Embedded Software Architecture

Universidade do Porto
FEUP Faculdade de Engenharia

Mário de Sousa

msousa@fe.up.pt

Embedded Software Architecture

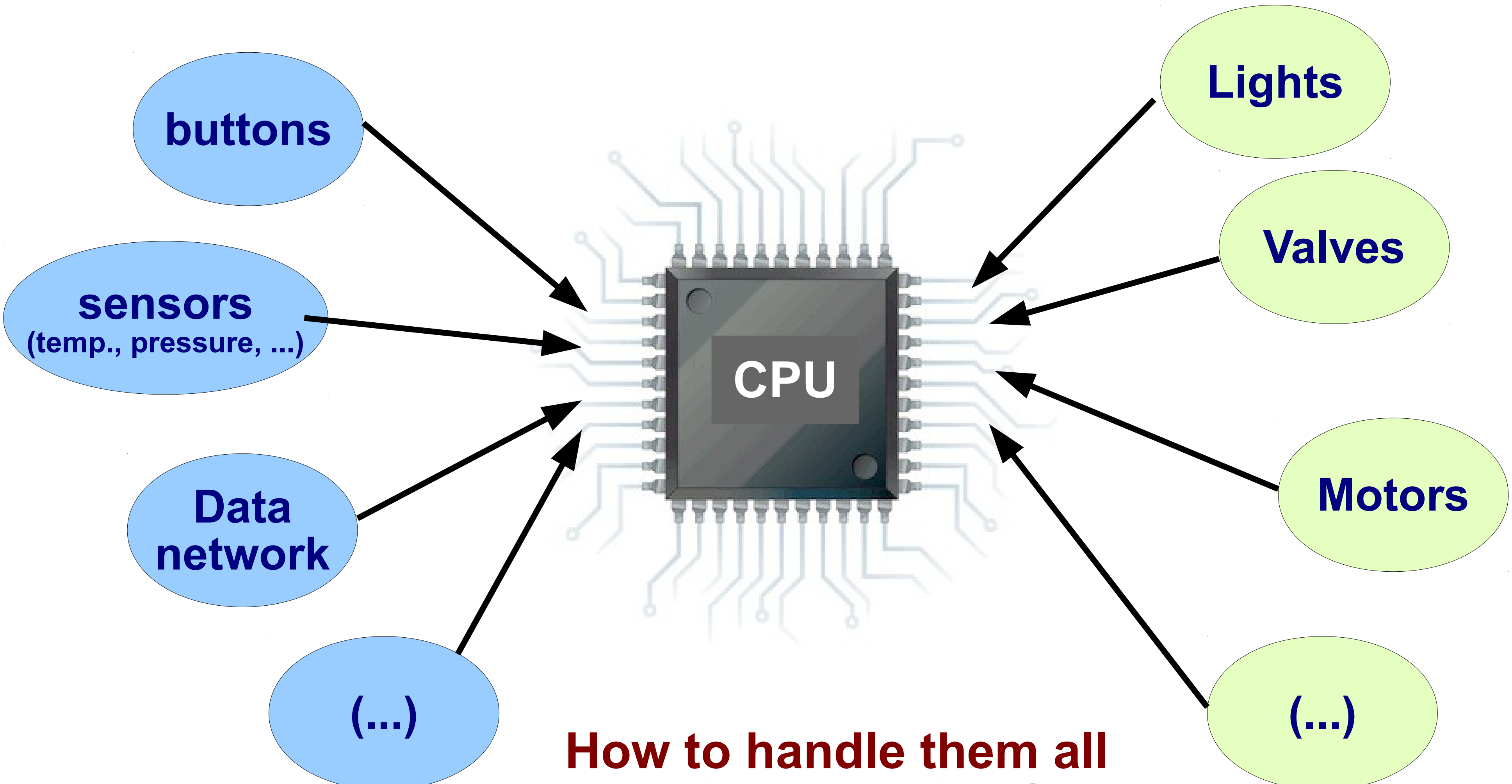
- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

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



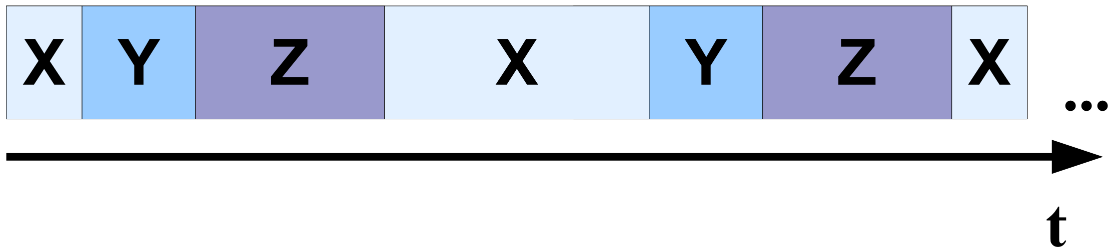
How to handle them all at the same time?

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);
```

FuncX.c

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

Cyclic Executive: Advantages

- Portability
  - Does not require special µProcessor resources (timers, interrupts, ...)
- Simple to
  - implement
  - test
  - debug
  - understand
- Deterministic => Reliable and Safe
  - Use for safety critical applications

Cyclic Executive: Drawbacks

- Difficult to guarantee exact timing delays
  - example:

    read speed every 5ms,

    update fuel/air mixture every 10ms.

- µProcessor is always active
  - May not be necessary for the current application
  - Uses up more energy

    (energy is limited resource in battery based applications)

Cyclic Executive: Introducing delays

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

  while (1) {
    /* X - Periodic Task
     *   - Period -> 200 ms
     *   - FuncX takes 10ms
     *     to execute
     */
    FuncX();
    /* Delay 190 ms */
    Busy_Sleep(190);
  }
}
```

- Waste of processing resource
  - It's best to avoid using delays...
- What if execution time of FuncX() is varies with each invocation?

Cyclic Executive: Introducing delays

- Delay using Software
  - Does not require µProcessor resources
  - Is dependent on
    - µProcessor architecture
    - Clock frequency
    - Compiler version
    - Compiler optimizations
  - Precision is dependent 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;
}
```

Cyclic Executive: Introducing delays

- 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;
}
```

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

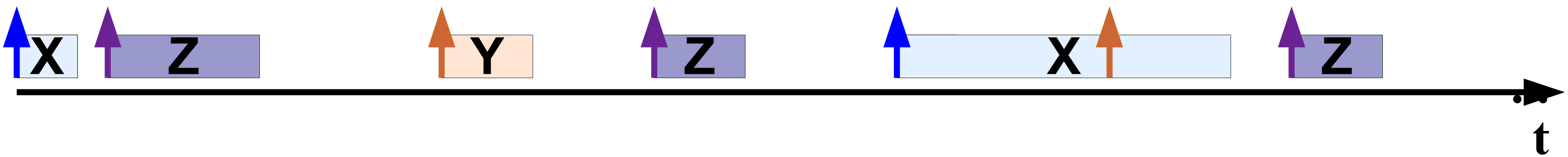
Interruption Based Executive

```
Main.c

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

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

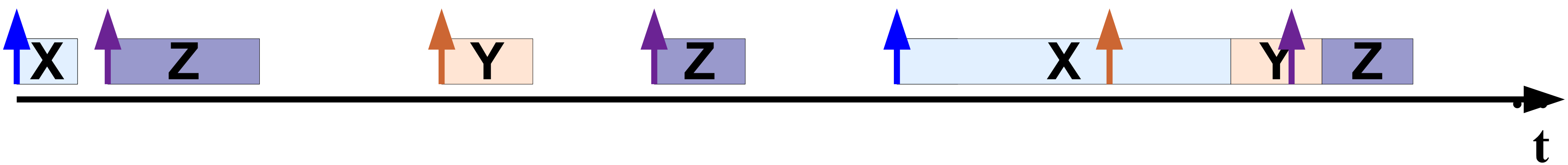
```
FuncX.c
int FuncX_init(void){
 /* Configure interrupt
  * that calls FuncX();
  */
 ...
};

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

Interruption Based Executive

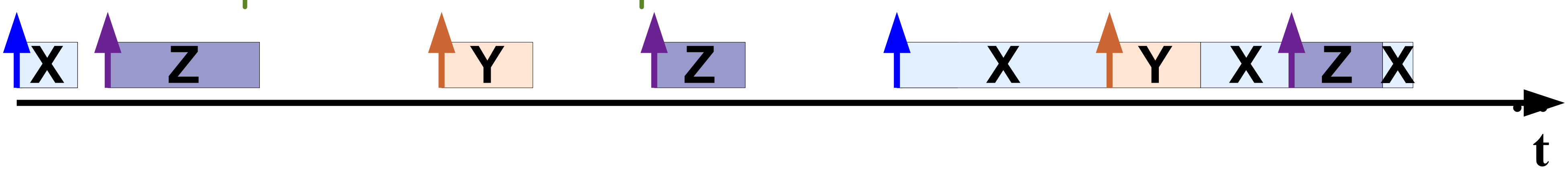
- Handling of interrupts that occur very close will depend on the hardware. They may be:
  - Ignored



  - Deferred



  - Pre-empt current interrupt routine



Interruption Based Executive

- Pre-emption...
  - ... **is good**, as it allows coexistence of very long, and very short, functions/tasks
  - ... **is bad** because of race conditions when accessing shared resources
- Some µProcessors support interrupt priorities
  - Applicable for deferred and pre-emption based handling of interrupts.
  - Usually very limited number of priorities
  - Usually priorities may not be configured; they come 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 will not be affected by task's execution time

- Drawback
  - Each periodic task requires it's 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

Universidade do Porto
FEUP Faculdade de Engenharia

Hybrid: Cyclic + Interrupts

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

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

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

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

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

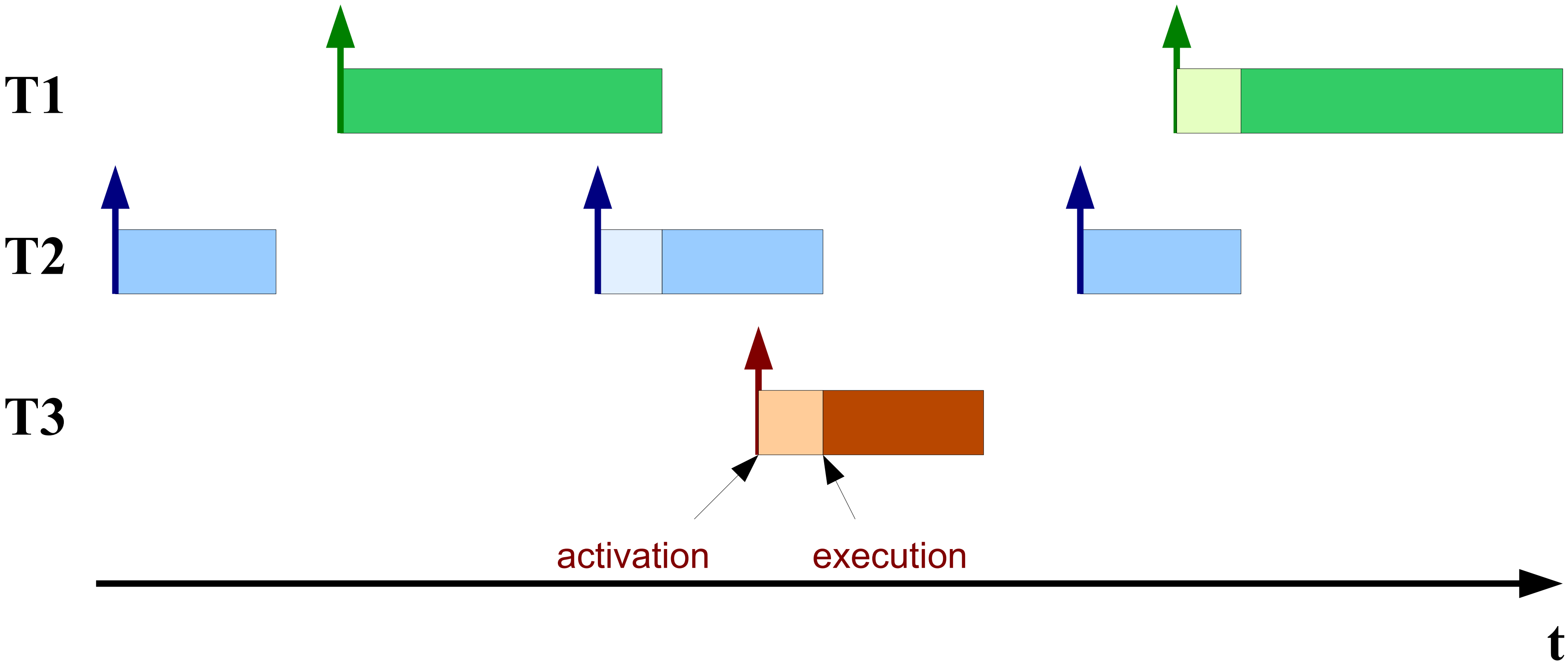
Embedded Software Architecture

- 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

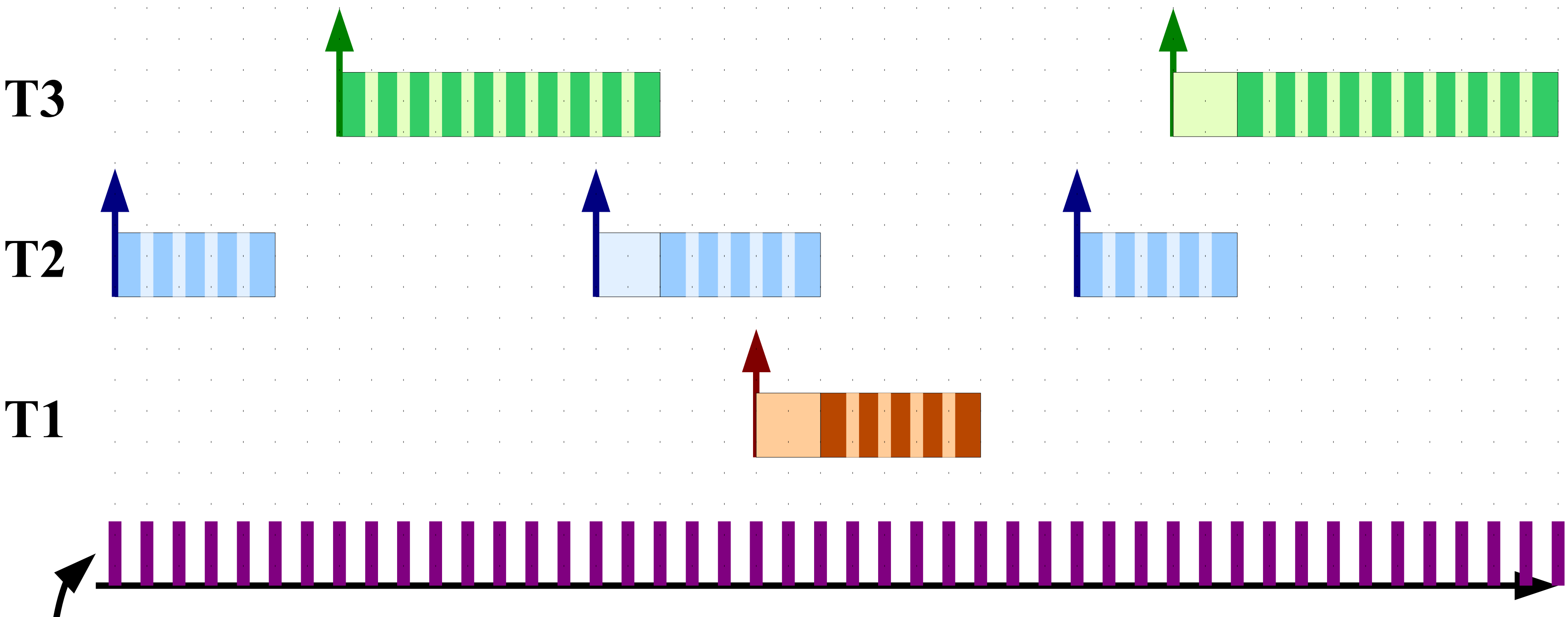
- Functions/tasks are scheduled to activate at well defined points in time
  - Periodic tasks (example: every 100 ms)
  - One-shot tasks (example: once in 10 ms)
  - *All done with a single hardware timer, that generates periodic interrupts with a frequency equal to the desired time resolution.*
- A task, once having started execution, completes this execution without being interrupted.
  - 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();
   }
}
```

Scheduler.c

```
int Sched_Init(void){
 /* - Initialise data
  *   structures.
  *
  * - Configure interrupt
  *   that periodically
  *   calls
  *   Sched_Schedule().
  */
 ...
}
```

Co-operative Multi-tasking

Scheduler.c

```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

```c
void Sched_Dispatch(void) {
 /* Verifies if any task
  * has an activation
  * counter > 0,
  * and if so, calls that
  * task.
  */
 ...
}
```

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;

Sched_Task_t Tasks[20];
```

One copy of this data structure for each task.

Array of structures for all tasks

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;

Sched_Task_t Tasks[20];
```

Scheduler.c

```
int Sched_Init(void){
  for(x=0; x<20; x++)
    Tasks[x].func = 0;
 /*
  * Also configures
  * interrupt that
  * periodically calls
  * Sched_Schedule().
  */
  ...
}
```

Co-operative Multi-tasking

Scheduler.h

```c
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[20];
```

Scheduler.c

```c
int Sched_AddT(
    void (*f)(void),
    int d, int p){
  for(int x=0; x<20; 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

```c
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[20];
```

Scheduler.c

```c
void Sched_Schedule(void){
  for(int x=0; x<20; 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;
    }}
}}
```

Universidade do Porto
FEUP Faculdade de Engenharia

Co-operative Multi-tasking

Scheduler.h

```c
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[20];
```

Scheduler.c

```c
void Sched_Dispatch(void){
  for(int x=0; x<20; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec--;
      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!

- If several tasks have multiple pending activation requests, they will run in round-robin.
(Sched_Dispatch() is called inside a while(1) loop!)

Scheduler.c

```
void Sched_Dispatch(void){
  for(int x=0; x<20; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec--;
      Tasks[x].func();
      /*  Delete task
       *  if one-shot
       */
      if(!Tasks[x].period)
        Tasks[x].func = 0;
  }
}}
```

Universidade do Porto
FEUP Faculdade de Engenharia

Co-operative Multi-tasking

Alternative is to use 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[ ] array, then we simply need to add...

- 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<20; x++) {
    if((Tasks[x].func)&&
       (Tasks[x].exec)) {
      Tasks[x].exec--;
      Tasks[x].func();
      /*  Delete task
       *  if one-shot
       */
      if(!Tasks[x].period)
        Tasks[x].func = 0;
      return;
}}}
```

Co-operative Multi-tasking

Alternative is to use 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<20; 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

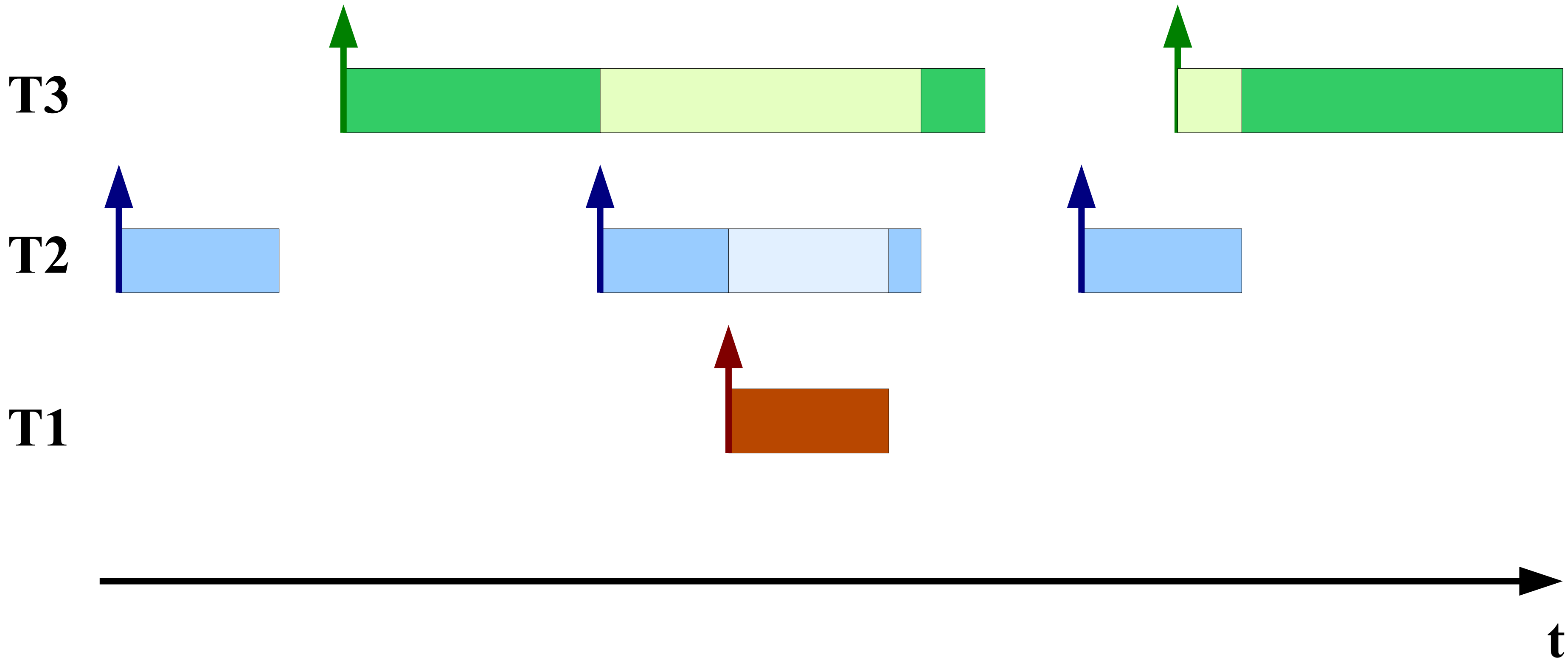
- The components of an embedded system
- Cyclic Executive
- Interruption based Executive
- Hybrid: Cyclic + Interrupts
- Co-operative Multi-tasking
- Pre-emptive Multi-tasking

Universidade do Porto
FEUP Faculdade de Engenharia

Pre-emptive Multi-tasking

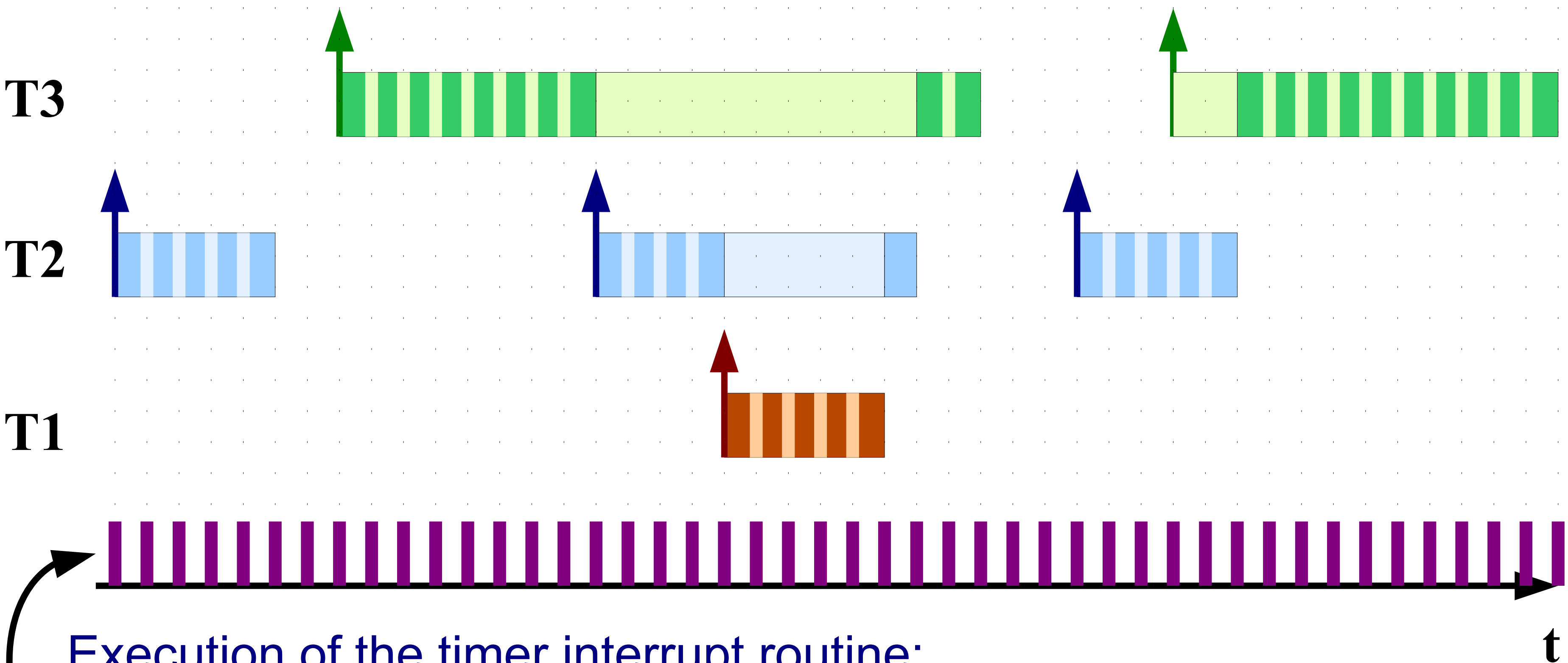
- Functions/tasks are scheduled to activate at well defined points in time
  - Periodic tasks (example: every 100 ms)
  - One-shot tasks (example: once in 10 ms)
- A task, once having started execution, may be interrupted by another task...
  - ... that has just been activated.
  - The decision whether or not to interrupt the currently executing task will depend on the priority of both these tasks.

Pre-emptive Multi-tasking



T3 – periodic – low priority
T2 – periodic – medium 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 timing, so it is possible to have 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

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 interrupt
  *   that periodically
  *   calls int_handler()*/
};
void int_handler(void){
  disable_interrupts();
  Sched_Schedule();
  Sched_Dispatch();
  enable_interrupts();
};
```

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

Pre-emptive Multi-tasking

NOTE:

Calling
disable_interrupts()
at beginning of interrupt handler, and
enable_interrupts()
at end of handler, is usually not necessary, as often the compiler inserts them automatically when the function is declared as an interrupt handler.

Scheduler.c

```
int Sched_Init(void){
 /*- Initialise data
  *   structures.
  *- Configure interrupt
  *   that periodically
  *   calls int_handler()*/
};
void int_handler(void){
  disable_interrupts();
  Sched_Schedule();
  Sched_Dispatch();
  enable_interrupts();
};
```

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[20];
int cur_task = 20;
```

One copy of this data structure for each task.

← Array of structures for all tasks

← Priority of currently executing task
(0 → high; 19 → low) (20 → no running task!)

Pre-emptive Multi-tasking

No changes!!

Scheduler.h

```c
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[20];
int cur_task = 20;
```

Scheduler.c

```c
void Sched_Schedule(void){
  for(x=0; x<20; 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[20];
int cur_task = 20;
```

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

Scheduler.c

```
void Sched_Dispatch(void){
 int prev_task = cur_task;
 for(x=0; x<cur_task; x++){
  if Tasks[x].exec {
    Tasks[x].exec--;
    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;
 }}};
```

Pre-emptive Multi-tasking

- What is missing...
  - Mechanisms for locking access to shared resources
    - mutex, semaphore, condition variables, etc....
    - If any of the above mechanisms are implemented, then priority inversion may occur!
  - Mechanisms to limit priority inversion
    - Basically, add more complex scheduling mechanisms
      (one of the many versions of the priority inheritance scheduling mechanisms:
      basic priority inheritance, immediate priority inheritance, priority ceiling, stack
      resource policy).
    - Note that the above implementation uses a single stack for running all tasks, so the only safe version of priority inheritance protocol is the Stack Resource Policy!!

Multi-tasking

... some more variations ...

- There are 2 methods of mapping Tasks onto Functions:
  - (1) A task (**`TaskX`**) that executes periodically, is mapped onto a function that is called periodically (**`FuncX()`**)
    - This is what we implemented above!
    - Persistent data that must remain available between task activations (i.e. function invocations) must therefore be declared as static variables!

```
FuncX() {
  static int var;

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

Multi-tasking

... some more variations ...

- There are 2 methods of mapping Tasks onto Functions:

  - (2) A task (**TaskX**) that executes periodically, is mapped onto a function (**FuncX()**) that is called once!
    - This model is used by POSIX !
    - This function will execute an infinite loop, with each iteration of the loop corresponding to one periodic activation of the task;
    - At the beginning of the loop, we insert a call to a timer. The call will block, and will be released periodically by the operating system!
    - Each task/function need its own stack memory! Sched_Dispatch() is therefore 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