Lock-Free Synchronization of Data Flow Between Time-Triggered and Event-Triggered Activities in a Dependable Real-Time System

Josef Templ, Johannes Pletzer, Wolfgang Pree
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract—**Time-triggered execution of periodic tasks provides the cornerstone of dependable real-time systems. In addition, there is often a need for executing event-triggered activities while the system would be otherwise idle. If time-triggered and event-triggered activities exchange information among each other, the data flow must be synchronized such that reading unfinished output data is avoided. We present a lock-free solution for these synchronization issues that is based exclusively on memory load and store operations and can be implemented efficiently on embedded systems without any operating system support. We also discuss the implications of our synchronization approach for the semantics of combined time-triggered and event-triggered execution in a dependable real-time system.**

Keywords-TDL; Lock-free; Synchronization; Time-triggered; Event-triggered; Synchronous; Asynchronous; Activity

I. INTRODUCTION

A dependable real time system performs safety critical tasks by periodic execution of statically scheduled activities [1]. The pre-computed schedule guarantees that the timing requirements of the system will be met in any case by taking the worst case execution time into account. Such operations are called time-triggered (alias synchronous) activities. The timing requirements of such activities are typically in the range of milliseconds or sometimes even below.

In addition, many dependable real time systems execute event-triggered (alias asynchronous) activities that are, for example, triggered by the occurrence of an external hardware interrupt or any other kind of trigger. Such asynchronous activities are not as time critical as synchronous tasks are, and can therefore be executed in a background thread while the CPU is otherwise idle.

Adding asynchronous activities to a time-triggered system could be done in a platform specific way by directly programming at the level of the operating system or task monitor. However, this approach has two drawbacks: (1) it is highly platform dependent and (2) it does not support proper synchronization of data exchanged between synchronous and asynchronous activities.

In order to tackle both problems we extended a TDL-based tool chain [2, 3] for time-triggered systems by asynchronous activities. TDL (Timing Definition Language) allows one to specify the timing behavior of a real-time system in a platform independent way that separates the specification of the timing behavior from the implementation of the tasks. We extended TDL by a notation for asynchronous activities and provided a runtime system for this extended TDL on a number of target platforms [4].

The resulting lock-free approach for data flow synchronization is not specific for TDL but—we believe—can be applied to other time-triggered systems that need to be extended with asynchronous activities. In the following we will therefore abstract from the concrete TDL syntax and focus on the underlying concepts only.

Our synchronization approach can be implemented efficiently without any operating system support such as monitors [5] or semaphores [6] and it avoids the need for dynamic memory allocation and the danger of deadlocks and priority inversions. It also keeps the impact of event-triggered activities on the timing of time-triggered activities as low as possible. For more information on nonblocking synchronization techniques please refer to [7, 8].

II. TIME-TRIGGERED ACTIVITIES

We assume that time-triggered activities have the highest priority in a dependable real-time system. The runtime system executes a pre-computed schedule and reads inputs and writes outputs at well-defined time instants, which are synchronized with a global time base such as the clock of a time-triggered bus system. There is always a distinguished time base which drives all time-triggered activities and that is why they are also called synchronous activities.

Asynchronous activities must not interfere with the timing properties of synchronous activities. This is achieved by running asynchronous activities in a thread with lower priority than synchronous activities. However, things get more complicated when synchronization of the data flow is involved, as will be described below.

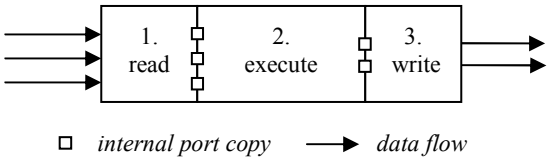

Figure 1. Assumed task model

III. ASYNCHRONOUS ACTIVITIES

The prototypical asynchronous activity is the invocation of an asynchronous task. It consists of (1) reading input data (also called input ports), (2) execution of the task's body, and (3) writing of output data (also called output ports). There may be other asynchronous activities as well (e.g. setting of actuator ports) but they do not introduce new problems because they can be seen as a special case of a task invocation. Figure 1 shows the task model that we assume.

The execution of a task's body is independent of the environment if input reading and output writing are separated from the implementation. Therefore we assume that internal copies of all input and output ports are maintained by the system. The task's body operates exclusively on these internal port copies.

Reading of input data may involve a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation, which has higher priority. Therefore we need to synchronize input data reading with the rest of the system such that all input ports are read atomically.

Like input data reading, writing of output data is a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation. It needs to be synchronized with the rest of the system such that all output ports are updated atomically.

A. Triggers for asynchronous activities

Asynchronous activities may be triggered by different events. We have identified the following three kinds of trigger events:

1) Hardware interrupt

A (nonmaskable) hardware interrupt has the highest priority in the system. It may even interrupt synchronous activities. We must therefore take care that the impact of hardware interrupts on the timing of synchronous activities is minimized. Hardware interrupts may be used e.g. for connecting the system with asynchronous input devices.

2) Asynchronous timer

A periodic or a single-shot asynchronous timer may be used as a trigger. Such a timer is independent from the timer that drives the synchronous activities because it introduces its own time base. An asynchronous timer may for example be used as a watchdog for monitoring the execution of the time-triggered operations.

3) Port update

Updating an output port may be considered an event that triggers an asynchronous activity. We assume that both a synchronous and an asynchronous port update may be used as a trigger event. In case of a synchronous port update, i.e. a port update performed in a time-triggered activity, we must take care that the impact on the timing of the synchronous activities is minimized. Port update events may e.g. be used for limit monitoring or for change notifications.

B. Semantics of asynchronous activities

Obviously, the triggering of an asynchronous activity must be decoupled from its execution. In addition, reading input ports for an asynchronous activity must be done at the time of execution, not at the time of triggering. Thereby we move as much work as possible into the asynchronous part and minimize the impact of trigger events on the timing of synchronous activities, which is particularly important for hardware interrupts and synchronous port updates.

If multiple different asynchronous activities are triggered, the question arises whether they should be executed in parallel or sequentially in a single thread. We opted for the sequential case because (1) on some embedded systems there is no support for preemptive task scheduling and (2) because data flow synchronization is simplified as will be shown later. In practice, we expect this not to be a severe restriction because time critical tasks will be placed in the synchronous part anyway.

We assume that asynchronous activities that are registered for execution may have different priorities assigned. The set of registered events thus forms a priority queue where the next activity to be processed is the one with the highest priority.

If one and the same asynchronous activity is triggered multiple times before its execution, the question arises if it should be executed only once or multiple times, i.e. once per trigger event. We opted for executing it only once because this avoids the danger of creating an arbitrary large backlog of pending activities at runtime if the CPU cannot handle the workload. In addition this decision also simplifies the mechanism for registering trigger events as will be shown later.

The following list summarizes our design decisions:

- Triggering of an asynchronous activity is decoupled from its execution.
- Reading input ports for an asynchronous activity is done at the time of execution, not at the time of triggering.
- Asynchronous activities are executed sequentially.
- The execution order of asynchronous activities is based on priorities.
- If one and the same asynchronous activity is triggered multiple times before its execution, it is executed only once.

IV. THREADING AND SYNCHRONIZATION

Figure 2 outlines the threads involved including their priority and the critical regions. The time-triggered activities are represented by a thread named TT-machine. This thread may need further internal threads but we assume that all synchronization issues are concentrated in a single thread that coordinates the time-triggered activities. It should also

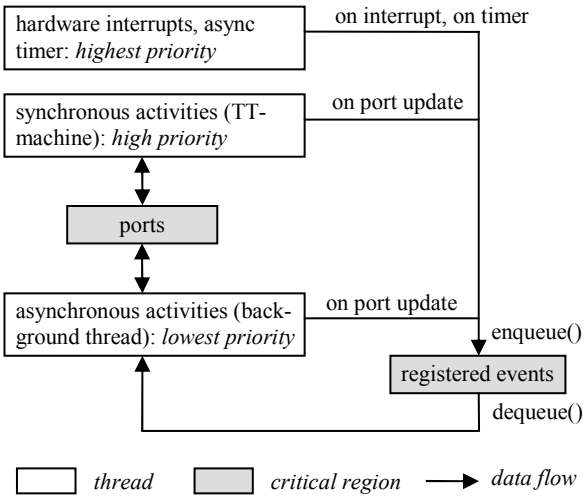

Figure 2. Threads and critical regions

be noted that an asynchronous timer thread could also run at a lower priority as long as it is higher than the priority of the asynchronous activities.

The following situations that need synchronization can be identified and will be described below in more details: (1) Access to the priority queue of registered events. (2) Reading the input ports for an asynchronous activity. This must not be interrupted by the TT-machine. (3) Updating the output ports of an asynchronous activity. This must be finished before the TT-machine uses the ports.

A. The priority queue of registered events

As mentioned before, asynchronous events are not executed immediately when the associated trigger fires but need to be queued for later execution by the background thread. Since asynchronous events may be associated with a priority, we need a data structure that allows us to register an event and to remove the event with the highest priority. Such a data structure is commonly referred to as a priority queue. It provides two operations enqueue and dequeue, which insert and remove an entry with the property that the element being removed has the highest priority. A number of algorithms exists for implementing priority queues with logarithmic behavior of the enqueue and dequeue operation. However, in our case it is more important to minimize the run time of enqueue in order to minimize its impact on the timing of synchronous activities.

Elements are enqueued when an asynchronous event occurs and the event is not yet in the queue. As mentioned earlier, an event can be a hardware interrupt, an asynchronous timer event, or a port update event. Port updates may origin from an asynchronous task or from a synchronous task that is executed by the TT-machine. enqueue will never be preempted by dequeue, however, enqueue may be preempted by another enqueue operation.

Elements are dequeued by the single background thread that executes asynchronous activities. This thread may be

	priority	pending
event 0	0	true
event 1	2	false
event 2	2	false
event 3	1	true

Figure 3. Array representation of trigger events

preempted by interrupts and by the TT-machine. Thus, dequeue may be preempted by enqueue operations.

As shown by the example in Figure 3 we chose an array representation of the triggerable events because this is both thread safe and provides for a fast and constant time enqueue operation. We use a Boolean flag per event that signals that an event is pending. The flag is cleared when an event is dequeued. From that time on it may be set again when the associated trigger fires. The flag remains set when the same trigger fires again while the flag is already set. The thread-safe enqueue operation boils down to a single assignment statement and the dequeue operation becomes a linear search for the event with the highest priority over all pending events. Registering an event from an unmaskable interrupt or from a synchronous port update thereby has only a negligible effect on the timing behavior of synchronous activities. The linear search in the background thread is expected to be acceptable for small to medium numbers of asynchronous events (< 100), which should cover all situations that appear in practice.

It should be noted that the array representation of the priority queue does not impose any restricion on the number of events the system can handle. There is one array element for every trigger and the number of triggers is known statically. Thus, the array can always be defined with the appropriate size.

The background thread for executing asynchronous operations is a simple infinite loop that runs with lower priority than the TT-machine thread. For a particular target platform there may be some refinements with respect to the CPU load, which is increased to 100% by permanently polling the event queue.

```
static Thread asyncThread = new Thread() {
  public void run() {
    for (;;) {
      int next = dequeue();
      if (next >= 0) {
        executeEvent(next);
      }
    }
  }
};
```

The procedure executeEvent is supposed to execute the asynchronous activity identified by next. Within its implementation there will be synchronization issues with respect to reading input ports and writing output ports as

described below. Instead of showing the complete implementation, which depends on the particular environment, we will focus on the synchronization issues only.

B. *Reading the input ports for an asynchronous task*

While performing asynchronous reading of input ports the following situation may arise: An asynchronous input port reading involving multiple input ports (or at least multiple memory load operations) has been started. The first port has been copied. The second port has not yet been copied but the TT-machine preempts the background thread and updates the source ports. When the background thread continues it would read the next port, which has a newer value than the first port. Moreover, this situation may in principle occur multiple times when the TT-machine preempts the background thread after the second port has been read, etc. We have to make sure that reading all of the input ports is not preempted by the TT-machine. Since asynchronous activities don't preempt each other, we know that there can only be one such asynchronous input port reading that is being preempted. Therefore we can introduce a global flag that is set by the TT-machine in order to indicate to the background thread that it has been preempted. The background thread then has to repeat its reading until all of the ports are read without any preemption. The following Java code fragments outline a possible implementation.

Asynchronous port reading within executeEvent uses a loop in order to wait for a situation where input port reading is not preempted by the TT-machine. Therefore, our solution does not qualify as a wait-free nonblocking algorithm [7]. It should be noted, however, that (1) starvation cannot occur in the TT-machine and (2) in practice it does also not occur in the background thread because even in the unlikely case that the TT-machine's schedule reserves 100% of the CPU, this refers to the worst case execution time, which typically will not always be required.

```
do {
  ttmachineExecuted = false;
  //copy input ports
  ...
} while (ttmachineExecuted);
```

The relevant TT-machine code, which is assumed to be placed in a central procedure of the TT-machine named ttmachineStep may look like this:

```
void ttmachineStep() {
  ttmachineExecuted = true;
  //perform operations for this time instant
  ...
}
```

C. *Updating the output ports of an asynchronous task*

In the case of asynchronous output port updates the following situation may arise: An asynchronous output port update involving multiple output ports (or at least multiple memory store operations) has been started. The first port has been copied. The second port is not yet copied but the TT-machine preempts the background thread and reads both output ports. Now one port is updated but the second is not. Since this interruption cannot be avoided, we must find a way for proper synchronization.

Since we assumed earlier that updating the output ports is separated from the implementation of a task, we can encapsulate the output port update operations of a task in a helper procedure that we call the task's *termination driver*. Since asynchronous activities don't preempt each other, we know that there can only be one such termination driver being preempted and it suffices to make that very instance available to the TT-machine by means of a global variable. Whenever the TT-machine performs its next step, it checks first if a termination driver has been interrupted. If so, it simply re-executes this driver! This means that the driver may be executed twice, once by the background thread and once by the TT-machine. This is only possible if the driver is *idempotent* and *reentrant*, i.e. its preemption and repeated execution does not change its result. Fortunately, termination drivers have exactly this property because they do nothing but memory copies and the source values are not modified between the repeated driver executions. The source values are the internally available results of the most recent invocation of this asynchronous task and only a new task invocation can change them. Such a task invocation, however, will not happen because the background thread executes all asynchronous activities sequentially.

It should be noted that the property of idempotency does not hold for copying input ports as discussed in the previous subsection because a preemption by the TT-machine may alter the value of a source port that has already been copied. This means that we really need two ways of synchronization for the two cases.

It should also be noted that setting the driver identity must be an atomic memory store operation. If storing e.g. a 32 bit integer is not atomic on a 16-bit CPU, an additional Boolean flag can be used for indicating to the TT-machine that a driver has been assigned. This flag must of course be set after the assignment of the driver's identity. If this initial sequence of assignments is preempted, the TT-machine will not re-execute the driver and that is correct because the driver has not yet started any memory copy operations.

The following Java code outlines the implementation of asynchronous task termination drivers and the corresponding code in the TT-machine. Setting, testing and clearing the driver identity is kept abstract because the details may vary between target platforms. Since Java lacks function pointers we use an integer id and a switch statement instead. Variations, e.g. using C function pointers or Java singleton classes, are of course possible.

```
void callDriver(int id) {
  switch (id) {
    ...
    case X: //termination driver for async task X
      assignAsyncTerminateDriverID(X);
      //perform memory copy operations
      ...
      clearAsyncTerminateDriverID();
      break;
```

```
    ...
  }
}
```

The relevant TT-machine code including the code introduced in the previous subsections looks like this:

```
void ttmachineStep() {
  ttmachineExecuted = true;
  if (asyncTerminateDriverIDassigned()) {
    callDriver(asyncTerminateDriverID);
  }
  //perform operations for this time instant
  ...
}
```

It suffices to clear the registered termination driver at the end of the termination driver itself. There is no need to do it after callDriver() in ttmachineStep because the driver's re-execution will clear it anyway.

The resulting runtime overhead for supporting asynchronous operations in the TT-machine is the assignment of the ttmachineExecuted flag and the test for the existence of a preempted asynchronous task termination driver, which is acceptable because this happens only once per TT-machine step. In case of preempting such a driver the time for re-execution must be added. When a port update trigger is used, then the enqueue operation is also a small constant time overhead that affects the TT-machine. There is no other runtime overhead for integration of event-triggered activities in the TT-machine.

V. Implementation State

We have implemented the proposed solution in the context of a TDL-based tool chain. Currently we support two target platforms, (1) the dSPACE MicroAutoBox, which is a widely used prototyping platform for embedded systems in the automotive industries, and (2) the NODE RENESAS platform provided by DECOMSYS (now Elektrobit). Both systems are programmed in C and support a FlexRay bus interface and the time-triggered activities are synchronized with FlexRay's global time base. The availability of a high-level description language (TDL) for timing properties as well as for asynchronous activities allowed us to generate the required glue code such as the event table, the termination drivers and all the code needed for the background thread and for data flow synchronization automatically. Even when we added support for distributing the data flow across multiple nodes we relied on the data flow synchronization approach presented in this paper. Currently we are working on TDL implementations for further platforms including a bare hardware based on an ARM7.

Table I shows the time needed for various operations on different platforms. The platform named *MicroAutoBox* uses a PowerPC 750FX CPU running at 800 MHz and the Microtec C compiler version 3.2 with optimization level 5. The platform runs the dSPACE Real-Time Kernel as its operating system. The platform named *ARM* uses an ARM7 TMTI CPU running at 80 MHz and the GNU C compiler with optimization level 2 and runs without an operating system. The platform named *RENESAS* uses a Renesas M32C/85 CPU running at 24 MHz and the GNU C compiler version 4.1 with optimization level 3. The platform runs the Application Execution System (AES) provided by DECOMSYS and executes the programs from read-only memory, which slows down the execution. This system does not support external interrupts for user level programs.

TABLE I. Measurement results [nanoseconds]

Platform (MHz)	Interrupt	Port Update	dequeue N
MicroAutoBox (800)	420	8	11 * N + 60
ARM (80)	700	200	287 * N + 500
RENESAS (24)	N.A.	1200	790 * N + 2500

The column *Interrupt* shows the time needed for an external hardware interrupt trigger, which includes the interrupt handling overhead and the enqueue operation. The column *Port Update* shows the time needed for a synchronous port update trigger, which consists only of the enqueue operation. The column *dequeue N* shows the time needed for the search for the next event to be processed as a linear function of the array size *N*. All timings are given in nanoseconds.

The values shown in the columns *Interrupt* and *Port Update* are critical for the timely execution of synchronous operations as they impose an overhead that may affect the TT-machine. Even on the slowest platform the required time is only slightly above one microsecond. In comparison with the *ARM* platform, the *Interrupt* time for *MicroAutoBox* shows that the operating system introduces a significant overhead.

The values in the column *dequeue N* only affect the background thread and are not visible to the TT-machine. On the slowest platform a time of 81.5 microseconds results for $N = 100$, which means that response times in the range of milliseconds can easily be achieved for asynchronous operations.

VI. Related Work

The xGiotto language [9] also aims at the integration of time-triggered and event-triggered activities. xGiotto's compiler is supposed to perform a static check for the absence of race conditions. Due to the specific design of xGiotto, a precise check is possible but not in polynomial time. Therefore, only a conservative check is done in the compiler. We do not need such a check at all as we defined appropriate semantics for event-triggered activities and use the proposed synchronization mechanisms for their integration into a time-triggered system.

RT-Linux [10] is an extension of the Linux operation system which adds a high priority real-time kernel task and runs a conventional Linux kernel as a low priority task. Its interrupt handling mechanism is similar to what we propose for the event queue as all interrupts are initially handled by the real-time kernel and are passed to a Linux task only when there are no real-time tasks to be run. In our approach, the only immediate reaction to an interrupt is its registration in the priority queue so that it can be processed later when no time-triggered activity is executed.

In [11] a nonblocking write (NBW) protocol is presented. The writer is executed by a separate processor and is not blocked. It updates a concurrency control field (CCF) which indicates whether it currently writes data to a shared variable. The reader uses the CCF to loop until no write operation is executed while it reads from the shared data structure. This relates closely to our synchronization strategy for reading input ports for an asynchronous activity. In our case the writer would be the TT-machine which is not blocked.

A comprehensive overview of the field of nonblocking synchronization can be found in [8]. Among other techniques, it also describes a so-called roll-forward synchronization approach by means of a helper function, which looks similar to the one we used for synchronizing output port writing.

VII. CONCLUSIONS

We have shown that a nonblocking lock-free solution for data flow synchronization between time-triggered and event-triggered activities in a dependable real-time system is indeed possible. Our solution does not need any operating system support such as monitors or semaphores and thereby avoids dynamic memory operations and the danger of deadlocks and priority inversions. There is also no need for switching off interrupts and the solution also works in a shared-memory multiprocessor system where the time-triggered and event-triggered activities are performed on separate CPUs. Our approach relies exclusively on atomic memory load and store operations, which are provided by every CPU in hardware. An appropriate semantics for asynchronous activities helped us to keep the solution simple and efficient.

ACKNOWLEDGMENT

We want to thank Peter Hintenaus for providing us with the measurement results for the ARM platform.

REFERENCES

[1] H. Kopetz, "Real-Time Systems - Design Principles for Distributed Embedded Applications," ISBN 0792398947, Springer, 2007.

[2] W. Pree and J. Templ, "Modeling with the Timing Definition Language (TDL)," Proceedings ASWSD 2006, LNCS 4922, 133-144, Springer, 2008.

[3] J. Templ, "Timing Definition Language (TDL) Specification 1.5," Technical Report, University of Salzburg, 2008, http://softwareresearch.net/pub/T024.pdf.

[4] J. Templ, J. Pletzer, and A. Naderlinger, "Extending TDL with Asychronous Activities," Technical Report, University of Salzburg, 2008, http://softwareresearch.net/pub/T022.pdf.

[5] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," Comm. ACM 17 (10), 549–557, 1974.

[6] E. W. Dijkstra, "Cooperating sequential processes," in "Programming Languages," Academic Press, New York, 1968.

[7] M. P. Herlihy, "A Methodology For Implementing Highly Concurrent Data Structures," Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York, 1990.

[8] M. B. Greenwald, "Non-Blocking Synchronization and System Design," PhD Thesis, CS-TR-99-1624, Stanford U., 1999.

[9] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido, "Event-driven programming with logical execution times," in "Hybrid Systems Computation and Control," Lecture Notes in Computer Science 2993, Springer, 2004.

[10] V. Yodaiken and M. Barabanov, "A Real-Time Linux," Proceedings of the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA, 1997.

[11] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW," Proceedings of the 14th IEEE Symposium on Real-Time Systems, 131-137, IEEE, New York, 1993.