A JITTER-FREE OPERATIONAL ENVIRONMENT FOR DEPENDABLE EMBEDDED SYSTEMS

Christo Angelov¹, Jesper Berthing² and Krzysztof Sierszecki¹

¹Mads Clausen Institute for Product Innovation, University of Southern Denmark, Grundtvigs Alle 150, 6400 Soenderborg, Denmark; ²Center for Software Innovation, Stenager 2, 6400 Soenderborg, Denmark

Abstract: The paper presents a new real-time kernel architecture featuring component-based design, as well as advanced algorithms based on Boolean vectors and bitwise vector processing of kernel data structures. These have been consistently applied to all aspects of task management, as well as task synchronization and communication. Hence, the execution time of system functions no longer depends on the number of tasks involved, resulting in predictable, jitter-free operation of kernel subsystems. This approach has been further extended to time management resulting in a new type of kernel component, which can be used to implement Distributed Timed Multitasking - a novel computational model providing for predictable jitter-free execution of hard real-time tasks as well as distributed transactions in a dynamic scheduling environment.

Keywords: Hard Real-Time Systems; Boolean Vector Processing; Distributed Timed Multitasking

1. INTRODUCTION

Modern embedded systems have to comply with stringent requirements with respect to system safety and dependability. There are essentially two approaches to task scheduling for dependable embedded systems: static scheduling vs. predictable dynamic scheduling using algorithms developed in modern Real-Time Scheduling Theory¹. The former approach is widely used with safety-critical systems in application areas such as aerospace and military systems, automotive applications, etc.^{1,2}. However, static scheduling

has a major disadvantage: its application results in closed systems that are generally difficult to re-configure and maintain. This is in contradiction to the requirement for an open and flexible system architecture that must support software reuse and reconfiguration.

The second approach is more promising but it requires the development of a new generation of so-called *safe* real-time kernels, which provide a secure and predictable environment for application tasks. That is accomplished through a number of specific features, such as predictable task scheduling, safe task interaction, extensive timing and monitoring facilities, and predictable kernel behaviour³. Unfortunately, available real-time kernels, with the exception of some experimental designs, do not meet the above requirements. In particular, most of them are implemented using linked-list queues resulting in substantial and largely varying overhead (kernel jitter), which complicates considerably task schedulability analysis^{4,5}. Therefore, it can be argued that the elimination of kernel jitter will make it possible to precisely estimate task and transaction response times early in the design process, taking into account kernel execution effects.

The above problem has been addressed in the MicroC/OS kernel where the ready task queue and event queues have been implemented as two-dimensional bit patterns⁶. This solution makes it possible to execute task insertion/deletion operations in constant time, irrespective of task priority, but that is not possible with multiple tasks (e.g. multiple tasks released simultaneously by one and the same event).

On the other hand, dynamic scheduling inherently introduces the problem of task execution jitter, which is detrimental to control system behaviour. One way of eliminating this effect is to enforce constant execution time on kernel execution paths via dummy code, and likewise - enforce task execution times within worst-case execution time bounds, as suggested for the *Asterix* real-time kernel⁷. That solution is expected to result in highly predictable and reproducible system behaviour but it could have a negative side effect, i.e. substantially increased kernel overhead and reduced system utilization. Another option is to combine static and dynamic scheduling within a hybrid real-time kernel⁸.

However, an even better approach has recently emerged, i.e. *Timed Multitasking* - a powerful yet elegant and conceptually simple computational model, which effectively eliminates task execution jitter⁹. Under that model task I/O drivers are executed atomically at precisely specified time instants (i.e. task release and deadline instants), whereas application tasks are executed in a dynamic scheduling environment. Hence, task jitter is of no consequence, as long as the tasks finish execution before their deadlines. Ultimately, timed multitasking makes it possible to engineer real-time

systems combining high flexibility inherent to dynamic scheduling and predictable operation, which is typical for statically scheduled systems.

It must be noted that this is an event-driven model, as against purely time-triggered models featuring atomic execution of task I/O drivers and split-phase execution of periodic tasks, e.g. *Giotto*¹⁰. In principle, that model accommodates all kinds of events, such as periodic timing events, aperiodic external and internal events, message arrival events, etc., and its implementation is not strictly related to a specific compiler or operating system. However, the existing implementation supports only local communication via interconnected ports implemented as shared data structures⁹. Another limitation comes from the fact that each task has to use its own set of timers (e.g. period, offset and deadline timers), which could result in considerable kernel overhead. That overhead might be substantially reduced by developing an integrated time manager capable of servicing multiple tasks simultaneously, e.g. tasks whose periods or deadlines expire at one and the same time instant.

The above problems have been addressed while developing the *HARTEX* kernel architecture, which is presented in the paper. Our approach is to eliminate linked lists and instead, use Boolean vectors (bit-strings) in order to emulate system queues³. Queues can thus be processed through bitwise Boolean operations resulting in jitter-free execution of kernel functions, independent of the number of tasks involved. This technique has been recently extended to time management resulting in a new type of kernel component, which can be used to efficiently implement Distributed Timed Multitasking - a novel computational model providing for predictable jitter-free execution of hard real-time tasks as well as distributed transactions in a dynamic scheduling environment.

The paper is structured as follows: Section 2 gives an overview of the main features of the *HARTEX* architecture. Section 3 presents jitter-free task management using Boolean vectors and bitwise vector operations on kernel data structures. Section 4 extends this approach to static time management - a newly developed technique that has been used in the latest timed-multitasking version of the kernel. Section 5 presents transparent task communication using content-oriented message addressing, in the context of both time-triggered and event-triggered distributed transactions.

2. HARTEX ARCHITECTURE

HARTEX (*HARd Real-Time Executive for Control Systems*) is a safe real-time kernel, which has been conceived as an operational environment for

component-based embedded applications conforming to the *COMDES* (formerly *ASPS*) model of distributed computation^{3,11,12}.

The *HARTEX* architecture exhibits a number of novel features, which are briefly summarized below:

- Component-based architecture supporting kernel reconfiguration and scalability
- Integrated task and resource scheduling via the System Ceiling Priority Protocol – a non-blocking synchronization protocol providing for predictable and efficient management of shared resources
- Boolean vector processing of kernel data structures, resulting in very low overhead and constant execution time of system functions, hence jitter-free operation of kernel subsystems
- Predictable jitter-free execution of real-time tasks in a distributed timed multitasking environment, using an advanced clock synchronization mechanism and a new type of time manager - the Static Time Manager
- Event notification via Boolean vector semaphores, providing for the instantaneous broadcast/multicast of events to multiple receiver tasks
- Integrated communication protocol supporting transparent content-oriented message addressing within local and/or remote interactions, including both state message and event message communication

Component-based design is an important feature of the *HARTEX* architecture. A number of reconfigurable kernel components have been specified and implemented: task managers supporting basic and/or extended tasks; dynamic and static time managers; external event manager; integrated time/event manager; task I/O manager, resource manager; software bus, etc. These can be used to configure various types of operating systems – from small kernels for stand-alone microcontrollers to distributed real time operating systems.

However, the main innovation of the architecture is the use of Boolean vector processing, which is presented in subsequent sections.

3. JITTER-FREE TASK MANAGEMENT

The *HARTEX* Task Manager provides support for both basic (non-blocking) and extended (blocking) tasks, in accordance with the state transition diagram shown in Fig. 1-a. Its ready task queue is emulated by a Boolean vector - the *Active Tasks Vector* (*ATV*) having as many bits as there are tasks in the system, whereby $ATV[i] = 1$ denotes an active task of priority i and task number i . Another vector is used to specify task operational status,

i.e. the *Enabled Tasks Vector (ETV)*: task i is enabled (disabled) if $ETV[i] = 1(0)$.

Figure 1. HARTEX state transition diagram and Boolean vector bit patterns

An active task can be in one of three substates: *ready*, *running* and *preempted*. Ready and preempted tasks are differentiated by means of the *Preempted Tasks Vector (PTV)* where $PTV[i] = 1(0)$ denotes a preempted (ready) task of the same *ATV* bit position. The running task is specified by an auxiliary variable – the *Running Task* as well as its corresponding *ATV* bit which is reset only after the task has been finished. Extended real-time tasks may have their *ATV* bit temporarily reset while being suspended. Basic tasks can never be suspended but they might be eventually preempted by higher-priority tasks.

The state encoding for various task states is shown in Fig. 1-b. It has been specifically designed to reduce the execution time of task state transition operations and corresponding task management primitives, which are effectively reduced to bit manipulation. Moreover, the use of the above encoding technique makes it possible to simultaneously execute multiple identical operations involving different tasks through bitwise operations on Boolean vectors. This is done in *constant* time, no matter how many tasks are involved in the operation, as illustrated by the primitive *release(tasks)*. Its argument is a Boolean vector specifying the tasks to be released (e.g.

vector 10101010b specifying tasks 1, 3, 5 and 7). That operation is executed as follows:

```
release(tasks)
{
    ATV = ATV OR (tasks AND ETV);
}
```

The execution time of the above primitive is in the (sub)microsecond range depending on the platform used, e.g. 4 μ s in an 8-bit *ATmega103* microcontroller running at 4 Mhz and operating on 16-bit vectors. With linked-list queues, the same operation might take from tens to hundreds of microseconds depending on the number of tasks involved even in relatively high-end processors. For instance, it has been shown experimentally that 20 periodic tasks are released in 570 μ s in an *M68020*-based computer running under the *Olympus* kernel, as against 76 μ s for a single task⁴.

Most other primitives are executed in the same manner, resulting in substantially reduced overhead and jitter-free execution, e.g. operations on Boolean vector semaphores³ and communication primitives that use such semaphores (see section 5)

Likewise, task scheduling is facilitated by the Boolean vector data structure. The task manager determines the highest priority active task to be executed by finding the highest priority non-zero bit in the *ATV*. This is done via a bit search procedure, which takes constant time to execute, and in some processors this can be accomplished with a single instruction (e.g. the *BSF* instruction in the Intel family of microprocessors). In fact, the scheduling algorithm is somewhat more involved because it implements integrated task and resource scheduling using the so-called System Ceiling Priority Protocol (known also as the Stack-Sharing Priority Ceiling Protocol¹).

Timed multitasking introduces an important extension to the conventional multitasking model, i.e. *split-phase* execution of real-time tasks, whereby task I/O drivers may be executed separately from the main task body. Specifically, task input drivers are executed when the task is released and output drivers – when the task deadline arrives, or when the task comes to an end (if no deadline has been specified). This requires the introduction of a separate kernel component – the *Task I/O Manager*, which is actually implemented as the highest priority system task.

When activated, the *Task I/O Manager* invokes all output drivers, and subsequently – all input drivers that have been registered for execution. I/O drivers are registered using two additional Boolean vectors – the *Output Drivers Vector (ODV)* and the *Input Drivers Vector (IDV)*. Specifically, input drivers are registered for execution by the corresponding task release events, e.g. periodic timing events, sporadic external and internal events, or message arrival events. Output drivers are registered for execution by

deadline timing events and in some cases by the task itself when it comes to an end (if no deadline has been specified).

Task I/O drivers are atomic pieces of code, whose execution time is negligible compared to that of the task body. These may be physical I/O drivers or communication drivers that are used to transparently exchange labeled messages (signals) between interacting local and/or remote subsystems^{11,12}. This has resulted into an extended variant of the basic timed multitasking model called *Distributed Timed Multitasking*, which is now supported by the latest version of *HARTEX* (see subsequent sections).

4. TIME MANAGEMENT FOR JITTER-FREE EXECUTION OF REAL-TIME TASKS

This section presents a kernel component - the *Static Time Manager (STM)*, which can be used to implement time-triggered transactions in the context of Timed Multitasking and its extended variant.

The Static Time Manager uses a table specifying the behaviour of a set of periodic hard real-time tasks. The latter consists of k records corresponding to superperiod time instants $0, 1, 2, \dots, k-1$. Each instant is defined with an offset from the beginning of the superperiod. Accordingly, each table entry has the following format:

{offset, tasks_with_deadline, tasks_to_release},

where *tasks_with_deadline* is a Boolean vector specifying the tasks whose deadline expires at the time instant given by *offset*, and *tasks_to_release* is another Boolean vector specifying tasks that have to be released at that same instant.

The Static Time Manager processes the scheduling table in the following manner:

```
with every clock tick do {
  update current time;
  if ( current_time < schedule[I].offset) return;
  else {
    generate a deadline violation vector if some
    tasks_with_deadline have not finished execution and
    disable those tasks;
    register the output_drivers of tasks that have
    finished execution in the Output Drivers Vector
    (ODV);
```

```

    release tasks specified by the tasks_to_release
    vector if not disabled (i.e. register those tasks in
    the Active Tasks Vector (ATV);
    register the input_drivers of released tasks in the
    Input Drivers Vector (IDV);
    I = I + 1 (mod (schedule length));
    release the Task I/O Manager;
    invoke the Task Manager;
}
}

```

The above algorithm is executed using Boolean vector processing techniques, as follows:

- A deadline violation vector (DVV) is generated, and the corresponding tasks - disabled:


```

      DVV = ATV & tasks_with_deadline;
      ETV = ETV & ~DVV;
      
```
- Output drivers are registered for execution with the task I/O Manager:


```

      tasks_finished = tasks_with_deadline & ~ATV;
      for all tasks in tasks_finished {
          ODV = ODV | task.output_drivers;
      }
      
```
- Tasks are released for execution:


```

      ATV = ATV | (tasks_to_release & ETV);
      
```
- Input drivers are registered for execution with the task I/O Manager:


```

      tasks_released = (tasks_to_release & ETV);
      for all tasks in tasks_released {
          IDV = IDV | task.input_drivers;
      }
      
```
- The Task I/O Manager is released and the Task Manager – invoked:


```

      release(taskIOManager);
      preempt();
      
```

The Task I/O Manager is treated as a high(est) priority system task. It executes the registered output drivers, then – input drivers and exits, whereupon the Task Manager invokes the next highest-priority application task registered in the ATV.

The superperiod table is processed in this manner upon successive tick interrupts, which amounts to the operation of multiple free-running timers measuring intervals such as period, offset and deadline for all tasks involved. It is obvious that the presented timing mechanism implements a static schedule with respect to timed input/output and task release actions, whose operation is largely similar to that of a rotating drum sequencer used in some

mechanical devices. Therefore, it has been denoted as the *Static Time Manager*, and alternatively – the *Drum Sequencer*.

Timed multitasking can be extended to distributed systems via signal drivers used to transparently exchange messages between local and/or remote tasks (see next section). However, a distributed application would require synchronized operation of multiple drum sequencers running concurrently in various network nodes.

5. TASK COMMUNICATION FOR DISTRIBUTED TIMED MULTITASKING

Application tasks interact by exchanging *signals*, i.e. messages having unique communication variable identifiers. That follows from the *COMDES* component model, which presumes that subsystems interact by exchanging signals via communication objects called *signal drivers*^{11,12}. The latter may be viewed as a special class of task I/O drivers within the timed multitasking model of computation. Accordingly, these can be invoked at precisely specified time instants resulting in time-triggered communication between subsystems and encapsulated tasks¹¹. Alternatively, it is possible to invoke signal drivers when a task is released by a sporadic event (input drivers) and at the end of task execution (output drivers), which is typical for event-triggered communication within phase-aligned transactions.

Signal driver operation is supported by the *HARTEX* message exchange layer³. Messages are actually exchanged by means of communication primitives, which are invoked by signal drivers in order to transparently transfer data from the source to the destination message buffers of interacting local and/or remote subsystems. Each message has a unique source buffer associated with an output signal driver (or the underlying network layer), and one or more destination buffers that are associated with the corresponding input signal drivers and receiver tasks (see Fig. 2).

HARTEX communication primitives are invoked with communication variable identifiers (names) within an implicit message addressing scheme, which is known as content-oriented message addressing, e.g. *broadcast_state_variable(TEMPERATURE)*. The variable name is subsequently used to transparently route a message to local and/or remote receivers. To that end, each message is uniquely specified by a kernel data structure - a Message Control Block (MCB) containing relevant information such as source buffer address, message length, receiver tasks, local and/or remote type of communication, blocking or non-blocking interaction, etc. In local communication the message identifier is used as an index for the

corresponding MCB, whereas in remote communication it is mapped onto a network message identifier (e.g. a CAN identifier).

Figure 2. Transparent communication via content-oriented message addressing

The message exchange layer may use the services of the underlying event notification layer in order to signal message arrival to one or more receiver tasks. However, receiver notification is needed only in event-driven communication, i.e. in sequential (phase-aligned) transactions where the receivers are released by message arrival events. In that mode of operation the input signal drivers are invoked by the Task I/O Manager, immediately after the task has been released by the message arrival event and its input drivers - registered in the *Input Drivers Vector*. Accordingly, the output drivers are invoked after the task has exited and its output drivers - registered in the *Output Drivers Vector*. This is explained below in the context of non-blocking state message communication.

The output signal driver composes the message into the corresponding source buffer and sends it off via the primitive *broadcast_state_variable(var_name)*. In case of local communication, the broadcast primitive invokes a vector semaphore operation, i.e. *signal_and_release(semaphore, receivers)*. This is used to notify receiver tasks about message arrival and release some or all of them, whereby the corresponding input drivers are registered for execution. In case of remote communication the message is queued for transmission and subsequently copied into a receiver-node source buffer. The underlying network layer is then responsible for signalling message arrival to the remote receivers, as explained above.

After message arrival notification, the message is copied from the source buffer to the corresponding destination buffer for each receiver task. That is done by the corresponding input signal driver, which is invoked by the Task I/O Manager immediately after the receiver task has been released. It is

actually accomplished with a kernel primitive invoked by the driver, i.e. *receive_state_variable(var_name, destination_buffer)*. Message copying has to be atomic, and in local communication this is guaranteed by the non-preemptive execution of signal drivers. In remote communication it might be necessary to use a protocol providing for mutual exclusion between an interrupt-driven network-layer driver and the input signal driver executing a *receive_state_variable* operation, e.g. the *NBW* protocol¹³.

Vector-semaphore signalling primitives are executed in constant time³, and on the other hand, each input driver is responsible for copying the message into its destination buffer, on per driver/task basis. That is why the broadcast primitives execute in constant time no matter how many receivers are involved.

Time-triggered communication is conceptually simpler, since receiver tasks need not be signalled. Obviously, that is not necessary because receiver tasks are released, and signal drivers - invoked by the Static Time Manager at precisely specified time instants, whereby message copying is executed within input drivers as explained above. Consequently, the Task Manager activates released tasks only after their input drivers have been executed and the corresponding messages - received (see section 4).

6. CONCLUSION

This paper has presented advanced task management and interaction techniques featuring Boolean vectors and parallel (bitwise) vector operations on kernel data structures, which have been developed for the *HARTEX* family of real-time kernels. That approach has been consistently applied to all aspects of task management and interaction: task scheduling, task execution request generation, task synchronization and communication. This has resulted in considerably reduced kernel overhead and jitter-free execution of kernel subsystems, whereby the execution time of system operations depends no longer on the number of tasks involved.

That is an outstanding feature with important implications. On the one hand, low kernel overhead results in faster task response and increased processor schedulability. On the other hand, constant duration of kernel operations makes it possible to precisely estimate task response times taking into account kernel execution effects, which will contribute to higher systems safety and predictability.

Ultimately, this technique has made it possible to efficiently implement a timed multitasking version of the *HARTEX* kernel providing for predictable jitter-free execution of hard real-time tasks in single-computer and distributed environments. Under that kernel task I/O drivers are executed

atomically at precisely specified time instants, whereas tasks are executed in a preemptive priority-driven environment and may have termination jitter. However, jitter is effectively eliminated, as long as tasks finish execution before their deadlines.

The presented techniques have been validated in a series of distributed motion control experiments, and jitter-free behaviour has been demonstrated during the execution of transactions with one-period delay from sampling to actuation, transactions with offsets as well as transactions with decoupled (pipelined) execution of transaction tasks.

REFERENCES

1. Liu, J.W.S.: *Real-Time Systems*. Prentice Hall (2000)
2. Kopetz, H.: *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers (1997)
3. Angelov, C., Ivanov, I., and Burns, A.: HARTEX - a Safe Real-Time Kernel for Distributed Computer Control Systems. *Software: Practice and Experience*, vol. 32, N3, (2002) 209-232
4. Burns, A., Tindell, K., and Wellings, A.: Effective Analysis for Engineering Real-Time Fixed-Priority Schedulers. *IEEE Trans. on Soft. Eng.*, 21, (1995) 475-480
5. Katcher, D., Arakawa, H., and Strosnider, J.: Engineering and Analysis of Fixed-Priority Schedulers. *IEEE Trans. on Soft. Eng.*, 19, (1993) 920-934
6. Labrosse, J.: *MicroC/OS-II. The Real-Time Kernel*. CMP Books (2002)
7. Thane, H., Pettersson, A., Sundmark, D.: The Asterix Real-Time Kernel. *Proc. of the 13th Euromicro International Conference on Real-Time Systems*, Delft, Netherlands (2001)
8. Isov, D., Norström, C.: Components in Real-Time Systems. *Proc. of the 8th International Conference on Real-Time Computing Systems and Applications*, Tokyo, Japan (2002)
9. Liu, J., Lee, E.A.: Timed Multitasking for Real-Time Embedded Software. *IEEE Control Systems Magazine: Advances in Software Enabled Control*, February (2003) 65-75
10. Henzinger, T.A., Horowitz, B., and Kirsch, C.M.: Embedded Control Systems Development with Giotto. *Proc. of the Conference on Languages, Compilers and Tools for Embedded Systems LCTES'01*, Salt Lake City, USA (2001)
11. Angelov, C., and Sierszecki, K.: Component-based Design of Software for Distributed Embedded Systems. *Proc. of the 10th IEEE International Conference on Methods and Models in Automation and Robotics*, Miedzyzdroje, Poland (2004)
12. Angelov, C., and Sierszecki, K.: A Software Framework for Component-Based Embedded Applications. *Proc. of the Asia-Pacific Software Engineering Conference APSEC'2004*, Busan, Korea, Dec. 2004
13. Kopetz, H., Reisinger, J.: The Non-Blocking Write Protocol NBW - a Solution to a Real-Time Synchronisation Problem. *Proc. of the IEEE Real-Time Systems Symposium* (1993) 131-137