Generic Embedded Computer

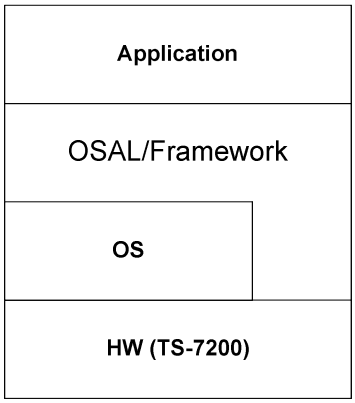
Course: Grundlæggende Indlejrede Systemer (GiS)
Modul 2, Multi programming and Real-Time systems
Group: Group 3
Date: Juni 09
Names: Thomas Ginnerup Kristensen
Henrik Arentoft Dammand
Anders Hvidgaard Poder

1. Introduction

Through several industrial projects we have experienced that a generic HW platform including basic peripherals, is suitable for a lot of different control 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.

There are many equally good alternatives to the TS-7200, even alternatives within the TS-family, yet the TS-7200 outperforms them with respect to cost (especially in low numbers) and flexibility. The TS-7200 has the needed functionality, without a lot of unnecessary extra functionality, which is common for a general purpose platform (even for an application-specific processor, as one might argue this is). Finally the TS-7200 has an acceptable delivery time and is readily available. For these reasons the TS-7200 has been chosen.

1.2. SW Platform

The framework shall be developed in C++ and encapsulate the rather verbose C API to the POSIX/RTAI compliant OS interface. The reason for this encapsulation is two-hold. Firstly the C-API of POSIX/RTAI is difficult to work with in OO-programming and second, the aim of this platform is a stripped-down RT-platform, which only exposes a limited number of functionalities and with many hard-coded decisions, e.g. FPS and IPCC.

Both commercial and open source SW frameworks are available and could be used instead of developing our own. Boost [R1] and POCO [R2] 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. As an example can be mentioned that at present the raw Boost download is over 50MB in size and contains 27091 files (approximately 7000 classes). Boost is also not designed specifically for Real-time, and much analysis would therefore

have to be done to ensure that it does not use functionality, which ruins the real-time performance. However, both Boost and Java RT has many good features which may be used as inspiration for the proprietary platform.

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. This is possible by eliminating all unnecessary functionality and only focus on the real-time conformant basic functionality needed.

OO has an advantage over functional languages (like C) in that it is a higher level programming language, therefore allowing for faster development. Of the OO languages which are light weight (do not require a VM), supported by the HW-platform and know by the developers (a feature which should not be ignored), C++ stands out as the obvious and only choice. As C++ is a widely known and supported language it will also make replacing the HW-platform or the developers, should the need arise, much simpler.

1.3. Objectives

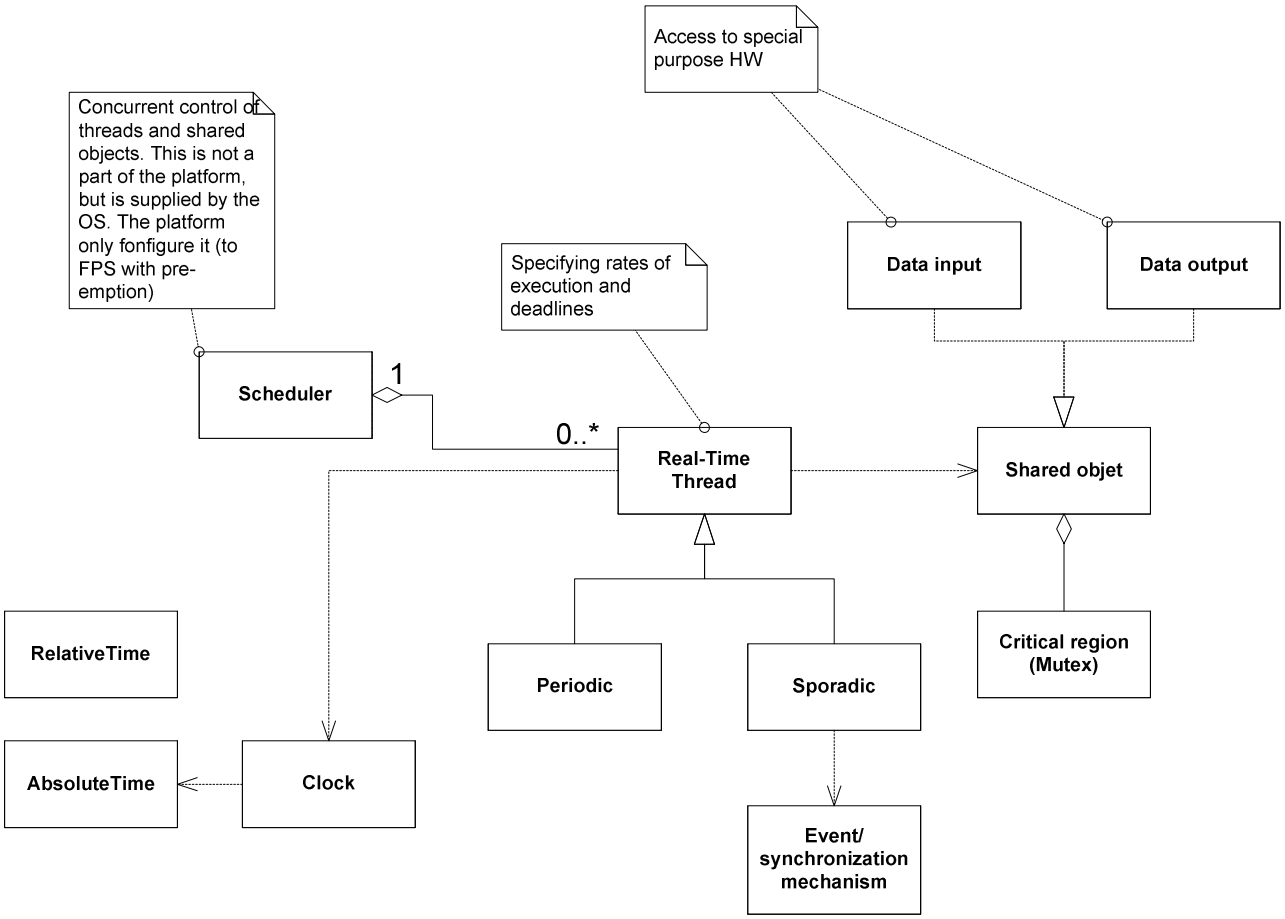
The objectives of the project are to:

- 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 an 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 briefly 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).
- Real-Time Thread — Base class of periodic and sporadic threads.
- Clock — The clock class provide facilities to handle and measure time.
- AbsoluteTime — The Time classes abstract the concept of time related to the
- RelativeTime — 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). The Mutex employ ICPP.
- 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 constructor 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 objects	**RT Java counterpart**
Periodic thread	RealtimeThread used with instances of PriorityParameters and PeriodicParameters as constructor parameters
Sporadic thread	RealtimeThread used with instances of PriorityParameters and SporadicParameters as constructor parameters
Real-Time thread	Thread
Clock	Clock
AbsoluteTime	AbsolutTime
RelativeTime	RelativeTime
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, and then passes objects by value. This makes the RT Java requirements for a stack quite small. POSIX/C++ support automatic objects created on the stack; hence POSIX/C++ is more stack memory intensive as entire objects can be passed by value. Therefore the Thread class is created with the stack size as constructor parameter. This makes it possible for the application programmer to individually state the stack size needed for each instance.

As RT Java is a good example of a high-level OO-API for a real time system we use this as inspiration for many of our classes. We are, however, not attempting to fully implement the Java API in C++, as this would be a monumental task, and would also be strange at times, as the Java RT rely on a VM with garbage-collection. This means that talks of RawMemoryAccess (required by VM's built in MMU) and ImmortalMemory (make sure memory is not garbage-collected) makes no sense in C++, and is naturally not included. Furthermore, the RT Java API is far larger than we desire, and many configuration options has been removed, e.g. the option of choosing scheduling-type (FPS, EDF, ...) and priority protection (Priority inheritance, ICCP, ...) will be hard-coded in the simple C++ platform to always be FPS and ICCP.

Furthermore the questionable aspects of RT Java's public API is removed, as it makes the RT analysis very difficult. As examples for these aspects can be mentioned Thread.sleep, Thread.join, Thread.stop, etc.

Finally it should be mentioned that RT Java, by using a VM, which either runs as the OS or runs as a single process in an OS, has implemented their own scheduler. It is not the goal of this assignment to implement a scheduler. It is expected that a suitable OS-scheduler is available; in our case the TS-Linux OS, which can be configured to run a FPS (pre-emptive). This means that all Scheduling-parameters is fixed and the scheduler itself is outside the control of the platform – naturally this potentially put some serious limitations on the portability of the platform, as any OS-specific functionality will essentially be non-portable. It is also not ensured that all the diagnostics and instrumentation of RT-Java can be supported by the platform. Finally anything relating to Security (Java SecurityException) is also not a part of the platform for obvious reasons.

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.

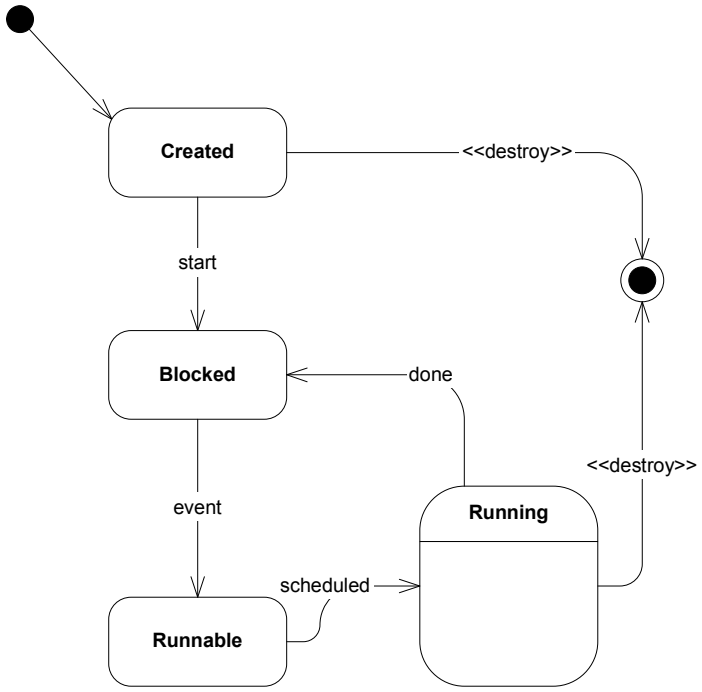
The application programmer must be able to specify the release parameter for the thread and supply the algorithm to be executed when released.

Furthermore all threads come with built in instrumentation to monitor that the period/minimum inter-arrival interval as well as the deadline of the thread is not violated. In RT Java this is done by

supplying a deadline and period to the thread, and then, if a deadline is missed or if a period cannot be respected, call the miss-/overrun-handler respectively. In this platform the same principle will be adopted, yet these handlers will not be called immediately when the violation occur, as this requires scheduler support. Instead they will be called when the offending threat is given the CPU (scheduled back in) and completes its scheduled work; i.e. the deadline overrun is verified at the end of the threats execution cycle and the period/min. inter-arrival time overrun is verified at the beginning of the threats execution cycle. This is not as accurate as the Java implementation, yet with the very limited scheduling functionality in the platform it is as good as it can get, and from a usage point of view it is acceptable.
RT Java also supports the possibility to have a cost overrun handler. This handler is called if the cost of the algorithm exceeds the specified time granted. This functionality can only be implemented with scheduler support and are therefore not included in the platform. This fact prerequisites that much effort must be put into the establishment of the cost, since no indication of an overrun is possible.

As mentioned, in C and C++ a stack size is quite important. Naturally when a stack size is supplied the stack can overflow (the stack can also overflow in Java; it is just a little more difficult as it requires a lot of functions, but an infinite recursive function call will do it). In Java a Stack-overflow results in an exception, and this makes sense as it is a design error. The same solution will be used in the platform.

Sporadic Thread:



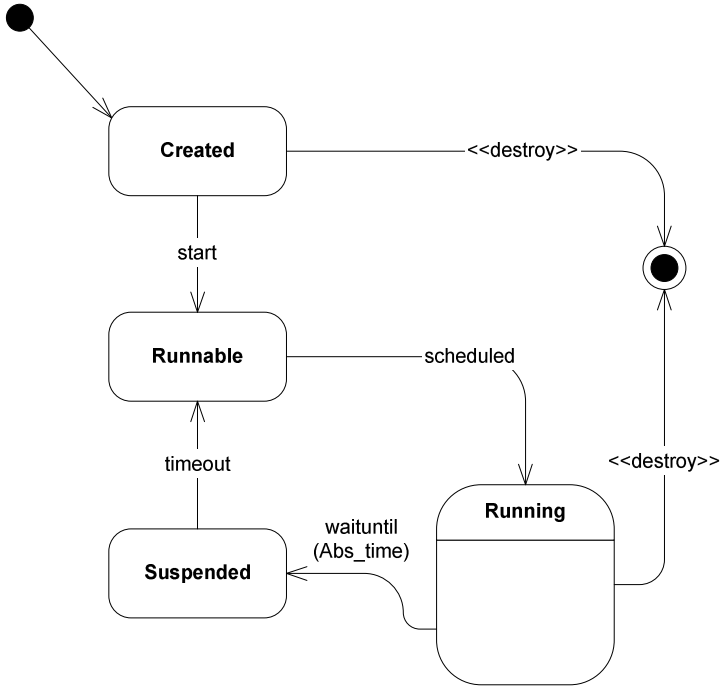
The sporadic thread is used to handle non-periodic events. During normal operation, the sporadic thread steps through the 3 states; Runnable, Running and Blocked. When the thread is created it

contains the priority, deadline and minimum inter-arrival time as well as the miss handler (for deadline misses) and overrun handler (for minimum inter-arrival time violations). When started 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 executes the algorithm supplied/implemented by the application programmer. After completion it waits for the event and thereby enters the Blocked state.

To perform runtime monitoring of RT criteria, the sporadic thread log the time it entered the Runnable state and compare this to the last time this occurred, thereby determining if the minimum inter-arrival time was violated. There are several options here; two events occur before the sporadic threat has an option of handling any of them, two events occur, the second while the sporadic thread is processing the first, and finally two events occur, the second not until the sporadic thread has completed processing the first. All three may occur faster than the minimum inter-arrival time, yet handling them is slightly different. The overrun handler is always called, and it is always the thread of the SporadicThread that calls the handler. If multiple events arrive before the sporadic handler has handled any of them the violation is simply tagged and then the excess events are simply ignored, if an event occurs while the sporadic process is handling one, the violation is tagged and the event set. The sporadic process will call the overrun handler next time around. The same procedure is used for events occurring too fast after the sporadic thread execution has completed.

The event is attached by the application programmer, and may be any event-method which calls release, including an ISR – this means that there are strong limitations on what is allowed in the release method (e.g. no potential blocking and as fast as possible) as it must be callable from an ISR.

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.

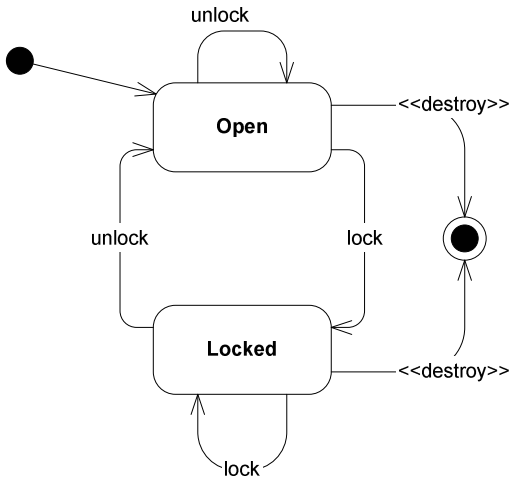
When the thread is created it contains the period, the threads priority and the deadline as well as the miss handler (for deadline misses) and overrun handler (for period violations), which all are supplied by the creator of the object.

A similar principle as the sporadic thread is used for runtime monitoring of RT criteria; the periodic thread log the time it entered the Runnable state. When the periodic thread completes its execution it compares this to the deadline and the period to determine if one or both of these are violated. And call the respective handler.

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 ICCP functionality of the Mutex can be implemented very simply by simply boosting the calling threads priority to the ceiling priority as soon as lock is called. However, it is also directly available through the POSIX/RTAI API, so this is not necessary.

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 only scheduling parameter that may be set by the user of 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.

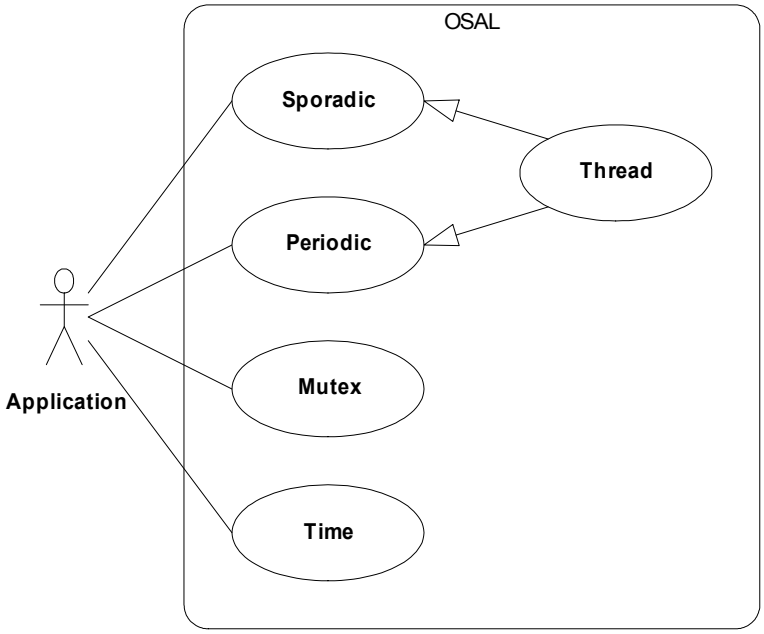
The applications will most possibly consist of both sporadic and periodic threads. To support sporadic threads and still make it possible to perform the response time analysis, the sporadic threads are given a worst case period (the minimum inter-arrival time), and are then treated as periodic threads.

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.

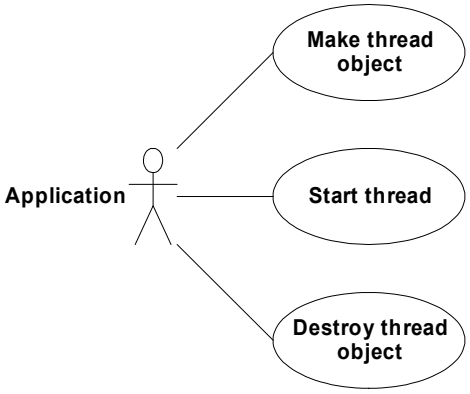
3.1. Use cases

Basically, the OS abstraction layer (OSAL) provides some facilities to support threads, mutexes 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.



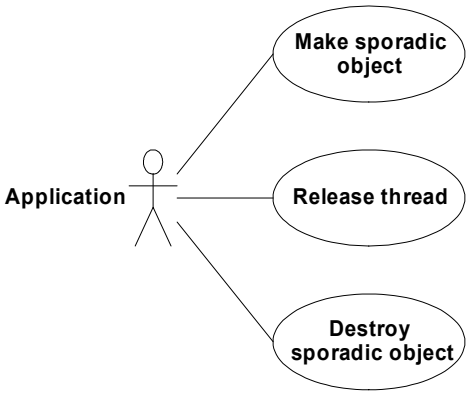
3.2. Thread

A thread is a function that runs as a process and can run concurrently with other threads. The thread class provide some basic thread management facilities such as start the thread and setting its priority (at creation time only) etc. The thread class is indented to be a base class for periodic or sporadic threads and should never be used directly.



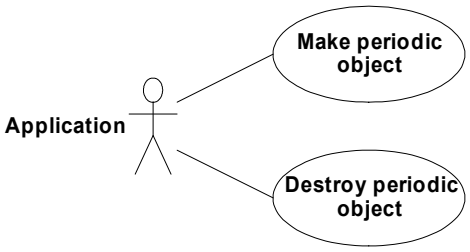
3.3. Sporadic

When application creates a sporadic thread object it must be set-up with several parameters such as priority, a thread function and stack size, minimum inter-arrival time, a deadline, an overrun handler and a miss handler. Release is a method for signalling the sporadic thread to execute the event function.



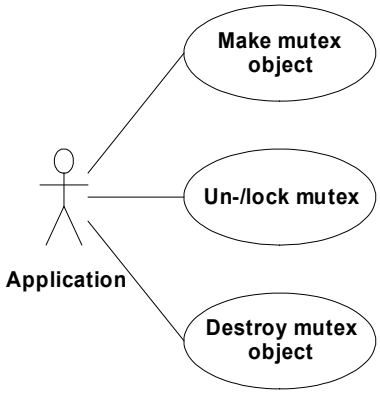
3.4. Periodic

A periodic thread is created with several parameters such as priority, a periodic thread function, stack size, a periodic time interval, a deadline, an overrun handler and a miss handler.



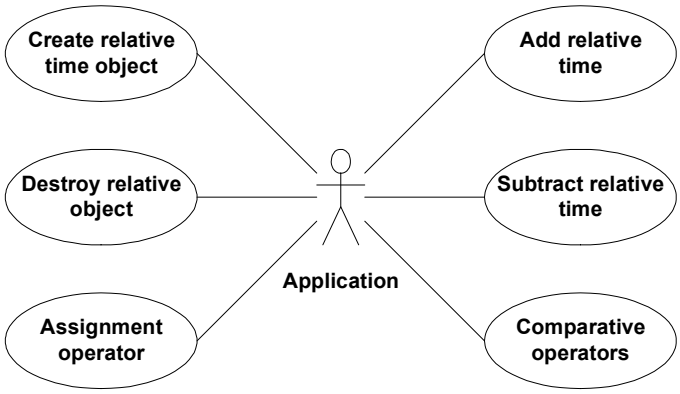
3.5. Mutex

A mutex object must be created with a parameter that indicates the priority ceiling. The lock method will block the thread if it is held by another tread. The unlock method unlocks the mutex so that it can be acquired by other threads.



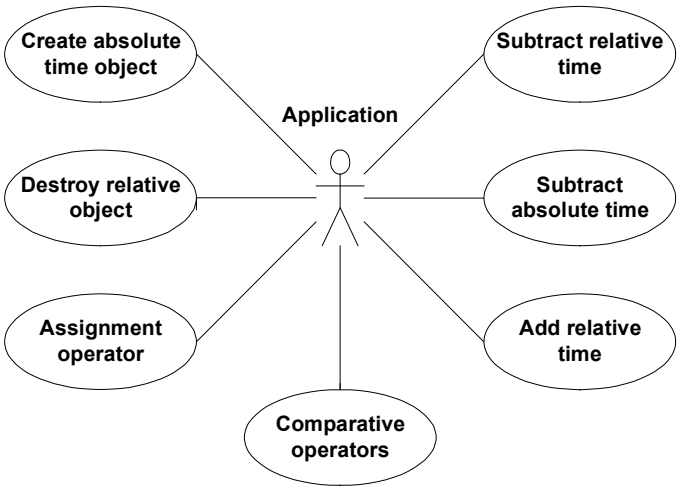
3.6. Relative time

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

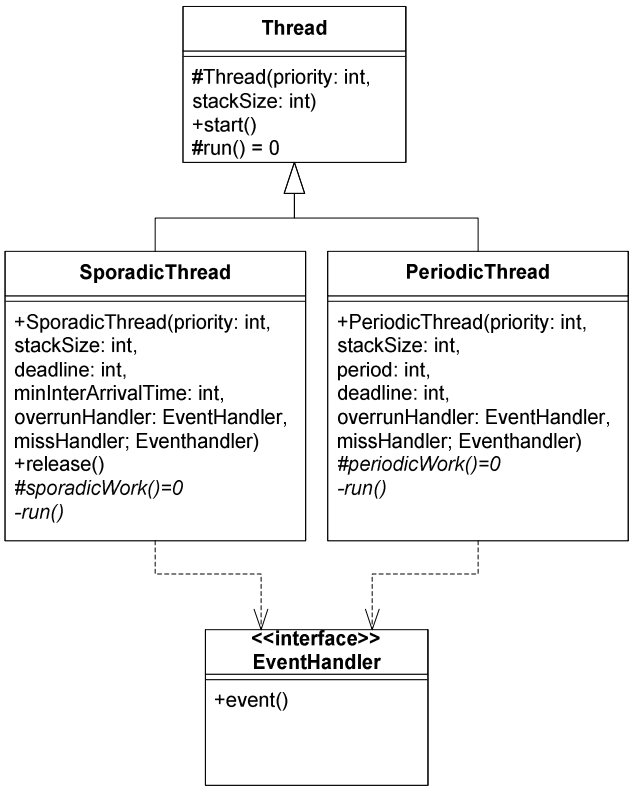
An absolute time object can be created in several ways. It can be created as an empty object or with an absolute time value. Simple arithmetic's such as addition or subtraction are supported and absolute times can be compared with each other.



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:



Thread

Thread(int priority, int stackSize)

Constructor - creates and initialize a thread with given stack size and priority.

start

start()

Starts the execution of the thread.

run

run()

Abstract method where the code to be executed in the thread is to be placed.

SporadicThread

SporadicThread(int priority, int stackSize, RelativeTime deadline, RelativeTime minInterarrival,
EventHandler overrun, EventHandler miss)

Contructor - creates and initialize a sporadic thread with given release parameters and event handlers.

release

release()

Release the thread for execution.

sporadicWork

void sporadicWork() = 0

Do whatever the sporadic thread needs to do. Must be overrided by a derived class.

run

run()

Private implementation of sporadic thread algorithm

PeriodicThread

PeriodicThread(int priority, int stackSize, RelativeTime deadline, RelativeTime interval,
EventHandler overrun, EventHandler miss)

Contructor - creates and initialize a periodic thread with given release parameters and event handlers.

periodicWork

void periodicWork() = 0

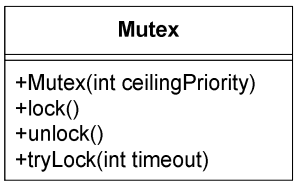
Do whatever the periodic thread needs to do. Must be overrided by a derived class.

run

run()

Private implementation of periodic thread algorithm

Mutex:



Mutex

Mutex(int ceilingPriority)

Contructor - creates a mutex with priority ceiling.

lock

void lock()

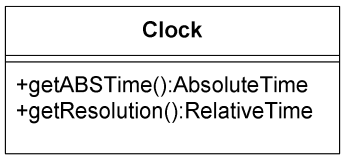
Locks the mutex. Blocks if the mutex is held by another thread.

unlock

void unlock()

Unlocks the mutex.

Clock:



getABSTime

AbsoluteTime getABSTime()

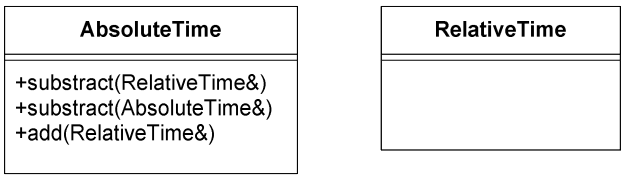
Get the current time.

getResolution()

RelativeTime getResolution()

Get the interval between ticks.

Time:



subtract

AbsoluteTime subtract(RelativeTime&)

Subtracts the relative time from absolute time. The result is a new absolute time.

subtract

RelativeTime subtract(AbsoluteTime&)

Subtracts two absolute times. The result becomes a relative time.

add

AbsoluteTime add(RelativeTime&)

Add a relative time to an absolute time. The result is a new absolute time.

Design

This section will be kept very short, and will only include the used POSIX/RTAI functionality.

POSIX function	Description
pthread_create	This C function is used to create and spawn a thread. It takes, via pthread attributes as well as directly, a priority, a starting function and possible arguments to the starting function. We will naturally supply the priority, a generic starting function, which will receive the this pointer of the calling thread to perform a callback on (this->run()). This function must be called from start.
pthread_mutex_t, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutexattr_setprioceiling	These methods may be used to create a POSIX mutex with immediate ceiling priority protocol as well as locking and unlocking it.

It is possible to destroy a thread, yet as this platform assumes that all threads are created and started during program initiation and that they run for the duration of the application lifetime it is unnecessary to include means of gracefully disposing of threads – the OS process clean-up system will automatically terminate the threads upon application termination.

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 a button. This may be seen in Figure 1.

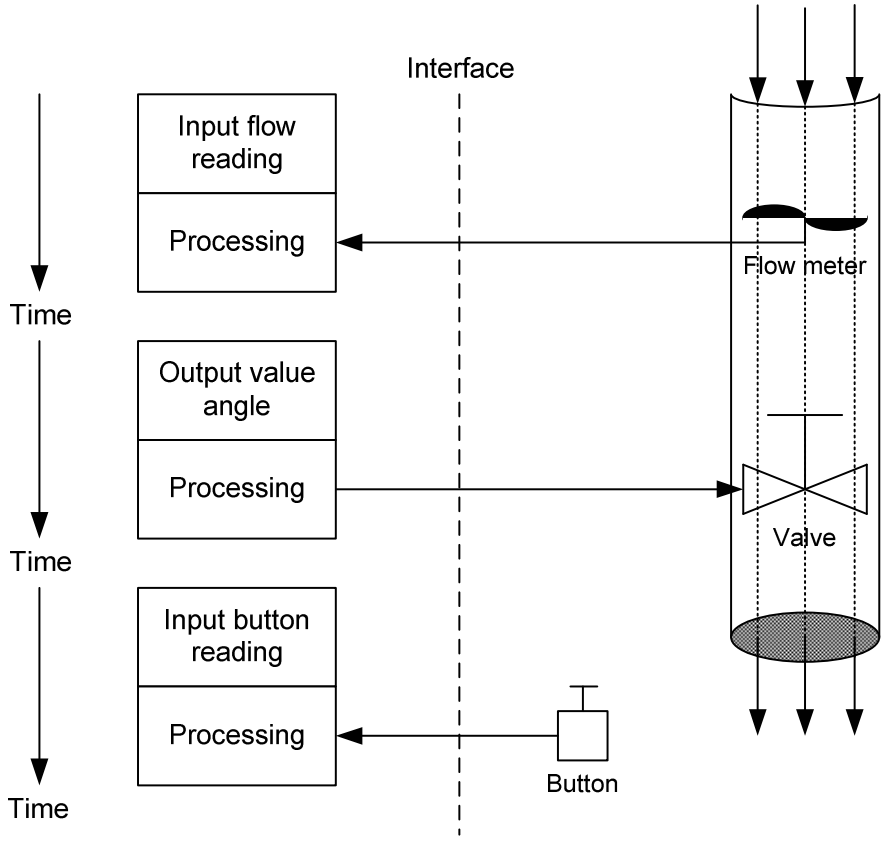


Figure 1: Experimental test set-up

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 shared data with the other processes. If we consider it an important configuration-button, then the sporadic process has higher priority than 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.

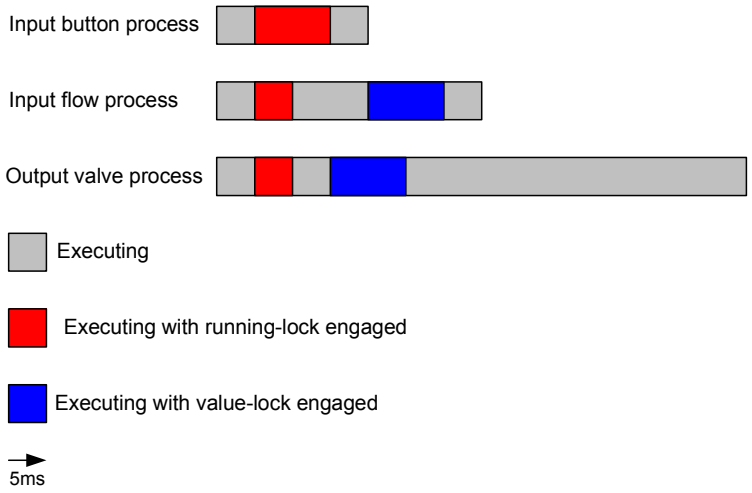
Priority-wise we define the valve to always function on the newest data, thereby making the input flow reading the most important. The periods are simply chosen, as is the deadlines and execution cost (highest priority = highest number).

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

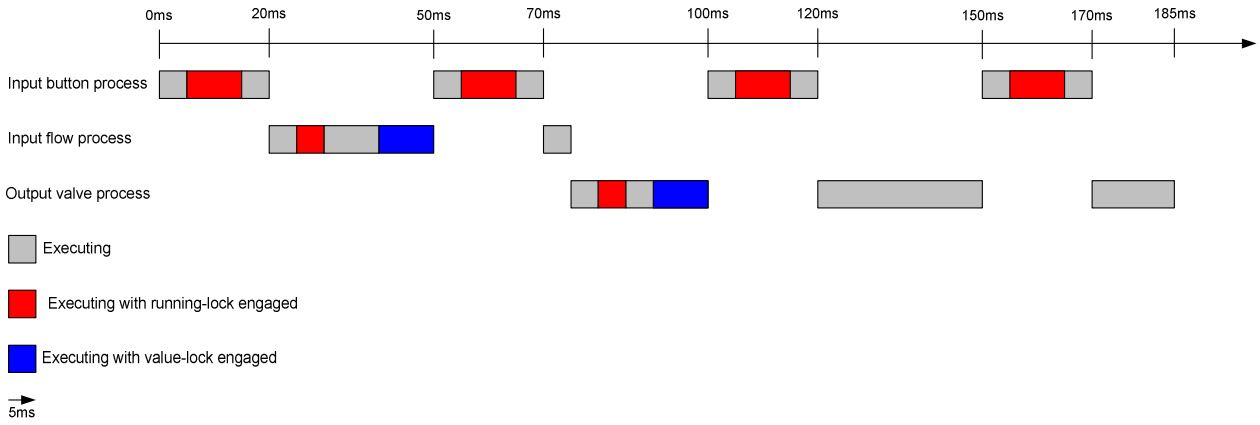
We could consider the release jitter. This is defined by the scheduler, which in this platform is defined by the operating system; TS-Linux. Though the exact value is unknown it is safe to say that it will be many orders of magnitude smaller than the other factors in the system. Lets set it at e.g. 10us. 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 is 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 processes we can set up the following colour-coded diagram for the execution cost.



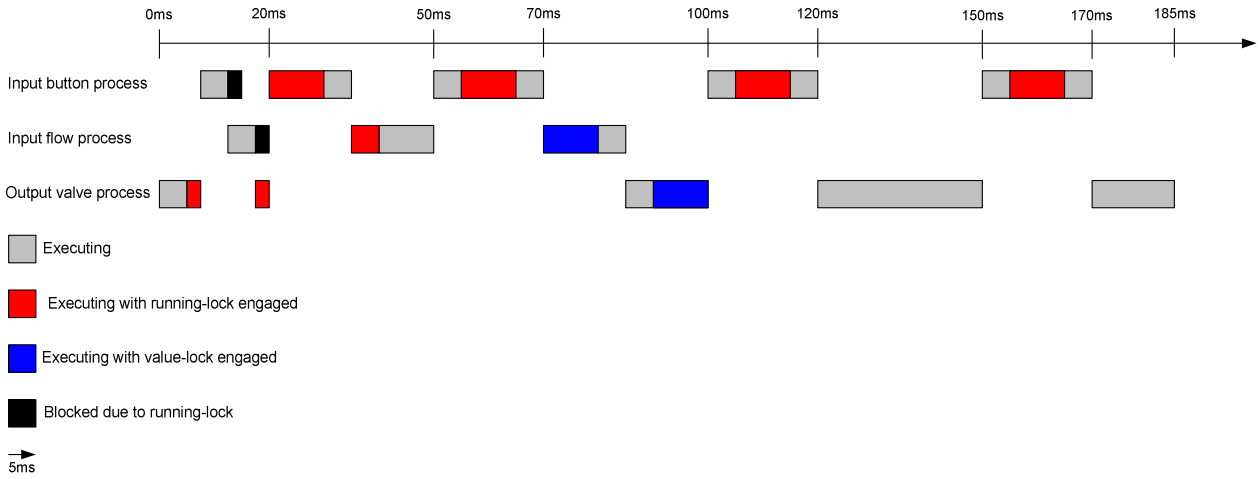
Based on this we can perform the response time analysis and determine if the system will always meet its deadlines.

Before we consider the mathematical solutions, we can do a graphical representation of the worst case without blocking, which are 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 manner, 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 for FPS with ICPP (there are also formulas for other scheduling and critical region principles, yet those are not important here).

$R_i = C_i + I_i + B_i$

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. This means that the process will be boosted to the ceiling 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 process with the lock might be scheduled out by another process with equal priority, which could then take another lock.

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_i^0 is calculated by simply setting $sum_j(ceiling(W_i^n / T_j)*C_j)$ to 0, which makes

$W_i^0 = C_i + B_i$

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.

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

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

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

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

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.

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

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

$W_1^2 = C_1 + B_1 + ceiling(W_1^1 / T_3) * C_3 + ceiling(W_1^1 / T_2) * C_2$

$= 70 + 0 + \text{ceiling}(145 / 50) * 20 + \text{ceiling}(145 / 500) * 35$
$= 70 + 0 + 3 * 20 + 1 * 35 = 165\text{ms}$

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

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

Giving us

$R_1 = 185\text{ms}$

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.

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 EventHandler
{
public:
  InputButtonReaderThread(Mutex& runningLock, RunningValues& runningValues)
    : SporadicThread(3, 40000, 30, 50, this, this),
      mRunningLock(runningLock), mRunningValues(runningValues)
  {
    // attach button interrupt to this->release method.
  }
  void sporadicWork()
  {
    mRunningLock.lock();
    // update running values
    mRunningLock.unlock();
  }
  void event()
```

```
  {
    // handle minimum inter-arrival time or deadline violation.
  }
private:
  Mutex& mRunningLock;
  RunningValues& mRunningValues ;
};
```

4.2.2. Input flow reading process

```
class InputFlowReaderThread : public PeriodicThread, public EventHandler
{
public:
  InputFlowReaderThread(Mutex& runningLock, Mutex& valueLock,
                        RunningValues& runningValues, FlowValues& flowValues)
    : PeriodicThread(2, 40000, 500, 200, this, this),
      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();
  }
  void event()
  {
    // handle overrun or deadline violation.
  }
private:
  Mutex& mRunningLock;
  Mutex& mValueLock;
  RunningValues& mRunningValues;
  FlowValues& mFlowValues;
};
```

4.2.3. Output valve angle process

```
class OutputValveAngleThread : public PeriodicThread, public EventHandler
{
public:
  OutputValveAngleThread(Mutex& runningLock, Mutex& valueLock,
                         RunningValues& runningValues, FlowValues& flowValues)
    : PeriodicThread(1, 40000, 500, 200, this, this),
      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
  }
  void event()
  {
    // handle overrun or deadline violation.
  }
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;
}
```

The current design does not include monitoring whether the pre-emption is performed correctly and whether the immediate ceiling priority protocol is obeyed. This is normally not something one verifies, as it is guaranteed by the operating system. It is however possible to instrument the software with some diagnostics code to perform these tests.

This can be done by logging 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. It is however

not something that should be done during production run, but merely if verification of the OS-functionality is desired.

5. Conclusion

We have demonstrated that it is possible to implement a real-time application using the API supported by the framework. However, the experimental test application is a somewhat simplified example and using this generic embedded computer solution to implement it, is like using a sledgehammer to crack a nut. This particular example could easily have been implemented with a simple 10$ microcontroller using a cyclic executive scheduling scheme. It does, however, show that the framework supports the basic building blocks for developing a true real-time application through a simple but powerful API.

It supports the application programmer in creating both sporadic and periodic processes using OO techniques, and makes it easy to elaborate the system's temporal requirements directly into active objects in the application. It provides means for communication between the processes using shared data regions and ensures that no deadlocks are possible in these regions (this of course is under the assumption that unique priorities are used).

The framework is highly inspired by the RT Java. Like RT Java, the framework holds predictable execution as first priority in all trade-offs, i.e. predictable and analysable scheduling was a crucial goal.
Since the framework is developed in C++, Java's sophisticated memory management cannot be supportet and is left to the application programmer. However, because of its non-determinism, carbage collection is often disabled in in RT Java applications, leaving the application programmer with the memory management responsibility.
The framework includes interfaces for miss and overrun handlers like RT Java. This means that it supports development of fault tolerant systems.

The approach have been to implement a stripped down RT framework where many decisions already have been made by the framework, hence makes the framework attractive and easy to use, this of course is on account of flexibility regarding other scheduling and communication techniques.

This said, configuring the Linux kernel to support RT scheduling, and verifying this, is a major task, that still lies ahead.

6. References

[R1] - Boost	http://www.boost.org/
[R2] - POCO	http://pocoproject.org/