In Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05). IEEE Press, 2005.

Towards Efficient Use of Shared Communication Media in the Timed Model

Guido Menkhaus, Michael Holzmann, Sebastian Fischmeister, Claudiu Farcas
Computer Science Department, University of Salzburg, Austria
{firstname. lastname}@cs.uni-salzburg.at

Abstract

Embedded computers are increasingly tighter integrated with each other forming distributed embedded systems that interact through a shared communication medium. Often these embedded systems perform services for safety-critical operations that require deterministic computation and predictable communication. However, since these systems are embedded, they must cope with limited resources. One fundamental challenge is to make efficient use of the shared communication medium, especially in the timed model, in which communication tends to cumulate at the end of harmonic periods of tasks.

In this article, we present an approach that uses Micro-tasks for splitting computation and communication into several sequentially executed steps to allow for better balanced load on the communication medium in the timed model. We discuss the approach and describe its implementation in OSEK/Works with TTCAN.

1. Introduction

Most modern control applications are implemented in software. It is therefore necessary to understand the different role of time from the perspective of software and control engineering [20]. The control engineering perspective is that processes evolve in continuous real-time, delays are small and jitter is negligible. From the perspective of software engineering, a set of tasks needs to be scheduled and time evolves discontinuously, since time elapses for a task only when it is active and running. Each task has a release time, a hard deadline and a worst-case execution time (WCET). Often, the deadline of a task equals the respective period of the task.

Jitter is defined as *time-related, abrupt, spurious variations in the duration of any specified related interval* [1]. Start jitter of a task is the time variation around the period of a task reading its inputs. Completion jitter of a task is the time variation around the period of a task completion point when updating its outputs. Start and completion jitter may lead to vacant sampling and sample rejection [23]. Sample rejection means that an input value of a task is obtained more than once between two consecutive release times of the task. Vacant sampling means that a new input value to a task is not available between two consecutive releases of a task and that an obsolete value is repeatedly used. Start and completion jitter of a task is due to the non-synchronicity of two data-dependent tasks. Apart from sample rejection and vacant sampling, severe jitter can degrade system performance [7].

The effects of start and completion jitter of a task are diminished by introducing the logical execution time (LET) of a task in the timed model [2][3][13]. In the timed model, the start of the LET marks the point in time when the input values are read. The end of the LET marks the point in time when the results of the computation of a task become available to other tasks or actuators. Even if the output of a task becomes available before the end of the LET, the output values will not be released prior to the expiration of the LET. The LET of a task is always greater or equal than the sum of the WCET and the worst-case communication time (WCCT); the WCCT is greater than zero in case values need to be transmitted to a different computational node in a distributed application. The concept of LET allows for precise synchronization of tasks for central and distributed applications. However, it disregards code efficiency and system reactivity, which results in unnecessary actuator updates and communication delays. In a distributed hard real-time application in which time-triggered and cyclicly executed tasks on different computational nodes need to exchange values over a network, applying the timed model leads to unbalanced network load [18]. The timed computation model defines the communication activities for each task to occur at the end of the LET. Since the periods of all tasks of an application start simultaneously at the beginning of the hyper period, there is a peak in computation (higher CPU load) at the beginning of the hyper period, whereas communication (network traffic) dominates at the end of the period.

In this article, we present an approach for balanced network load for the timed model. To balance the CPU and the network load and to distribute it equally over the hyper period, we introduce Micro-tasks. Micro-tasks define a sequence of executing tasks. They allow for communicating task results over a network regardless of the fact that from a semantic point of view outputs are only available at the end of the LET of a task. Micro-tasks split the computation of a task into a set of Micro-tasks and execute actuator updates intertwined with the invocation of those Micro-tasks [8] thereby allowing for the fine grained control of software tasks and their distributed interaction. We discuss an infrastructure for distributed hard real-time applications that bases on the Timing Definition Language (TDL) in which the computation of tasks and the communication of messages is controlled by a virtual machine.

The remainder of the article is structured as follows: Section 2 gives a short overview of concepts of the timed model. Related work is discussed in Section 3. Communication and efficient computation with Micro-tasks is introduced in Section 4. An overview over the TDL infrastructure for distributed real-time applications is given in Section 5. Section 6 presents the implementation of the TDL virtual execution environment on OSEK/Works and the TDL communication system is discussed in Section 7. Section 8 presents results and the article closes with a conclusion in Section 9.

2. Short Overview of Concepts for the Implementation of the Timed Model

The concept of languages representing the timed model bases on time-triggered cyclic computation and communication. The LET describes the timing behavior of a task. It denotes the invocation period, i.e. its relative release time and the deadline of a task. According to a scheduling scheme, the task starts after it has been released at the beginning of the LET and is active afterwards and completes its execution before the LET has elapsed. A communication schedule defines when the communication system transfers values from one computational node to another and when the next transmission will take place.

Timing Definition Language. The Timing Definition Language (TDL) is a software description language intended for timed computation. TDL allows for defining the timing behavior of a set of task [22]. It separates the timing constraints of an applications from the functional implementation, which must be provided separately, for example, using an imperative programming language such as C.

The most important programming abstractions of TDL are modules, modes, tasks, and ports: A module declares a set of modes. A mode defines a set of task, actuator, and mode switch invocations that are executed periodically and concurrently. A TDL module can only be in one mode at a time, but can change from one mode to another at the end of a mode period. A TDL mode specifies the period, i.e., the length of one computation cycle. A task period is determined with respect to the execution period of the mode to which the task is assigned (by dividing the period of the mode by the task frequency). A task has a set of input ports, output ports and a set of drivers reading and writing data from and to the ports. Ports are logical points of interconnection between tasks, modes, and modules. Drivers copy values between task input and output ports, read sensors, and update actuator. Mode drivers read sensors and update mode ports, which are a subset of the task output ports.

The communication subsystem ensures the timely transmission of values from one computational node to another. In a distributed application, in which a task on one computational node transmits its results to remote tasks, tasks output ports are logically distributed across the communicating nodes.

Virtual Machine for Task Monitoring. A task, to comply to the LET, is dispatched and monitored by a virtual machine, the E-machine [12]. The E-machine supervises the timing behavior of a set of tasks and ensures their timing consistency and their communication requirements. In a distributed environment, the communication requirements and the local design of timing definitions of the tasks may raise global design restrictions (scheduling restrictions) that need to be resolved and synchronized with the global communication schedule of all nodes.

The E-machine executes platform-independent E-code and calls platform-dependent application code. The application code is responsible for logical correctness and the E-code ensures timing consistency. The E-code consists of a small set of instructions for basic control flow and processing, that allows for synchronous calls of input and output port drivers, task and actuator scheduling and initializing the execution of a set of E-code instructions at some point in time in the future. A task, violating its timing properties, causes a run-time exception that is handled by an exception handler.

Autonomous Communication System for Value Transmission. The communication system of TDL supplies a data communication system that allows for transmitting values from task output ports of a TDL program to input ports of a task running on a different computational node. It uses a time-triggered communication subsystem to transmit data. It works autonomously: sending and receiving of messages happens without any interaction from the application program [18].

Synchronization. Time-triggered computation and communication, when distributed on several computational nodes, requires a synchronized time base among the participating nodes. The common time base is a precondition for synchronization of data-dependent tasks and for using a time-division multiple access (TDMA) bus arbitration scheme. In TDMA, a node accesses a single transmission channel without interference from other nodes. Usually, it is implemented by communicating nodes allocating predetermined time slots on the channel.

Determinism for Value and Time Predictability. The introduction of the E-machine ensures compliance to the LET, which allows precise synchronization of tasks for local and distributed applications. Assuming clock synchronization of the participating computational nodes, this leads to value and time deterministic distributed systems. Determinism is a desirable property for software systems that control physical systems that are ruled by deterministic physical laws [9]. In a deterministic system, an external observer can consistently predict the future behavior of the system. A systems is said to be value deterministic, if the same sequence of inputs produces the same sequence of outputs. If the system produces for the same sequence of inputs the same sequence of outputs at always the same time, then it will be value and time deterministic.

Timed Model for Determinism. In distributed applications, in which values from task output ports are transmitted over the communication medium to input ports of tasks on different computational nodes, the LET is composed of the logical computation time (LCT) of a task and the logical transmission time (LTT) of the task's output values (see Figure 1). The LCT specifies a fixed time interval in which the task is active. The task starts after it has been released, may be preempted, but resumes and always completes its execution before the LCT has elapsed. This LCT remains the same for e.g. different platforms or for different task implementations. Then, the LCT ensures that a set of tasks exhibits invariant timing behavior. The underlying assumption is that the set of tasks is schedulable, this means that there is enough time available within the interval to execute each task of the set.

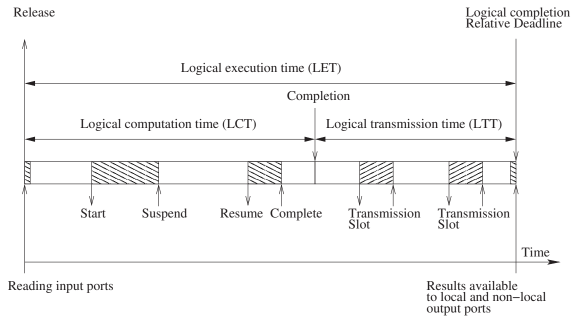

Release
Logical completion
Relative Deadline
Logical execution time (LET)
Completion
Logical computation time (LCT)
Logical transmission time (LTT)
Start
Suspend
Resume
Complete
Transmission Slot
Transmission Slot
Time
Reading input ports
Results available to local and non−local output ports

Figure 1. Implementation of computation and communication of a task in LET.

The length of the LET determines the length of the LTT. The real-time system designer specifies the LET and might need to reconcile it with the (1) LTT specifications of all tasks in the system and (2) the worst case communication time of the communication medium.

3. Related Work

Approaches to computation and communication in distributed real-time systems can broadly be divided into two categories. Time-triggered systems use a global time base for controlling the release time and the deadlines of tasks and the sending and receiving of messages. Event-triggered systems release a task after the registration of an event that activates the task. A message will be sent immediately if the communication media is free. If the media is occupied, the transmission will be delayed until the media is free.

The time-triggered protocol (TTP) is a communication protocol for fault-tolerant distributed hard real-time systems [16]. It provides time-triggered transmission of messages, distributed clock-synchronization, and a membership service. The communication on the bus is done with static and periodic TDMA rounds. Every message that is sent from any node has to be specified and an automatic scheduler generates a bus schedule matching the specifications. Implemented as message descriptor list, the schedule specifies exactly when a node has to send a certain message and when messages from the other nodes have to be received. The task descriptor list describes the cyclic scheduling of application tasks. This list specifies the instances of time of starting and stopping tasks. This system guarantees value and time determinism on a global level, however on a single computational node, time determinism and value determinism cannot be guaranteed, since tasks are not executed according to the timed model and values are not communicated at predefined instances of time between locally executed tasks.

Event-triggered communication must provide collision detection and resolution or avoidance techniques. CAN uses a bit arbitration for collision avoidance [5]. Bitwise arbitration allows for determining the priority of each message sent. Messages with a higher priority continue to send while nodes with low priority messages delay their transmission. Bitwise arbitration causes non-deterministic transmission delays.

DaVinci supports the control engineer to develop applications for distributed platforms [24]. It provides two tools which separate the design of functional components from the integration of those components to a specific, possibly distributed platform. The runtime behavior of a DaVinci application depends on the prioritization of tasks and bus messages. The unrestricted way of using, for example, semaphores or other resources may lead to phenomena such as priority inversion, where determinism cannot be guaranteed.

In this article, we present the TDL system for time-triggered computation and efficient communication that aims at value and time determinism of a hard real-time application on each computational node and among all distributed nodes.

4. Intertwined Communication and Computation with Micro-tasks

In [18], the authors suggest to split the LET into an LCT and an LTT. In this model, the results of the task are passed to the communication subsystem for transmission on the shared communication medium at the end of the LCT. Thus, communication tends to accumulate at the end of harmonic task periods and the hyperperiod of each modules. If several modules have the same mode period, then the shared communication medium will be congested at certain times.

Figure 2 shows an example of a situation in which communication piles up. Task T_1 and T_2 share the same frequency and the same mode period. Task T_1 communicates two messages and Task T_2 communicates one message. Both tasks send the their messages after the LCT and before the LET expires. However, since the communication medium is shared, the remaining time of the LET is insufficient to transmit all messages and the bus scheduler might try to resolve this conflict by reducing the LCT of one of the tasks. This however, might entail other conflicts or contradict with system requirements.

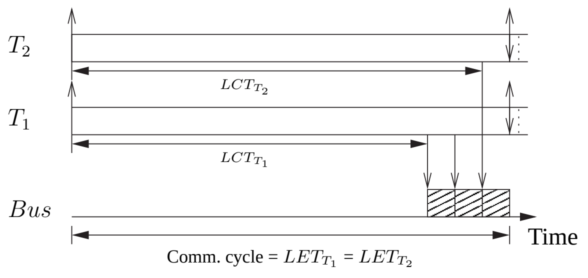


Figure 2. Communication cumulates at the end of the period and decreases the available LCT of Task T_1 and Task T_2.

To provide more flexibility for the bus scheduler, we split task T_1 into a sequence of Micro-tasks. Each Micro-task computes results and if no succeeding Micro-task updates this results subsequently in the same period, the value will be passed to the communication system and thus the bus scheduler will be enabled to schedule the transmission of this value even before the LCT of the parent task has elapsed.

Figure 3 shows an example of intertwined computation of Micro-tasks and communication of their results. In contrast to the previous example, Task T_1 is now split into three Micro-tasks: mt_1, mt_2, mt_3. Each Micro-task computes results, but only the results of Micro-task mt_1 and mt_2 must be transmitted over the bus. The bus scheduler is able to schedule the transmission of the results of Micro-task mt_1 and mt_2 immediately after their completion. As Figure 3 shows, in total Task T_1 has a larger LCT than in Figure 2 ($LCT_{T_1} = \sum_i LET_{mt_i}$), since communication is intertwined with computation.

4.1. Micro-tasks in TDL

Listing 1 implements the TDL program for the example of Figure 3 (T_2 omitted). The syntax and semantics of Micro-tasks are similar to the declaration of tasks (for

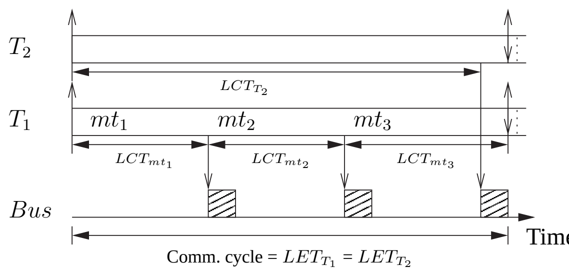


Figure 3. Single values are computed using Micro-tasks and are sent prior to the end of the LCT of Task T_1.

full description of the syntax and semantic of Micro-tasks, see [8]).

```
task mt1 [WCET = 100 ms] {
  input int i;
  output int o;
  uses f1(i,o);
}
task mt2 [WCET = 200 ms] {
  input int i;
  output int o;
  uses f2(i,o);
}
task mt3 [WCET = 50 ms] {
  input int i, s;
  output int o;
  uses f3(i,s,o);
}
task T1 {
  input int i;
  output int o1, o2;
  state int s;

  [LET = 200 ms] mt1 {i := this.i};
  this.o1 = mt1.o;
  [LET = 300 ms] if g2() then mt2 {i := mt1.o};
  this.o2 = mt2.o;
  [LET = 100 ms] mt3 {i := this.i, s := this.s};
  this.s = mt3.o;
}
start mode main [1000 ms] {
  task
    [1] T1 {i := s1};
    ....
}
```

Listing 1: Example TDL snipplet for Figure 3.

Task T_1 comprises three Micro-tasks. First, it releases mt_1 that has to finish its execution within the next 200 milliseconds (LET = 200 ms and WCET = 100ms). Micro-task mt_1 computes the output o_1 of Task T_1 as the assignment shows in the declaration of task T_1. Micro-task mt_2 executes, if the guard g_2 evaluates to true. If it executes, it will finish within the next 300 milliseconds as specified in its LET. Micro-task mt_2 computes the output value o_2 of task T_1. Finally task T_3 computes the new value for the state variable s.

4.2. E-code Generation

The TDL compiler compiles the TDL program into E-code. To provide more flexibility for the bus scheduling, the output values are transmitted to the communication subsystem as they are no longer modified for the remaining of the LET of the parent task. This increases the time interval (LTT) in which the values may be transmitted by the communication system and thus increases the number of possibilities for the bus scheduler to find a schedulable solution.

Listing 2 shows the resulting E-code for Listing 1. The most important E-code instruction are the $call$, $schedule$, $future$, and $return$ instruction: The $call$ instruction synchronously executes drivers of a task. The $schedule$ instruction releases a task to be activated by the dispatcher of the operating system. The $future$ instruction initiates the execution of a block E-code instructions at some time in the future and the $return$ instruction finishes the execution of the block of E-code of the current label. The E-code starts with the instruction at label $L0$. In the first two instructions, the E-machine calls the drivers for reading input values into the input ports of task T_1 and Micro-task mt_1. Then, task mt_1 and T_1 are scheduled. Execution of the E-code is then continued at Label L1 after 200 ms, indicated by the future instruction. As the output value o_1 of Task T_1 does not change after the output value of Micro-task mt_1 has been assigned, the value is passed to the communication system after 200 milliseconds calling driver nw_1. The same is done for the output value o_2 of Task T_1, analogous to the E-code instruction at Label L0.

```
L0:   call d[T1]
      call d[mt1]
      schedule mt1
      schedule T1
      future 200000, L1
      return
L1:   call d[nw1]
      call d[mt2]
      schedule mt2
      future 300000, L2
      return
L2:   call d[nw2]
      call d[mt3]
      schedule mt3
      future 100000, L0
      return
```

Listing 2: The E-code for the TDL program in Listing 1.

5. Overview of TDL for OSEK and TTCAN

In the following, we present a prototype implementation of the TDL infrastructure for distributed hard real-time applications. Figure 4 provides an overview. The TDL runtime part of the infrastructure consists of TDL-Exec and TDL-Comm: TDL-Exec implements the E-machine and TDL-Comm is a communication system that allows for transmitting values from task output ports of a TDL program to input ports of a task running on a different node. TDL-Exec and TDL-Comm are implemented on OSEK with sTTCAN as communication system.

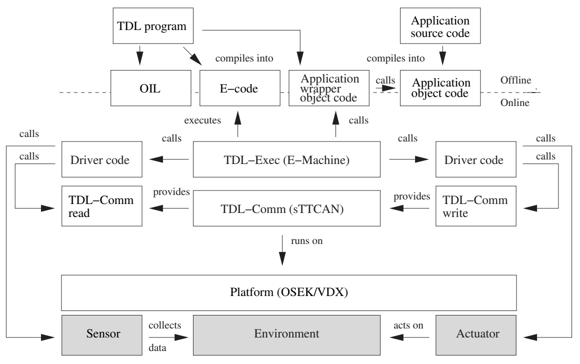


Figure 4. TDL infrastructure for distributed real-time applications.

OSEK/VDX started as a joint project of the automotive industry becoming an open standard for real-time software control systems. Current specifications are limited to single processor systems, with support for distribution via specific communication interfaces such as CAN. The operating system is designed to be modular and configurable to support a wide range of embedded platforms with small memory footprints targeting low-end microprocessors/micro-controllers. The operating system does not support dynamic generation of system objects, such as tasks, services, or resources.

The configuration of tasks and their properties, such as priority, stack space, preemption flag, and alarms and resources are statically defined at compile time in a specific configuration file using the OSEK Implementation Language (OIL). During runtime, the number of tasks and their properties described in the OIL file cannot change. Each OSEK task has a statically defined priority that is used by the OS dispatcher to decide which task to run. OSEK tasks can have one of two types: basic and extended. The main difference between them is resource access. Any basic task can either be in the ready, running, or suspended state. Extended tasks may have the additional state of waiting for a resource. In addition to task priorities, the timing aspects of tasks can be controlled with alarms based on user defined counters. The counters are all derived from the system counter or some external timer interrupt. The alarms can be setup to activate new instances of tasks just once or in a cyclical fashion.

6. TDL-Exec for OSEK

The TDL execution environment (TDL-Exec) consists of the E-machine. It must exhibit the same properties in all implementations no matter the underlying real-time operating system. The current implementation is based on standard

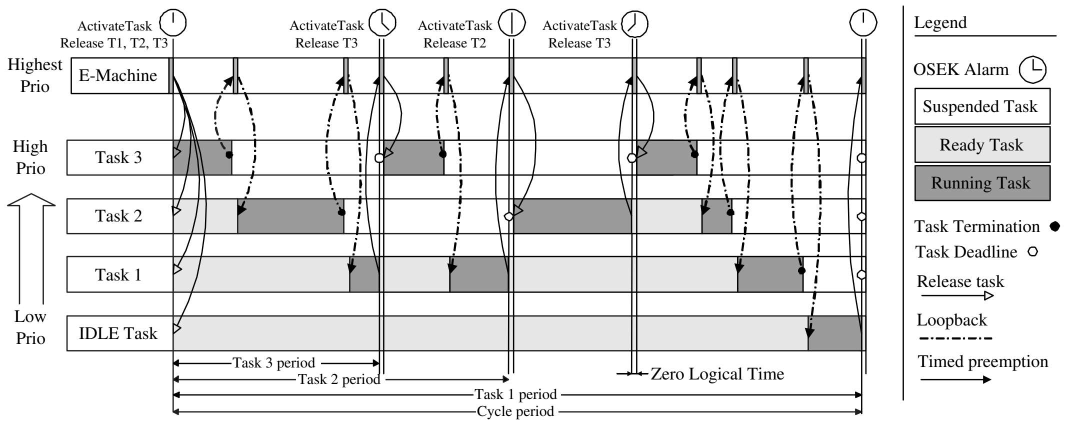


Figure 5. Rate Monotonic scheduling with OSEK

ANSI C-code, making it highly portable to most embedded platforms. Since each operating system has different ways to implement basic concepts such as threads, tasks, delays or alarms, all of these OS dependent aspects are separated from the application code and the implementation of the E-machine for easier maintenance and portability.

The TDL toolchain may use model-based development tools such as Simulink [17] to produce application source code and the corresponding TDL program [21]. The TDL compiler compiles the TDL program into E-code and application wrapper code that is used to build the final real-time application conforming with the timed model. In the case of the OSEK platform, an additional OIL configuration file is generated and linked with the E-code, the wrappers, the application code, the E-machine and OSEK libraries producing the final executable that is executed on the target platform. As the OSEK OS does not provide any facilities for supporting a filesystem or dynamic loading of user code, the E-code is expressed as a static C data structure that is linked with the functional code and compiled in in the final application. The task wrappers represent OSEK tasks. The OSEK tasks that encapsulate the application code are released by the E-machine according to the corresponding E-code instructions, this means in a time-triggered way. The E-code sequence of instructions actually encodes the application's reactive behavior and performs the lock-free resource sharing (such as sensors or actuators) among different logically concurrent tasks.

6.1. Task scheduling with the OSEK E-machine

The E-machine mediates between soft (logical) time and real (physical) time and releases the tasks of the application. It monitors the released tasks, their deadlines and their communication requirements and constraints imposed by TDL-Comm.

The OSEK internal scheduler supports plain static priority based scheduling. To implement a different scheduling policy such as EDF or RM [6], OSEK provides the chaining tasks feature. Chaining tasks are used in the OSEK E-machine to complement the scheduling scheme of OSEK. The E-machine selects the next active task according to the current scheduling policy, such as EDF. However, depending on the implementation of the OSEK chaining tasks feature and the underlying OSEK task scheduler/dispatcher the functionality may not be available under all OSEK OS variants.

6.2. E-machine Implementation under OSEK

The E-code $schedule$ instruction releases a task by toggling its release flag and turning it into the ready state. After a sequence of $schedule$ instructions follows the E-code $future$ instruction that initiates the execution of a sequence of E-code after a specific amount of time has elapsed. Depending on the CPU speed of the target platform and the OSEK OS variant the clock resolution may not be up to microseconds, but instead to hundreds of microseconds or a couple of milliseconds. On the sample platform (KANIS board with MPC555 CPU), the clock accuracy was determined to be around $500 \mu s$. For lower time intervals the clock drifting effect corrupts the overall behavior of the application. The $return$ instruction ends the interpretation of the current E-code block. The E-machine keeps track of all $future$ instructions parameters: future time and future E-code instruction pointer, planning the activity for the next cycles.

During the future time interval the released tasks are run by the E-machine scheduler according to its scheduling policy. Since the E-machine is a task itself, the E-machine is triggered (1) by an OSEK alarm in a cyclic fashion with time intervals equal to the last future time (from the last E-code $future$ instruction after a sequence of $schedule$ instructions) and (2) by application tasks after finishing their computation. This loop-back mechanism (the E-machine is triggered by application tasks, that it itself releases) that returns control from the application tasks to the E-machine before the future time has elapsed, allows running one task at a time. Preempting a task is only necessary for scheduling a new task, instead of periodically preempting tasks with a certain frequency as many RTOS do [10]. The loop-back mechanism improves the compactness of CPU time allocation for application tasks and generally provides less CPU load and allows for better utilization of the remaining CPU time for non real-time tasks, such as diagnostic activities.

Approaches using static non-preemptive scheduling with no context switches (except for interrupt handling and other OS critical activities) can schedule only specific sets of tasks [6]. Our approach is more flexible with respect to the set of tasks since the mechanism of our scheduling greatly reduces the number of context switches required. Implementing an optimal scheduler such as EDF or RM, the CPU utilization level can be computed in advance for any given set of tasks and in dynamic systems this may be a decisive aspect for dynamically loading additional functionality on the system. In our approach the benefit comes from the reduced number of context switches (that is not negligible on low-end micro-controllers running high frequency tasks).

Figure 5 describes the Rate Monotonic scheduling policy implemented with OSEK. Each task has a priority proportional to its frequency: a higher frequency task has a higher priority and a task with low frequency has a lower priority. A task is released by the E-machine via ActiveTask function and the OSEK scheduler selects the active task with the highest priority to run. A higher priority task preempts a lower priority task until it finishes its computation. If a tasks communicates with another task on a different computational node, the task completes its execution by looping back to the E-machine to perform communication related functions.

6.3. Scheduling Micro-Tasks

From the E-machine point of view the Micro-tasks are regarded as regular tasks. The E-code contains the actual instructions to release the Micro-tasks at the right moments and to run the drivers when needed. The main difference is that the E-machine will be triggered more often than in the case of regular tasks (equal to the number of Micro-tasks a regular task is composed of). The performance impact of this side-effect is acceptable and is backed up by the additional flexibility in implementing arbitrary computation / communication workloads.

7. TDL-Comm on top of sTTCAN for OSEK

The TDL infrastructure accommodates the TDL Communication Subsystem (TDL-Comm) for distributed applications. TDL-Comm implements a distributed shared variable space in which TDL ports are shared among several computational nodes. TDL-Exec accesses shared TDL public ports via the TDL-Comm interface.

TDL-Comm builds on a time-triggered communication subsystem. The time-triggered communication subsystem of each network node processes all communication activities. TDL-Comm accesses the communication network interface of the node's local network controller which contains all messages received or sent by the network controller. It copies data between this communication network interface and the TDL-Comm interface in order to make them accessible to TDL-Exec.

7.1. TDL-Comm Interfaces

TDL-Comm exposes three interfaces to the TDL-Exec of a computational node:

- *Comm Interface.* The Comm Interface contains the data shared among the nodes. The data is represented by means of TDL ports. TDL tasks of TDL-Exec access the shared ports within the Comm Interface using port specific drivers. The drivers copy values of output ports of a task to the input ports of the Comm interface, which forwards them to the communication network interface. The Comm Interface has a set of output ports, whose values are destined for input ports of tasks, which are then retrieved via tasks drivers.
- *FT Interface* The FT interface contains information regarding the status of replication and fault tolerant communication.
- *Native Interface.* The native interface contains supports, for example, access to platform specific services, such as the error counters of a CAN controller, etc.

7.2. TDL-Comm toolchain.

TDL-Comm consists of an offline and an online part. The runtime behavior of the TDL-Comm online part is defined a priori (before runtime) by the offline part (TDL-Comm scheduler).

Offline. The offline part analyzes the distributed application and generates the communication schedule. The TDL-Comm compiler scans all TDL modules and a module using resources (sensors, actuators, or ports) of a different module located on a remote node indicates that communication is required. The use of resources of a different module is indicated by the import relation between TDL modules. The mapping of TDL modules (parts of functionality) to computational nodes is defined in a dedicated configuration file which is read by the TDL-Comm compiler.

The communication requirements are the basis for creating a global bus schedule. After generation of the global bus schedule, the TDL-Comm compiler generates a static communication schedule list for each computational node, which defines the behavior of the TDL-Comm online part during runtime.

Online. The TDL-Comm online part processes the a priori defined static communication schedule list. The communication schedule list defines activities which are synchronized with the timing of TDL-Exec and which are cyclically repeated in synchronization with the TDL mode period. The activities are sending and receiving of messages or the preparation of messages for transmission depending on the underlying communication subsystem. Other activities are copying data between the TDL Comm interface and the communication subsystem, marshaling TDL ports into data frames of the communication subsystem, voting fault tolerant values and updating the status fields within the FT interface.

7.3. Software TTCAN Implementation for OSEK

The TDL-Comm implementation uses a standard physical CAN link [5] and bases its concepts on TTCAN [11]. TTCAN extends the CAN protocol by providing time-triggered communication via the standard physical CAN link [5]. TDL-Comm is based on a proprietary implementation of TTCAN in software which we call sTTCAN. This implementation was required because of the lack of availability of a production TTCAN controller chip and tool chain support on the one hand and because of the greater flexibility to adopt the protocol to TDL specific needs on the other hand.

7.3.1. Clock Synchronization Time-triggered communication requires a globally synchronized time base that needs to be established among the participating nodes [15]. The synchronized local clocks of the nodes enable the participating nodes to access the bus via TDMA arbitration scheme. Each node has certain instances (time slots) assigned in which it has exclusive access to the communication medium thus avoiding collisions and allowing for predictable jitter-free communication.

The synchronization of the local clocks of the nodes is accomplished using a master-slave clock synchronization scheme based on the TTP/A fireworks protocol [14]. One of the nodes act as clock synchronization master cyclically broadcasting a synchronization message (sync frame) which is used by the other nodes (slaves) to synchronize their clocks to the clock of the master. On reception of the sync frame of the master, the slaves generate a time stamp with their local clock. Each sync frame contains a time stamp of the masters clock at the time of sending the sync frame. The slaves calculate the current deviation of their clocks from the master clock subtracting the time stamp received within the sync frame from the time stamp generated with their own clock. This deviation is compensated within the current synchronization period (until the next sync frame is received) by adjusting the speed of the local clock.

Time stamping is done using the time stamping feature of the MPC555 CAN controller. It automatically generates a time stamp on reception of a message, i.e., at the instant of detection of the start of frame symbol on the bus. Thus it is possible to do time stamping at the slave nodes without the use of interrupts. Using interrupts would cause indeterminism, because of possible preemption of the TDL-Exec, and furthermore executing interrupt service routines to generate time stamps is not accurate because of jitter.

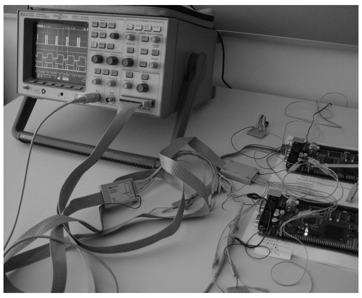
7.3.2. Software TTCAN Synchronizing the OSEK system timers of the computational nodes enables a software implementation of a time-triggered communication protocol via the standard CAN bus entirely under OSEK. The communication schedule list is locally stored at each node. For each slot there is an entry in the communication schedule list denoting the point in time a slot starts, the activity (send or receive), the length of the slot, and the number of the message it has to send or to receive within this slot.

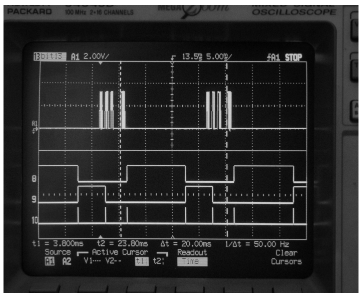
The communication pattern is recurring cyclically, the length of the cycle is synchronized to the TDL mode period. The sync frame sent by the clock master denotes the start of a new communication round. On its reception all nodes start processing their static communication schedule list from the beginning. Transmitting messages usually does not require CPU interaction if a time-triggered communication subsystem is used, because it usually has its own autonomous controller. However, the sTTCAN implementation needs CPU interaction to trigger the sending of messages.

Sending of messages is done setting-up OSEK alarms successively for each send entry in the communication schedule list. If the alarm fires, the alarm callback routine triggers the CAN controller to send the appropriate message denoted in the actual entry of the communication schedule list. Then, the next alarm is set-up according to the instance denoted in the next entry of the communication schedule list. This is repeated until the end of the list is reached. With the beginning of the next communication round processing starts anew from the beginning of the list.

Receiving of messages does not need CPU interaction. If a message is received which is intended for the node it is automatically stored in one of the 16 buffers of the CAN controller.

7.3.3. TDL-Comm with Micro-task Support In the case of the prototype implementation on top of sTTCAN the TDL-Comm Interface is represented by the 16 message buffers of the CAN controller. Each of the 16 message buffers can hold one message. The prototype implementation of TDL-Comm consists of a set of specific TDL drivers which directly access the data within the CAN buffers and copy them to a group of dedicated TDL ports representing the TDL-Comm interface. The communication schedule list determines the number of the buffer entry (message buffer) an activity has to be carried out on at a specific point in time (see Table 1). The beginning of an activity, i.e., the start of a time slot is denoted in ms (Start Time) relative to the start of the round, the number of the buffer entry is denoted as the


(a) The setup.


(b) Congestion at the end of the period.

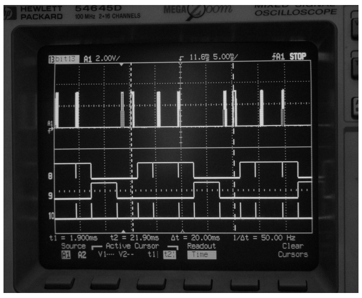

(c) Balanced load (Micro-tasks).

Figure 6. TDL infrastructure setup using Micro-tasks to balance network load in the timed model.

number of the corresponding message buffer and the length of the time slot is denoted as length of the message to be transmitted within. The timing denoted within the list, i.e., the start time of a slot is converted into OSEK system timer ticks (MT) in order to set up the corresponding alarms (refer to section 7.3.2).

Pos.	Start Time	#Buffer	Length
1	2	1	4
2	15	2	2

Table 1. Communication schedule.

The scheduling list presented in the table 1 defines the communication schedule for one node that executes two activities within each communication round. The first entry of the list indicates that during the time slot beginning at $ms2$ relative from the start of the round it has to send a four byte message which is stored in buffer number one. The second entry indicates the transmission of a two byte message during the time slot beginning at $ms15$ stored in buffer number two. The timing of the communication subsystem and the E-machine rely both on the local OSEK system timer and on the fact that the OSEK system timers of all involved nodes are synchronized.

Regarding Micro-tasks, access to the TDL-Comm is completely transparent to the TDL-Exec. Both, ordinary TDL tasks as well as Micro-tasks access public ports within the TDL-Comm interface via dedicated drivers. To communicate a value to a remote node, TDL-Exec executes a specific driver which is declared in the E-code resulting from a TDL program running on the local node. The driver copies the output value from the output port of a task or a Micro-task to the TDL-Comm Interface. At the receiving node, a dedicated driver, declared in the E-code of the TDL program running at this node, fetches the received value from the buffer of the CAN controller it was received in and copies it to the input port of the receiving task or Micro-task.

From the point of view of TDL-Comm the treatment of the TDL-Comm interface ports accessed by ordinary TDL tasks and the treatment of the ports accessed by Micro-tasks is different regarding timing aspects. In the case of the former the instances at which the ports within the interface are updated are determined by the LCT restrictions imposed on the timing of communicating tasks by the schedule of the bus. In the case of the latter they are determined by the instances defined in the Micro-task code at which the drivers accessing the TDL-Comm interface are invoked.

Because of the explicit definition of the timing of activities within the LET, the Micro-task approach allows for communication of values at the beginning of the LET and thus for utmost flexibility regarding the bus scheduling.

8. Results

Our target platform is the Motorola MPC555, a micro controller commonly used for automotive applications [19]. The target board is the KANIS OAK EMUF [4], which features integrated physical layer drivers for CAN and enhances the I/O palette offered by the controller with an additional Ethernet link. Each computational node of the example distributed TDL application is a KANIS OAK EMUF board. The nodes in this example are interconnected via a CAN bus link (see Figure 6(a)).

We present the results of the examples introduced in section 4: Task execution with communication at the end of their period and Micro-task implementation with intertwined communication during the LET of the parent task. Figure 6(b) and 6(c) present images of an oscilloscope showing the task dispatching, the E-machine activities, and the network frames. Each image of the oscilloscope shows four different signals: The analog line A1 (uppermost signal) is connected to the CAN bus signal and displays network frames (e.g., sync frame and data frame). Each round starts with the sync frame on the CAN bus. The cycle length of the application is 20ms and is delimited in the oscilloscope images by the two vertical dashed lines (cursors). The digital input signals 8, 9, and 10 (three lowermost signals) are connected to digital outputs of the board and display the states of tasks and the E-machine task. A high signal means that a task is running and a low signal denotes a task that is stopped or suspended.

Figure 6(b) shows the behavior of the implementation without Micro-tasks. After start of the period by the sync frame (right next to the cursor line), the E-machine task becomes active (peak on oscilloscope at input line 10) releasing tasks T_1 and T_2. T_1 is scheduled immediately (oscilloscope at input line 8 changes to high) after the E-machine task has terminated. After task T_1 has completed its computation, the E-machine task is resumed by the chain-task loopback (peak at input line 10). It invokes the communication driver for task T_1 and schedules task T_2 (input line 9) for execution. While T_2 executes, the CAN controller communicates the results of T_1 in two slots. After task T_2 has finished its computation (input line 9 changes back to low) its result is communicated via the CAN bus in the last slot before the next round starts.

Figure 6(c) presents the behavior of the implementation using Micro-tasks. At the start of the period, the E-machine task (peak at input line 10) releases tasks T_1 and T_2. Task T_1 consists of three Micro-tasks mt_1, mt_2, and mt_3. Micro-task mt_1 starts (input line 8 changes to high) immediately after the E-machine task has finished. Micro-task mt_1 finishes its computation (input line 8 changes to low) and triggers the E-machine task, which invokes the drivers that copy the result of mt_1 to the communication system. Then it schedules Micro-task mt_2 (input line 8 changes to high again). Concurrently with the execution of mt_2, the result of mt_1 is transmitted via the CAN bus (network frame on input line A1). Analogous the result of mt_2 is communicated while mt_3 executes. After its completion the E-machine schedules task T_2 (input line 9 changes to high). The result of T_2 are communicated over the bus after completion of its computation (input line 9 changes to low).

9. Conclusion

In distributed hard real-time systems, deterministic and efficient usage of shared communication media is of great importance. The timed computation model for real-time systems implements value and time determinism. However, it favors value and time determinism over time efficiency. In this paper, we presented Micro-tasks for intertwined computation and communication. Micro-tasks allow for efficient and balanced load on the communication medium, providing the bus scheduler with more flexibility with respect to the time interval in which data can be transmitted.

We described the TDL infrastructure for distributed real-time applications presenting TDL-Exec and TDL-Comm. TDL-Exec provides the E-machine that dispatches and monitors the execution of tasks in real-time. TDL-Comm accommodates a communication system for real-time data transmission. We discussed the execution of Micro-tasks in the TDL infrastructure and presented an implementation for OSEK/Works and sTTCAN.

As future work, we will explore how to use Micro-tasks to impose certain communication patterns on the bus such as predefined communication gaps.

References

[1] The new IEEE standard dictionary of electrical and electronics terms, IEEE standard, 1992.

[2] R. Alur and D.L.Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183 – 235, 1994.

[3] R. Alur, L. Fix, and T. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2):253–273, 1999.

[4] W. Bals. *Hardware Manual OAK_EMUF*. Ing. Büro W. Kanis GmbH, Brückenweg 2, D-82327 Tutzing, 2002.

[5] Bosch. *CAN Specification, Version 2*. Robert Bosch GmbH, 1991.

[6] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2000.

[7] D.-J. Chen and M. Sanfridson. Introduction to Distributed Systems for Real-Time Control. Technical Report TRITA MMK 1998:22, Department of MD, KTH, Sweden, 2000.

[8] S. Fischmeister and G. Menkhaus. Task Sequencing for Optimizing the Computation Cycle in a Timed Computation Model. In *23rd DASC'04*. IEEE Press, 2004.

[9] G. Franklin, D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Prentice Hall, 1997.

[10] FSMLab. *Real-Time Linux*, 2004.

[11] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communications on CAN. In *Proceedings 7th International CAN Conference*, 2000.

[12] T. A. Henzinger and C. Kirsch. The embedded machine: Predictable, portable real-time code. In *PLDI*, 2002.

[13] C. Kirsch. Principles of Real-Time Programming. *Springer LNCS*, 2491, 2002.

[14] H. Kopetz. TTP/A The fireworks protocol. In *SAE International Congress and Exposition*, 1995.

[15] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[16] H. Kopetz. The time-triggered model of computation. *RTSS98*, 1998.

[17] MathWorks, www.mathworks.com. *Simulink*.

[18] G. Menkhaus, M. Holzmann, and S. Fischmeister. Time-triggered Communication for Distributed Control Applications in a Timed Computation Model. In *23rd DASC'04*. IEEE Press, 2004.

[19] Motorola. *MPC555/556 User's Manual*, 2000.

[20] J. Sifakis. Modeling Real-Time Systems-Challenges and Work Directions. In *EMSOFT*, pages 373–389. Springer-Verlag, 2001.

[21] G. Stieglbauer and W. Pree. Visual and Interactive Development of Hard Real Time Code. In *ASWSD'04*, 2004.

[22] J. Templ. TDL Specification and Report. Technical report, Computer Science, University of Salzburg, 2004.

[23] M. Törngren. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. *Real-Time Systems*, 14(3):219 – 250, 1998.

[24] M. Wernicke. New Design Methodology from Vector simplifies the Development of Distributed Systems. *Vector Informatik Press Release*, 2003.