Embedded Software Architectures

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mário de Sousa
msousa@fe.up.pt

Luis Almeida
lda@fe.up.pt

Embedded Software Architecture

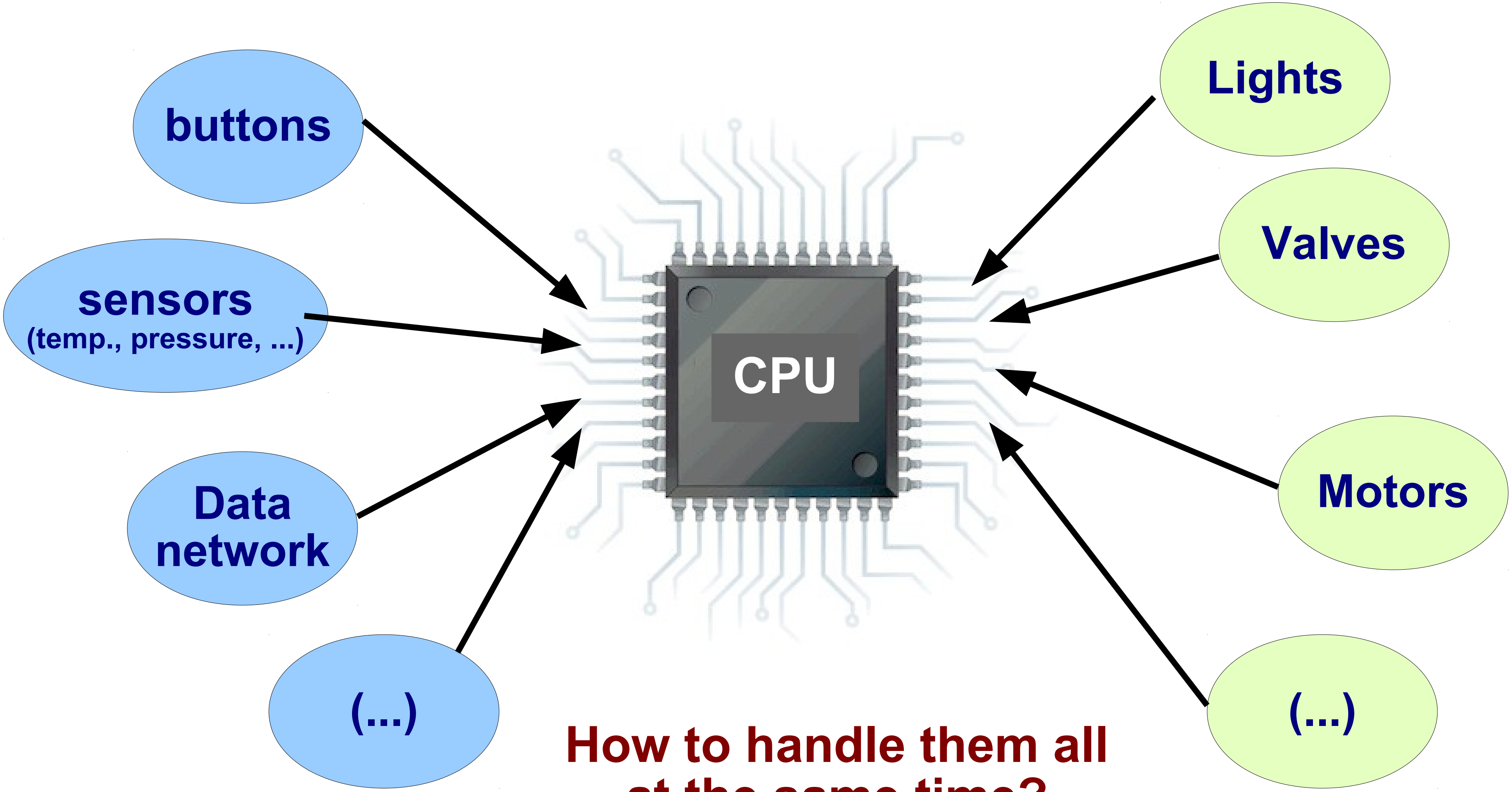
- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Components of an Embedded System

- "An embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based."

  Michael J. Pont, in "Embedded C", Addison Wesley, 2002

Components of an Embedded System



buttons

sensors
(temp., pressure, ...)

Data network

(...)

CPU

Lights

Valves

Motors

(...)

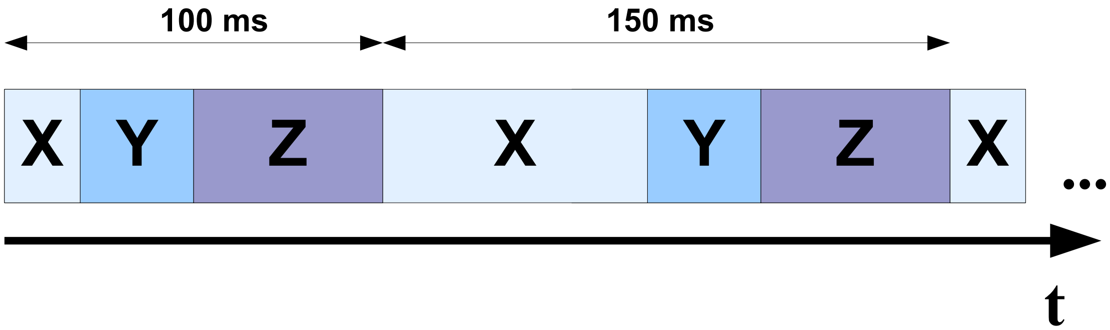
How to handle them all at the same time?

Embedded Software Architecture

- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Cyclic Executive: code structure

```
int main (void) {
  Hardware_init();

  FuncX_init();
  FuncY_init();
  FuncZ_init();

  while (1) {
    FuncX();
    FuncY();
    FuncZ();
  }
}
```



Cyclic Executive: code organization

Main.c

```
#include “FuncX.h”
#include “FuncY.h”

int main (void) {
  Hardware_init();
  FuncX_init();
  FuncY_init();
  while (1) {
    FuncX();
    FuncY();
  }
}
```

FuncX.h

```
int FuncX     (void);
int FuncX_init(void);
```

FuncX.c

```
int FuncX(void){
  /* Algorithm X    */
  /* implementation */
  ...
};

int FuncX_init(void){...}
```

Cyclic Executive: Advantages

- Portability
  - Does not require special µProcessor resources (timers, interrupts, ...)
- Simplicity
  - implementation
  - testing
  - debugging
  - understanding
- Deterministic => Reliable and Safe
  - Fit for safety critical applications

Cyclic Executive: Drawbacks

- Difficult to guarantee exact timing behaviour
  - examples:

    read speed every 5ms,
    update fuel/air mixture every 10ms.
  - Timings change when updating the SW

    may affect the behavior of closed loop control
- µProcessor is always active
  - May not be necessary for the current application
  - Uses up more energy

    (energy is limited resource in battery-based applications)

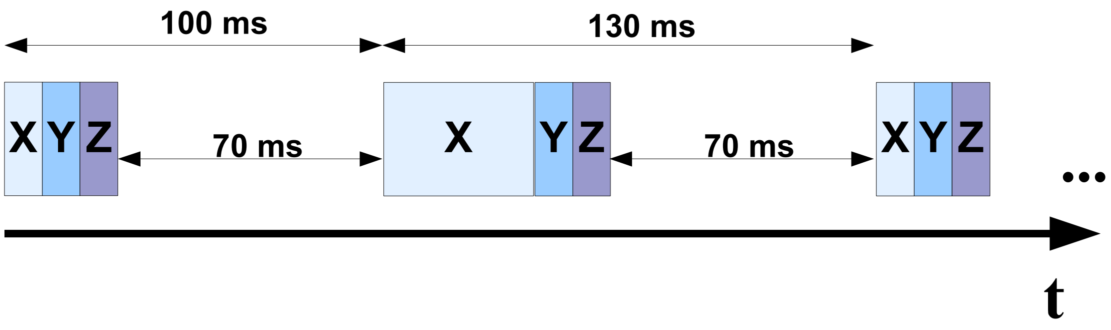
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Cyclic Executive: Controlling Repetition Period with Delays

```
int main (void) {
  FuncX_init();
  FuncY_init();
  FuncZ_init();

  while (1) {
    /* X,Y,Z - Periodic Tasks
     *       - Period -> 100 ms
     * - Funcs X,Y,Z each take
     *   10ms to execute
     */
    FuncX();
    FuncY();
    FuncZ();
    /* Delay 70 ms */
    Busy_Sleep(70);
}}
```

- Main Features
  - Waste of processing resource
  - Imprecise timing

- What if...
  FuncX takes longer to execute?
  - Not good timing control



Busy Waiting in Software

- Delay using Software
  - Independent of µProcessor resources
  - Is dependent on
    - µProcessor architecture
    - Clock frequency
    - Compiler version
    - Compiler optimizations
  - Precision depends on generated assembly code.
  - very short delays are feasible

```
int Busy_Sleep(int value) {
  while(value--)
    for(x=0; x<=65535; x++);
  return 0;
}
```

Warning: compiler optimizations may simply ignore this code!

```
int Busy_uSleep(void) {
  int x;
  x++;
  x++;
  return 0;
}
```

Busy Waiting in Hardware

- Delay using Hardware
  - Use a µProcessor **timer**
  - Code is **not portable**
    (depends on available timer)
  - Precision will depend on timer's clock frequency.

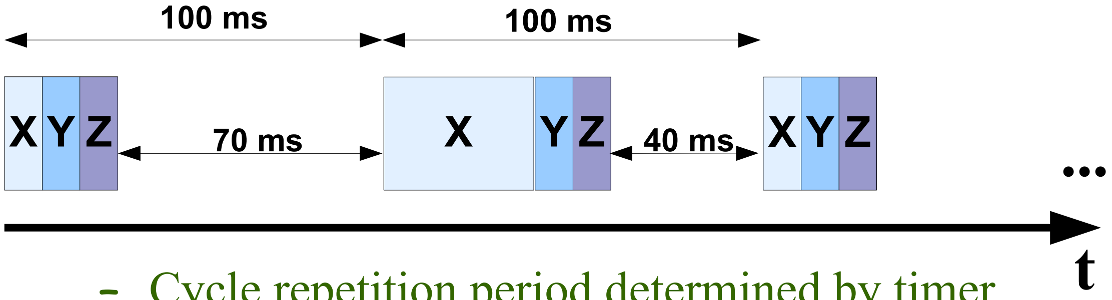
```
int Busy_Sleep(int value) {
  init_hardware_timer();
  while(!hardwaretimer_end);
  return 0;
}
```

Cyclic Executive: Repetition Period with Hardware Timer

```
int main (void) {
  FuncX_init();
  FuncY_init();
  FuncZ_init();
  Timer_Init();
  while (1) {
    /* X,Y,Z - Periodic Tasks
     *       - Period -> 100 ms
     * - Funcs X,Y,Z each take
     *   10ms to execute
     */
    FuncX();
    FuncY();
    FuncZ();
    sync(); /* Sync cycle */
  }
}
```

Warning: tell compiler that external event may change variable's value at any time. Without this we may enter an infinite loop.

```
volatile int go = 0;

ISR (TimerIntVector){
  go = 1; /* set flag */
}
void sync (void) {
  while(!go); /* wait */
  go = 0;  /* reset flag */
}
```

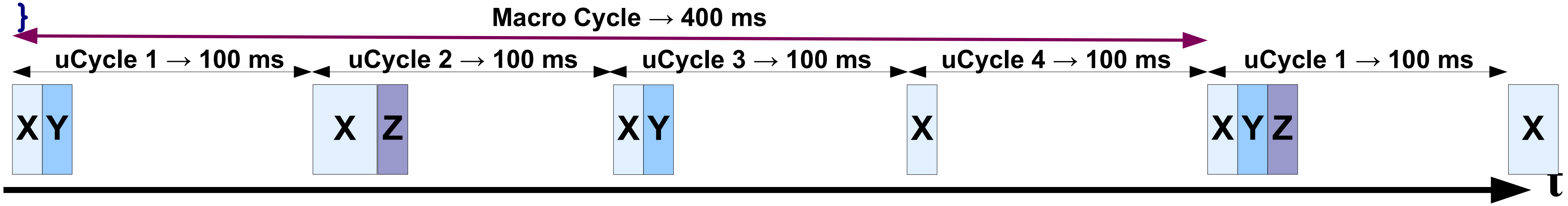


- Cycle repetition period determined by timer
- Cycle repetition period independent of function execution time

Cyclic Executive:
Static Cyclic Scheduling Table

```
int main (void) {
  FuncX_init();
  FuncY_init();
  FuncZ_init();
  Timer_Init();

  while (1) {
    micro1();sync();//uCycle 1
    micro2();sync();//uCycle 2
    micro3();sync();//uCycle 3
    micro4();sync();//uCycle 4
  }
}
```

```
volatile int go = 0;
ISR (TimerIntVector)
  {go = 1; /* set flag */}
void sync (void) {
  while(!go); /* wait */
  go = 0;   /* reset flag */
}
void micro1(void) {FuncX();FuncY();}
void micro2(void) {FuncX();FuncZ();}
void micro3(void) {FuncX();FuncY();}
void micro4(void) {FuncX();}
```



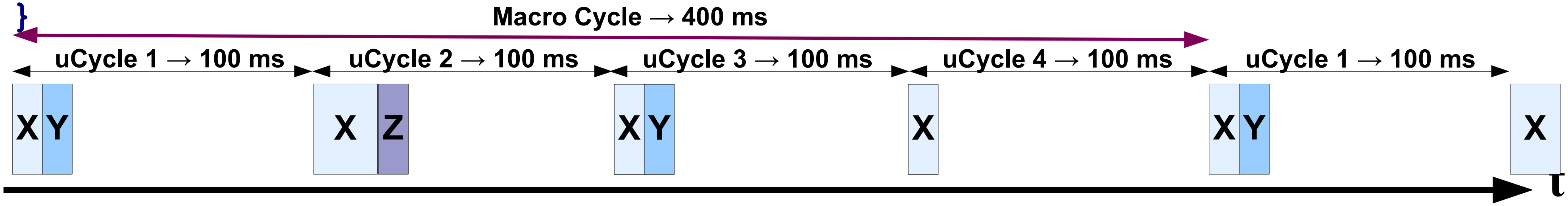
Cyclic Executive:
Static Cyclic Scheduling Table

- Timer → set to the period of the micro-cycle (uCycle)
- sync() → wait for timer / beginning of next cycle
- Period of uCycle → GCD of task periods (GCD: Greatest Common Denominator)
- In the example:
  - FuncX → period = 100 ms; offset = 0 ms
  - FuncY → period = 200 ms; offset = 0 ms
  - FuncZ → period = 400 ms; offset = 100 ms

Attention to uCycle over-runs !

```
while (1) {
  micro1();sync();//uCycle 1
  micro2();sync();//uCycle 1
  micro3();sync();//uCycle 1
  micro4();sync();//uCycle 1
}
```

```
void micro1(void) {FuncX();FuncY();}
void micro2(void) {FuncX();FuncZ();}
void micro3(void) {FuncX();FuncY();}
void micro4(void) {FuncX();}
```



Embedded Software Architecture

- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Interruption Based Executive

- Each task is an ISR that invokes respective function

(ISR: Interrupt Service Routine)

Main.c

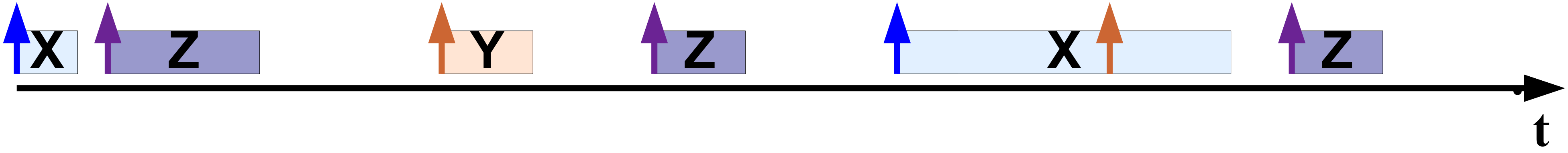
```
#include "FuncX.h"
#include "FuncY.h"
#include "FuncZ.h"

int main (void) {
  Hardware_init();
  FuncX_init();
  FuncY_init();
  FuncZ_init();
  while (1);
}
```

FuncX.c

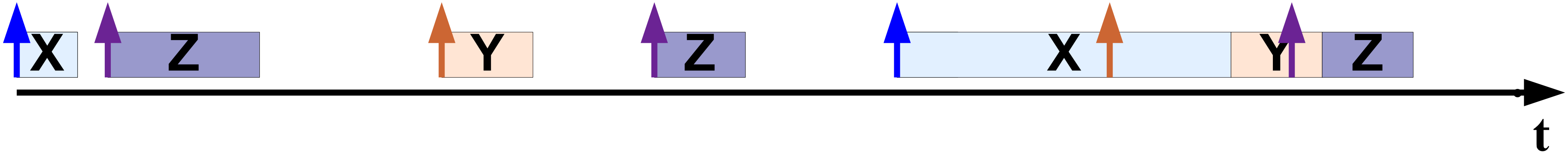
```
int FuncX_init(void){
 /* Configure interrupt
  * that calls FuncX();
  */
 ...
};

int FuncX(void){
  /* Implement    */
  /* Algorithm X  */
  ...
};
```

Interruption Based Executive

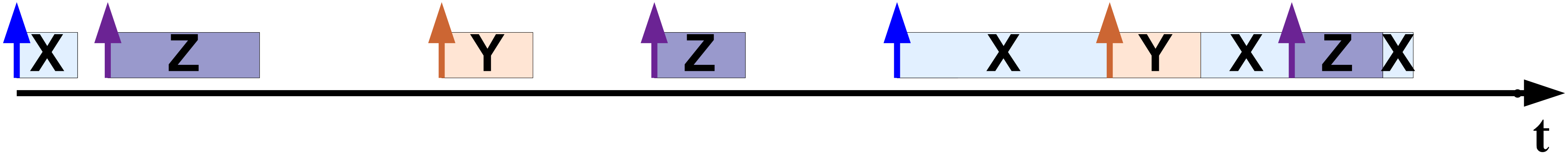
- Interrupts that occur during the processing of other interrupts can be... (depending on the hardware)
  - Ignored



  - Deferred



  - Pre-empt current interrupt routine



Interruption Based Executive

- Pre-emption...
  - … **is good**, as it allows coexistence of **very long**, and **very short and frequent** functions/tasks
  - … **is bad** because of **race conditions** when accessing shared resources (global variables, buffers, etc…)
- Some µProcessors support interrupt priorities
  - Applicable for deferred and pre-emption based handling of interrupts.
  - Usually very **limited number** of priorities
  - Priorities are frequently hardwired to the interrupt source (external pin, timer, USART, ...)

Interruption Based Executive

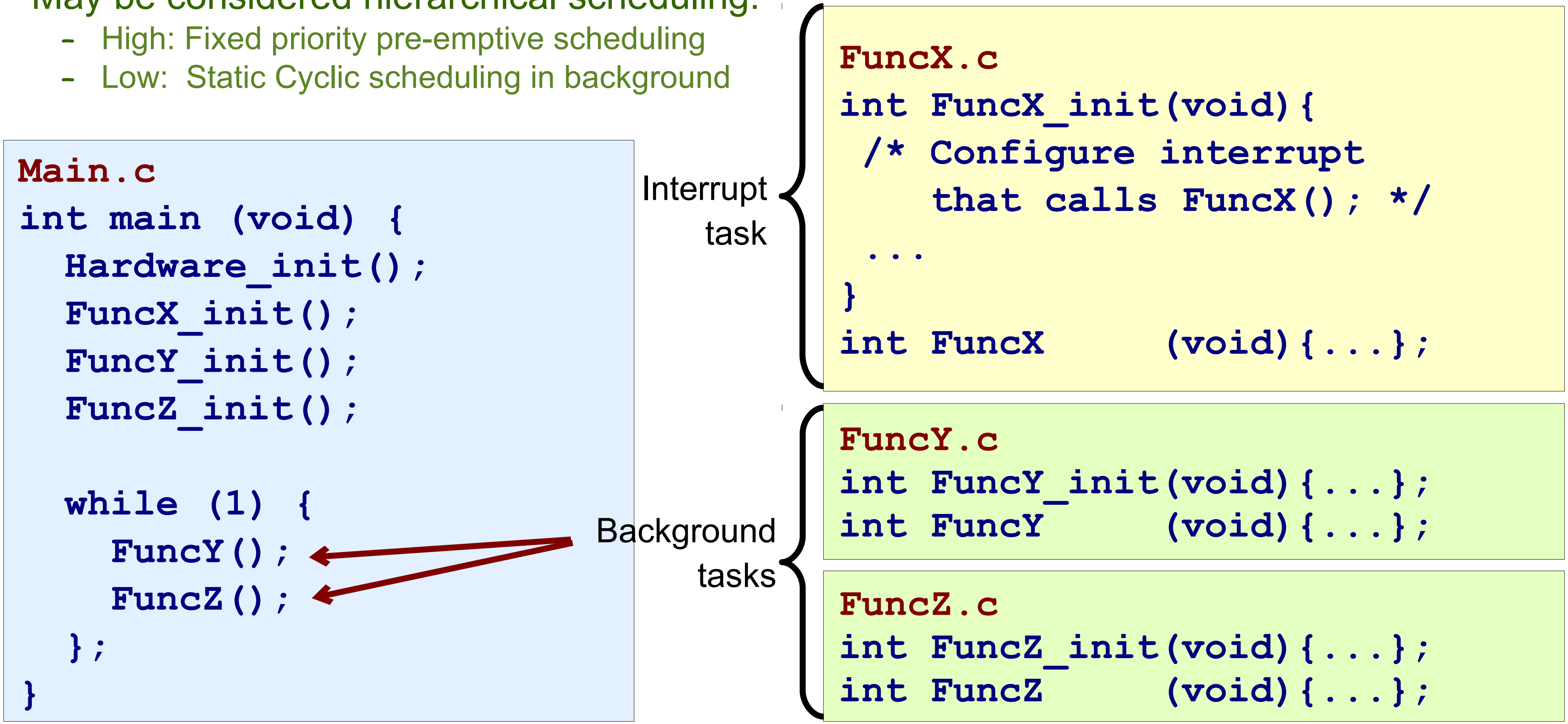
- Advantages:
  - Simple to generate perfectly periodic tasks
    - Interrupt is generated by external hardware timer
      (timer may be internal or external to the µProcessor)
    - Activation period is not affected by task's execution time
  - Very reactive for **aperiodic** tasks
    - Triggered directly by interrupts from communication port, keyboard, …
- Drawback
  - **Each task** is attached to **one** physical hardware **interrupt!**
  - In particular, **each** periodic task requires **its own** hardware timer!

Embedded Software Architecture

- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Hybrid: Cyclic + Interrupts

- Some tasks are ISR based, while other tasks run in background
- May be considered hierarchical scheduling:
  - High: Fixed priority pre-emptive scheduling
  - Low: Static Cyclic scheduling in background



Main.c

```
int main (void) {
  Hardware_init();
  FuncX_init();
  FuncY_init();
  FuncZ_init();

  while (1) {
    FuncY();
    FuncZ();
  };
}
```

Interrupt task

FuncX.c

```
int FuncX_init(void){
 /* Configure interrupt
    that calls FuncX(); */
 ...
}
int FuncX     (void){...};
```

Background tasks

FuncY.c

```
int FuncY_init(void){...};
int FuncY     (void){...};
```

FuncZ.c

```
int FuncZ_init(void){...};
int FuncZ     (void){...};
```

Embedded Software Architecture

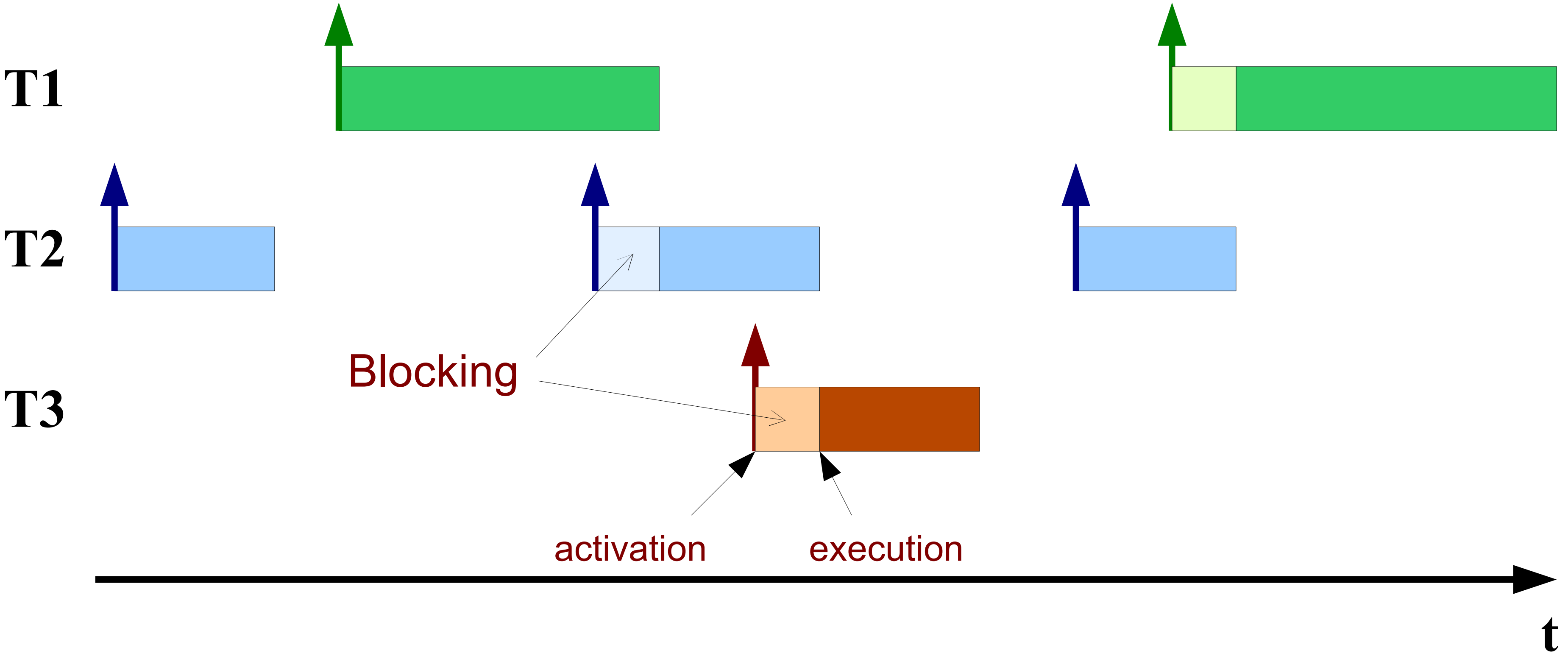
- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Co-operative Multi-tasking

- Tasks are scheduled to activate at well defined instants
  - **Periodic tasks** (example: every 100 ms starting at 10ms)
  - **One-shot tasks** (example: once at 30 ms)
  - All done with a **single hardware timer**, that generates periodic interrupts with a frequency equal to the desired time resolution (GCD of task periods).

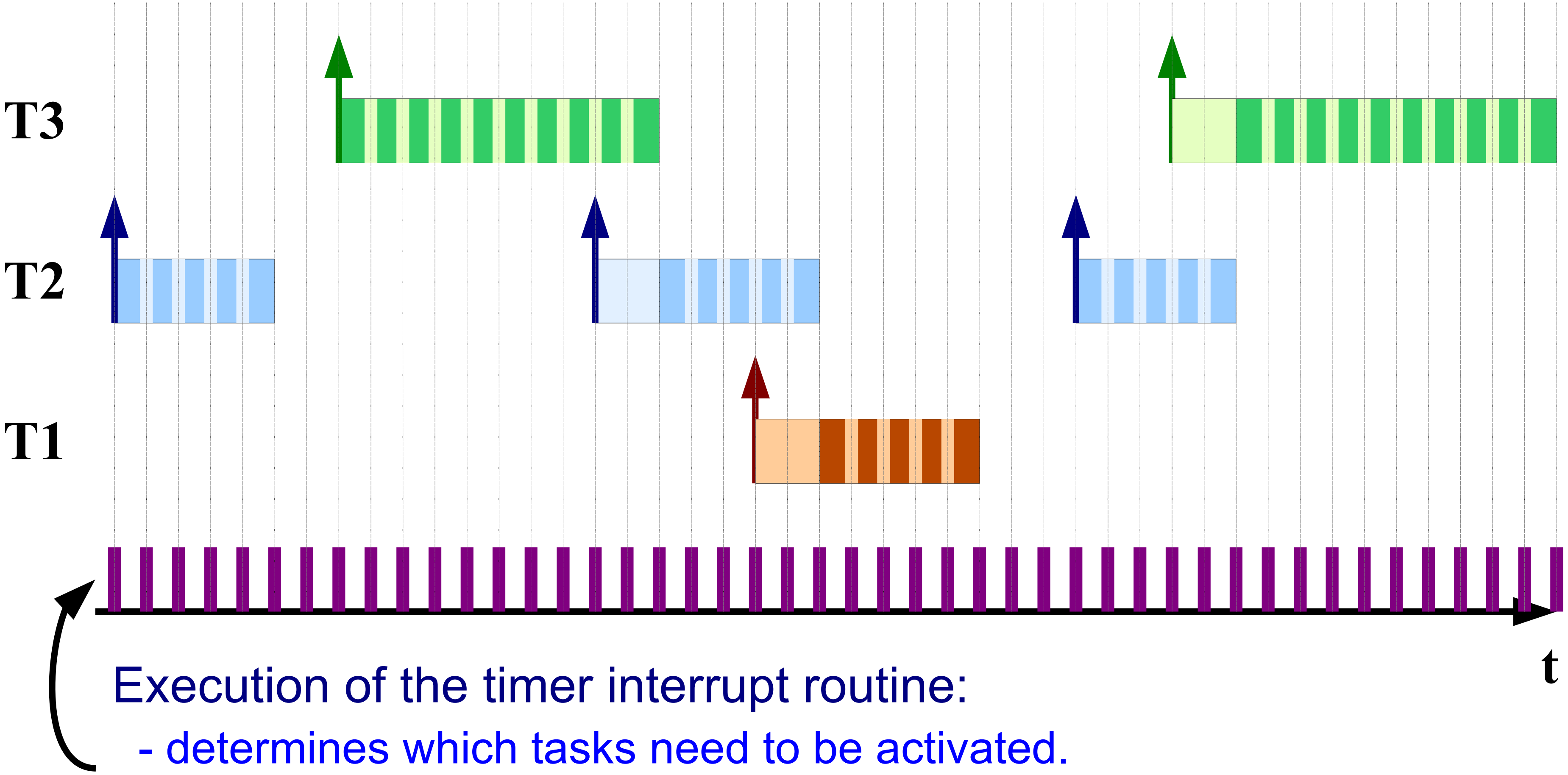
- A task, once having started execution, completes **without being suspended** (non pre-emptive execution).
  - If a second task is activated while the first is being executed, the second task will only start execution after the first task finishes.

Co-operative Multi-tasking



T1 – periodic (activates every X ms)
T2 – periodic
T3 – one-shot (activates once in Y ms time)

Co-operative Multi-tasking



Execution of the timer interrupt routine:

- determines which tasks need to be activated.
- in reality, execution time should be much shorter than shown in the graph.
- activation period determines the resolution of the possible activation times.

Co-operative Multi-tasking

Main.c

```
int main (void) {
  Sched_Init();
  /* periodic task */
  FuncX_init();
  Sched_AddT(FuncX, 0, 4);
  /* one-shot task */
  FuncY_init();
  Sched_AddT(FuncY, 50, 0);

  while (1) {
    Sched_Dispatch();
  }
}
```

Kernel **initialization** (data and tick timer)

Create a **new task** with function, offset, and period

every

start in

Execute tasks that have been marked as **ready** to execute

Scheduler.c

```
void int_handler(void){
  Sched_Schedule();
}
```

Every **tick** check for task activations, and mark those tasks as **ready** to execute

Co-operative Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks              */
  int period;
   /* ticks until next activation */
  int delay;
   /* function pointer             */
  void (*func)(void);
   /* activation counter           */
  int exec;
} Sched_Task_t;
```

One copy of this data structure for each task.

```
#define NT 20
Sched_Task_t Tasks[NT];
```

Maximum number of tasks (`#define NT 20`)

Array of structures for all tasks (`Sched_Task_t Tasks[NT];`)

Co-operative Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks     */
  int period;
   /* ticks to activate   */
  int delay;
   /* function pointer    */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;

#define NT 20
Sched_Task_t Tasks[NT];
```

Scheduler.c

```
int Sched_Init(void){
  for(int x=0; x<NT; x++)
    Tasks[x].func = 0;
 /*
  * Also configures
  * interrupt that
  * periodically calls
  * Sched_Schedule().
  */
  ...
}
```

Co-operative Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks     */
  int period;
   /* ticks to activate   */
  int delay;
   /* function pointer    */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;

#define NT 20
Sched_Task_t Tasks[NT];
```

Scheduler.c

```
int Sched_AddT(
    void (*f)(void),
    int d, int p){
  for(int x=0; x<NT; x++)
    if (!Tasks[x].func) {
      Tasks[x].period = p;
      Tasks[x].delay  = d;
      Tasks[x].exec   = 0;
      Tasks[x].func   = f;
      return x;
    }
  return -1;
}
```

Co-operative Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks    */
  int period;
   /* ticks to activate  */
  int delay;
   /* function pointer   */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;

#define NT 20
Sched_Task_t Tasks[NT];
```

Scheduler.c

```
void Sched_Schedule(void){
  for(int x=0; x<NT; x++) {
    if(Tasks[x].func){
      if(Tasks[x].delay){
        Tasks[x].delay--;
      } else {
        /* Schedule Task */
        Tasks[x].exec++;
        Tasks[x].delay =
          Tasks[x].period-1;
    }}
}}
```

Co-operative Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks     */
  int period;
   /* ticks to activate   */
  int delay;
   /* function pointer    */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;

#define NT 20
Sched_Task_t Tasks[NT];
```

Scheduler.c

```
void Sched_Dispatch(void){
  for(int x=0; x<NT; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec=0;
      Tasks[x].func();
      /*  Delete task
       *  if one-shot */
      if(!Tasks[x].period)
        Tasks[x].func = 0;
    }
  }
}
```

Co-operative Multi-tasking

Note that:

- Sched_Dispatch() will execute **all tasks** that have pending activations exactly once.
- All pending tasks get a chance to execute!

- **Late** activations can be **discarded** for load stability purposes (set / reset **`exec`**),

OR

- **Late** activations can be **accumulated** (increment / decrement **`exec`**), tasks run in **Round-Robin**

(accumulating activations can cause irrevoverable overloads. If really needed, use with care!)

Scheduler.c

```
void Sched_Dispatch(void){
  for(int x=0; x<NT; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec=0;
      Tasks[x].func();
      /*  Delete task
       *  if one-shot */
      if(!Tasks[x].period)
        Tasks[x].func = 0;
    }
  }
}
```

Co-operative Multi-tasking

Alternative using **priorities**

- If several tasks have multiple pending activation requests, first exhaust the highest priority task's activation requests.
- If we consider tasks to be stored in **decreasing priority order** in **`Task_List[]`**, then we simply need to **`return`** after executing
- Sched_Dispatch() will now execute the **highest priority task** with a pending activation exactly once!

Scheduler.c

```
void Sched_Dispatch(void){
  for(int x=0; x<NT; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec=0;
      Tasks[x].func();
      /*  Delete task
       *  if one-shot */
      if(!Tasks[x].period)
        Tasks[x].func = 0;
      return;
    }
}}
```

Co-operative Multi-tasking

Alternative using **priorities**

OPTIONAL

- Allow the user to specify the task priority when adding the task.

Scheduler.c

```
int Sched_AddT(
    void (*f)(void),
    int d, int p, int pri){
  ~~for(int x=0; x<NT; x++)~~
    if (!Tasks[pri].func) {
      Tasks[pri].period= p;
      Tasks[pri].delay = d;
      Tasks[pri].exec  = 0;
      Tasks[pri].func  = f;
      return pri;
    }
  return -1;
}
```

Embedded Software Architecture

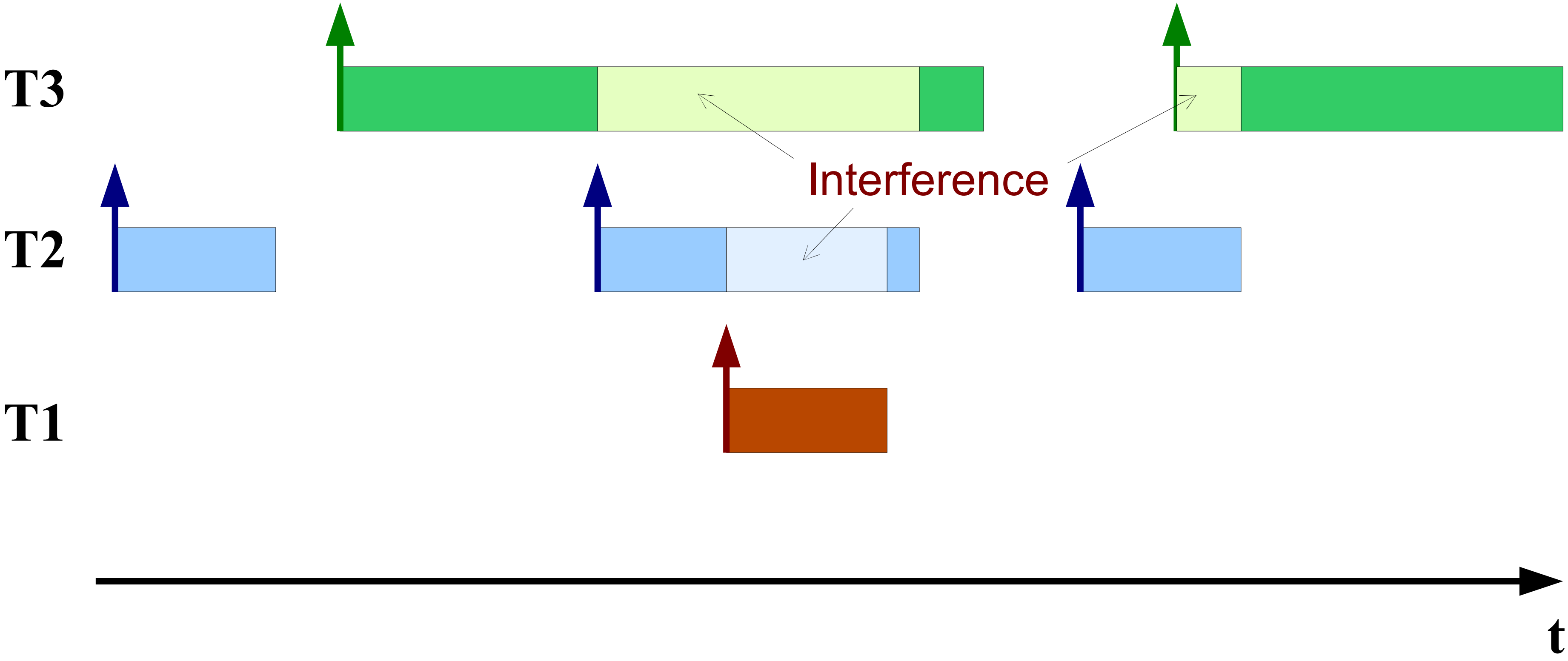
- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Pre-emptive Multi-tasking

- Tasks are **scheduled** to activate at well defined instants
  - **Periodic tasks** (example: every 100 ms)
  - **One-shot tasks** (example: once at 30 ms)

- A task, once started, **can be suspended** (pre-empted) to allow executing a **higher priority** task that has been activated
  - Pre-emption can occur recursively in the sense that a pre-empting task can itself be pre-empted and so on.

Pre-emptive Multi-tasking



Priority –

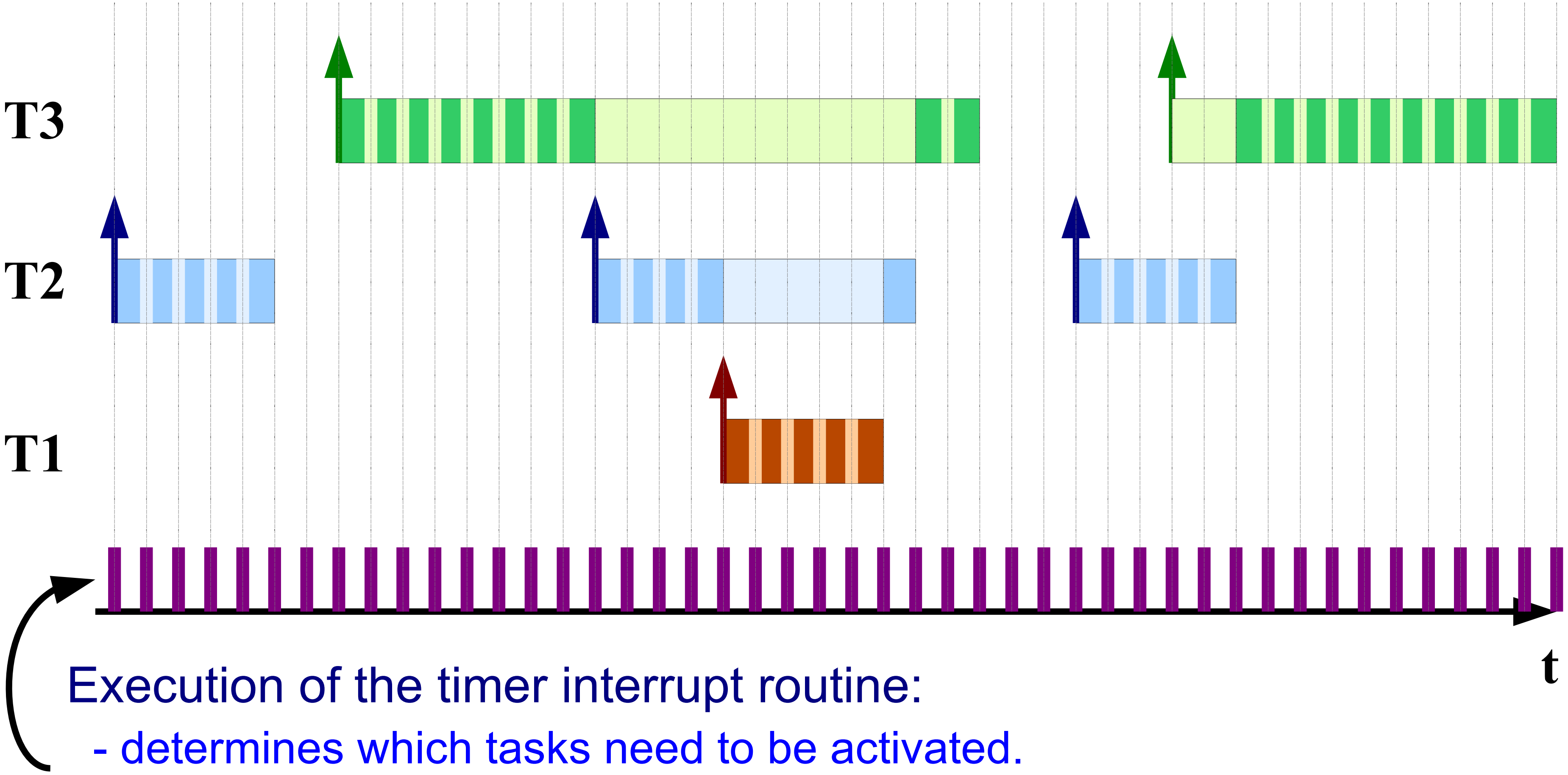
T3 – periodic – low priority

T2 – periodic – medium priority

Priority +

T1 – one-shot – high priority

Pre-emptive Multi-tasking



Execution of the timer interrupt routine:

- determines which tasks need to be activated.
- if a newly activated task has higher priority than currently executing task, then switch execution to the higher priority task.

Pre-emptive Multi-tasking

- Advantages
  - Uses a single timer for all tasks → supports many periodic tasks.
- Drawbacks
  - Possible **race conditions** when accessing shared resources
    (shared variables, shared USART, etc...).
  - Implementation is a little **more tricky**
    (depends on how tasks are mapped onto C functions!)
  - Requires more memory for **stack**
    (many tasks may be activated simultaneously, with all local variables on the stack!)

Pre-emptive Multi-tasking

No changes!!

Main.c

```
int main (void) {
  Sched_Init();
  /* periodic task */
  FuncX_init();
  Sched_AddT(FuncX, 0, 4);
  /* one-shot task */
  FuncY_init();
  Sched_AddT(FuncY, 50, 0);

  while (1)
   {/* do nothing! */};
}
```

Scheduler.c

```
int Sched_Init(void){
 /*- Initialise data
  *  structures.
  *- Configure periodic
  *  interrupt
  */
};

void int_handler(void){
  Sched_Schedule();
  Sched_Dispatch();
};
```

Boxes highlight changes that need to be made to previous non pre-emptable version.

Pre-emptive Multi-tasking

No changes!!

```
Scheduler.c

void Sched_Schedule(void)
{
 /* Verifies if any
  * task needs to be
  * activated, and if so,
  * increments by 1 the
  * task's pending
  * activation counter.
  */
 ...
}
```

```
Scheduler.c

void Sched_Dispatch(void)
{
 /* Verifies if any task,
  * with higher priority
  * than currently
  * executing task,
  * has an activation
  * counter > 0,
  * and if so, calls that
  * task.
  */
}
```

Pre-emptive Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks                 */
  int period;
   /* ticks until next activation */
  int delay;
   /* function pointer                */
  void (*func)(void);
   /* activation counter              */
  int exec;
} Sched_Task_t;
Sched_Task_t Tasks[NT];
int cur_task = NT;
```

One copy of this data structure for each task.

Array of structures for all tasks

Priority of currently executing task

(0 → high; (NT-1) → low) (NT → background!)

Pre-emptive Multi-tasking

No changes!!

Scheduler.h

```
typedef struct {
   /* period in ticks     */
  int period;
   /* ticks to activate   */
  int delay;
   /* function pointer    */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;
Sched_Task_t Tasks[NT];
int cur_task = NT;
```

Scheduler.c

```
void Sched_Schedule(void){
  for(int x=0; x<NT; x++) {
    if !Tasks[x].func
      continue;
    if Tasks[x].delay {
      Tasks[x].delay--;
    } else {
      /* Schedule Task */
      Tasks[x].exec++;
      Tasks[x].delay =
           Tasks[x].period;
    }
}};
```

Pre-emptive Multi-tasking

Scheduler.h

```
typedef struct {
   /* period in ticks    */
  int period;
   /* ticks to activate  */
  int delay;
   /* function pointer   */
  void (*func)(void);
   /* activation counter */
  int exec;
} Sched_Task_t;
Sched_Task_t Tasks[NT];
int cur_task = NT;
```

Scheduler.c

```
void Sched_Dispatch(void){
 int prev_task = cur_task;
 for(int x=0; x<cur_task; x++){
  if Tasks[x].exec {
    Tasks[x].exec=0;
    cur_task = x;
    enable_interrupts();
    Tasks[x].func();
    disable_interrupts();
    cur_task = prev_task;
    /*Delete if one-shot */
    if !Tasks[x].period
      Tasks[x].func = 0;
    ~~return;~~
  }}};
```

If Sched_Schedule() schedules 2 or more tasks to run, we must execute them all in the same tick!

Pre-emptive Multi-tasking

- What is missing...
  - Mechanisms for **locking** access to **shared resources**
    - e.g. mutexes, semaphores, condition variables, etc…
      possibly with priority inheritance, priority ceiling, stack resource policy, ...

    - Some of these are compatible with a single stack (stack resource policy)
    - but most require one stack per task (priority inheritance, priority ceiling…)

  - Mechanisms to **handle aperiodic** tasks
    - Add servers with adequate policy
      (e.g., periodic, polling deferrable, sporadic, total/constants bandwidth…)

  - Making the kernel **tickless**
    - Timer is no longer periodic,
      timer is configured to fire **only** when a task needs to be activated

Implementing tasks recurrent execution

1. A task (`TaskX`) that executes recurrently invokes its associated function (`FuncX()`) **every activation**
   - This is what we implemented in previous slides
   - **Persistent data** that must remain available **between** task activations (i.e. function invocations) must be declared as **static variables**!
   - **Initialization** must be done in a **separate** dedicated function
   - Allows using a **single stack** for all tasks

```
FuncX_init() {
  ...
}

FuncX() {
  static int var;

  /* func X algorithm */
  ...
}
```

Implementing tasks recurrent execution

2. A task (`TaskX`) that executes recurrently invoke its associated function (`FuncX()`) **only once**
   - This model is used by **POSIX**
   - This function will run its **initialization**, and then enter an **infinite loop**
   - The **execution** is released by a **timer** at the beginning of the loop
   - Requires one **separate stack** per task
   - **Sched_Dispatch()** is much **more complex**, since it must change the **stack pointer**, and save the current context (CPU registers, etc...)

```
FuncX() {
  int var;
  timer_t t1;
  t1 = timer_create(50);

  while (1) {
    timer_wait(t1);
    /* func X algorithm */
    ...
  }
}
```

Bibliography

- Main
  - “Patterns for Time-Triggered embedded Systems”
    Michael J. Pont, Addison-Wesley, 2001
    (Chapters 9, 11, 13, 14, 15, 16, 17)

  - “Fundamentals of Embedded Software”,
    Daniel W. Lewis, Prentice Hall, 2001

Bibliography

- Additional

How to implement a pre-emptive executive on ATMEGA µProcessors, using the second task mapping method.

- “Programming the Atmel ATmega32 in C and assembly using gcc and AVRStudio”, Dr.-Ing. Joerg Mossbrucker, EECS Department, Milwaukee School of Engineering, updated Dec. 08 2009
  (Just 4 pages long! Page 4 describes how gcc uses the CPU registers!)

- “Multitasking on an AVR –
  Example implementation of a multitasking kernel for the AVR”
  Richard Barry, March 2004