Generic Embedded Computer

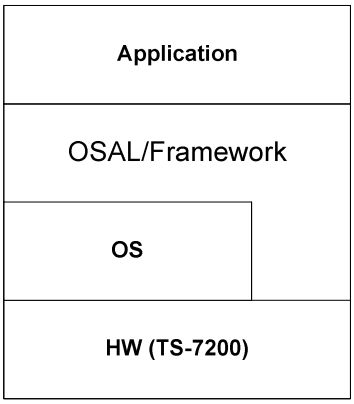
Course: Grundlæggende Indlejrede Systemer (GiS)
Modul 2, Multi programming and Real-Time systems
Group: Group 3
Date: Juni 09
Names: Thomas Ginnerup Kristensen
Henrik Arentoft Dammand
Anders Hvidgaard Poder

1. Introduction

Through several projects we have experienced that a generic HW platform including basic peripherals, is suitable for a lot of different control and regulation applications.
For that reason we want to develop a basic embedded computer solution, which we will offer our customers as a cheap, flexible and very "time-to-market" friendly platform.

This basic embedded computer solution shall include both a generic HW board and a SW framework, which will enable us to have a given prototype running for demonstration in a very short time period.

The solution must of course supply the most basic functionality used in an embedded computer. Moreover the solution must be extendable with new HW interfaces, such as new communication media, e.g. CAN bus, GSM modem etc.



The system layers

1.1. HW Platform

The HW candidate for the project is the TS-7200 SBC from Technologic systems.

The TS-7200 is a full featured SBC based on the ARM9 CPU and provides a standard set of on-board peripherals, including:

- 200 MHz ARM9 processor with MMU
- 32 MB of High Speed SDRAM
- PC/104 expansion bus
- 2 COM ports
- 10/100 Ethernet interface
- Compact Flash Card interface
- Watchdog timer, SPI bus

The TS-7200 is a cheap and flexible solution and includes the basics we are looking for. It includes the PC/104 expansion bus, which makes it highly flexible regarding extensions.
Moreover Technologic systems supplies a wide range of extension boards to be used in conjunction with the TS-7200, which also makes is very interesting in a business point of view.

The TS-7200 is shipped with a Linux OS, TS-Linux – installed on the internal flash. The TS-Kernel is based on the 2.4.26 Linux Kernel.
This Linux release does not support Real Time functionality, however modifications has been made so that it is possible to enable RT capabilities on the TS-Linux via RTAI. Some configuration is needed to enable this.

1.2. SW Platform

The framework shall be developed in C++ and encapsulate the rather verbose C API to the POSIX compliant OS interface.

Both commercial and open source SW frameworks are available and could be used instead of developing our own. Boost and POCO are examples of such open source frameworks.However, the learning curve of using these frameworks is quite steep and they include much functionality not needed for the target of our framework.

An effective framework with a simple OO interface, which supplies the basic building blocks for an RT application, will enable the user to make robust programs where focus is on the application and not the API.

1.3. Objectives

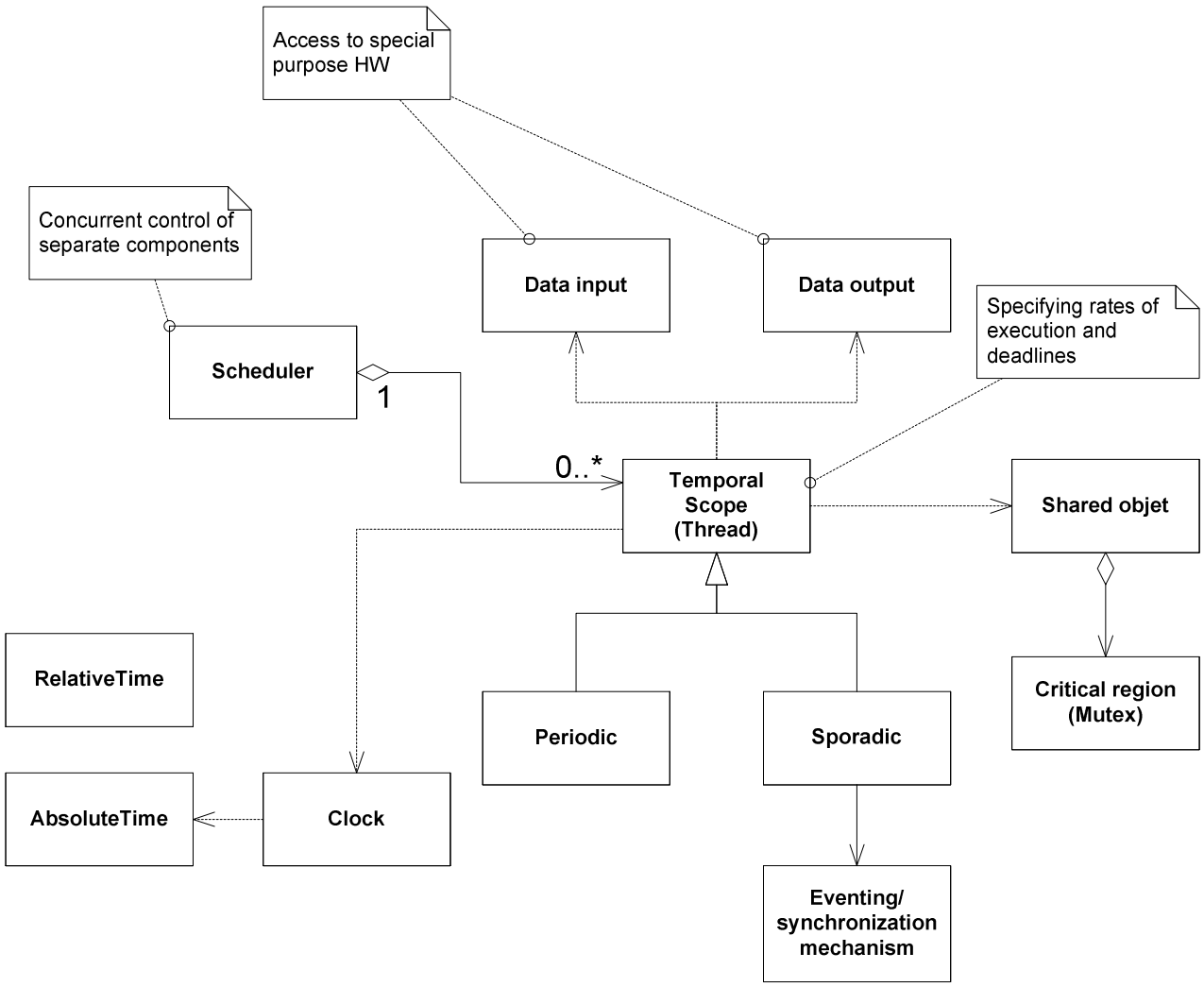
The objectives of the project are to:

- Design the SW framework and select a subset of the POSIX API to be used in the SW framework.
- Apply real-time capabilities to the OS
- Verify and test the real-time performance of the OS run on the target HW.

2. Problem domain

Since the platform must provide the basic building blocks for an RTS computer application, the problem domain analysis will focus on the characteristics of a RTS. The system must provide means for addressing and handling these characteristics.

Problem definition



Logical view of the problem domain.

The problem domain consists of a number of objects which is identified by the logical class view given above. The individual objects are described shortly in the list below.

- Periodic — This temporal scope represents a thread that is scheduled in periodic intervals dictated by the system (usage). When the application has created a periodic object it must hereafter wait to be launched for execution initiated by the application. When it is launched the scheduler controls when the thread is running or waiting determined by the interval.
- Sporadic — This temporal scope represents a thread that is dependent on events. The sporadic thread is very much like the periodic thread except that it is controlled by the use of events (external/internal).
- Temporal Scope — Base class of periodic and sporadic threads.
- Clock — The clock class provide facilities to handle and measure time.
- Time — The Time class abstracts the concept of time related to the timeframe of the physical environment it controls. Both a relative and absolute time representation is supported.
- Mutex — The Mutex provides mutual exclusion of access to shared data regions (critical regions).
- Scheduler — The scheduler is provided by the underlying OS and supports Fixed-Priority Scheduling (FPS) with pre-emption. The scheduler dictates that each thread must be given a unique priority. It is the application programmer's responsibility to supply this priority. This is done as a construction parameter for the Thread class.
- Data- input/output — These classes represents interface for the I/O and special purpose HW, e.g. GPIO, UARTs etc. A specific class is created for each interface and acts as a hardware abstraction layer.

In order to ensure that the objects identified include the central components for supporting real-time applications, verification is performed by comparing the components with the RT Java API.

Identified object	**RT Java counterpart**
Periodic thread	RealtimeThread used with instances of PriorityParameters and PeriodicParameters as construction parameters
Sporadic thread	RealtimeThread used with instances of PriorityParameters and SporadicParameters as construction parameters
Thread (Temporal scope)	Thread
Clock	Clock
AbsoluteTime	RelativeTime
RelativeTime	AbsolutTime
Mutex	MonitorControl (PriorityCeilingEmulation)
Scheduler	PriorityScheduler
Data input/output	N/A

As can be seen from the list, the central components (types) are supported by the framework. A major difference between POSIX/C++ and Java is the use of stack memory. Java allocates all

objects on the heap, hence only functions calls, references and primitive types is pushed onto the stack. POSIX/C++ support automatic objects created on the stack; hence POSIX/C++ is more stack memory intensive. This use is of course dependant on the application. Therefore the Thread class is created with the stack size as construction parameter. This makes it possible for the application programmer to individually state the stack size needed for each instance.

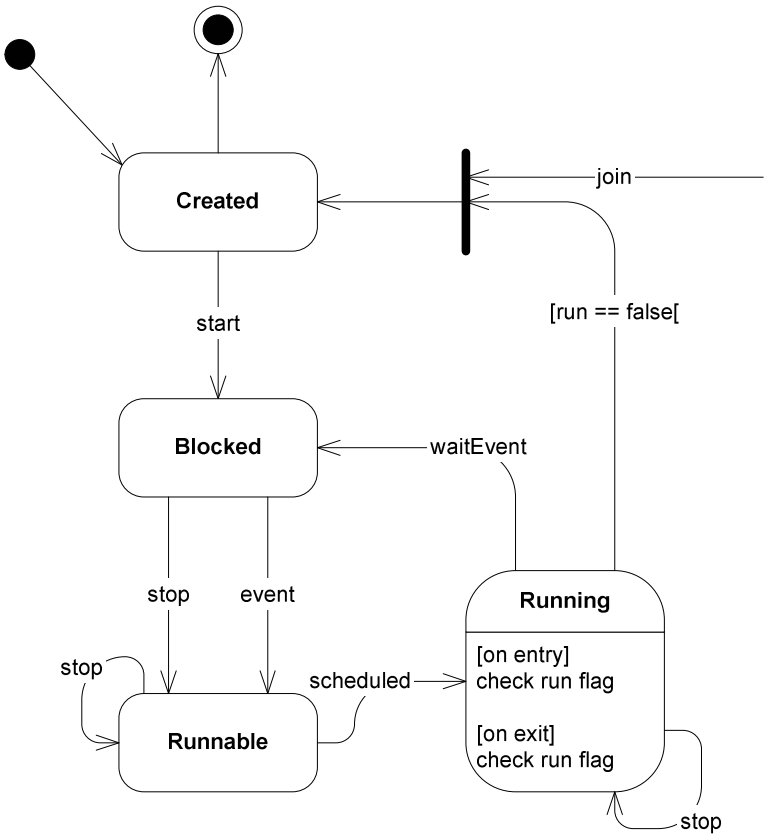
2.1. Object states

2.1.1. Active objects:

In the problem domain the temporal scopes within the system is represented by threads, the active objects. A thread can be either periodic or sporadic, and it represents the timing requirements of the end application.

The application programmer must be able to specify the release parameter for the thread and supply the algorithm to be executed when released.

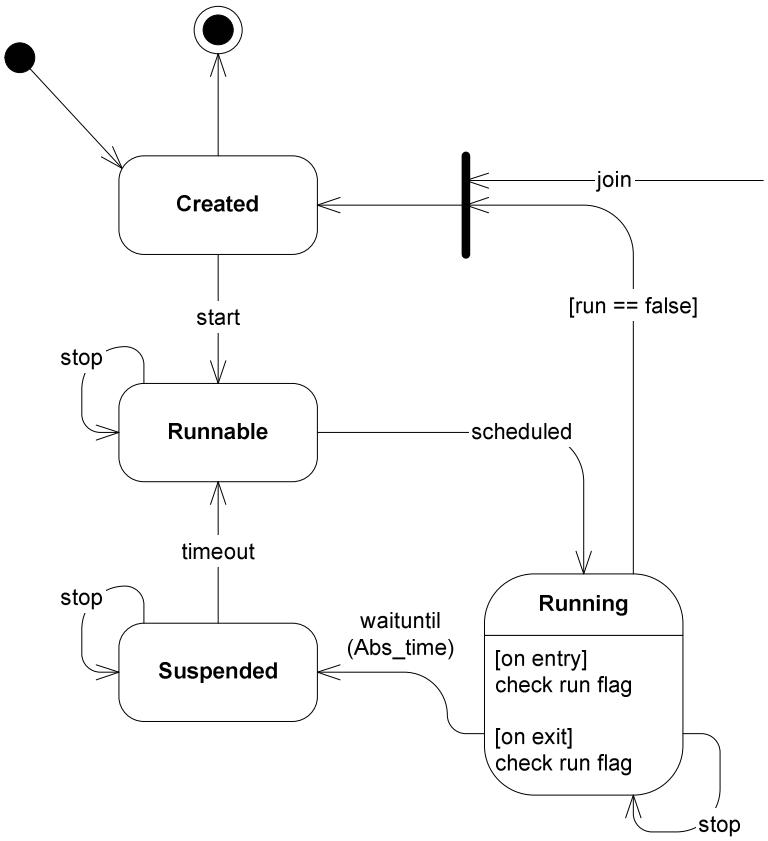
Sporadic Thread:



The sporadic thread is used to handle non-periodic events. During normal operation, the sporadic thread steps through the 3 states; Runnable, Running and Blocked. When the thread is created it enters the Blocked state waiting for the event. When the event occurs the thread becomes Runnable and it is now up to the scheduler to make the thread enter the Running state.

When the Sporadic algorithm, provided by the application programmer, has ended, the thread enters the Blocked state again waiting for the event.
Each time the thread enters the running state it inspects the internal run flag to determine if the thread should return. If not, it executes the algorithm supplied/implemented by the application programmer. When the algorithm is ended it again inspects the running flag. If the flag is not set it waits for the event and thereby enters the Blocked state.
Stop can be called in any of the normal operating states. If stop is called in the Blocked state (most likely) it sets the run flag to false and releases the thread (creating an event). In that way the thread becomes runnable and will, on entry to the Running state, return to the created state.
The client can synchronize with the thread by calling join, which is a blocking call. I.e. when join returns, the thread has returned.

Periodic thread:



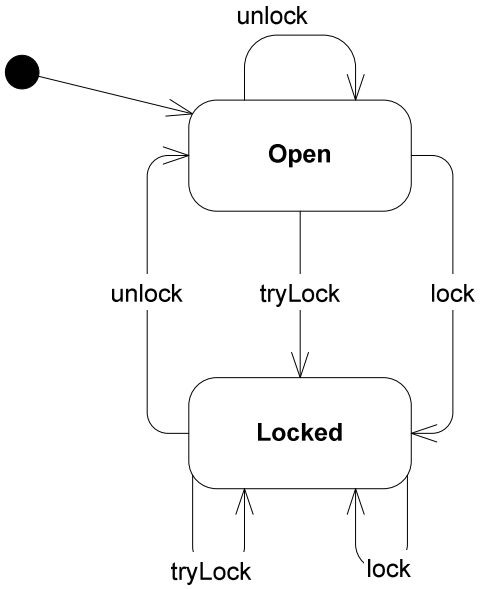
Like the Sporadic thread, the periodic thread switches between 3 states during normal operation; Runnable, Suspended and Running.
When the thread is started it becomes Runnable, hereafter the scheduler runs the thread, making the thread run the periodic algorithm supplied by the application programmer.
After performing the periodic algorithm the thread suspends itself until the next period.
When the next period is reached the thread becomes Runnable again waiting to be scheduled.
A call to stop will set the internal run flag to false and the thread will return on the next entry into the Running state.

When the thread is created it contains the period and the threads priority, which both are supplied by the creator of the object.

2.1.2. Synchronization objects:

Since the threads in the system often cooperate to perform the overall task of the system, synchronization and shared data areas are used. These areas are critical regions and must be protected with mutual exclusion.

Mutex:



The Mutex has 2 states; open and locked. The Mutex is used to protect critical regions, i.e. shared data areas. When a thread wishes to enter a critical region it must obtain the entry lock. When a thread gets the lock, the Mutex enters the Locked state. When the thread exits the critical the lock is released and the Mutex enters the Open state.

The Mutex lock is obtained by calling the lock method on the Mutex. If the state is Open the thread gets the lock and can continue into the critical region. When the thread excites from the critical region it must call unlock to relinquish the lock and the state changes back to Open. If the state is locked when the thread calls lock to get access, the thread is blocked until the Mutex re-enters the Open state (initiated by the thread holding the lock). The tryLock is used when the thread wants to wait for the Mutex to become available for a specified duration, i.e. if the thread calls tryLock, with a timeout of 0, when the Mutex has been locked by another thread (Locked state), it returns immediately indicating that the lock was not obtained.

Clock and Time:

The Clock and Time classes do not contain logical states and hence not described here.

2.2. The execution environment:

In order to get real-time support, each thread must be scheduled in a deterministic manner so it is possible for the application programmer to perform response time analysis of the system.
Using Fixed-Priority Scheduling (FPS) with pre-emption supports this determinism and the OS must be configured to support this.

Since the scheduling must be based on a static priority, there must be means for the application programmer to specify the priority for each thread – this is the scheduling parameter dictated by the platform.

An advantage of using the FPS is that it is straight forward to perform the response time analysis once the period, cost and deadline of all the processes is established.
Also, using the rate monotonic priority assignment, the relative priorities between the threads in the system is given once the application programmer has elaborated the periods.

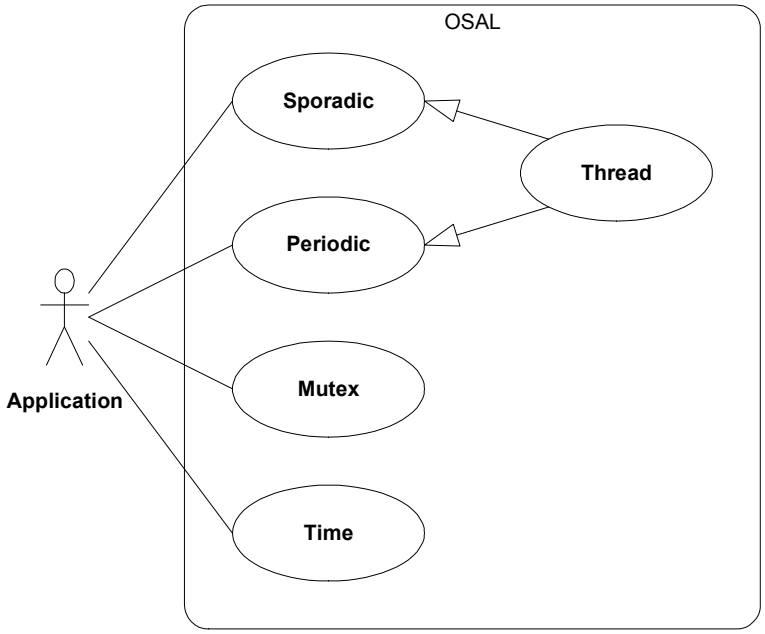
The applications will most possibly exists of both sporadic and periodic threads. To support sporadic threads and still make it possible to perform the response time analysis, the sporadic threads are given a worst case period (the shortest period), and are then treated as periodic threads.

3. Application domain analysis

The objective of this analysis is to establish the API needed to support the basic building blocks for the embedded application found in the Problem Domain analysis.

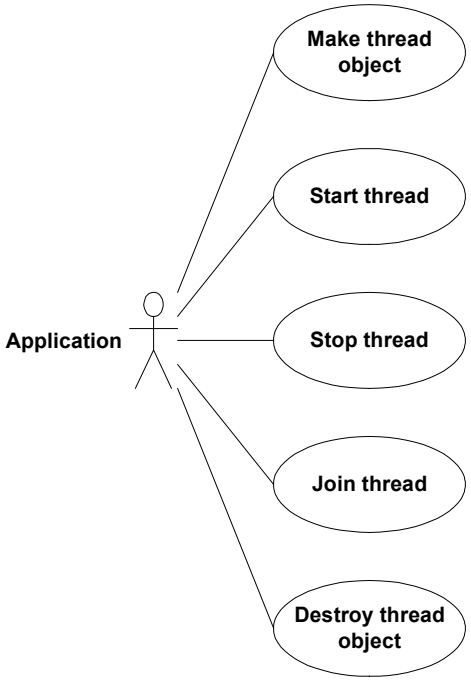
3.1. Use cases

Basically, the OS abstraction layer (OSAL) provides some facilities to support threads, mutexs and time functionality. The OSAL hides the complexity of the POSIX interface and minimizes the effort of creating threads and thread synchronization. The actor is the user application.



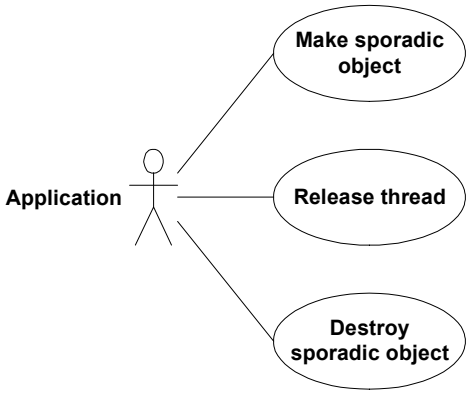
3.2. Thread

A thread is a targeted function that runs in a process and can run concurrently with two or more threads. The application should have access to some basic thread management facilities such as create/destroy threads, thread star/stop methods and thread prioritization etc.



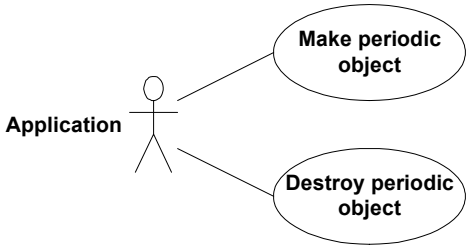
3.3. Sporadic

When application creates a mutex object it must be setup with several parameters such as priority, an event function targeted for execution and stack size. Release is a method for signalling the sporadic thread to execute the event function.



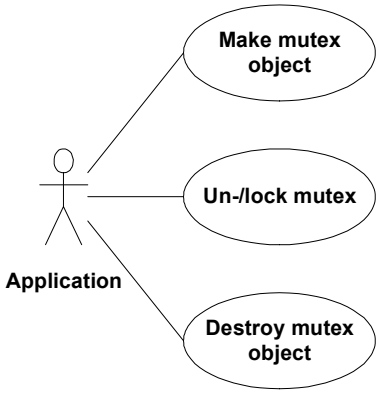
3.4. Periodic

A periodic thread is created with several parameters such as priority, a periodic function targeted for execution, stack size and a periodic time interval.



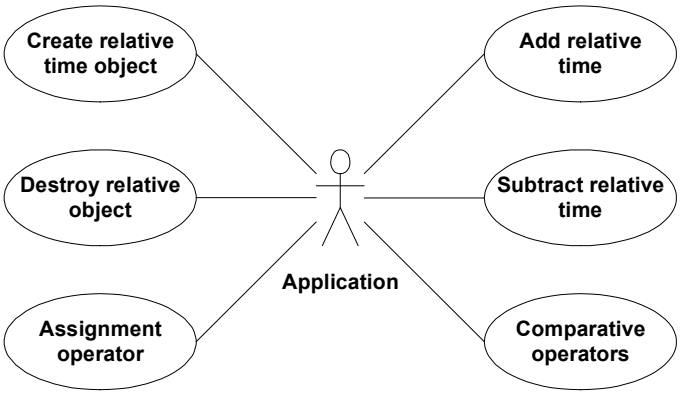
3.5. Mutex

A mutex object can be created with a parameter that gives the priority ceiling. The lock method will block the thread if it is held by another tread. If the same thread attempts to relock a mutex it will result in an exception. The unlock method unlocks the mutex so that it can be acquired by other threads.



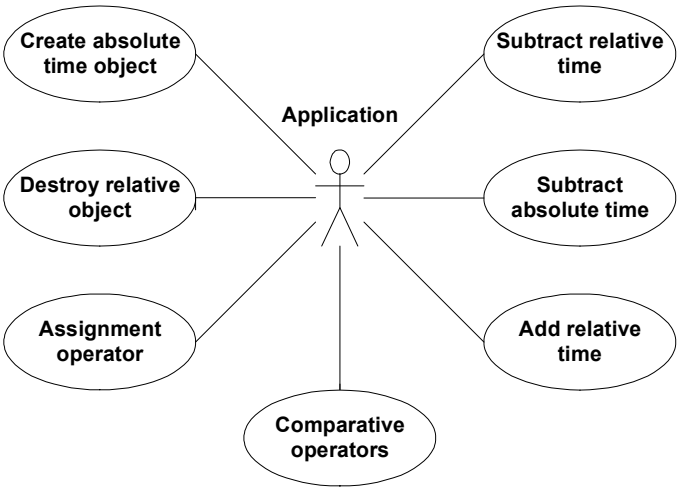
3.6. Relative time

A relative time object can be created in several ways. It can be created as an empty object or with a relative time value. Simple arithmetic's such as addition or subtraction are supported and relative times can be compared with each other.



3.7. Absolute time

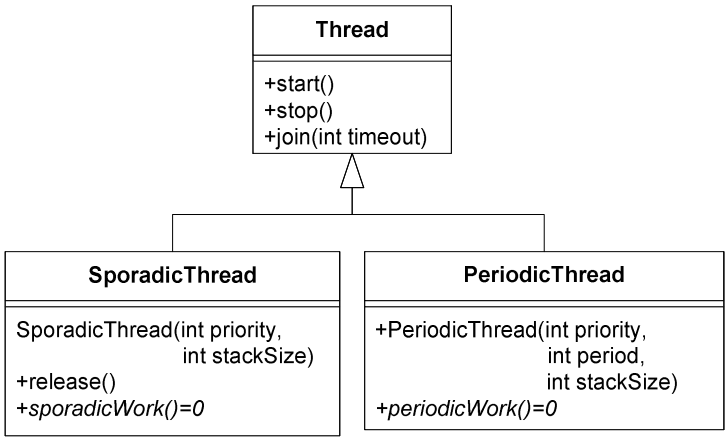
An absolute time object can be created in several ways. It can be created as an empty object or with an absolute time value. Simple arithmetic's such as addition or subtraction are supported and absolute times can be compared with each other.



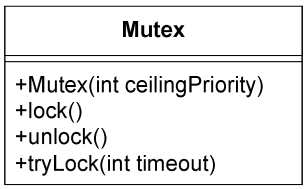
3.8. API

The API is a collection of interface classes with a set of public methods. These methods are identified on the basis of the state transitions for the individual objects. However some of the state transitions are not directly based on external events, and therefore do no result in a method on the given interface class.

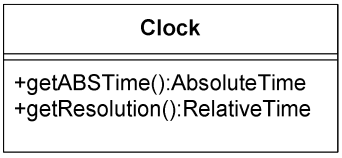
Sporadic and Periodic thread:



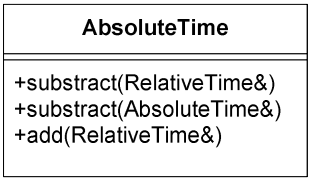
Mutex:

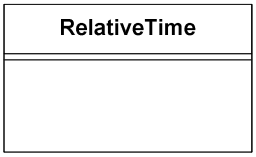


Clock:



Time:





Design

Her skal være noget om hvilke POSIX funktioner der benyttes.
Klasse diagrammer i framework.
Vi skulle også helst have noget schedulerings teori ind.

Hvad med konfigurering af Linux OS. Skal det være under implementations sektion.?

4. Accept test

4.1. Response time analysis

The platform developed during this project may be used to develop a real-time application. In order to validate the platform a fictive application will be described, analysed and then used to validate the platform (time permitting).

The platform works with sporadic and periodic threads, as well as timers and mutexes (critical regions). It uses pre-emptive fixed priority scheduling and priority inheritance via the immediate ceiling priority protocol. These subjects have been discussed elsewhere in the report and will not be further elaborated here.

4.1.1. Experimental test application

The experimental test application takes its basis in the set-up from Burns & Wellings figure 1.1, page 4. The set-up is elaborated with a sporadic task in the form of an on/off-button. This is shown in Figure 1.

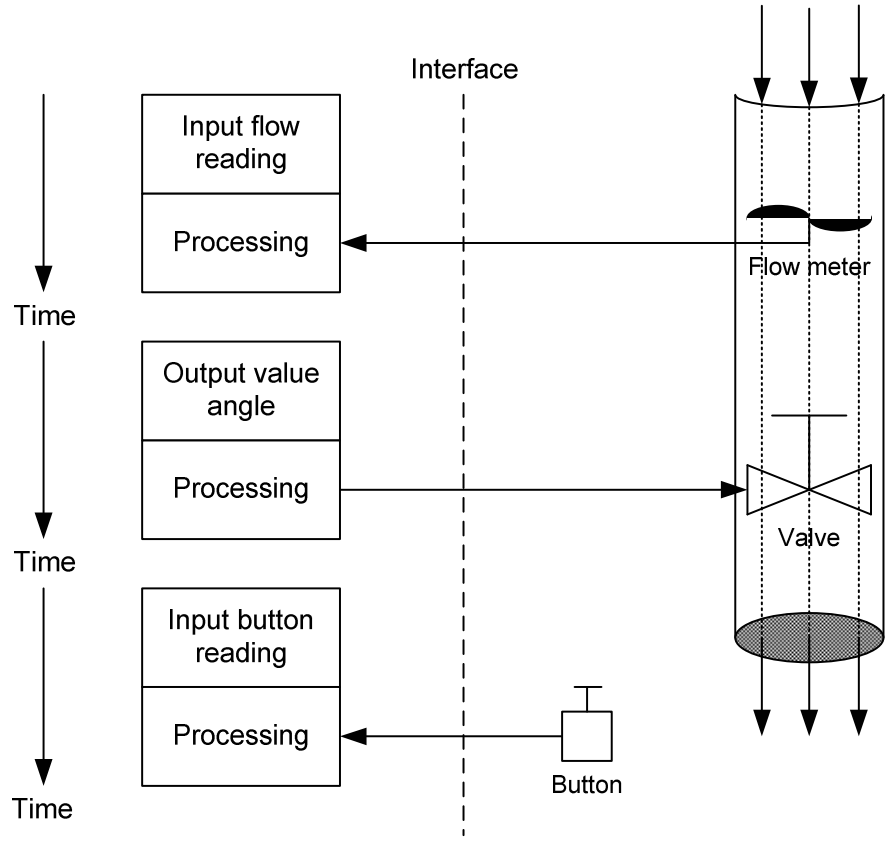


Figure 1: Experimental test set-up

The first challenge is the shared data between the input flow reading and the output value angle, which are both periodic processes. The second challenge is the sporadic process monitoring the button, and its control of the other processes. If we consider it an emergency-button then the sporadic process must be able to pre-empt the other processes.

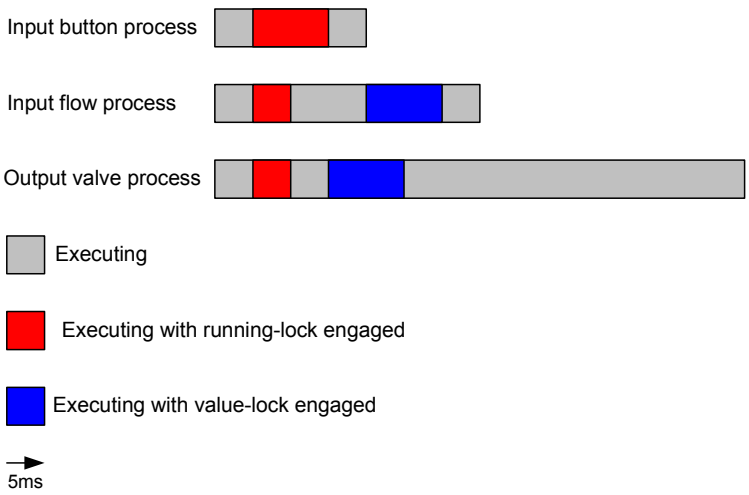
This, all in all, calls for two mutexes to protect the data shared between the input flow process and output valve process and the input button process and the other processes respectively. Naturally this is only true if the data is more than a simple primitive, as it can then be assigned and read atomically, requiring no protection. We consider the flow values and the button data to be multiple primitives or complex data structures.

Priority-wise we define the valve to always function on the newest data, thereby making the input flow reading the most important. The periods are simply chosen, as is the deadlines and execution cost (highest priority = highest number).

Process	Priority	Type	Period/min. interval (T)	Deadline (D)	Execution cost (C)
Input button reading process	3	Sporadic	50ms	30ms	20ms
Input flow reading process	2	Periodic	500ms	200ms	35ms
Output valve angle process	1	Periodic	500ms	200ms	70ms

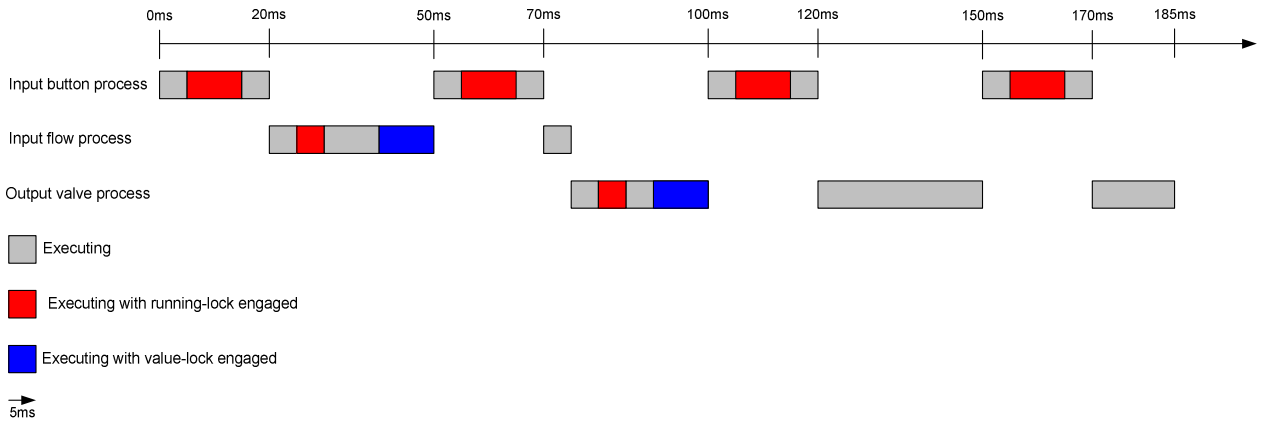
We could consider the release jitter. This is defined by the scheduler, which in this platform is defined by the operating system; TS-Linux. Though the exact value is unknown it is safe to say that it will be many orders of magnitude smaller than the other factors in the system. Lets set it at e.g. 10us. Given that only around 2 - 14 possible scheduling occurrences can happen in a full period, the maximum total disturbance is about 140us, which can safely be considered negligible.

We need to consider the execution cost of the three processes. As there are potential blocking involved we have to factor this in, but for now we focus on the raw machine instruction execution cost. By analysing the machine code for the three process we can set up the following colour-coded diagram for the execution cost.



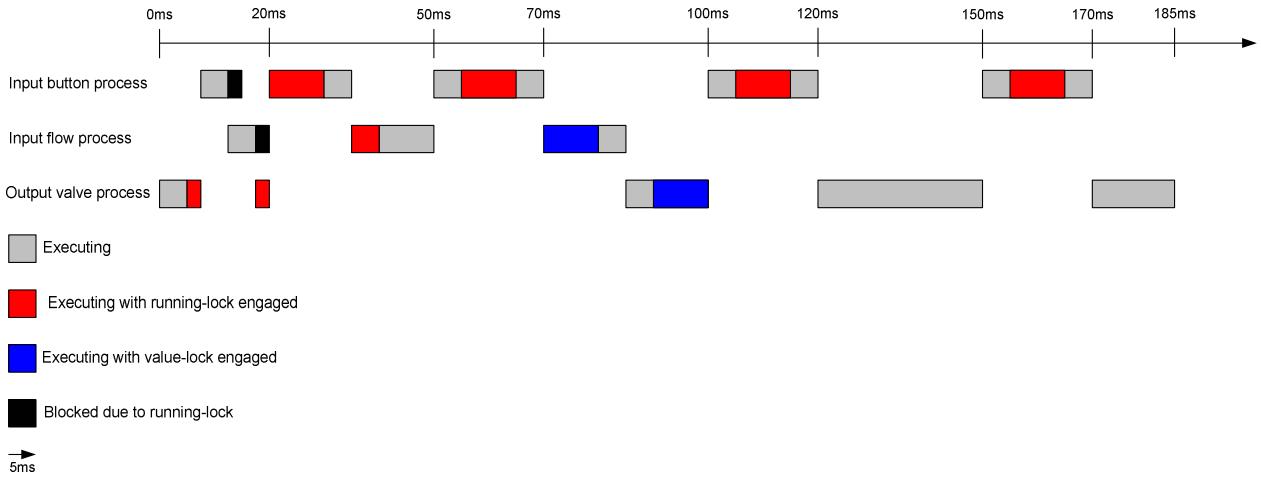
Based on this we can perform the response time analysis and determine if the system will always meet its deadlines.

Before we consider the mathematical solutions, we can do a graphical representation of the worst case without blocking, which is all processes being released simultaneous and the sporadic process continuously arriving at its minimum inter-arrival time. As the sporadic process is a button it is naturally not logical that it will arrive in this manor, yet this fact is irrelevant for this analysis.



Process	Time from initiation until completion
Input button reading process	20ms
Input flow reading process	75ms
Output valve angle process	185ms

Unfortunately this is not necessarily the real worst-case (at least it might not be), as there is potential blocking from the lower-priority processes, as may be seen below. An important note here is that this example considers no priority inheritance and no ceiling protocol what so ever – otherwise the example would not be feasible. Please also not that here the starting points are offset from each other, and that this actually improves the response time of the input flow process, even though it suffers blocking; this shows the potential power of offset, yet unfortunately this has to be used very judiciously, as it is a strongly NP-hard problem choosing optimal offsets, and if used with a sporadic process it is an impossibility.



Process	Time from initiation until completion
Input button reading process	27,5ms
Input flow reading process	74,75ms
Output valve angle process	185ms

Blocking causes priority inversion, allowing multiple lower priority processes to block a higher priority processes. As we can see this causes the highest priority process to be delayed considerably.

So, how do we make sure we take all these possible scheduling-times into consideration? There are potentially an infinite amount of scheduling paths. Luckily there are mathematical formulas to help us with this.

$R_i = C_i + I_i + B_i$

R_i is the response time for process i.
C_i is the execution cost of process i.
I_i is the interference imposed by higher priority processes on process i.

B_i is the blocking from lower priority processes suffered by process i.

$I_i = sum_j(ceiling(R_i/T_j)*C_j)$

$B_i = max_k(usage(k,i)*C(k))$

The blocking is for ICCP only, other protocols have different blocking calculations. The blocking equations can be simplified verbally to: “The cost of the single largest critical region in a lower priority process, which is also shared by process i”.The reason this is true, is because the immediate ceiling priority protocol (unlike the original ceiling priority protocol) always assign the highest priority of anyone using the critical section whenever there is contention about the access to the critical section. This means that is a higher priority process is blocked by a lower priority process, the lower priority process will be boosted to the highest priority, meaning there is no way for any other process, using the critical section, to gain execution time, and therefore multi-process blocking is impossible. This is only true when all processes are assigned different priorities or if no round-robin is employed. If this was not the case then the low-priority process with the lock might be scheduled out by another process with equal priority, which could then take another lock. If the processes all have different priorities this is not the case; when the low priority process is boosted to the higher priority it is only because the higher priority process has attempted to take the lock, and therefore this process is effectively blocked and the boosted process is then the only runnable process with that priority.

To manually calculate I_i requires a little more mathematics. This is due to the fact that R_i appear on both sides of the equation, and it contains a sum. We therefore have to do the calculations using a special principle.

The equation

$W_i^{n+1} = C_i + B_i + sum_j(ceiling(W_i^n / T_j)*C_j)$

can be repeated for n going from 0 towards infinity. The calculations must be stopped when two identical results are reached for two consecutive values of n. W^0_i is calculated by simply setting the sum to 0. Naturally it is not possible for the highest priority process to suffer interference, so for that process $R_i = C_i + B_i$.

Based on these equations we can perform the calculations. We need to define a value of i for each of the processes, and here we can e.g. use the static process priority, as it is, and must be, unique to the process.

As the highest priority process share only one critical section with lower priority processes, and the maximum length of this critical section in the lower priority processes are 5ms, we can determine B_3 as 5ms. And naturally there cannot be any interference from other processes, as they are all lower priority.

$R_3 = 20\ (C_3) + 0\ (I_3) + 5\ (B_3) = 25ms.$

For the second highest priority it is a little more complicated. The potential blocking is again the longest critical section in a lower priority process also used by this process. Here the value-lock is

shared, and potential blocking is then 10ms. With this we can attempt to calculate the response time.

There is only one process with higher priority, so there is only one process to consider for interference.

$W_2^0 = C_2 + B_2 = 35 + 10 = 45ms$

$W_2^1 = C_2 + B_2 + \text{ceiling}(W_2^0 / T_3) * C_3 = 35 + 10 + \text{ceiling}(45 / 50) * 20 = 35 + 10 + 1 * 20 = 65ms$

$W_2^2 = C_2 + B_2 + \text{ceiling}(W_2^1 / T_3) * C_3 = 35 + 10 + \text{ceiling}(65 / 50) * 20 = 35 + 10 + 2 * 20 = 85ms$

$W_2^2 = C_2 + B_2 + \text{ceiling}(W_2^1 / T_3) * C_3 = 35 + 10 + \text{ceiling}(85 / 50) * 20 = 35 + 10 + 2 * 20 = 85ms$

Giving us

$R_2 = 85ms$

The lowest priority is similar in calculation, except the lowest priority cannot suffer blocking and it has two processes which may cause interference.

$W_1^0 = C_1 + B_1 = 70 + 0 = 70ms$

$W_1^1 = C_1 + B_1 + \text{ceiling}(W_1^0 / T_3) * C_3 + \text{ceiling}(W_1^0 / T_2) * C_2$
$= 70 + 0 + \text{ceiling}(70 / 50) * 20 + \text{ceiling}(70 / 500) * 35$
$= 70 + 0 + 2 * 20 + 1 * 35 = 145ms$

$W_1^2 = C_1 + B_1 + \text{ceiling}(W_1^1 / T_3) * C_3 + \text{ceiling}(W_1^1 / T_2) * C_2$
$= 70 + 0 + \text{ceiling}(145 / 50) * 20 + \text{ceiling}(145 / 500) * 35$
$= 70 + 0 + 3 * 20 + 1 * 35 = 165ms$

$W_1^3 = C_1 + B_1 + \text{ceiling}(W_1^2 / T_3) * C_3 + \text{ceiling}(W_1^2 / T_2) * C_2$
$= 70 + 0 + \text{ceiling}(165 / 50) * 20 + \text{ceiling}(165 / 500) * 35$
$= 70 + 0 + 4 * 20 + 1 * 35 = 185ms$

$W_1^4 = C_1 + B_1 + \text{ceiling}(W_1^3 / T_3) * C_3 + \text{ceiling}(W_1^3 / T_2) * C_2$
$= 70 + 0 + \text{ceiling}(185 / 50) * 20 + \text{ceiling}(185 / 500) * 35$
$= 70 + 0 + 4 * 20 + 1 * 35 = 185ms$

Giving us

$R_1 = 185ms$

Now we can compare the result of the response time analysis to the deadlines previously defined

Process	Response time (R)	Deadline (D)
Input button reading process	25ms	30

Input flow reading process	85ms	200
Output valve angle process	185ms	200

From this we can see that all processes will meet their deadlines. In our example we have used rate monotonic priority assignment, but that is not a demand for the ICPP and the response time analysis to be valid. The rate monotonic priority assignment is simply guaranteed to be at least as schedulable as any other in pre-emptive fixed priority-based scheduling.

4.2. Implementation/pseudo-code

To implement the above using out platform is relatively simple, and the below pseudo-code example will show this.

4.2.1. Input button reading process

```
class InputButtonReaderThread : public SporadicThread
{
public:
  InputButtonReaderThread(Mutex& runningLock, RunningValues& runningValues)
    : SporadicThread(3, 40000), mRunningLock(runningLock),
      mRunningValues(runningValues)
  {
    // attach button interrupt to this->release method.
  }
  void sporadicWork()
  {
    mRunningLock.lock();
    // update running values
    mRunningLock.unlock();
  }
private:
  Mutex& mRunningLock;
  RunningValues& mRunningValues ;
};
```

4.2.2. Input flow reading process

```
class InputFlowReaderThread : public PeriodicThread
{
public:
  InputFlowReaderThread(Mutex& runningLock, Mutex& valueLock,
                        RunningValues& runningValues, FlowValues& flowValues)
    : PeriodicThread(2, 500, 40000), mRunningLock(runningLock),
      mValueLock(valueLock), mFlowValues(flowValues)
  { }
  void periodicWork()
  {
    runningLock.lock();
    // Evaluate whether to continue based on running values
    runningLock.unlock();
```

```
    // Read flow values from flow meter
    mValueLock.lock();
    // Update the flow values
    mValueLock.unlock();
  }
private:
  Mutex& mRunningLock;
  Mutex& mValueLock;
  RunningValues& mRunningValues;
  FlowValues& mFlowValues;
};
```

4.2.3. Output valve angle process

```
class OutputValveAngleThread : public PeriodicThread
{
public:
  OutputValveAngleThread(Mutex& runningLock, Mutex& valueLock,
                         RunningValues& runningValues, FlowValues& flowValues)
    : PeriodicThread(1, 500, 40000), mRunningLock(runningLock),
      mValueLock(valueLock), mFlowValues(flowValues)
  { }
  void periodicWork()
  {
    runningLock.lock();
    // Evaluate whether to continue based on running values
    runningLock.unlock();

    mValueLock.lock();
    // Access shared memory and read the flow values.
    mValueLock.unlock();

    // Update the valve angle based on the read flow values
  }
private:
  Mutex& mRunningLock;
  Mutex& mValueLock;
  RunningValues& mRunningValues;
  FlowValues& mFlowValues;
};
```

4.2.4. Main

```
int main(int argc, char* argv[])
{
  Mutex runningLock(3); // 3 is ceiling priority for running-lock
  Mutex valueLock(2); // 2 is ceiling priority for value-lock
  RunningValues runningValues;
  FlowValues flowValues;

  InputButtonReaderThread buttonReader(runningLock, runningValues);
  InputFlowReaderThread flowReader(runningLock, valueLock ,
                                   runningValues, flowValues);
  OutputValveAngleThread valveAngle(runningLock, valueLock ,
                                    runningValues, flowValues);
```

```
  buttonReader.start();
  flowReader.start();
  valveAngle.start();

  // Block "forever"
  return 0;
}
```

As the current design do not include the parts of RT Java, which monitors whether the dead-lines are met, whether the pre-emption is performed and whether the immediate ceiling priority protocol is obeyed, it is not possible to directly determine whether the platform meets its requirements just by running this test. It is however possible to instrument the software with some diagnostics code to perform these tests.

This can be done by

1. Use the Clock to get the absolute time when the ...Work methods are entered and left and then compare the relative time between these values with the deadline for the process.
2. Log the absolute times when a process becomes runnable, blocked or delayed (requires some OS-support, yet much can be done by instrumenting the Mutex and Thread classes). Analysing this log will describe which processes are allowed to run when.
3. Monitor how often the sporadicWork is performed and thereby determining if the minimum interval has been violated.