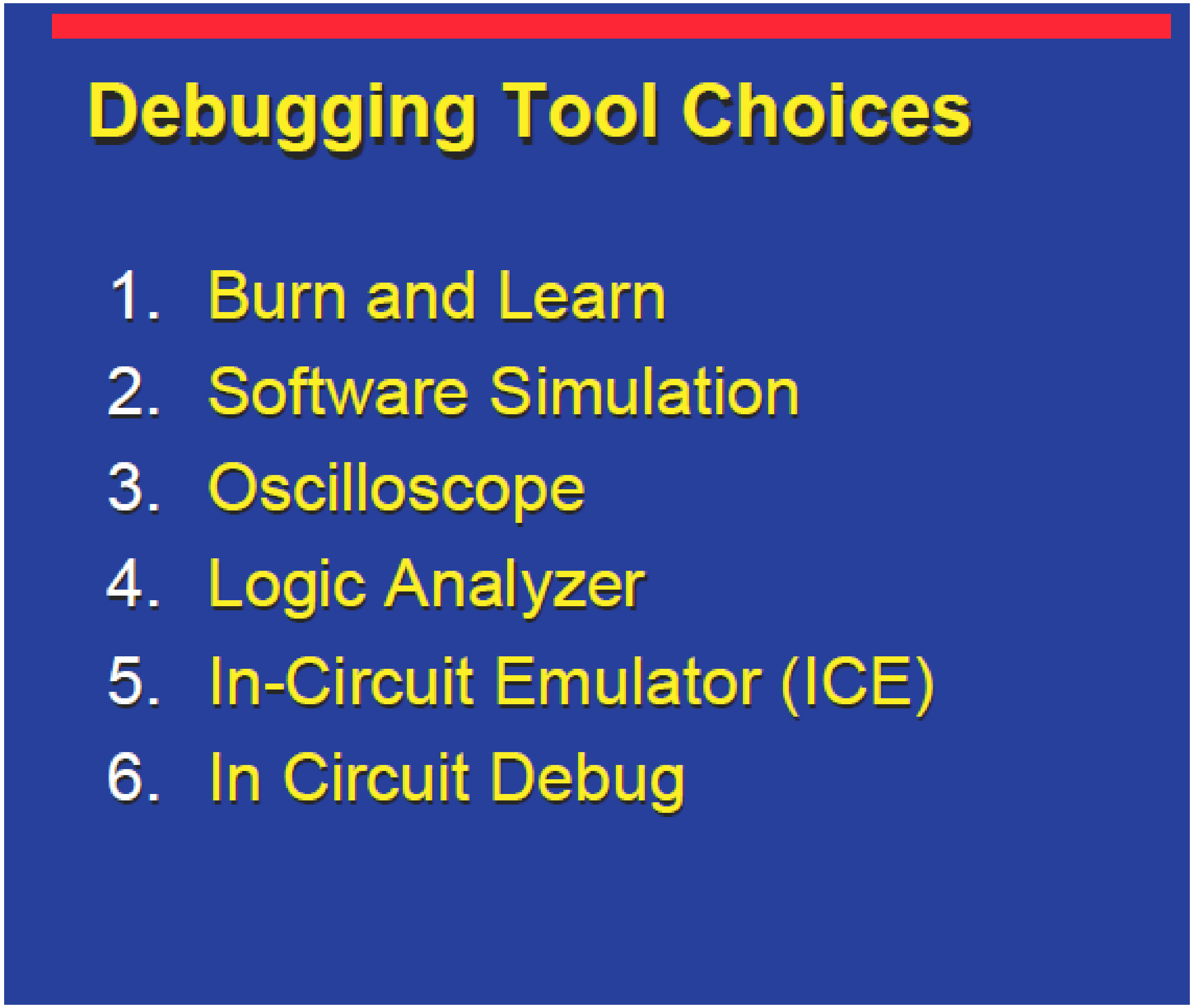
Factors in Choosing Debug Tools

Cost

Ease of use

Special features/capabilities

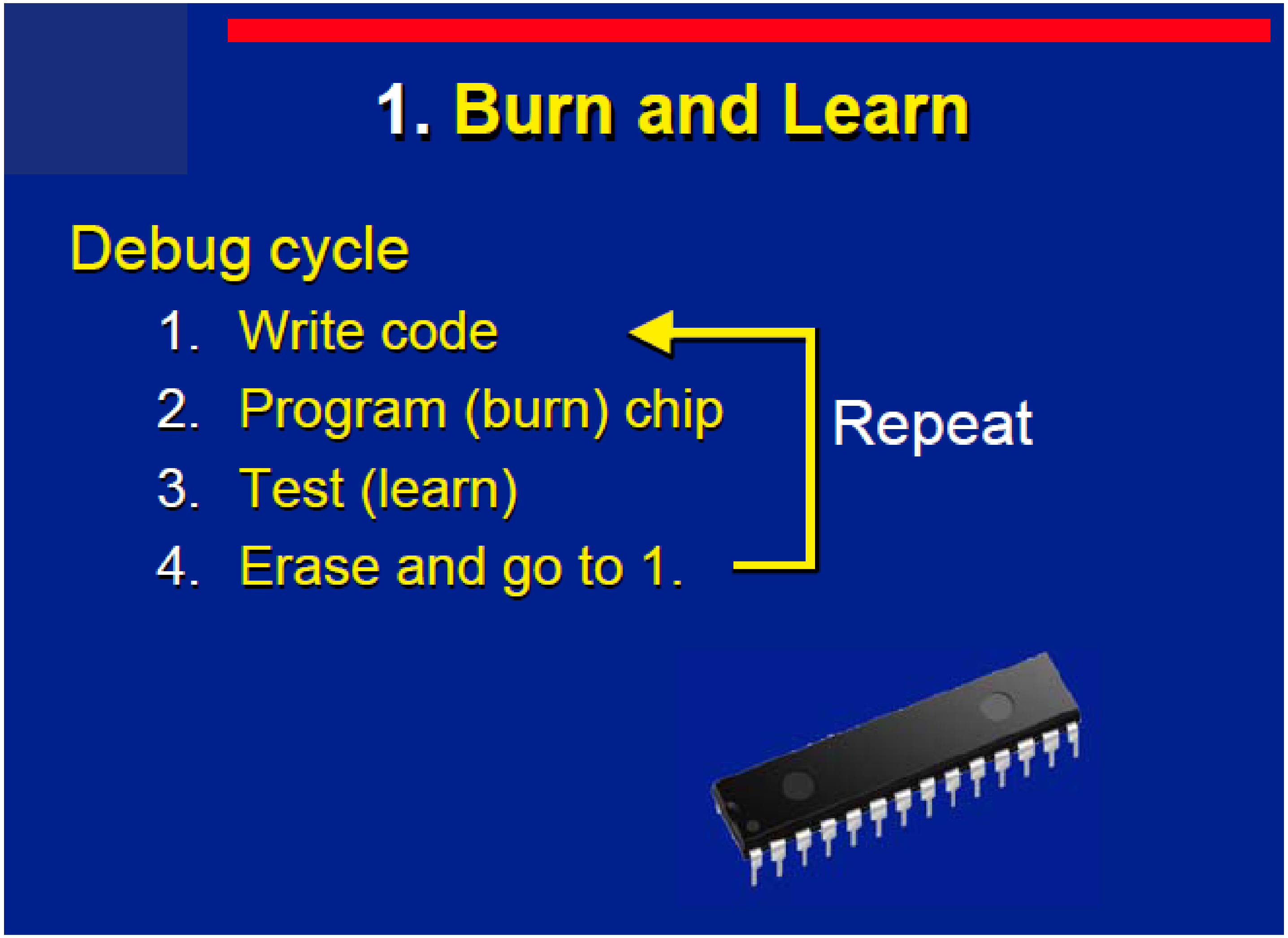
Embedded Firmware – Debugging



Debugging Tool Choices

1. Burn and Learn
2. Software Simulation
3. Oscilloscope
4. Logic Analyzer
5. In-Circuit Emulator (ICE)
6. In Circuit Debug

Embedded Firmware – Debugging

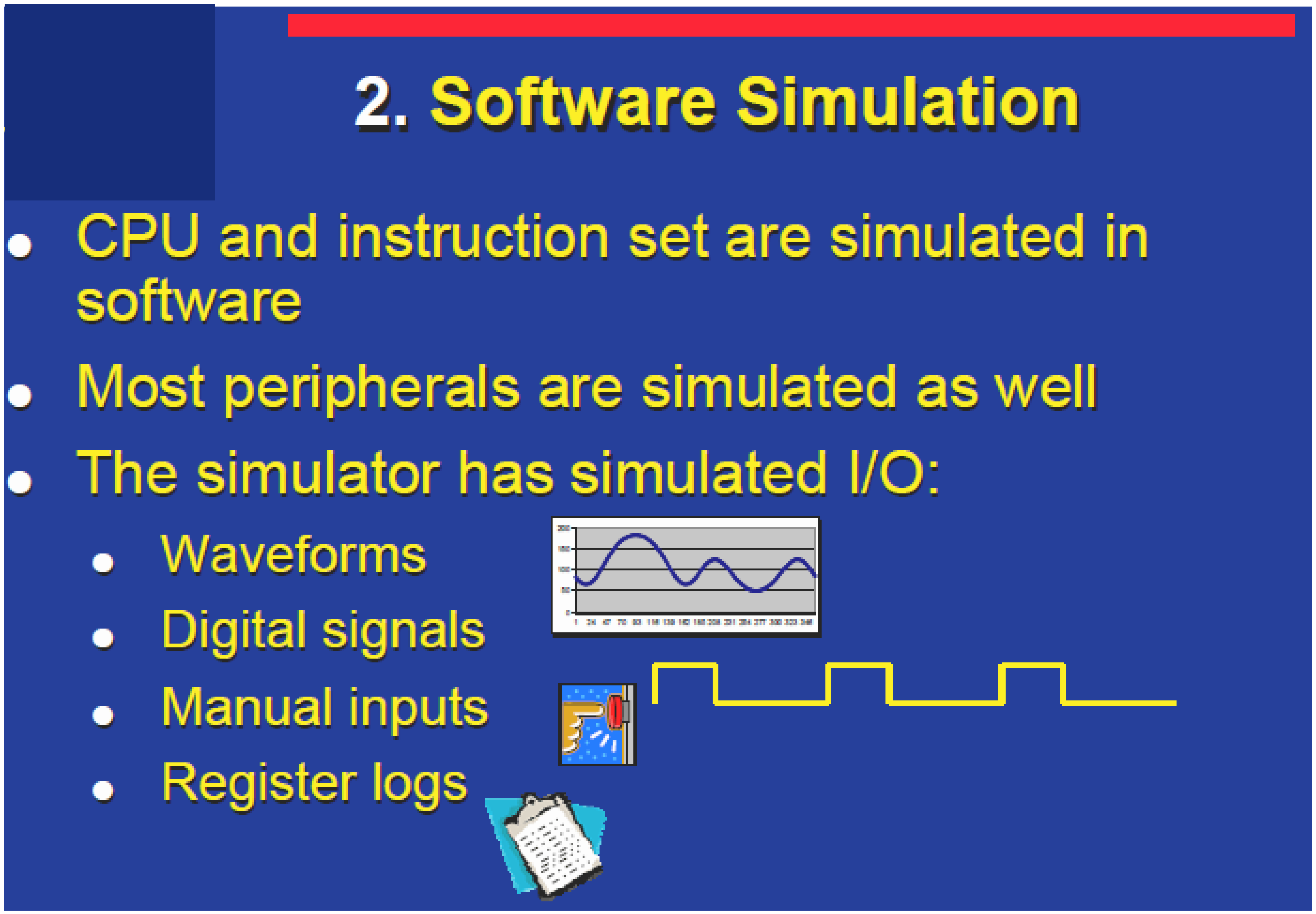


1. Burn and Learn

Debug cycle

1. Write code
2. Program (burn) chip
3. Test (learn)
4. Erase and go to 1.

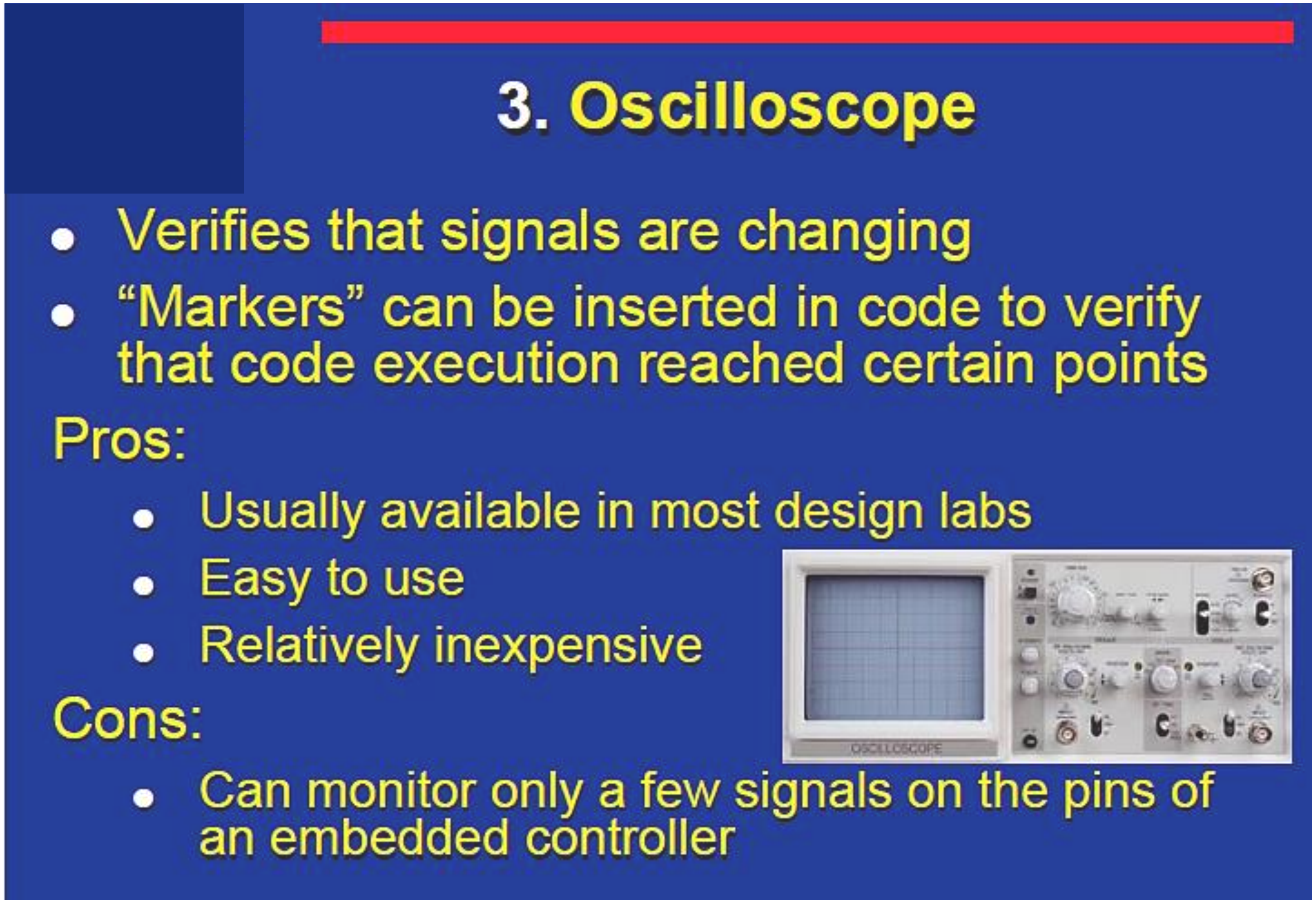
Embedded Firmware – Debugging



2. Software Simulation

- CPU and instruction set are simulated in software
- Most peripherals are simulated as well
- The simulator has simulated I/O:
  - Waveforms
  - Digital signals
  - Manual inputs
  - Register logs

Embedded Firmware – Debugging



3. Oscilloscope

- Verifies that signals are changing
- “Markers” can be inserted in code to verify that code execution reached certain points

Pros:

- Usually available in most design labs
- Easy to use
- Relatively inexpensive

Cons:

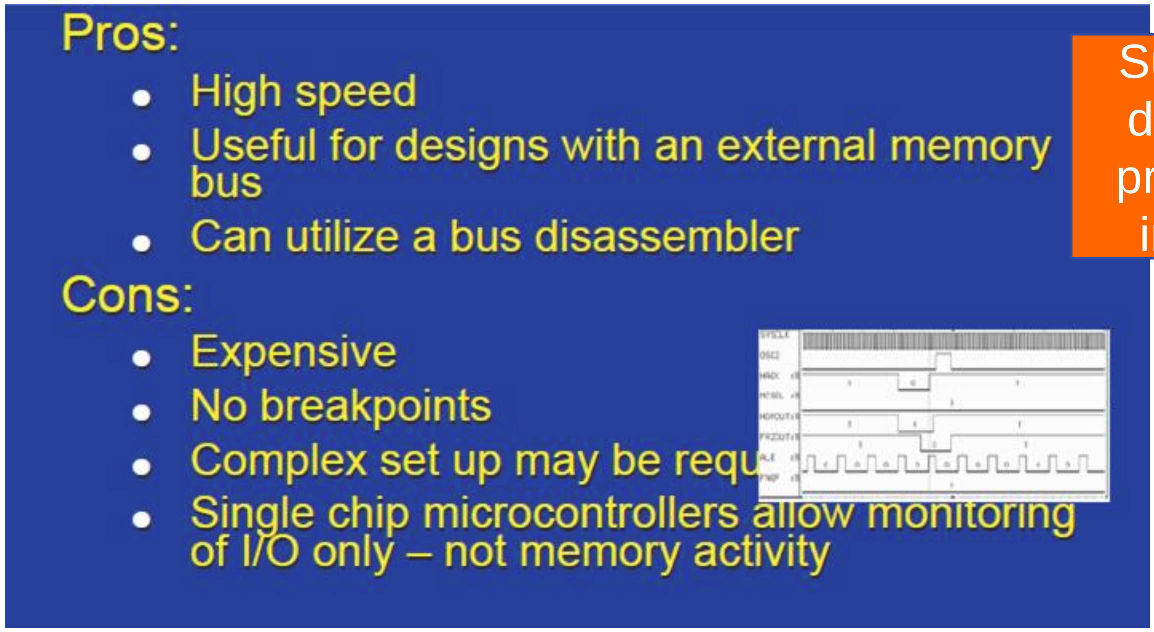
- Can monitor only a few signals on the pins of an embedded controller

University of Glasgow

Embedded Firmware – Debugging

4. Logic Analyzer

A logic analyzer is a hardware test instrument that is used to monitor digital signals. It can be a very good tool for debugging an embedded system with an external memory bus. A logic analyzer can log address, instruction and some data information as it executes.



Pros:

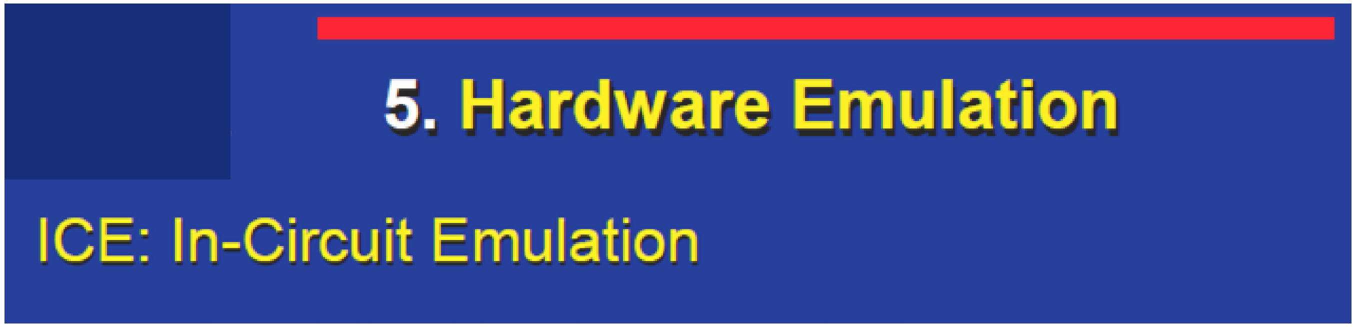
- High speed
- Useful for designs with an external memory bus
- Can utilize a bus disassembler

Cons:

- Expensive
- No breakpoints
- Complex set up may be requ
- Single chip microcontrollers allow monitoring of I/O only – not memory activity

Suitable for debugging problems in interrupts

Embedded Firmware – Debugging

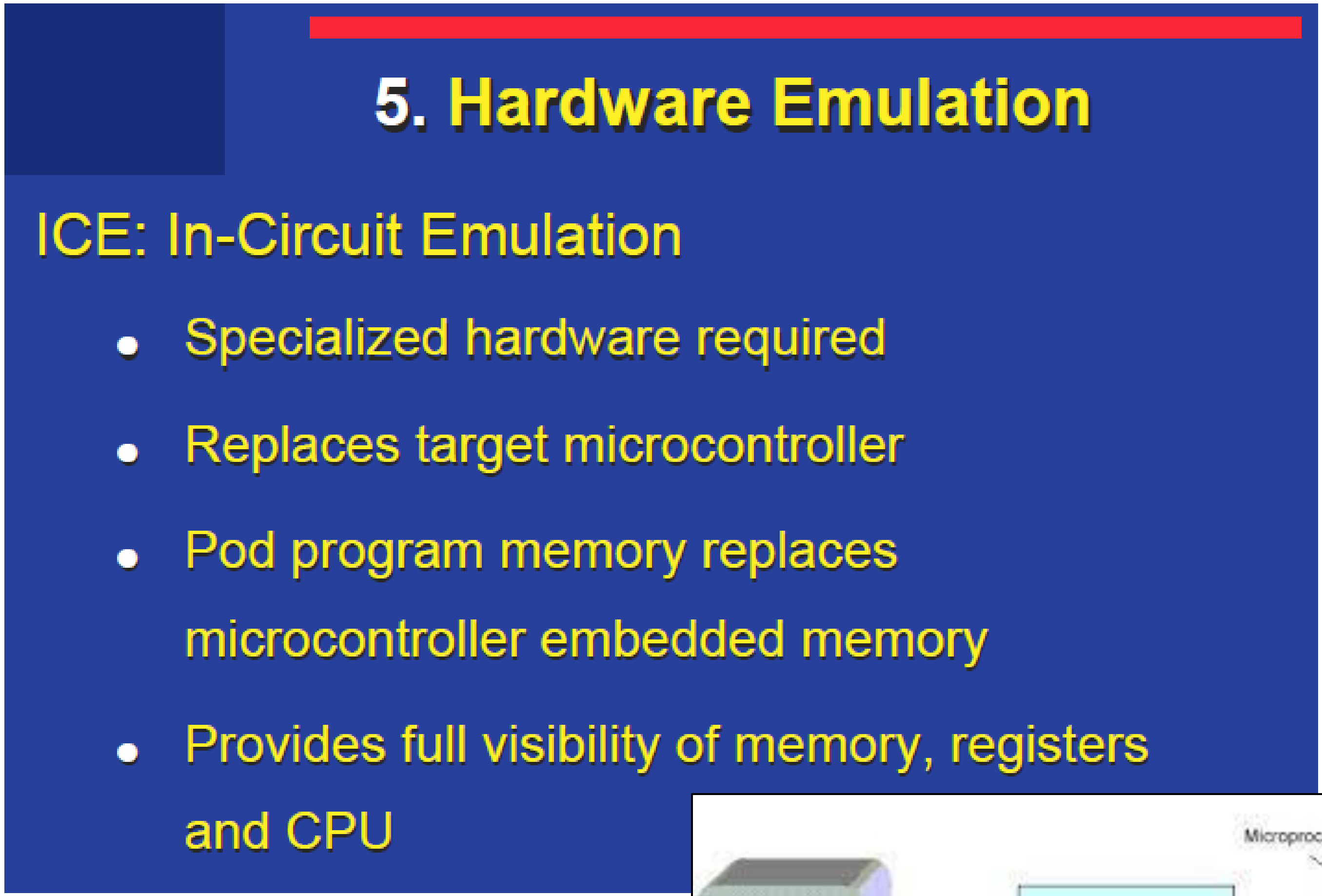


5. Hardware Emulation

ICE: In-Circuit Emulation

- In-Circuit Emulator (ICE) takes the place of the target processor.
- It contains a copy of target processor, plus RAM, ROM, and its own embedded software.
- It allows you to examine the state of the processor while the program is running.
- It uses the remote debugger for human interface.
- It supports software and hardware breakpoints.
- It has real-time tracing.
- It stores the information about each processor cycle which is executed.
- It allows you to see in what order things happen.

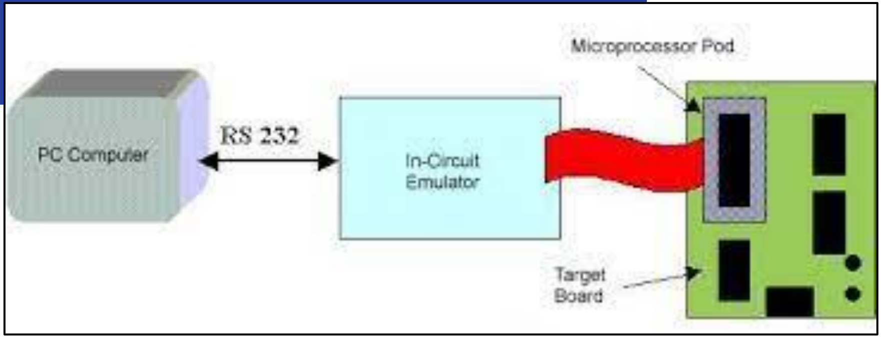
Embedded Firmware – Debugging



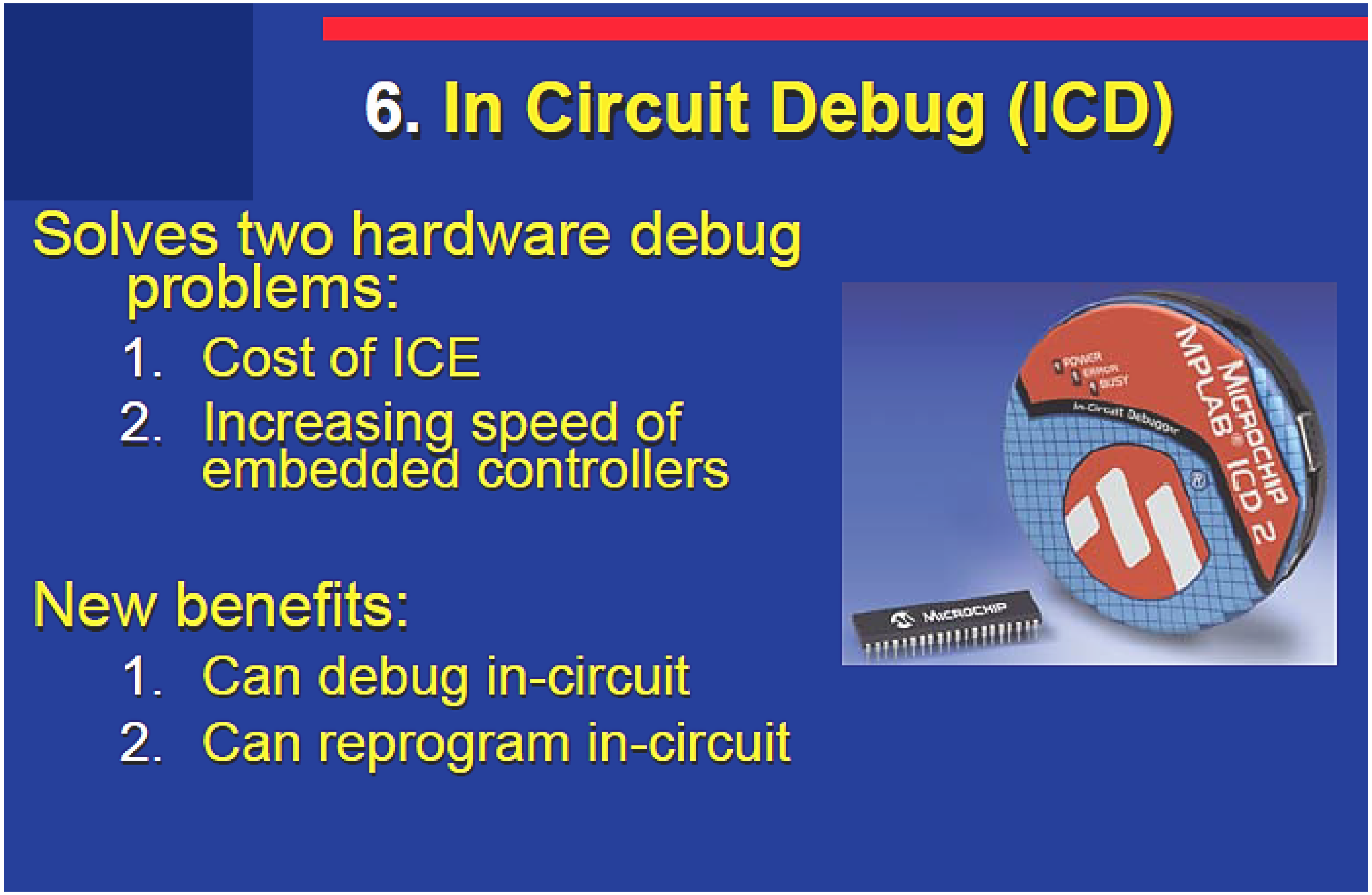
5. Hardware Emulation

ICE: In-Circuit Emulation

- Specialized hardware required
- Replaces target microcontroller
- Pod program memory replaces microcontroller embedded memory
- Provides full visibility of memory, registers and CPU



Embedded Firmware – Debugging



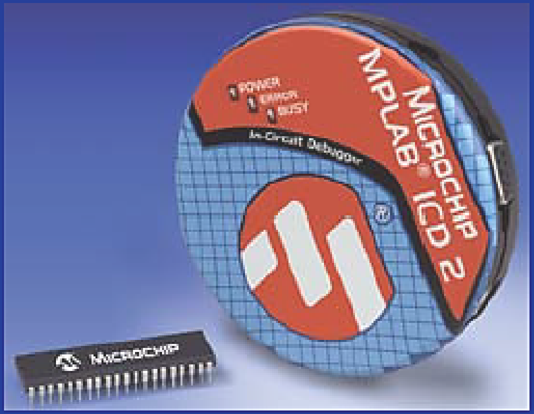
6. In Circuit Debug (ICD)

Solves two hardware debug problems:

1. Cost of ICE
2. Increasing speed of embedded controllers

New benefits:

1. Can debug in-circuit
2. Can reprogram in-circuit



Embedded Firmware – Debugging

µVision® Debugger

The µVision debugger provides a single environment in which you may test, verify, and optimize your application code.
The debugger includes traditional features like simple and complex breakpoints, watch windows, and execution control and provides full visibility to device peripherals.