The Asterix real-time kernel

Henrik Thane, Anders Pettersson, Daniel Sundmark

Mälardalen Real-Time Research Centre, Sweden,
henrik.thane@mdh.se

1 Introduction

This paper describes a real-time kernel, Asterix, that in a practical manner makes use of many of the recent advances made in the real-time systems research community. The basic ambition behind the development of the Asterix real-time kernel was to pack state-of the art research results into package such that it can be easily used and understood by people in the embedded systems industry. From an academic point of view the Asterix real-time kernel fulfills all the basic requirements necessary for facilitating different types of timing analyses. For a software designer this signifies that the Asterix real-time kernel has the means to satisfy engineering of real-time software in the same fashion as civil engineers make use of structural calculus when designing bridges or houses. The Asterix real-time kernel is in combination with its support environment in a unique position to provide the embedded systems industry with a development kit that can increase the reliability, safety, and testability of their applications with several magnitudes compared to existing development systems.

From the outset of the development project we decided that the kernel would be distributed as an open source program. For a customer this has several benefits: nothing can be cheaper than free, and risks taken by relying on a small company for providing a real-time kernel can be minimized by having access to the source code. In summary, the kernel packs state-of-the art features into a package that is all free and open.

Although the Asterix real-time kernel defines the state-of-the-art with respect to other real-time kernels its greatest strengths lies in its open platform and its support by extremely powerful development, and verification tools.

Key features of the Asterix kernel are:

- **The execution strategies.** The Asterix real-time kernel handles execution strategies ranging from strictly statically scheduled systems via fixed priority scheduled systems to event-triggered systems, or any combination of them.
- **Memory consumption.** From an industrial point of view we have during the design of the kernel considered memory consumption and minimized the kernel and the application memory footprints.
- **Monitoring support**. Built in monitoring support makes it possible to use of state-of-the art testing, and debugging tools like deterministic testing [14] and deterministic replay debugging [15] and visualization. The kernel also provides facilities for measuring execution times of tasks with a high resolution.
- **Wait and Lock-free communication.** The kernel supports a communication type, using buffers, that decreases the need for explicit synchronization using e.g., semaphores. As a consequence blocking times can be reduced as well as increasing the analyzability of the entire real-time system [3][6].
- **Execution time jitter reduction**. The kernel provides a mechanism for minimizing the execution time jitter of individual tasks as well as the jitter originating from the kernel it self. This has been shown to increase the testability of the application enormously, since all executions of the target system will be reproducible [14]. It is also very important for control applications in general to minimize the jitter.
- **Compiling kernel**. The Kernel is compiling which means that the kernel is very resource efficient in terms of allocating memory only necessary for a set of tasks. This means that if the target system only contains 5 tasks, data is only allocated for 5 tasks. If the system contains 127 tasks data is only

allocated for 127 tasks. The obvious benefits are that we minimize the overhead, both in memory and in execution time jitter. This overhead is otherwise inherent to all kernels that handle an arbitrary number of application tasks.

- **Exception handling.** The kernel has a well defined exception handling architecture, with different layers of abstraction, separating temporal and functional error handling, as well as system level and user level exceptions.
- **Formally verified.** The kernel design is also in the process of being formally verified. This means that we will be able to provide customers with verified versions of the Asterix real-time kernel.
- **Portable.** The kernel design is such that it will be easy to port to any processor. We will provide test suites for validation of new implementations.
- **Analyzable.** Applications which employs the services provide by the Asterix real-time kernel will be possible to analyze with respect to temporal behavior, synchronization correctness, and proper use of communication mechanisms. This analyzability is not only of importance when developing safety critical application it is also a desirable property which can be used to shorten the time to market. The reason is that the Asterix kernel enables analysis of designs in an early phase of a project and thereby avoids costly and time-consuming re-designs in a late phase of the project because of lack of computational resources.

Document outline. We will in the remainder of this document describe the key features of the Asterix real-time kernel in more detail. We begin with the execution strategy, and continue with monitoring mechanisms, and jitter reduction.

2 The Asterix execution strategy

The execution strategy of the Asterix real-time kernel is based on fixed priority scheduling with support for preemption. This means that the kernel is multitasking, i.e., multiple tasks can share the same computing resource (CPU), where the task with the highest priority executes. Preemption, or task interleaving, means that an executing task can be preempted during its execution by another task, and then allowed to resume after the completion of the preempting task.

The kernel supports both periodic time triggered tasks and aperiodic event triggered tasks. All tasks are of terminating character, which means that when a task has completed its execution, it terminates and waits until a new period or a new event occurs. This has the benefit of decreasing the coupling in the system by abstracting the reactivation of the tasks away from the source code. The responsibility for reactivation is left to the kernel and the schedule where analyses are easier to apply.

Table 2-1 A static schedule for a period/LCM of 400 ms.

Task	*Period*	*r*	*priority*	*Deadline*
A	400	0	4	100
B	400	40	3	400
C	400	40	2	400
A	400	100	4	200
A	400	200	4	300
A	400	300	4	400
D	400	350	1	400

A task in Asterix is defined by its:

- **P** – *Priority* (which is unique)
- **T** – *Periodtime*. For aperiodic tasks the periodtime is not defined.
- **O** – *Offset*. A periodic task can delay its reactivation with an offset relative its periodtime.
- **BCET and WCET**. The best case and worst case execution time.
- **DL** – *Deadline.*

Depending on how we define these task attributes the Asterix real-time kernel can be configured to run a static schedule, i.e., a predefined timetable [18] (Table 2-1), or periodic tasks according to Fixed Priority Scheduling [1] (Table 2-2). A system can also be defined as event triggered by not giving period times of the tasks and by defining activator signals. Due to this general execution strategy, we can mix any of the different execution paradigms. We can for example have statically scheduled systems where some tasks are event triggered or fixed priority scheduled. The approach of assigning priorities to tasks even for statically scheduled timetables gives robustness since lower priority tasks cannot disrupt the execution of higher priority tasks if they fail and run berserk.

Table 2-2 A Fixed Priority schedule. . Same system as in table 2-2.

Task	*Period*	*r*	*priority*	*Deadline*
A	100	0	4	100
B	400	40	3	400
C	400	40	2	400
D	400	350	1	400

Every tasks in the Asterix kernel can be in four states: Waiting, Signal blocked, Ready, and Executing. The states and valid transitions between them are illustrated in Figure 2-1 and exemplified in table 2-3.

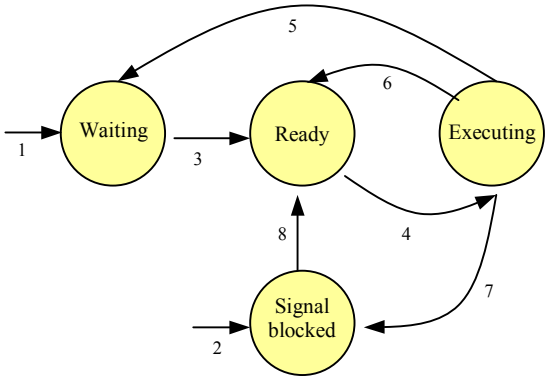

Figure 2-1. The states and transitions possible in the Safe2Run real-time kernel.

Table 2-3. The transitions in the Safe2Run real-time kernel.

Transition	*Cause*
1	Periodic task starts
2	Aperiodic task starts
3	A task is ready to execute due to period start
4	The scheduler decides to start the highest priority task available in ready and Executing.
5	When a periodically executing task has terminated
6	When the executing task is preempted.
7	When an executing task is suspended until a signal occurred
8	When an aperiodic task is triggered by a signal.

2.1.1 Synchronization

In the Asterix real-time kernel we can synchronize tasks in two distinct ways: on-line, in the source code, using semaphores and signals, and off-line, in the schedule, using offsets. The decision to allow both types of synchronization mechanisms was that they have mutually exclusive benefits depending on the type of problem to be solved. That is for certain problems the solution or analysis would be more complex if we used e.g., offsets than semaphores and vice versa. In the Asterix real-time kernel we allow both types of synchronization to be used at the same time.

For multitasking systems the use of semaphores is notorious due to the possibility of deadlocks and starvation caused by priority inversion. Semaphores have proven to cause many intricate problems and elusive bugs. However, depending on the actual algorithm used to implement the semaphore synchronization mechanism we can eliminate deadlock and starvation situations. This can be achieved by using priority-ceiling algorithms. In the Asterix real-time kernel we have implemented a very simple, memory conservative and predictable algorithm, the Immediate Inheritance Protocol. Another benefit of this protocol in conjunction with the terminating character of all tasks is that we can make use of a single stack, and thus decrease memory use.

2.1.2 Communication

In the Asterix real-time kernel we provide a mechanism called wait and lock-free communication (WLFC). This type of shared memory communication allows tasks to communicate with each other without any blocking, that is, the need for explicit synchronization using e.g., semaphores is eliminated. The shared memory communication is of simplex type and based on a set of buffers that can be read by a set of tasks and written to by *one* task. The reading tasks are guaranteed that the value they read is nonvolatile during their execution, i.e., atomicity is guaranteed. The writing task is also guaranteed a free buffer for writing. The wait-free in WLFC means that a reading task does not have to wait for the latest produced value; it will always be available for the receiver. When a task *A*, starts to execute it is given a reference to the latest written value by a producer, *P*. This value is guaranteed to be unmodified during the execution of task *A*, even if it is preempted by the writer, *P*, and *P* produces a new value. However if now a second reading task, *B*, preempts *A* after *P* has written a new value, task *B* will at its start receive a reference to this new latest value.

The cost for this type of communication is memory. The number of buffers need for each wait and lock-free communication channel is *no_buffers* = *no_readers* + 2. If a software designer feels that the memory needed for WLFC is too costly then the designer can resort to shared memory and use semaphores for synchronization. Another option is if the communicating tasks run with the same periodicity then we can make use of offsets for synchronization and just use one memory buffer.

Wait and lock-free communication is superior for systems where communication between

asynchronous/multi-rate tasks occur [3][6]. In addition WLFC gives decreased blocking times and reduced scheduling complexity. All data transfer is also performed by the tasks themselves, not by the kernel, which decreases kernel overhead and jitter. The data transfer overhead is debited to the tasks involved in the transaction, and therefore subject to execution time estimation.

2.1.3 Hard and soft tasks

The tasks in the Asterix real-time kernel are divided into two categories: Hard tasks and soft tasks. The difference between the tasks are that the hard tasks are required to have passed a schedulability analysis (see Section 2.1.5), while the soft tasks have no such requirement. Since we can mix both types of tasks in a system we must guarantee that the soft tasks cannot disrupt the execution of the hard tasks. This guarantee is fulfilled by statically assigning priorities to soft tasks that are all lower than the priorities of the hard tasks. In order to guarantee that no soft task can block a hard tasks the semaphore mechanism is devised such that the set of semaphores used by the soft tasks are disjunct with the set of semaphores used by the hard tasks. These sets are defined off-line, and are actually required in order to set the correct priority ceilings in the immediate priority inheritance protocol. If a soft task under suspicious circumstances still would access a hard semaphore an exception handler would be invoked and the situation would be detected.

2.1.4 Pre-runtime configuration

As the Asterix real-time kernel is compiling we need a means to specify the constitution of the system and to initialize all data structures describing the tasks and their attributes. This is done in a configuration file, which upon execution outputs the necessary data structures that can be complied together with the application code and the kernel. Figure 2-2 illustrates a configuration file.

```
SYSTEMMODE = NORMAL;
RAM = 512000;
MODE mode_1{
   RESOLUTION = 1000;
   HARD_TASK ht_1{
                        ACTIVATOR       = 100; //period time
                        OFFSET          = 0;
                        DEADLINE        = 50;
                        PRIORITY        = 10;
                        STACK           = 50;
                        ROUTINE         = ht_1_routine;
                        ARGUMENTS       = "1, 2, 3";
                        ERR_ROUTINE     = ht_1_error_routine;};
   HARD_TASK ht_2{
                        ACTIVATOR       = 50; //period time
                        OFFSET              = 0;
                        DEADLINE        = 20;
                        PRIORITY        = 20;
                        STACK               = 50;
                        ROUTINE         = ht_2_routine;};
   SOFT_TASK st_1{
                        ACTIVATOR       = sig_1; //trigger signal
                        OFFSET              = 0;
                        DEADLINE        = 10;
                        PRIORITY        = 10;
                        STACK               = 50;
                        ROUTINE         = st_1_routine;};
   WAITFREE w_1{
                        WRITER          = ht_1;
                        READER          = ht_2;
                        NUM_BUF         = 3;
                        TYPE            = "my_type";};
   SIGNAL sig_1{
                        USER            = ht_1;
                        USER            = st_1;};
   SEMAPHORE sem_1{
                        USER            = ht_1;
                        USER            = ht_2;};
};
```

Figure 2-2. The Configuration of a system.

2.1.5 Timing analysis

Timing analysis is performed at two levels, the task level and the system level.

At the task level the worst case execution time for each task is analyzed or estimated. This analysis is comparably simple on Asterix compared to do on tasks running on traditional RTOS such as WxWorks and QNX since Asterix requires terminating tasks. Further, since a task cannot be blocked after it has entered the state execution (only pre-empted), the worst case execution time can be calculated or measured for the code of task in isolation.

At system level we analyze if the composed system fulfil its timing requirements by using either fixed priority analysis or a pre-runt-time scheduler. Both kind of analysis is mature and proven to be useful in industrial applications [19][20].

When designing a system we can assign time budgets to the tasks that are not implemented by intelligent guesses based on experience. By doing this we gain two positive effects. First, the system level timing analysis can be done before implementation and hence we have a tool for estimating the performance of the system. Second, the time budgets can be used as an implementation requirement.

By applying this approach we make the design process less adhoc with respect to real-time performance. That is, the first time one can find timing problems in traditional system design is when the complete system or subsystem has been implemented. If a timing problem is found adhoc optimization starts which most surely will make the system difficult to maintain.

3 Monitoring

In the Asterix real-time kernel we have unique support for observations of the target application. These observations can be used for execution time measurements of the application tasks, but most significantly these observations can be used for visualization, deterministic replay debugging and deterministic testing of the target system.

3.1 Deterministic replay

Deterministic replay is a software based technique proprietary to the Asterix development environment that allows reproducible debugging of single tasking, multi-tasking, and distributed real-time systems [15]. During runtime, information is recorded with respect to interrupts, task-switches, timing, and data. The system behavior can then be deterministically reproduced off-line using the recorded information. A standard debugger can be used without the risk of introducing temporal side effects, and we can reproduce interrupts, and task-switches with a timing precision corresponding to the exact machine instruction at which they occurred. The technique also scales to distributed real-time systems, so that reproducible debugging, ranging from one node at a time, to multiple nodes concurrently, can be performed.

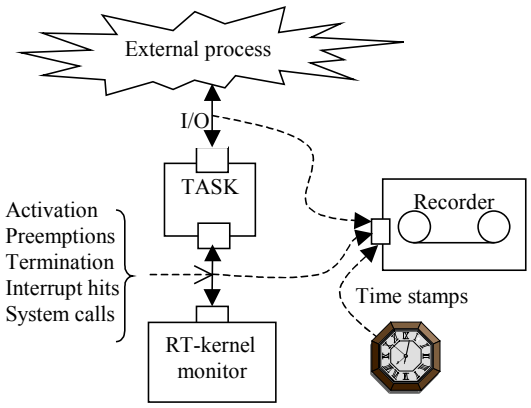

Figure 3-1. Kernel monitoring and recording.

3.2 Deterministic testing

For testing of sequential software it is usually sufficient to provide the same input (and state) in order to reproduce the output. However, for real-time systems it is not sufficient to provide the same inputs for reproducibility – we need also to control, or observe, the timing and order of the inputs and the concurrency of the executing tasks. Based on the monitoring mechanisms built into the Asterix real-time kernel and a proprietary testing method found in the Asterix development environment deterministic testing of the target application can be easily performed. The testing method includes an analysis technique that given a set of tasks and a schedule derives all execution orderings that can occur during run-time [14]. The method also includes a testing strategy that using the derived execution orderings can achieve deterministic, and even reproducible, testing of real-time systems. Each execution ordering can be regarded as a sequential program and thus techniques used for testing of sequential software can be applied to real-time system software. The analysis and testing strategy can also be extended to encompass interrupt interference, distributed computations, communication latencies and the effects of global clock synchronization.

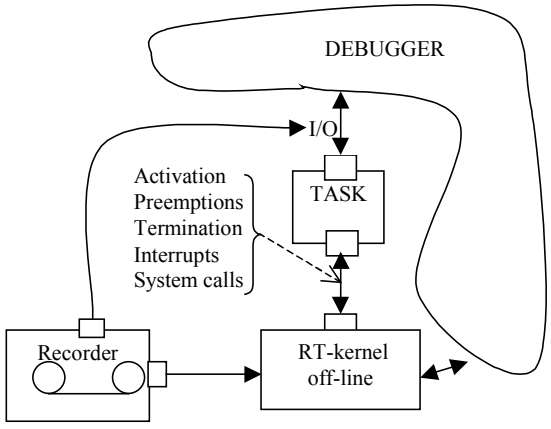

Figure 3-2. Offline kernel with debugger.

4 Jitter reduction

The kernel provides a mechanism for minimizing the execution time jitter of individual tasks as well as the jitter originating from the kernel it self. This has been shown to increase the testability of the application enormously, since all executions of the target system will be reproducible [14][13]. It is also very important for control applications in general to minimize the jitter.

Figure 4-1 illustrates the possible execution orderings for a schedule running on the Asterix real-time kernel without jitter reduction. That is all the possible task starts, task preemptions and task terminations yielded by the varying execution times of tasks and their varying start times due to delay caused by higher priority tasks. Figure 4-2 illustrates the same system with jitter reduction turned on.

From a verification perspective the fewer the scenarios the system exhibit the better the possibilities are for testing and debugging the system since the number of behaviors to consider is reduced. In fact there is an exponential relation between the jitter in the system and the number of execution scenarios. In other words the testability of a system is enormously influenced by the jitter. The potential testability, and therefore indirectly the reliability, of the system benefits greatly by the jitter reduction techniques used in the Asterix real-time kernel. Typical testability gains are in the range of billions for an industrial application with modest complexity and even more for more complex systems.

This feature makes the Asterix real-time kernel suitable for use in mission critical, and safety critical applications, or simply in applications where the funds are limited but the desire is to get more-bang-for-the-buck (reliability).

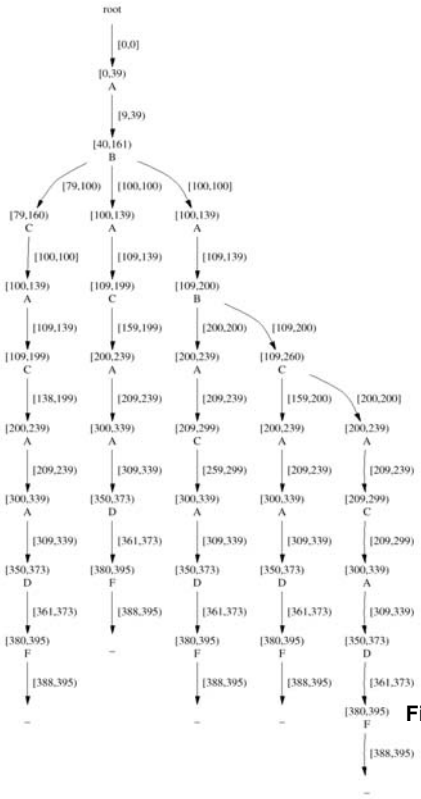

Figure 4-1. The possible execution order scenarios for a system with jitter.

Figure 4-2. No jitter.

5 Conclusions

In this paper have we presented a novel real-time kernel Asterix which support development of hard real-time systems.

We have presented the supported execution strategy, the monitoring and testing facilities, and a mechanism for jitter reduction. We are currently in the process of adopting Asterix to different micro controllers.

6 References

[1] Audsley N. C., Burns A., Davis R. I., Tindell K. W. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective.* Real-Time Systems journal, Vol.8(2/3), March/May, Kluwer A.P., 1995.

[2] Audsley N. C., Burns A., Richardson M.F., and Wellings A.J. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software, pp. 127-132, Atlanta, Georgia, May, 1991.

[3] Chen J. and Burns A. *Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus*. 10th Euromicro Workshop on Real-Time Systems, June 1998.

[4] Eriksson C., Mäki-Turja J., Post K., Gustafsson M., Gustafsson J., Sandström K., and Brorsson E. *An Overview of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, vol. 36, pp. 66-80, Oct. 1996.

[5] Joseph M. and Pandya P. *Finding response times in a real-time system.* The Computer Journal – British Computer Society, 29(5), pp.390-395, October, 1986.

[6] Kopetz H. and Reisinger J. *The Non-Blocking Write Protocol NBW: A Solution to a Real- Time Synchronization Problem*. In Proceedings of he 14th Real-Time Systems Symposium, pp. 131-137, 1993.

[7] Kopetz H. *Sparse time versus dense time in distributed real-time systems*. In the proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460-467, 1992.

[8] Kopetz H., Damm A., Koza Ch., Mulazzani M., Schwabl W., Senft Ch., and Zainlinger R.. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, (9):25-40, 1989.

[9] Kopetz H.. *Event-Triggered versus Time-Triggered Real-Time Systems*. Lecture Notes in Computer Science, vol. 563, Springer Verlag, Berlin, 1991.

[10] Lui C. L. and Layland J. W.. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

[11] Puschner P. and Koza C. *Calculating the maximum execution time of real-time programs*. Journal of Real-time systems, Kluwer A.P., 1(2):159-176, September, 1989.

[12] Sandström K., Eriksson C., and Fohler G. *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*. In proceedings of the 5th Int. Conference on Real-Time Computing Systems and Applications (RTCSA'98). October 1998, Japan.

[13] Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA'99), Hong Kong, December, 1999.

[14] Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. Proc. 20th IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

[15] Thane H. and Hansson H. *Using Deterministic Replay for Debugging of Distributed Real-Time Systems*. In proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00), Stockholm, June 2000.

[16] Thane H. *Design for Deterministic Monitoring of Distributed Real-Time Systems.* Technical report, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, 1999.

[17] Tindell K. W., Burns A., and Wellings A.J. *Analysis of Hard Real-Time Communications.* Journal of Real-Time Systems, vol. 9(2), pp.147-171, September 1995.

[18] Xu J. and Parnas D. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. 16(3):360-369, 1990.

[19] Christer Norström, Kristian Sandström, Mikael Gustafsson, Jukka Mäki-Turja, and Nils-Erik Bånkestad. *Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry*. In proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS01), Washington, US, April 2001. IEEE Computer Society.

[20] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg *Volcano a revolution in on-board communications*. Volvo Technology Report. 98-12-10.