Real-Time Systems

Exercise #1

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Laboratory assignment

Organization:

The laboratory sessions (one session/week in study weeks 4–8)

- To get guidance regarding the assignment problems, and also get help with debugging of software issues
- To demonstrate your solutions to each of the assignment problems, and have them approved

The project groups (3 students per group)

- The groups' weekly laboratory session takes place on:
  - RTS A xx: Mon @ 13:15-17:00
  - RTS B xx: Wed @ 08:00-11:45
  - RTS C xx: Wed @ 15:15-19:00*

  (* study weeks 5 & 7: Wed 17:15-21:00)

Location:

Jupiter building,
campus Lindholmen

Laboratory assignment

Organization (cont'd):

The loan equipment

- You will be offered loan equipment by the department, and work on the assignment at home on your own computer.
- The loan equipment for each project group consists of:
  - one MD407 processor card
  - one USB cable + one CAN cable
- You pick up the loan equipment during study week 3, and return the loan equipment during exam week.
- The equipment is picked up in the lab room at Lindholmen campus during your scheduled lab time slot. To return the equipment you book an appointment with the examiner.

Laboratory assignment

Basic prerequisites for approval:

Respecting the Rules of Conduct (see separate document)

- Contribute to your group: (be present, be active)
- Respect the laboratory sessions: (read guidelines / lab-PM)
- Respect the deadlines: (submit assignments in time)
- Refrain from cheating:
  Do not copy other people's code!
  Do not share your code with other groups!

Laboratory performance of satisfactory quality (≥ grade 3)

Laboratory assignment

Grading of the ‘Laboratory’ course element:

The grade (U, 3, 4, 5) will reflect your practical skills at the laboratory sessions.

The grade is determined by the following:

- The quality of laboratory performance
  - sub-score is awarded* based on a set of four criteria
  - sub-score sets the final grade

(* See corresponding modules in Canvas for details)

Laboratory assignment

Laboratory performance sub-score criteria:

Implementation

- How many of the coding challenges in Part 2 that you can successfully implement and demonstrate.

Design

- How well you know the design and behavior of your code.

Debugging

- How well you identify, and solve, problems with your code.

Paradigm

- How well you understand, and make use of, the Concurrent, Object-oriented, Reactive, Timing-aware programming method.

Laboratory assignment – Part 0

Getting started: [compulsory]

- Compile the template code using the cross compiler
- Upload the machine code to the target computer

Interacting with the target computer: [compulsory]

- Take input from the workstation's keyboard
- Generate output to the workstation's console window

Preparatory work for Part 1 and Part 2: [compulsory]

- Pre-compute periods for all tones that will be played
- Prepare data structures to allow a melody to be transposed to different keys

Laboratory assignment – Part 1

Tone generator: [compulsory]

- Generate a 1 kHz tone (square wave signal) and output it to the audio jack on the target computer

Background load: [compulsory]

- Add a background task with a scalable load
- Experiment: disturb tone generator by increasing the load
- Repeat the experiment with deadline scheduling enabled

Worst-case execution times: [compulsory]

- Measure the execution times of the program code in the background load task and the tone generator task

Laboratory assignment – Part 2

Brother John music player: [compulsory]

- Capable to play tones in a 12-tone scale in different keys
- Capable to play the melody “Brother John”, and be able to change key and/or tempo while playing

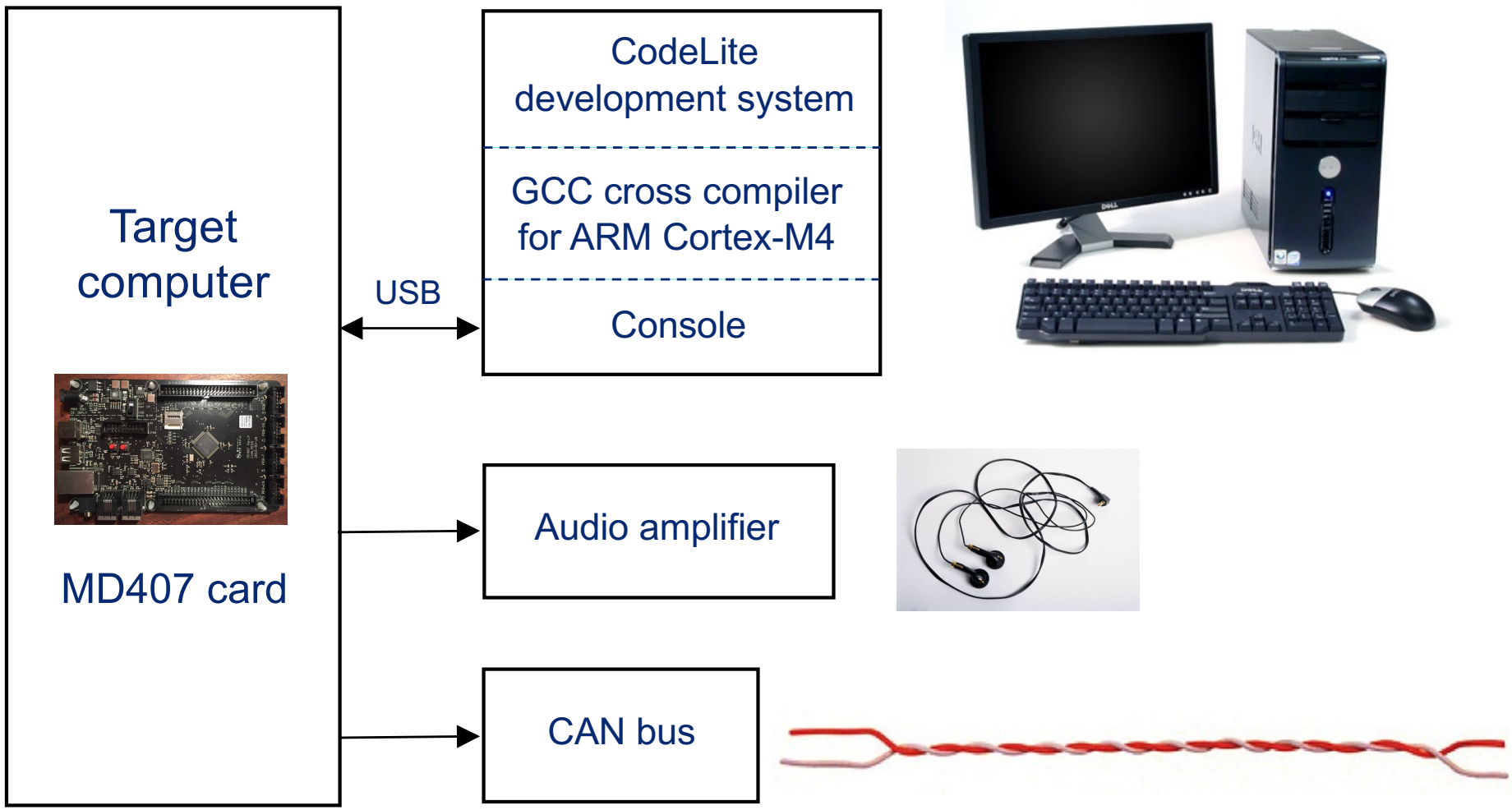
Loopback CAN: [required for grade 4 or higher]

- Capable to control the music player via the CAN bus
- Capable to run in two different modes: conductor / musician

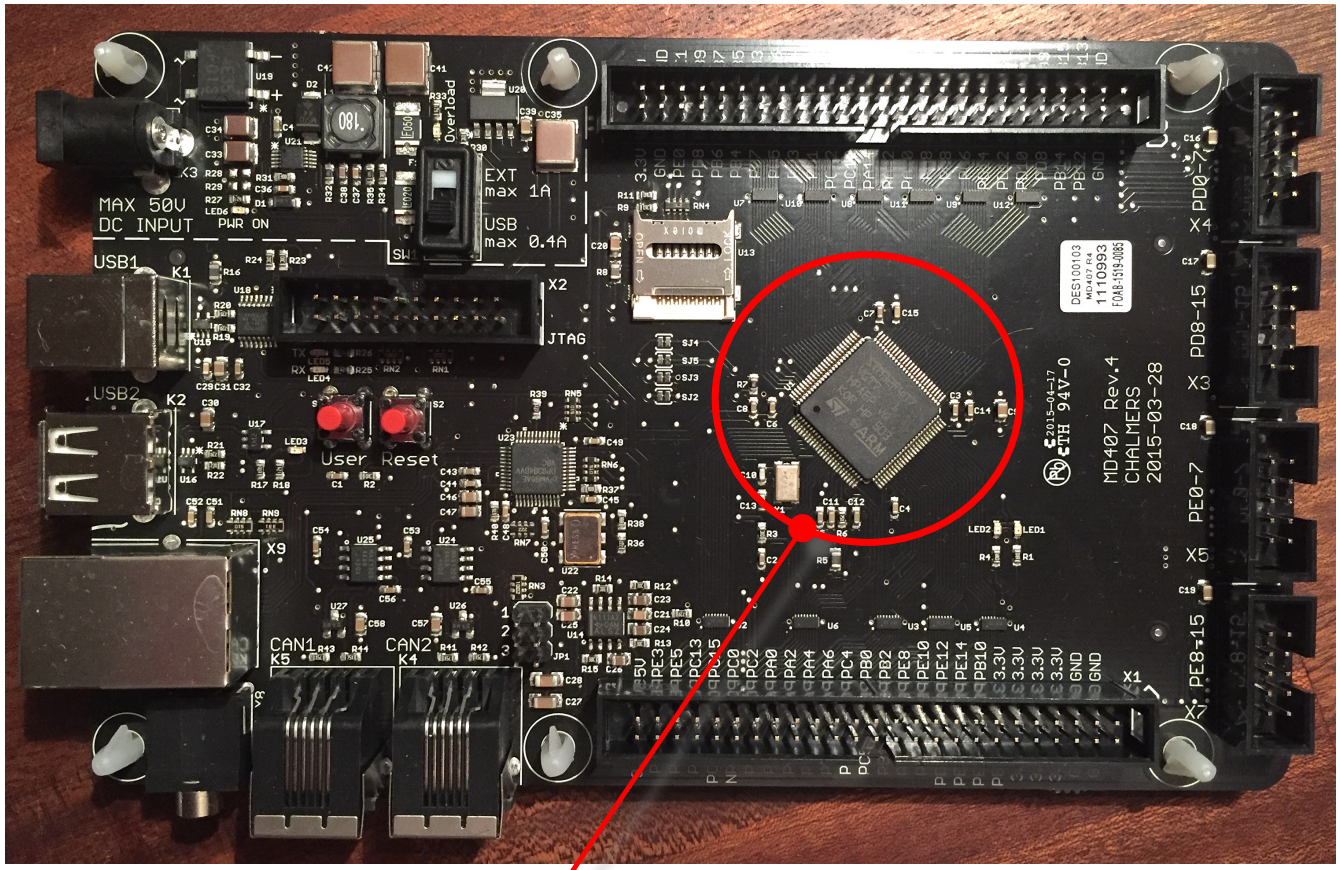
Tap/Blink tempo with button/LED : [required for grade 5]

- Extend the music player with extra interactive functionality
- Utilize red “user” button and green LED on processor card

Laboratory assignment – Setup

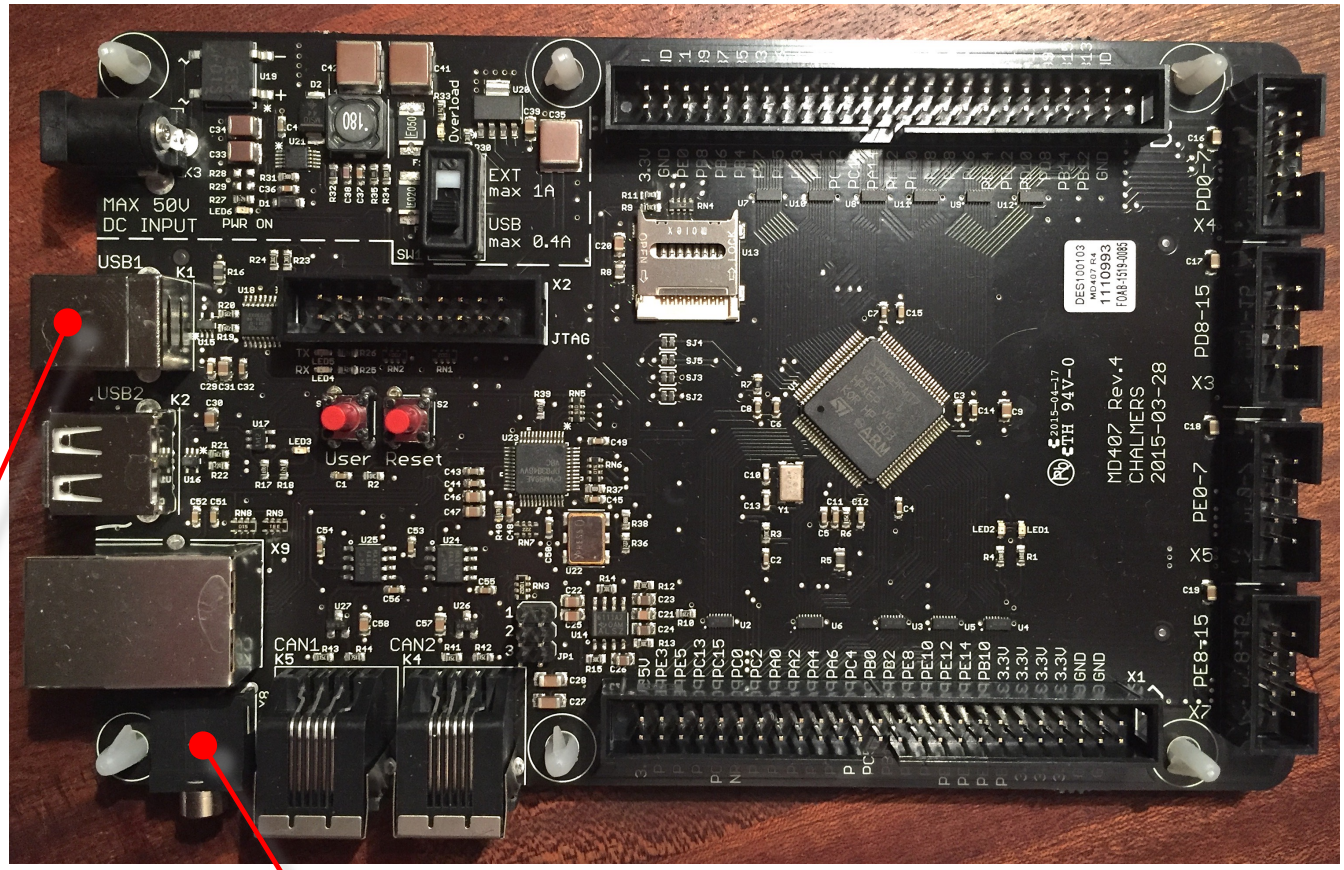


The MD407 computer card



STMicroelectronics' STM32F407 microcontroller /w ARM Cortex-M4 core

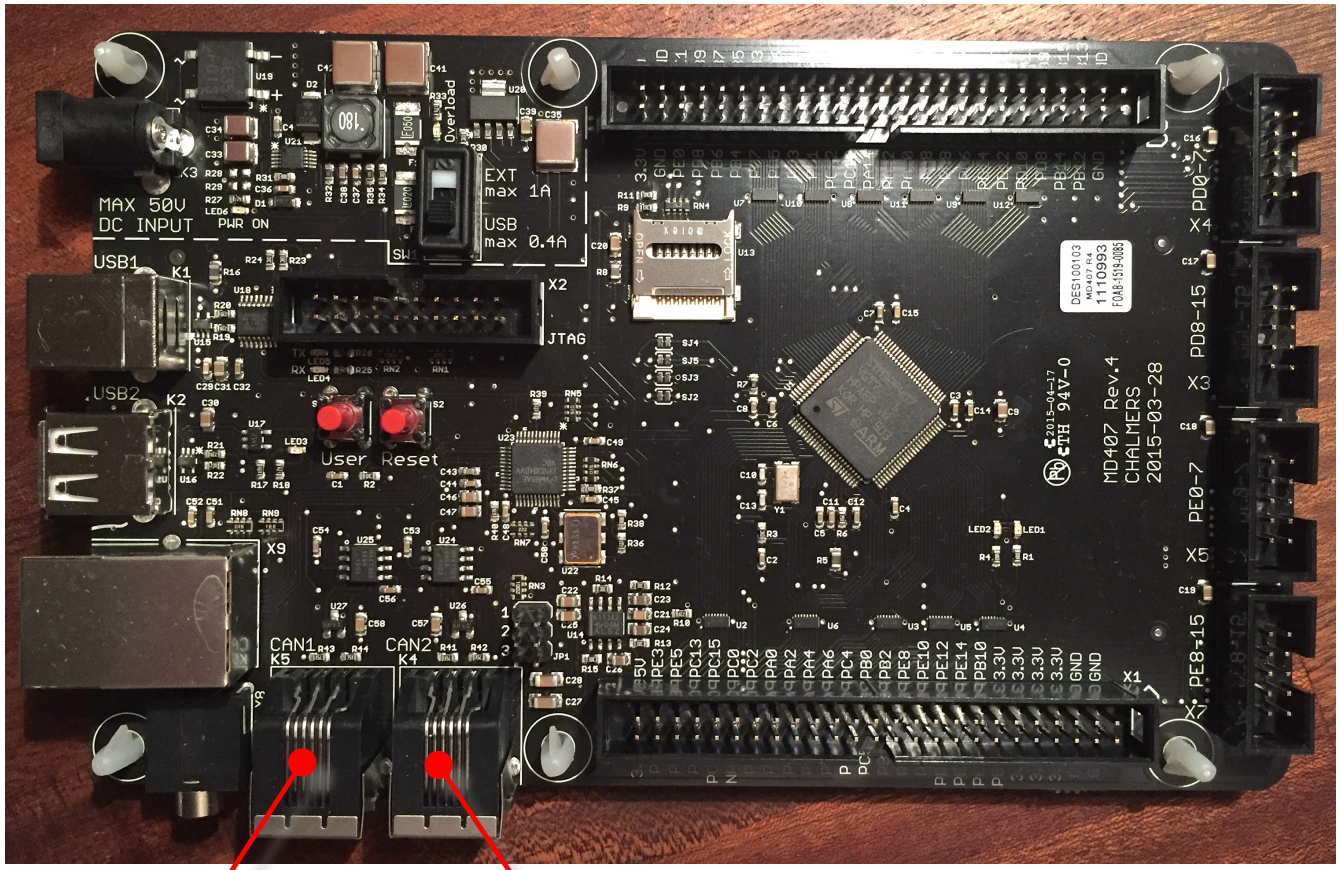
The MD407 computer card



USB debug port

Audio output jack

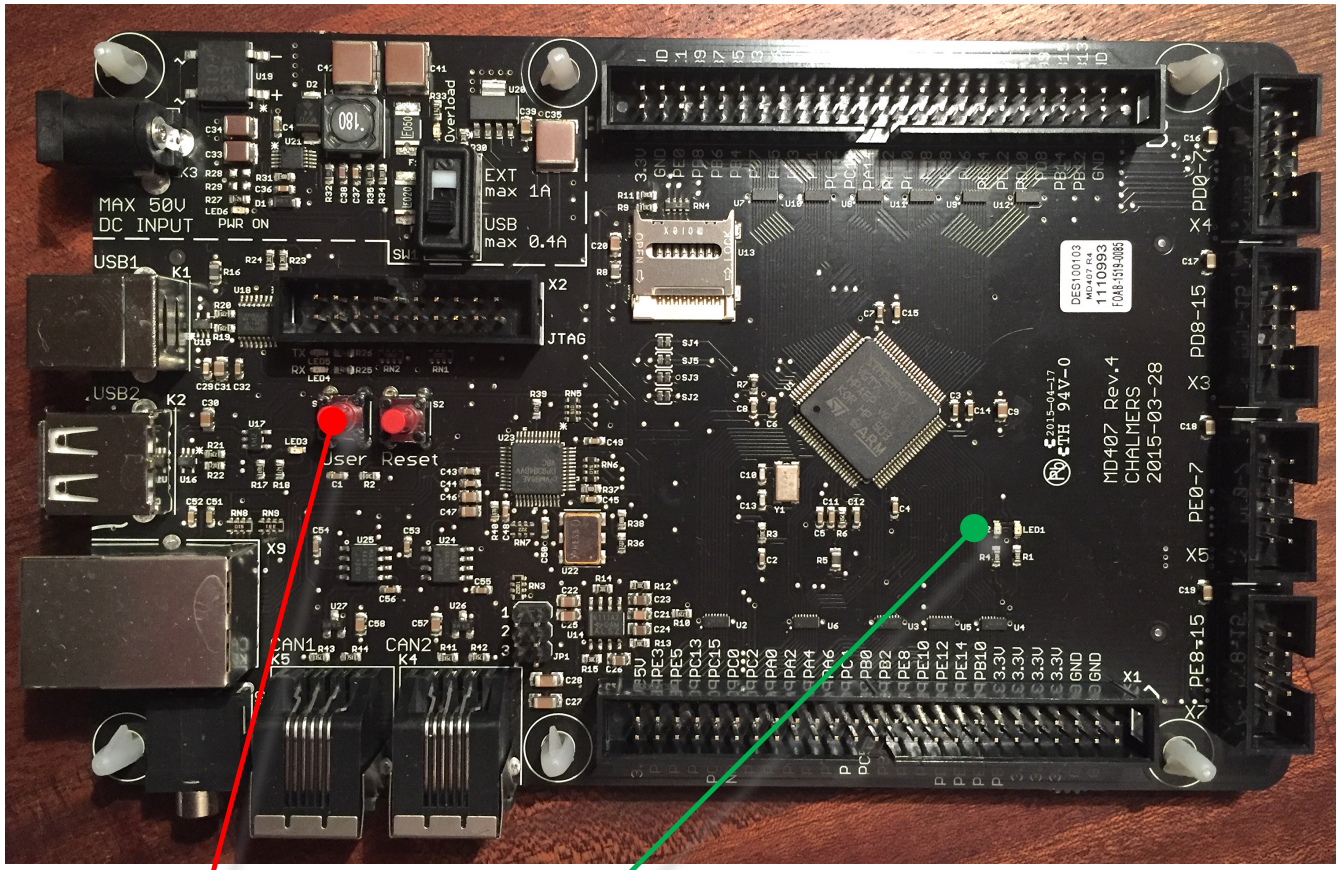
The MD407 computer card



Main CAN bus

Loopback CAN bus

The MD407 computer card



Red "User" button

Green LED

The MD407 computer card

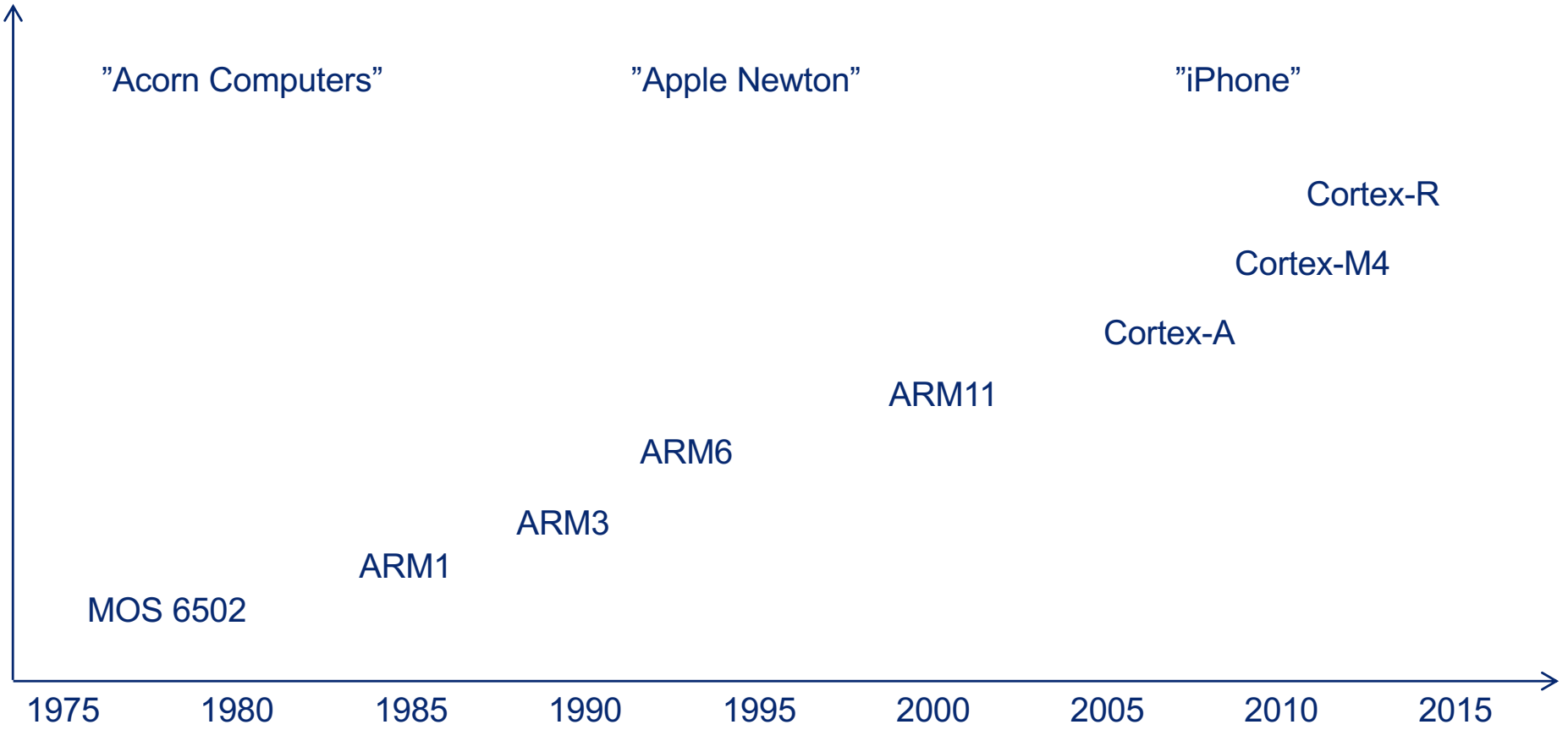
The STM32F407 microcontroller:

Based on the ARM Cortex-M4 processor core

- 168 MHz processor clock
- 32-bit registers (data and address)
- 16-bit instructions (Thumb)
- 1 MB of Flash memory (for resident monitor/debugger)
- 128 kB of RAM (for user programs)
- On-chip floating-point unit
- On-chip CAN modules, serial communications interfaces, parallel ports, digital-to-analog converters, high-resolution timers, ...

The MD407 computer card

The ARM processor family tree:



STM32F407 block diagram:

CPU – Cortex-M4 core

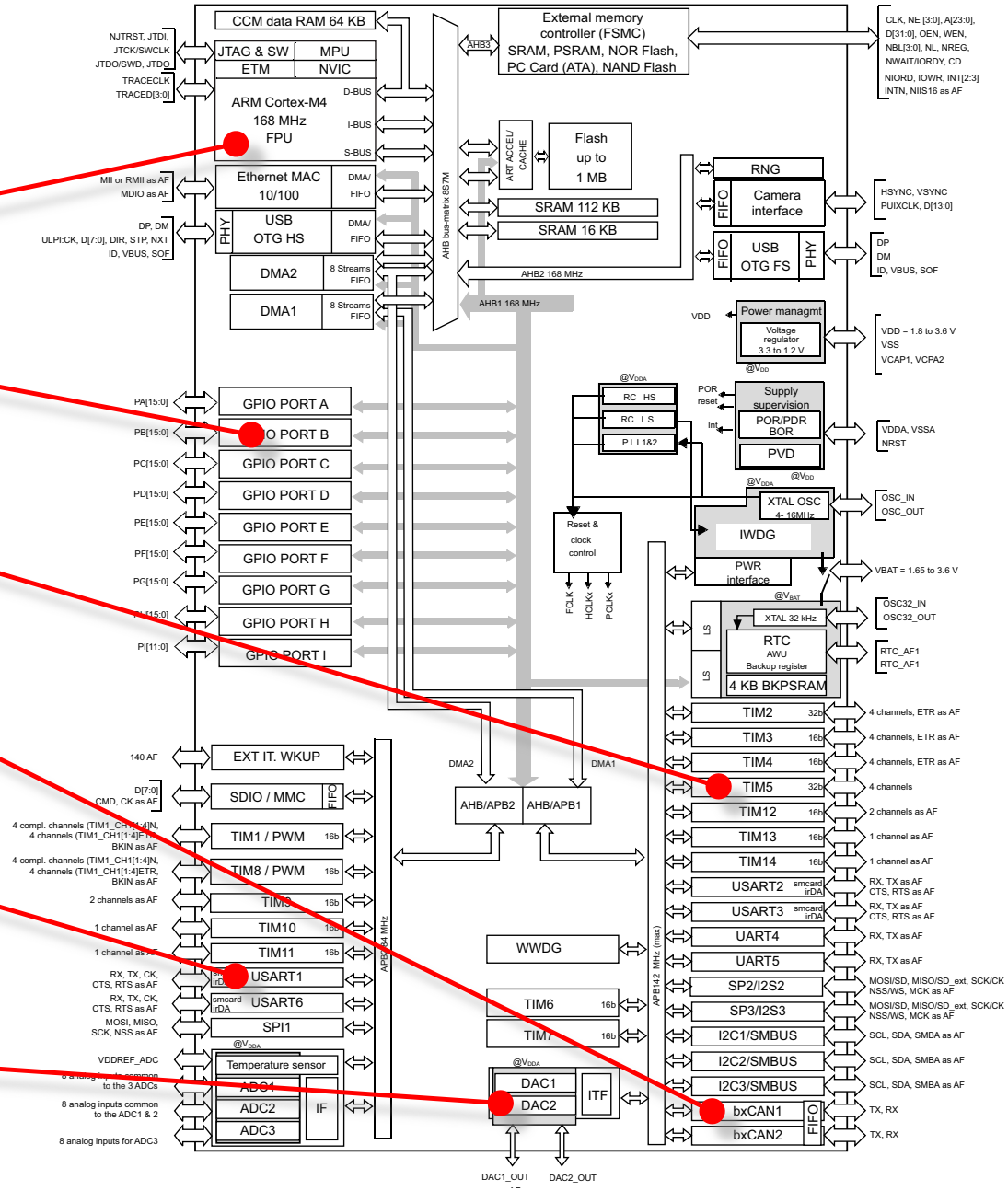
Port B – connects to red “User” button and green LED

TIM5 – used by TinyTimber for high-resolution clock

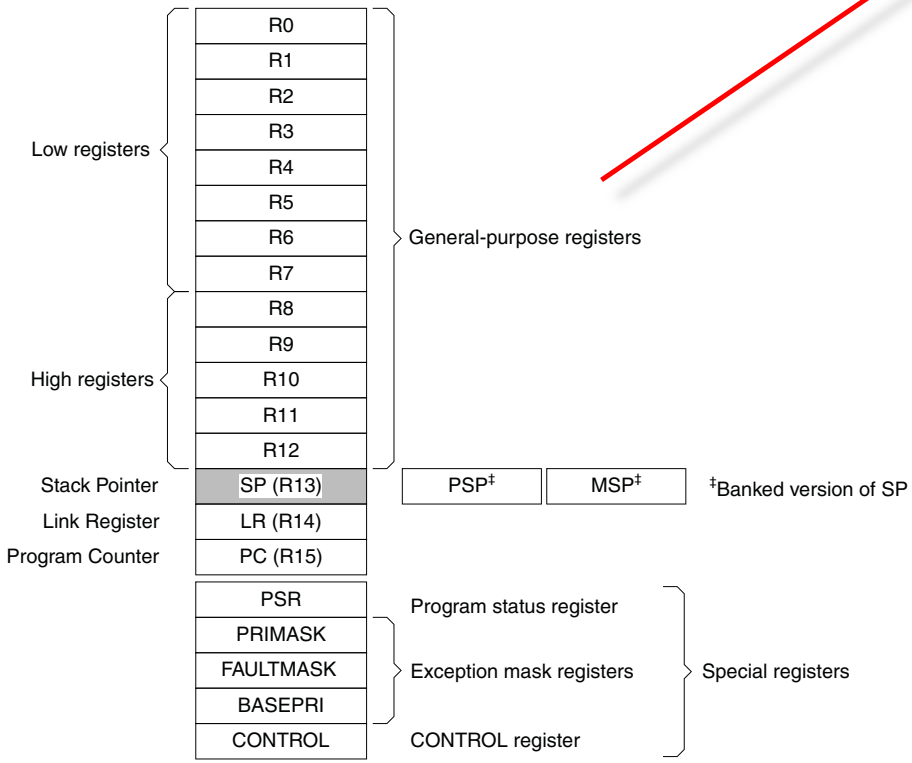
bxCAN1 – available for your connection to other cards

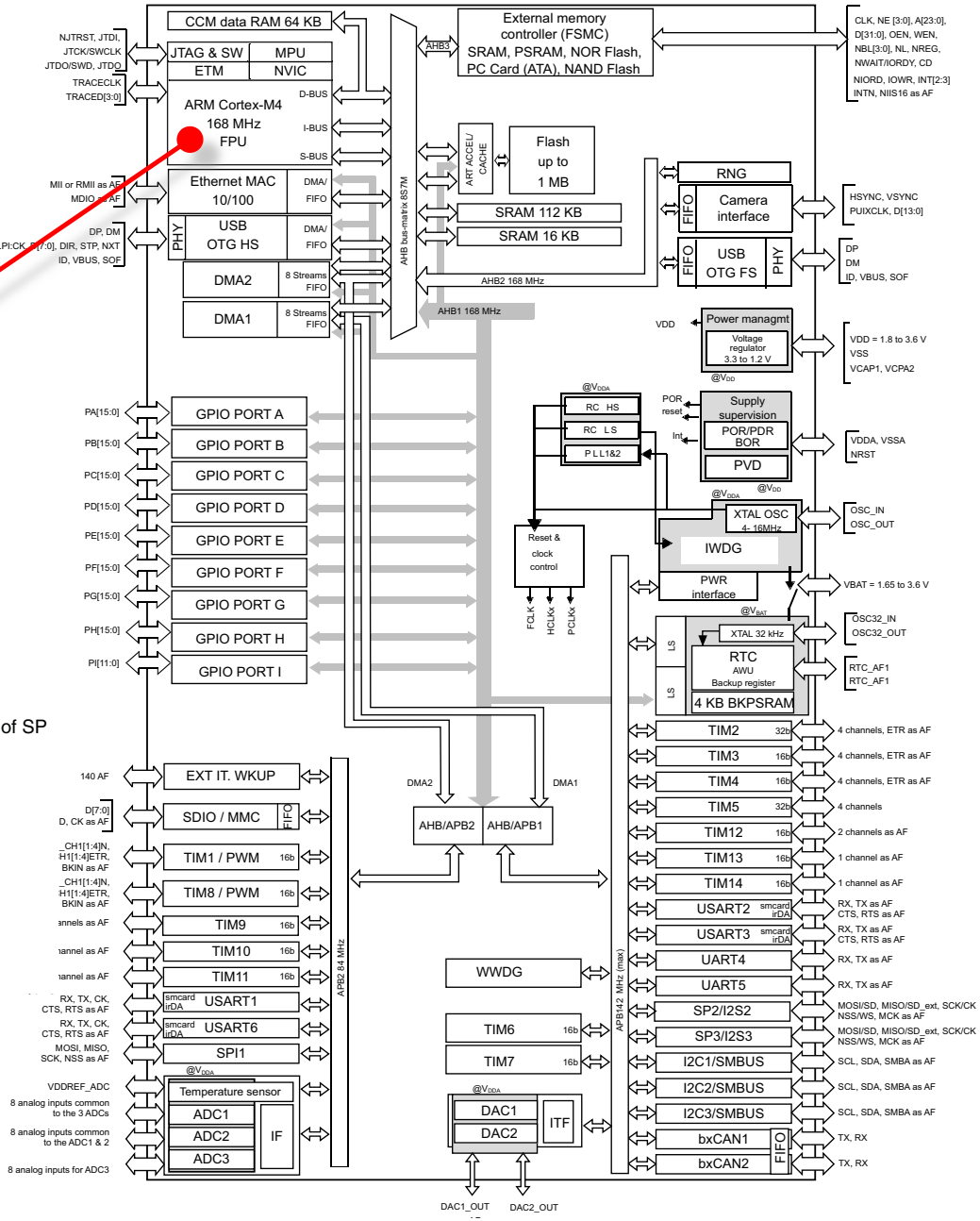
USART1– connects to console via USB debug port

DAC2 – available for your sound generation

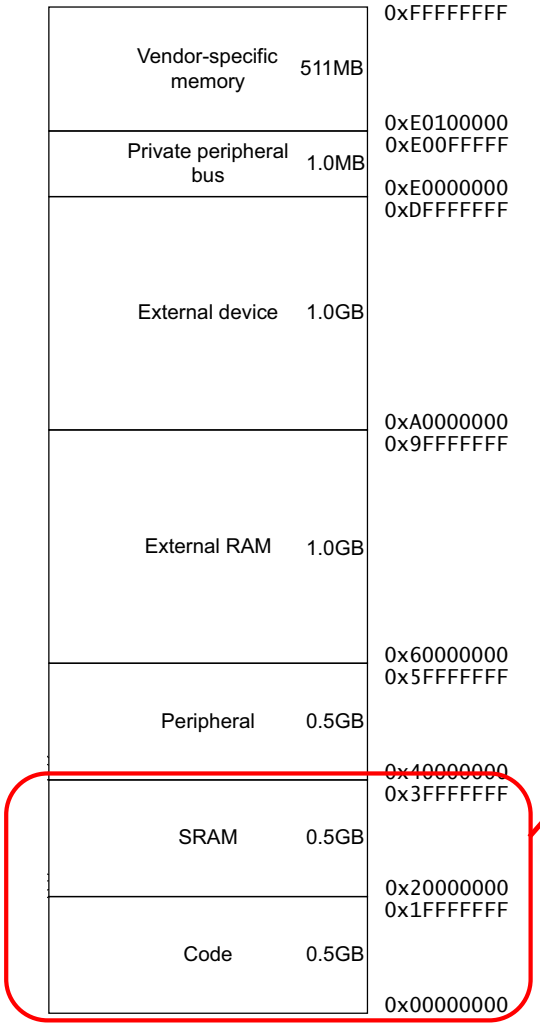


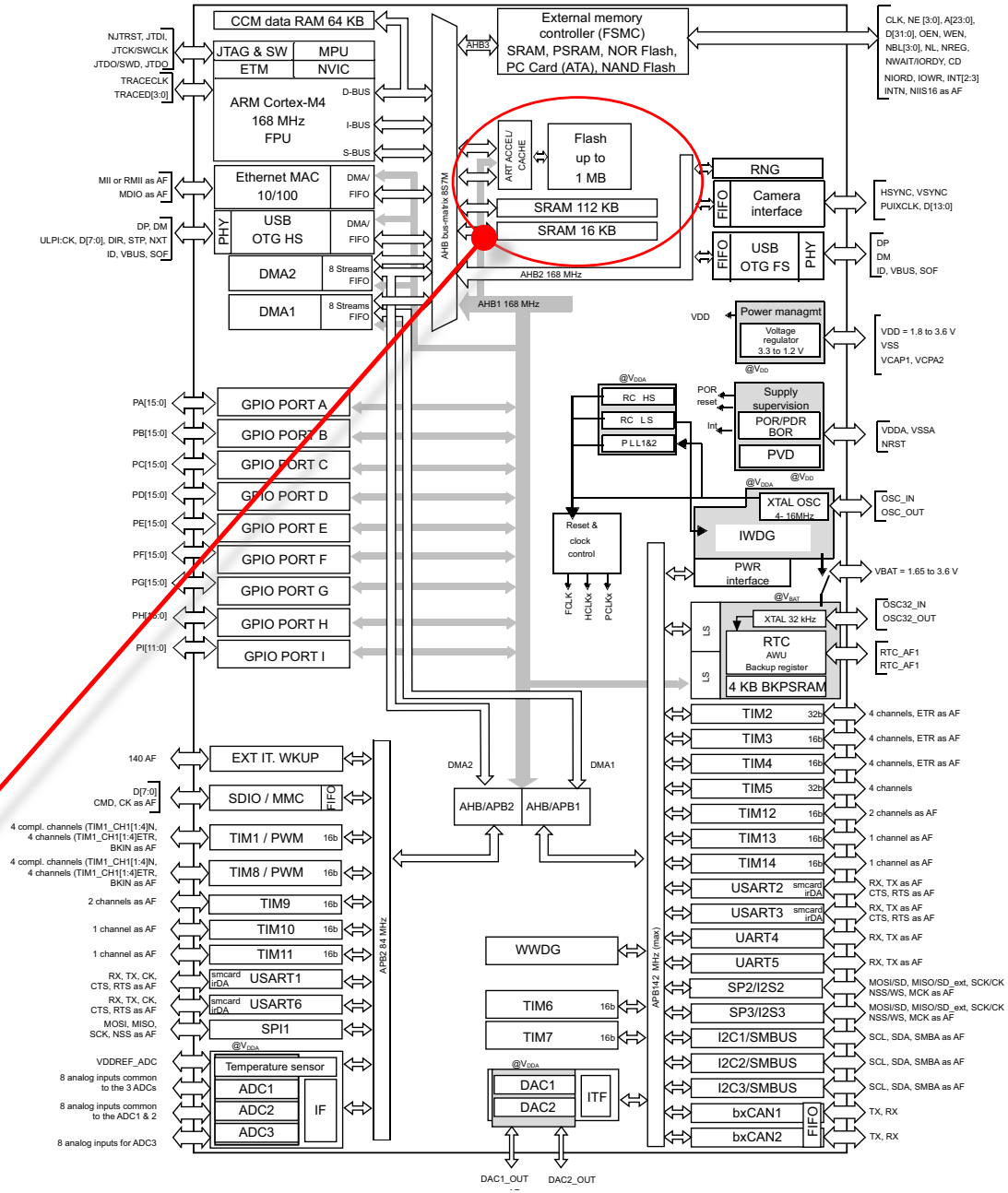
STM32F407 processor registers:



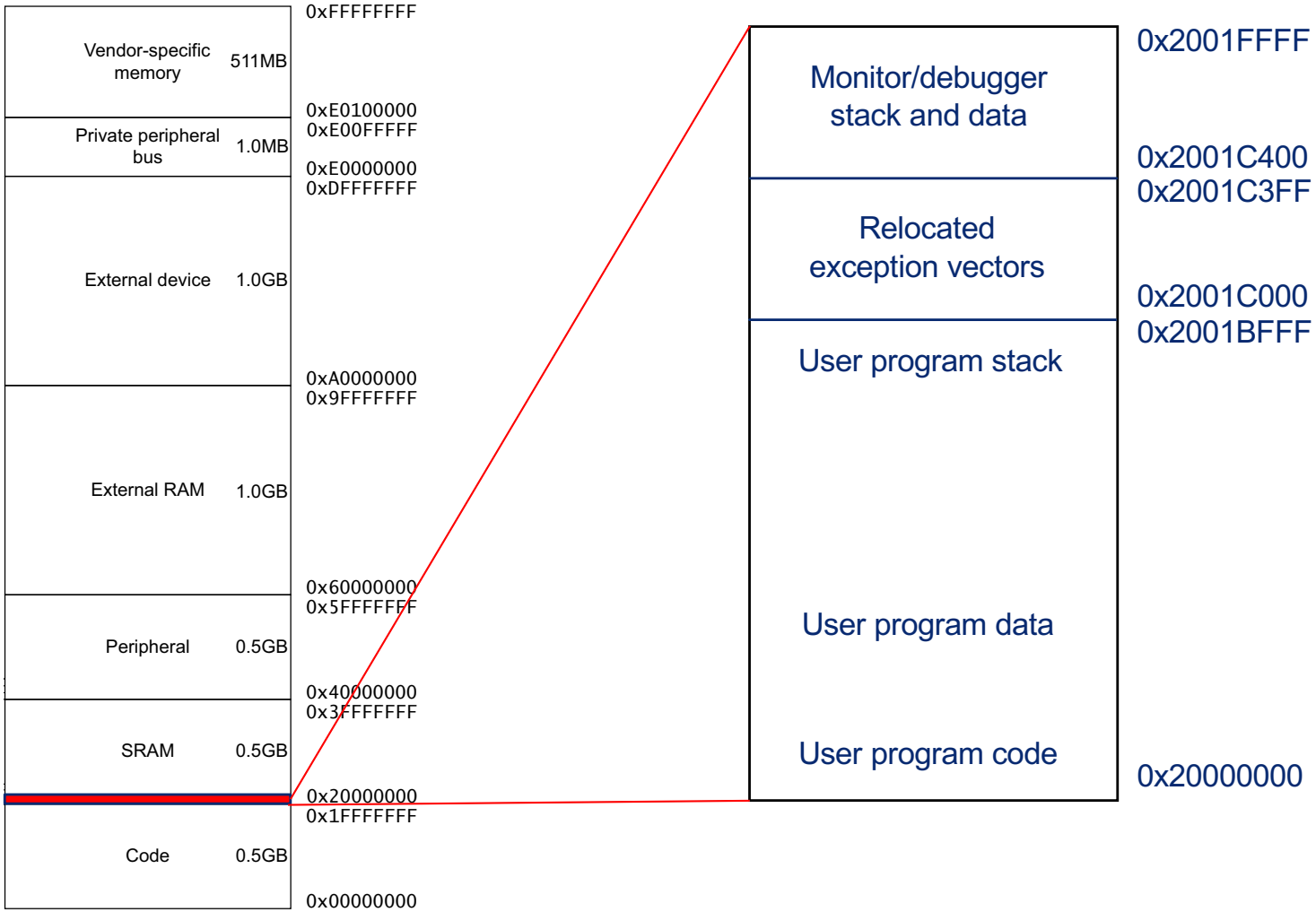


STM32F407 address space:

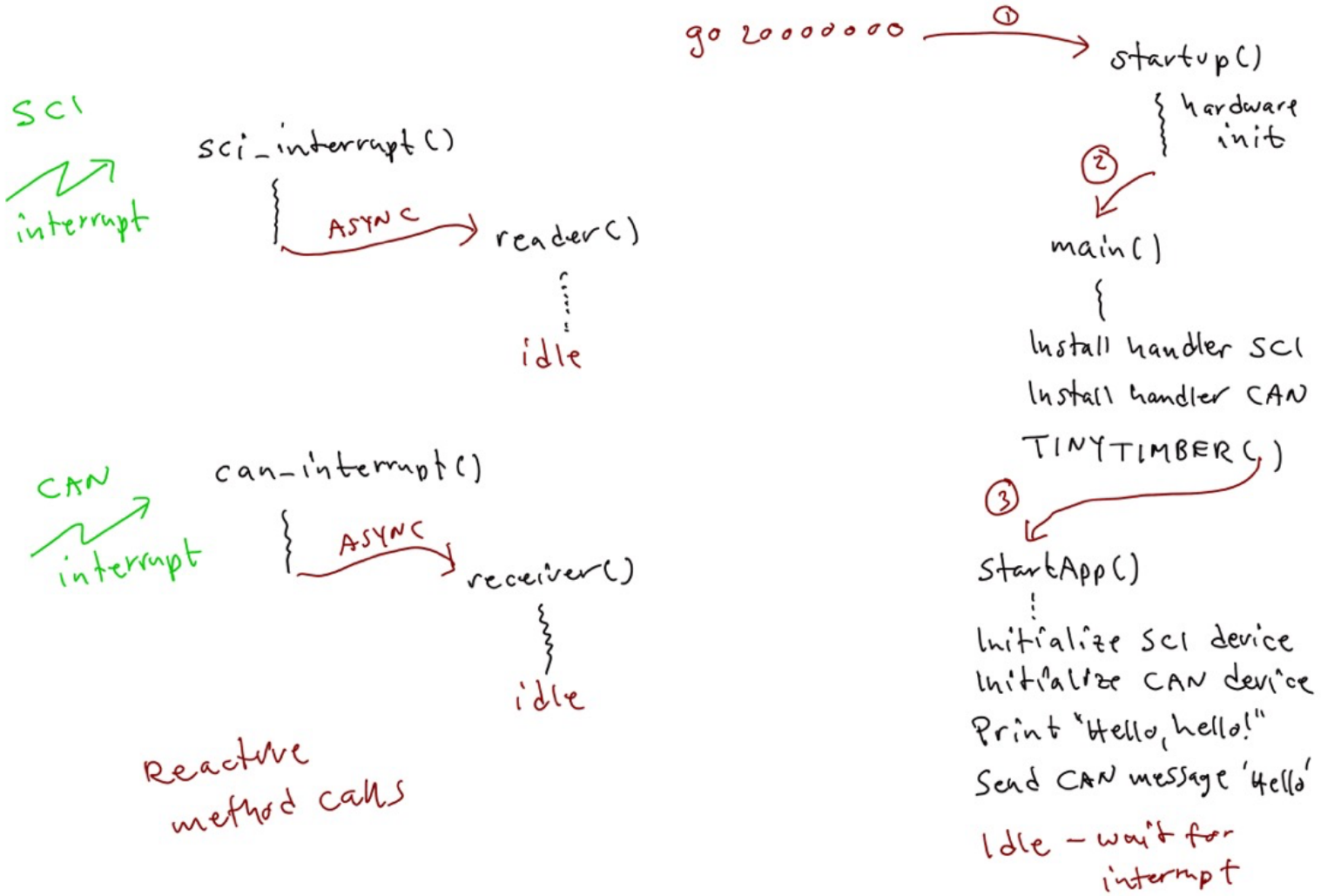




STM32F407 address space:



Exercise #1 – blackboard scribble



Real-Time Systems

Exercise #2

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

Call-back in TinyTimber

Device-driver call-back functionality in TinyTimber:

- The device drivers in TinyTimber contain interrupt handler code for any hardware event that may be associated with the device.
- For example, when pressing a key on the console keyboard the interrupt handler of the SCI device driver is activated.

  The SCI interrupt handler will then call the method provided by the user (via the call-back information) when creating the corresponding SCI device object.

  The typed keyboard character will be provided as the second parameter in the call-back method call.

Example: SCI integer parser

Problem: Input decimal integer values from the keyboard and print the values to the console using the TinyTimber kernel.

- The call-back and print-out functionality of the SCI device driver should be used.
- The typed individual characters of the integer value should be stored within a suitable object.
- The input of an integer value should be ended by typing the delimiter character ‘e’.
- The input integer value character string should be converted to C integer data type before being printed out.

Call-back in TinyTimber

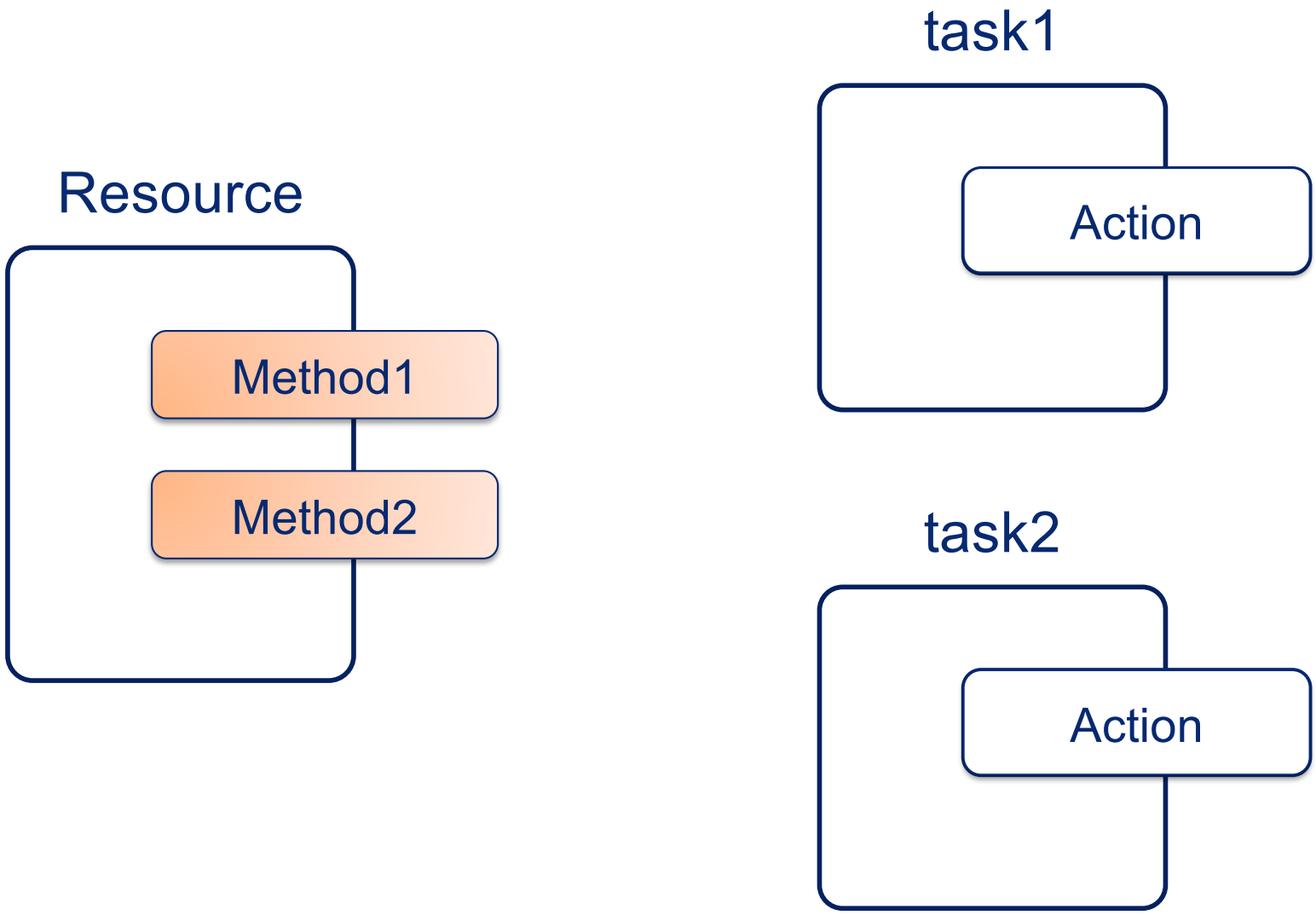
Object call-back functionality in TinyTimber:

- TinyTimber guarantees that an object is handled like an exclusive resource during the execution of a method that belongs to the object if that method is called using `SYNC()`.

  If multiple concurrent calls, using `SYNC()`, are made to methods belonging to the same object, only one call will be granted access to the object. The other calls will be blocked (put in a waiting queue.)

  When the object is available again, one of the blocked calls will be unblocked and the corresponding method is executed by means of a basic call-back functionality in TinyTimber.

Call-back in TinyTimber



Call-back in TinyTimber

Object call-back functionality in TinyTimber (cont'd):

- Although this basic call-back functionality is sufficient in many cases, TinyTimber lacks one powerful property that protected objects, monitors and semaphores have:

  The basic call-back functionality in TinyTimber cannot account for conditions relating to the contents of an object.

  Note: this prevents us from implementing blocking versions of the `Get/Put` methods in the circular buffer example used in Lecture #5.

- Thus, in order to use advanced resource management with TinyTimber we must provide a call-back functionality add-on.

Call-back in TinyTimber

Call-back functionality add-on:

- A task requests access to a certain resource (object) with a call to a method belonging to that resource (object).
- If access is not granted (because a condition regarding the object state prevents this) the method call will be blocked.
- If the calling task used `ASYNC()` to request the resource the task itself is not blocked but continues executing code.
- Implementing call-back functionality means that a calling task supplies `ASYNC()` with information about a method to wake up (call back) when the resource becomes available.
- Since multiple tasks may want to request the resource, the provided call-back information must be stored in a queue.

Call-back in TinyTimber

Method parameter and return-value convention:

- TinyTimber uses a uniform approach to method definitions: all methods must have two parameters of specific types
  - The first parameter must be a pointer to an object of the class to which the method belongs. This pointer (often named '`self`') allows the methods to access the state variables of the object.
  - The second parameter must be of type '`int`' and can be used as an input parameter to the method (but can also be ignored).
- For this reason calls to method operations in the kernel (`TINYTIMBER()`, `ASYNC()`, `SYNC()`, `AFTER()`, ...) must include these parameters in addition to a method reference.
- The return value of a method must be of type '`int`', unless no value is returned (in which case type '`void`' is used).

Call-back in TinyTimber

Method parameter and return-value work-around:

- If an input parameter of type '`xxx`' (different than '`int`') is needed for the method, type casting the argument to type '`int`' must be performed at call time; then the parameter is type-cast back to type '`xxx`' within the method itself.
- If multiple input parameters are needed, they should be stored in a struct, and a pointer to the struct should be passed as the argument at call time (with appropriate type casting).
- This work-around is also applicable to return values.

Example: semaphores in C

Problem: Implement a class `Semaphore` in C using the TinyTimber kernel.

- The object should receive an initial value when it is created.
- The object should have two methods, `Wait` and `Signal`, that work in accordance with the definition of semaphores.
- The methods should have support for call-back functionality

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

`Init(s,n):` assign `s` an initial value `n`

`Wait(s):`

```
if s > 0 then
   s := s - 1;
else
   ”block calling task”;
```

`Signal(s):`

```
if ”any task that has called Wait(s) is blocked”
then
   ”allow one such task to execute”;
else
   s := s + 1;
```

Example: semaphores in C

Solution overview:

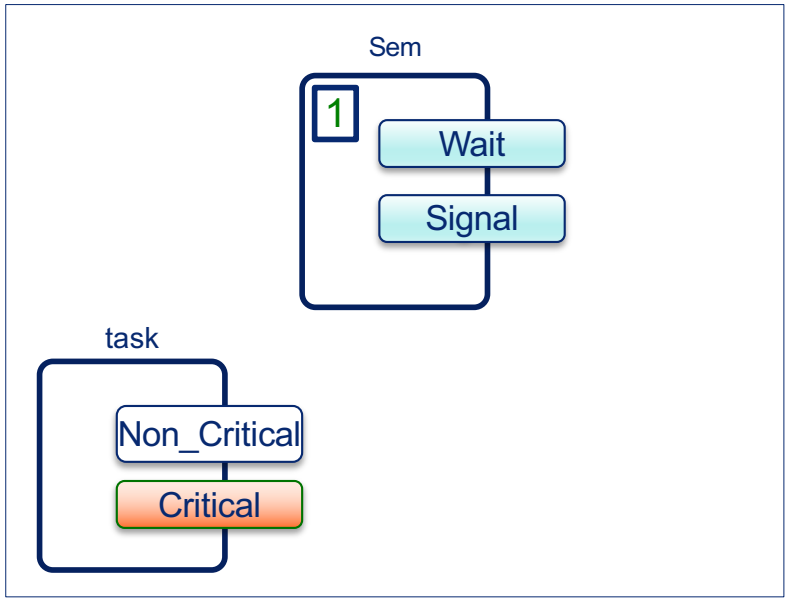
1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

```
// Template code for one task using semaphore Sem

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;              // store call-back info for this task
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb ); // acquire semaphore
}                                        // (note 'int' type-cast)

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);           // release semaphore
    ASYNC(self, Non_Critical, 0);    // restart "loop"
}
```

Example: semaphores in C

```
// Template code for one task using semaphore Sem

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb );
}

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);
    ASYNC(self, Non_Critical, 0);
}
```



Example: semaphores in C

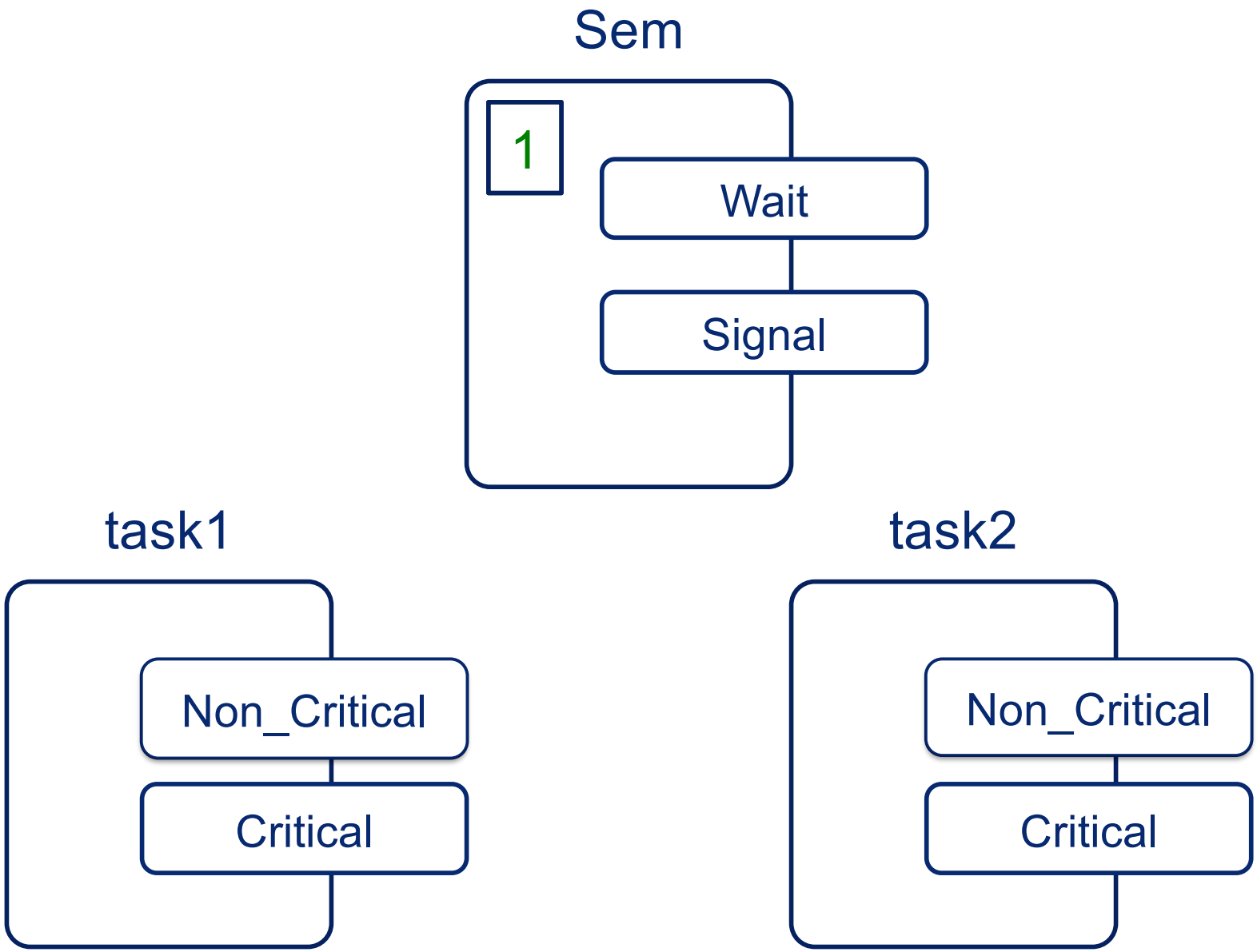
```
// Program code for two identical tasks using the same semaphore

typedef struct {
    Object super;
} Application;

Application app = { initObject() };

// Define Task class with two methods, Non_Critical() and Critical()

typedef struct {
    Object super;
    CallBlock cb;     // stored call-back info (see Appendix)
} Task;

void Non_Critical(Task*, int);
void Critical(Task*, int);

#define initTask() { initObject(), initCallBlock() }
```

Example: semaphores in C

```
// Create the semaphore and the two identical tasks

Semaphore Sem = initSemaphore(1);   // binary semaphore

Task task1 = initTask();
Task task2 = initTask();

void Non_Critical(Task *self, int unused) {
    Do_Non_Critical_Work();

    self->cb.obj = self;
    self->cb.meth = Critical;
    ASYNC(&Sem, Wait, (int) &self->cb ); // acquire semaphore
}

void Critical(Task *self, int unused) {
    Do_Critical_Work();

    SYNC(&Sem, Signal, 0);          // release semaphore
    ASYNC(self, Non_Critical, 0);
}
```

Example: semaphores in C

```
// TinyTimber first scheduled event

void kickoff(Application *self, int unused) {
    ASYNC(&task1, Non_Critical, 0);
    ASYNC(&task2, Non_Critical, 0);    // spawn two identical tasks
}

// Start the TinyTimber run-time system

// We enter main() after system startup
// and hand over control to TinyTimber

int main() {
    TINYTIMBER(&app, kickoff, 0);
}
```


Sem
1
Wait
Signal
task1
Non_Critical
Critical
task2
Non_Critical
Critical

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

```
// Define a class Semaphore with Wait and Signal methods,
// as well as an initialization macro

typedef struct {
    Object    super;
    int       value;
    Caller    queue; // queue of stored call-back info (see Appendix)
} Semaphore;

// Note that TinyTimber methods only accept type 'int' for the second
// parameter. This means that, if we want to send a parameter of another
// scalar type (i.e. a pointer), we will have to trick the system by
// "type casting" to 'int' before a call, and then back to the original
// type within the method.

void Wait(Semaphore*, int);
void Signal(Semaphore*, int);

#define initSemaphore(n) { initObject(), n, 0 }
```

Example: semaphores in C

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

Example: semaphores in C

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

`Init(s,n):` assign `s` an initial value `n`

`Wait(s):`

```
if s > 0 then
  s := s - 1;
else
  ”block calling task”;
```

`Signal(s):`

```
if ”any task that has called Wait(s) is blocked”
then
  ”allow one such task to execute”;
else
  s := s + 1;
```

Example: semaphores in C

```
// Implement the methods Wait and Signal

void Wait(Semaphore *self, int c) {
    Caller wakeup = (Caller) c;  // type-cast back from 'int'
    if (self->value > 0) {
        self->value--;
        ASYNC(wakeup->obj, wakeup->meth, 0);
    }
    else
        c_enqueue(wakeup, &self->queue); // (see Appendix)
}

void Signal(Semaphore *self, int unused) {
    if (self->queue) {
        Caller wakeup = c_dequeue(&self->queue); // (see Appendix)
        ASYNC(wakeup->obj, wakeup->meth, 0);
    }
    else
        self->value++;
}
```

Appendix: the call-back queue

Solution overview:

1. Show example application code that uses a semaphore
2. Define a class Semaphore with Wait and Signal methods, as well as an initialization macro
3. Implement the Wait and Signal methods
4. Define a data type for call-back information, that can also be stored as an element in a queue (Appendix)
5. Implement functions for manipulating a queue containing elements of the call-back information data type (Appendix)

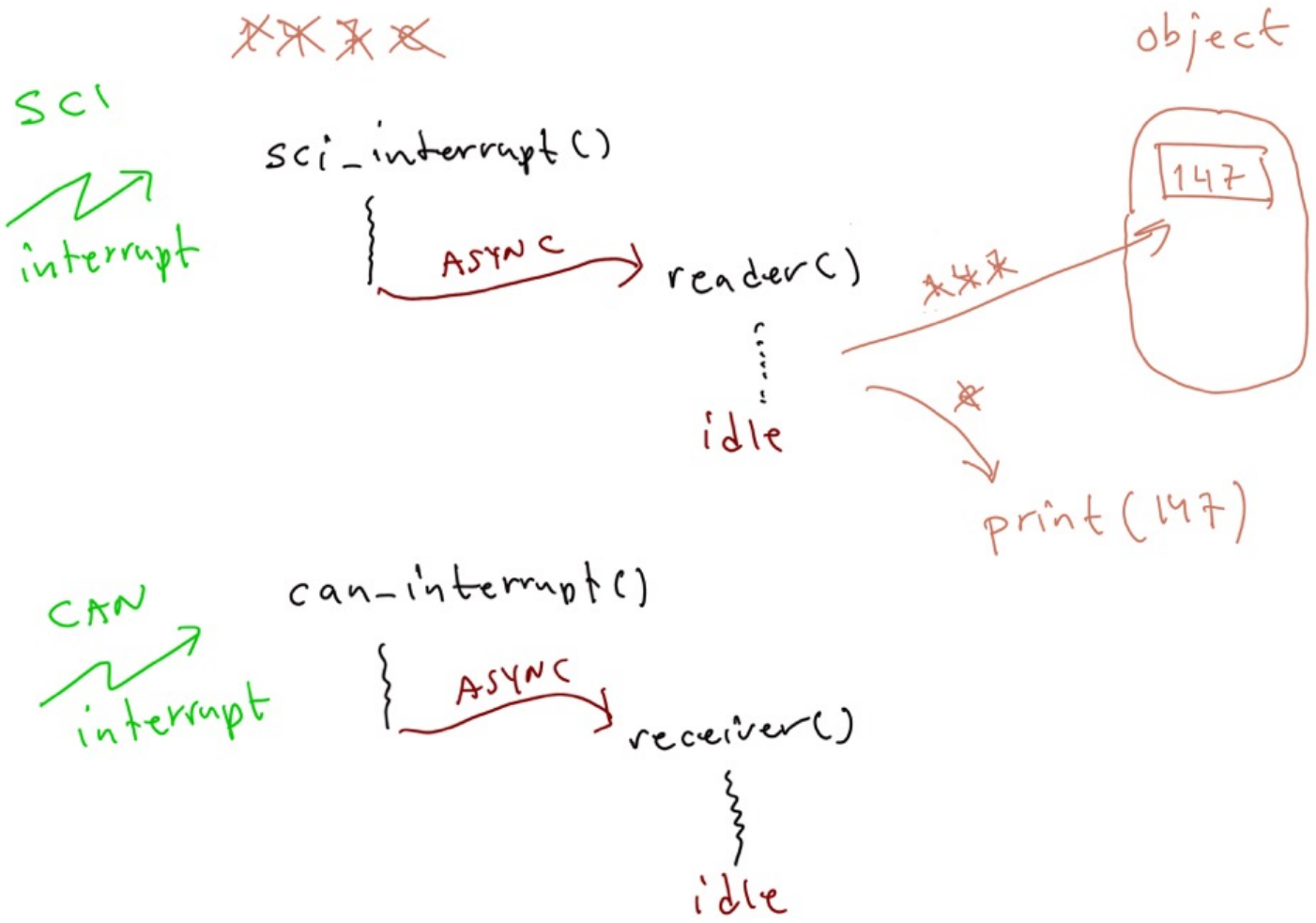
Appendix: the call-back queue

```
// Define a data type for call-back information, that can also
// be used as an element in a queue

struct call_block;
typedef struct call_block *Caller;

typedef struct call_block {
    Caller    next;  // for use in linked lists
    Object    *obj;
    Method    meth;
} CallBlock;

#define initCallBlock() { 0, 0, 0 }
```

Appendix: the call-back queue

```
// Implement functions for manipulating a queue containing
// elements of the call-back information data type

void c_enqueue(Caller c, Caller *queue) {
    Caller prev = NULL, q = *queue;
    while (q) { // find last element in queue
        prev = q;
        q = q->next;
    }
    if (prev == NULL)
        *queue = c;   // empty queue: put 'c' first
    else
        prev->next = c; // non-empty queue: put 'c' last
    c->next = NULL;
}

Caller c_dequeue(Caller *queue) {
    Caller c = *queue;
    if (c)
        *queue = c->next; // remove first element in queue
    return c;
}
```

Exercise #2 – blackboard scribble



Exercise #2 – blackboard scribble

SCI integer parser

```
#include "TinyTimber.h"
#include "sciTinyTimber.h"   <- SCI device driver functions
#include <stdlib.h>          <- to use atoi() function
#include <stdio.h>           <- to use snprintf() function

typedef struct {
  Object super;
  int index;                 <- next place to store typed character
  char buffer[50];           <- to store typed character
} App;

App app = {initObject(), 0, {0}};
void reader(App *, int);

Serial sci0 = initSerial(SCI_PORT0, &App, reader);
```

call-back information to SCI driver

Exercise #2 – blackboard scribble

```
void reader (App * self, int c) {        typed character on console keyboard
    switch (c) {
        case '0' ... '9':
        case '-':                        } valid characters for (decimal) integer
            self->buffer[self->index++] = c;
            break;
        case 'e':                        ← end of integer delimiter
            self->buffer[self->index] = '\0';   ← terminate C character string
            int value = atoi(self->buffer);     ← convert string to integer data type
            self->index = 0;             ← clean input buffer
            char snbuf[100];
            snprintf(snbuf, 100, "Entered integer: %d\n", value);
            SCI_WRITE(&sci0, snbuf);     ← output text to console
            break;
        default:
            break;
    }
}
```

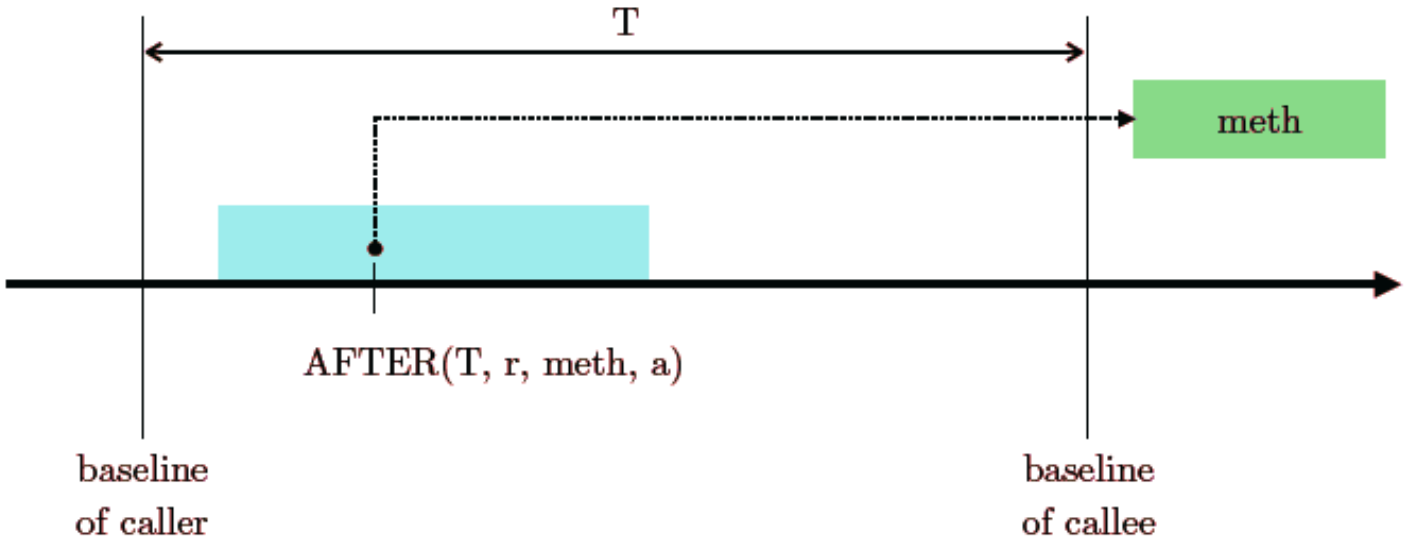
Real-Time Systems

Exercise #3

Victor Wallsten

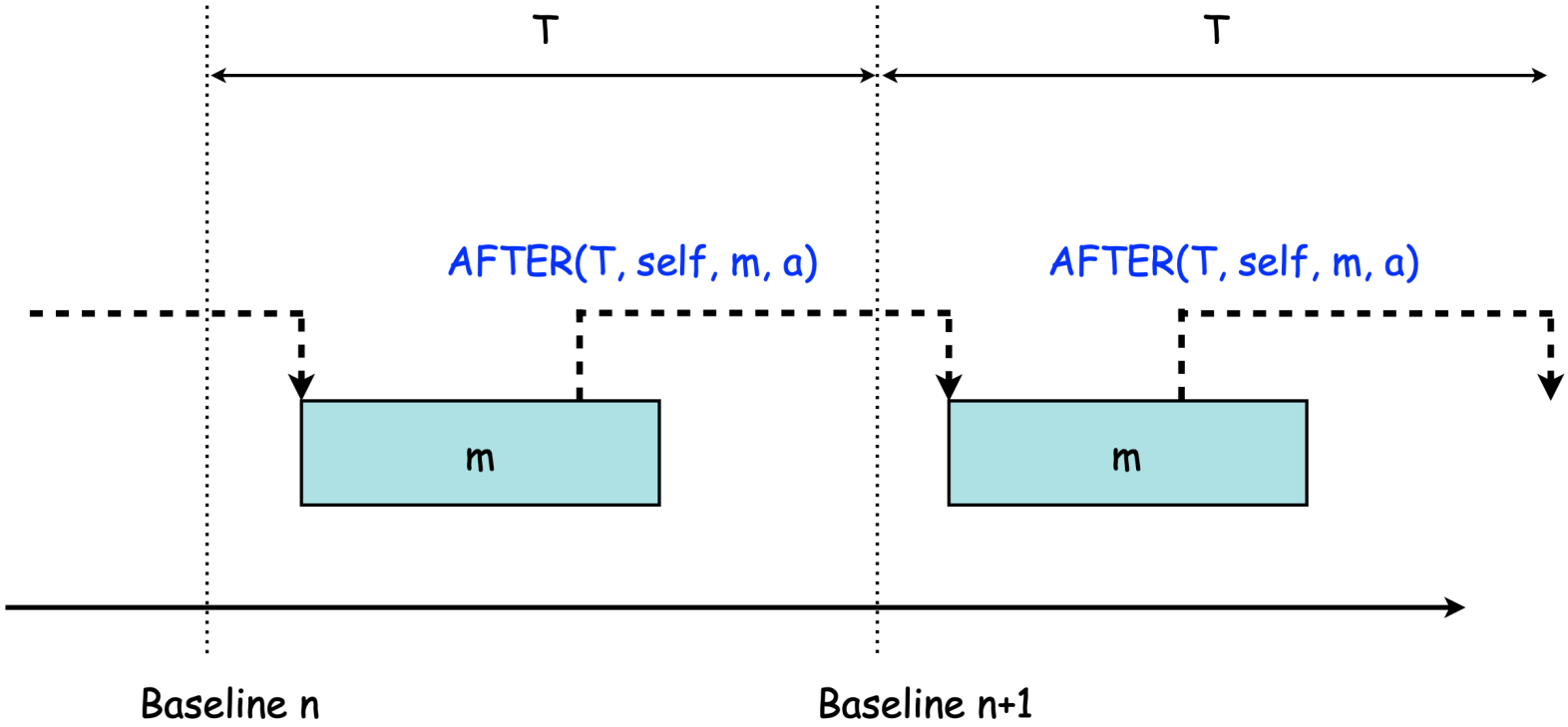
Department of Computer Science and Engineering
Chalmers University of Technology

Examples with AFTER()



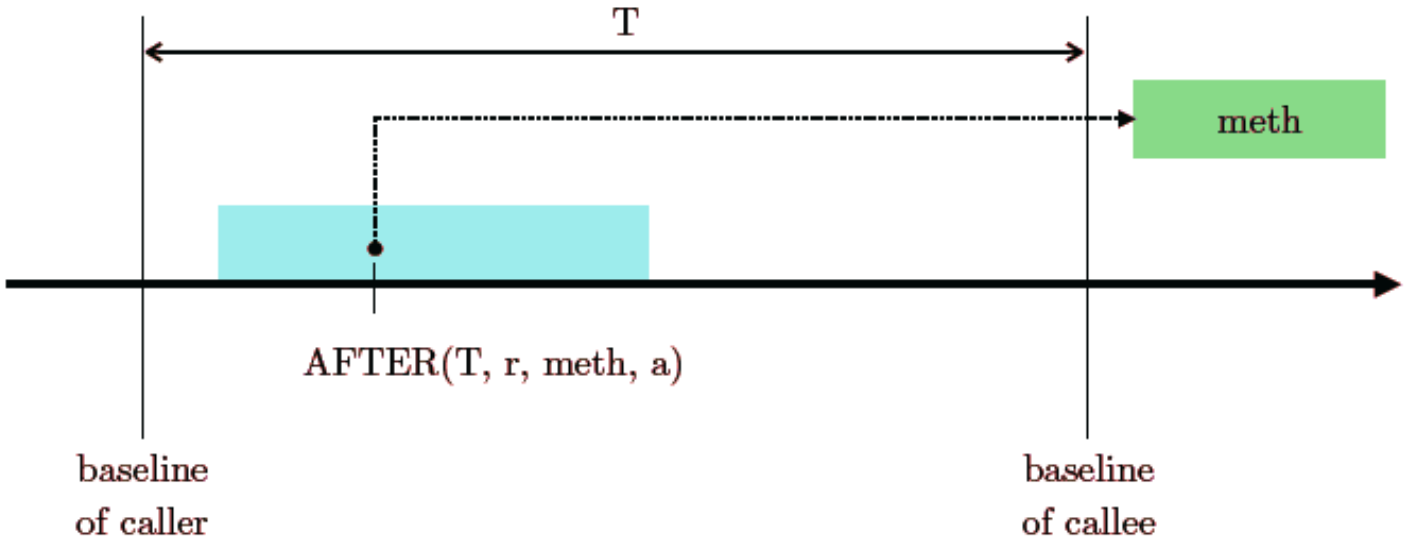

```
// Call method 'do_some_work' in object 'task1'
// after 2 ms
AFTER(MSEC(2),&task1,do_some_work,0);

// Call method 'do_more_work' in object 'task2'
// after 500 µs
AFTER(USEC(500),&task2,do_more_work,1);
```

Periodicity with AFTER()




```
// Call yourself after 15 ms
int do_some_work(MyObject *self, int n) {
      ... // do some work
      AFTER(MSEC(15),self,do_some_work,0);
}
```

Some more about `AFTER()`



An `AFTER()` call with a baseline of 0 means that the called method runs with the same baseline as the caller.

```
ASYNC(&obj,meth,n) == AFTER(0,&obj,meth,n);
```

Some more about AFTER()

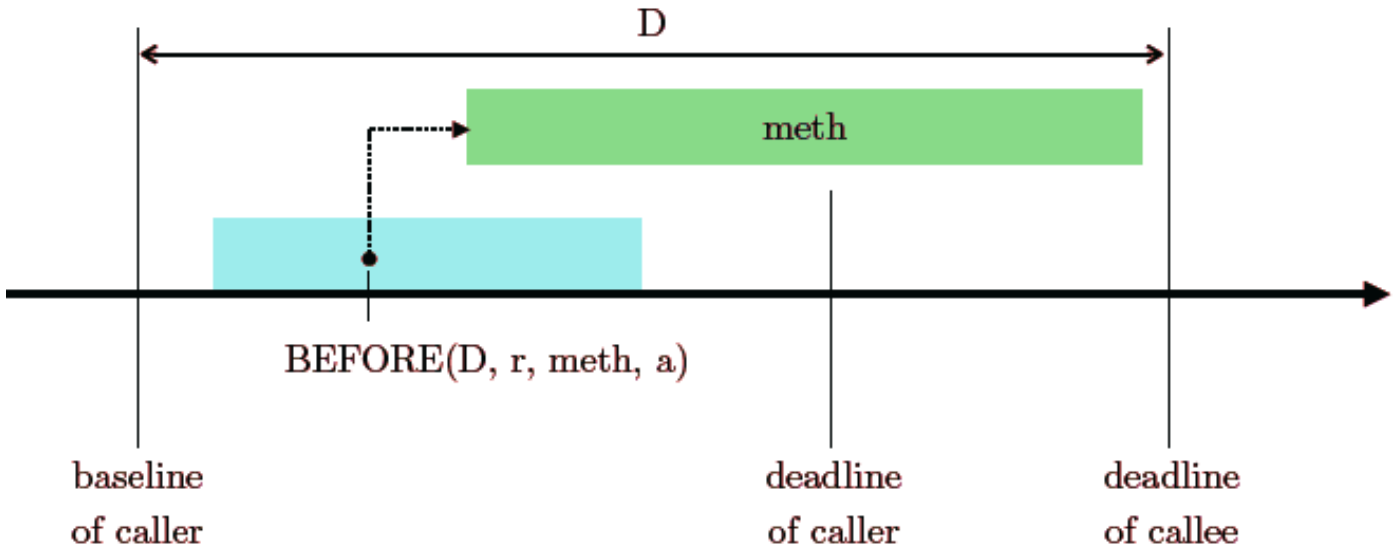
Using the baseline to derive time offsets makes the actual time the AFTER() call is made less critical!

```
int do_some_work(MyObject *self, int n) {
    ... // do some work
    AFTER(SEC(T),&obj,do_more_work,0);
}
```

has the same behavior as

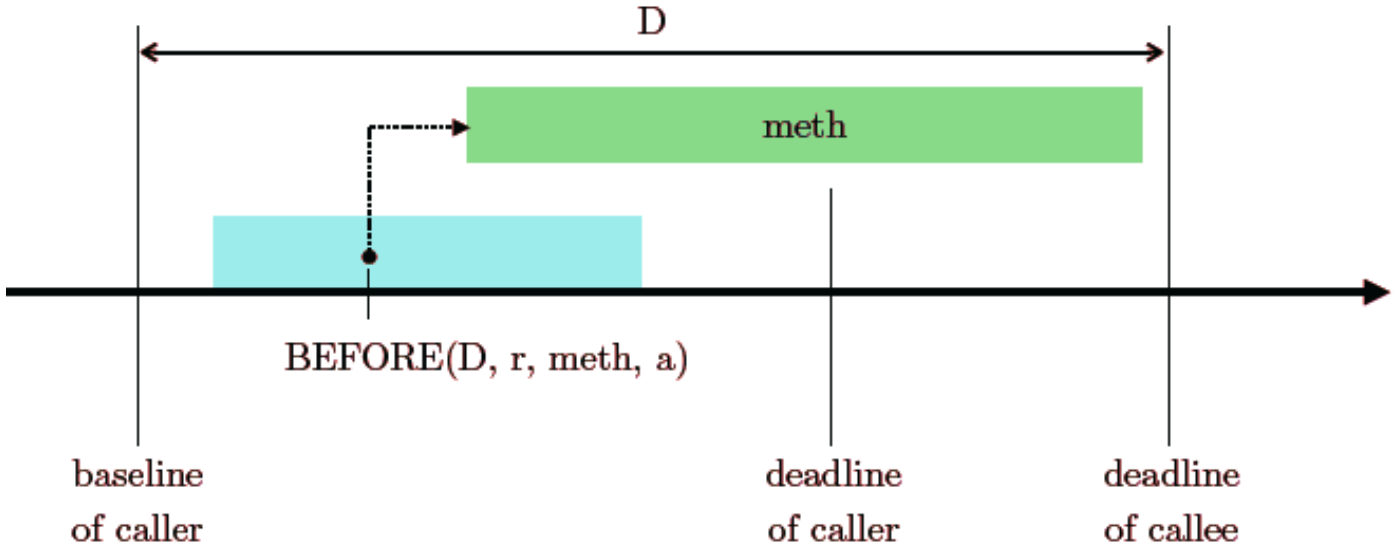
```
int do_some_work(MyObject *self, int n) {
    AFTER(SEC(T),&obj,do_more_work,0);
    ... // do some work
}
```

Examples with BEFORE()




```
// Call method 'stop' in object 'engine' with a
// deadline of 30 ms
BEFORE(MSEC(30),&engine,stop,0);

// Call method 'move' in object 'motor' with a
// deadline of 2 µs
BEFORE(USEC(2),&motor,move,1);
```

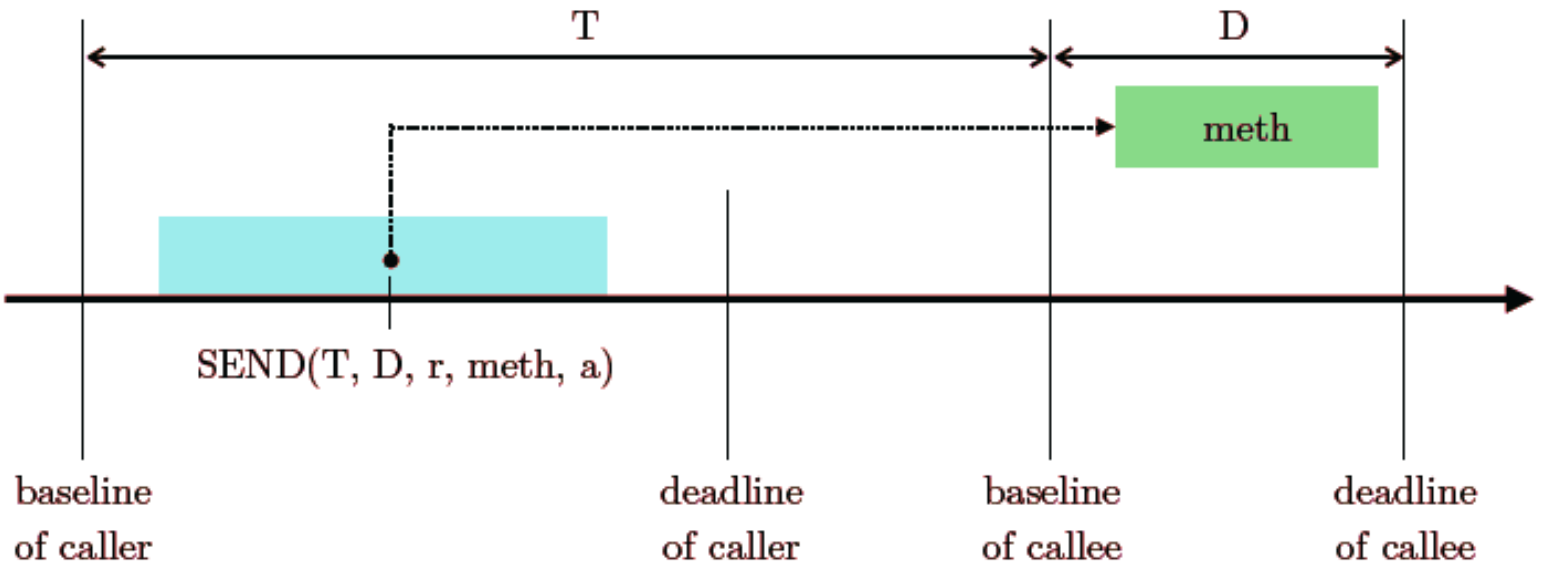
Some more about `BEFORE()`



The `BEFORE()` call has an implicit baseline of 0, i.e., the called method runs with the same baseline as the caller.

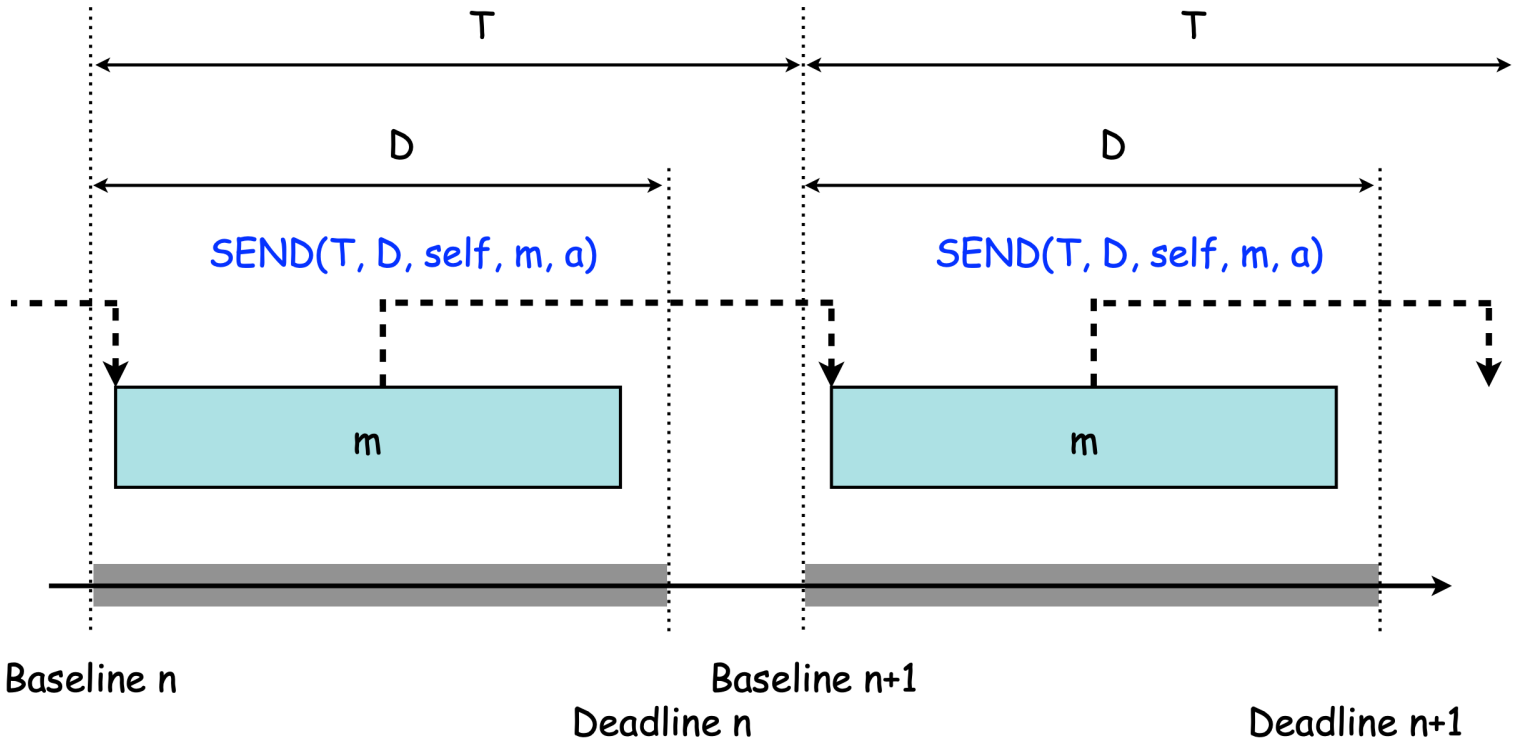
To assign a deadline to a delayed method call, you need to use the `SEND()` call.

Examples with SEND()



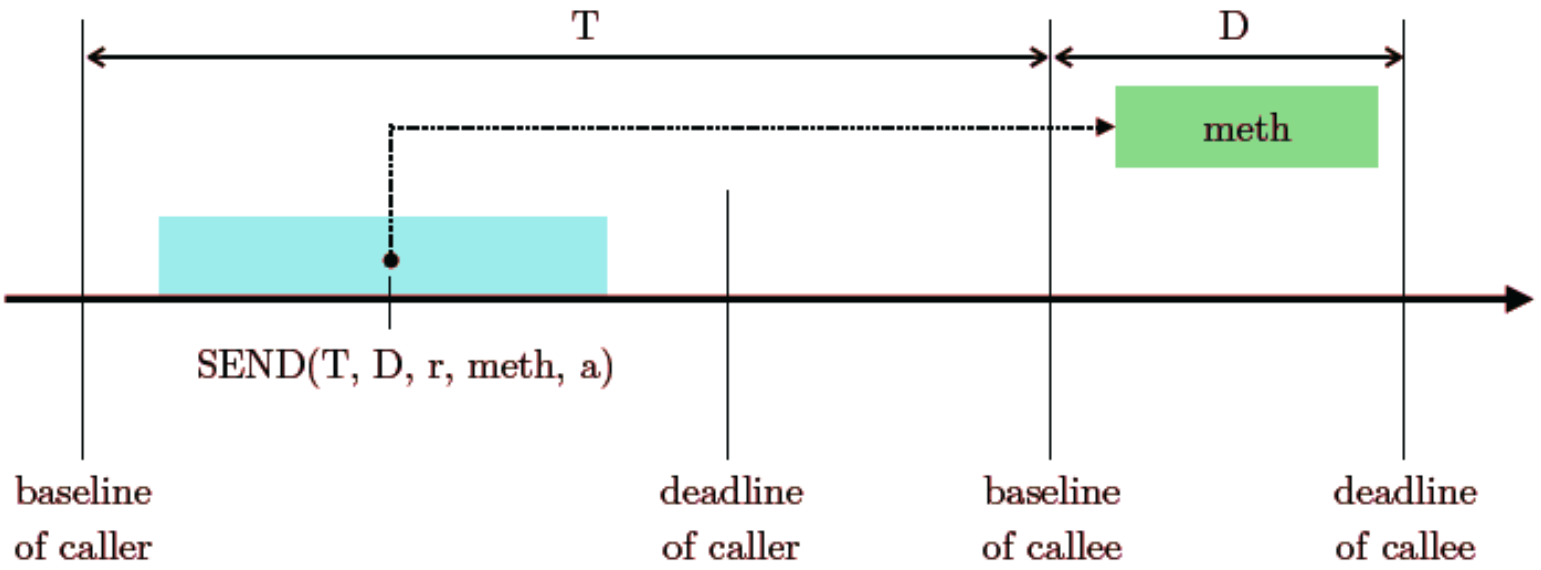

```
// Call method 'stop' in object 'engine' after 2 s,
// with a deadline of 30 ms
SEND(SEC(2),MSEC(30),&engine,stop,0);

// Call method 'move' in object 'motor' after 50 ms,
// with a deadline of 2 µs
SEND(MSEC(50),USEC(2),&motor,move,1);
```

Periodicity with SEND()




```
// Call yourself after 15 ms, with a deadline of 30 µs
int do_some_work(MyObject *self, int n) {
      ... // do some work
      SEND(MSEC(15),USEC(30),self,do_some_work,0);
}
```

Some more about SEND()



The SEND() call is the fundamental building block for the AFTER, BEFORE and ASYNC calls.

```
AFTER(T,&obj,meth,n) == SEND(T,0,&obj,meth,n);
BEFORE(D,&obj,meth,n) == SEND(0,D,&obj,meth,n);
ASYNC(&obj,meth,n) == SEND(0,0,&obj,meth,n);
```

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 μs and 150 μs to `task1` and `task2`, respectively.

Example: periodic tasks

Problem: Implement two periodic tasks with a shared object in C using the TinyTimber kernel.

- Assume that an object `actobj` of type `Actuator` is shared by two periodic tasks `task1` and `task2` with periods 300 μs and 500 μs, respectively.
- Both tasks may concurrently call method `update` of shared object `actobj` with a value 10 and 20, respectively.
- The old value of object `actobj` should be returned by the `update` method, to be used by the tasks.
- Add deadlines of 100 μs and 150 μs to `task1` and `task2`, respectively.
- Stop the execution of `task1` and `task2` after 100 ms and 200 ms, respectively.

Given code for Actuator object

```
typedef struct {
    Object super;
    int state;
} Actuator;

// Declare the update method
int update(Actuator *, int);

// Initialization macro
#define initActuator() { initObject(), 0 }

// Create an object of type Actuator
Actuator actobj = initActuator();
```

Given code for Actuator object

```
// This method updates the hardware with a new setting, and
// returns the old setting.

int update(Actuator *self, int new_value){
    int old_value = self->state;
    self->state = new_value;
    ...              // code updating the actuator hardware
    return old_value;
}
```

Template code for periodic tasks

```
typedef struct {// Class definition
    Object super;



} TaskObject;

// Method declarations
void task1code(TaskObject *, int);
void task2code(TaskObject *, int);

// Initialization macro
#define initTaskObject(        ) { initObject()                }

// Create two objects of type TaskObject
TaskObject task1 = initTaskObject(                              );
TaskObject task2 = initTaskObject(                              );
```

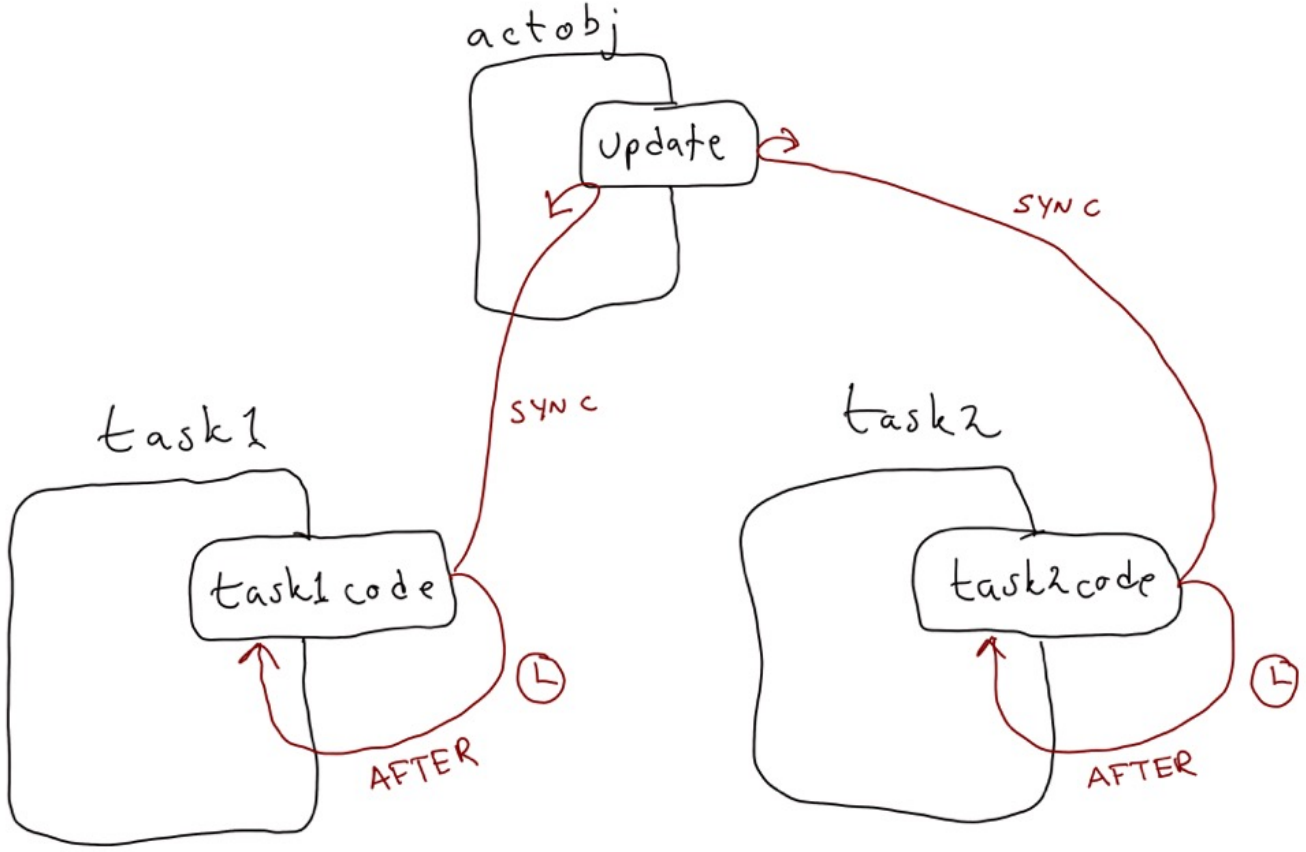
Template code for periodic tasks

```
// Each task sends a new value to method actobj, and uses
// the old value returned from method actobj
void task1code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something with the old value


}

void task2code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with the old value


}
```

Template code for periodic tasks

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   // Give an initial value of 10 for task1, and 20 for task2









}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Exercise #3 – blackboard scribble



Exercise #3 – blackboard scribble

```
typedef struct {// Class definition
    Object super;
    Time period;

} TaskObject;

// Method declarations
void task1code(TaskObject *, int);
void task2code(TaskObject *, int);

// Initialization macro
#define initTaskObject( P      ) { initObject(), P            }

// Create two objects of type TaskObject
TaskObject task1 = initTaskObject( USEC(300)                  );
TaskObject task2 = initTaskObject(USEC(500)                   );
```

Exercise #3 – blackboard scribble

```
// Each task sends a new value to method actobj, and uses
// the old value returned from method actobj
void task1code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something with the old value
   AFTER (self->period, self, task1code, value);

}

void task2code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with the old value
   AFTER (self->period, self, task2code, value);

}
```

Exercise #3 – blackboard scribble

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   // Give an initial value of 10 for task1, and 20 for task2
   ASYNC(&task1, task1code, 10);
   ASYNC(&task2, task2code, 20);




}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Exercise #3 – blackboard scribble

```
typedef struct {// Class definition
    Object super;
    Time period;
    Time deadline;

} TaskObject;

// Method declarations
void task1code(TaskObject *, int);
void task2code(TaskObject *, int);

// Initialization macro
#define initTaskObject( p, d  ) { initObject(), p, d        }

// Create two objects of type TaskObject
TaskObject task1 = initTaskObject( USEC(300), USEC(100)     );
TaskObject task2 = initTaskObject(USEC(500), USEC(150)      );
```

Exercise #3 – blackboard scribble

```
// Each task sends a new value to method actobj, and uses
// the old value returned from method actobj
void task1code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something with the old value

   SEND(self→period, self→deadline, self, task1code, value);

}

void task2code(TaskObject *self, int value){

   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with the old value

   SEND(self→period, self→deadline, self, task2code, value);

}
```

Exercise #3 – blackboard scribble

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   // Give an initial value of 10 for task1, and 20 for task2
   BEFORE(USEC(100), &task1, task1code, 10);
   BEFORE(USEC(150), &task2, task2code, 20);




}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Exercise #3 – blackboard scribble

```
typedef struct {// Class definition
    Object super;
    Time period;
    Time deadline;
    int running;
} TaskObject;

// Method declarations
void task1code(TaskObject *, int);
void task2code(TaskObject *, int);

// Initialization macro
#define initTaskObject( P, d ) { initObject(), P, d, 1 }

// Create two objects of type TaskObject
TaskObject task1 = initTaskObject( USEC(300), USEC(100) );
TaskObject task2 = initTaskObject(USEC(500), USEC(150) );
```

Exercise #3 – blackboard scribble

```
// Each task sends a new value to method actobj, and uses
// the old value returned from method actobj
void task1code(TaskObject *self, int value){
 if (self→running) {
   int old_state = SYNC(&actobj, update, value);
   ...       // do something with the old value

   SEND(self→period, self→deadline, self, task1code, value);
 }
}

void task2code(TaskObject *self, int value){
 if (self→running) {
   int old_state = SYNC(&actobj, update, value);
   ...       // do something else with the old value

   SEND(self→period, self→deadline, self, task2code, value);
 }
}
```

Exercise #3 – blackboard scribble

```
// How to begin the initial invocation?

void kickoff(TaskObject *self , int unused) {
   // Give an initial value of 10 for task1, and 20 for task2
   BEFORE(USEC(100), &task1, task1code, 10);
   BEFORE(USEC(150), &task2, task2code, 20);
   AFTER(MSEC(100), &task1, stop, 0);
   AFTER(MSEC(200), &task2, stop, 0);
}

int main() {
   TINYTIMBER(&task1, kickoff, 0);
   return 0;
}
```

Exercise #3 – blackboard scribble

```
// Method which stops the periodic task

void stop (Taskobject *self, int unused) {
    self->running = 0;
}
```

Real-Time Systems

Exercise #4

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

WCET Analysis using Shaw's Method

"The estimated WCET is the execution time of the longest structural path through the program."

The following example is based on Problem 3 in the Exercise Compendium (Collection of Examples).

Example: WCET analysis

Problem: Consider the function `Calculate()`.

a) Using Shaw's method, estimate the WCET for function `Calculate()` in terms of 'Z' (with Z ≥ 0).

```
int Calculate (int Z){
   int R;
   if(Z == 0)
      R = 1;
   else if (Z == 1)
      R = 1;
   else
      R = Calculate(Z-1) + Calculate(Z-2);
   return R;
}
```

Example: WCET analysis

Problem: Consider the function `Calculate()`.

a) Using Shaw's method, estimate the WCET for function `Calculate()` in terms of 'Z' (with $Z \geq 0$).

- Each *declaration* or *assignment* statement costs 1 time unit
- Each *compare* statement costs 1 time unit
- Each *return* statement costs 1 time unit
- Each *addition* or *subtraction* operation costs 4 time units.
- A *function call* costs 2 time units plus WCET for the function in question.
- All other language constructs can be assumed to take 0 time units to execute.

Example: WCET analysis

Problem: Consider the function `Calculate()`.

b) Function `main()` calls function `Calculate()` with parameter 5. What is the WCET of function `main()`?

c) The deadline for executing function `main()` is 180 time units. Determine whether the deadline is met or not.

```
int main(){
    int ans;
    ans = Calculate(5);
}
```

Example: WCET analysis

Problem: Now the program runs on a new processor that has a faster ALU. The execution costs of addition and subtraction are equal, but smaller than that of the older processor.

Let ‘x’ represent the execution time of an addition/subtraction operation. All other language constructs are assumed to have the same cost as in sub-problem a).

d) What is the WCET for function `main()` in terms of ‘x’?

e) What is the maximum cost of an addition/subtraction operation so that the deadline of function `main()` is met?

Exercise #4 – blackboard scribble

Declare/Assign : 1 compare : 1	Return : 1 Add/Subtract : X	Call func : 2 + WCET(func) other : 0

Case 1 : z == 0

WCET(Calc(0)) = {Declare, R} + {Compare, z==0} +

{Assign, R} + {Return, R} = 1+1+1+1 = 4

Case 2 : z == 1

WCET(Calc(1)) = {Declare, R} + {Compare, z==0} +

{Compare, z==1} + {Assign, R} + {Return, R} = 1+1+1+1+1 = 5

```
int Calc(int z){
    int R;
    if (z==0) R=1;
    else if (z==1) R=1;
    else R = Calc(z-1) + Calc(z-2);
    return R;
}
```

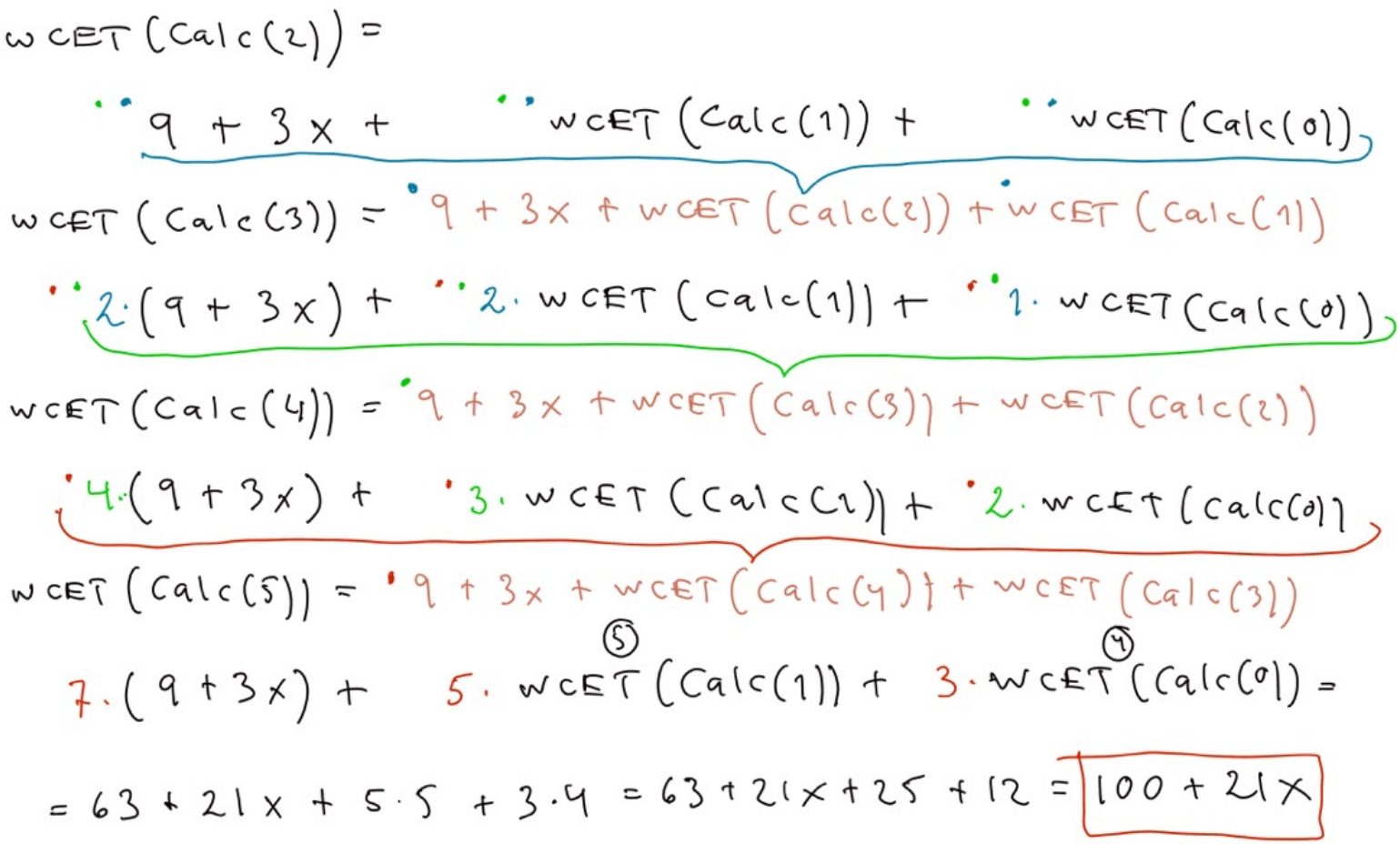
Exercise #4 – blackboard scribble

Declare/Assign : 1 compare : 1	Return : 1 Add/Subtract : X	Call func : 2 + WCET(func) other : 0

Case 3 : z > 1 (let x be cost for add & sub)

$$WCET(Calc(z>1)) = \{Declare^{1}, R\} + \{Compare^{1}, z==0\} + \{Compare^{1}, z==1\} +$$
$$\{Subtract^{X}, z-1\} + \{Call^{2}, Calc(z-1)\} + WCET(Calc(z-1)) +$$
$$\{Subtract^{X}, z-2\} + \{Call^{2}, Calc(z-2)\} + WCET(Calc(z-2)) +$$
$$\{Add^{X}, Calc(z-1)+Calc(z-2)\} + \{Assign^{1}, R\} + \{Return^{1}, R\} =$$
$$1+1+1+X+2+WCET(Calc(z-1)) +$$
$$X+2+WCET(Calc(z-2)) + X+1+1 =$$
$$9+3X+WCET(Calc(z-1))+WCET(Calc(z-2))$$

Exercise #4 – blackboard scribble



Exercise #4 – blackboard scribble

b) Derive WCET for main() (let x=4)

$$WCET(main()) = \{Declare_1, ans\} + \{Call_2, Calc(5)\} + WCET(Calc(5))^{100+21x} + \{Assign_1, ans\} =$$

$$1 + 2 + WCET(Calc(5)) + 1 = 4 + WCET(Calc(5)) =$$

$$4 + 100 + 21x = 104 + 21x = \{x=4\} = 104 + 84 = 188$$

c) Check deadline for main()

$$WCET(main()) = 188 > Deadline = 180 \Rightarrow Fail!$$

```
int main(){
  int ans;
  ans = Calc(5);
}
```

Exercise #4 – blackboard scribble

d) Derive WCET of main() in terms of X

According to subproblem b):

$WCET(main()) = 104 + 21X$

e) Find maximum cost X

Deadline

$WCET(main()) = 104 + 21X \leq 180 \Rightarrow 21X \leq 76$

That is: $X \leq \frac{76}{21} \approx 3,6$

If only integer values of X: Select $X = 3$ in order to meet deadline

Real-Time Systems

Exercise #5

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

Scheduling

"With cyclic executives the schedule is generated off-line, and stored in a time table. The schedule can be generated by simulating a run-time system with pseudo-parallel execution."

"With pseudo-parallel execution the schedule is generated on-line, as a side-effect of tasks being executed. Ready tasks are sorted in a queue and receive access to the processor based on priority."

Example 1: Cyclic executive

Problem: Consider a real-time system with two periodic tasks that should be scheduled using a time table. The parameters for the two tasks are given below. Both tasks arrive the first time at time 0.

a) Construct a time table for the execution of the two tasks. The tasks are allowed to preempt each other.

	C_i	D_i	T_i
τ_1	2	5	5
τ_2	4	7	7

Example 1: Cyclic executive

Problem: Consider a real-time system with two periodic tasks that should be scheduled using a time table. The parameters for the two tasks are given below. Both tasks arrive the first time at time 0.

a) Construct a time table for the execution of the two tasks. The tasks are allowed to preempt each other.

b) Does your schedule constitute the best possible schedule, or does there exist a superior one?

	C_i	D_i	T_i
τ_1	2	5	5
τ_2	4	7	7

Example 2: Pseudo-parallel execution

Problem: Consider a real-time system with three periodic tasks. The parameters for the three tasks are given below. All tasks arrive the first time at time 0.

a) Can you guarantee the schedulability of the task set using the RM scheduling algorithm?

Task	C_i	D_i	T_i
τ_1	1	7	7
τ_2	1	14	14
τ_3	4	18	18

Example 2: Pseudo-parallel execution

Problem: Consider a real-time system with three periodic tasks. The parameters for the three tasks are given below. All tasks arrive the first time at time 0.

a) Can you guarantee the schedulability of the task set using the RM scheduling algorithm?

b) Add a task τ_4 with $D_4 = T_4 = 100$ and $C_4 = x$ to the task set. What is the maximum value of x such that the new task set is schedulable for RM scheduling based on Liu and Layland's utilization test?

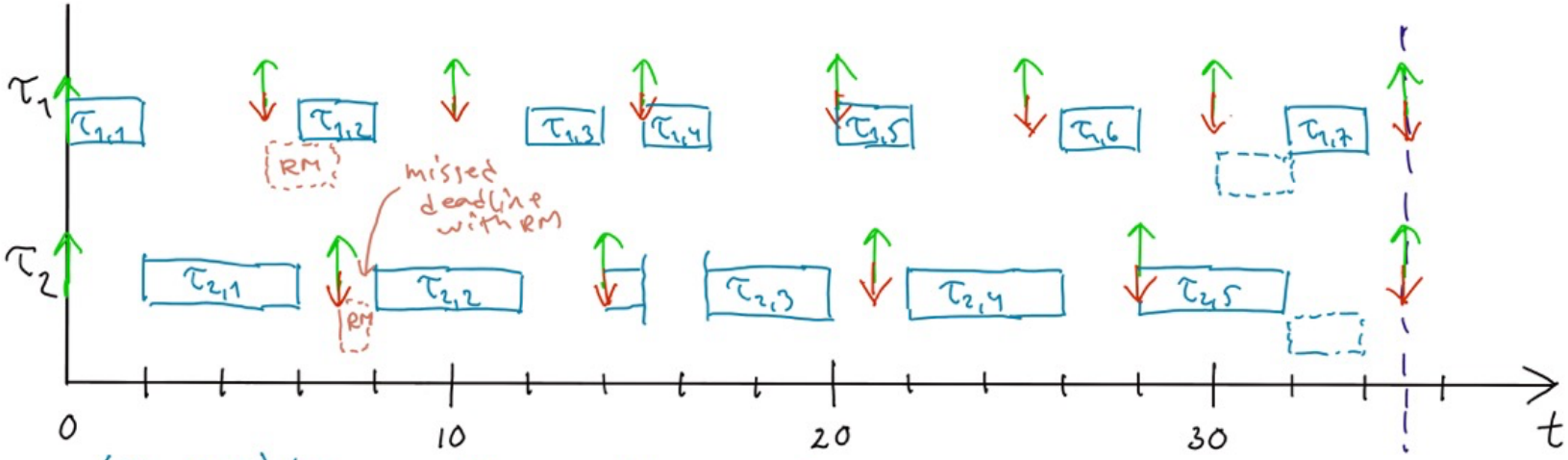
Task	C_i	D_i	T_i
τ_1	1	7	7
τ_2	1	14	14
τ_3	4	18	18

Exercise #5 – blackboard scribble

a) Simulate the execution of tasks using EDF scheduling with the hyperperiod

$LCM\{5, 7\} = 35$

Task	C_i	D_i	T_i
τ_1	2	5	5
τ_2	4	7	7



$(\tau_{1,1}, 0, 2)\ (\tau_{2,1}, 2, 6)\ (\tau_{1,2}, 6, 8)\ (\tau_{2,2}, 8, 12)\ (\tau_{1,3}, 12, 14)\ (\tau_{2,3}, 14, 15)\ (\tau_{1,4}, 15, 17)$

$(\tau_{2,3}, 17, 20)\ (\tau_{1,5}, 20, 22)\ (\tau_{2,4}, 22, 26)\ (\tau_{1,6}, 26, 28)\ (\tau_{2,5}, 28, 32)\ (\tau_{1,7}, 32, 34)$

b) Since EDF scheduling is known to be optimal for the given assumption this is the best possible schedule. [How would RM behave?]

Exercise #5 – blackboard scribble

a) The utilization of the task set is

$$U = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{1}{7} + \frac{1}{14} + \frac{4}{18} \approx 0{,}44$$

Task	C_i	D_i	T_i
τ_1	1	7	7
τ_2	1	14	14
τ_3	4	18	18

$D_i = T_i$

The utilization bound U_{RM} for $n=3$ tasks is

$$U_{RM} = n\left(2^{1/n} - 1\right) = 3 \cdot \left(2^{1/3} - 1\right) \approx 0{,}78 \qquad U < U_{RM}$$

The test succeeds $\Rightarrow$ The task set is schedulable!

Exercise #5 – blackboard scribble

b) The utilization of the task set is

$$U = \sum_{i=1}^{4} \frac{C_i}{T_i} = \frac{1}{7} + \frac{1}{14} + \frac{4}{18} + \frac{X}{100} = 0{,}44 + 0{,}01X$$

The utilization bound U_{RM} for 4 tasks is

$$U_{RM(4)} = 4\left(2^{1/4} - 1\right) \approx 0{,}76$$

The test succeeds if

$$U \leq U_{RM(4)} \Rightarrow 0{,}44 + 0{,}01X \leq 0{,}76 \Rightarrow$$

$$\Rightarrow 0{,}01X \leq 0{,}76 - 0{,}44 \Rightarrow X \leq 100 \cdot 0{,}32 = 32$$

Task	C_i	D_i	T_i
τ_1	1	7	7
τ_2	1	14	14
τ_3	4	18	18
τ_4	X	100	100

Note: this result assumes Liu & Layland's sufficient test.
other analysis methods could give higher values of X.
(e.g. response-time analysis)

Real-Time Systems

Exercise #7

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

Multiprocessor scheduling

Partitioned scheduling: "If all tasks are assigned using the Rate-Monotonic-First-Fit (RMFF) algorithm, then all tasks are schedulable if the total task utilization does not exceed 41% of the total processor capacity."

Global scheduling: "If tasks with the highest utilization are given highest priority and the remaining tasks are given RM priorities according to RM-US, then all tasks are schedulable if the total task utilization does not exceed 33.3% of the total processor capacity."

Example 1: RMFF scheduling

Problem:

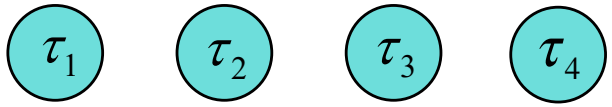
There are two approaches for scheduling tasks on multiprocessor platform: the *partitioned* approach and the *global* approach. The table below shows C_i (WCET) and T_i (period) for six periodic tasks to be scheduled on $m = 3$ processors. The relative deadline of each periodic task is equal to its period.

	C_i	T_i
τ_1	2	10
τ_2	10	25
τ_3	12	30
τ_4	5	10
τ_5	8	20
τ_6	7	100

The task set is schedulable using rate-monotonic first-fit (RMFF) partitioned scheduling algorithm. Show how the task set is partitioned on $m = 3$ processors so that all the deadlines are met using RMFF scheduling?

Example 2: RM-US scheduling

Problem: Consider the task set below for a system using global scheduling on m=3 processors. Show that the task set is schedulable on the processors assuming that task priorities are given according to RM-US[m/(3m-2)].



Task	C_i	T_i
τ_1	1	7
τ_2	4	19
τ_3	9	20
τ_4	11	22

Exercise #7 – blackboard scribble

Example 1:

The utilization of the task set is:

$$U = \sum_{\forall i} \frac{C_i}{T_i} = 0{,}2 + 0{,}4 + 0{,}4 + 0{,}17 + 0{,}4 + 0{,}07 = 1{,}97$$

The utilization bound for RMFF is {Oh & Baker}

$$U_{RMFF} = m\left(2^{1/2} - 1\right) = \{m = 3\} = 3\left(2^{1/2} - 1\right) \approx 1{,}24$$

(m: # of processors)

Task	C_i	T_i	U_i
τ_1	2	10	0,2
τ_2	10	25	0,4
τ_3	12	30	0,4
τ_4	5	10	0,5
τ_5	8	20	0,4
τ_6	7	100	0,07

Since $U > U_{RMFF}$ the test fails!

However, since the test is only sufficient failure does not imply non-schedulability.

We will use the RMFF algorithm to show schedulability.

Exercise #7 – blackboard scribble

Rate-Monotonic First-Fit (RMFF) {Dhall & Liu}

- Number the processors $M_1, M_2, \ldots M_m$
- Assign tasks in order of increasing periods
- For each task τ_i, choose the lowest-indexed (previously-used) processor M_j such that τ_i, together with all tasks that have already been assigned to M_j, can be feasibly scheduled according to Lin & Layland's test for RM

Task	C_i	T_i	U_i
τ_1	2	10	0,2
τ_2	10	25	0,4
τ_3	12	30	0,4
τ_4	5	10	0,5
τ_5	8	20	0,4
τ_6	7	100	0,07

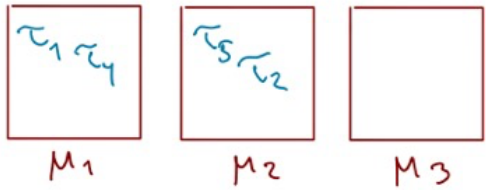
The three processors are numbered M_1, M_2 and M_3

The order of allocation (RM priority) is

$\tau_1, \tau_4, \tau_5, \tau_2, \tau_3, \tau_6$

Exercise #7 – blackboard scribble

~~τ_1~~, ~~τ_4~~, ~~τ_5~~, ~~τ_2~~, τ_3, τ_6



Task	C_i	T_i	U_i
τ_1	2	10	0,2
τ_2	10	25	0,4
τ_3	12	30	0,4
τ_4	5	10	0,5
τ_5	8	20	0,4
τ_6	7	100	0,07

Task τ_1 can be allocate to μ_1 since it is empty.

Task τ_4 can be allocated to μ_1 since

$U_1 + U_4 = 0,2 + 0,5 = 0,7 \leq U_{RM(2)} = 2(2^{1/2} - 1) \approx 0,82$

Task τ_5 cannot be allocated to μ_1 since

$U_1 + U_4 + U_5 = 0,2 + 0,5 + 0,4 = 1,1 > 1$

Task τ_5 can be allocated to μ_2 since it is empty

Task τ_2 cannot be allocated to μ_1 since

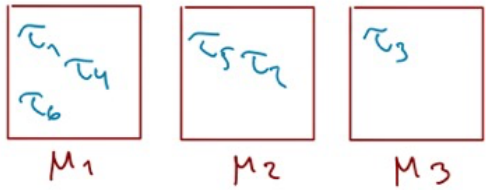
$U_1 + U_4 + U_2 = 0,2 + 0,5 + 0,4 = 1,1 > 1$

Task τ_2 can be allocated to μ_2 since

$U_5 + U_2 = 0,4 + 0,4 = 0,8 \leq U_{RM(2)} \approx 0,82$

Exercise #7 – blackboard scribble

~~$\tau_1, \tau_4, \tau_5, \tau_2, \tau_3, \tau_6$~~



Task	C_i	T_i	U_i
τ_1	2	10	0,2
τ_2	10	25	0,4
τ_3	12	30	0,4
τ_4	5	10	0,5
τ_5	8	20	0,4
τ_6	7	100	0,07

Task τ_3 cannot be allocated to μ_1 since

$U_1 + U_4 + U_3 = 0,2 + 0,5 + 0,4 = 1,1 > 1$

Task τ_3 cannot be allocated to μ_2 since

$U_5 + U_2 + U_3 = 0,4 + 0,4 + 0,4 = 1,2 > 1$

Task τ_3 can be allocated to μ_3 since it is empty

Task τ_6 can be allocated to μ_1 since

$U_1 + U_4 + U_6 = 0,2 + 0,5 + 0,07 = 0,77 \leq U_{RM(3)} = 3(2^{1/3} - 1) \approx 0,78$

All tasks could successfully be assigned with RMFF, so the tasks are schedulable on 3 processors.

Exercise #7 – blackboard scribble

Example 2:

Task	C_i	T_i	U_i
τ_1	1	7	0,14
τ_2	4	19	0,21
τ_3	9	20	0,45
τ_4	11	22	0,5

The utilization of the task set is

$$U = \sum_{\forall i} \frac{C_i}{T_i} = 0,14 + 0,21 + 0,45 + 0,5 = 1,30$$

The utilization bound for RM-US is

$$U_{RM\text{-}US} = m^2/(3m-2) = \{m=3\} = 9/7 \approx 1,29$$

Since $U > U_{RM\text{-}US}$ the test fails!

However, since the test is only sufficient failure does not imply non-schedulability. Instead, we will show schedulability using response-time analysis for global scheduling.

$$R_i = C_i + 1/m \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j + C_j \right)$$

Exercise #7 – blackboard scribble

$m = 3$ processors

Calculate utilization-separation (US) bound:

$$m/(3m-2) = 3/(3 \cdot 3 - 2) = 3/7 \approx 0{,}43$$

	Task	C_i	T_i	U_i
M	τ_1	1	7	0,14
L	τ_2	4	19	0,21
H	τ_3	9	20	0,45
H	τ_4	11	22	0,5

Derive task priorities:

- Based on the RM-US bound (0,43) tasks τ_3 and τ_4 are considered "heavy" tasks, and are assigned highest priority.
- Task τ_1 has highest RM priority
- Task τ_2 has lowest RM priority

Since we have 3 processors, tasks τ_3, τ_4 and τ_1 are trivially schedulable ($C_i < T_i$) on one processor each

However, for task τ_2 we need to calculate the response time!

Exercise #7 – blackboard scribble

$$R_i^{n+1} = C_i + \frac{1}{m} \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i^n}{T_j} \right\rceil C_j + C_j \right)$$

Task	C_i	T_i	U_i
τ_1	1	7	0,14
τ_2	4	19	0,21
τ_3	9	20	0,45
τ_4	11	22	0,5

$$R_2^0 = C_2 = 4$$

$$R_2^1 = C_2 + \frac{1}{m}\left(\left(\left\lceil \frac{R_2^0}{T_3} \right\rceil C_3 + C_3\right) + \left(\left\lceil \frac{R_2^0}{T_4} \right\rceil \cdot C_4 + C_4\right) + \left(\left\lceil \frac{R_2^0}{T_1} \right\rceil \cdot C_1 + C_1\right)\right) =$$

$$4 + \frac{1}{3}\left(\left(\left\lceil \frac{4}{20} \right\rceil \cdot 9 + 9\right) + \left(\left\lceil \frac{4}{22} \right\rceil \cdot 11 + 11\right) + \left(\left\lceil \frac{4}{7} \right\rceil \cdot 1 + 1\right)\right) = 4 + \frac{1}{3}(18+22+2) = 4 + \frac{42}{3} = 18$$

$$R_2^2 = 4 + \frac{1}{3}\left(\left(\left\lceil \frac{18}{20} \right\rceil \cdot 9 + 9\right) + \left(\left\lceil \frac{18}{22} \right\rceil \cdot 11 + 11\right) + \left(\left\lceil \frac{18}{7} \right\rceil \cdot 1 + 1\right)\right) = 4 + \frac{1}{3}(18+22+4) = 4 + \frac{44}{3} = 18{,}67$$

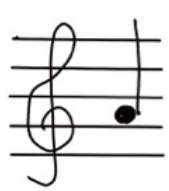
$$R_2^3 = 4 + \frac{1}{3}\left(\left(\left\lceil \frac{18{,}67}{20} \right\rceil \cdot 9 + 9\right) + \left(\left\lceil \frac{18{,}67}{22} \right\rceil \cdot 11 + 11\right) + \left(\left\lceil \frac{18{,}67}{7} \right\rceil \cdot 1 + 1\right)\right) = 4 + \frac{1}{3}(18+22+4) = 4 + \frac{44}{3} = 18{,}67$$

Since $R_2^3 \leq T_2 = 19$ task τ_2 is also schedalable

Special #1 – blackboard scribble

Tone: frequency + waveform + envelope (ADSR)

Note: "name" + length + tuning frequency



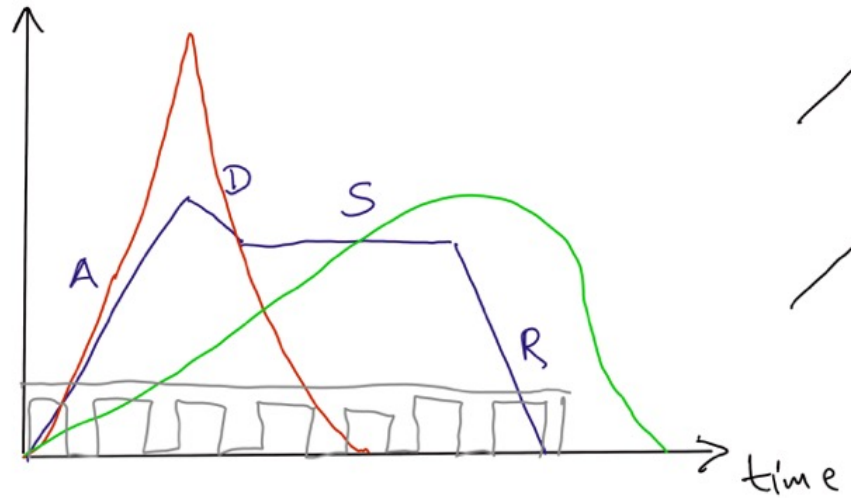
"A" quarter note 440 Hz
(1 note/beat)

f_0

$f_0 \; \frac{1}{3} 3f_0 \; \frac{1}{5} 5f_0 \; \cdots$

$f_0 \; \frac{1}{2} 2f_0 \; \frac{1}{3} 3f_0 \; \frac{1}{4} 4f_0 \cdots$

$f_0 \; \frac{1}{9} 3f_0 \; \frac{1}{25} 5f_0 \; \cdots$



Special #1 – blackboard scribble

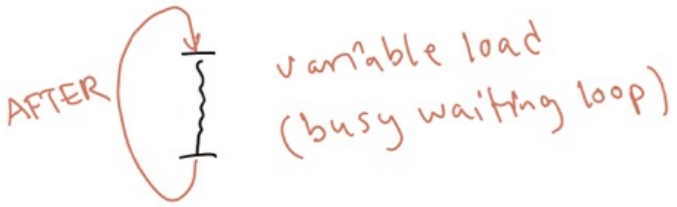
Tone generator

"0" = value 0

"1" = value 1-5

$T = \frac{1}{f}$ $f = 1\,kHz \Rightarrow T = 1\,ms$

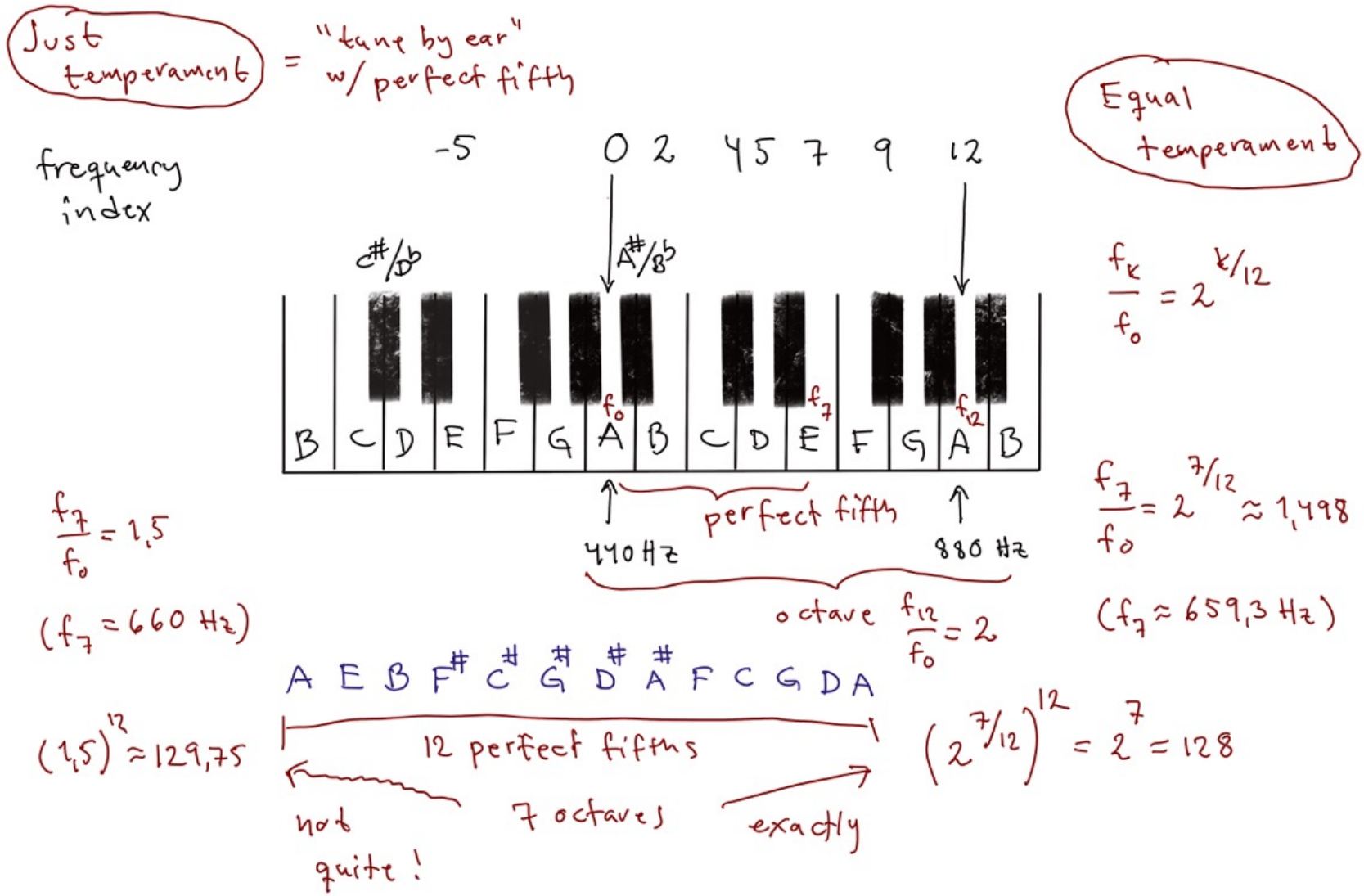
Background load



no deadlines ⇓ both tasks equally important

with deadlines ⇓ prioritize tone generator

Special #1 – blackboard scribble



Special #2 – blackboard scribble

Tone: frequency + waveform + envelope (ADSR)

Note: "name" + length + tuning frequency

Tempo: beats per minute (bpm)

120–150 bpm (techno)

> 300 bpm (speed metal, bebop jazz)

Beat: basic unit of time in a melody ("the pulse")

Measure/bar: a segment in a melody with a certain number of beats

Example time signatures:

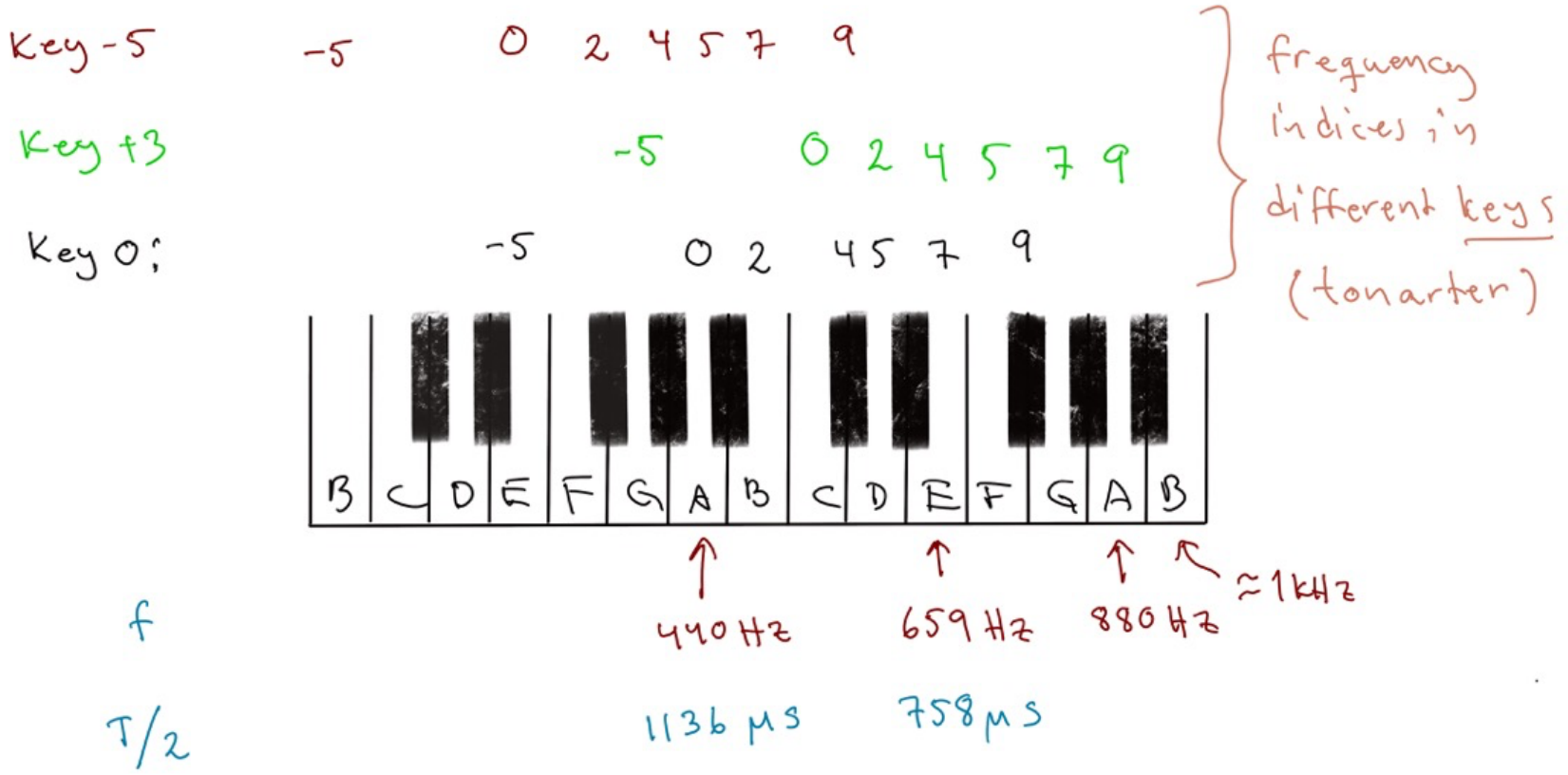
4/4 = 4 beats in a measure (ex. most music)

3/4 = 3 beats per measure (ex. waltz)

5/4 = 5 beats per measure (ex. Mission Impossible, Take 5)

7/4 = 7 beats per measure (ex. "Money" Pink Floyd)

Special #2 – blackboard scribble



Special #2 – blackboard scribble

Lab assignment:

Default tempo = 120 bpm

[60 ... 240] bpm

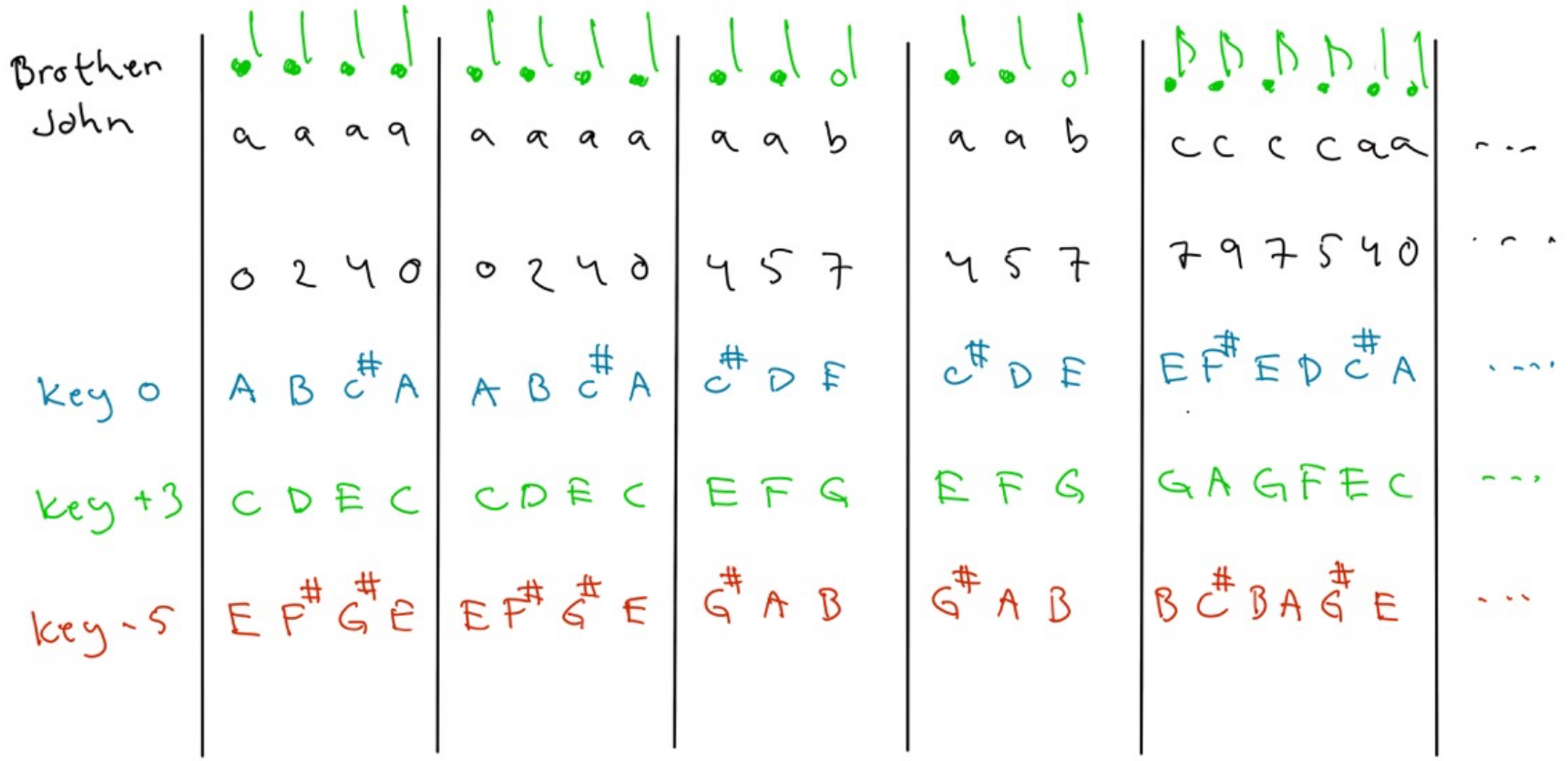
a = ♩ : quarter note (fjärdedelsnot) (= 1 beat)

b = 𝅗𝅥 : half note (halvnot) (= 2 beats) = 2a

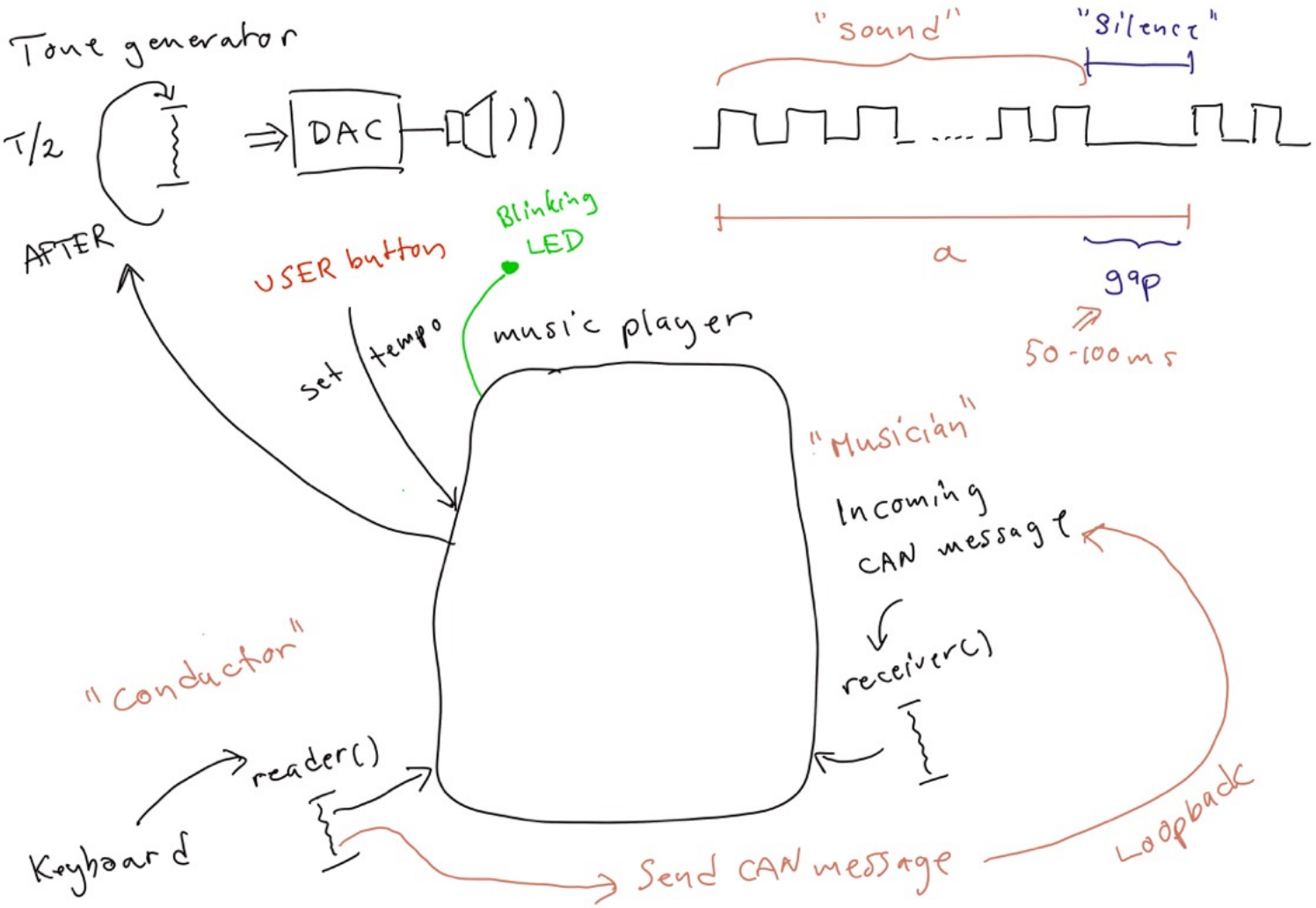
c = ♪ : eighth note (åttondelsnot) (= ½ beat) = ½ a

For 120 bpm ⇒ a = 0,5 s

Special #2 – blackboard scribble



Special #2 – blackboard scribble



Special #3 – blackboard scribble

2. Draw timing diagram in the interval [0,60]

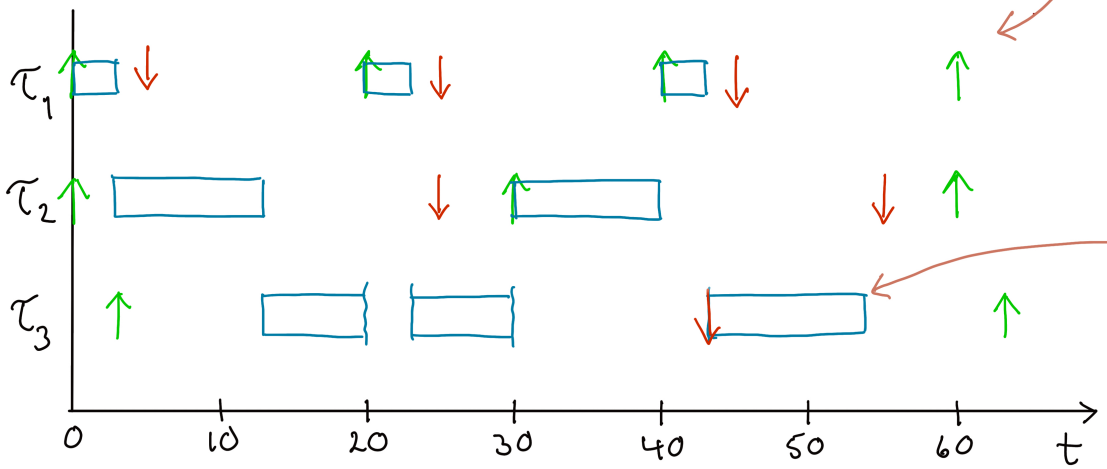
a) Simulate DM scheduling

Assume $O_1 = O_2 = 0$ and $O_3 = 3$

	Task	C_i	D_i	T_i
H	τ_1	3	5	20
M	τ_2	10	25	30
L	τ_3	25	40	60

All task executions must be completed regardless of whether the task misses its deadline or not

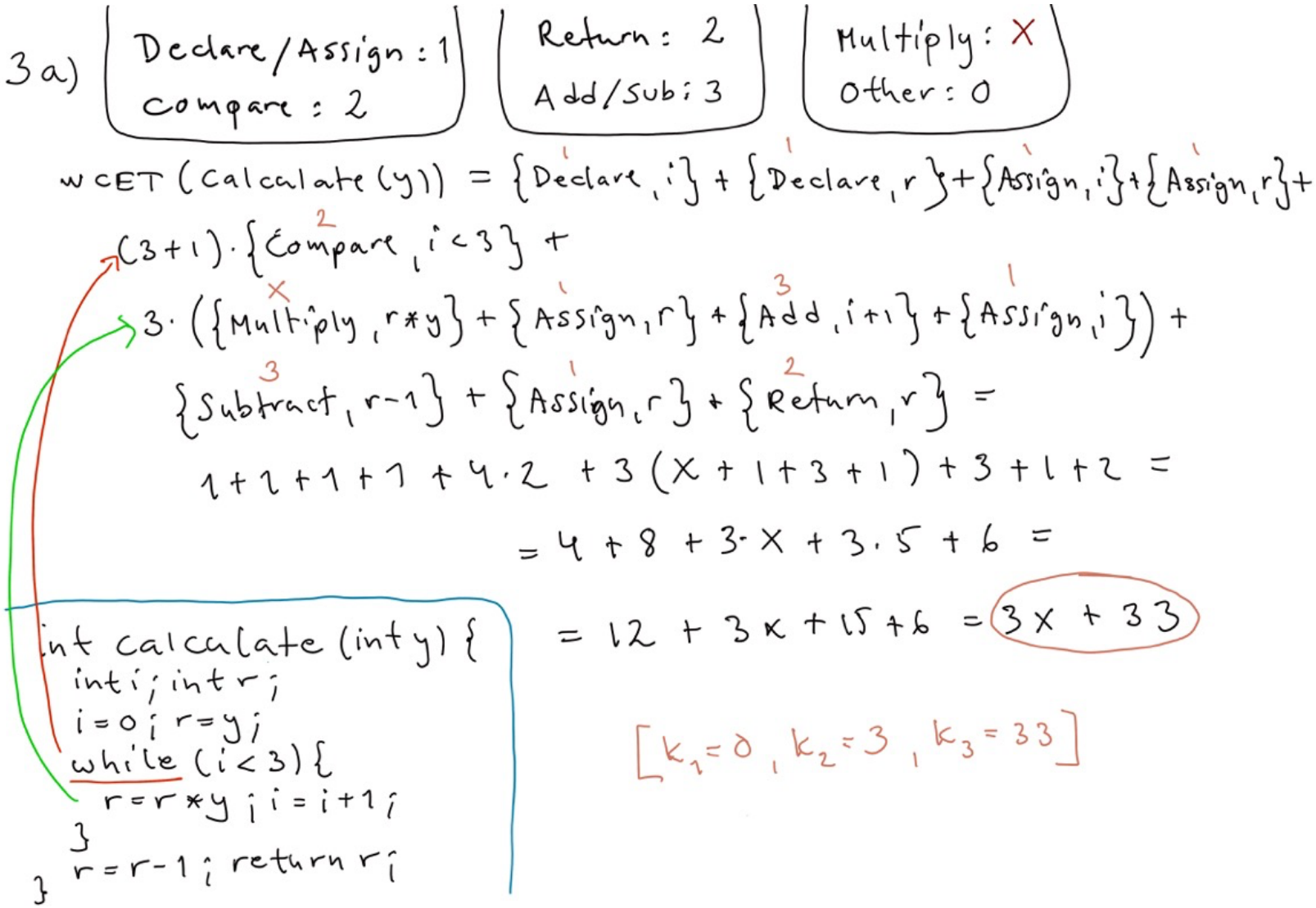
Note: arrivals and deadlines must be drawn in the diagram (as up- and down-arrows)



This execution must not be forgotten, despite the missed deadline at t = 43!

b) Timetable: $(\tau_1,0,3)(\tau_2,3,13)(\tau_3,13,20)(\tau_1,20,23)(\tau_3,23,30)(\tau_2,30,40)(\tau_1,40,43)(\tau_3,43,54)$

Special #3 – blackboard scribble



Special #3 – blackboard scribble

3b) Requirement on WCET:

$$0 \leq \text{"Estimated WCET"} - \text{"Real WCET"} < \varepsilon$$

↑ Pessimistic

to make sure assumptions made in schedulability analysis also apply at run-time

↑ Tight

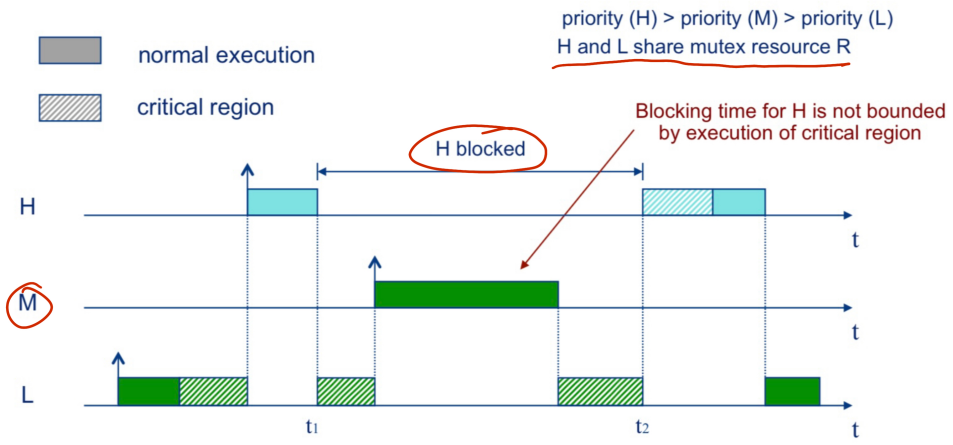
to avoid unnecessary pessimism in the schedulability analysis, which could cause feasibility tests to be too inaccurate to be useful

Special #3 – blackboard scribble

3c) Priority inversion: (assuming static task priorities)

In task sets with shared objects, when a task waiting for access to an object is blocked additional time by a task with lower priority that does not even use the object.

Example: (Lecture #6)



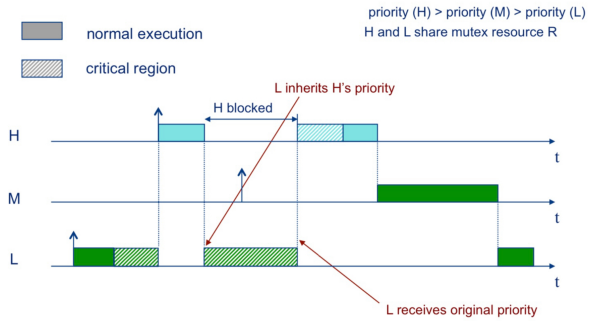
Special #3 – blackboard scribble

3 d) Priority inheritance protocol: (assuming static priorities)
when a task blocks one or more higher-priority tasks it temporarily assumes ("inherits") the highest priority of the blocked tasks

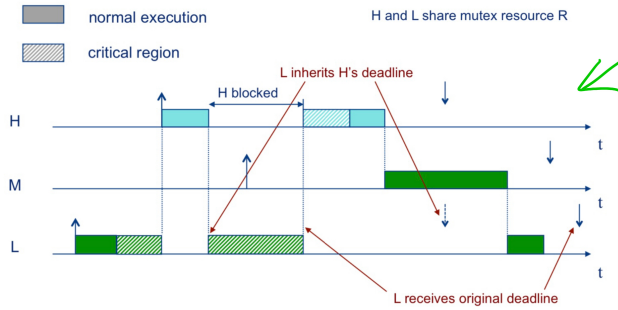
Deadline inheritance protocol: (assuming EDF priorities)
when a task blocks one or more tasks with deadlines earlier in time, it temporarily assumes ("inherits") the deadline earliest in time of the blocked tasks

(see Lecture #6)

PIP



DIP

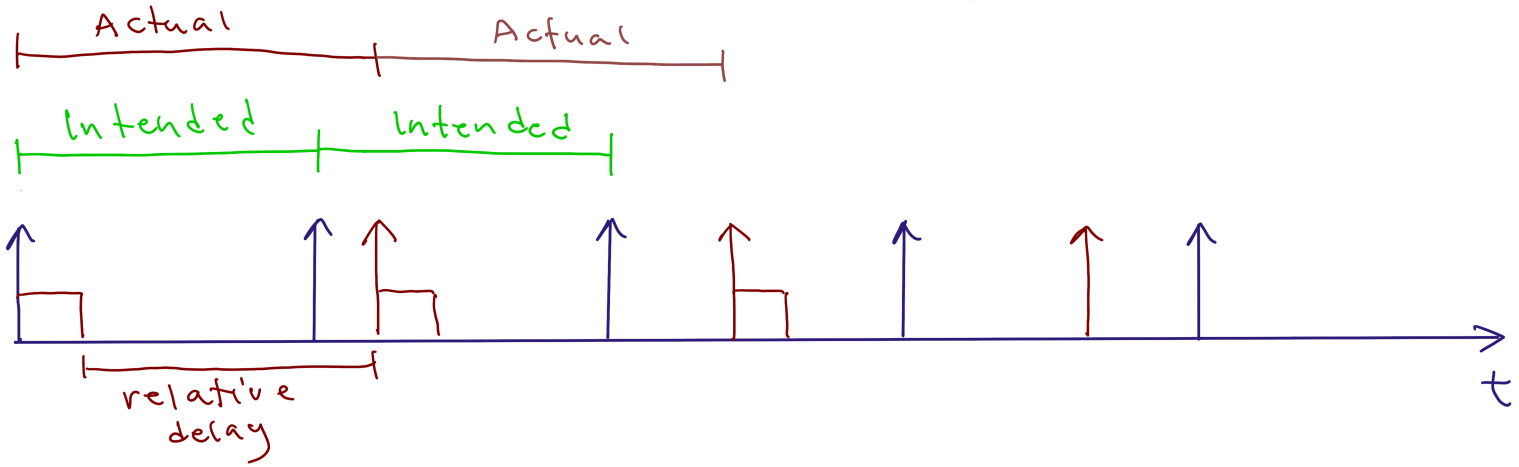


Special #3 – blackboard scribble

3 e) Systematic time skew:

When using a (relative) delay statement in the software to implement periodic executions the actual period becomes longer than the intended one.

Reason: the WCET of the periodically executed code will affect the period length

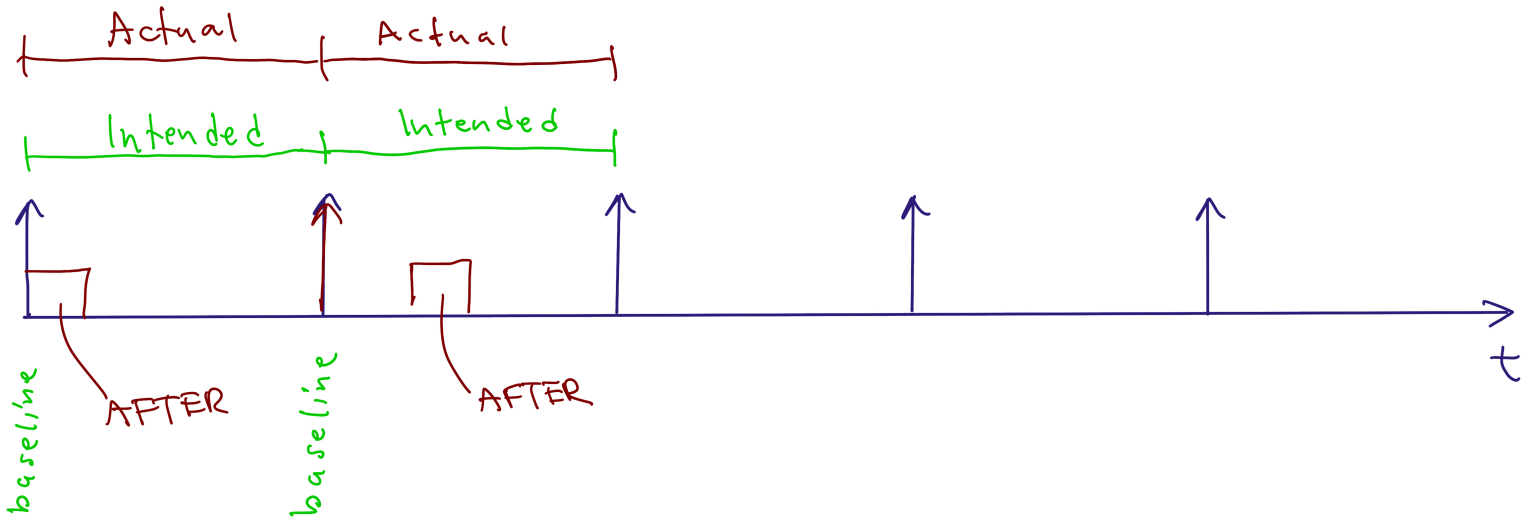


Special #3 – blackboard scribble

3 f) The AFTER operation in TinyTimber

Baseline: the earliest time for a method to start executing.

The AFTER operation adds a delayed earliest start time (offset) measured from the calling method's baseline.



Real-Time Systems — EDA223/DIT162

Example Written Exam – Basic Part

Examiner:

Professor Jan Jonsson, Department of Computer Science and Engineering

Responsible teacher:

Jan Jonsson, phone: 031–772 5220

Visits the exam room on two occasions.

Aids permitted during the exam:

Chalmers-approved calculator

NOTE: Electronic dictionaries may not be used.

Content:

The **Basic part** of the exam consists of 3 problems worth a total of 30 points.

The **Advanced part** of the exam consists of 4 problems worth a total of 30 points.

Grading policy:

Obtaining less than 24 points in the Basic part leads to a **Fail** grade.

If you obtain at least 24 points in the Basic part, then your grade is determined by the total number of points in the Basic and Advanced parts combined as follows:

24–35 points ⇒ **grade 3**	24–43 points ⇒ **grade G** (GU)
36–47 points ⇒ **grade 4**	
48–60 points ⇒ **grade 5**	44–60 points ⇒ **grade VG** (GU)

Results:

When the grading is completed overall result statistics, and a time and location for inspection, will be announced on the course home page. Individual results will be available in Ladok.

Language:

Your solutions should be written in English

IMPORTANT ISSUES

1. Use separate sheets for each answered problem, and mark each sheet with the problem number.
2. Justify all answers. Lack of justification can lead to loss of credit even if the answer is correct.
3. Explain all calculations thoroughly. If justification and method is correct then simple calculation mistakes do not necessarily lead to loss of credit.
4. If some assumptions in a problem are missing or you consider that the made assumptions are unclear, then please state explicitly which assumptions you make in order to find a solution.
5. Write clearly! If we cannot read your solution, we will assume that it is wrong.

Good Luck!

Basic part

PROBLEM 1

The following sub-problem concerns processor-utilization analysis.

Consider a real-time system with three periodic tasks and a run-time system that employs preemptive single-processor scheduling using the rate-monotonic (RM) priority-assignment policy. The table below shows C_i (WCET), D_i (deadline) and T_i (period) for each task τ_i. All tasks arrive at $t = 0$.

	C_i	D_i	T_i
τ_1	3	20	20
τ_2	10	30	30
τ_3	25	60	60

a) Perform Liu & Layland's utilization-based analysis for the given task set. (2 points)

The following sub-problem concerns response-time analysis.

Consider a real-time system with three periodic tasks and a run-time system that employs preemptive single-processor scheduling using the deadline-monotonic (DM) priority-assignment policy. The table below shows C_i (WCET), D_i (deadline) and T_i (period) for each task τ_i. All tasks arrive at $t = 0$.

	C_i	D_i	T_i
τ_1	3	5	20
τ_2	10	25	30
τ_3	25	40	60

b) Perform response-time analysis for the given task set. The final (converged) response time should be calculated for each task in the task set, regardless of whether the response-time analysis for that task fails or not. (4 points)

PROBLEM 1 (cont'd)

The following sub-problems concern processor-demand analysis.

Consider a real-time system with three periodic tasks and a run-time system that employs preemptive single-processor scheduling using the earliest-deadline-first (EDF) priority-assignment policy. The table below shows C_i (WCET), D_i (deadline) and T_i (period) for each task τ_i. All tasks arrive at $t = 0$.

	C_i	D_i	T_i
τ_1	3	5	20
τ_2	10	25	30
τ_3	25	40	60

c) Show that the largest interval to examine is $L_{\max} = 60$ for the given task set. The solution should include calculations of the hyper-period upper bound as well as the upper bound proposed by Baruah, Rosier and Howell. (2 points)

d) Calculate the complete set of control points within $L_{\max} = 60$ for the given task set. (2 points)

e) Perform processor-demand analysis for the given task set, and the complete set of control points calculated in sub-problem d). The analysis should be performed for every control point in the set, regardless of whether the analysis in another control point fails or not. (4 points)

PROBLEM 2

The following sub-problems concern hyper-period analysis.

Consider a real-time system with three periodic tasks and a run-time system that employs preemptive single-processor scheduling. The table below shows O_i (offset), C_i (WCET), D_i (deadline) and T_i (period) for each task τ_i.

	O_i	C_i	D_i	T_i
τ_1	0	3	5	20
τ_2	0	10	25	30
τ_3	3	25	40	60

a) Simulate the execution of the task set using the deadline-monotonic (DM) priority-assignment policy, and construct a timing diagram that spans the entire hyper period. The arrival and deadline of each task instance should be clearly indicated in the diagram in the form of an up-arrow (arrival) and down-arrow (deadline), respectively. All task executions must be completed, regardless of whether a task instance misses its deadline or not. (5 points)

b) Derive a time table, corresponding to the timing diagram constructed in sub-problem a), that clearly states the start and stop times for each task instance (or task segment, if a task instance is preempted). (1 point)

PROBLEM 3

The following sub-problems concern analysis of worst-case execution time (WCET).

Consider the following C program code for function `calculate`:

```
int calculate(int y) {
   int i;
   int r;
   i = 0;
   r = y;
   while (i < 3) {
      r = r * y;
      i = i + 1;
   }
   r = r - 1;
   return r;
}
```

Assume the following costs for the language statements:

- Each declaration and assignment statement costs 1 μs to execute.
- Each evaluation of the logical condition in an `if`- or `while`-statement costs 2 μs.
- Each add and subtract operation costs 3 μs.
- Each multiply operation costs X μs.
- Each return statement costs 2 μs.
- All other language constructs can be assumed to take 0 μs to execute.

a) Derive the WCET of function `calculate` by using the analysis method proposed by Shaw. The WCET should be expressed in the form $\mathbf{k_1 \cdot y + k_2 \cdot X + k_3}$, where k_1, k_2 and k_3 are integer constants, y is the function parameter, and X is the cost for the multiply operation. (2 points)

b) Explain why is it preferred that WCET estimates for tasks in a real-time system are *pessimistic* as well as *tight*. (2 points)

The following sub-problems concern scheduling concepts.

c) Describe the meaning of *priority inversion*. (2 points)

d) State the major difference between the *priority inheritance protocol* (PIP) and the *deadline inheritance protocol* (DIP). (1 point)

The following sub-problems concern real-time programming.

e) Describe the meaning of *systematic time skew* in the context of implementation of periodic task executions. (2 points)

f) State the prominent feature of the `AFTER()` operation in TinyTimber that facilitates implementation of periodic tasks without systematic time skew. (1 point)
