Extending TDL with Asynchronous Activities

Josef Templ, Johannes Pletzer, Andreas Naderlinger

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

Technical Report T022
October 2008

Extending TDL with Asynchronous Activities

Step 1: Asynchronous data-flow on local node

Josef Templ, Johannes Pletzer, Andreas Naderlinger

Oct 22nd 2008 / SVN Revision 1128

ABSTRACT

This report describes the extension of the Timing Definition Language (TDL) with asynchronous activities. Asynchronous activities are activities that are executed in the background while synchronous activities are performed by the TDL E-machine. Asynchronous activities do not change the timing behavior of synchronous activities. However, it is allowed that synchronous and asynchronous activities exchange data in a properly synchronized way. We describe the TDL language extensions, ecode file format extensions, the threading model and synchronization between the synchronous E-machine und asynchronous activities, and platform plug-in extensions for a number of target platforms including a prototypical Java platform, AES/NODE<RENESAS>, dSPACE/MicroAutoBox, and the MATLAB/Simulink environment.

In this first step towards full support of asynchronous activities in TDL, we do not support asynchronous data-flow between remote nodes but focus only on asynchronous data-flow on local nodes.

Table of Contents

List of Figures

1. Introduction

A dependable real time system performs safety critical tasks by periodic execution of statically scheduled activities. 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 (wcet) into account. Such operations are also called *synchronous* (alias *time triggered*) activities.

In addition, many dependable real time systems execute *asynchronous* (alias *event triggered*) 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 periodic tasks are, but can be executed in a background thread while the CPU is idle otherwise.

The Timing Definition Language (TDL) supports the platform independent specification of the synchronous aspects of a real time system. Adding asynchronous activities 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 platform dependent and (2) it does not support proper synchronization of data exchanged between synchronous and asynchronous activities. Therefore we extended TDL by a notation for asynchronous activities and provided a runtime system for this extended TDL on a number of target platforms.

2. TDL Language Extensions

Asynchronous activities are introduced at the level of the TDL module construct. Every module may optionally declare asynchronous activities as the last section within the module construct. In the following paragraphs we show only the extension of the grammar. For a complete grammar please refer to [TDL1.5spec].

```
TDLModule ::=
  "module" qualIdent "{"
    ...
    ["asynchronous" asyncDecl]
  "}".
```

An asynchronous activity in TDL is an activity that is carried out in the spare time between execution of timed (synchronous) activities and thereby does not disturb the real time properties of a system. Its execution may be triggered by a variety of events. Asynchronous activities are never preempted by other asynchronous activities but may be preempted by synchronous activities. The TDL runtime system takes care of the synchronization of the data flow between synchronous and asynchronous activities such that reading input ports, updating output ports, and performing actuator updates are atomic actions.

```
asyncDecl ::= "{" {asyncSequence} "}" .
asyncSequence ::="[" asyncEvent "]" guard
  { taskDesignator inputParams ";"
```

| *actPortDesignator ":=" portDesignator ";"*

} .

TDL supports the grouping of asynchronous activities into sequences that are triggered as one unit and executed strictly sequential. Any such sequence has an associated trigger event, an optional guard, and a sequence of asynchronous activities. An asynchronous activity may be a task invocation or an actuator update. A task may either be invoked synchronously or asynchronously but not both. Also, an actuator update must either be done synchronously or asynchronously but not both.

Triggering an asynchronous activity sequence means that the sequence is registered for execution at some later time at the discretion of the TDL runtime system. Any additional triggering of a registered activity sequence is ignored until the execution of this activity sequence starts. Parameter passing takes place as part of the execution not at the time of registration.

asyncEvent ::= eventAttr ["," priorityAttr] .

eventAttr ::= attrName "=" (portDesignator | constExpr) .

priorityAttr ::= attrName "=" constExpr.

The kind of event that triggers the execution of an asynchronous activity sequence is specified by the attribute name `interrupt`, `timer`, or `update`. In case of an interrupt, the attribute value must be an integer greater or equal to zero. This logical interrupt number may need to be mapped to platform specific interrupt specifications outside the TDL source code. In case of a timer, the attribute value must be an integer greater than zero. It describes the period of a timer in microseconds. In case of a port update, the attribute value must be the name of an output port. Whenever this port receives a value it triggers the asynchronous activity sequence.

The priority is specified by the attribute name `priority` and a value greater or equal to zero where higher numbers mean higher priority. The priority attribute affects the queuing order of registered asynchronous activity sequences and should not be mixed up with a thread priority level. The default priority is the lowest value.

Example:

```
asynchronous {
  [interrupt=1, priority=5]
  if guard1(s1) then t1(s1, t.o); a1 := t1.o;
  [timer=10ms, priority=4]
  t2(s2);
  [update=t1.o]
  t3();
}
```

3. ecode File Format Extensions

The ecode file format has been extended by a section for describing the asynchronous declarations of a module in a straight forward way. The following paragraphs describe the changes of the ecode file format. For a complete definition please refer to [TDL1.5spec].

ECodeFile ::= 'E' 'C' '1' '0' moduleName:string pubKey:int4 moduleKey:int4
...
0x88 Asyncs
0x89 Ecodes.

The ecode file format version has been increased to `EC10` and the new section `Asyncs` has been introduced right before `Ecodes`. Therefore the section number of Ecodes has been increased by 1. The reason for introducing Asyncs before Ecodes is that some tools (e.g. the compiler) don't need to read the Ecodes section and are able to stop reading earlier. Two new kinds of drivers have been introduced.

Drivers ::= nofDrivers:int4
{...
| asyncRelease:0x7 srcPorts:PortList dstPorts:LocalPortList
| asyncActuator:0x8 srcPort:QualPortID actPortID:int4
}.

Asyncs ::= nofAsyncs:int4 {AsyncEvent guardID:int4 AsyncActs}.

AsyncEvent ::=
interrupt:0x0 number:int4
| timer:0x1 period:int4
| update:0x2 port:QualPortID.

AsyncActs ::= nofAsyncActs:int4
{ task:0x0 taskID releaseDriverID:int4
| actuator:0x1 updateDriverID:int4
}.

4. Threading and Synchronization

The extensions of TDL for asynchronous activities have been carefully designed for allowing an implementation without the need for synchronization primitives such as semaphores or monitors. The reasons why we try to avoid these constructs are that (1) they may introduce deadlocks and priority inversions and (2) they may not be available on all target platforms. The only synchronization primitive that we want to rely on is the atomic memory store operation of a CPU.

Fig. 1 outlines the involved threads including their priority and the critical regions involved. Please note that a timer thread could also run at a lower priority as long as it is higher than the priority of the asynchronous activities.

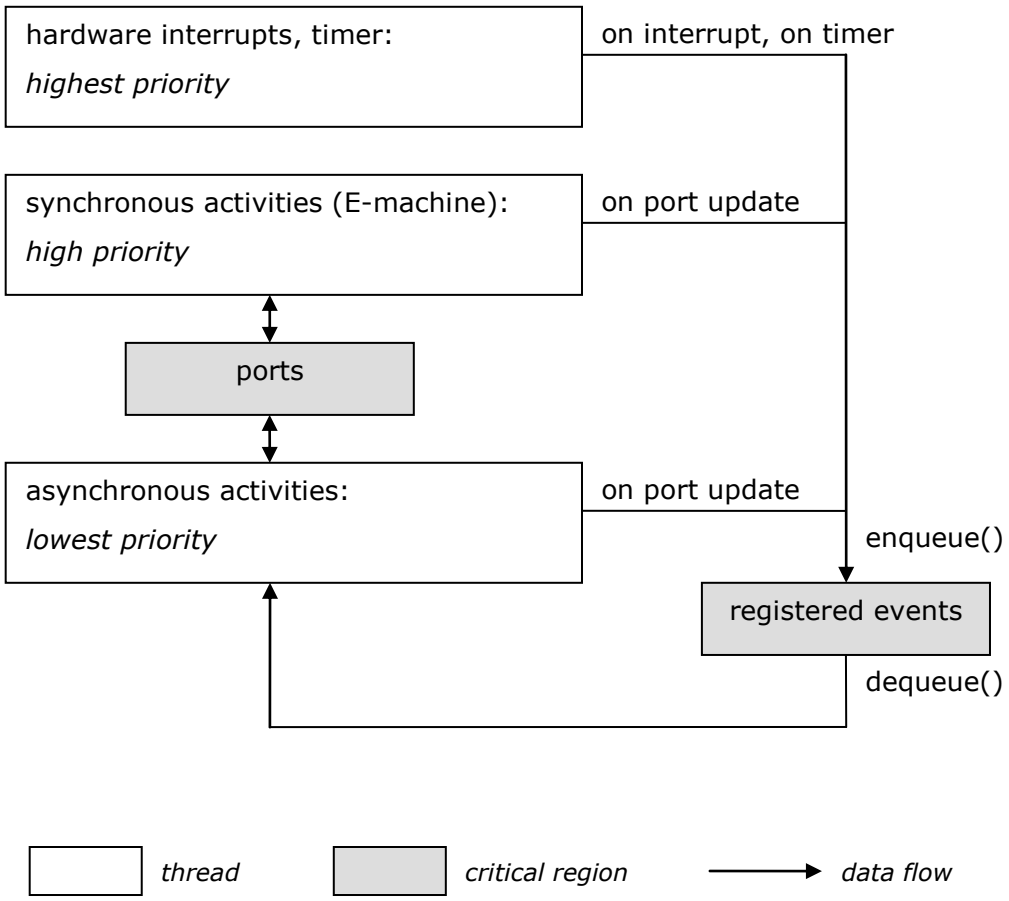


Fig. 1 Threads and critical regions

The following situations that need synchronization can be identified:

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 E-machine.
3. Updating the output ports of an asynchronous activity. This must be finished before the E-machine uses the ports.

4.1. The priority queue of registered events

Asynchronous events in TDL are not executed immediately when the associated trigger fires but need to be queued for later execution by the background thread. Since asynchronous events in TDL are 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 important property that the element being removed has the highest priority. A number of tree-based or heap-based algorithms exist for implementing priority queues with logarithmic behavior of `enqueue` and `dequeue`. As we will see, however, in our case it is more important to minimize the effect of `enqueue` to a small constant time that can be ignored safely. In addition, we try

to avoid the need for advanced synchronization primitives, which rules out logarithmic priority queue algorithms.

Elements are enqueued when an asynchronous event occurs and the event is not yet in the queue. An event can be a hardware interrupt including a non-maskable interrupt, a 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 E-machine. `enqueue` will never be preempted by `dequeue`, however, `enqueue` may be preempted by another `enqueue` operation.

Elements are dequeued by the background thread that executes asynchronous activities. This thread may be preempted by interrupts and by the E-machine.

A simple Boolean array representation of the registered events is both thread safe and provides for a 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 one and the same trigger fires again while the flag is already set. The `enqueue` operation works with a small constant time and the `dequeue` operation becomes a linear search over all possible 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 is expected to be acceptable for small to medium numbers of asynchronous events (< 100), which covers all situations that appear in practice.

The following Java code fragments show the implementation of the priority queue. The particular example being shown has 5 asynchronous events with priorities ranging from 0 to 2. Only the initialization of the table is application specific and will typically be generated automatically. Visibility modifiers have been stripped off for simplicity. The class `AsyncSequenceDesc` describes the runtime data needed for an entry of the priority queue. The indices of the array `asTable` correspond with the indices used in `enqueue` and `dequeue`.

```
class AsyncSequenceDesc {

  boolean pending;
  final int priority;

  AsyncSequenceDesc(int priority) {
    this.priority = priority;
  }
}


//table of all AsyncSequenceDescs

static AsyncSequenceDesc[] asTable;

static { // application specific initialization
  asTable = new AsyncSequenceDesc[5];
  asTable[0] = new AsyncSequenceDesc(1);
  asTable[1] = new AsyncSequenceDesc(2);
  asTable[2] = new AsyncSequenceDesc(2);
  asTable[3] = new AsyncSequenceDesc(2);
  asTable[4] = new AsyncSequenceDesc(0);
}
```

```
//add to priority queue unless it is already in queue
static void enqueue(int idx) {
  asTable[idx].pending = true;
  asyncThread.interrupt(); //see Note 1
}

//remove from priority queue
static int dequeue() {
  int maxPriority = -1;
  int maxPriorityIndex = -1;
  for (int i = 0; i < asTable.length; i++) {
    AsyncSequenceDesc as = asTable[i];
    if (as.pending && as.priority > maxPriority) {
      maxPriority = as.priority;
      maxPriorityIndex = i;
    }
  }
  if (maxPriorityIndex >= 0) {
    asTable[maxPriorityIndex].pending = false;
  }
  return maxPriorityIndex;
}
```

Note 1: If the Java background thread for executing asynchronous events polls the priority queue without sleeping, the CPU load is increased to 100%. A more sophisticated solution is to let the background thread sleep when it finds an empty event queue and wake it up when the next event arrives. This is achieved by calling `asyncThread.interrupt()`. For any target platform the polling behavior of the background thread must be carefully examined.

The background thread for executing asynchronous operations is a simple endless loop that runs with lower priority than the E-machine thread as shown below. The presented solution also contains a sleep mechanism for reducing the CPU load when no asynchronous events are available.

```
static Thread asyncThread = new Thread() {
  public void run() {
    for (;;) {
      int next = dequeue();
      if (next >= 0) {
        executeAsyncSequence(next);
      } else {
        try {Thread.sleep(10);}
        catch (InterruptedException x) {
          //an async event has arrived while sleeping, see Note 1
        }
      }
    }
  }
};
```

4.2. Reading the input ports for an asynchronous activity

Reading of input ports for an asynchronous activity occurs upon several occasions. All of them share the common property that reading must appear as

an atomic action, i.e. that it is not preempted by an activity that potentially modifies one of the ports being read. The following places for reading input ports for an asynchronous activity have been identified.

1) A guard is evaluated asynchronously. All guard input ports must be read at once.
2) A task is invoked asynchronously. All task input ports must be read at once.
3) An actuator update is performed asynchronously. The source port of the actuator update must be read. Although this affects only a single port, this may involve more than a single memory read operation e.g. when the type of the port is `long`.

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 E-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 E-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 E-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 E-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 show a possible implementation.

asynchronous port reading

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

relevant E-machine code:

```
void emachineStep() {
  emachineExecuted = true;
  //interpret ecodes for this time instant
  ...
}
```

4.3. Updating the output ports of an asynchronous activity

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 E-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 of asynchronous output port updates. The driver structure being used by TDL E-machines comes to a rescue here.

Writing to output ports is always encapsulated in an auxiliary procedure called a *task termination driver*, which is called by the E-machine in order for performing application specific operations such as copying output port values. The E-machine only knows the driver's identity (via the ecode being executed) and the driver implementation knows what memory should be copied. Since asynchronous activities don't preempt each other, we know that there can only be one such asynchronous output port update being preempted. Therefore we encapsulate the asynchronous output port update into a driver and make the driver's identity available to the E-machine. Whenever the E-machine performs its next step, it checks first if such a driver has been interrupted. If so, it simply re-executes this driver. This means that the driver will be executed twice, once by the background thread and once by the E-machine. This is only possible if the driver is idempotent, i.e. its repeated execution does not change its result. Fortunately, task terminate drivers have exactly this property because they do nothing but memory copies and the source values are not modified between the repeated executions. The same does not hold for copying input ports as discussed in the previous section because a preemption by the E-machine may alter the values of a source port already copied. This means that we really need two ways of synchronization for the two cases.

It should 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, an additional boolean flag can be used for indicating to the E-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 E-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 E-machine. Setting, testing and clearing the driver identity is kept abstract because the details may vary between target platforms.

asynchronous task termination driver structure:

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

    ...
  }
}
```

relevant E-machine code:

```
void emachineStep() {
  emachineExecuted = true;
  if (asyncTerminateDriverIDassigned()) {
    call(asyncTerminateDriverID);
  }
  //interpret ecodes for this time instant
  ...
}
```

The resulting runtime overhead for support of asynchronous operations in the E-machine is the assignment of the `emachineExecuted` flag and the test for the existence of a preempted asynchronous task terminate driver, which is acceptable because this happens only once per logical time instant. 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 E-machine. There is no other runtime overhead involved in the E-machine.

4.4. Asynchronous sensor read

Reading of a sensor value may be involved in all of the places where input ports are read. Since reading input ports is protected against preemption by the E-machine, this applies also to embedded sensor reads. The following code fragment shows the complete picture.

asynchronous port reading including sensors

```
do {
  emachineExecuted = false;
  //read and update required sensor ports
  ...
  //copy input ports
  ...
} while (emachineExecuted);
```

A more subtle situation arises in connection with reading of sensors by the E-machine. For reasons of consistency, the E-machine avoids reading a sensor twice at one and the same logical time instant but reuses the value already read. For that purpose additional internal state per sensor is maintained by the driver that reads a sensor and updates the sensor port. When a sensor is read by an asynchronous activity, this logic must be circumvented. We cannot simply read sensors the same way as the E-machine does but must call the getter function of the sensor directly, which results in reading the sensor every time it is requested by an asynchronous activity. Thus, asynchronous activities use the current value of a sensor at the time the sensor is requested. For the (rare) case of using a sensor multiple times in an asynchronous activity, e.g. if a sensor appears multiple times in the argument list of a guard, we read the sensor only once. Whenever the sensor is read, the corresponding sensor port is updated and for copying the input ports this sensor port value is used. It should be noted that this does not interfere with the E-machine semantics because the sensor port is only used at logical time instants and never in between.

5. Platform Implementations

5.1. AbstractNodePlatform

The TDL tool chain involves an abstraction for platform specific code generation both at the level of clusters and at the level of nodes and thereby represents a *plug-in architecture* that can be extended for an open set of target platforms. The contract for plug-in classes at the node level is defined in the interface `NodePlatform`. As a convenience class, the abstract base class `AbstractNodePlatform`, which implements this interface and provides additional

convenience features, is available. All our node platform classes extend this abstract class.

For the purpose of supporting target specific code generation for asynchronous activities we provide some base functionality in the class `AbstractNodePlatform`. In particular, we provide a method that prepares auxiliary data structures that are expected to be required by all target specific node plug-in classes as outlined below. Preparing the async data structures takes into account which modules are placed on a particular node and which are imported from a remote node. The latter are referred to as stub-modules. The TDL compiler provides the involved data structures (`AsyncDecl`, `QualPortID`, `FunCall`, `TaskDecl`) as part of the abstract syntax tree of a module.

All async event sequences from non-stub modules on this node:

```
List<AsyncDecl> asyncs;
```

Maps all async interrupt numbers to async event sequences from non-stub modules on this node:

```
SortedMap<Integer, List<AsyncDecl>> asyncInterruptMap;
```

Maps all async timer periods to async event sequences from non-stub modules on this node:

```
SortedMap<Integer, List<AsyncDecl>> asyncTimerMap;
```

Maps all update ports to async event sequences from non-stub modules on this node:

```
Map<QualPortID, List<AsyncDecl>> asyncUpdateMap;
```

All async guard calls from non-stub modules on this node:

```
List<FunCall> asyncGuards;
```

All async tasks from non-stub modules on this node:

```
List<TaskDecl> asyncTasks;
```

Prepares all async data structures by iterating over all non-stub modules and async sequences. This method must be called explicitly by subclasses.

```
void prepareAsyncTables() {
  //prepares the async data structures
  ...
}
```

5.2. JavaPlatform

The experimental `JavaPlatform` plug-in class has been used for prototyping the proposed TDL extensions. The driver structure and synchronization issues have been studied and tried out for the Java platform first, because this is much easier

to test and debug than on an embedded system, for example. The results have then been applied to other target platforms described in the subsequent sections.

We do not repeat the code fragments that have been shown before in the sections on synchronization but focus on the implementation of the various triggers for asynchronous events and the threading model being applied. The class `JavaPlatform` extends the class `AbstractNodePlatform` and thereby inherits the functionality for preparing the auxiliary async data structures.

5.2.1. Node specific glue code

As a container for the implementation of asynchronous execution, we introduced a new Java class per node, called `NodeAsyncs$$`. The name has been chosen such that it cannot produce any naming conflict with a glue code class needed for a module. The class `NodeAsyncs$$` is always generated automatically when the class `JavaPlatform` is requested to emit target specific code even if there are no asynchronous activities at all. For keeping the structure simple and self consistent we chose to embed all code that is required on the node level in this class, i.e. it contains also code that could in principle be factored out (e.g. methods `enqueue`, `dequeue`) into an application independent class.

Asynchronous execution starts when the class `NodeAsyncs$$` is loaded, i.e. by means of static initializers. The Java-based E-machine thus only has to load this class and can then forget about it.

The background thread for execution of asynchronous activities runs as a Java daemon thread with minimum priority as shown by the following initialization sequence.

```
static {
  asyncThread.setPriority(Thread.MIN_PRIORITY);
  asyncThread.setDaemon(true);
  asyncThread.start();
}
```

Asynchronous timers are implemented in the same way as Java threads, where a separate thread is used for every different timer period. All timer threads are implemented by a single class that is instantiated per timer period as shown below.

```
void startTimerThread(final int period) {
  Thread t = new Thread() {
    public void run() {
      for (;;) {
        try {
          switch (period) {
            case 5000: //async 5ms timer
              //enqueue all async events with 5ms period
              ...
              Thread.sleep(5);
              break;

            //more cases for other timer periods
            ...
```

```
            }
          } catch (InterruptedException x) {
            //should not occur
          }
        }
      }
    };
    t.setPriority(Thread.MIN_PRIORITY);
    t.setDaemon(true);
    t.start();
  }

  static {
    startTimerThread(5000);
    //start more timer threads, one per period
    ...
  }
```

In order to handle external hardware interrupts, the class `NodeAsyncs$$` provides a method that acts as an interrupt handler as shown below. It is expected that this method is called by an external thread that emulates an external hardware interrupt for example by pressing a button in a dialog window or by stochastically calling this method.

```
  //asynchronously handle external interrupt n
  public static void handleInterrupt(int number) {
    switch (number) {
      case 0:
        //enqueue all async events triggered by interrupt 0
        ...
        break;
      //more cases for other interrupt numbers
      ...
    }
  }
```

5.2.2. Module specific glue code

The module specific glue code class `M$.java` for a TDL module `M` generated by the class `JavaPlatform` has been adapted in order to cover the synchronization needs exactly as described in the previous sections. In addition, terminate drivers have been extended such that port updates may trigger asynchronous events by calling `enqueue`. This holds for both synchronous and asynchronous terminate drivers and is a trivial extension.

5.3. CPlatform

The class `CPlatform` provides the foundation for glue code generation for platforms using the C programming language. It extends the class `AbstractNodePlatform` and thereby inherits the functionality for preparing the auxiliary async data structures.

We only had to adapt the previously described implementation of the synchronization and execution mechanisms for asynchronous TDL activities for the C programming language. This required the adaptation of the glue code generation on node and module level and also the extension of our C runtime system containing the E-machine and a simple dispatcher for non-preemptive

scheduling. For the correct update of output ports of asynchronous tasks (see 4.3 above) we added the required functionality to the E-machine step function which is called on every periodic invocation of the E-machine. In contrast to `JavaPlatform`, the priority queue is not implemented in the generated node specific code but is part of the static runtime system. We added separate files named `tdl_async.c` and `tdl_asnyc.h` containing the following data type and functions:

```
typedef struct {
  char pending; //flag indicating pending async sequence
  int priority;
} tdl_async_AsyncDecl;

void tdl_async_init(tdl_async_AsyncDecl* asyncs,
                    int nofAsyncs);

void tdl_async_enqueue(int index);

int tdl_async_dequeue(void);
```

5.3.1. Node specific glue code

In contrast to `JavaPlatform`, `CPlatform` already used a node specific file named `tdl_main_.c`, which for example contained code for E-machine initialization and the reception of communication frames for distributed TDL systems. We added the following functionality for handling asynchronous TDL activities:

- Generation of the `tdl_asnyc_AsyncDecl` struct as defined above, which contains a flag indicating whether the sequence is currently pending and a priority for every asynchronous sequence executed on a node.
- Calling `tdl_async_init` to initialize the priority queue with the asynchronous sequences struct.
- Generation of the execution function `void executeAsyncSequence(int n)`, which executes the appropriate drivers for a given asynchronous activity sequence `n`.
- Generic interrupt handlers of type `void handleInterruptX(void)`, where `X` is the logical TDL interrupt number. The body of such a function contains calls to `tdl_async_enqueue` for all asynchronous activities triggered by interrupt `X`.

The actual execution of asynchronous activities as well as the registration of hardware interrupts and the implementation of timer triggers, is highly platform dependent and therefore not done in `CPlatform`.

5.3.2. Module specific glue code

The module specific glue code class `M_.c` for a TDL module `M` generated by the class `CPlatform` has been adapted in order to cover the synchronization needs exactly as described in the previous sections. In addition, terminate drivers have been extended such that port updates may trigger asynchronous events by calling `tdl_async_enqueue`. This holds for both synchronous and asynchronous terminate drivers and is a trivial extension. Note that in contrast to the interrupt and timer triggers, the port update trigger is therefore independent of any specific C platform.

5.4. EmbeddedCPlatform

The class EmbeddedCPlatform extends CPlatform and adds common functionality for embedded systems which typically lack a file system. Consequently, for example the E-code is encoded in C and not provided as a file. Regarding the TDL extension for asynchronous activities this class did not require any modifications as the driver adaptations are done in `CPlatform` and the implementation of timers and interrupt triggers is too platform specific to handle it at this point of the class hierarchy.

5.5. AESPlatform for NODE<RENESAS>

`AESPlatform` generates code for a FlexRay prototyping board manufactured by DECOMSYS (now Elektrobit) called NODE<RENESAS>. It extends the class `EmbeddedCPlatform` and therefore also inherits all the functionality of `CPlatform`. AES (Application Execution System) is a simple operating system provided with the board. It uses a static dispatch table to specify time-triggered tasks.

The AES operating system provides no support for task preemptions or task priorities. However, it does have a so-called *idle task* which is specified by implementing the hook function `skAES_ApplIdleTask`. The idle task runs when no time-triggered task is running. It is therefore well suited for executing asynchronous TDL activity sequences as its priority is lower than the E-machine, which we implemented as a time-triggered task. The idle task simply has to perform an infinite loop which calls the dequeue operation of the priority queue and executes the appropriate async sequence if the queue is not empty:

```
void skAES_ApplIdleTask (void) {
  for (;;) {
    int index = tdl_async_dequeue();
    if (index >= 0) {
      executeAsyncSequence(index);
    }
  }
}
```

Unfortunately there is no documented support in AES for interrupt handling. Consequently this trigger mechanism is not available for this platform. The TDL tool chain plug-in checks this limitation and yields an error message when a TDL module contains an interrupt trigger and is deployed on the AES platform.

Timers are currently also not documented on AES, as it provides no functions that would allow any kind of time measurement or the installation of timer interrupts. However it is possible to configure and access the CPU timers directly and we are confident that this will allow us to implement timer triggers by using such a timer for time measurement in the idle task and trigger asynchronous activities according to the period specified by TDL timer triggers.

5.6. MABXPlatform for dSPACE MicroAutoBox

`MABXPlatform` generates code for an automotive prototyping solution provided by dSPACE and called MicroAutoBox. It extends the class `EmbeddedCPlatform` in the same way as `AESPlatform` does. The MicroAutoBox operating system supports task preemptions, task priorities, and interrupt handling and therefore provides sufficient features to fully support all trigger types of asynchronous TDL activities.

The MicroAutoBox operating system requires the implementation of a function `void main(void)` which is executed upon startup and typically contains

initialization calls and task definitions. This function can also be used in order to execute asynchronous TDL activities as it runs at the lowest priority and is preempted by all other tasks. Therefore we use the same endless loop as described above in `AESPlatform` and execute it at the end of the `main` function.

The MicroAutoBox operating system provides interrupt handling via the following functions:

```
DS1401_Int_Handler_Type ds1401_set_interrupt_vector(
      UInt32 IntID,
      DS1401_Int_Handler_Type Handler,
      int SaveRegs)
```

This function binds an interrupt `IntID` to an interrupt handler `Handler` of type `void (*DS1401_Int_Handler_Type)(void)`. Our MicroAutoBox 1401/1505/1507 has 4 external interrupt lines available which correspond to the `IntID`s `DS1401_IR4`, `DS1401_IR5`, `DS1401_IR6`, and `DS1401_IR8`. We map them to the TDL logical interrupts 0..3. If a higher interrupt number is used we notify the user that such a TDL module cannot be deployed on the platform and abort the code generation process. As task function the `handleInterruptX` function as described above can be used directly by passing it as a function pointer. The parameter `SaveRegs` is a Boolean flag specifying whether to save and restore all registers before and after the execution of the interrupt handler. We use the default setting `SAVE_REGS_ON`. In order to enable the interrupt the following function must be called for every `IntID`:

```
void ds1401_enable_hardware_int(UInt32 IntID)
```

Timer triggers are implemented on the MicroAutoBox platform by scheduling a periodic task with the specified period of the asynchronous activity. Here is an example for scheduling a periodic task `timer10000` with a period of 10 ms:

```
rtk_p_task_control_block timer10000TCB;
timer10000TCB = rtk_create_task(timer10000, 1, ovc_count, NULL,
                                INT_MAX, 0);
rtk_bind_interrupt(S_INTERVAL_A, 3, timer10000TCB, 0.01f, C_LOCAL, 0,
                   NULL);
rtk_set_task_type(S_INTERVAL_A, 3, RTK_NO_SINT, rtk_tt_periodic,
                  NULL, 0.0f, 1);
```

According to the threading model proposed above, hardware interrupts and timers have the highest priority in the system. Therefore we specify the highest priority `1` as second argument of `rtk_create_task`. The period is defined in seconds as floating point value `0.01f` in argument four of `rtk_bind_interrupt`.

We also must generate a corresponding task specification for every timer trigger which is passed as a function pointer in argument one of `rtk_create_task`. In our example it is called `timer10000` and triggers the asynchronous sequence number `1`:

```
static void timer10000(rtk_p_task_control_block pTCB) {
  tdl_async_enqueue(1);
}
```

Please refer to the MicroAutoBox documentation for a detailed description of all related functions and their parameters.

5.7. Simulink

The Simulink platform is used for simulating the behavior of TDL modules within the simulation environment MATLAB/Simulink. With the introduction of asynchronous activities, TDL looses its property that simulation results exactly match with the behavior on an arbitrary potentially distributed hardware platform. In general, the actual invocation instants of asynchronous activities do not match on different platforms. Additionally, platform specifics are not modeled in the simulation, thus the Simulink platform is neither aware of any target specific scheduling mechanism, network topology, nor expected execution times, etc. However, the Simulink platform is still useful for studying the interaction of synchronous and asynchronous TDL activities together with the plant model.

The implementation of the Simulink platform does not build on the AbstractNodePlatform but is independent. This is due to the different requirements for a simulation platform. The Simulink E-machine is implemented as an S-Function [SLsfunc] block written in C. The functionality code must be provided as function-call subsystems. The individual glue code elements, i.e. drivers, etc., are also represented as function-call subsystems and signals between them. The glue code is generated automatically.

The Simulink simulation engine executes the blocks sequentially in a fixed order. There is no concurrency and consequently no need for synchronization between the synchronous and the asynchronous TDL part. To ensure the priority of synchronous over asynchronous activities, asynchronous activities are also handled by the E-machine S-Function block.

Events coming from timers or interrupts are provided to the E-machine with an additional input port. As it is the case for the synchronous counterparts, asynchronous task invocations and actuator updates are initiated by means of function-call triggers. Fig. 2 exemplifies the data-flow to and the control-flow from the E-machine block.

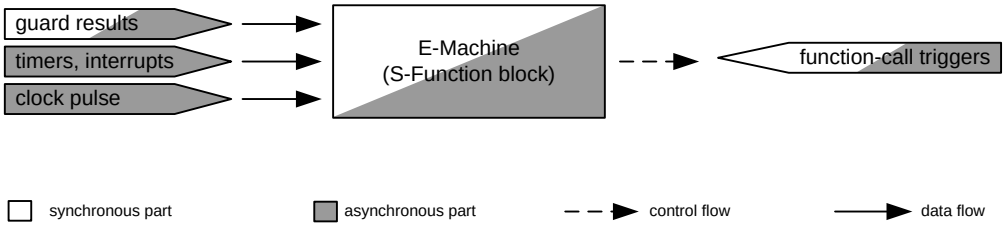
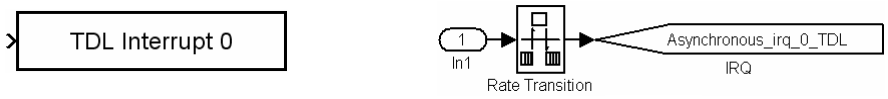


Fig. 2 E-machine data- and control-flow

Timers are implemented as part of the glue code using the Simulink built-in Pulse Generator block. The block's sample time is set to twice the timer period and the duty cycle (pulse width) to 50 % in order to not affect the overall model sample time. The pulse generator is connected with the E-machine, which interprets both rising and falling edges as timer interrupts. To handle different rates of the pulse generator and the E-machine block, a rate transition block is inserted between them.

For interrupt events, we introduce the TDL Interrupt block in the TDL library (see Fig. 3). The implementation of this block simply forwards the incoming signal to the E-machine using a Goto-From block pair. Again a rate transition block resolves different sample rates. Instances of this block may be placed in the model and connected to source blocks. The E-machine interprets both rising and falling signals as interrupts.



(a) (b)

Fig. 3 The TDL Interrupt block (a) and its implementation (b)

Every time the E-machine is invoked, it executes the following three steps:

1. execute synchronous activities (interpret Ecodes)
2. register timer and interrupt events (indicated by changes of input signals)
3. execute pending asynchronous events

At any time, asynchronous activities execute after synchronous activities. Thus, the different priority levels are respected. However, the execution of synchronous tasks does not delay asynchronous activities. We decided to treat them independently, because we make no assumptions on the scheduling mechanism and therefore nothing can be said about the actual execution times within the logical execution time of a task. By default, an asynchronous task executes in zero time. However, if the task has defined a platform independent worst-case execution time *wcet*, output ports of this task are only updated after this wcet has elapsed and the execution of other asynchronous activities is deferred for this time span.

In addition to the logical time instants required by the Ecodes, the E-machine has to be invoked whenever asynchronous activities have to be started respectively terminated.

The invocation times of the E-machine depend on the following properties:

1. logical timing of synchronous activities
2. asynchronous timer periods
3. worst-case execution times of asynchronous tasks
4. interrupt arrival times

While the properties 1-3 are expressed in TDL, interrupt arrival times (property 4) depend on the blocks connected to TDL Interrupt blocks.

We define the E-machine S-Function block to inherit its sample time:

```
ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
```

This causes the E-machine to execute whenever an interrupt occurs or a timer elapses. Additionally, we use a *Digital Clock* block that is connected to the E-machine (see Fig. 2). The sample time of this block is set to the GCD (greatest common divisor) of the period of all synchronous activities and of the worst-case execution time of all asynchronously executed tasks. Hence, we cover all relevant time instants.

6. Future Work

The obvious next step for fully supporting asynchronous activities in TDL is the integration of asynchronous dataflow between remote nodes. On clusters based on the FlexRay protocol we envision the usage of dynamic slots for broadcasting data produced by asynchronous task invocations to any interested remote node.

This approach would be perfectly in line with our approach of transparent distribution of synchronous activities.

In addition to asynchronous dataflow between remote nodes it might also be necessary to support calling of imported asynchronous activities both on local and remote nodes. This would call for something like an RPC (remote procedure call) interface for asynchronous tasks.

For the AES platform we plan to make timer events available by directly accessing the hardware. This approach might also work for interrupts.

As future work for the Simulink platform we envision to optionally consider platform specific aspects in the simulation. When the designated hardware target has been chosen, information about the module to node mapping, the scheduling mechanism and the worst-case execution times is available and can be considered in the E-machine. As actual execution times vary, they might be decided stochastically. With this approach, we should get quite meaningful results that come close to the behavior on the target platform, without changing the simulation model.

7. Conclusions

Supporting asynchronous activities within a synchronous world turned out to be possible without the need for advanced synchronization primitives such as monitors or semaphores. The compiler extensions and ecode file format changes were mostly straight forward and the syntax is simple and fits naturally within the rest of a TDL module.

One crucial point is the implementation of a thread safe priority queue for asynchronous events. With our approach of using a boolean flag per event and a linear search for the next event to be processed we arrived at a correct implementation without any further synchronization needs. Registering an event is done in a small constant time that could also preempt the E-machine without any noticeable delay. Due to its simple loop body the linear search for the next event to be processed is expected to be fast enough for small and medium numbers of different asynchronous events.

Another crucial point is the synchronization of the dataflow between asynchronous activities and the synchronous TDL E-machine. We presented a solution that relies completely on the atomicity of a single memory store operation. Our solution has a minimal runtime overhead, is easy to port to any naked hardware, and avoids the danger of dead locks and priority inversions.

The platform implementation for the experimental Java platform turned out to be a viable foundation for later porting the solution to real-world target platforms and a significant part of the extensions has been reused by a specific platform by inheriting from an abstract base platform class. In general, the existing platform class hierarchy did not need any changes and turned out to be a big help even for the task of introducing asynchronous events in a synchronous world.

8. References

[TDL1.5spec] J. Templ. TDL - Timing Definition Language Specification 1.5. Technical Report, University of Salzburg, 2008.

[SLsfunc] The MathWorks. Simulink 7, Writing S-Functions. 2008.