Design of Real-Time Software

Implementing the real –time task model

By courtesy of Prof. Dr. Johan J. Lukkien
Department of Mathematics and Computer Science
System Architecture and Networking
Eindhoven University of Technology

Real-time software structure

- How does a “real-time” program look?
  - in the end, the problem analysis yields a set of tasks with deadlines; these must be encoded into computer code

- What is the typical platform that a programmer encounters? E.g.,

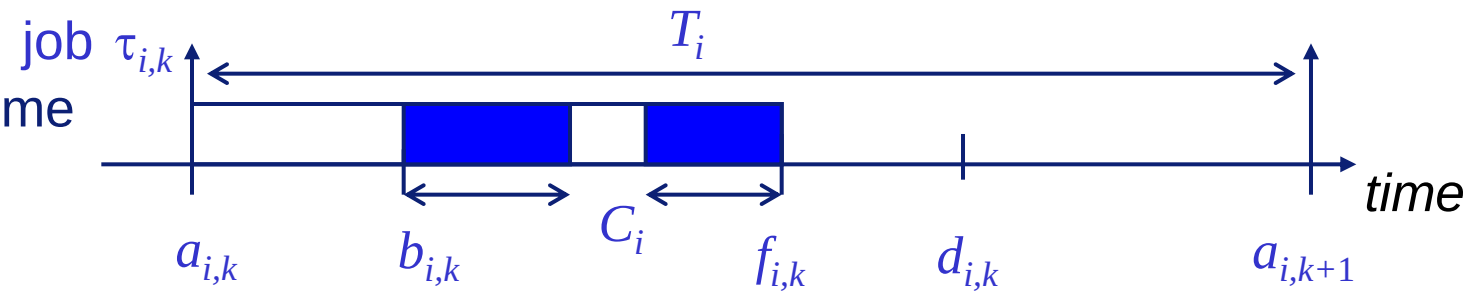
  - just very basic hardware abstraction
    - device registers, interrupts

  - a library, representing a better abstraction and giving some supporting routines
    - typically for creating and managing *tasks*
    - terminology: a (*real-time) executive:*
      - the executive is the result of combining such a library with user code
      - but also: the entire code that is loaded into an embedded system

  - an operating system (could be resident, could be linked to the executive)
    - providing the notion of *concurrently* executing entities: processes, threads
      - and other high-level abstractions like file systems and memory management
    - adding *spatial isolation,* and possibly also *temporal isolation* for these concurrent entities

Performance metrics to evaluate an implementation

- Event latency: the difference between arrival time and begin time
- Response time of library or OS calls

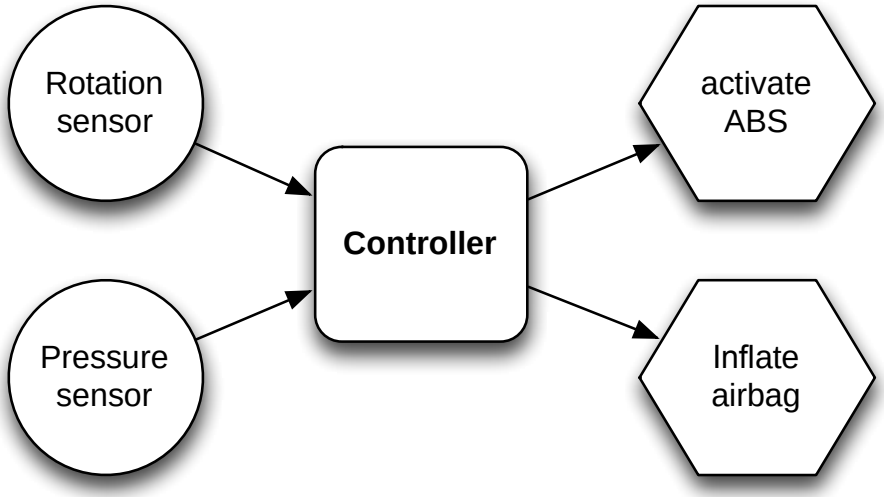


- Jitter – variations
  - on event latency ($b_{i,k}$)
  - on (periodic) task release ($a_{i,k}$)
  - on library or OS calls (affecting C_i)
  - a large jitter:
    - limits accuracy
    - leads to bad worst-case estimates

- These figures determine the time granularity and accuracy of the system
  - a system with an event latency of 1ms cannot deal with 10 μsec periodic events
  - a task release jitter of 50% leads to a safety boundary of 50%

Example: read two sensors

- Two tasks:
  - Task1: Read rotation sensor to detect skidding and activate ABS if threshold crossed
  - Task2: Read pressure sensor to detect a collision and inflate airbag if a threshold crossed




```
void Task1() {
  a = ReadRotation();
  if (a > A) ActivateABS();
}

void Task2() {
  p = ReadPressure();
  if (p > P) InflateAirbag();
}
```

Agenda

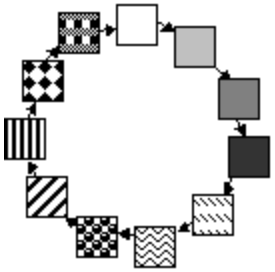
- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

Single executive (“cyclic executive”)

- No supporting system (OS) assumed
- Application is written as a single repetition executing the tasks repeatedly
- Several variants:
  - AFAP – as fast as possible
  - time driven AFAP
  - multiple rates AFAP
  - multiple rates periodic
- Shortcomings in
  - interrupt handling (through “polling”)
    - handling is one of the tasks
    - or ISR is just registration of occurrence, complete handling not possible
  - effective coding of a tasks’ relative importance
  - dynamicity
    - in variations of timing properties of the tasks
    - in less predictable environments
  - code structure, maintenance, update

AFAP

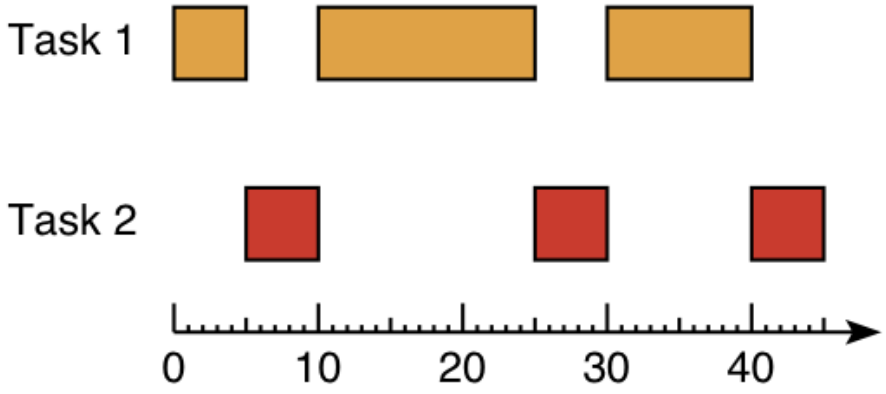
```
while (true) {
    Task1();        /* each task probably starting with a test on
    Task2();         * whether work is needed
    Task3();         */
    .....
    Taskn();
}
```



Pictures by D. Kalinsky

- Latency and jitter: (worst-case) loop length
  - incurs drift: summing jitter over time
  - impossible to predict accurately when a task executes
- ‘Busy’ waiting – may waste energy
- Scheduling and programming are mixed
  - static importance level: all tasks in principle equally important
- Time analysis and execution model are entangled
  - adding a task is difficult
- Communication via global data

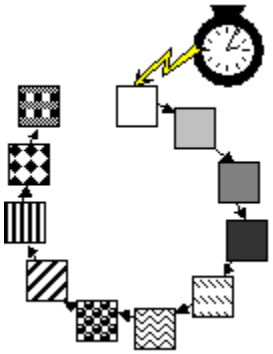
Example: read two sensors (AFAP)




```
while (true) {
  Task1();
  Task2();
}
```

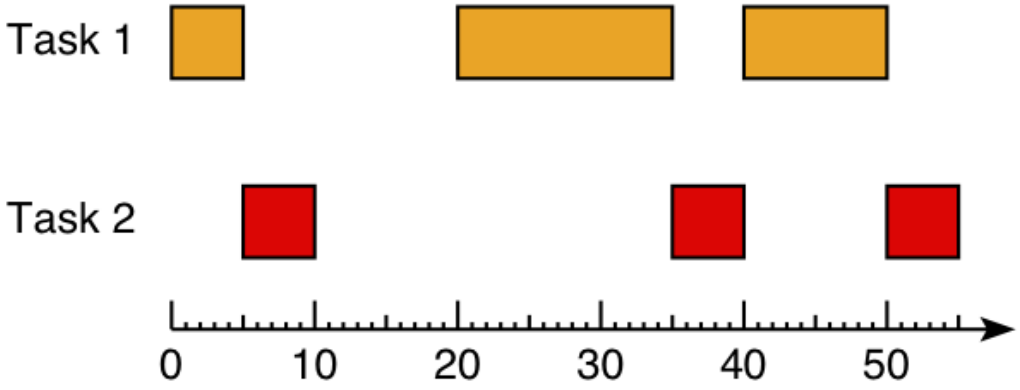
Time-driven AFAP

```
Upon timer event:
    Task1();        /* each task probably starting with a test on
    Task2();         * whether work is needed
    Task3();         */
    .....
    Taskn();
    'Give up execution (until next timer event)';
```



- Timer starts the task sequence
- Notion of periodic execution
  - same period for all tasks
- More predictable
  - no drift
  - limited release jitter
  - timing of one task heavily dependent on computation times of other tasks

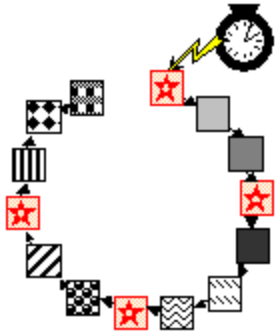
Example: read two sensors (time-driven AFAP, period 20)



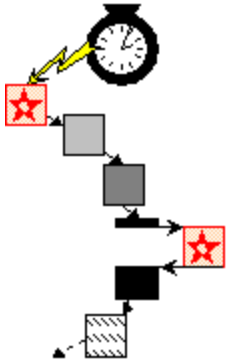

```
while (true) {
   'wait for timer';
   Task1();
   Task2();
}
```

Multi-rate time-driven AFAP

```
Upon timer event:
    Task1();        /* each task probably starting with a test on
    Task2();           * whether work is needed
    Task3();           */
    Task1();
    .....
    Taskn();
    'Give up execution (until next timer event)';
```



- *Task1()* executes four times per period, hence four times as fast (see picture)
- Still, significant jitter and heavy inter-dependence of tasks
  - possible to make this more accurate by splitting tasks
  - ... needs very precise timing
  - ... nightmare for maintenance
  - ... better be outcome of a computation



Multi-rate periodic

Upon timer event:

CurrrentTime = Timer;

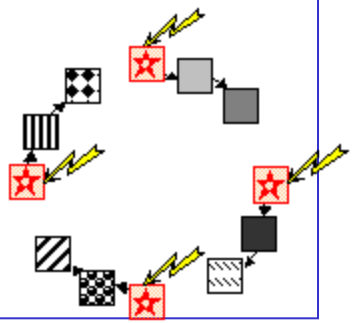
if (*IsTimeFor (CurrentTime, 1)*) *Task1();*

if (*IsTimeFor (CurrentTime, 2)*) *Task2();*

if (*IsTimeFor (CurrentTime, 3)*) *Task3();*

.....

'Give up execution (until next timer event)';

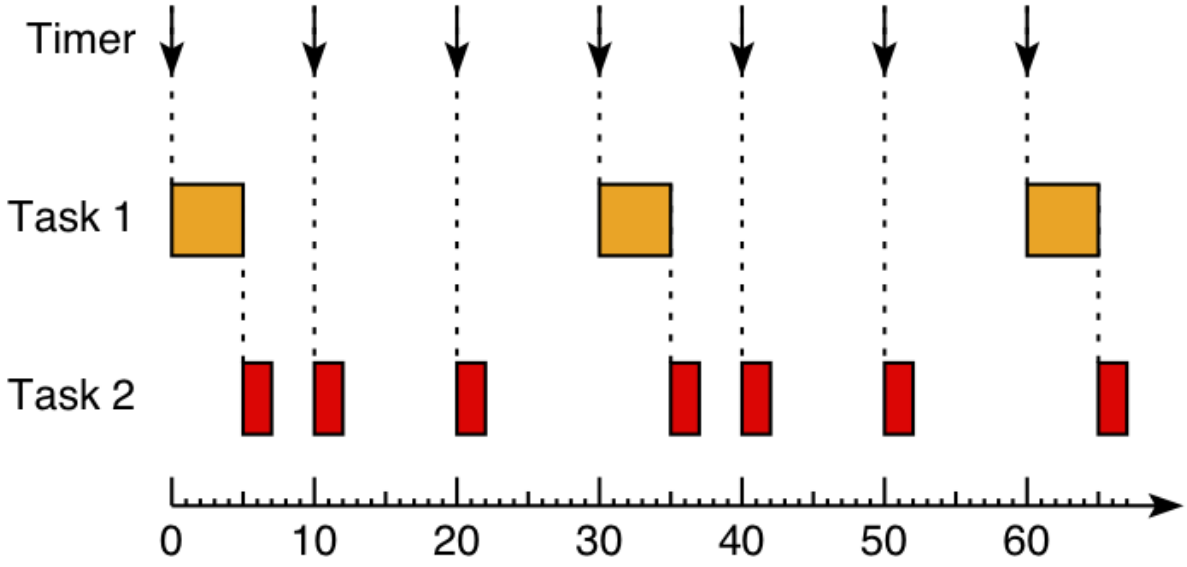


- *IsTimeFor()* determines for a task whether it should run at the current time
  - extreme case: just a single task per timer event
  - implementation: use queues, associated with the timer

- Requires timer to 'fire' at greatest common divisor of task periods
  - the overall period is then the least common multiple of these task periods

- Needs careful calibration to make sure tasks complete in time
- By manipulating the *phasing* (= the time the first occurrence is started), a better spread in time can be achieved
  - at the expense of a higher timer interrupt rate
  - leads to table-driven execution, the table gives the precise times when tasks are to be executed

Example: read two sensors (multi-rate periodic)




```
while (true) {
   'wait for timer';
   if (IsTimefor(1)) Task1();
   if (IsTimeFor(2)) Task2();
}
```

Summarizing, cyclic executive

- Pro's
  - tasks execute strictly sequentially
    - no interference or *race conditions*
    - limited problems due to sharing
  - in principle, very simple implementation
    - no advanced platform requirements
- Con's
  - for precise timing, very precise analysis and tuning is needed
    - combinatorial problem to compute the table
  - difficult to deal with variations in computation times
  - idle times are lost, cannot be re-used easily
    - e.g. the notion of a 'background task'
  - dynamic addition of new tasks cumbersome
  - poor (non-optimal) response time
    - since jobs run to completion, it is possible to find two jobs of any task such that a job of any other task is in between
    - solution: subdivide tasks

Exercises

- **C.1** Given is a system with 3 tasks, with (period, computation time) pairs of (4,1), (8,1), (16,1).
  - Give timer interrupt frequency and give function *IsTimeFor(),* for the case of the tasks having the same phasing.
  - Determine response times – worst case for this phasing.
  - Try to find an optimal phasing.

- **C.2** Consider a system with two tasks, one with a period of 2 and a computation time of 1 and the other with a period of 20 and a computation time of 5.
  - Give timer interrupt frequency and give function *IsTimeFor(),* for the case of the tasks having the same phasing.
  - What is the problem? Solve it by proposing subtasks.
  - Determine response times – worst case for this phasing.
  - Try to find an optimal phasing.

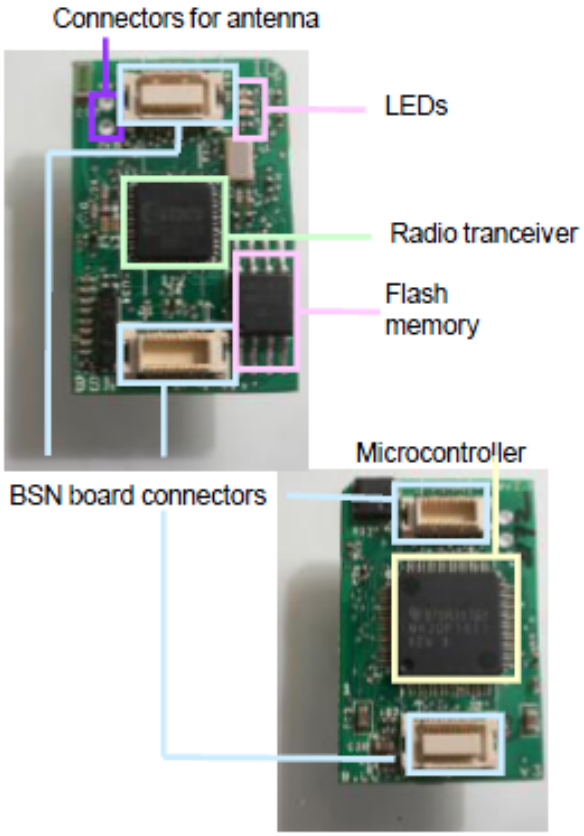
- **C.3** Discuss C.2 again but now with task 1 having a period of 6. Is there still a need for subtasks?

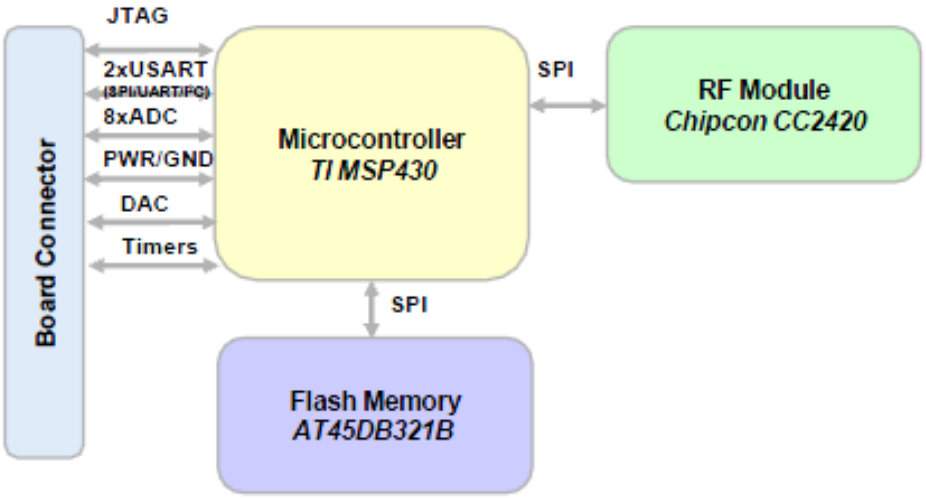
Agenda

- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

BSN node

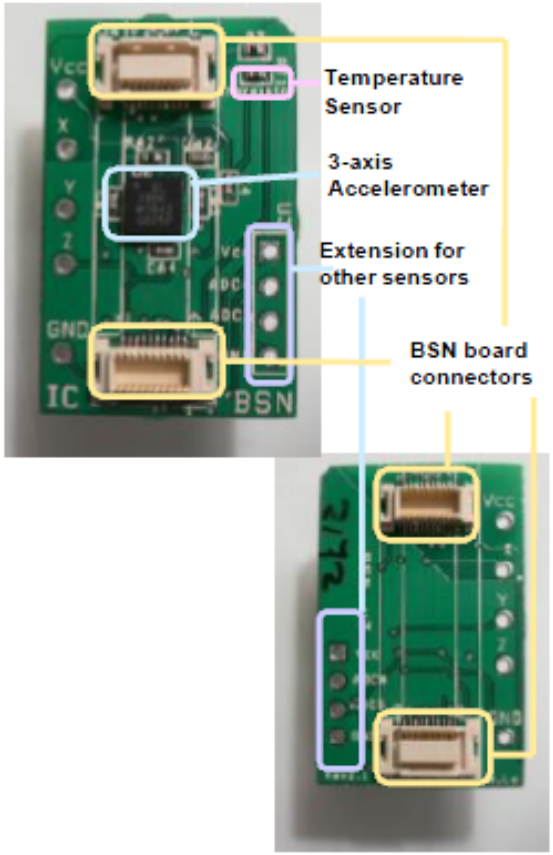
- The BSN node consists of 6 major components:
  - Microcontroller – TI MSP430F1611
  - Radio transceiver – Chipcon CC2420
  - Flash memory – Atmel AT45DB321DB
  - LEDs
  - BSN board connectors
  - Mounting for antenna





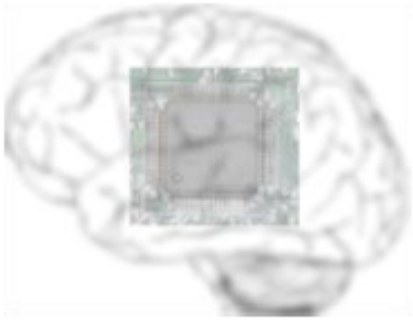
Sensor board

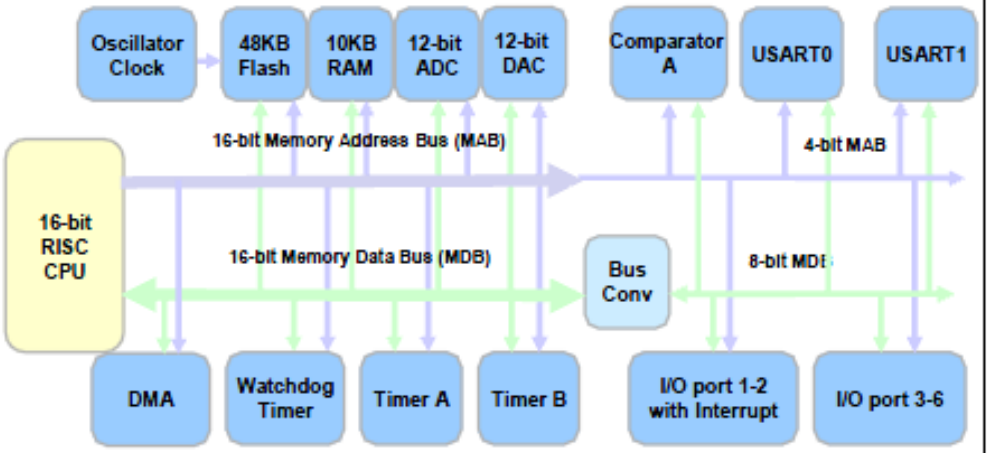
- The sensor board consists of the following
  - 3D accelerometer
  - Temperature sensor
  - BSN board connectors
  - Extension for an additional sensors
- Both accelerometer and the temperature sensor obtain power from the "SENSOR_PWR" pin of the BSN board connector and the pin "SENSOR_PWR" can be activated or deactivated by the software. As such, sensors can be switched on or off to save power.
- The Analog Device ADXL 330 3D accelerometer is used, and the analog outputs of the 3 axes are connected to channels ADC2 (X axis), ADC3 (Y axis) and ADC0 (Z axis)
- The Panasonic ERTJ1VR103J temperature sensor is used and it is connected to channel ADC1



BSN node
Microcontroller – MSP430F1611

- Low supply-voltage 1.8V
- Ultra low-power consumption
  - Active mode: 330uA at 1MHz, 2.2V
  - Standby mode: 1.1uA
  - Off Mode (RAM Retention): 0.2uA
- Fast wake-up time <6us
- 16-bit RISC with 125ns Instruction Cycle Time
- 8 x 12-bits A/D converter
- 2 x 12-bits D/A converter
- 16-bit timer
- 2x Serial communication interface (USART)
- 48KB+256B Flash memory
- 10KB RAM
- 3 Channels internal DMA





Output: use the three leds on the board

- Files: Led.h, Led.c

void InitLeds (set of Leds)
void SetLeds (set of Leds, uint8_t on)
void ToggleLeds (set of Leds)

- *InitLeds () c*alled at the beginning of the program

- Examples
  - *InitLeds (RED | GREEN | YELLOW)*
  - *SetLeds (RED, ON); SetLeds (BLUE | YELLOW, OFF)*
  - *ToggleLeds (BLUE | RED)*

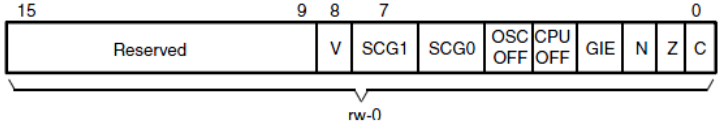
MSP430 clocks

- Clocks, generated from oscillators (high and low frequency)
  - low frequency external ~32KHz, high frequency external ~4Mhz, medium internal ~900kHz (DCO, digitally controlled oscillator)
  - clock source for a module can be set by software
  - ACLK
    - auxiliary clock, 32768 KHz max. typically
  - MCLK
    - master clock, drives instruction execution
  - SMCLK
    - sub-main clock, drives most peripherals
  - (Watchdog timer)

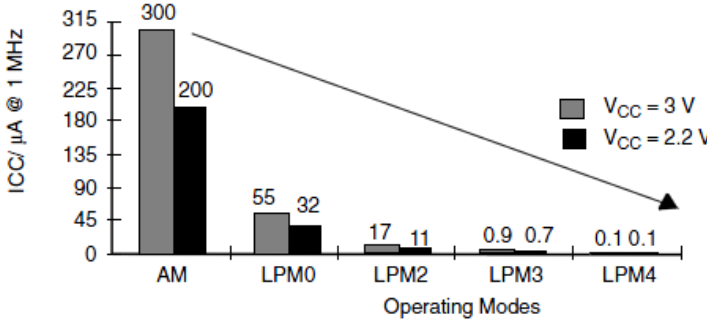
- Low power modes disable some clocks

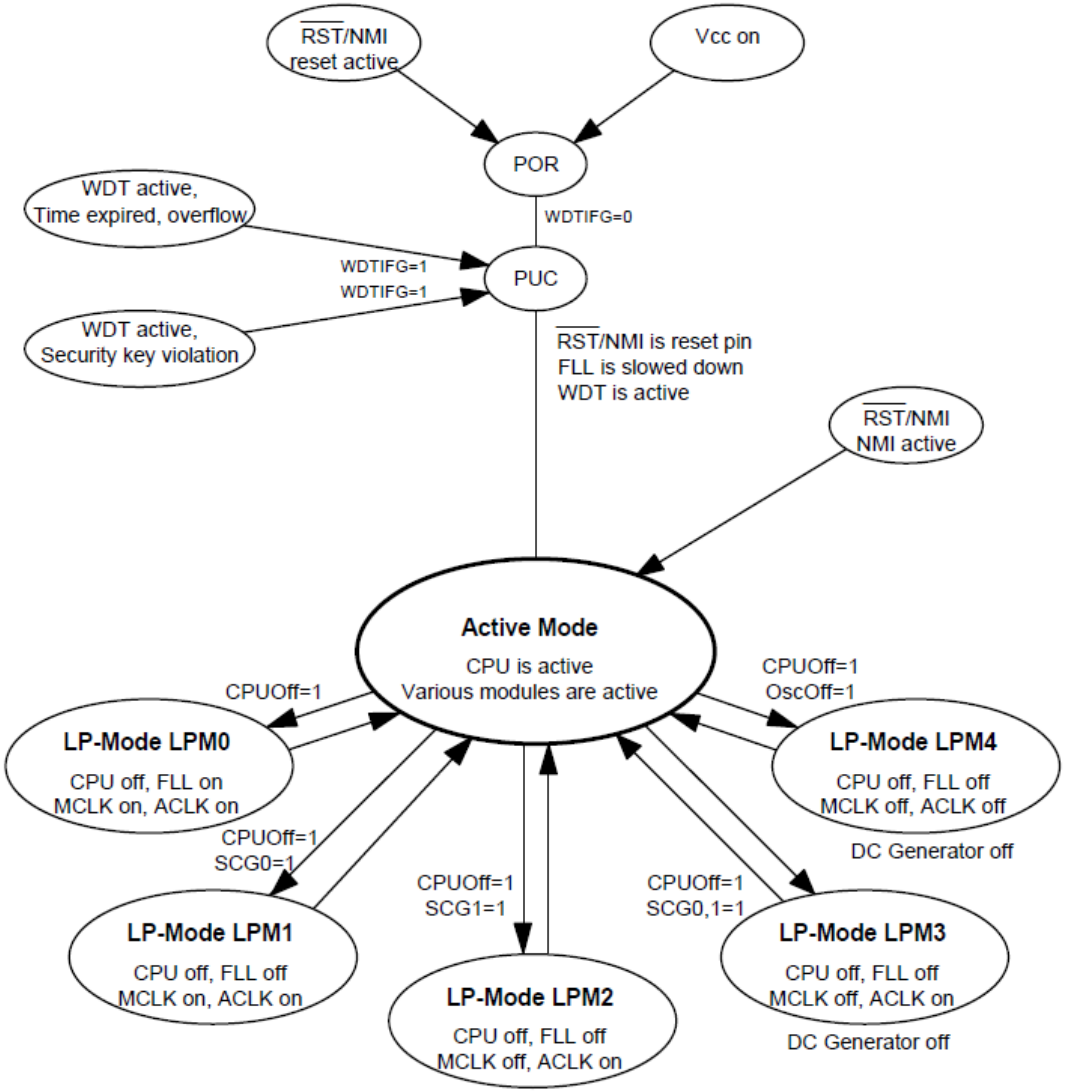
- Application can select a clock to drive a certain module

Power saving modes



status register





Starting the Clock

- Files: Clock.h, Clock.c

void InitClock (void)
void EnterLowPowerMode3()
void ExitLowPowerMode3()

- *InitClock () c*alled at the beginning of the program
  - start ACLK @ 32Khz (standard crystal)
  - generates timer interrupt @ 1024Hz (= *TicksPS*)
- *ExitLowPowerMode3()* called by interrupt routine
- *EnterLowPowerMode3()* called by main program

- Definition of constant *TicksPS*

Agenda

- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

A simple multi-rate periodic scheduler

- Main program *registers* tasks with the scheduler
  - task: a (C) function
  - parameters:
    - *phasing, period* (in time units) – initial time lapse and period
      - note: phasing is with respect to ‘now’
    - *priority, flags* – relative execution order and task type (just TRIGGERED for now)

- Scheduler – and function *IstimeFor()* – is, in fact, a data structure controlled by the timer interrupt routine:
  - timer routine counts down time units of each registered task
  - executes those tasks for which the count down reaches 0

- Files: Scheduler.h, SchedulerNPBasic.c

API: Application structure

1. Setup the timer interrupt
   - E.g. set the timer period
2. Initialize the Task Control Blocks (TCB)
   - Allocate memory for TCBs
3. Register tasks
   - Fill in the TCB for each task
4. Enable interrupts
   - Including timer interrupt
5. Spin forever
   - Tasks will execute inside the timer interrupt handler
   - Idling puts processor in low-power mode

```
int main() {
  /* setup the timer interrupt */

  InitTasks();

  RegisterTask(0, 30, Task1, 1, 0);
  RegisterTask(0, 10, Task2, 2, 0);

  /* enable interrupts */

  while (true)
    { EnterLowPowerMode3(); }
  return 0;
}
```

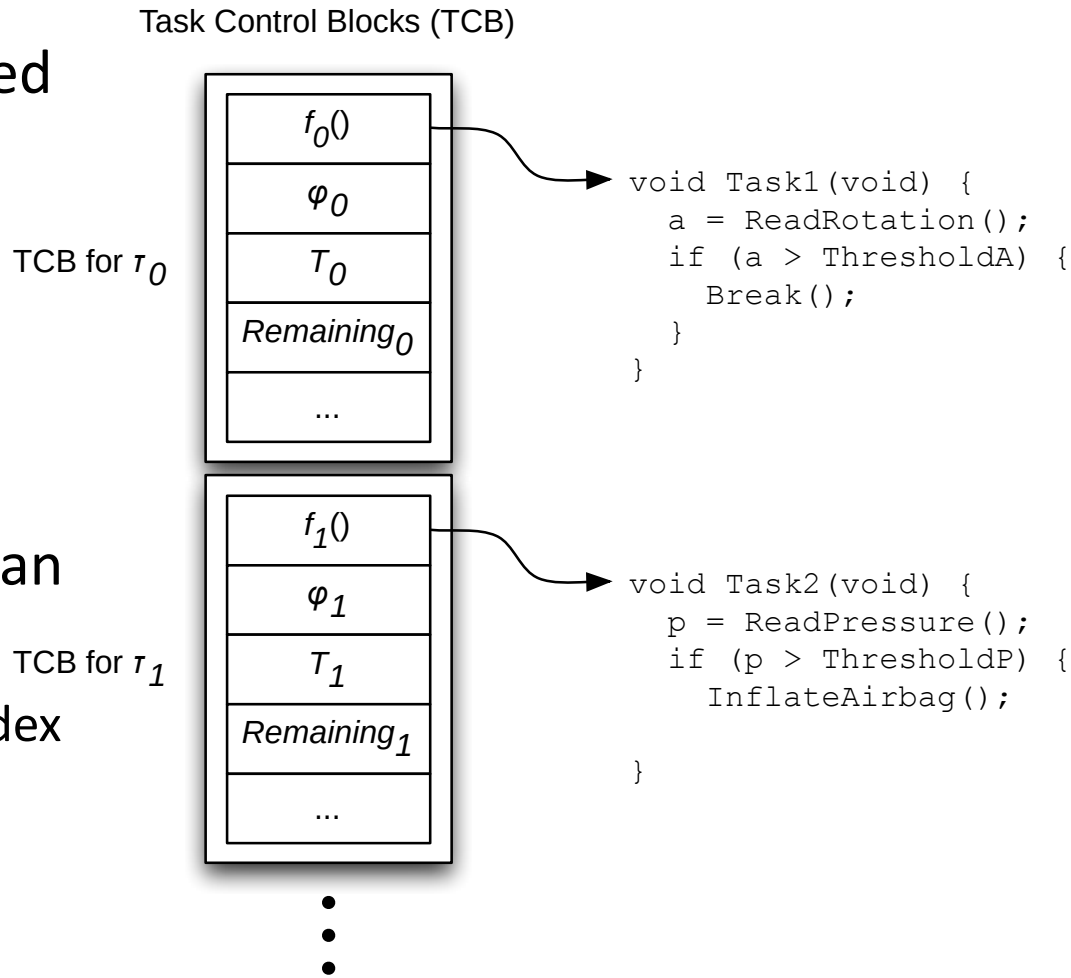
API: Startup

void InitTasks (void)

- Called at the beginning of the program
  - after the clock has started
  - before any tasks are registered
  - before interrupts are enabled

Task Control Block

- Task parameters are stored in the Task Control Block (TCB)
  - Pointer to the C function
  - Phasing, period, etc.
- The scheduler maintains an array of TCBs
  - Task priority is implicit: index of the TCB in the array



API: Task registration with kernel

uint8_t RegisterTask (uint16_t Phasing, uint16_t Period,
*void (*TaskFunc) (void), uint8_t Prio, uint8_t Flags)*

- Can be called at any time after initialization. The task id (*Prio*) must refer to a free task id., i.e., not used before or successfully unregistered before. Registration changes a free task id. into an occupied one.

- *Phasing:* offset in time units before task is activated
  - synchronized at next clock tick (current does not count)
- *Period:* period of activation measured in 1/*TicksPS*
  - must be positive
- *TaskFunc():* function to be used as task body
- *Prio:* priority and task identification
- *Flags:* specify additional properties (next slide)
- *Return status:*
  - *E_SUCCESS*
  - *E_BUSY:* task is already in use
  - *E_WRONGPAR:* error in parameter (Period)

API: Task registration with kernel

uint8_t RegisterTask (uint16_t Phasing, uint16_t Period,
*void (*TaskFunc) (void), uint8_t Prio, uint8_t Flags)*

- Flags:
  - *TRIGGERED*: task is of type triggered (default, may be omitted).

API: Critical sections

uint16_t IntDisable (void)
void RestoreSW (uint16_t sw)

- Functions to create a critical section with. *IntDisable()* returns the current status word to be restored upon completion of the critical section. Typically, the status word is stored in a local variable. In this way, nested critical sections are possible.

- These functions are intended for system use and very brief critical sections. Using them for arbitrary applications is not advised (why?).

Testing

- Use blinking of the leds as tasks
- Use busy-wait counting to give tasks significant length

- File: SchedTest.c
- macro’s NONPRE and NONPREBASIC

Improvements

- Problem with this scheduler implementation:
  - during (non-interruptable) task execution, timer interrupts are missed!

- Improvement:
  - The interrupt routine records, the main program handles the tasks
  - File: SchedulerNP.c
  - macro: NONPRE

- Remaining problem:
  - race condition between timer and main program
  - overrun
  - non preemptable execution

Exercises

- Run the program
- Answer the following questions
  - Which function has a higher priority: BlinkYellow or BlinkGreen?
  - The period of BlinkRed is 0. What happens with the BlinkRed function?
  - After some time, BlinkGreen is activated. Does its execution fit within one timer period?
  - How many interrupts can be pending simultaneously? What happens when more interrupts arrive? Same question about activations.
  - The example application suffers from drift. How can this drift be observed?
  - How to make sure phasings are with respect to a single initial point in time?

Exercises

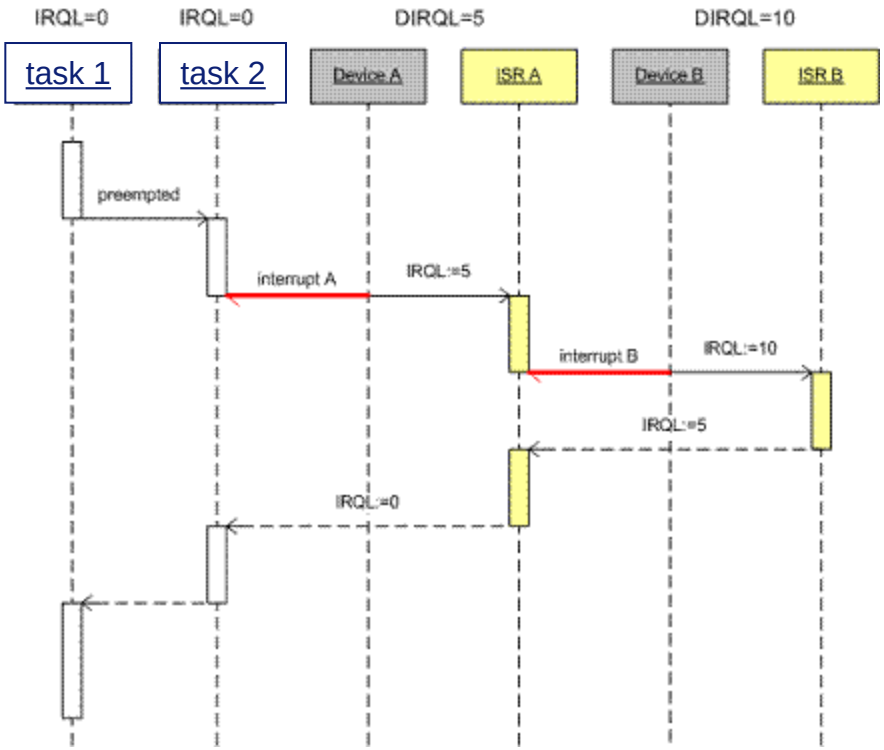
- Give an initialization for the previous exercises (C.1-C.3 on page 14). Run it for tasks that just do busy wait and show what happens using the leds.
- Measure the duration of the timer interrupt routine. Based on that, estimate the maximum number of tasks this system may be able to serve.

Agenda

- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

Preemption

- *Preemption*: stop current task execution and switch control to a new task
- *Reason for preemption*: occurrence of task trigger, e.g. timer or external (device, sensor) event
- Timing analysis required: need information about occurrences of these events
- Preemption results in inconsistencies with respect to shared data structures
  - coordination (synchronization) is needed to deal with this



Adding preemption

- A typical issue to worry about is how the interrupted state is stored.

- Observations:
  - if we use *N* priorities, and if at most one job of a task can be active, at most *N* different states are needed
  - if tasks run until completion, the only reason to store their state is to execute a higher priority task

- Hence,
  - we store the state of a task as a sequence of stack frames
  - a new frame is created upon a timer interrupt (Time Triggered)
  - or as a regular function call in the context of the task that generates the event (Event Triggered, not implemented)
    - a normal task or an interrupt handler
    - we foresee a function *Activate* to implement this

- Files: SchedulerPre.c

API: Task registration with kernel

uint8_t RegisterTask (uint16_t Phasing, uint16_t Period,
*void (*TaskFunc) (void), uint8_t Prio, uint8_t Flags)*

- Flags:
  - *TRIGGERED*: task is of type triggered (default, may be omitted).
  - *TT*: time triggered. Task is activated by the timer, in principle just once.
    - TT is induced when *Phasing>0* or *Period>0*
  - *NOINTERRUPT:* do not pre-empt (useful, but dangerous)

- Further issues:
  - Tasks are not automatically activated, unless they are specified as time triggered. In order to activate an event driven task, *Activate* has to be called.

Exercises

- Retry your previous exercise (see next slide). Experiment with different priorities to demonstrate the effect of preemption. Give a clear scenario.

- Add sporadic event triggered tasks to this system. A sporadic event triggered task is released by an interrupt or another task. To that end write a function *Activate (p)* that activates task *p* that is not declared as TT. Notice that *p* must be executed according to its priority.

- Generalize *Activate()* such that you can give a delay indicating after how long the indicated task is to be started. For an example specification, see slide after next slide.

- In order for a system to be predictable, the requirement is that a sporadic task have a minimal inter-arrival time. Enforce the latter: give this minimal inter-arrival time upon registration and deny or postpone early execution.

- Make it possible to include tasks that are triggered *N* times. Design the corresponding API first.
- For all cases: define suitable examples that show your work.

Exercises

- Give an initialization for the previous exercises (C.1-C.3 on page 14). Run it for tasks that just do busy wait and show what happens using the leds.
- Measure the duration of the timer interrupt routine. Based on that, estimate the maximum number of tasks this system may be able to serve.
- Optional:
  - Maintain a clock using this interrupt routine.
  - Examine the criteria on slide 2. Explain how can you perform measurements.to determine these quality metrics.
    - instrumentation
    - oscilloscope
    - simulation
  - hint: determine the event latency (from timer interrupt to start of task), minimum and maximum
    - priority dependent

API: Task activation

uint8_t Activate (uint8_t Prio, uint16_t Ticks)

- Activate task *Prio, Ticks* units from now. The current timeslot counts as one. A value of 0 means immediate activation. A positive value is only allowed if *Prio* is not already scheduled for time triggering.

- *return:*
  - *E_SUCCESS*
  - *E_NOTASK:* task *Prio* does not exist
  - *E_BUSY:* task *Prio* is already scheduled for time triggering

Agenda

- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

Resource use

- A program uses a number of resources, determined statically and dynamically:
  - time, processor time
  - memory (code size, ram size)
  - energy

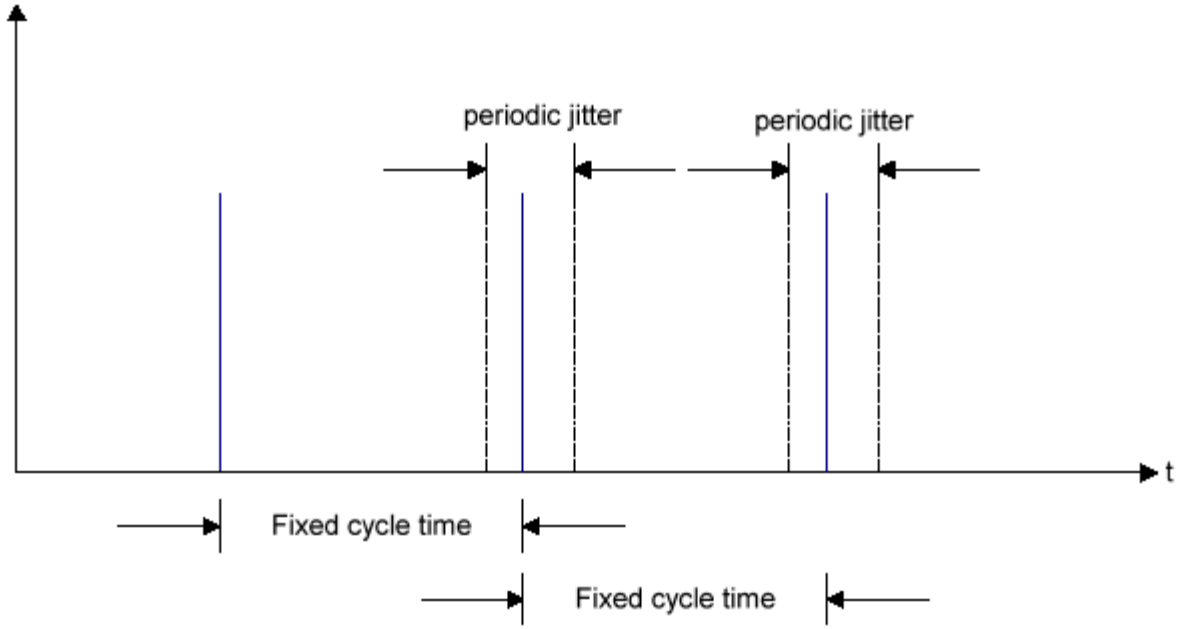
- The resource usage depends heavily on the exact way the program is constructed
  - Architecture dependent:
    - which operations are costly on this particular processor?
    - how much memory do certain operations take?
    - software power management
  - Longest and average code path
  - Use of pointers, indexing
  - Code reuse

- Optimization goals are often conflicting

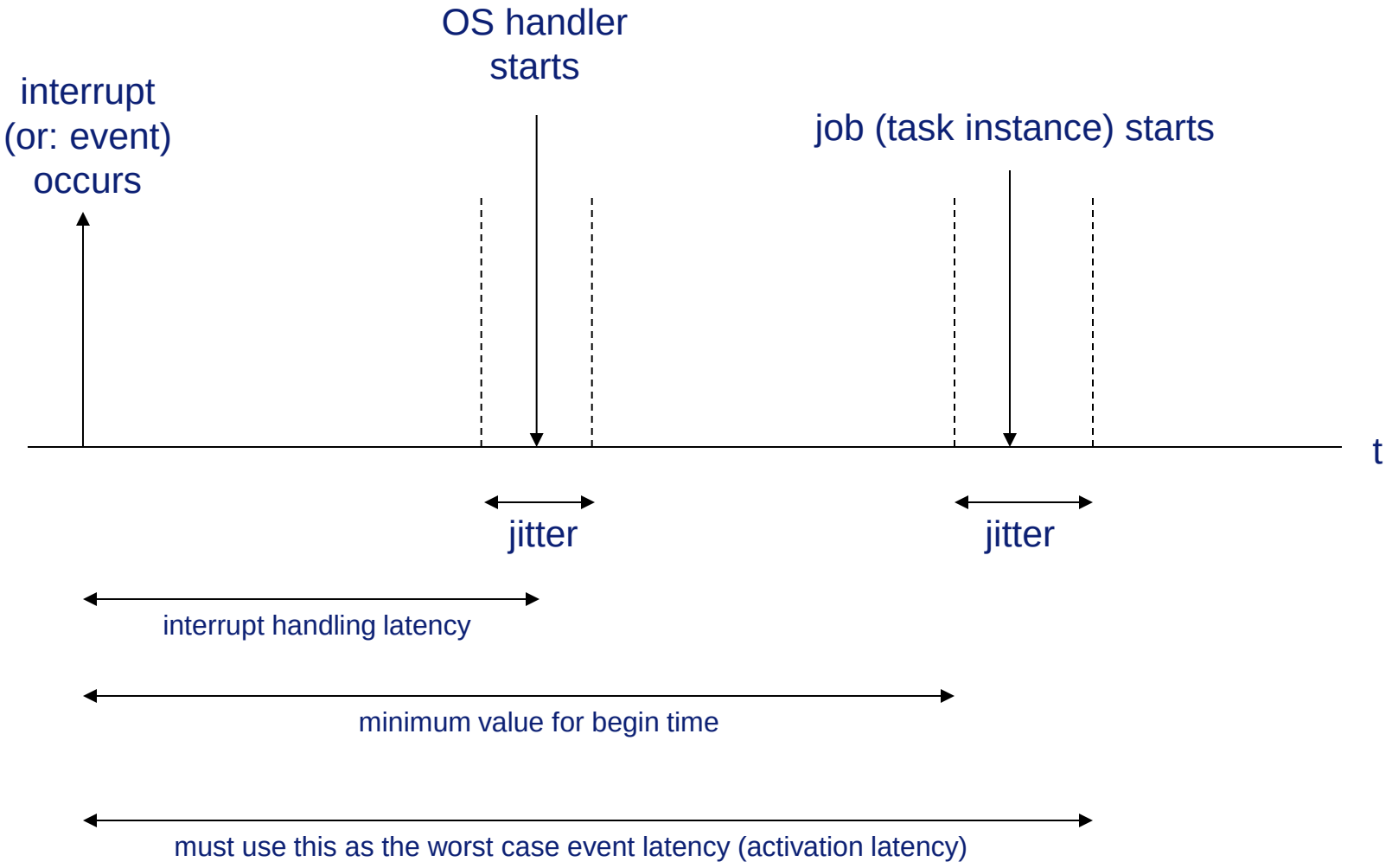
Jitter upon periodic task release

- Jitter has several contributions
  - inaccuracy in generating the timer event
  - variations in response to this event
  - variations in actually starting the user handler

- Notice: need to avoid adding up jitter over time (drift)
  - schedule on *absolute* times, not *relative*



Latencies and jitters



Optimization

- Techniques
  - analyze cost contributions
  - generate code through macros
  - trade indexing and pointers
  - reuse (store) intermediate computations
  - reuse code: uglify
    - avoid (or use, depending on the goal) inlining
    - create functions just for the purpose of sharing code
    - use global variables for parameter passing, parameter-less functions
  - try to have largest and shortest code paths close to average

- Maintain portability
  - of the scheduler (or of an OS)
    - avoid the use of machine-specifics, e.g. .asm
  - of user programs
    - maintain semantics of API

Accounting for overhead

- Time is spent in the interrupt routines and in the task execution
  - Interrupts can be regarded as highest priority tasks
  - Switching to a task and other administration can be regarded as task computation time

- Most interrupts (except for the timer) are event driven
  - hence, in order to guarantee minimal inter-arrival time, the interrupt enable must be cleared and set in due time
  - this can be done through a high priority ‘NOINTERRUPT’ time driven task

Exercises

- Create a simple task with a period of 2 time units; run it without any other tasks
- Measure the duration of
  - the interrupt routine
  - the interrupt routine plus *HandleTasks ()*
    - choose the mentioned task at priority 0
  - compute the worst-case utilization due to this administration
- What is the worst-case activation delay (start of interrupt handler until start of task), disregarding any interference by other tasks?
- Reduce the overhead by
  - a more clever recording of the timer queue
    - one way of improving is to put tasks into a queue with a time relative to the previous task in the queue. Then only the head of the queue needs adaptation.
  - a more clever means to select the next task for execution
- Show the outcome of your work in timing diagrams (created by yourself or by the simulator)

Exercises - optional

- Measure / model the utilization of the interrupt handlers (timer, uart if available)
  - e.g. using the oscilloscope or examining the simulator
  - what is the time resolution on which an application can run (the minimum clock tick)? And with the current tick size, how large can NUMTASKS be?
  - can you reduce the execution time of the interrupt routine? Is improvement of the average case sufficient?
    - one way of improving is to put tasks into a queue with a time relative to the previous task in the queue. Then only the head of the queue needs adaptation.
- Examine one of your previous solutions.
  - reduce it as much as you can in size by employing some of the mentioned techniques (may need to use some compiler flags as well)
- Examine once again your solution
  - estimate (in number of instructions, or in microseconds) the important delays as mentioned in the slides for the two solutions that you have
  - reduce the delays (note that this conflicts with reducing size) and reduce the execution time as much as you can
    - focus on the parts that have a high contribution to the utilization
  - can you also reduce the energy use? how?

Evaluating the programming model

- A programmer decomposes her program (-execution) into a number of tasks *to be called by the scheduler*
  - the ‘Hollywood principle’: don’t call us, we call you
  - also called: *reversal of control*

- Task execution cannot (must not) block!
  - hence, waiting on a condition (an event occurrence or resource availability) cannot be done within a task
    - context would need to be stored across suspension
    - hence, this is work for the programmer now

- Two approaches are possible
  - use two tasks – *split phase approach,* (basis of e.g. TinyOS):
    - first task runs up to the suspension point
    - second task is started as event when the blocking condition becomes false
  - do not start the task until the blocking condition is false
    - the ‘Stack Resource Protocol’ supports this

Agenda

- Cyclic executive
- The MSP430 and the BSN.v3 node
- Cyclic executive implementation
- Preemptive tasks
- Performance
- Evaluation

Evaluating the programming model

- A programmer decomposes her program (-execution) into a number of tasks *to be called by the scheduler*
  - the ‘Hollywood principle’: don’t call us, we call you
  - also called: *reversal of control*

- Task execution cannot (must not) block!
  - hence, waiting on a condition (an event occurrence or resource availability) cannot be done within a task
    - context would need to be stored across suspension
    - hence, this is work for the programmer now

- Three approaches are possible
  - use two tasks – *split phase approach,* (basis of e.g. TinyOS):
    - first task runs up to the suspension point
    - second task is started as event when the blocking condition becomes false
  - do not start the task until the blocking condition is false
    - the ‘Stack Resource Protocol’ supports this
    - only for shared resources
  - use a generic concept: thread with its own context

Useful examples of blocking tasks

- We can create a ‘background’ task of lowest priority that runs continuously
  - more or less a ‘regular’ programming approach
  - however, such task can never suspend!

- Consider a task that sends a sequence of bytes to the screen
  - approach:
    - use a buffer to write the data in
    - this buffer is read by an event-driven task controlled via the interrupt routine, or by a periodic task
    - however, when not enough empty buffer places are available, the sending task has to block

- Consider a task that needs a slow input
  - e.g. sensor reading

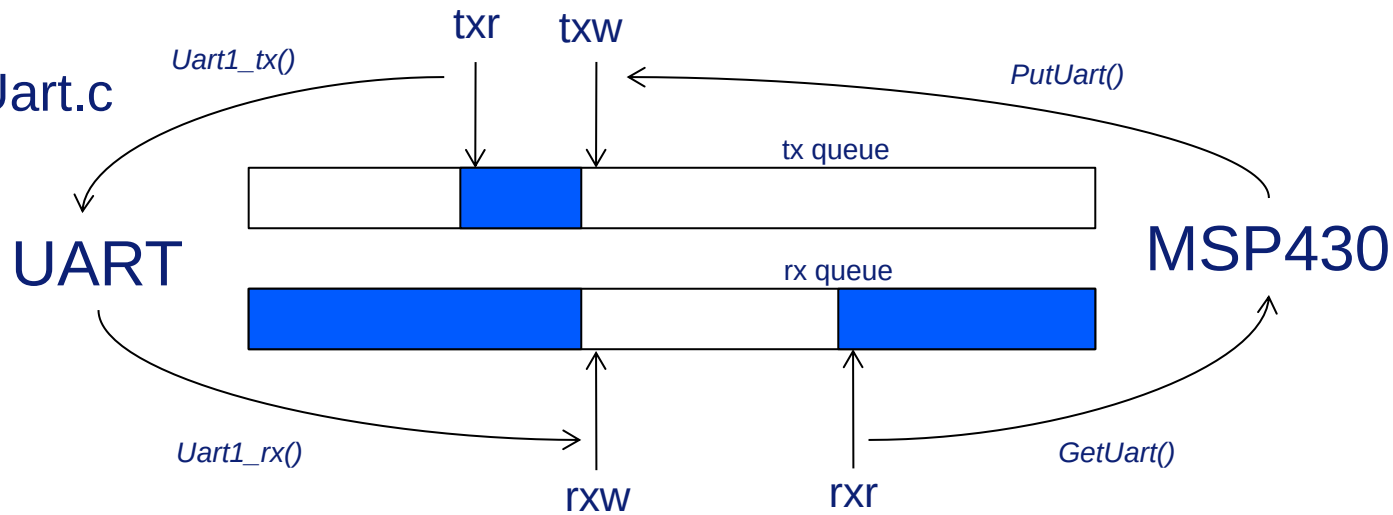
Communication between tasks

- Communication goes through shared variables
  - no blocking synchronization possible
  - hence, queues (“FIFOs”) are a most useful means
  - access procedures (to queues) and operating system calls must not block but return error status

- Exclusion must be provided since preemption is possible
  - using interrupt enable/disable

- Example: communication using the UART

- Files: Uart.h, Uart.c



Exercise – optional

- Create a task that sends a string to the output
  - what to do if this string is longer than the buffer size?
  - design and discuss solutions to this problem; try to avoid unnecessary waiting times

- Make sure the uart interrupt routines become sporadic tasks.

- Recall that a Task is formally the *complete* response to an event. This differs from the concept of a task in our mini kernel. Suppose that we have two time triggered Tasks for which this response consists of some computational work followed by transmitting a byte over the uart.
  - How would you encode such a Task?
  - If you use the queue as above, how is the priority of the Task maintained?

Multiple threads

- Task suspension...
  - ....due to preemption can be addressed in a single stack execution
  - ....due to self-suspension (a form of blocking) needs separate context

- Giving a task a separate context yields the notion of a *thread*
  - unit of execution, of concurrency
  - context represented by execution state: stack (local variables) and program counter

- Two options:
  - create a new thread context upon each task invocation
  - create a thread context once, and let tasks be iterations of the same thread
    - synchronize the iterations using the clock interface (timeout)
    - synchronize with the kernel using a *semaphore*

Thread programming model

- Typically, as part of a Real Time OS
- More expensive than our simple task model
  - need to save and restore context
  - more complicated synchronization primitives
- Typical task:

```
Time driven:
  while true do
    task();   /* may contain blocking
                 statements */
    Suspend until next period
  od
```

```
Event driven:
  while true do
    SemWait (Start);
    task();   /* may contain blocking
                 statements */
  od
```

- No reversal of control, no split phase execution

What further API calls do we need?

- We regard a thread as just another task
  - simply with a different type

- Must be able to
  - reserve some stack space, e.g. by using dynamic memory (*malloc(), free()*)
  - Start a thread: *InitThread (Taskp t, unint16_t SP)*
    - SP represents a stack space
  - Suspend the current thread: *Suspend()*
  - Switch execution context: *Switch(Taskp t)* exchanges current context with context stored in *t*
  - Finalize a thread: implicit, upon leaving the thread function
    - must invoke cleanup code in that case

- Other operations on tasks remain
  - activate a thread: *Activate()*
  - *RegisterTask (Phasing, Period, ThreadFunction, Prio, THREAD|TT)*
    - indicate the type through the flags; the time-driven scheduler will activate it
    - even periodically, if requested

Building with the primitives

```
uint8_t Delay (uint16_t Ticks)
{
    Activate (CurrentTask(), Ticks); Suspend ();
}

void BlinkerA (void)
{
    while (1) {
        ToggleLeds (GREEN);
        Delay (TicksPS);
        ToggleLeds (YELLOW);
        Delay (TicksPS);
    }
}

/* notice: drift */
```

```
void BlinkerB (void)
{
    Delay (TicksPS*3);
    while (1) {
        SetLeds (RED, ON);
        Delay (TicksPS);
        Activate (BLINKERC, 0);
        Suspend ();
}   }

void BlinkerC (void)
{
    while (1) {
        Suspend ();
        SetLeds (RED, OFF);
        Delay (TicksPS);
        Activate (BLINKERB, 0);
}   }
```

Activate()

```
uint8_t Activate (uint8_t Prio, uint16_t Ticks)
{
    if (Prio >= NUMTASKS) return (E_NOTASK);

    uint8_t rtc   = E_SUCCESS;
    Taskp t       = &Tasks[Prio];
    uint16_t sw = IntDisable ();
    if (t->Flags == 0) rtc = E_NOTASK;
    else
      if (Ticks == 0) {
            t->Activated++;
            Pending |= Prio>BusyPrio;    // not: HandleTasks()
      }
      else  if (t->Flags & TT) rtc = E_BUSY;
              else { t->Remaining = Ticks-1; t->Flags |= TT; }
    RestoreSW (sw);
    return (rtc);
}
```

Semaphores (Dijkstra)

- Semaphore *s* is an integer with initial value $s_0 \geq 0$ and *atomic* operations *P(s)* and *V(s).* The effect of these operations is defined as follows:

  P(s): < wait until s>0; s := s-1 >

  V(s): < s := s+1 >

- A thread that executes *P(s)* is suspended until *s>0.*
- A thread executing *V(s)* possibly releases a suspended thread

- Used names:
  - *P(s): SemWait (s)*
  - *V(s): SemPost (s)*

Semaphore implementation

```
typedef struct {
  int16_t v;          /* Semaphore value; negative values represent  # waiters */
  uint8_t Waiters [SEMMAXW];
                      /* Queue of waiters; maximal length is SEMMAXW */
} Sem;

typedef Sem *Semp;          /* Semaphore pointer                        */
```

- Semaphore variable *v* is also used for counting the number of waiters (in that case, semaphore value is 0).
- We add a parameter '*NoBlock*' to *SemWait()* to indicate that the caller does not want to wait
  - or must not wait, if it is not a thread

SemWait()

```
uint8_t SemWait (Semp s, uint8_t NoBlock)
{
    uint8_t  rtc      = E_SUCCESS;
    uint16_t sw       = IntDisable();

    NoBlock  |=   (CurrentTask()->Flags & TRIGGERED);
    if (-s->v == SEMMAXW || (NoBlock && s->v<=0))   rtc = E_SEMFAIL;
    else {
        s->v--;
        if (s->v<0) {
            s->Waiters[-s->v-1] = CurrentPrio();
            Suspend ();
        }
    }
    RestoreSW (sw);
    return (rtc);
}
```

SemPost()

```
void SemPost  (Semp s)
{
    uint16_t sw = IntDisable();
    if (s->v < 0) {
        uint8_t mi = 0;
        uint8_t i;
        for (i = 1; i<-s->v; i++)
            if (s->Waiters[i]>s->Waiters[mi]) mi = i;
        Activate (s->Waiters[mi], 0);
        s->Waiters[mi] = s->Waiters[-s->v-1];
    }
    s->v++;
    RestoreSW (sw);
}
```

Exclusion

- Semaphores are often used for mutual exclusion
  - *.....SemWait(m); Critical Section; SemPost(m).....*
  - initial value of *m:* 1

- Question:
  - what is better: using semaphores for exclusion, or using enable/disable interrupt?
    - think about overhead and blocking times

Exercises

- Do the same steps as on slide 48, but now for a thread that is triggered by the timer. Show the effect of the extra overhead because of thread switching.

- Run this example side by side with a time triggered task.

- Create an example of two threads that share a global variable *x*. Model the access to *x* by a *CountDelay*. For correct execution, access to *x* must be under mutual exclusion, using a semaphore *m.*
  - Show a situation in which the highest priority thread has to wait on the lower priority thread because of this.
  - Show how a third thread of middle priority can extend this waiting.

Exercises

- Add threads to the current system. To that end you must study somewhat more of the MSP430 instruction set architecture. Wikipedia gives a good and concise summary.

  A thread will be a function, typically one with a repetition in it. Design the API carefully in advance. You will need:
  - a context for a thread, viz., a piece of memory allocated to a thread that serves as its stack
    - hence, some memory management is required
  - functions
    - initialize a thread context
    - switch control to a thread
      - this includes saving and restoring registers
    - finalize a thread (upon ending its function)
    - a suspend function, that suspends thread execution; otherwise the thread would always run because of our fixed priorities

- Which activation patterns can be supported by our system (with or without threads)?

Exercise

- Consider the following Task that is time triggered every 50 time units
  - examine an ADC input – this yields a value *v* at most 10
  - light the red led
  - wait for *v* clock cycles
  - light the yellow led
  - wait for *v* clock cycles
  - put off the two leds
  - output a character on the serial interface

  Reading from the ADC you may model as drawing a random number.

1. Encode this Task using the scheme of your choice and run it.
2. Do this (also) without busy waiting.
3. Determine its timing behavior: worst case execution time and worst case response time.
4. Is your solution predictable?
5. What type of job constraints do you see in the example?