Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

OpenComRTOS: A Runtime Environment for Interacting Entities

Bernhard H.C. SPUTH [a,1], Oliver FAUST [a] Eric VERHULST [a] and Vitaliy MEZHUYEV [a]
[a] *Altreonic*

Abstract. OpenComRTOS is one of the few Real-Time Operating Systems for embedded systems that was developed using formal modelling techniques. The goal was to obtain a proven trustworthy component with a clean architecture that delivers high performance on a wide variety of networked embedded systems, ranging from single processor to distributed systems. The result is a scalable relibable communication system with real-time capabilities. Besides, a rigorous formal verification of the kernel algorithms led to an architecture which has several properties that enhance safety and real-time properties of the RTOS. The code size in particular is very small, typically 10 times less than a typical equivalent single processor RTOS. The small code size allows a much better use of the on-chip memory resources, which increases the speed of execution due to the reduction of wait states caused by the use of external memory.

To this point we ported OpenComRTOS to the MicroBlaze processor from Xilinx, the Leon3 from ESA, the ARM Cortex-M3, the Melexis MLX16, and the XMOS. In this paper we concentrate on the Microblaze port, which is an environment where OpenComRTOS competes with a number of different operating systems, including the standard operating system Xilinx Micro Kernel. This paper reports code size figures of the OpenComRTOS on a MicroBlaze target. We found that this code size is considerably smaller compared with published code sizes of other operating systems.

Keywords. OpenComRTOS, Embedded systems, System engineering, RTOS

1. Introduction

Real-Time Operating Systems (RTOSs) are a key software module for embedded systems, often requiring properties of high reliability and safety. Unfortunately, most commercial, as well as open source implementations cannot be verified or even certified, e.g. according to the DoD_178B [1] or IEC61508 [2] standards. Similarly, software engineering is often done in a non-systematic way, although well defined and established Systems Engineering Processes exist [3,4]. The software is rarely proven to be correct even though this is possible with formal model checkers [5]. In the context of a unified systems engineering approach [6] we undertook a research project where we followed a stricter methodology, including formal model checking, to obtain a network-centric RTOS which can be used as a trusted component.

1.1. General requirements for OpenComRTOS

The history for this project goes back to the early 1990's when a distributed real-time RTOS called Virtuoso (Eonic Systems) [7] was developed for the INMOS transputer [8]. This pro-

[1]Corresponding Author: *Bernhard H.C. Sputh; Altreonic; Gemeentestraat 61A bus 1; 3210 Linden Belgium.* E-mail: bernhard.sputh@openlicensesociety.org.

cessor had built-in support for concurrency as well as interprocess communication and was enabled for parallel processing by way of 4 communication links. Virtuoso allowed such a network of processors to be programmed in a topology transparent way. Later, the software evolved and was ported from single chip micro-controllers to systems with over a thousand Digital Signal Processors until the technology was acquired by Wind River and after a few years removed it from the market. The OpenComRTOS project was motivated by the lessons learned from developing three Virtuoso generations. These lessons became part of the requirements. We list the most important ones:

- Scalability: The RTOS should support very small single processor systems, as well as widely distributed processing systems interconnected through external networks like the internet. To achieve that, the software components must be independent of the execution environment. In other words, it must be possible to map the software components onto the network topology.
- Heterogeneous: The RTOS should support systems which consist of multiple nodes, with different CPU architectures. Naturally, different link technologies should be usable as well, ranging from low speed links such as RS232 up to high speed Ethernet links.
- Efficiency: The essence of multi-processor systems is communication. The challenge, from an RTOS point of view, is keeping the latency to a minimum while at the same time maximizing the performance. This is achieved when most of the critical code resides in the limited amount of on-chip memory.
- Small code size: This has a double benefit: a) performance and b) less complexity. Less complex systems have fewer potential sources of errors and side-effects.
- Dependability: As testing of distributed systems becomes very time consuming, it is mandatory that the system software can be trusted from the start. As errors typically occur in "corner cases", the use of formal methods was deemed necessary.
- Maintainability and ease of development: The code needs to be clear and simple to facilitate the development of e.g. drivers, the latter have often been the weak point in system software.

OpenComRTOS provides a runtime environment which supports these requirements. The remainder of this paper focuses on this runtime environment and the execution on a MicroBlaze target. But, before we discuss the details of OpenComRTOS in greater detail, we deduce two general points from the list of requirements.

The scalability requirement imposes that data-communication is central in the RTOS architecture. The trustworthiness and maintainability aspects are addressed in the context of a Systems Engineering methodology. The use of common semantics during all activities is crucial, because only common semantics enable us to generate most of the implementation code from the modelling and simulation phase. Generated code is more trustworthy compared to handwritten code. To be able to use an "Interacting Entities" paradigm requires a runtime environment that supports concurrency and synchronization/communication in a native way between concurrent entities. OpenComRTOS is this runtime environment

2. OpenComRTOS architecture

Even with the problems mentioned above, Virtuoso was a successful product. The goal was to improve on its weaknesses. Its architecture had a high performance, but very hard to port and to maintain. Hence, for OpenComRTOS we adopted a layered architecture which is based on semantic layering. The lowest functionality level is limited to priority based preemptive multitasking. On this level Tasks exchange standardized Packets using an intermediate entity we call Ports. Two tasks rendezvous by one task sending a 'put' request and the other task

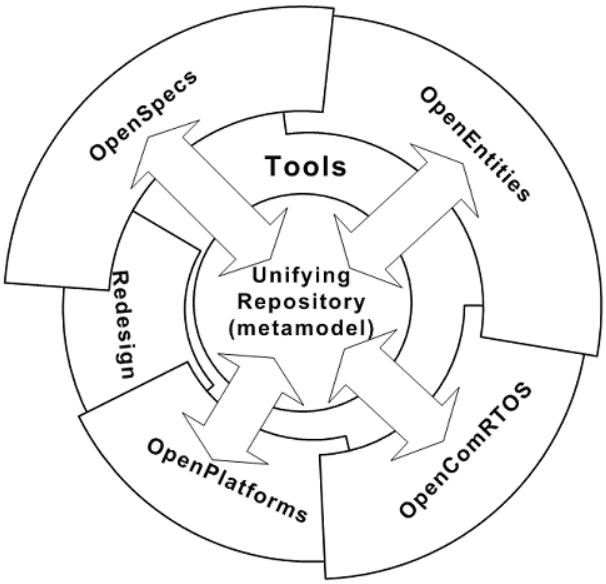

Figure 1. Open License Society: the unified view

sending a 'get' request. The port behaves similar to the JCSP Any2AnyChannel [9]. Hence, Tasks can synchronise and communicate using Packets and Ports. The Packets are the essential workhorse of the system. They have header and data fields and are exclusively used for all services, rather than performing function calls or using jump tables. Hence, it becomes straightforward to provide services that operate in a transparent way across processor boundaries. Furthermore, Packets are very efficient, because kernel operations often come down to shuffling Packets around (using handlers) between system level data structures.

At the next semantic level we added more traditional RTOS services like events, semaphores, etc (see Table 2 on Page 6 for the included RTOS services). Finally, the architecture was kept simple and modular by developing kernel and drivers as Tasks. All these Tasks have a 'Task input Port' for accepting Packets from other Tasks. This has some unusual consequences like: a) the possibility to process interrupts received on one processor on another processor, b) the kernel having a lower priority than the drivers or even c) having multiple kernel Tasks on a single node.

2.1. Systems Engineering approach

The Systems Engineering approach from the Open License Society [6], outlined in Figure 1, is a classical one as defined in[3,4], but adapted to the needs of embedded software development. It is first of all an evolutionary process using continuous iterations. In such a process, much attention is paid to an incremental development requiring regular review meetings by several of the stakeholders. On an architectural level, the system or product under development is defined under the paradigm of "Interacting Entities", which maps very well on an RTOS based runtime system. Applied to the development of OpenComRTOS, the process was started by elaborating an initial set of requirements and specifications. Next, an initial architecture was defined. From this point onwards, two groups started to work in parallel. The first group worked out an architectural model, while a second group developed initial formal models using TLA+/TLC [10]. These models were incrementally refined.

Note that no real attempt was made to model the complete system at once. This is not possible in a generic way, because formal TLA models cannot be parametrised. For example, one must model a specific set of tasks and services which leads very quickly to a state space explosion which limits the achievable complexity of such models. Hence, we modelled only specific parts, e.g. a model was built for each class of services (Ports, Events, Semaphores,

etc.). This was sufficient and has the benefit of having very clean, orthogonal models. Due to the orthogonality of the models there is no need to model the complete system, which has the big advantage that they can be developed by different teams.

At each review meeting between the software engineers and the formal modelling engineer, more details were added to the models, the models were checked for correctness and a new iteration was started. This process was stopped when the formal models were deemed close enough to the implementation architecture. Next, a simulation model was developed on a PC (using Windows NT as a virtual target). This code was then ported to a real 16bit micro controller in form of the MLX16 from Melexis, who at this time were sponsoring the development of OpenComRTOS. The MLX16 a propriety micro controller used by Melexis to develop application specific ICs, it has up to 2kiB RAM and 32kiB Flash. On this target a few specific optimizations were performed during the implementation, while fully maintaining the design and architecture. The software was written in ANSI C and verified for safe coding practices with a MISRA rule checker [11].

2.2. Lessons learnt from using formal modelling

The goal of using formal techniques is the ability to prove that the software is correct. This is an often heard statement from the formal techniques community. A first surprise was that each model gave no errors when verified by the TLC model checker. This is actually due to the iterative nature of the model development process and partly its strength. From an initially rather abstract model, successive models are developed by checking them using the model checker and hence each model is correct when the model checker finds no illegal states. As such, model checkers can't prove that the software is correct. They can only prove that the formal model is correct. For a complete proof of the software the whole programming chain as well as the target hardware should be modelled and verified. This is an unachievable goal due to its complexity and the resulting state space explosion. Nevertheless, it was attempted in the Verisoft project [12]. The model itself would be many times larger than the developed software. This indicates that if we would make use of verified target processors and verified programming language compilers, model checking becomes practical, because it is limited to modelling the application.

Other issues, related to formal modelling, were also discovered. A first issue is that the TLC model checker declares every action as a critical section, whereas e.g. in the case of a RTOS, many components operate concurrently and real-time performance dictates that on a real target the critical sections are kept as short as possible. This forced us to avoid shared data structures. However, it would be helpful to have formal model assistance that indicates the required critical sections.

2.3. Benefits obtained from using formal modelling

As was outlined above, the use of formal modelling was found to result in a much better architecture. This benefit results from successive iteration and review of the model. Another reason for the better architecture is the fact that formal model checkers provide a higher level of abstraction compared to the implementation. In the project we found that the semantics associated with specific programming terms involuntarily influence choices made by the architecture engineer. An example was the use of both waiting lists and Port buffers, which is one of the main concepts of OpenComRTOS. A waiting list is associated with just one waiting action but one overlooks the fact that it also provides buffering behaviour. Hence, one waiting list is sufficient resulting in a smaller and cleaner architecture.

Formal modelling and abstract levels have helped to introduce, define and maintain orthogonal architectural concepts. Orthogonality is key to small and safe, i.e. reliable, designs. Similarly, even if there was a short learning curve to master the mathematical notation in

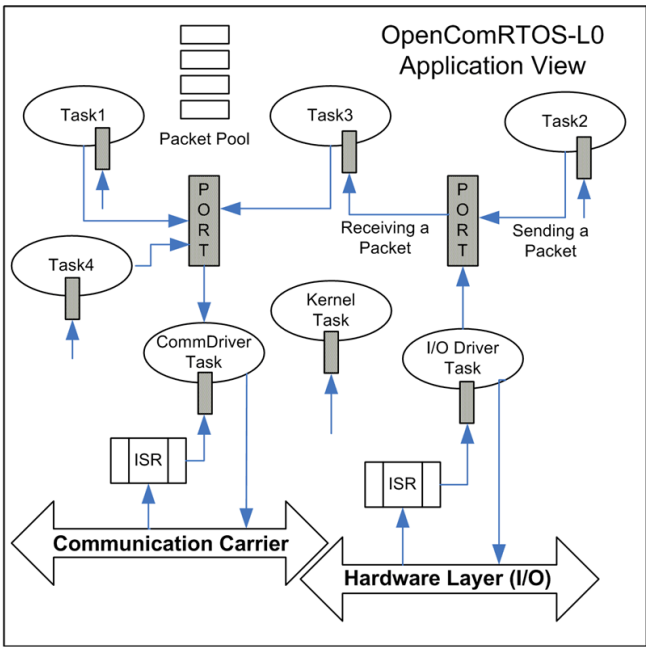

Figure 2. OpenComRTOS-L0 view

TLA, with hindsight this was an advantage vs. e.g. SPIN [13], which uses a C-like syntax. The latter leads automatically to thinking in terms of an implementation code with all its details, whereas the abstraction of TLA helps to think in more abstract terms. This also highlights the importance of specifying first before implementation is started.

A final observation is that using formal modelling techniques turned out to be a much more creative process than the mathematical framework suggests. TLA/TLC as such was primarily used as an architectural design tool, aiding the team in formulating ideas and testing them in a rather abstract way. This proved to be teamwork with lots of human interaction between the team members. The formal verification of the RTOS itself was basically a side-effect of building and running the models. Hence, this project has shown how a combination of teamwork with extensive peer-review, formal modelling support and a well defined goal can result in a "correct-by-design" product.

2.4. Novelties in the architecture

OpenComRTOS has a semantically layered architecture, Table 1 provides an overview over the available services at the different levels. At the lowest level the minimum set of Entities provides everything that is needed to build a small networked real-time application.

The Entities needed are Tasks (having a private function and workspace), and Interacting Entities, called Ports, to synchronize and communicate between the Tasks (see Figure 2). Ports act like channels in the tradition of Hoare's CSP [14], but they allow multiple waiters and asynchronous communication.

One of the Tasks is a Kernel Task which schedules the other Tasks in order of priority and manages Port-based services. Driver Tasks handle inter-node communication. Pre-allocated as well as dynamically allocated Packets are used as carriers for all activities in the RTOS, such as: service requests to the kernel, Port synchronization, data-communication, etc. Each Packet has a fixed size header and data payload with a user defined but global data size. This significantly simplifies the Packet management, particularly at the communication layer. A router function also transparently forwards Packets in order of priority between the network nodes.

In the next semantic level services and Entities were added, similar to those which can be found in most RTOSs: Boolean events, counting semaphores, FIFO queues, resources,

memory pools, etc. The formal modelling leads to the definition of all these Entities as semantic variants of a common and generic entity type. We called this generic entity a "Hub". In addition, the formal modelling also helped to define "clean" semantics for such services, whereas ad-hoc implementations often have side-effects. Table **??** summarises the semantics.

Layer	Available Entities
L0	Task, Port
L1	Task, Port, Boolean Event, Counting Semaphore, FIFO Queue, Resource, Memory Pool
L2	Mobile Entities: all L1 entities moveable between Nodes.

Table 1. Overview of the available Enties on the different Layers

L1 Entity	Semantics
Event	Synchronisation on a Boolean value.
Counting Semaphore	Synchronisation with counter allowing asynchronous signalling.
Port	Synchronisation with exchange of a Packet.
FIFO queue	Buffered communication of Packets. Synchronisation when queue is full or empty.
Resource	Event used to create a logical critical section. Resources have an owner Task when locked.
Memory Pool	Linked list of memory blocks protected with a resource.

Table 2. Semantics of L1 Entities

Services variants	Synchronising Behavior
"Single-phase" services	
_NW	Non Waiting: when the matching filter fails the Task returns with a RC_Failed.
_W	Waiting: when the matching filter fails the Task waits until such events happens.
_WT	Waiting with a time-out. Waiting is limited in time defined by the time-out value.
"Two-phase" services	
_Async	Asynchronous: when the entity is compatible with it, the Task continues independently of success or failure and will resynchronize later on. This class of services is called "two-phase" services.

Table 3. Service synchronization variant

The services are offered in a non-blocking variant (_NW), a blocking variant (_W), a blocking with time out variant (_WT), and an asynchronous variant (_A) for services where this is applicable (currently in development). All services are topology transparent and there is no restriction in the mapping of Task and kernel Entities onto this network. See Tables 2 and 3 for details on the semantics.

Using a single generic entity leads to more code reuse, therefore the resulting code size is at least 10 times less than for an RTOS with a more traditional architecture. One could of course remove all such application-oriented services and just use Hub based services. Unfortunately, this has the drawback that services loose their specific semantic richness, e.g. resource locking clearly expresses that the Task enters a critical section in competition with other Tasks. Also erroneous runtime conditions, like raising an event twice (with loss of the previous event), are easier to detect at application level compared with the case when only a generic Hub is used.

During the formal modelling process, we also discovered weaknesses in the traditional way priority inheritance is implemented in most RTOSs. Fortunately, we found a way to reduce the total blocking time. In single processor RTOS systems this is less of an issue, but

in multi-processor systems, all nodes can originate service requests and resource locking is a distributed service. Hence, the waiting lists can grow longer and lower priority Tasks can block higher priority ones while waiting for the resource. This was solved by postponing the resource assignment until the rescheduling moment. Finally, by generalization, also memory allocation has been approached like a resource locking service. In combination with the Packet Pool, this opens new possibilities for safe and secure memory management, e.g. the OpenComRTOS architecture is free from buffer overflow by design.

For the third semantic layer (L2), we plan to add dynamic support like mobility of code and of kernel Entities. A potential candidate is a light-weight virtual machine supporting capabilities as modelled in pi-calculus [15]. This is the subject of further investigations and will be reported in subsequent papers.

2.5. Inherent safety support

By its architecture the L1 semantic layers are all statically linked, hence an application specific image will be generated by the tool-chain during the compilation process. As we don't consider security risks for the moment, our concern is limited to verifying if the code is inherently safe.

A first level of safety is provided by the formal modelling approach. Each service is intensely modelled and verified with most "corner cases" detected during design time prior to writing code. A second level is provided by the kernel services themselves. All services have well defined semantics. Even when they are asynchronously used, the services become synchronous when resources become depleted. At such moments, a Task is forced to wait, which allows other Tasks to proceed and free up resources (like Packets, buffer space, etc.); Hence, the systems becomes "self-throttling". A third level is provided by the data structures, mostly based on Packets. All single-phase services use statically allocated Packets which are part of the Task context. These Packets are used for service requests, even when going across processor boundaries. They also carry return values. For two phase services Packets must be allocated from a Packet Pool. When the Pool is empty, the system will start to throttle until Packets are released. Another specific architectural feature is the fact that the system can never run out of space to store requests because there is a strict limit of how many requests there can be in the system (the number of packets). All queues are represented by linked list, and each packet contains the necessary header information, therefore no buffers are required to handle requests, which therefore cannot overflow. In the worst case, the application programmer defined insufficient Packets in the Pool and the buffers will stop growing when all Packets are in use. A last level is the programming environment. All Entities are defined statically, so they are generated together with all other system level data structures by a tool, hence no Entities can be created at runtime. Of course, dynamic support at L2 will require extra support. However, this can only be achieved reliably with hardware support, e.g. to provide protected memory spaces. The same applies to using stack spaces. In OpenComRTOS interrupts are handled on a private and separate stack, so that the Task's stack spaces are not affected. On the MLX16 such a space can be protected, but it is clear that such an inexpensive mechanism should become the norm for all embedded processors. A full MMU is not only too complex and too large, it is also simply not necessary. The kernel has various threshold detectors and provides support for profiling, but the details are outside the scope of this paper.

3. OpenComRTOS on Embedded Targets

Porting OpenComRTOS to the Microblaze soft processor was the first major work done by Altreonic. One reason for choosing the Microblaze CPU as a first target was the prior experience of the new team with the Microblaze environment [16,17]. This section compares the

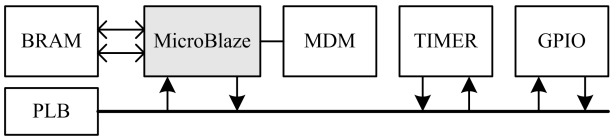

Figure 3. Hardware setup of the test system

Microblaze port with the port of OpenComRTOS to the MLX16. It also gives performance and code size figures for other available ports of OpenComRTOS.

The Microblaze soft processor is realised in a Field Programmable Gate Array (FPGA). FPGAs are emerging as an interesting design alternative for system prototyping and implementation for critical applications when the production volume is low [18]. We realised the target architecture with the Xilinx Embedded Developer Kit 9.2 and synthesized with Xilinx ISE version 9.2 on an ML403 board with a Virtex-4 XC4VFX12 FPGA clocked at 100 MHz. Our architecture, shown in Figure 3, is composed of one MicroBlaze processor connected to a Processor Local Bus (PLB). The PLB enables accessing TIMER and GPIO. The TIMER is used to measure the time it takes to execute context switches. The GPIO was used for basic debugging. The processor uses local memory to store code and data of the Task it runs. This memory is implemented through Block RAMs (BRAMs). The MicroBlaze Debug Module (MDM) enables remote debugging of the MicroBlaze processor.

3.1. Code size figures

This section reports the code size figures of OpenComRTOS on the MicroBlaze target. To put these figures into perspective we did two things. First the OpenComRTOS code size figures on the MicroBlaze target are compared with the ones on the other targets. To this point we have ported OpenComRTOS to the MicroBlaze processor from Xilinx [19], the Leon3 as used by ESA[20], the ARM Cortex-M3[21], and the XMOS XS1-G4[22]. The second comparison is concerned with the code size figures for a simple semaphore example. This example has been implemented using a) Xilinx Micro-Kernel (XMK) and b) OpenComRTOS. The later example is more important, because we can show that the OpenComRTOS version uses only 75% code size compared to the XMK version, to achieve the same functionality.

Table 4 reports the code size figures for individual L1 Services for all different targets we support. The total code size of 'Total L1 Services' is just the sum of the individual code sizes. The Service 'L1 Hub shared' represents the code necessary to achieve the functionality of the Hub, upon which all other L1 Services depend. This explains why adding the Port functionality requires only 4-8 Bytes more code.

In general the code size figures are lower for the MLX16, ARM-Cortex-M3 and XMOS due to their 16bit instruction set. Both Microblaze and Leon3 in contrast use a 32bit instruction set. Even among the targets with 16bit instruction sets we can see vast differences in the code size. One reason for this is the number of registers these targets have. The MLX16 has only four registers which need to be saved during a context switch. In contrast the XMOS port has to save 13 registers during a context switch. This has also an impact on the performance figures, which are shown in Table 6 on page 10

3.1.1. Comparing OpenComRTOS against the Xilinx Micro-Kernel

The Xilinx Micro-Kernel (XMK) is an embedded operating system from Xilinx for its Microblaze and PPC405 cores. In this section we compare the size of a comparable application example between OpenComRTOS and XMK for the Microblaze target. A complete comparison of code size figures between XMK and OpenComRTOS is not possible, because these operating systems offer different services. However, to give an indication of the code size efficiency, we implemented a simple application based on two services both OS offer. Figure

Service	MLX16	MicroBlaze	Leon3	ARM	XMOS
L1 Hub shared	400	4756	4904	2192	4854
L1 Port	4	8	8	4	4
L1 Event	70	88	72	36	54
L1 Semaphore	54	92	96	40	64
L1 Resource	104	96	76	40	50
L1 FIFO	232	356	332	140	222
L1 PacketPool	NA	296	268	120	166
Total L1 Services	1048	5692	5756	2572	5414

Table 4. OpenComRTOS L1 code size figures (in Bytes) obtained for our different ports

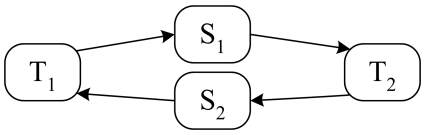

Figure 4. Semaphore loop example project

4 shows two tasks (T_1 T_2), which exchange messages and synchronise on two semaphores (S_1, S_2), in both cases 1KiB stack was used. Table 5 shows that the complete OpenComRTOS program requires about 15% less memory when compared with the memory usage of XMK. This is an important result, because with OpenComRTOS there is more RAM available for user applications. This is particularly important when, either for speed reasons or for PCB size constraints, the complete application has to run in internal (BRAM) memory.

OS	.text	.data	.bss	total
XMK	12496	348	7304	20148
OpenComRTOS	9400	1092	6624	17116

Table 5. XMK vs. OpenComRTOS code size in Bytes

3.2. Performance figures

The performance figures were evaluated by measuring the loop time. We define this loop time as the time a particular target takes to complete one loop in the semaphore loop example. The resulting measurement values allow us to compare the performance of OpenComRTOS on different target platforms.

OpenComRTOS abstracts the hardware from the application programmer, therefore the application source code, which is executed by the individual targets, stays the same. To show how compact OpenComRTOS application code is, Listings 1 and 2 show the source code for the Semaphore loop example which was used to measure the loop time figures.

Listing 1 shows the code for task `T1` which represents T_1. The `Arguments` of the function call are not used. Line 2 defines 3 variables of type 32 bit unsigned int. All the work is done wihtin the infinite loop, starting from Line 3. In Line 4 the number of elapsed processor cycles is stored in the `start` variable. The code block from Line 8 to 8 signals semaphore 1 (`S1`) 1000 times and tests semaphore 2 (`S1`) also 1000 times; for the semantics of `L1_SignalSemaphore` and `L1_TestSemaphore` see Table 2. In Line 9 the elapsed processor cycles are stored in the `stop` variable.

For completeness, Listing 2 shows the source code for `T2` which represents T_2. Similarly to `T1` the task is represented by a function whose parameters are not used. Within the while-loop, from Line 2 to 5, semaphore `S1` is tested before semaphore `S2` is signaled. Both calls are blocking, as indicated by the postfix '`_W`', see Table 3.

```
void T1 (L1_TaskArguments Arguments){
   L1_UINT32 i=0, start=0, stop=0;
   while(1) {
      start = L1_getElapsedCycles();
      for (i = 0; i < 1000; i++){
         L1_SignalSemaphore_W(S1);
         L1_TestSemaphore_W(S2);
      }
      stop = L1_getElapsedCycles();
   }
}
```

Listing 1: Souce code for task `T1`

```
void T2 (L1_TaskArguments Arguments){
   while(1) {
      L1_TestSemaphore_W(S1);
      L1_SignalSemaphore_W(S2);
   }
}
```

Listing 2: Souce code for task `T2`

After having obtained the `start` and `stop` values for all the targets we use the following Equation to calculate the loop time.

$$\text{Loop time} = \frac{\text{stop} - \text{start}}{\text{Clock speed} \times 1000} \tag{1}$$

This equation does not take into account the overhead from getting the elapsed clock cycles and from the loop implementation. This overhead is negligible compared with the processing time for signalling and testing the semaphores. Table 6 reports the measured loop times for the different targets.

	MLX16	MicroBlaze	Leon3	ARM	XMOS
Clock speed	6MHz	100MHz	40MHz	50MHz	100MHz
Context size	4×16bit	32×32bit	32×32bit	16×32bit	14×32bit
Memory location	internal	internal	external	internal	internal
Loop time	$100.8\mu s$	$33.6\mu s$	$136.1\mu s$	$52.7\mu s$	$26.8\mu s$

Table 6. OpenComRTOS loop times obtained for our different ports

The loop times expose the differences between the individual architectures. What sticks out is the performance of the MLX16[1], which despite its low Clock speed of only 6MHz is faster than the Leon3 running at more than 6 times the Clock frequency. One of the main reasons for this is that the MLX16 has only to save and restore 4 16bit registers during a context switch compared to 32 32bit registers in case of the Leon3. Furthermore, the Leon3 uses only external memory, whereas all other targets use internal memory.

[1] Stripped down version of OpenComRTOS

4. Conclusions

The OpenComRTOS project has shown that even for software domains which are often associated with 'black art' programming, formal modelling works very well. The resulting software is not only very robust and maintainable but also respectably compact and fast. It is also inherently safer than standard implementation architectures. Its use however must be integrated with a global systems engineering approach, because the process of incremental development and modelling is as important as using the formal model checker itself. The use of formal modelling has resulted in many improvements of the RTOS properties. The previous section analysed two distinct RTOS properties. Namely, code size and speed measurements. With a code size as low as 1kiB a stripped down version of OpenComRTOS fits in the memory of most embedded targets. When more memory is available, the full kernel fits in less than 10kiB on many targets. Compared with the Xilinx Micro-Kernel OpenComRTOS has about 75% of the code size. The loop time measurements brought out the differences between individual target architectures. In general however, the measured loop times confirm that OpenComRTOS performs well on a wide verity of possible targets.

Acknowledgments

The OpenComRTOS project is partly funded under an IWT project for the Flemish Government in Belgium. The formal modelling activities were provided by the University of Gent.

References

[1] RTCA. *DO-178B Software Considerations in Airborne Systems and Equipment Certification*, January 1992.

[2] ISO/IEC. *TR 61508 Functional Safety of electrical / electronic / programmable electronic safety-related systems*, January 2005.

[3] The International Council on Systems Engineering (INCOSE) aims to advance the state of the art and practice of systems engineering. `www.incose.org`.

[4] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS Modeling for System Level Design. In *DATE03*, page 10130, Washington, DC, USA, 2003. IEEE Computer Society.

[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR Manual*. `http://www.fsel.com/fdr2_manual.html`.

[6] The Open License Sociaty researches and develops a systematic systems engineering methodology based on interacting entities and thrustworthy components. `www.openlicensesociety.org`.

[7] Eonic Systems. *Virtuoso The Virtual Single Processor Programming System User Manual*. Available at: `http://www.classiccmp.org/transputer/microkernels.htm`.

[8] M. D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers and Transputers: Function, Performance and Applications*. IOS Press, Amsterdam Netherlands, 1993.

[9] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)*, volume 1, pages 51–57. CSREA, CSREA Press, jun 2000.

[10] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] The MISRA Guidelines provide important advice to the automotive industry for the creation and application of safe, reliable software within vehicles. `http://www.misra.org`.

[12] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft Approach to Systems Verification. In *VSTTE 2008*, Toronto, Canada, 2008.

[13] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[14] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[15] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[16] Bernhard Sputh, Oliver Faust, and Alastair R. Allen. Portable csp based design for embedded multi-core systems. In *Communicating Process Architectures 2006*, sep 2006.

[17] Bernhard Sputh, Oliver Faust, and Alastair R. Allen. A Versatile Hardware-Software Platform for In-Situ Monitoring Systems. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 299–312, jul 2007.

[18] Antonino Tumeo, Marco Branca, Lorenzo Camerini, Marco Ceriani, Matteo Monchiero, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. A dual-priority real-time multiprocessor system on fpga for automotive applications. In *DATE08*, pages 1039–1044, 2008.

[19] Xilinx. *MicroBlaze Processor Reference Guide*. `http://www.xilinx.com`.

[20] Gaisler Research AB. *SPARC V8 32-bit Processor LEON3 / LEON3-FT CompanionCore Data Sheet*. `http://www.gaisler.com/cms/`.

[21] ARM. *An Introduction to the ARM Cortex-M3 Processor*. `http://www.arm.com/`.

[22] XMOS. *XS1-G4 Datasheet 512BGA*. `http://www.xmos.com/`.