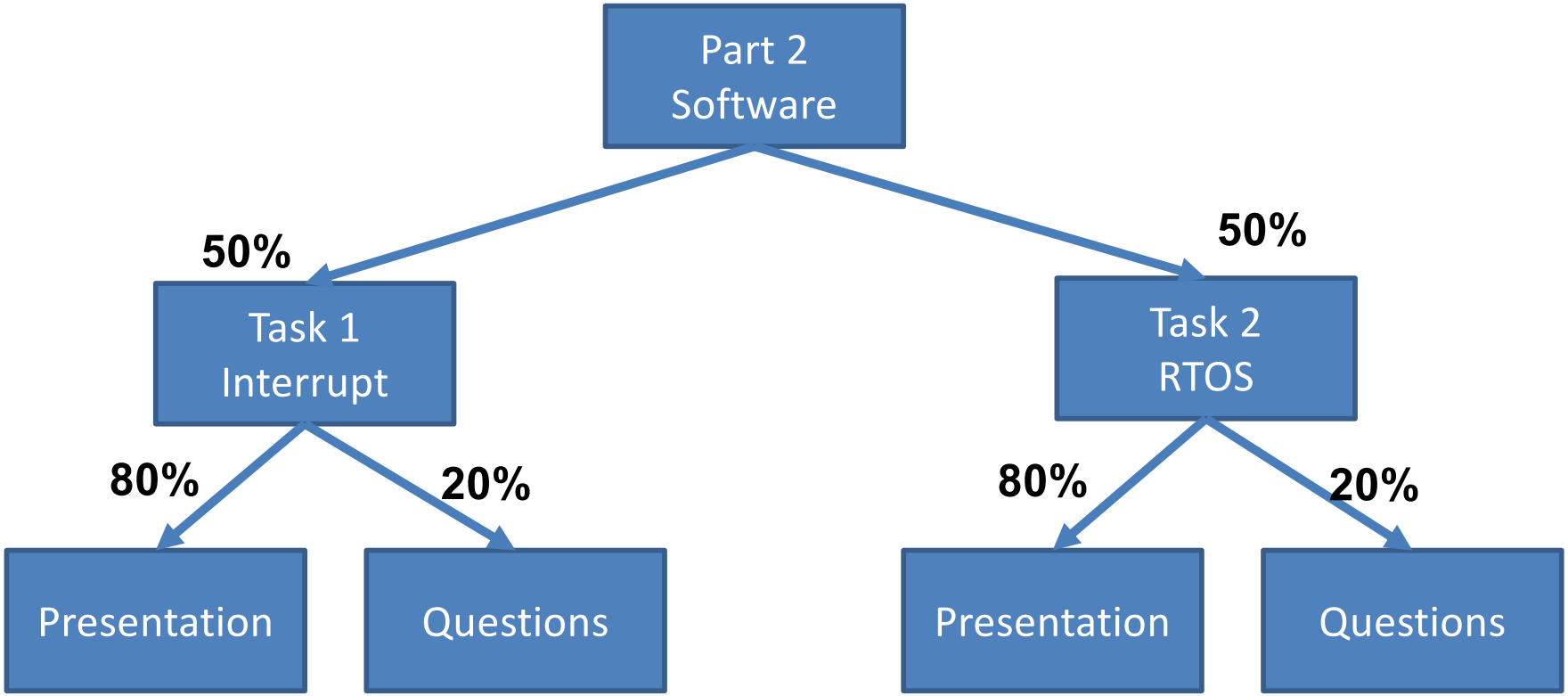
Embedded and Real Time Systems
Part 2

Prof. Reza Nejabati

Introduction to Part 2

Assessment



- Presentations
  - Demo
  - PowerPoint presentation
  - Rubric in handbook
- Questions
  - Testing understanding of the important concepts

All marked in the lab sessions

Timetable

Week	14/Nov	21/Nov	28/Nov	5/Dec	12/Dec
Monday (10am-1pm)	Lab session 1 (Task 1)	Lab session 3 (Task 1)	Lab session 4 (Task 2)	Lab session 6 (Task 2)	Extra session
Friday (10am-1pm)	Lab session 2 (Task 1)	Assessment (Task1)	Lab session 5 (Task 2)	Assessment (Task2)	Extra session

Lecturer & Assistants

- **Prof. Reza nejabati**
  - **Open office : On request, always & anytime & any date ,**
  - **Room MVB 4.19**
  - **Please email : reza.nejabati@bristol.ac.uk**

- **Assistants**

  - **Elliott Worsey**
  - **Zichao Shen**
  - **Kaan Olgu**
  - **Victor Marot -**
  - **Florence Hobbs**
  - **Rui Wang**

Delivery - Problem based learning

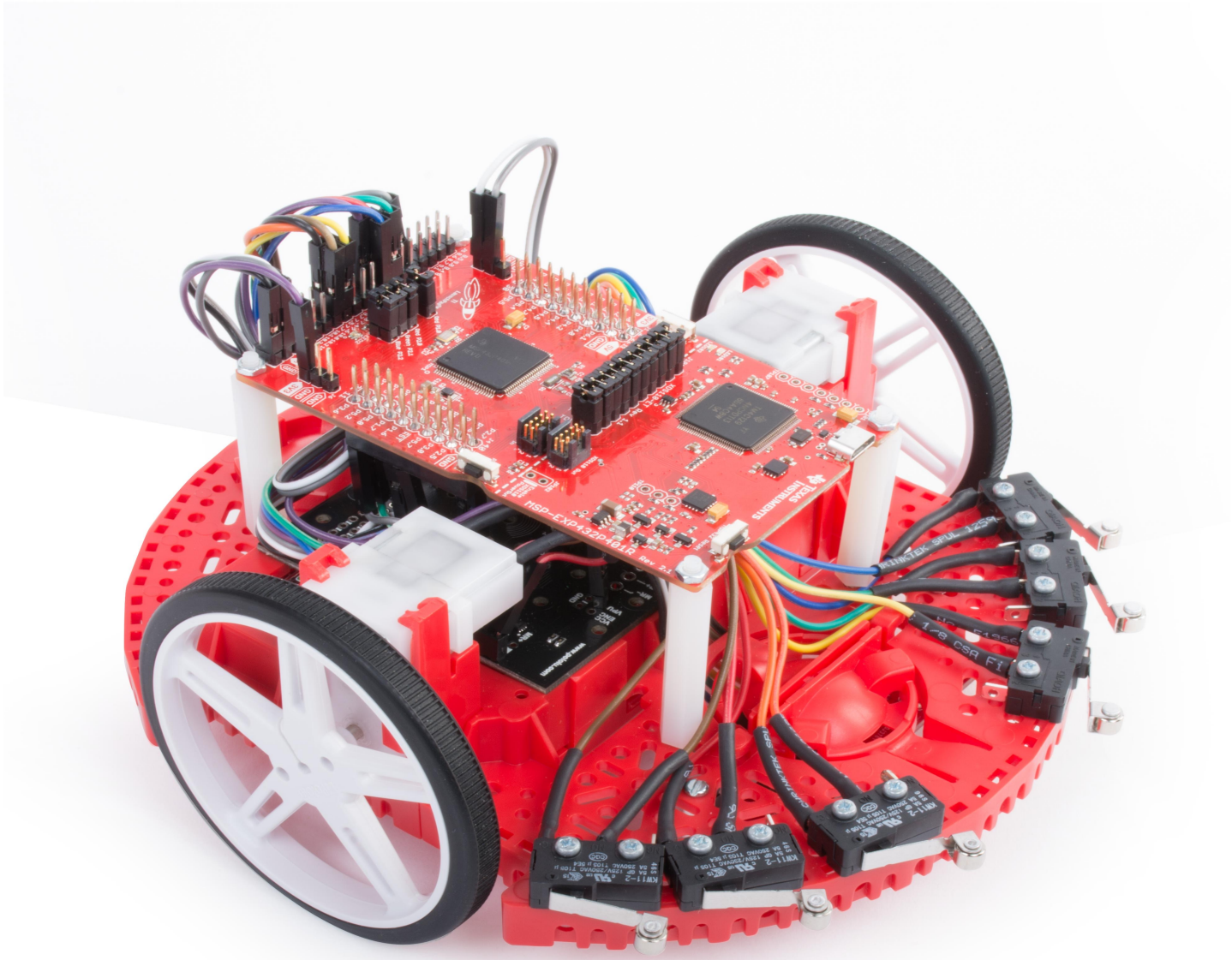
- What is it?
  - Learning objectives presented as unknown aspects of case-studies
  - Combines collaborative and self-directed learning
  - Progress is facilitated by tutors

- How should you approach this?
  - Books
  - Data sheets
  - Manuals
  - The internet

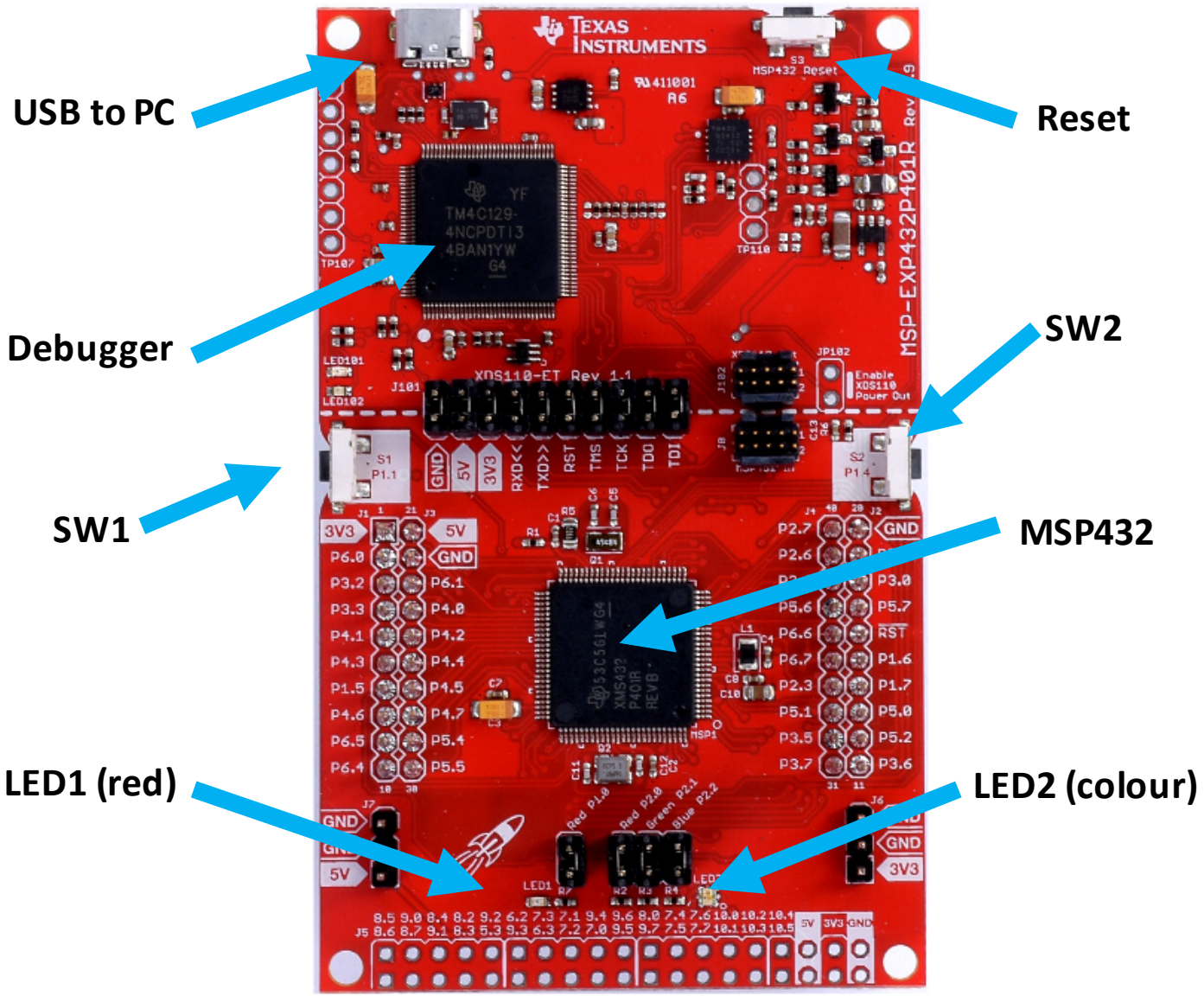
- Why?
  - Representative of how a real project may be presented

Hardware Platform

Texas Instruments Robotic System Learning Kit, TI-RSLK

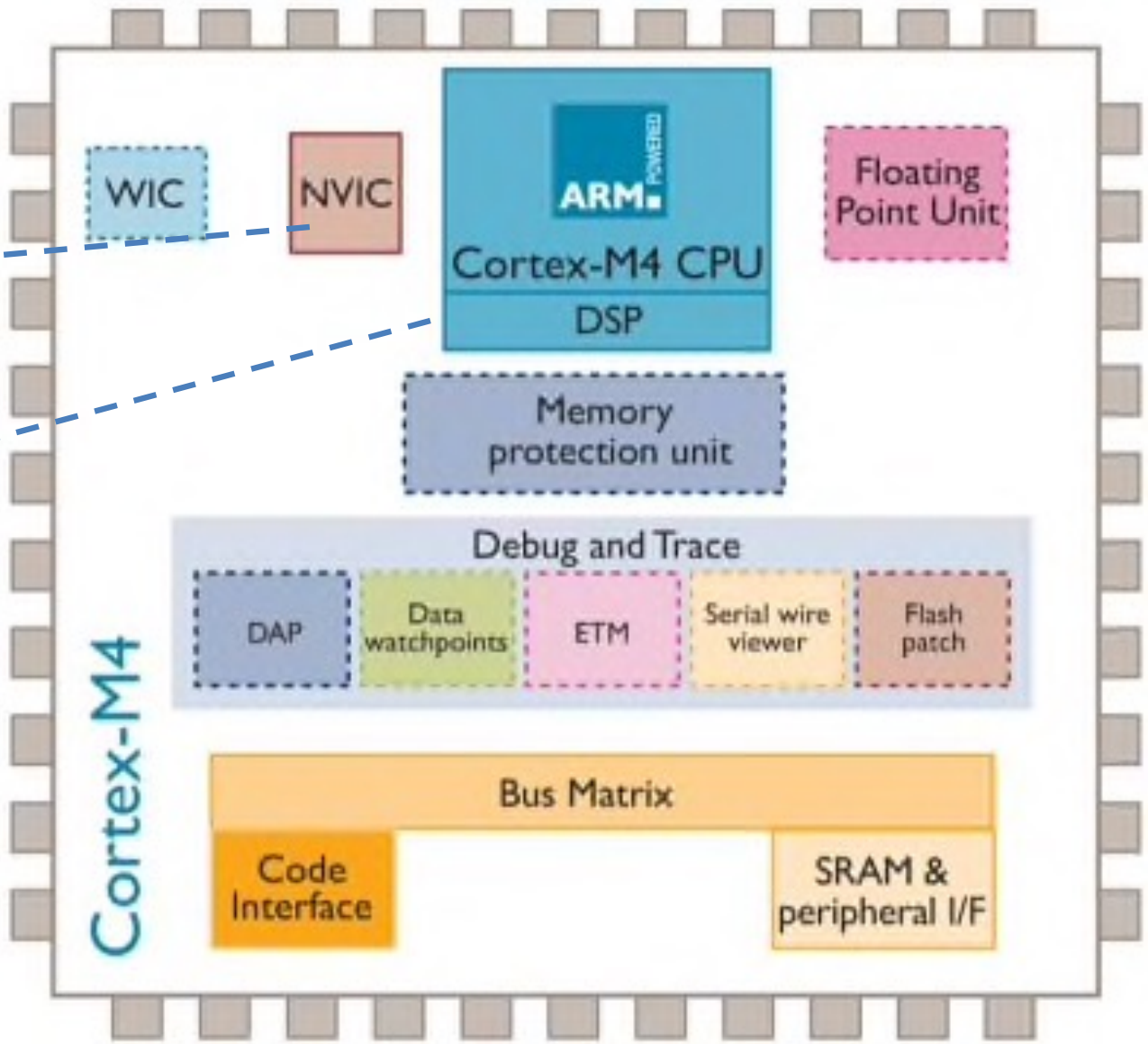


ARM 32-bit Cortex-M4F microcontroller

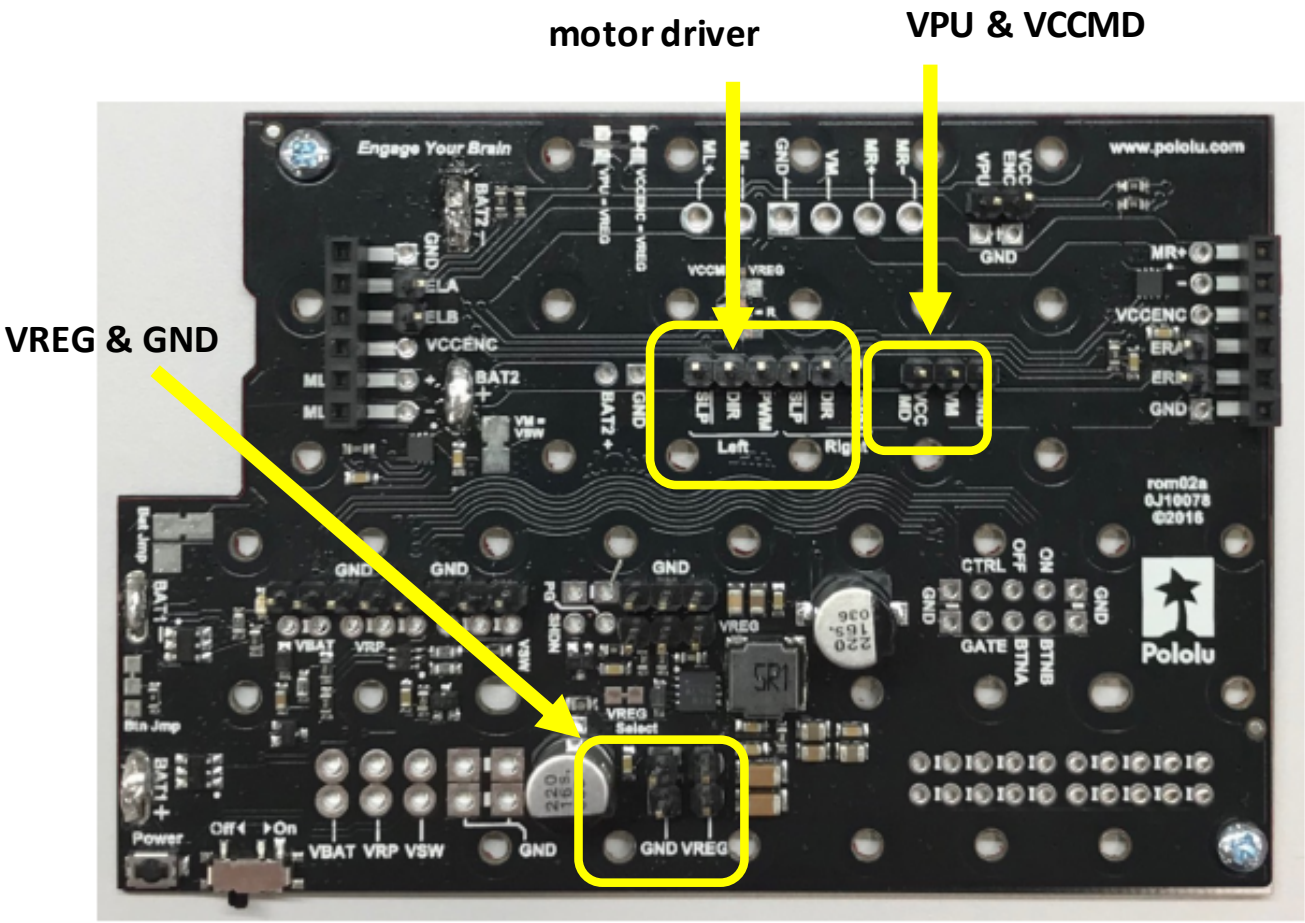


Nested Vector Interrupt Controller

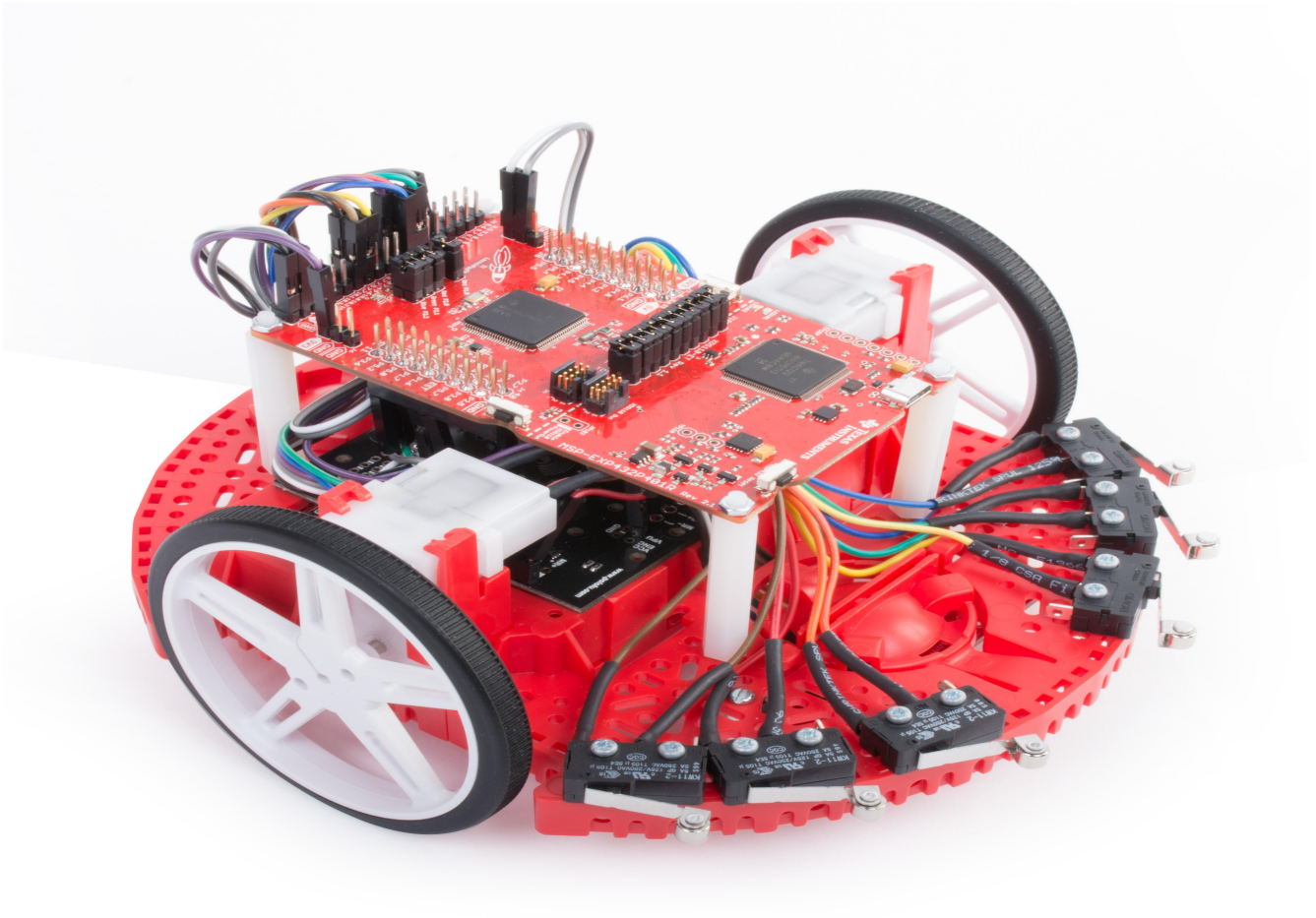
ARM Processor Core



Motor Drive



Programming using Interrupts

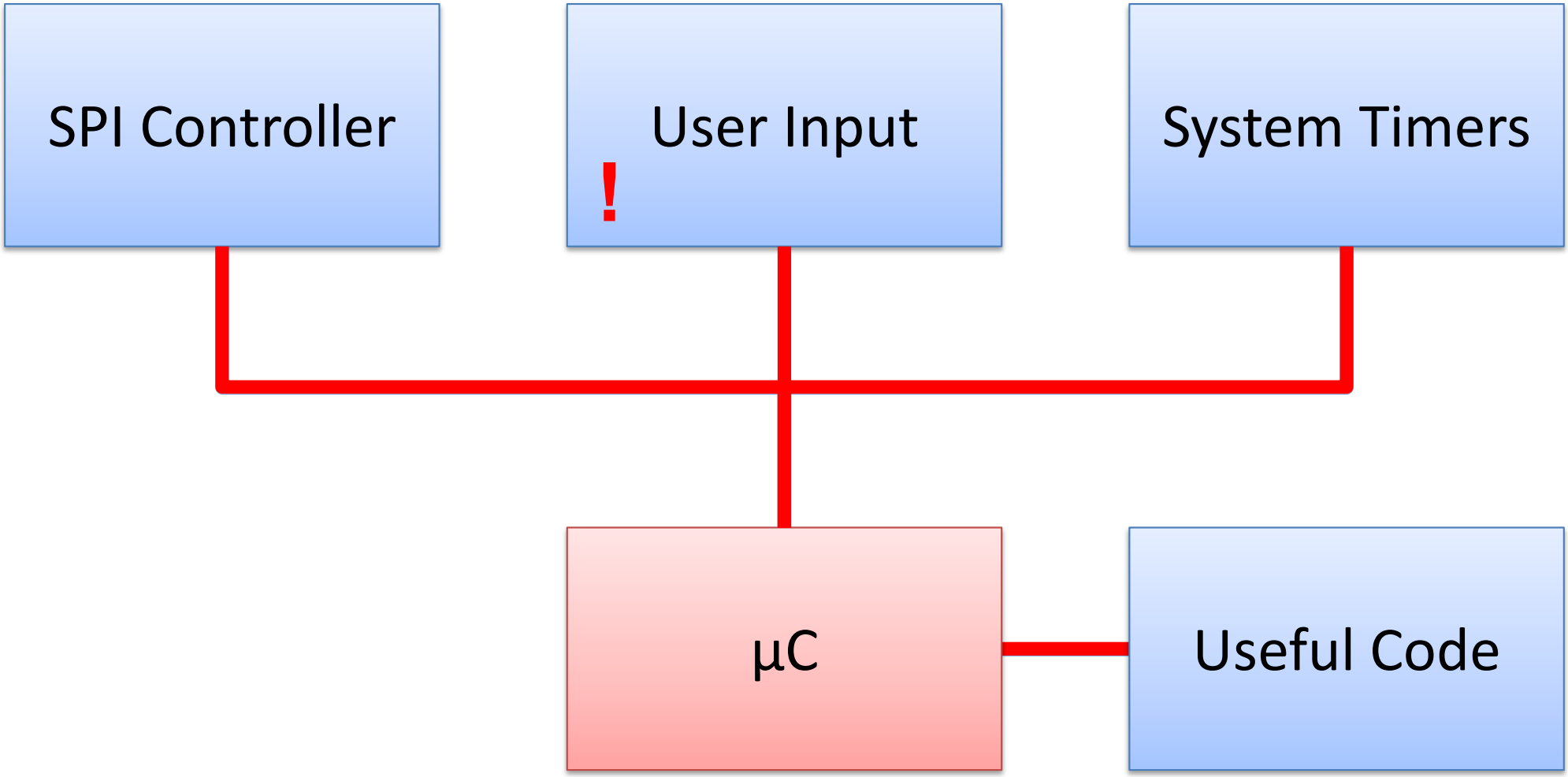


How can a microcontroller serve several devices?

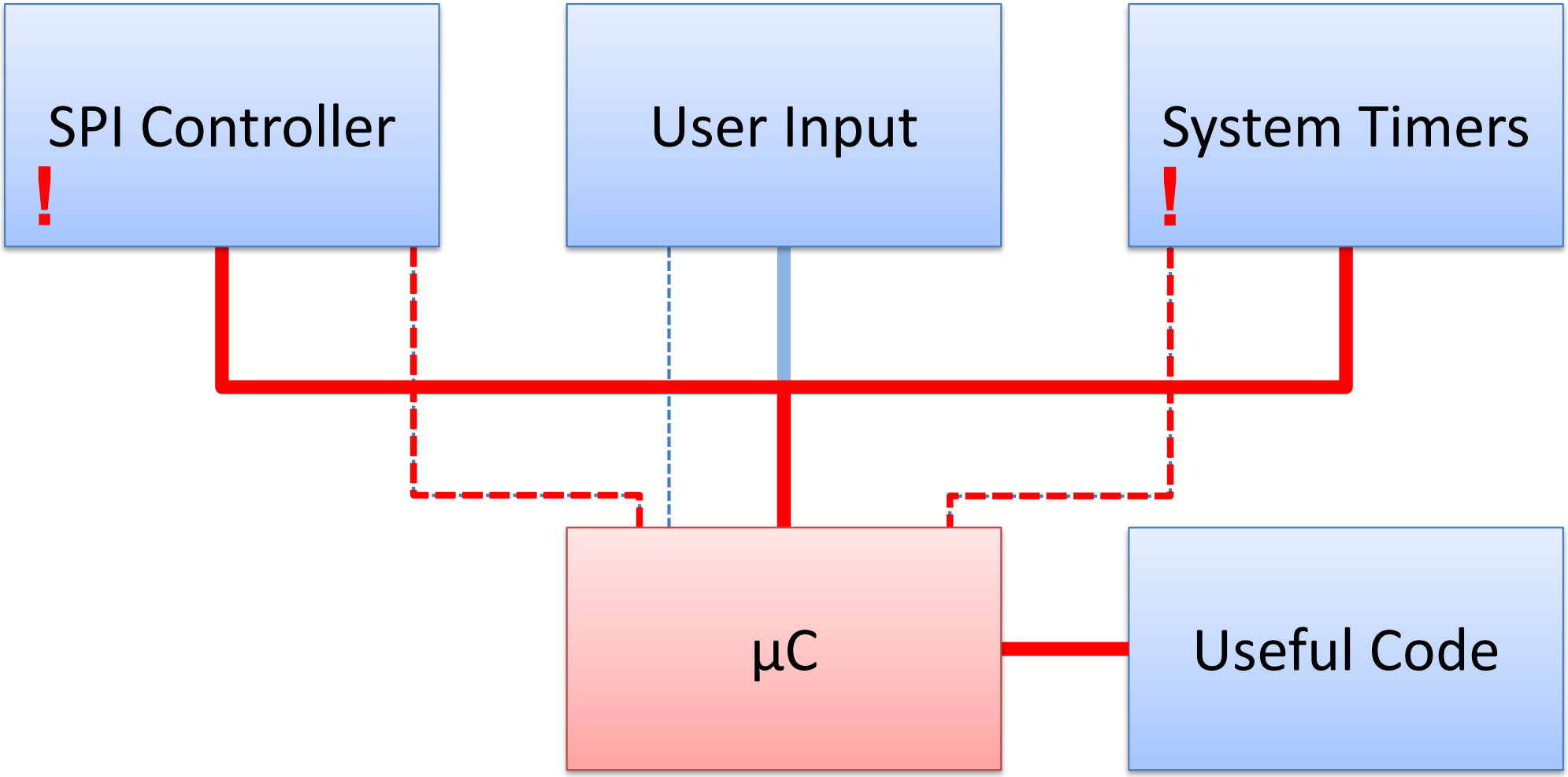
- Polling
  - The microcontroller continuously monitors the status of a given device and when the status condition is met, it performs the service

- Interrupts
  - When a device needs service it notifies the microcontroller by sending an interrupt signal
  - On receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device by running a certain part of the program

Example of polling



Example of interrupts



Polling vs. Interrupts

- Polling
  - Program is tied up waiting for flag
  - Inefficient use of processor

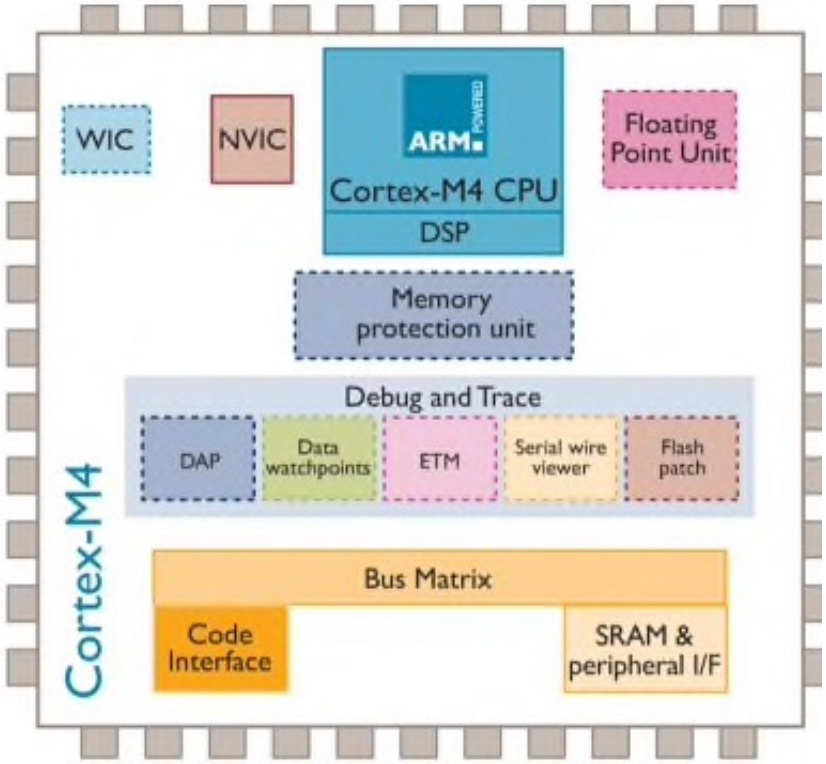
- Interrupts
  - Program can be executing other tasks
  - Efficient use of processor
  - Many devices can be served
  - Service according to device priority
  - Processor can ignore (mask) a device request for service

e.g. Lecturer asking each one of you to see if you have any questions on a slide before continuing to the next slide

e.g. Lecturer continues to the next slide unless someone raises their hand indicating that they have a question on the slide

How do interrupts happen?

- Interrupts start with a signal from the hardware
- Most I/O chips have a pin that they assert when they require service
- This pin interfaces to an input pin on the microcontroller called an *interrupt request or IRQ,* notifying it that a chip wants attention
- With System-on-Chip (SoC) devices (like the one we use in the LAB) where I/O devices are integrated in one chip the IRQs are integrated with the microcontroller and not accessible outside
- For external devices there are external IRQ pins

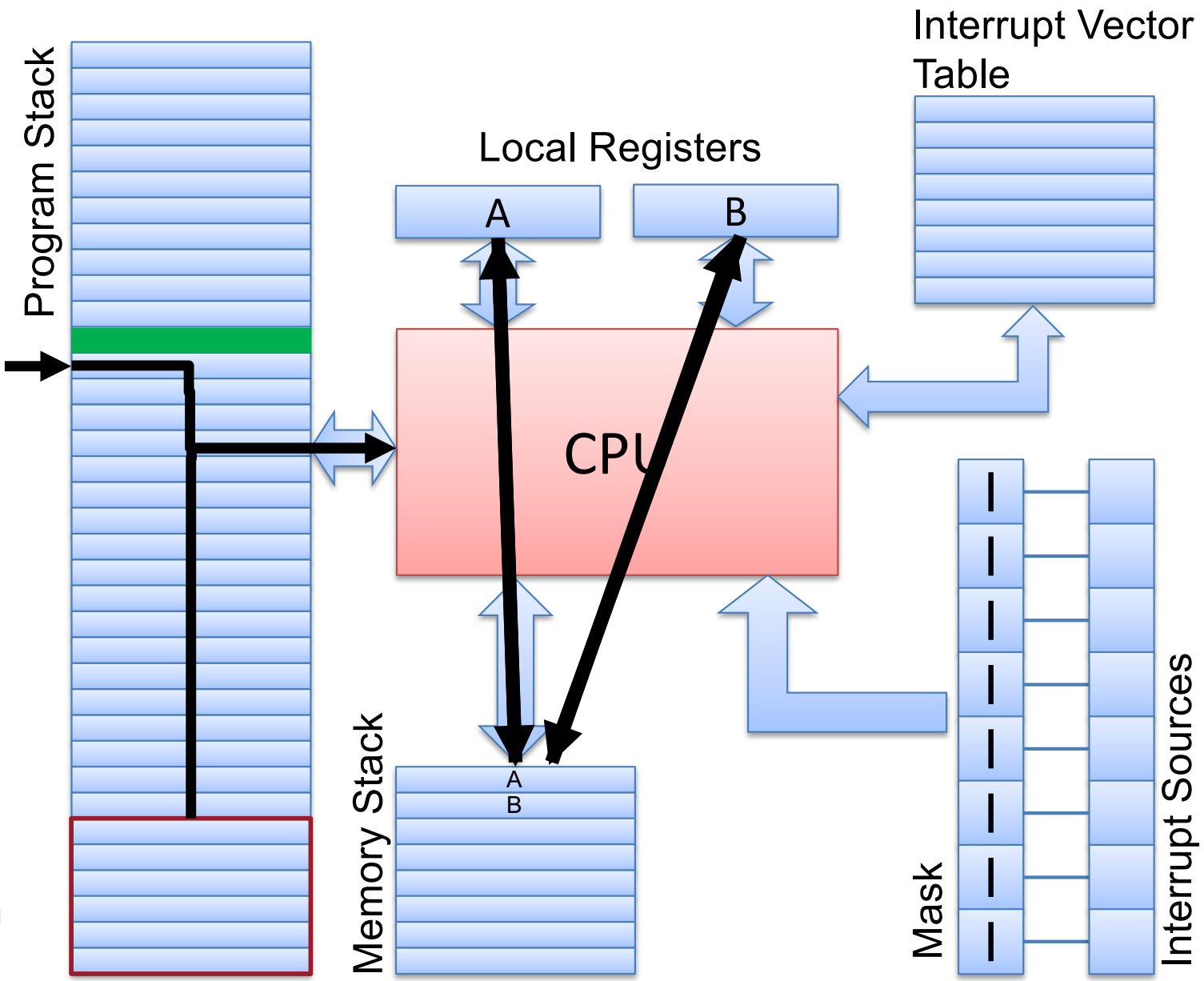


What happens when an interrupt occurs?

- For every interrupt, there is an **interrupt service routine** (ISR), or handler written by the programmer
- Interrupts have to be enabled/disabled globally and locally by the programmer so that any interrupts happenings (apart from exceptions and resets) are anticipated by the programmer
- For every interrupt, there is a fixed location in memory that holds the starting address of its ISR
- The group of memory locations set aside to hold the addresses of ISRs is called the **interrupt vector table**

What happens when an interrupt occurs?

- Initialise interrupt vector table with ISR address
- Enable maskable interrupts
- Interrupt signal occurs
- Complete current instruction
- Calculate return address
- Store key registers on stack
- Fetch highest priority interrupt vector
- Transfer program control to ISR
- Clear interrupt source
- Restore registers
- Restore main program control



Maskable vs. non-maskable interrupts

- Maskable interrupts
  - Bus transaction completing
  - ADC read completed
  - Hardware signal
  - Timer signal
- Non-maskable interrupts
  - Reset
  - Power failure
  - Piano falling onto development platform

These can all be ignored by the processor

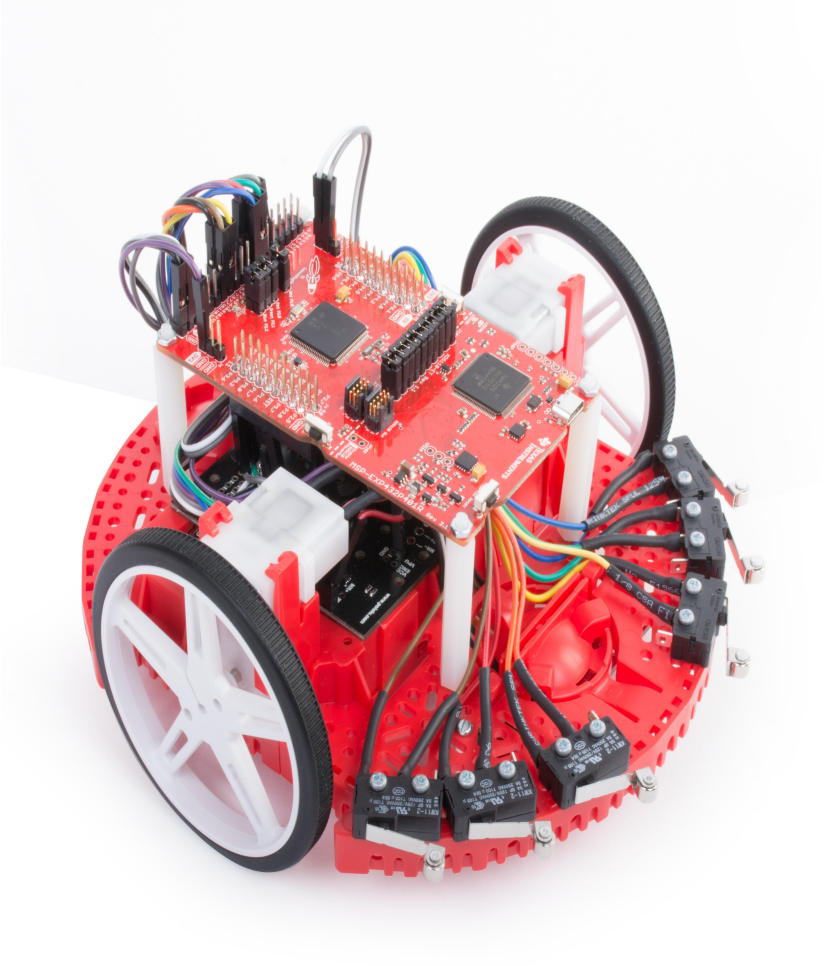
These cannot be ignored by the processor for various reasons

Some things to consider…

- If interrupts take place too often the overhead may become significant
- Processing during interrupts should be minimal
- Long ISRs reduce system responsiveness
- If nested ISRs are allowed then interrupt priority is used (only higher priority interrupts are allowed to interrupt current ISR)

Problem 1 description

- Develop a program that allows the robot that has two modes of operation:
  - **Autonomous Mode:** Upon pressing switchSW1, the robot operates Using predefined route and immediately stop when any of bump switches are touched.
  - **Free-motion mode:** upon pressing switchSW2, the robot freely move forward but will change direction of movement according to the interrupted bump switches.



- Use the coloured LED to represent the interrupted bump switches.

- Devise a way that allows the interrupt latency to be measured and compare it with the poling mode i.e. bump switches are monitored with polling

- **Ensure all of this can run at the same time simultaneously.**

Introduction to Real Time Operating Systems

What is a Real Time Operating System?

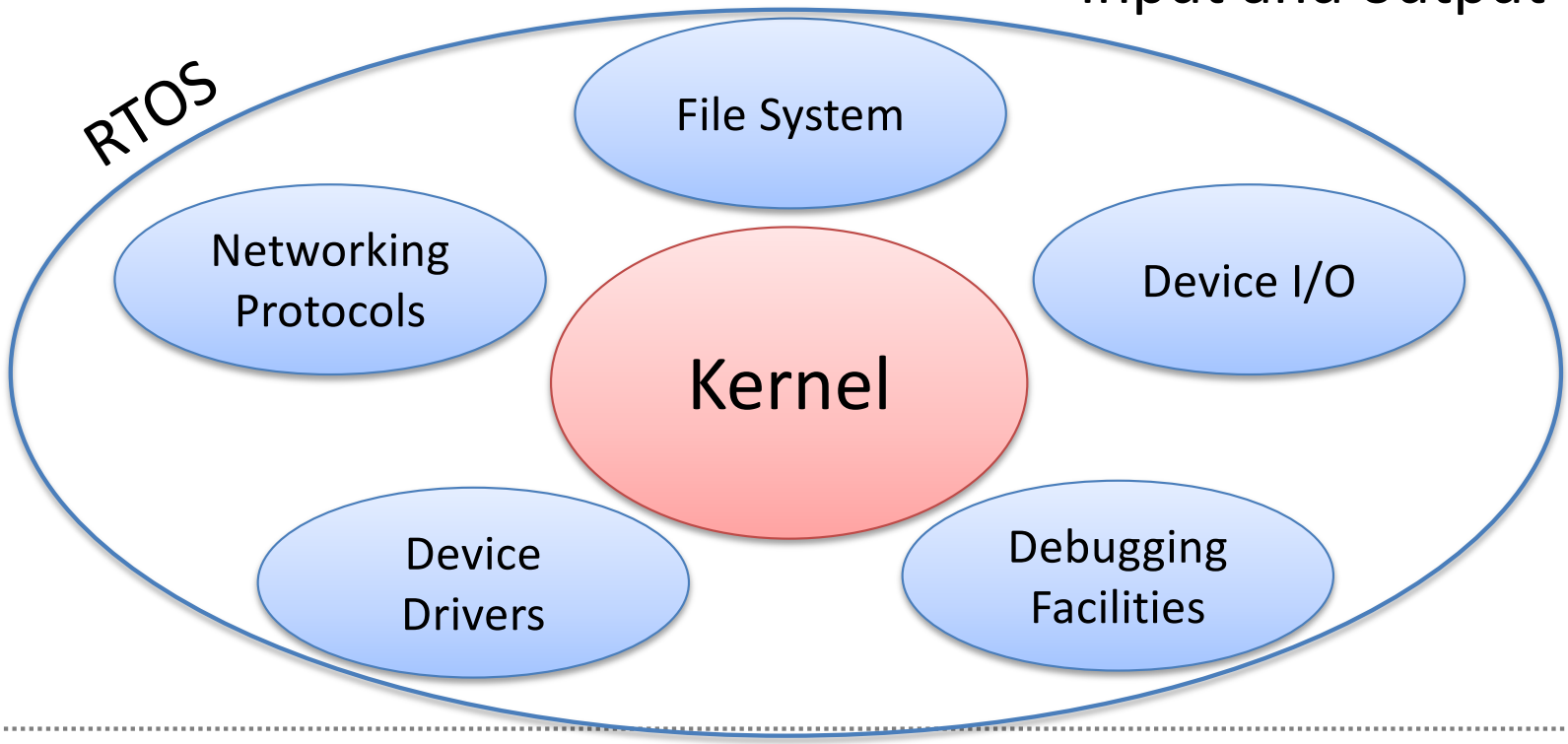
A program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code

What is a Real Time Operating System?

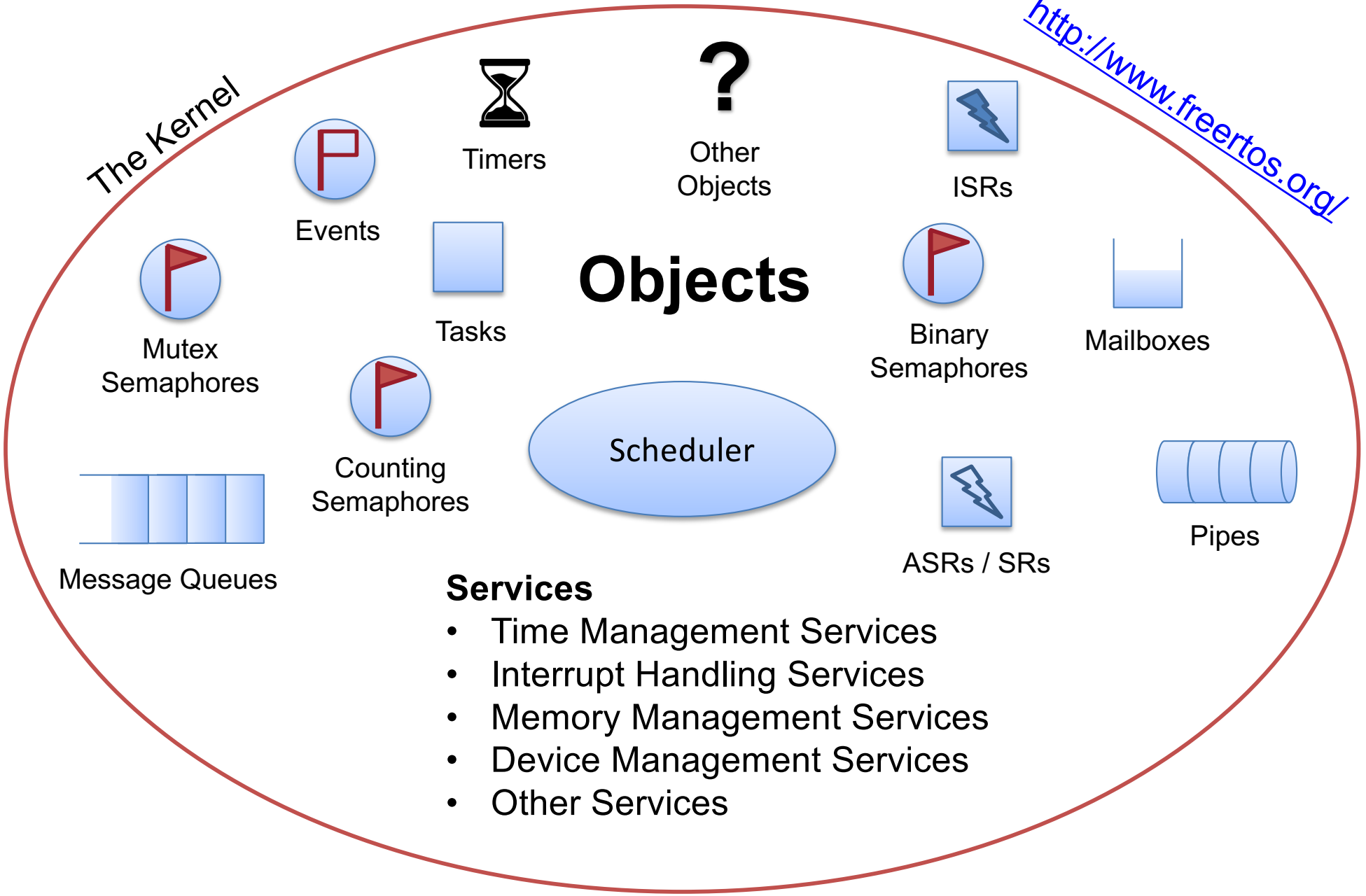
The kernel is a computer program that manages input/output requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer.

Basic responsibilities of a Microkernel:

- Scheduling of threads or processes
- Synchronization
- Input and output



What is a Real Time Operating System?



Why use a Real Time Operating System?

- Without an OS, multithreading is achieved with interrupts - timing is determined by external events
- Real-time operating systems support a variety of ways of controlling when threads execute (priorities, pre-emption policies, deadlines, …)
- Generic OSs (Linux, Windows, OSX, …)
  - Provide thread libraries and provide no fixed guarantees about when threads will execute.
  - Kernel is too feature reach
  - Kernel is not modular, fault-tolerant, configurable, modifiable, … .
  - Takes too much space
  - Not power optimized
  - Not designed for mission-critical applications

When to use a Real Time Operating System?

Reasons for using an RTOS

- When efficient scheduling is needed for multiple tasks with time constraints
- Task synchronisations is needed
- Interrupt latency control is essential

Reasons not to use an RTOS

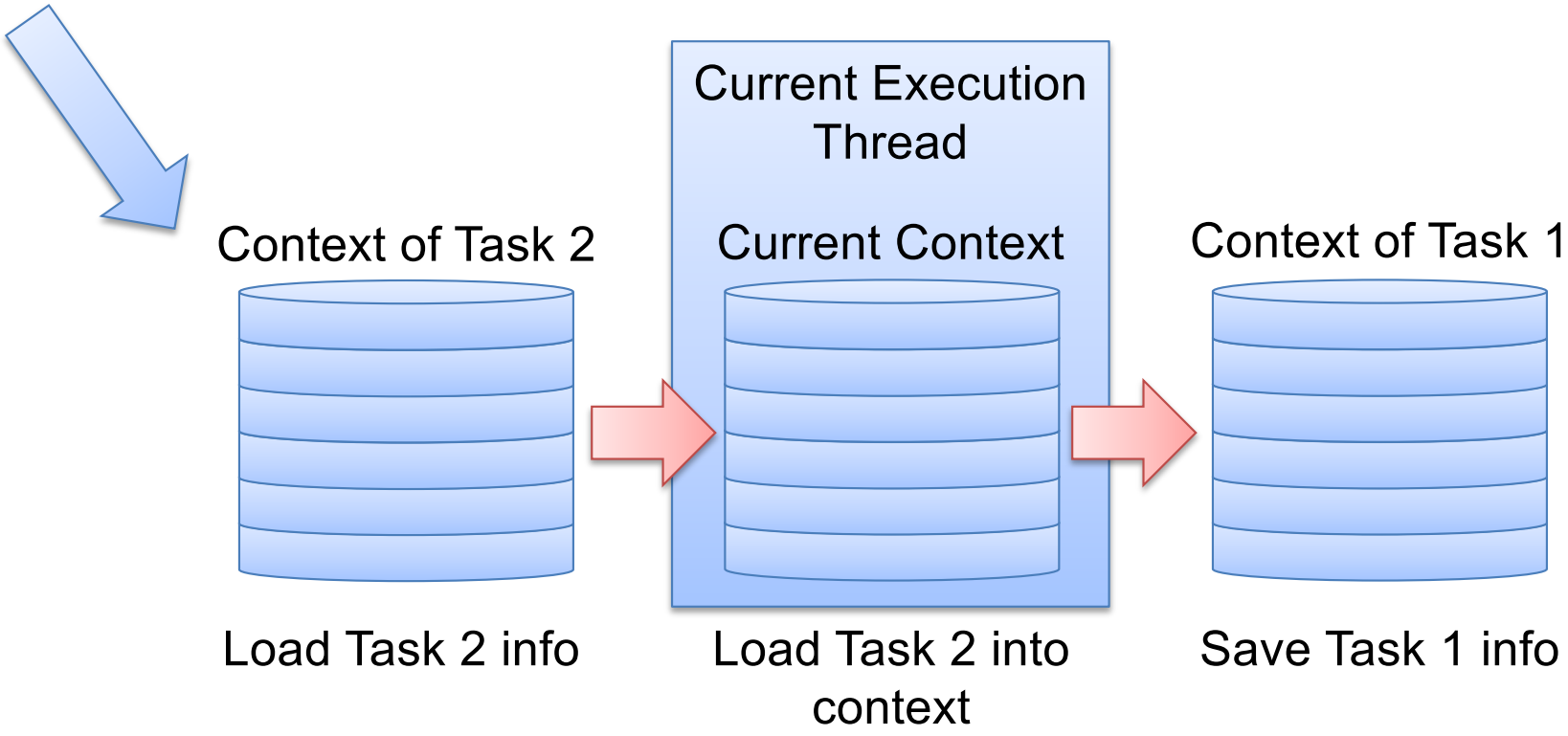
- Small scale embedded systems
- When standard lib functions in C can be used instead of RTOS functions (malloc(), free(), fopen())

Scheduling

Context switching

The context includes:

- Copy of all state (machine registers)
- Address at which to resume executing the thread
- Status of the thread (e.g. blocked on mutex)
- Priority, WCET (worst case execution time), and other info to assist the scheduler



Scheduling Algorithms – When can tasks switch?

Non-pre-emptive scheduling

- When the current thread completes

With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one

In a **pre-emptive** scheme, there will be an immediate switch to the higher-priority process

With **non-pre-emption**, the lower-priority process will be allowed to complete before the other executes

Pre-emptive schemes enable higher-priority processes to be more reactive, and hence they are preferred

Pre-emptive scheduling

- Upon a timer interrupt
- Upon an I/O interrupt (possibly)
- When a new thread is created, or one completes.
- When the current thread blocks on or releases a mutex
- When the current thread blocks on a semaphore
- When a semaphore state is changed
- When the current thread makes any OS call
  - File system access
  - Network access

Scheduling Algorithms

Fixed-Priority Scheduling (FPS)

- Each process has a fixed, static, priority which is computed pre-run-time
- The runnable processes are executed in the order determined by their priority
- In real-time systems, the "priority" of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity

Dynamic-Priority Scheduling

- The runnable processes are executed in the order determined by the absolute deadlines of the processes
- The next process to run being the one with the shortest (nearest) deadline
- Although it is usual to know the relative deadlines of each process (e.g. 25ms after release), the absolute deadlines are computed at run time and hence the scheme is described as dynamic

Fixed-Priority Scheduling (FPS)

One kind we will look at is **Rate Monotonic Scheduling (RMS)**

- Each process is assigned a (unique) priority (*P*) based on its period (*T*); the shorter the period, the higher the priority i.e, for two processes i and j

$$T_i < T_j \Rightarrow P_i > P_j$$

- RMS is optimal in the sense of feasibility
  - Feasibility is defined for RMS to mean that every task executes to completion once within its designated period

Rate Monotonic Scheduling Example 1

Assign priorities to the processes (1 is the lowest priority)

Process	Period (T)	Computation Time (C)	Priority (P)	Utilization (U)
A	50			
B	40			
C	30			

Schedulability test

We can test whether a process set can be scheduled using this formula:

$$U \leq N\left(2^{\frac{1}{N}} - 1\right)$$

Where U is the utilization defined by:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

The bit on the right is the utilization bound

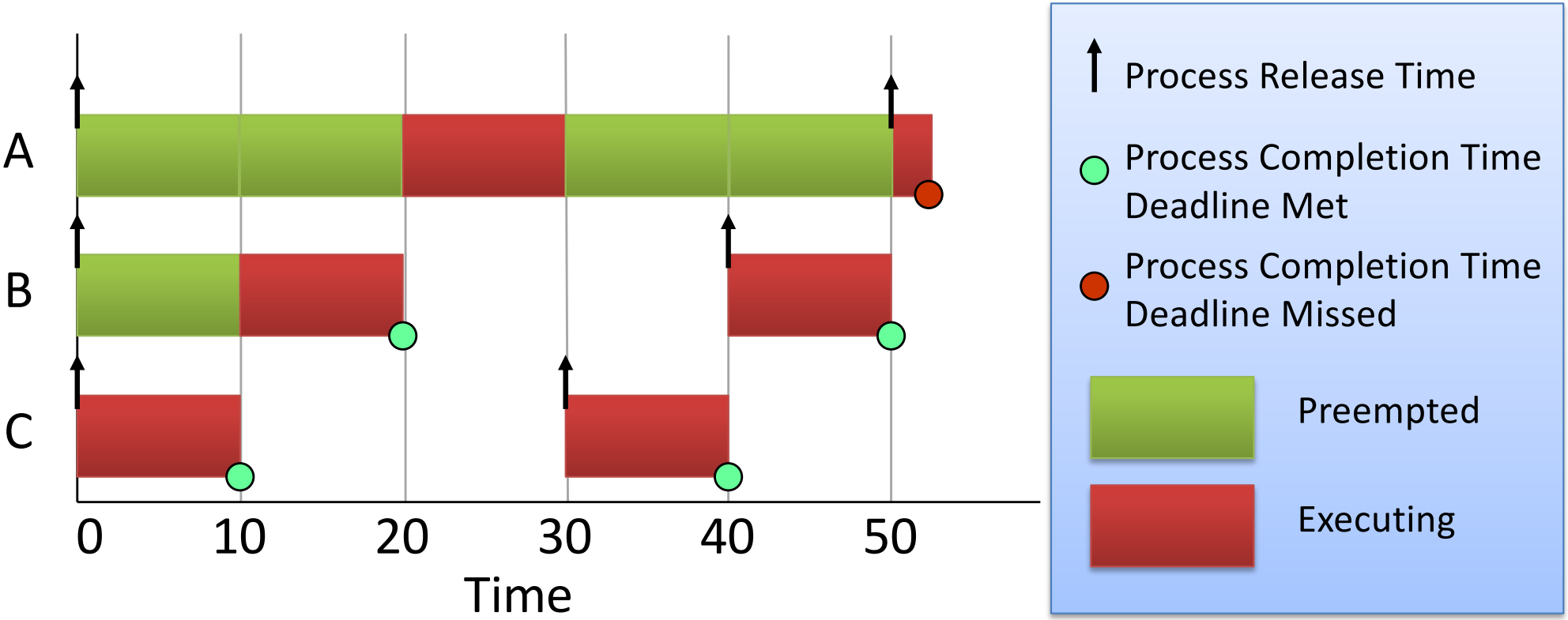
This tends to 0.78 for N=3

The combined utilization for this set is 0.82. This is above the threshold for N=3 processes!

Rate Monotonic Scheduling Example

Let's actually see what would happen!

Process	Period (T)	Computation Time (C)	Priority (P)	Utilization (U)
A	50	12	1	0.24
B	40	10	2	0.25
C	30	10	3	0.33



Rate Monotonic Scheduling Example 2

Another process set:

Process	Period (T)	Computation Time (C)	Priority (P)	Utilization (U)
A	80	40	1	0.50
B	40	10	2	0.25
C	20	5	3	0.25

Schedulability test

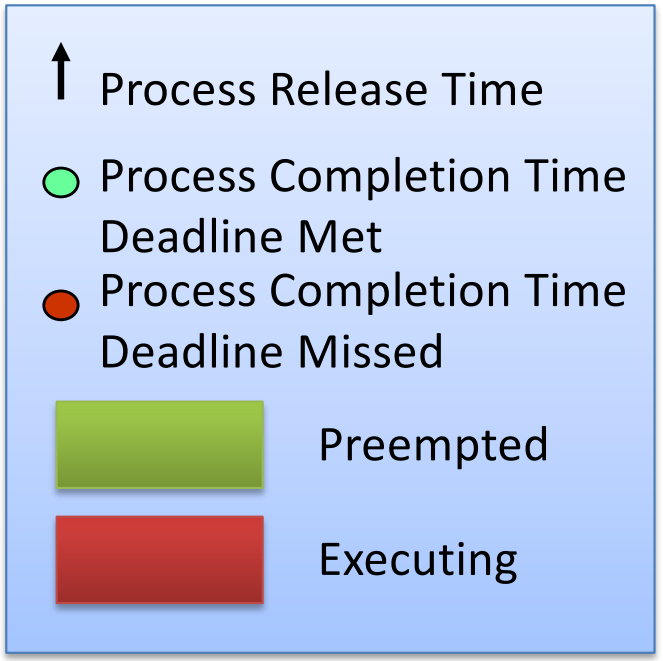
- The combined utilization is 1!
- The threshold for N=3 is 0.78, **but this set will still meet its deadlines**

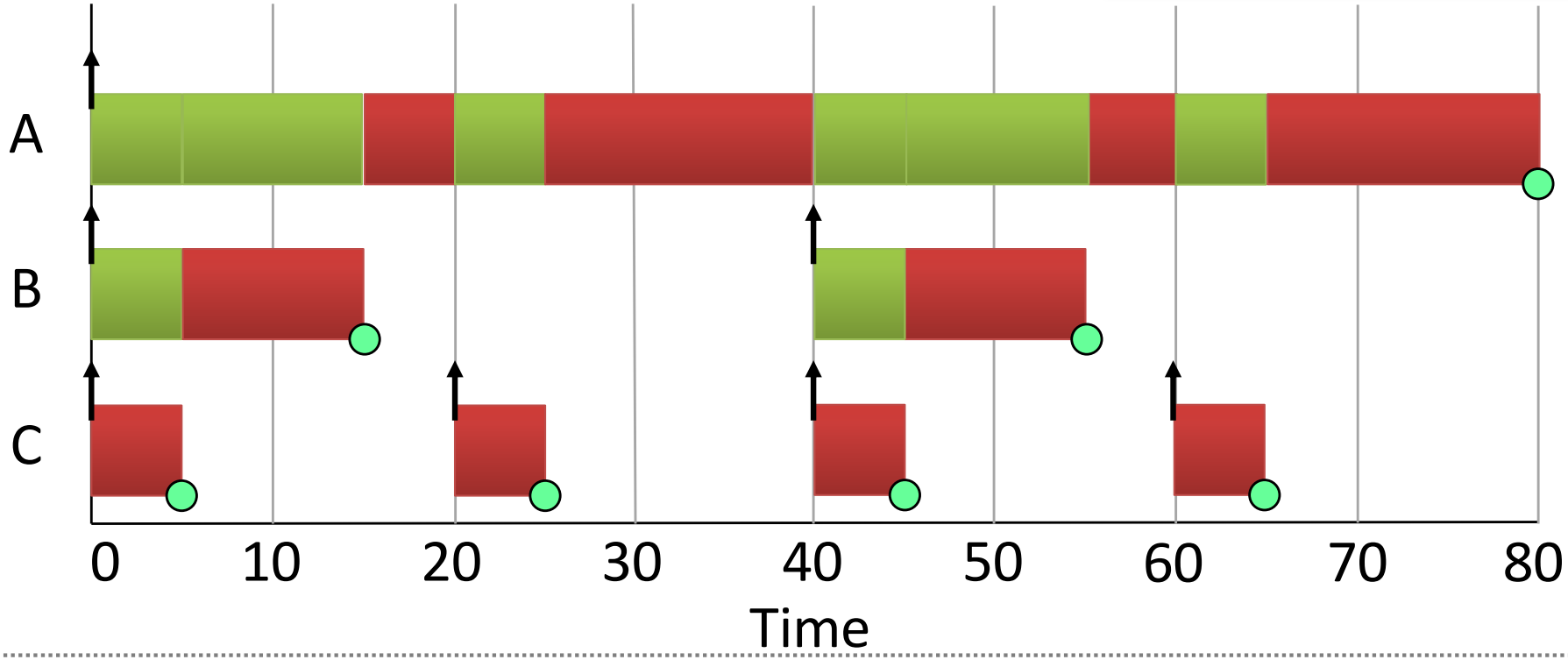
The test is said to be sufficient but not necessary

Rate Monotonic Scheduling Example 2

Let's actually see what would happen!

Process	Period (T)	Computation Time (C)	Priority (P)
A	80	40	1
B	40	10	2
C	20	5	3





Dynamic-Priority Scheduling

One kind we will look at is **Earliest Deadline First**

Horn' Rule: *Given a set of n independent task with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.*

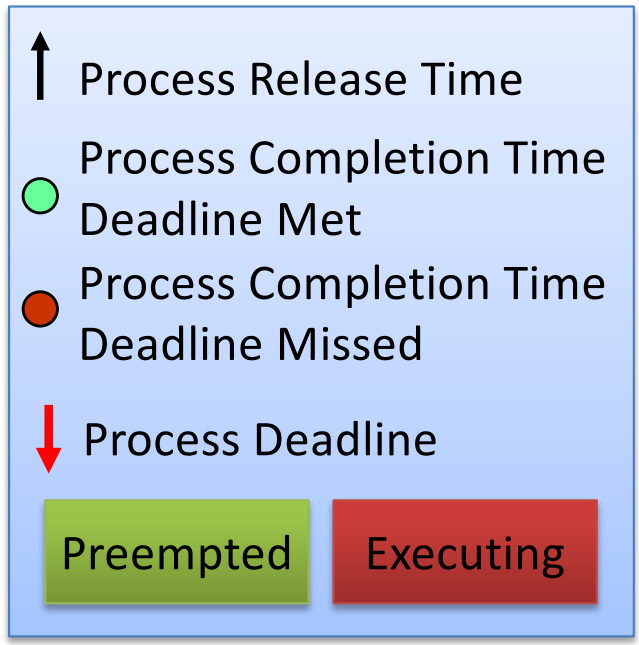
Our algorithm will always run the task with the closest deadline. Check will be performed when a new task is released, or when a running task completes execution.

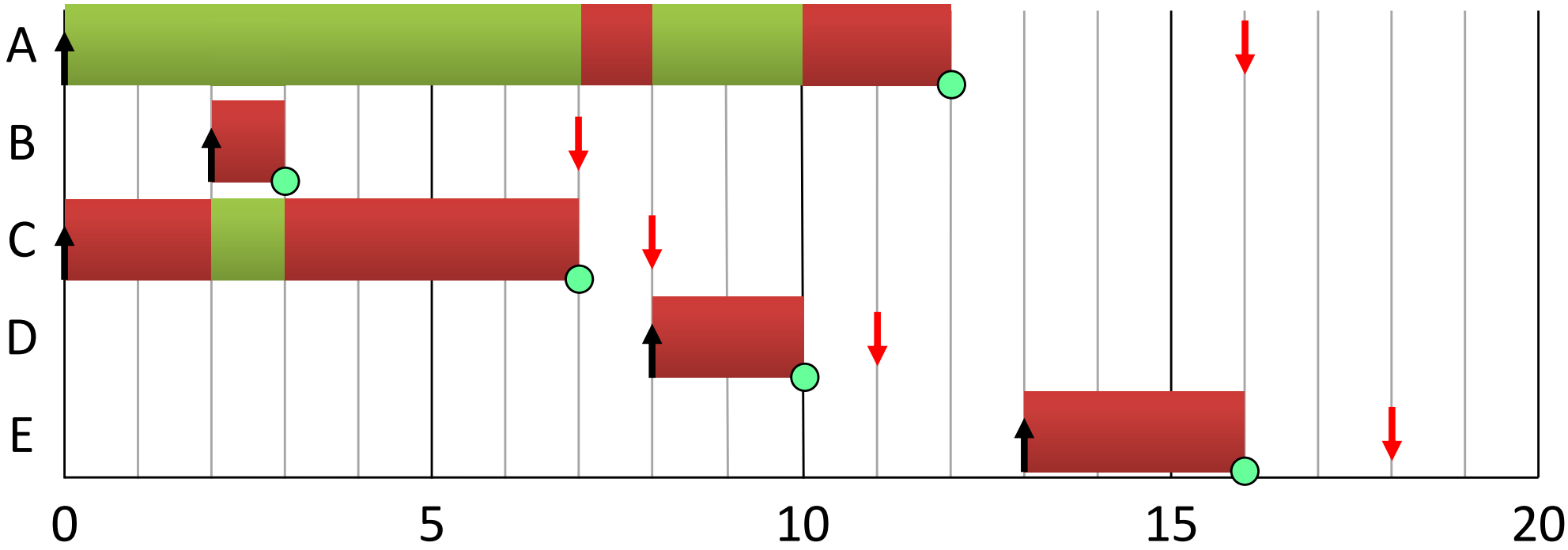
Schedulability test

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

Dynamic-Priority Scheduling - Example

Process	Computation Time (C)	Arrival time	Deadline
A	3	0	16
B	1	2	7
C	6	0	8
D	2	8	11
E	3	13	18





Tasks vs. interrupts

Interrupts

- Context is changed immediately (more responsive)
- ISRs **must** be as short as possible

Tasks

- Context switch is managed by scheduler
- Can contain lengthy code and maintain responsiveness

Example approach:

You want to initiate a lengthily operation when a button is pushed and don't want to risk the button press being missed

Solution: use an interrupt to service the button and store a variable that indicates to a regular task that it needs to run

Resource Sharing and Synchronisation

What is all this sharing about?

What needs sharing?

- Data structures
- Variables
- Main memory area
- Files
- Registers
- I/O units

Why?

- Many resources do not allow simultaneous access – mutual exclusion

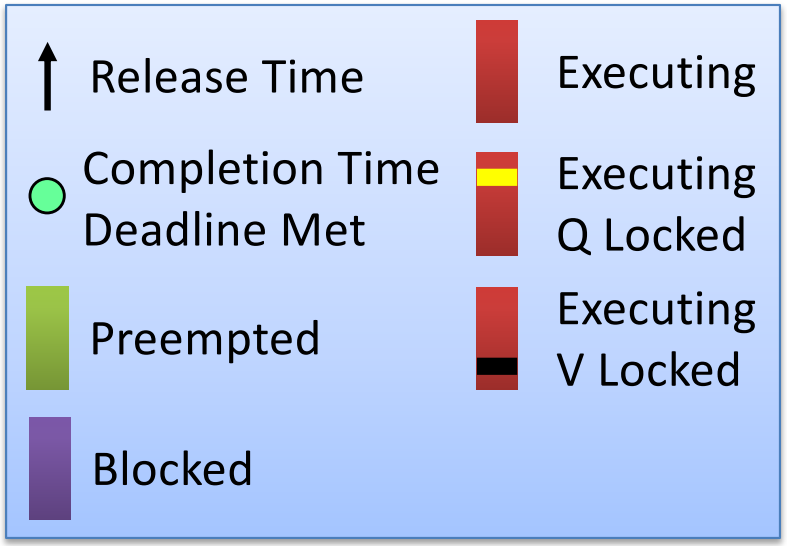
What do we use to control sharing??

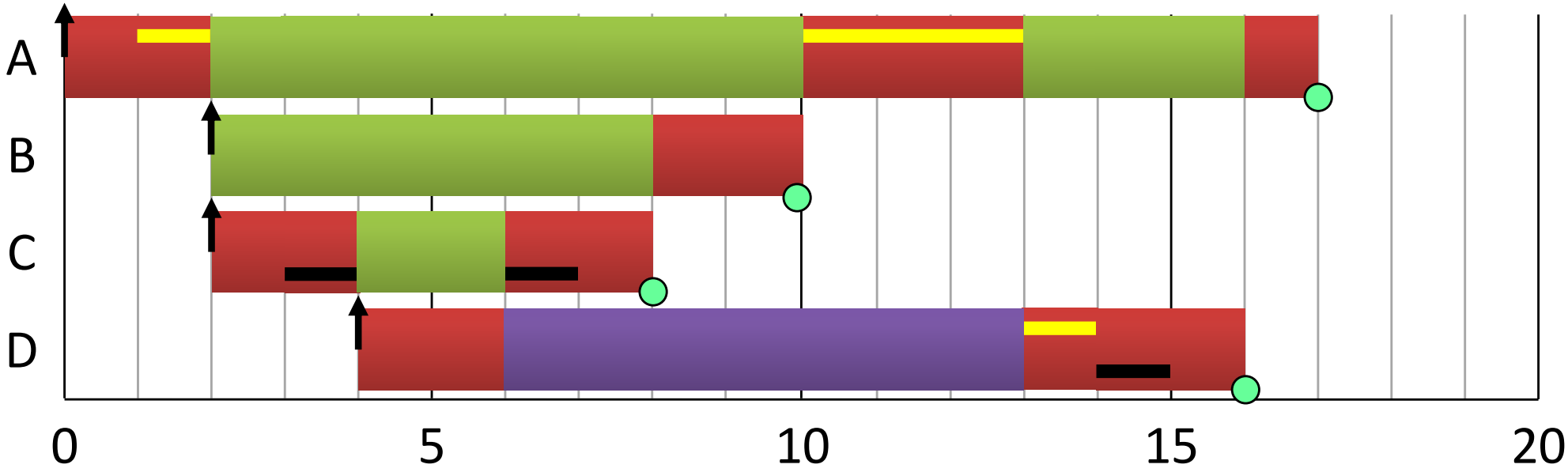
- Semaphores
  - a signalling mechanism , it allows a number of threads to access shared resources
- Mutexes
  - Is an object owned by thread. Mutex allows only one thread to access a particular resource
- Critical sections
  - Disables the scheduler and interrupts

Priority inversion

What happens if a high priority task is blocked waiting for a resource that a low priority task has a lock on?

Process	Release Time	Priority (P)	Sequence
A	0	1	EQQQQE
B	2	2	EE
C	2	3	EVVE
D	4	4	EEQVE

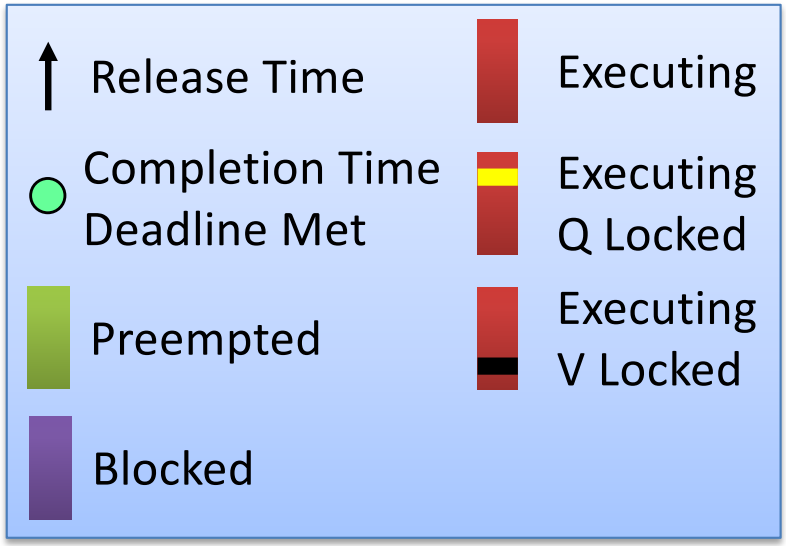


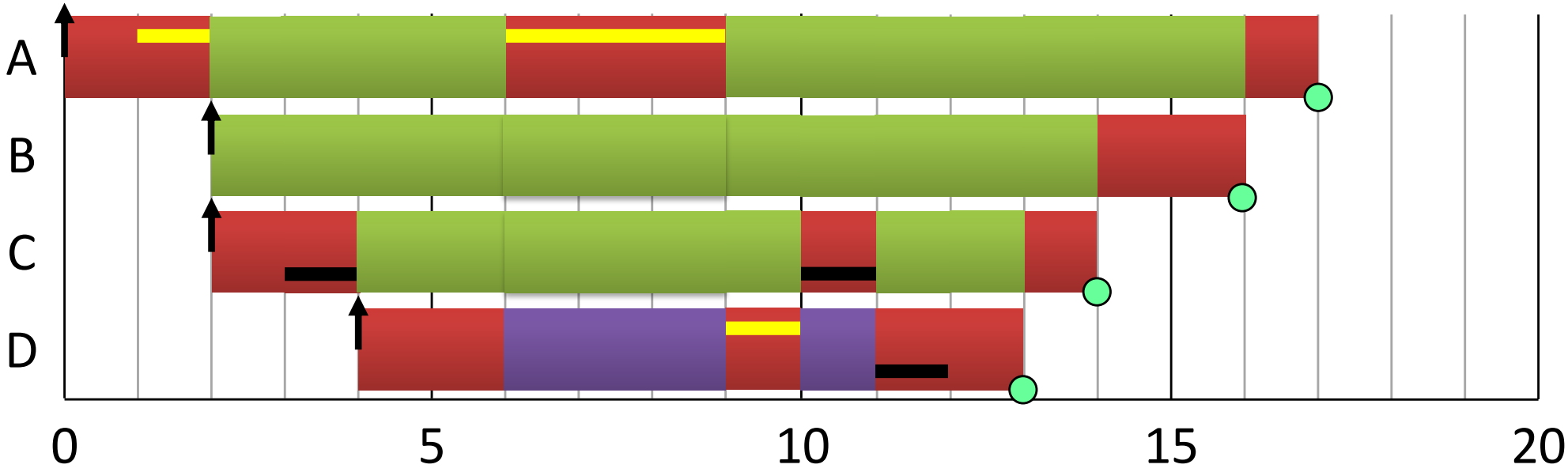


Priority inheritance

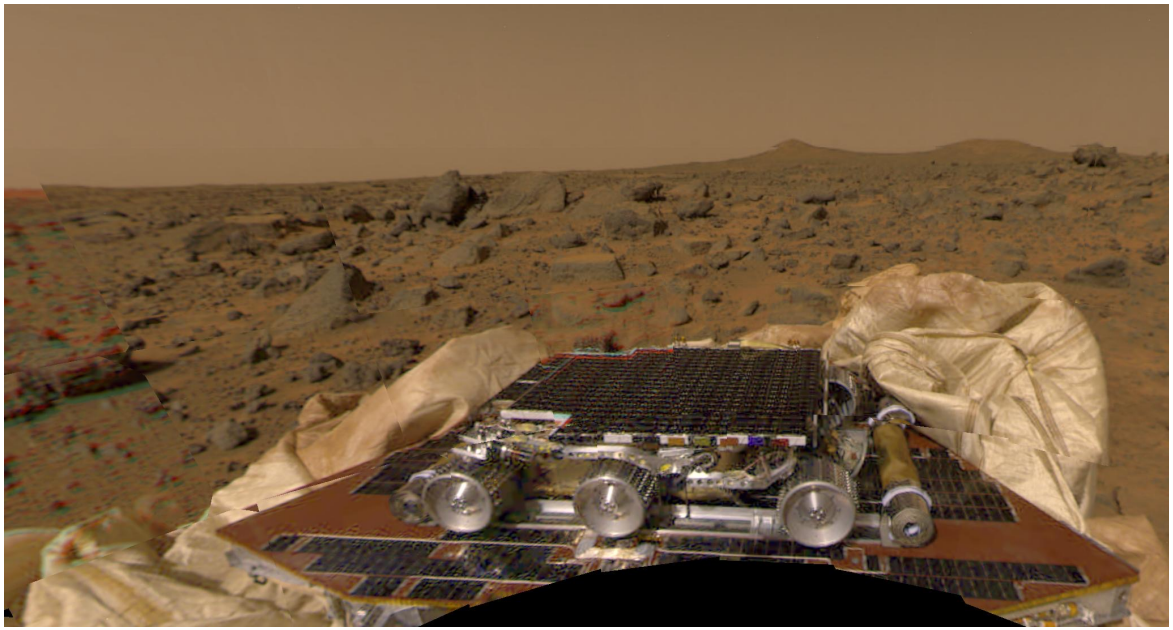
If process *p* is blocking process *q*, the *p* runs with *q*'s priority

Process	Release Time	Priority (P)	Sequence
A	0	1	EQQQQE
B	2	2	EE
C	2	3	EVVE
D	4	4	EEQVE





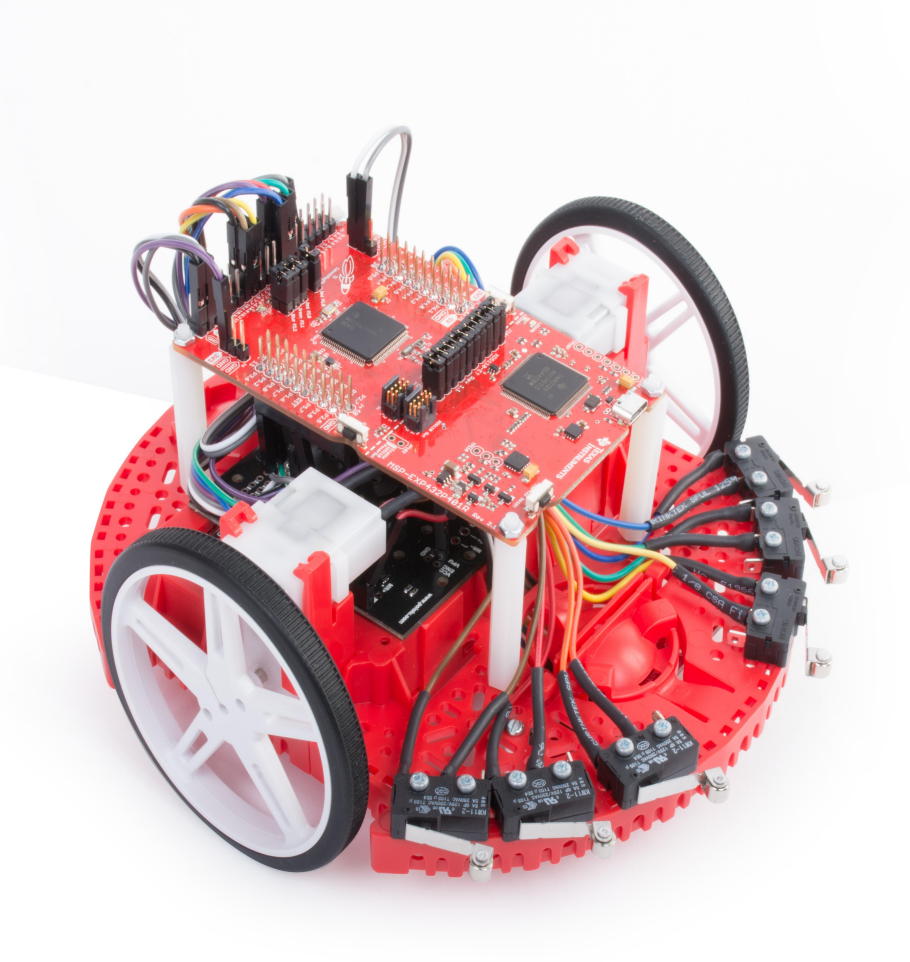
Priority inheritance



The Mars Rover Pathfinder landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. The problem was diagnosed on the ground as **priority inversion**, where a low priority meteorological task was holding a lock blocking a high-priority task while medium priority tasks executed.

Problem 2

- Figure out what data and resources your system needs to share and plan how you will share these.
- Write tasks that allow the music playing to while performing all functions a described in problem 1.
- Write tasks that show some information on all the above to the coloured LED and red LED.
- **Schedule this system so that all the tasks run and meet their deadlines**



- See you on Monday in the Electronic Lab