In Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05), 2005.

Non-blocking Deterministic Replacement of Functionality, Timing, and Data-Flow for Hard Real-Time Systems at Runtime

Sebastian Fischmeister
University of Pennsylvania
sfischme@seas.upenn.edu

Klemens Winkler
University of Salzburg
klemens.winkler@cs.uni-salzburg.at

Abstract

Embedded systems are usually an integral component of a larger system and are used to control and/or directly monitor this system by using special hardware devices. The complexity of the whole system, which the embedded control system monitors, increases steadily. Consequently, the initial version of the control software that is used at the time of deployment may be inadequate and may need to be updated. Often this requires the whole system to be shut down to have the software replaced. This is not a desirable solution.

In this work, we propose a non-blocking mechanism embedded into an infrastructure for RTLinuxPro for deterministic replacement of system functionality, task timing, and data-flow for hard real-time systems. We explain the mechanism, discuss its implementation using RTLinuxPro, and present a case study of a stop watch in which we replace single functionality and timing behavior at runtime without compromising the timeliness of tasks or the correctness of the output values. The contribution is to show how such a mechanism can work, how it can be implemented, and what problems arise in multi-mode real-time applications.

1. Introduction

Embedded hard real-time systems have a growing demand for flexibility and evolving requirements [2]. This includes invalidating an old version of the control software and activating a new one possibly while the system is running. Traditional embedded hard real-time systems are monolithic; that means the developer designs the application (hardware and software), verifies its properties (e.g., timeliness and logical correctness), builds the hardware target, and deploys the software on this target. In case some functionality needs to be changed, the developer halts the system, deploys the new software on the target, and restarts the target. Updating the system software may require the system to be shut down or even to replace hardware. Modern embedded real-time systems host potent microprocessors as the core system component and functionality is implemented in software instead of dedicated hardware (see [1, 5, 8]). This supports the growing demand for flexibility and allows for support of evolving requirements and the resulting functionality updates without replacing hardware and more significantly, it allows for system updates and exchanging functionality at runtime.

Exchanging functionality of a running application can be implemented in several ways. In general, one specific functionality (feature) comprises several functions. When exchanging functionality, the system can either replace it or use the new functionality as add-on to the existing one. Replacing functionality implies that the system discards existing (old) functionality. For instance, it can do so by deallocating the code and then unloading it. When the system needs new functionality as add-on to existing one, it does not require the existing (old) functionality to be discarded. Add-on functionality may use parts of the old functionality but does not have to. A disadvantage of add-ons is that over time old functionality cumulates in the memory.

In this paper we propose a non-blocking mechanism embedded into an infrastructure of how to replace functionality, timing, and data-flow in a deterministic way at runtime without interrupting the running hard real-time application.

The remainder of the paper is structured as follows. Section 2 explains the concepts used in the mechanism and in the implementation. Section 3 discusses related work that supports functionality replacement in non–real-time, soft real-time, and hard real-time applications. Section 4 presents the replacement mechanism and Section 5 shows, how this mechanism is embedded into an infrastructure implemented for RTLinuxPro. Section 6 demonstrates how we can replace functionality and timing in a running hard real-time application using a stop-watch case study. Section 7 evaluates the mechanism and its implementation and shows the problems and pitfalls when implementing multi-mode applications. Finally, Section 8 draws conclusions from this work.

2. Concepts

In our implementation, we use standard principles and concepts from the domain of real-time systems that we briefly outline in the following paragraphs.

2.1. Programming Model

Logical correctness and timeliness are the two most important aspects of a real-time system. Different models provide solutions of how to achieve time determinism and value determinism. On top of these models, programming languages and systems provide means of how to program control systems with real-time constraints.

For this work, we use the timed model, as proposed in [7]. The timed model uses time-triggered periodic computation of tasks and each task executes the stages (1) reading inputs (e.g., sensor data), (2) call task-specific functionality, and (3) write outputs (e.g., update actuators). The time between (1) and (3) can be seen as the logical execution time of a task (LET). The task is released at the beginning and terminated at the end of the LET. The release and the termination event are time-triggered. Within these two events, the task-specific functionality is executed according to a scheduling scheme. The start of the LET specifies the point in time when the input values are read. The end of the LET specifies the point in time when the output values are written (even if the task has completed before that time and the output would already have been available). The worst case execution time (WCET) of a task is the maximum time span a task may execute on a specific platform. Since a task needs to terminate before the end of its LET, the WCET is smaller or equal than the LET. The timed model is time and value deterministic.

2.2. TDL & Infrastructure

The Timing Definition Language (TDL) implements the timed model. It bases on Giotto and includes extensions such as the concept of a module, improved language syntax, and clean-house implementations of the underlying infrastructure on several hardware platforms.

TDL introduced *modules* as a compilation unit, to support modular decomposition. A module is equivalent to a real-time application. Each module contains one or more *modes*. A mode is a state of operation of a control application, which is executed periodically. For example, an airplane control application could consist of a take off, touch down, and a cruise mode. Each mode consists of a set of *tasks*. A task encapsulates a computation to be executed by a control application. A task has a WCET and a frequency, which determines the LET of the task. The relation between frequency, LET, and mode period is: $LET = \text{mode period}/\text{frequency}$. For a detailed description of TDL see [13].

TDL uses an embedded machine (E-machine) [7] to ensure timing consistency of task executions. Figure 1 shows the E-machine in its context. The TDL compiler compiles the TDL program into embedded code (E-code). The E-machine runs on the target, gets information from the environment through sensors, and updates the environment through actuators. It is a virtual machine that interprets the platform-independent E-code and calls the platform-dependent functionality code.

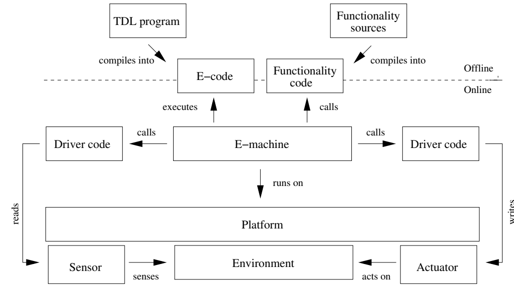


Figure 1. The embedded machine in its context.

3. Related Work

Replacing functionality is already common in non–real-time systems. Oberon [14] was one of the first programming languages allowing for dynamic functionality loading at runtime. Also, Java [9] allows for loading and unloading of classes at runtime. In general, systems using shared libraries also usually allow for replacing these libraries at runtime. For example, Linux with *dlopen* and *dlsym* but also Windows. However, these solutions are not apt for the domain of real-time systems, since they provide no timing guarantees and can only act as a vision what to achieve in the real-time domain.

In the area of soft real-time (i.e., the task should meet the deadline, but misses causes no catastrophes), functionality replacement is also common and some systems implement it. For example, B. Ravindran describes in [11] a resource management architecture for engineering dynamic real-time distributed systems together with resource management middle-ware algorithms. In his work, he considers migrating application program components between host machines, however, this approach is still limited to the area of soft real-time computing and does not guarantee timeliness for hard real-time tasks. Montgomery proposes in [10] to use pseudo-linked files to replace functionality at runtime. It does not require a custom execution environment and is a straightforward method. The approach presented

in this paper is similar, except, (a) it does not require the software developer to use function pointers for replaceable functions, (b) it provides time determinism and thus provides deterministic behavior in the case of the replacement action, and (c) incremental replacement actions do not cumulate in the memory as they do in the heap in Montgomery's approach. On the other hand, we require the developer to use our runtime environment, though it is part of a large tool chain.

In [12], Real et al. propose a mode-change protocol especially tailored for promptness of mode changes after the mode-change request has occurred. Mode-change protocols can be used to switch from one version of the software to another. Following the proposed taxonomy in [12], the our approach is synchronous. It is always schedulable, promptness depends on the length of the hyper period of the application, and it is always consistent. In contrast to their work and other work with offsets, our protocol aims at determinism in the time domain. Using the proposed system, the developer is able to tell the exact moment in time, the system will switch and thus is able to understand and analyze side effects especially in distributed real-time systems.

4. Replacement Mechanism

The mechanism consists of two parts, one executed offline and the other executed online. The developer usually runs the offline part on his workstation. This part consists of an admission test and tells the developer, whether the replacement action can be done without additional modification of the system or he must replace further parts such as E-code. The online part of the mechanism describes how the system executes the replacement action.

4.1. Offline

If the timing of the application has been changed, then standard tools can check for time and value determinism. If only the functionality code has been changed, then the following provides the offline check that lists what actions need to be done.

The set $\mathcal{F}$ contains all codeifiable functionality and the set F contains all application-specific functionality f. The set F' contains all new functionality f', that will replace F. Each application runs a number of tasks t out of the set of all tasks T. To define the relation between tasks and functionality, we use the mapping $uses : T \rightarrow F$ with $uses(t) = f$ for which f is the functionality that Task t requires and $uses^{-1}$ defines which functionality is used by which task. We define $old : F' \rightarrow F$ with $old(f') = f$ for which f' is the functionality that replaces f. We define $wcet : \mathcal{F} \rightarrow N$ to describe the worst-case execution time of one functionality. We define $LET : T \rightarrow N$ and $LET(t) = n$ for which n is an integer number and is the logical execution time of task t.

If $\forall f' \in F' : wcet(f') \leq wcet(old(f'))$ (1) holds, then the replacement action will be granted and require with no further changes of the application. If (1) does not hold and $\forall f' \in F' : wcet(f') \leq LET(uses^{-1}(f'))$ (2), then the developer will have to perform a schedulability check of the whole node, but will not necessarily have to change the timing specification. In the worst case, if $\exists f' \in F' : wcet(f') > LET(uses^{-1}(f'))$ (3), then the developer will have to do a schedulability check and generate new E-code for the node.

4.2. Online

The online behavior of the mechanism relies on time determinism of the underlying system. The new elements are loaded into the system and at a certain point in time t, the application switches from the old to the new version (e.g., from the old functionality to the new functionality). The moment t must guarantee, that the application is in a switch-able state, i.e., the replacement action does not alter the behavior of the application in an unintended way (e.g., replacing a function while the application is executing it). A time-deterministic system allows the developer to specify the moment t.

Listing 1 shows our implementation and how the system identifies this moment t is automatically at runtime. The update task runs concurrently with the E-machine. Whenever the processor is idle (i.e., all tasks released by the E-machine have finished execution), the update task will resume operation. The update task uploads the new elements on the target. During the registration, the update task prepares the elements to be used later. For example, it generates data structures for new functionality or calculates the new value of the program counter (PC) in case of modified E-code. After the registration has finished, the update task will assign the $updateFlag$ the value $true$.

The E-machine operates as usual, however, at the end of a mode period, it checks whether the $updateFlag$ has the value $true$. If yes, then it will update its internal function pointers and will change the E-code program counter to point to the new location. Similar to the concept of drivers and guards in TDL, this check and the assignments runs in logical zero time. The LET of the tasks and the internal logical time allows us to implement this: although the operation takes physical time, we do not update the logical time internally managed by the E-machine. Functionality switches WCET, similar to driver WCET, is accounted for in the WCET analysis of the task.

[

Update task:
```
upload
```

```
    register
    updateFlag ← true
||
  E-machine instrumentation:
    if ( endof(hyperperiod) ∧ updateFlag )
      F = F′
      pc = newPC
      active E code = new E code
      updateFlag ← false
    end if
]
```

Listing 1. Moment t in our implementation.

5. TDL for RTLinuxPro

RTLinuxPro (version 2.1) is a hard real-time, POSIX-compatible operating system. The real-time kernel at the heart of RTLinuxPro is built on top of the interrupt-control hardware and is responsible for the execution of real-time tasks. It treats the Linux kernel as a low priority task and implements real-time applications via kernel modules. These modules augment the kernel and the kernel executes these modules according to a selected scheduling scheme. If processing time is left (i.e., all real-time threads are idle), then the kernel will execute non–real-time tasks in the Linux environment.

Figure 1 shows an overview of the TDL infrastructure in RTLinuxPro. The E-machine and the application-specific modules augment the Linux kernel. A control panel runs in user space and allows the developer to interact with the running E-machine. The following sections describe the implementation in more detail.

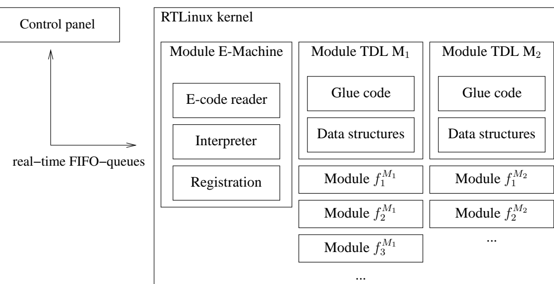


Figure 2. The TDL infrastructure implemented in RTLinuxPro.

5.1. RTLinuxPro Compiler Plugin

Although TDL itself is platform independent, the E-machine is not. To execute a TDL module on a specific platform, the TDL compiler generates platform-independent E-code but also platform-specific glue and functionality code.

- **E-Code [7].** Platform independent binary code called E-code describes the timing and data flow behavior of the TDL application. The E-Code file contains a full description of constants, types, structures, ports, tasks, drivers, guards and modes, as well as a set of instructions that will be sequentially executed by a specific platform dependent E-machine.

- **Glue code.** The E-code does only state the ID of a functions, which the E-machine has to call. The glue code provides means for the E-machine to call the functions specified in the E-code. It comprises port definitions, driver calls, guard calls, and RTLinuxPro specific functions such as module initialization or cleanup. The TDL compiler derives the code for the port, driver, and guard sections from the TDL program. The E-machine needs the RTLinuxPro specific section to load and unload TDL modules dynamically at runtime. The TDL compiler generates these sections through target-specific plugins such as the RTLinuxPro compiler plugin.

- **Functionality skeleton.** The TDL compiler generates a skeleton for the necessary functionality code. This code contains all function prototypes that are used in the TDL program and need to be implemented by the application developer. Figure 2 shows that each functionality is implemented by its own kernel module. The developer only has to code the functionality behavior though, the generated skeleton includes all other functions and data (e.g., module key, loading and unloading functions).

5.2. RTLinuxPro E-Machine

Figure 2 shows how the E-machine and two modules (M_1 and M_2) and their functionality augment the Linux kernel. The E-machine consists of three functional blocks: the E-code reader, the E-code interpreter, and the registration service (update task).

RTLinuxPro Applications. Each TDL module is a separate RTLinuxPro application. An RTLinuxPro application is equivalent to a Linux kernel module and therefore needs to provide standard kernel module life cycle methods such as `init_module` and `cleanup_module` (see [6]). One kernel module can spawn several real-time threads using the function `pthread_create`. Only threads created with this function have real-time properties. Since RTLinuxPro applications are kernel modules, they run in kernel space and have to communicate with user-space applications via real-time FIFO queues.

E-code Reader. During the initialization phase, binary E-code is passed to the E-machine via a real-time FIFO. The E-machine parses the E-code whenever there is time left and creates all necessary internal data structures needed for module registration and execution. The E-code reader gets a lower priority than the E-machine interpreter and real-time tasks and therefore does not interfere with the timeliness of concurrently executing modules. If the length of the E-code file and the worst-case utilization of the whole system is known, then the time needed for parsing can be calculated in advance.

Interpreter. The interpreter is responsible for the runtime behavior of the E-machine. It schedules modules dynamically and interprets their E-code instructions sequentially. Listing 2 shows the runtime thread routine. The function `getNextScheduledModule` returns either the first TDL module in the ready queue or `NULL` (in case no TDL module has been registered).

The E-machine executes the following steps for each TDL module. The function `runBlock` executes a sequence of E-code instructions[1] until it reaches a `return` instruction. Execution starts at the TDL module's current program counter, which is increased by one after each instruction. The instructions `call, if, jump, return` and `switch` are executed immediately, whereas the instructions `schedule` and `future` are stored to be used in one of the following functions: The function `dispatchTasks` evaluates the offline generated dispatch table and stored `schedule` instructions. It releases the scheduled tasks one after the other until either the time budget of the module has elapsed or there are no more scheduled tasks. If the budget of the module has elapsed, then the currently running task thread will get suspended. The intrinsic property of TDL similar to related timed-model computational systems, using only simple tasks with no synchronization points, allows us to implement this without a resource reservation mechanism. If there are no more tasks to release and the budget has not elapsed, then the E-machine will wait until the time budget of the current module will have elapsed. Low priority threads such as the update thread, can use this time.

At each point in time at most one task is executing. If a module is registered, then the runtime thread of the E-machine will be active. If the E-machine and all application tasks threads are idle and a module is in its initialization phase, then the update thread will be running. The priority requirements of Equation 1 of task priorities are crucial to guarantee timeliness.

$$\text{priority}(\text{E-machine runtime thread}) > \text{priority}(\text{Task threads}) > \text{priority}(\text{Initialization thread}) \quad (1)$$

[1] For a list of instructions and their explanation see [7].

The E-machine runtime thread controls all other threads. The task threads are time critical and must not be interrupted by initialization threads. However, they are also not allowed to interrupt the E-machine runtime thread. Finally, the update thread has the lowest priority and will run only, if all other threads are idle.

The remaining functions of Listing 2 are `setModeTime`, which updates the mode time, and the functions `calculateNewBudget` and `reschedule`, which handle the module-level scheduling.

```
while (true) {
  Module m = getNextScheduledModule();
  if (m != NULL) {
    runBlock(m);
    dispatchTasks(m);
    setModeTime(m);
    calculateNewBudget(m);
    reschedule();
  } else {
    runtimeThread.exit();
  }
}
```

Listing 2. E-machine runtime behavior.

Scheduling. The E-machine uses a two-level dynamic scheduling algorithm similar to the one proposed in [4]. The E-machine uses the first level to schedule TDL modules *(module-level scheduling)* and the second level to schedule tasks within modules *(task-level scheduling)*. For task-level scheduling, each module calculates an independent earliest deadline first (EDF) [3] schedule for each of its modes at module initialization. The whole processing time, assigned to the TDL module by module-level scheduling, is divided into fractions and is assigned to individual tasks according to the pre-calculated EDF schedule within the module.

Registration service. The registration service provides a service to dynamically register new TDL modules at the RTLinuxPro E-machine. It uses the module-loading mechanism of the Linux kernel. When a new TDL module is loaded, the kernel calls the `init_module` function of the kernel module that encapsulates the TDL module. This Linux specific function was previously defined by the TDL compiler and is used to register functionality code at the E-machine. The registration service provides a `register_module` function that is called from within `init_module`. The parameters of this function are used to pass function pointers to the E-machine, that correspond to the module specific functionality code. The developer does not care about these function calls, since they are generated by the TDL compiler.

The function pointers are a key element in the implementation of the functionality-replacement mechanism. At runtime, the E-machine follows the function pointers and invokes functionality code such as task drivers, guards, or task implementations. We use these function pointers to update/replace functionality code. When a module registers new functionality code, the update task stores the new function pointers, but the E-machine does not immediately use them. Whenever the mode period of a module's mode ends, the E-machine is allowed to switch to the new functionality code.

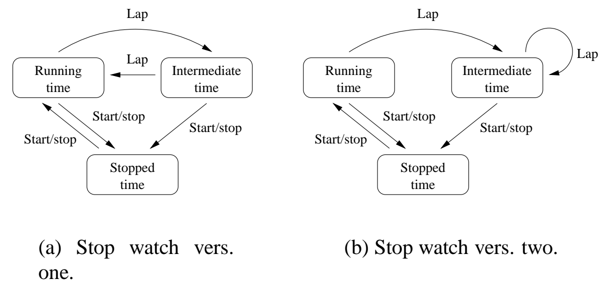
5.3. Control Panel

The control panel is a user-space Java application that gives control over the running E-machine and displays status information about the E-machine and loaded modules. For debugging purposes the control panel can force the E-machine to halt and resume execution (the button labeled *"Halt E-machine"*). The button labeled *"load new module"* opens a dialog to select a shell script to load a new module. The script needs to call the Linux `insmod` and `cat` programs to insert the module and pass its E-code to a FIFO queue. The button labeled *"unload"* calls the `rmmod` program to unload marked TDL modules. A module is marked, if the check box in the very right column of the table is enabled. The buttons labeled *"replace functionality"* and *"replace E-code"* execute shell scripts to replace the behavior of the marked module, respectively. Since functionality code is encapsulated in separate modules, it is replaced by unloading and loading the corresponding kernel modules. E-code is replaced by passing new binary E-code file to the E-machine through a FIFO queue.

The communication between the user-space Java application and the E-machine is implemented via real-time FIFO queues (see Figure 2). These FIFO queues cannot interrupt the real-time behavior, because reading from the FIFO queues is done by low priority threads and writing to them is non-blocking. If a FIFO queue is full, because the control panel does not read its content, then this content will be overwritten by further write calls. This can lead to incorrect panel information, but cannot interfere with the real-time application.

6. Case Study

As proof of concept, we implemented a case study that is simple, shows the key points of our work, and cannot be done with related work while guaranteeing value and time determinism on the local node when replacing functionality at runtime. The application is a simple stop watch with two buttons: start/stop and lap. In the first version of the software (see Figure 3(a)), the stop watch starts counting when



(a) Stop watch vers. one. (b) Stop watch vers. two.

Figure 3. Stop watch v1 and v2.

the start/stop button is pressed. It stops counting when this button is pressed again[2]. If the user presses the lap button while the stop watch is counting, then the display will show an intermediate time until the user presses the lap button again. Then the stop watch will continue to display the current running time again.

```
module StopWatch {
  sensor
    int S_startstop uses getButtonStartStop;
    int S_lap       uses getButtonLap;

  actuator
    int A_sec uses setSec;
    int A_tenth uses setTenth;

  task T_sec [150 ms] {
    input  int sens_lap;
           int sens_startstop;
    output int o_sec := 0;
    state  int s_sec := −1;
           int s_lap := 0;
           int s_startstop;
    uses t_sec_Impl(sens_lap,
           sens_startstop, o_sec,
           s_sec, s_lap, s_startstop);
  }

  task T_tenth [50 ms] {
    input  int sens_lap;
           int sens_startstop;
    output int o_tenth := 0;
    state  int s_tenth := −1;
           int s_lap := 0;
           int s_startstop;
    uses t_tenth_Impl(sens_lap,
           sens_startstop, o_tenth,
           s_tenth, s_lap, s_startstop);
  }

  start mode main [1000 ms] {
    task
      [1]  T_sec{sens_lap := S_lap;
            sens_startstop := S_startstop;}
      [10] T_tenth{sens_lap := S_lap;
            sens_startstop := S_startstop;}

    actuator
      [1] A_sec := T_sec.o_sec;
      [10] A_tenth := T_tenth.o_tenth;
  }
}
```

[2]For sake of brevity, it is not possible to reset the stop watch to zero.

Listing 3. TDL program of the stop watch application.

Listing 3 shows the TDL program of the stop watch. The two visible counters on the stop watch run autonomously, i.e., the task which displays seconds (T_{sec}) runs once per second and the task which displays tenths of seconds (T_{tenth}) runs ten times per second. Task T_{sec} and Task T_{tenth} do not communicate with each other. The functionality code for T_{sec} (`t_sec_Impl`) increases a counter each time it is executed (the timer value is stored in the state port `s_sec`). The functionality code for T_{tenth} (`t_tenth_Impl`) increments its counter by ten each time it is executed. The actuators A_{sec} and A_{tenth} update the external display of the stop watch.

The stop watch has two buttons: start/stop and lap. Figure 4 shows the functional behavior of version one of the stop watch. If the user presses the start/stop button, then the watch will start counting. If the user presses the lap button, then `t_sec_Impl` and `t_tenth_Impl` will continuously report the counter value at which the button has been pressed. However, the watch will continue to increase the internal counter (storing the intermediate time in `s_lap`). If the user presses the lap button again, then the two tasks T_{sec} and T_{tenth} will continue reporting the computed counter value of each execution (i.e., the running time).

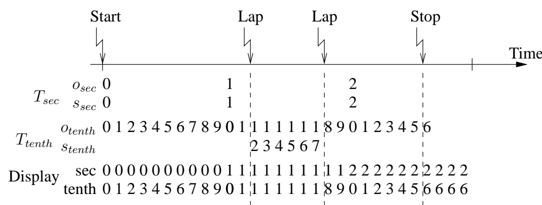


Figure 4. Value output of the stop watch version one.

6.1. Specifications & Timing Values

In our test lab, we have several workstations running RTLinuxPro to build research prototypes. These machines run an AMD 2600 processor mounted on a Matsonic 8167C motherboard, and have two Ethernet cards: one for Internet (a VT6102 Via Rhine II) and one for real-time communication (a 3C509c TX).

For our system, RTLinuxPro guarantees a worst-case execution time of a context switch of 15us and we use the provided “advance timer” concept of the system to take care of this extra time.

In our implementation, we do not use interrupts for accessing any kind of value (e.g., sensor values). We poll values, which is the natural implementation in the timed model. This eliminates most of the jitter and context switches. The remaining part is taken care of in the LET of the tasks. The semantic specification of the case study in TDL uses 45 lines of code that compiles into 242 lines of C code (exclusive the functionality code) and 130 E-code instructions (excluding the E-code header of 16 ports, 2 tasks, 14 driver, 1 mode).

6.2. Functionality Replacement

Now, we extend the functionality of the stop watch by replacing the implementation of the two tasks (`t_sec_Impl` and `t_tenth_Impl`). Their new functionality should be (see Figure 3(b)): whenever the user presses the lap button, the display shows the current intermediate time (instead of switching between intermediate time and current time as in version one). This new functionality code does not have a greater WCET than the previous version (just different constants in the assignment), so we do neither have to ship new E-code nor have to recalculate the utilization of the whole node. Also, we do not have to change the TDL program specified in Listing 3. Instead, we change the functionality code, compile it, generate a Linux kernel module, and load it into the kernel. The initialization mechanism inside the module triggers the update task of the E-machine. The update task registers the new functionality code and at the end of a mode period (in the example shown in Figure 5 at the end of the second mode period) the new functionality code takes effect.

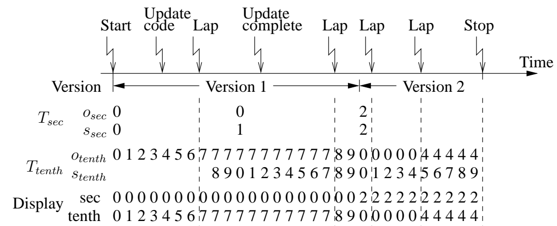


Figure 5. Replacing functionality: Output of the stop watch version one and after an update of version two.

6.3. Timing Replacement

To update the timing or the data-flow of tasks, we have to modify the TDL program. For the case study, we will change the resolution of the stop watch and decrease it to 200 milliseconds. Listing 4 shows the modified part of the TDL program. Line 6 and 11 now denote the new timing behavior. To decrease the resolution, we decreased the fre-

quency of Task T_{tenth} and the actuators from ten (see Listing 3) to five.

```
1 // ... prior code left out ...
   start mode main [1000 ms] {
3    task
       [1]  T_sec { sens_lap := S_lap;
5               sens_startstop := S_startstop;}
       [5]  T_tenth { sens_lap := S_lap;
7               sens_startstop := S_startstop; }

9    actuator
       [1] A_sec := T_sec.o_sec;
11     [5] A_tenth := T_tenth.o_tenth;

13   mode [1] if startStop( S_startstop )
              then stopped;
15 }
   // ... further code left out ...
```

Listing 4. Snipplet of the TDL program of stop watch version 3.

Using the timing/data-flow replacement mechanism, we now update the software at runtime. Figure 6 shows the timing of the update mechanism together with the timing of Task T_{tenth} prior to and after the update. During the first mode period, we issue the update request. It triggers an immediate release of the update task (T_{update}). The priority of the update task is lower than the priority of the E-machine and any of the application specific tasks. Consequently, the scheduler only executes it, when all other tasks are finished and the system would be idle. For example in Figure 6, the dispatcher does not resume T_{update} in the second mode period until Task T_{sec} has completed. T_{update} finishes execution near the end of the second mode period, however, the E-machine does not switch to the new timing until the end of the mode period. At the end of the second mode period shown in Figure 6, the registration service updates the timing behavior and the E-machine uses the new E-code from this moment on. As the figure shows in the third mode period, Task T_{tenth} is only executed five times a mode period[3].

7. Evaluation

The proposed mechanism is non blocking. This implies that the mechanism is free of locks and does not have a critical section. Consequently, its implementation does not require any synchronization mechanism such as a mutex or a semaphore and it does not interfere with the running hard real-time application in any circumstance. As TDL itself guarantees time and value determinism and the mechanism does not interfere with TDL, we can replace program functionality, timing, and data-flow of system and still guarantee time and value determinism.

[3]The output on the display now requires to multiply the value with two to display the correct tenth of a second.

The mechanism requires determinism in the time domain. Although the update task sets up the new functionality, timing, or data-flow while the system is idle, the essential migration from the old to the new version is time triggered. So, the proposed mechanism only works with systems that implement the timed model such as TDL [13] or Giotto [7].

The proposed mechanism and our implementation only supports replacing task timing and inter-task data-flow at the same time. This is neither a limitation of the mechanism nor of the implementation and they can easily be altered to support separate replacement of task timing and data-flow.

The implementation allows only one active update task at a time. To run several update tasks concurrently requires additional offline checks and different runtime behavior. The update task is initialized by the system developer and can be released at any time (it is event triggered). Since we cannot guarantee the timeliness of the update task that has started prior to the second update task — we can guarantee a WCET of the update tasks, for we know the utilization of the node — concurrent update tasks hinder time and value determinism of the update behavior of the system.

TDL modes complicate the WCET calculation and replacement actions. A mode consists of several, possibly concurrently executed task invocations and actuator updates. When the application switches from one mode to another, it may change the timing behavior and functionality of the application. Although modes and mode switches are defined prior to system deployment, they still complicate the calculation of the WCET and the replacement action. After a mode switch, the succeeding mode may lead to a higher utilization of the node, and then a mode dependent calculated WCET of the update task would be invalid. To guarantee the timeliness, the WCET of the update task is calculated for the highest possible utilization of the node. If the update task registers new E-code that does not include the mode to which the application would switch to at the end of the current mode period, then the system will fail. To prevent this, the new E-code must not reduce the number of available modes and "old" modes must remain in the application.

In the runtime environment, state information is kept in two places. System state is present in the input and output ports of the tasks, only local state is encoded in the task. Replacing one task with another results in loss of local state information, however, the system state is still present, since replacing the tasks does not flush the information already stored in the output ports. At the beginning of the new hyper period, the input ports of the task are immediately written.

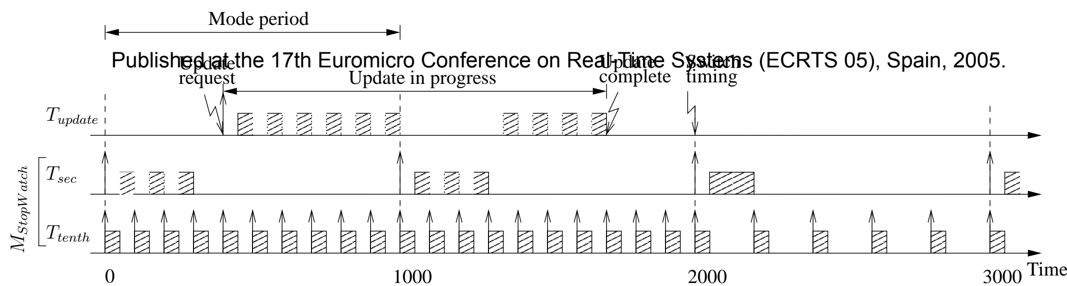


Figure 6. Replacing the timing of Task T_{tenth}.

8. Conclusions

In this paper, we proposed a mechanism for functionality, timing, and data-flow replacement for hard real-time systems. The mechanism is non-blocking as it does not require a critical section and does not interfere with the running hard real-time application. The mechanism trades promptness of the update request present in related work for time determinism of the replacement action, which is an important property for our further research. The mechanism itself is embedded in an infrastructure implemented in RTLinuxPro. Using this infrastructure, we present a case study of a stop watch, in which we exchange functionality of the lap button and decrease the timing resolution of the stop watch.

Although, we are now able to replace functionality, timing, and data flow at runtime in hard real-time systems, the evaluation shows that multi-mode applications complicate replacement actions and require special treatment by the developer. In our future work, we plan to further inspect functionality and especially timing replacement actions for multi-mode applications in the timed model and extend the work towards multi-node software replacement actions in distributed real-time systems.

9. Acknowledgments

We would like to thank the person, who uses the pseudonym “Der Herr Hofrat”, for providing help and hints via the RTLinuxPro mailing list, while we were implementing the infrastructure.

References

[1] Selected Topics in Embedded Systems Design: Roadmaps for Research, May 2004. ARTIST IST-2001-34820.

[2] L. Almeida. A word for operational flexibility in distributed safety-critical systems. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 177–186. IEEE Press, Jan. 2003.

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

[4] Z. Deng, J. W. S. Liu, and S. Sun. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. Technical Report UIUCDCS-R-96-1981, 1996.

[5] S. Fischmeister, editor. *Embedded Systems Knowledge Base — Austria and Surrounding Regions*. Austrian Computer Society, Sept. 2004.

[6] FSMLabs Inc. RTLinuxFree API calls. `http://www.fsmlabs.com/developers/man_pages/function_list.htm`, 2004.

[7] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.

[8] E. A. Lee. What's Ahead in Embedded Software. *IEEE Computer*, 33(9):18–26, Sept. 2000.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems Inc., second edition. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.ht%ml`.

[10] J. Montgomery. A model for updating real-time applications. *Real-Time Systems*, 27(2):169–189, 2004.

[11] B. Ravindran. Engineering dynamic real-time distributed systems: Architecture, system description language, and middleware. *IEEE Transactions on Software Engineering*, 28(1):30–57, 2002.

[12] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[13] J. Templ. TDL Specification and Report. Technical Report T002, Computer Science, University of Salzburg, 2004.

[14] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992. ISBN 0-201-56543-9.