Embedded and Ambient Systems
2021.09.28.

SW architectures of embedded systems

Méréstechnika és
Információs Rendszerek
Tanszék

Budapest University of Technology and Economics
Department of Measurement and Information Systems

SW development alternatives

- Resources! (CPU, MEM, Energy)
- Different approach compared to a PC: HW-based programming
- Direct handling of:
  - Polling
  - Interrupt (IT)
- Low level programming (Assembly)
  - To solve less complex tasks
  - Time critical applications
  - Difficult development and debugging
  - Exploiting special peripheral
- High level programming (C, C++, Java?)
  - Less efficient (not always)
    - Some specialties are difficult to understand by humans, e.g. delayed branch, pipeline design…
  - Faster development, reengineering and scalability
  - ASM code parts can be inserted in a C-language environment
- Embedded operation system
- Graphical programming languages, automatic code generation

Services

- Basic tasks
  - Observations
  - Handling peripherals
  - Handling events
  - Timing
  - Communications
  - Data processing
- Problem:
  - Processor: sequential operation
  - Events: occur in an asynchronous manner, overlapped in time
- Various requirements (on program structure):
  - E.g. the program of the microwave oven is finished. Not a critical application, e.g. 1s delay is not even noticed
  - Direction indicator in a car: not that much time critical but safety critical therefore the requirements are more severe
  - Braking system in a car is strongly time- and safety critical (1s delay does matter)
- Handling of tasks has to be planned (process scheduling)

Considerations of program structure used

- Considerations:
  - Resources available / softver-overhead
    - Overhead due to extra computation of process scheduling
  - Memory (storage capacity) available (RAM, ROM)
  - Predictability (planning of the SW system in advance)
  - Scalability, re-engineering
    - Need for extra development due to inserting a new task
  - Time needed for executing a task
  - Reaction time for an external asynchronous event
  - Prioritization of tasks
  - Usage of processor
    - Energy saving operation, how much the resources of the processor is exploited
  - Protection (memory, run time)
  - Recursion, support of function (re)calls
  - Implementation of HW handling
  - Implementation of communications between tasks
  - Application field (e.g. consumer electronics, automotive industry)

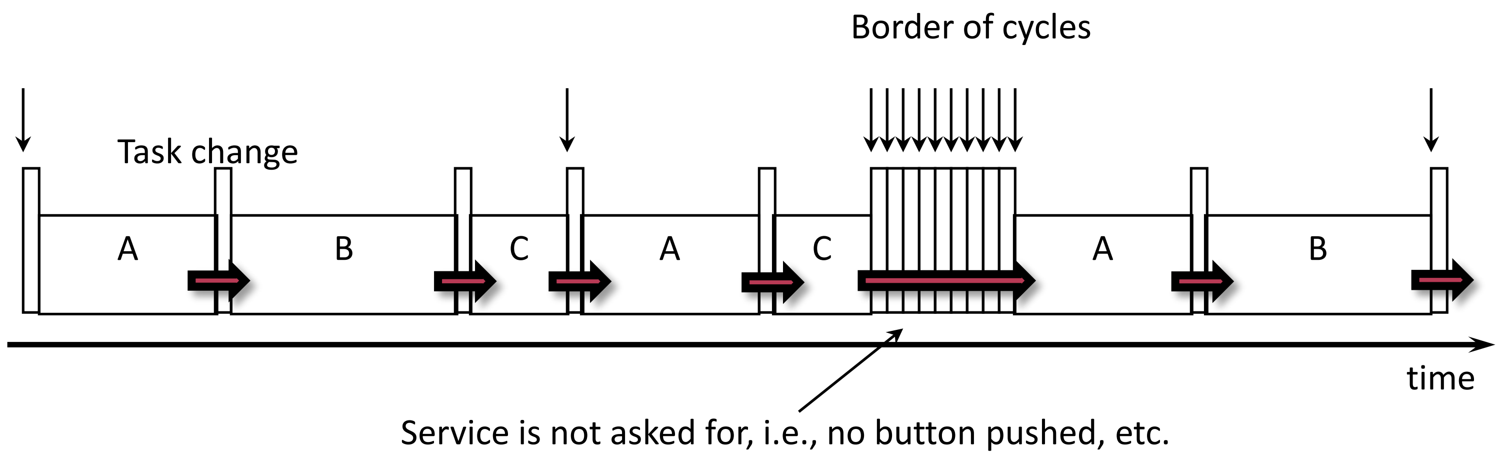
Program structuring disciplines

- Cyclic programming
  - Simple cyclic
  - Weighted cyclic
  - Time-controlled cyclic
  - Strict time-controlled cyclic
- Cyclic process scheduling with interrupt (IT)
- Scheduled functions

Simple cyclic program structure

- Tasks are executed one after the other in a cyclic manner (e.g. bicycle computer)

```
void main() {
while (TRUE){
        if (button1_pushed==true) {change_menu() ;}
        if (sensor_state==active) {calculate_speed() ;}
         if (speed_calculated==true) {display_speed();}
        ...
        }
}
```



(but the processor runs /checking if a process should be run or not/->not energy friendly!)

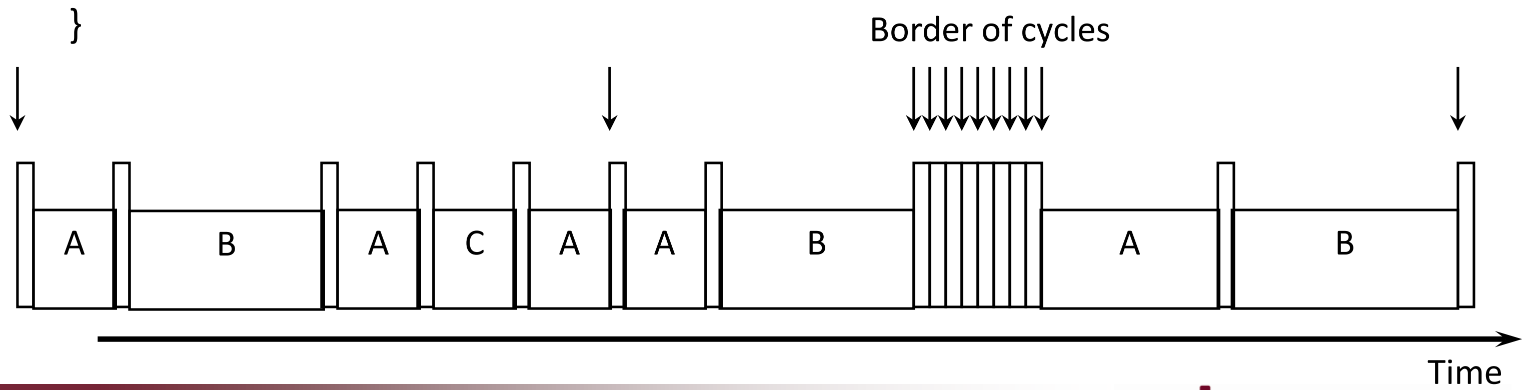
Simple cyclic program structure

- Simple structure
- Communications between tasks:
  - Shared variables, no problem since the are not preemptive: only one task runs at a time
- Scalability:
  - Pros: simple structure, fast development at the beginning
  - Cons: fixed structure
- HW handling: polling (not IT)
- If a new task is inserted the response time is increased
- Not preemptive (only one task runs until it finishes its job)
  - Mutual exclusion is not a problem (more than one process cannot run)
  - A long lasting process can block the running of others
- Applicable only where response time is not critical
- Not energy friendly since the processor operates continuously

Weighted cyclic program structure

- The tasks are executed one after the other in a cyclic manner, but certain tasks are checked more frequently to make it run or not

```
void main() {
while (TRUE){
        if (sensor_state==active) {calculate_speed() ;}
        if (sensor_state==active) {calculate_speed() ;}
        if (button1_pushed==true) {change_menu() ;}
        if (sensor_state==active) {calculate_speed() ;}
        if (sensor_state==active) {calculate_speed() ;}
        if (button1_pushed==true) {change_menu() ;}
         if (speed_calculated==true) {display_speed();}
...
        }
}
```



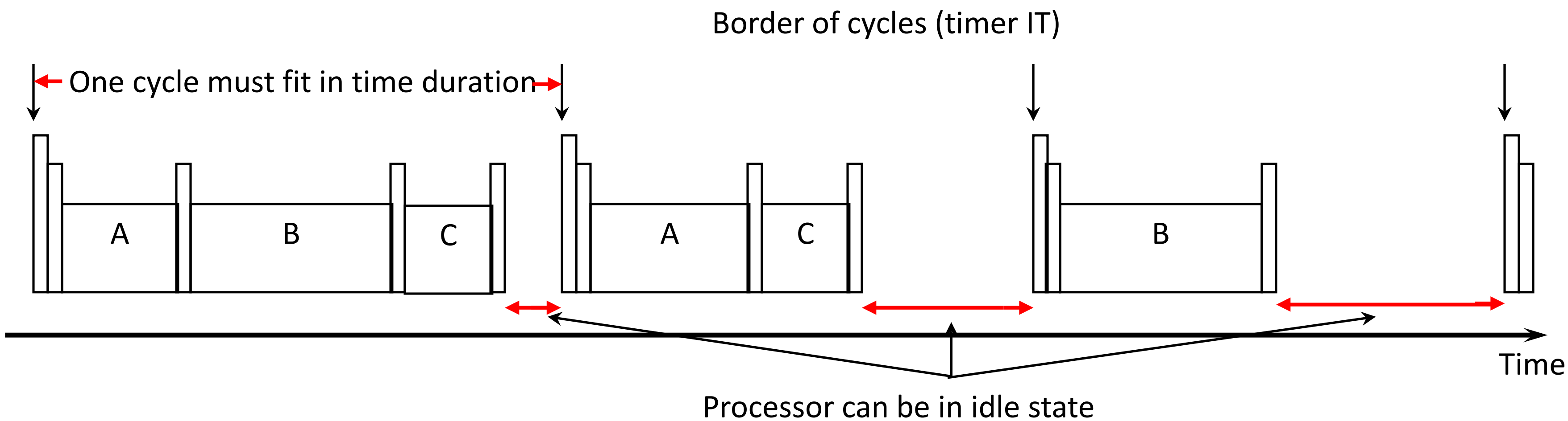
Weighted cyclic program structure

- Simple structure
- Communications between tasks:
  - Shared variables, no problem since the are not preemptive: only one task runs at a time
- Scalability:
  - Pros: simple structure, fast development at the beginning
  - Cons: fixed structure
- HW handling: polling (not IT)
- If a new task is inserted the response time is increased
- Not preemptive (only one task runs until it finishes its job)
  - Mutual exclusion is not a problem (more than one process cannot run)
  - A long lasting process can block the running of others
- Applicable only where response time is not critical
- Not energy friendly since the processor operates continuously
- **A basic level of priority can be assured**

Time-controlled cyclic program structure

- Polling is not continuous but controlled by a timer
- In a time-controlled cycle the structure can be simple cyclic or weighted cyclic

```
TimerITServiceRoutine(){
        if (button1_pushed==true) {change_menu();}
        if (sensor_state==active) {calculate_speed();}
        if (speed_calculated==true) {display_speed();}
}
```

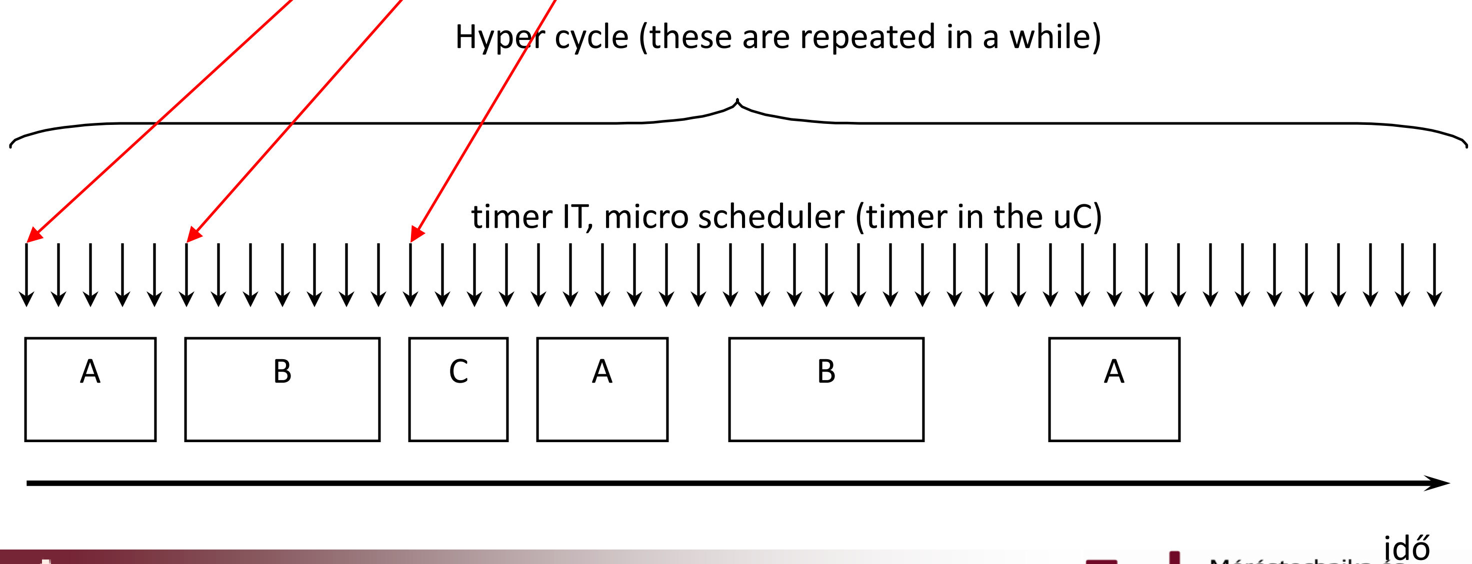


Time-controlled cyclic program structure

- During one cycle the properties of simple cyclic and weighted cyclic structures are valid here
- Good choice for systems using scheduled control, e.g., sampling signal processing systems
- Cycle time must be less than the required response time
  - Run time of a cycle must fit between two timer IT
- Advantage over simple cyclic and weighted cyclic structures is energy friendly operation
  - Processor can be in idle state between the executed tasks and next timer IT

Strict time-controlled cyclic structure

- The execution of each task starts at a scheduled time in a strict sense
- Administration:
  - In a table: time instants and function references (in hyper cycle)
  - The operating system or scheduler supervise the time instants and starts the "tasks"



Strict time-controlled cyclic structure

- Scalability:
  - Pros: start of running can be calculated precisely
  - Cons: inserting a new task requires re-scheduling every other tasks
- HW handling: polling
- Non-preemptive: one task runs at a time
  - No problem with shared variables
- Every task must fit in its assigned time slot
  - The run time of every task must be known (at least its possible worst case runtime)
- Good for real-time systems: strict timings

Cyclic process scheduling with interrupt (IT)

```
FLAG        button, sensor;
void interrupt Button_IT_Handler() { Button_fast_A(); button=TRUE; }
void interrupt Sensor_IT_Handler() { Sensor_fast(); sensor=TRUE; }
void main() {
 while (TRUE){
            if (button)  {button=FALSE; Service_button(); }
            if (sensor)  {sensor =FALSE; Service_ sensor(); }
             display_speed();}
            ...
            }
}
```

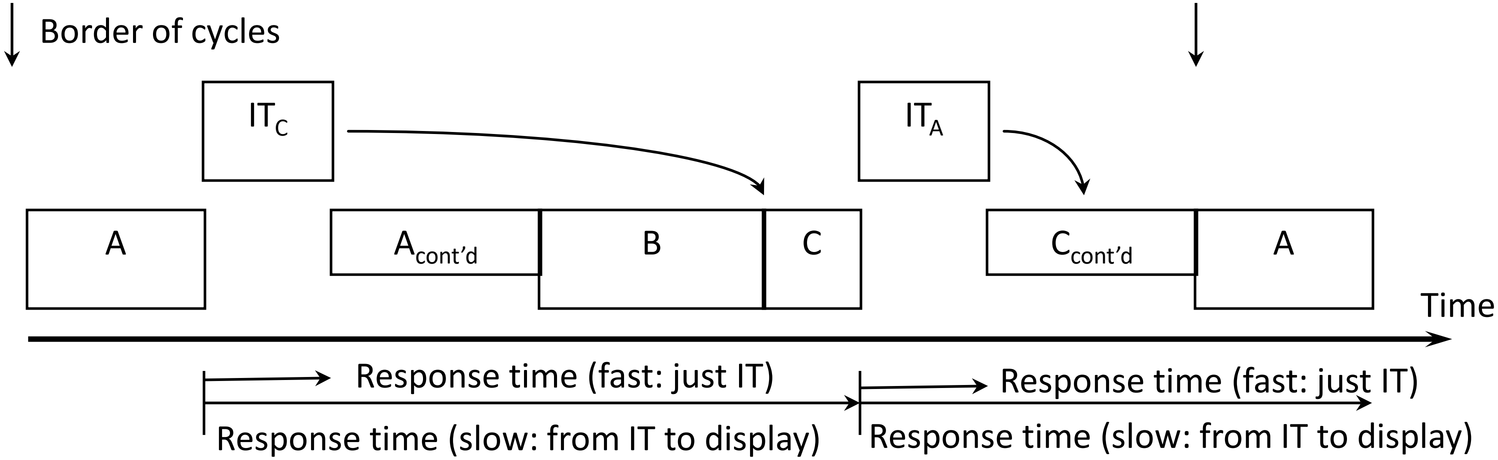
Low-level events are handled by interrupts
E.g. magnetic sensor position

Time-stamp of sensor=TRUE is known and saved

Simple cyclic

Calculation of speed based on time-stamp can happen later

Time critical: exactly when the sensor passed
Non-time critical: calculation and dislay speed



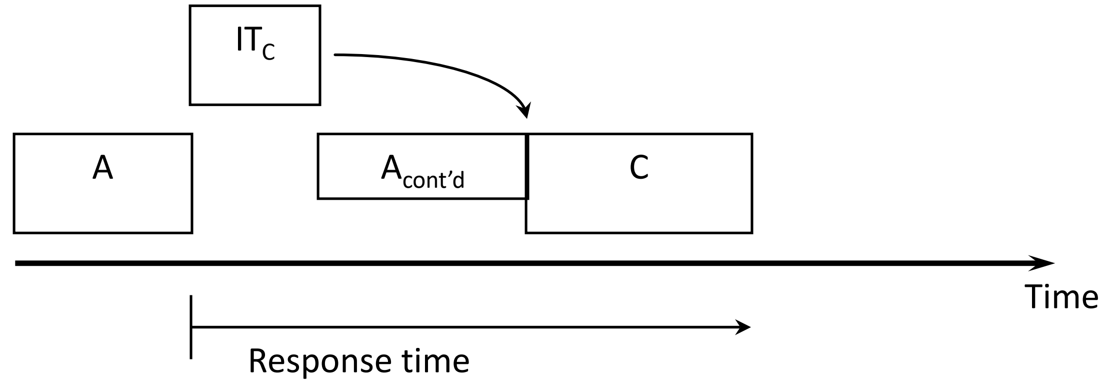
Cyclic process scheduling with interrupt (IT)

- IT (interrupt) is needed when polling is not enough since the application is time-critical
  - Independently, certain peripherals can be handled by polling
- Deterministic behavior is not true any more
  - IT may happen any time and program have to tolerate it
- Mutual exclusion must be assured for interrupts
  - Not to overwrite a variable during interrupt
- Response time is increased by time of interrupts
- Frequently applied solution (expected in many cases)
- Inserting a new task increases response time
- IT routine: execute only the most important tasks, further processing can be done later

Scheduled functions

- Every task is implemented in a function
- In case of an event (like interrupt) to execute the function, the function is put in a function queue
- If a function to be executed exists then the scheduler calls that from the queue
- Uniform function format is used

```
void interrupt Button_IT_Handler() { Button_fast_A(); PutFunction(Service_button))}
void interrupt Sensor_IT_Handler() { Sensor_fast(); PutFunction(Service_sensor)}
void interrupt display_timer_IT_Handler() {PutFunction(Service_display_timer)}
void Service_button();
void Service_sensor();
void Service_display_timer();
void main() {
  while (TRUE){
     while (IsFunctionQueueEmpty()) ;
     CallFirstFromQueue();
   }
}
```



Scheduled functions

- HW handling: interrupt
- Communications between tasks:
  - Task – task : no problem
  - Task – IT: mutual exclusion must be assured-take care of shared variables
- Scalability:
  - Inserting a new task is easy
  - The running environment requires extra care
- Calling from the function queue:
  - FIFO
  - Based on priority
- Operation is similar to embedded systems

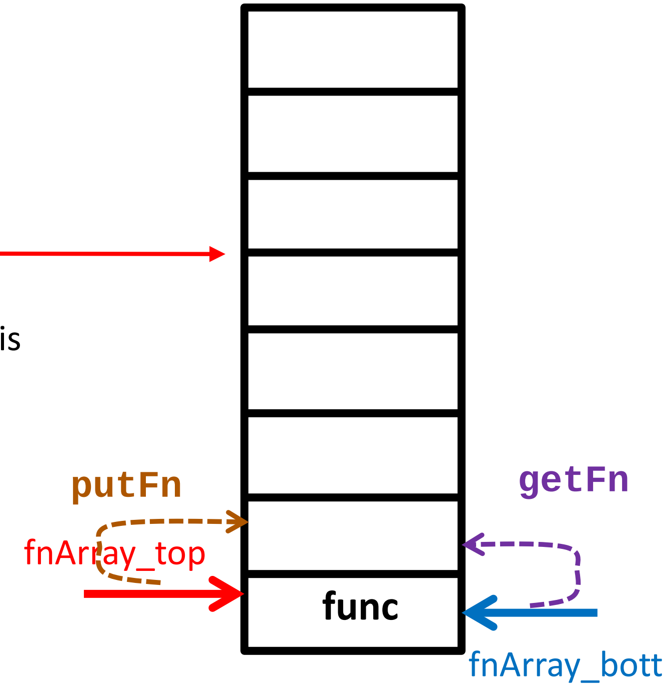
Implementation of scheduled functions

- A possible implementation

```
typedef void (*fp)(void);
fp functionToCall;
#define N_FN 8
#define N_FN_MASK (N_FN-1)
fp fnArray[N_FN];
uint16_t fnArray_top=0;
uint16_t fnArray_bott=0;
```

Function pointer

Array of function pointers




```
void putFn(void (*func)(void)){
     fnArray[fnArray_top] = func;
     fnArray_top = (fnArray_top+1)&(N_FN_MASK);
}
```

Pointer of new function is placed in the array

```
int32_t getFn(){
     int32_t retVal;
     fp functionToCall;
     if (fnArray_top == fnArray_bott){
          retVal = -1;
     } else {
          retVal = fnArray_bott;
          fnArray_bott = (fnArray_bott+1)&(N_FN_MASK);
          functionToCall = fnArray[retVal];
          functionToCall();
     }
     return (retVal);
}
```

Check if new function exists or not

No new function

New function exists

Function is called

Considerations

- Choose the simplest scheduling method that is still able to meet the requirements
- Task scheduling has to be planned carefully since change of concept or even inserting a new task may lead to huge extra work

Interrupts definition

- Running of the program is interrupted due to an external event, and the code that belongs to the interrupting event starts running
- The code of the interrupting event is “inserted” into the main program
- Returning of the main program from the interrupted state the main program should not “notice” that it had been interrupted. To assure that:
  - Work registers have to be restored
  - Processor status registers have to be restored
  - Stack has to be restored
  - In short: context change has to be done
- In embedded systems: several different architectures and solutions exist, therefore general considerations has to be completed in a device-specific manner