DIRECTOR'S BRIEF

REAL TIME OPERATING SYSTEMS

März 2010

OpenComRTOS: Formally developed RTOS for Heterogeneous Systems

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 dependable component with a clean architecture that delivers high performance on a wide variety of networked embedded systems, ranging from a single processor to distributed systems. The result is a scalable reliable 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. This paper reports code size and preformance figures of the OpenComRTOS on these processors.

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. The software is rarely proven to be correct even though this is possible with formal model checkers [3]. In the context of a unified systems engineering approach [4] 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.

The history of this project goes back to the early 1990's when a distributed real-time RTOS called Virtuoso (Eonic Systems) [5] was developed for the INMOS transputer.

In Kooperation mit:

From Deep Space to Deep Sea

Altreonic

DIRECTOR'S BRIEF

A Rollercoaster can be fun. Was it IEC-61508 Safety Certified?

It became straightforward to provide services that operate in a transparent way across processor boundaries.

This processor 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:

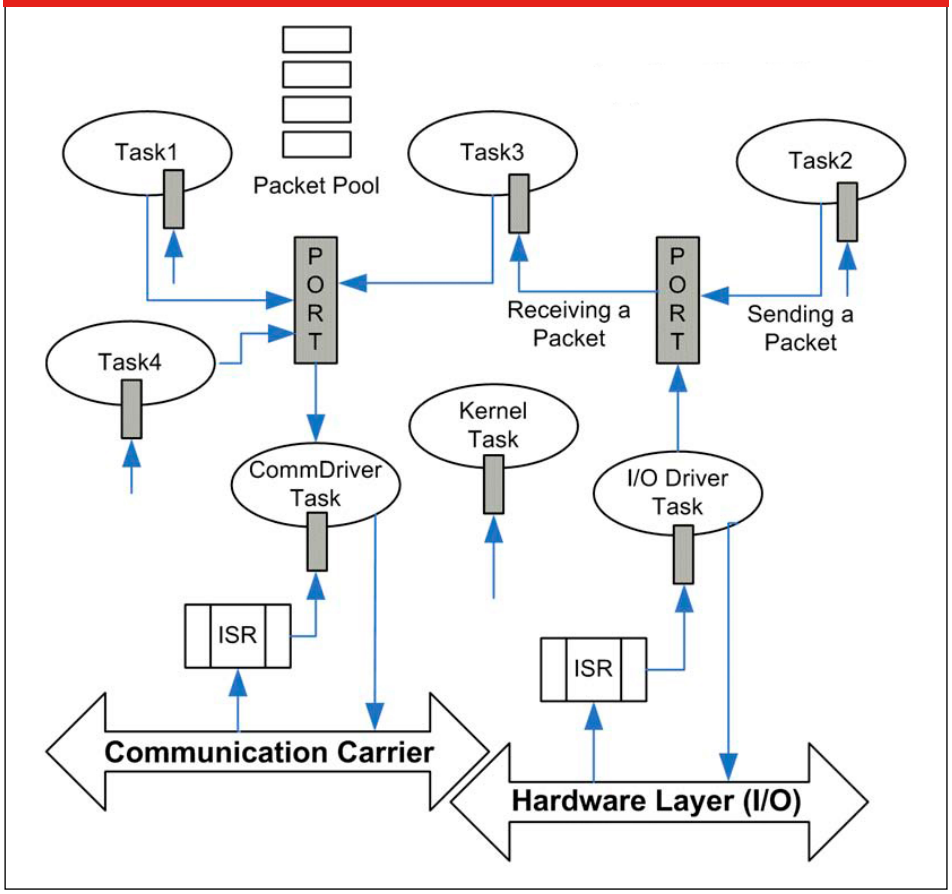
- **Scalability:** The RTOS should support very small single processor systems, as well as widely distributed processing systems interconnected through external networks like the internet. To achieve that, the software components must be independent of the execution environment. In other words, it must 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.

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 was 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 Port. Two tasks rendezvous by one task sending a 'put' request and the other task sending a 'get' request to the Port. Hence, Tasks can synchronise and communicate using Packets and Ports. Hence, it becomes straightforward to provide services that operate in a transparent way across processor boundaries.

FIGURE 1: OpenComRTOS-L0 Application View



At the next semantic level we added more traditional RTOS services like events, semaphores, etc (see Table 2 on Page 4 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.

2.1 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 1). Ports act like channels in the tradition of Hoare's CSP [6], 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. The priority of a Packet is the same as the priority of the Task from which the Packet originates.

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 2 summarises the semantics.

TABLE 1: Overview of the available Entities on the different Layers

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

TABLE 2: Semantics of L1 Entities

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.

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.

TABLE 3: Service synchronization variant

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	
_A	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.

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 OpenCom-RTOS architecture is free from buffer overflow by design.

3. OPENCOMRTOS ON EMBEDDED TARGETS

Porting OpenComRTOS to the Microblaze soft processor was the first major work done by Altreonic. This section compares the Microblaze port with the port of OpenComRTOS to the MLX16. It also gives performance and code size figures for other available ports of OpenComRTOS.

3.1 Code size figures

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.

TABLE 4: OpenComRTOS L1 code size figures (in Bytes) obtained for our different ports

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**

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 5 on page 6.

LISTING 1: Source code for task T1

```
    void T1 (L1_TaskArguments Arguments){
2     L1_UINT32 i=0, start=0, stop=0;
      while(1) {
4       start = L1_getElapsedCycles();
        for (i = 0; i < 1000; i++){
6         L1_SignalSemaphore_W(S1);
          L1_TestSemaphore_W(S2);
8       }
        stop = L1_getElapsedCycles();
10      }
    }
```

LISTING 2: Source code for task T2

```
    void T2 (L1_TaskArguments Arguments){
2     while(1) {
        L1_TestSemaphore_W(S1);
4       L1_SignalSemaphore_W(S2);
      }
6   }
```

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 within 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.

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

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

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 5 reports the measured loop times for the different targets. Each run of the loop requires eight context switches, this is caused by the fact that the Semaphores are accessed in the

TABLE 5: OpenComRTOS loop times obtained for our different ports

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

kernel context. Therefore, any access to a Semaphore requires to switch into the kernel context and afterwards to switch back to therequesting task.

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.

The OpenComRTOS project has shown that formal modelling works very well.

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. 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.

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

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

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

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

[1] Stripped down version of OpenComRTOS

IMPRESSUM

Der „Director's Brief" ist ein Produkt der Deutsche Messe Interactive GmbH, Messegelände, 30521 Hannover

Geschäftsführer:
Dr. Michael Breyer (v.i.S.d.P.)
Tel.: +49 (511) 330.601.0
Fax: +49 (511) 330.601.08

Web: www.messe-interactive.de

Redaktionelle Mitarbeit
Oliver Häußler

Layout:
Claudia Wolff