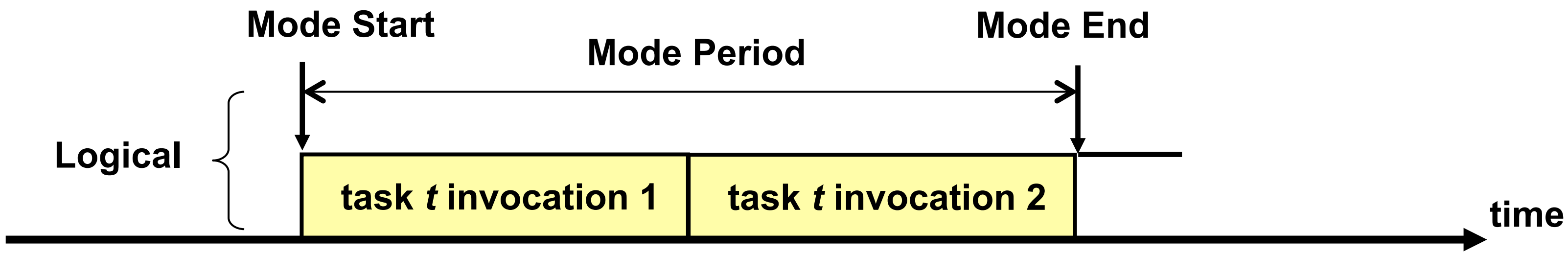
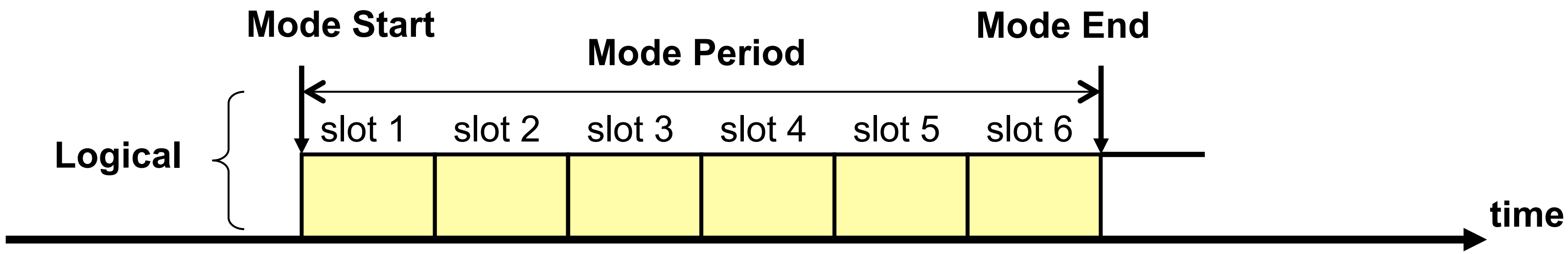
Periodic Execution – Giotto Modes



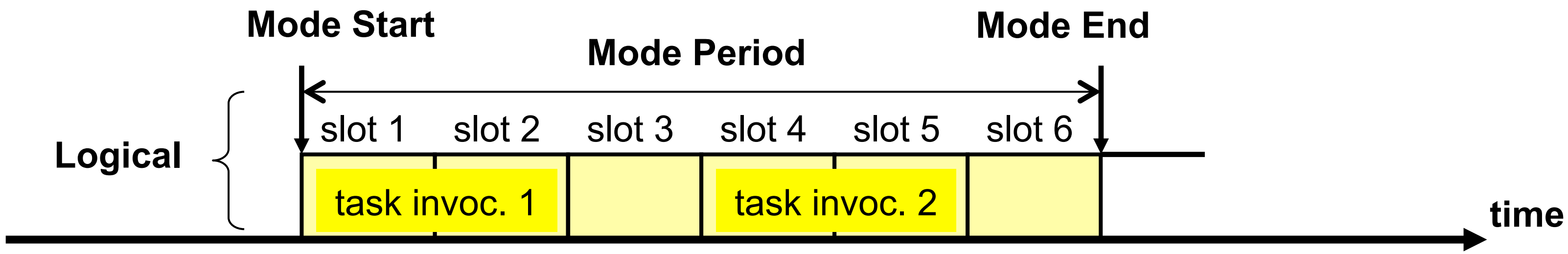
- Every mode has a fixed period.
- A task *t* has a frequency *f* within a mode.
- The mode period is filled with *f* task invocations.
- The LET of a task invocation is *modePeriod / f.*

TDL Slot Selection



- f = 6

TDL Slot Selection



- f = 6

- task invocation 1 covers slots 1 – 2
- task invocation 2 covers slots 4 – 5

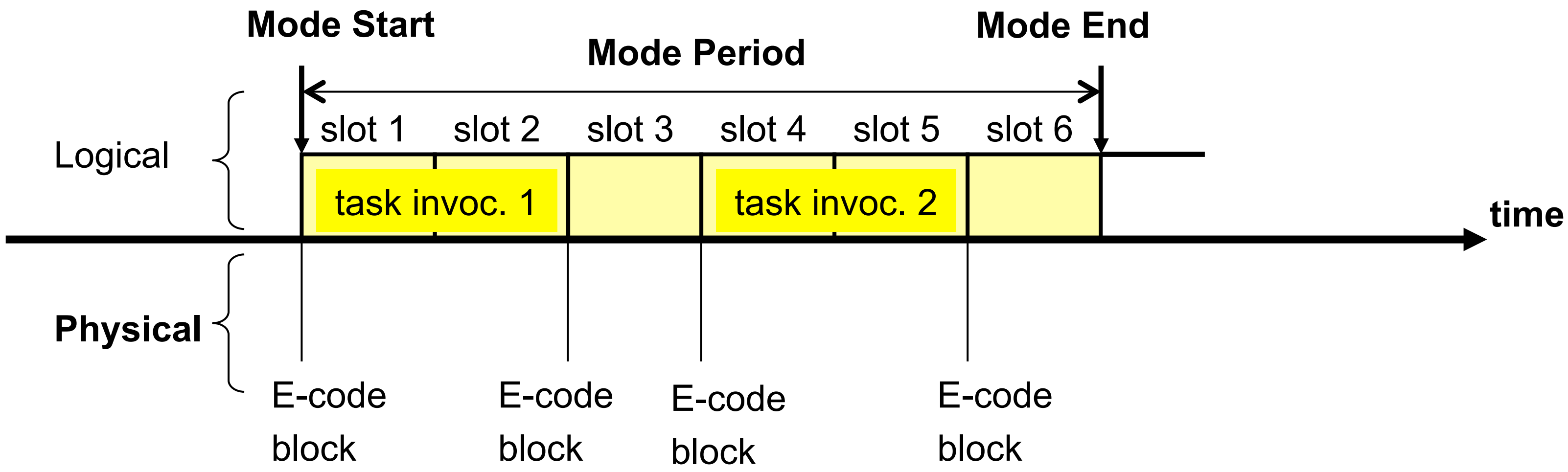
TDL Slot Selection Allows One to Specify

- an arbitrary repetition pattern
- the LET more explicitly
- gaps
- task invocation sequences
- optional task invocations
- Giotto periodic task invocation as a special case (default)

TDL Execution

- based on a virtual machine, called *E-machine*
- executes virtual instruction set, called *E-code*
- E-code is generated by TDL compiler from TDL source
- covers one mode period
- contains one E-code block per logical time instant

E-code Blocks



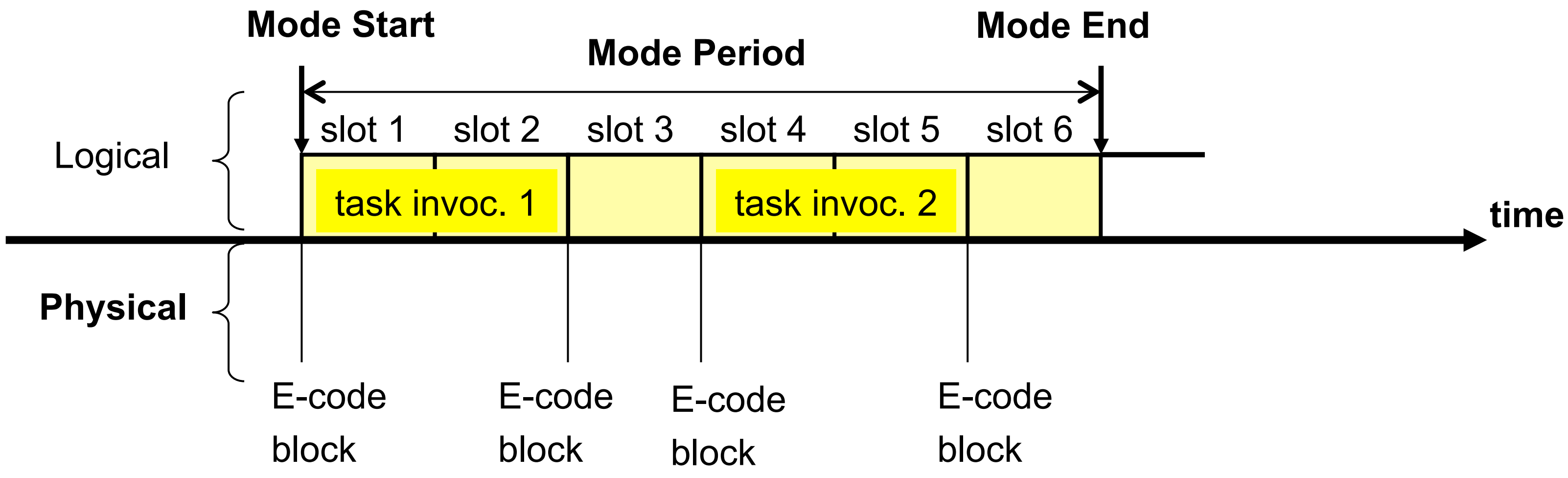
- E-Code block follows fixed pattern:
  1. task terminations
  2. actuator updates
  3. mode switches
  4. task releases

E-code Compression

- adjacent E-code blocks may be identical
- compression feature would be welcome
- new instruction:

  `REPEAT <targetPC>, <N>`
- jumps *N* times to *targetPC*, then to *PC* + 1.
- uses a counter per module
- counter is reset upon mode switch

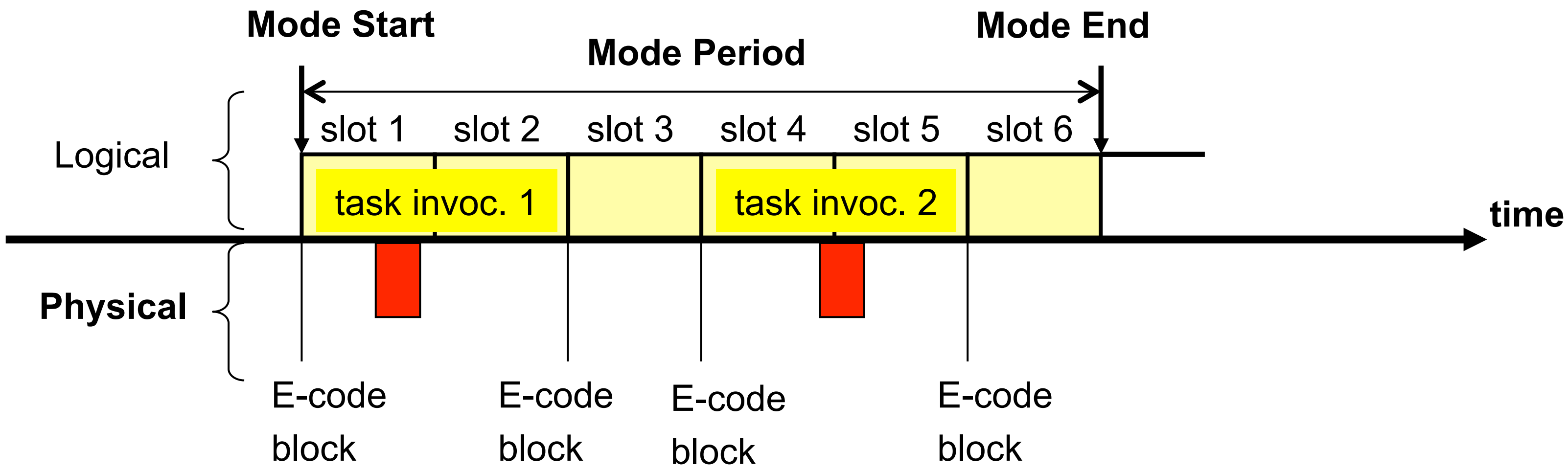
Adding Asynchronous Activities



Priority levels

- black: highest priority (E-code)

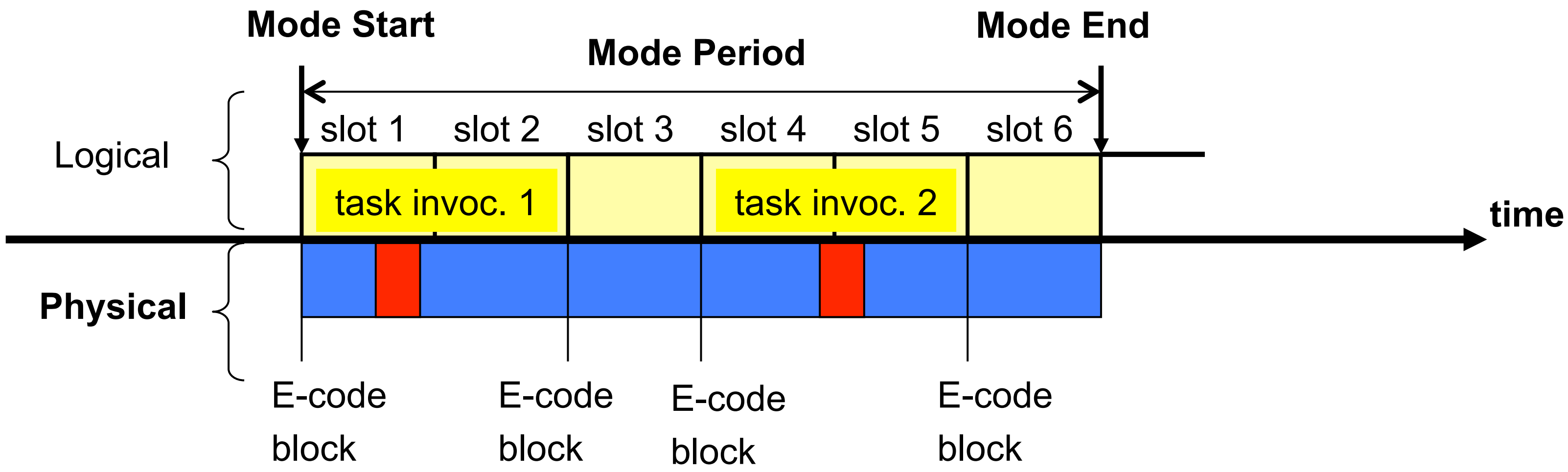
Adding Asynchronous Activities



Priority levels

- black: highest priority (E-code)
- red: lower priority (synchronous tasks)

Adding Asynchronous Activities



Priority levels

- black: highest priority (E-code)
- red: lower priority (synchronous tasks)
- blue: lowest priority (asynchronous activities)

Asynchronous Activities Rationale

- event-driven background tasks
- may be long running
- not time critical
- may be implemented at platform level, but:
  - platform specific
  - unsynchronized data-flow to/from E-machine

- support added in TDL
- **Goal**: avoid complex synchronization constructs and the danger of deadlocks and priority inversions

Kinds of Asynchronous Activities

- task invocation
  - similar to synchronous task invocations except for timing
  - input ports are read just before physical execution
  - output ports are visible just after physical execution
  - data flow is synchronized with E-machine
- actuator updates
  - similar to synchronous actuator updates except for timing
  - data flow is synchronized with E-machine

Trigger Events

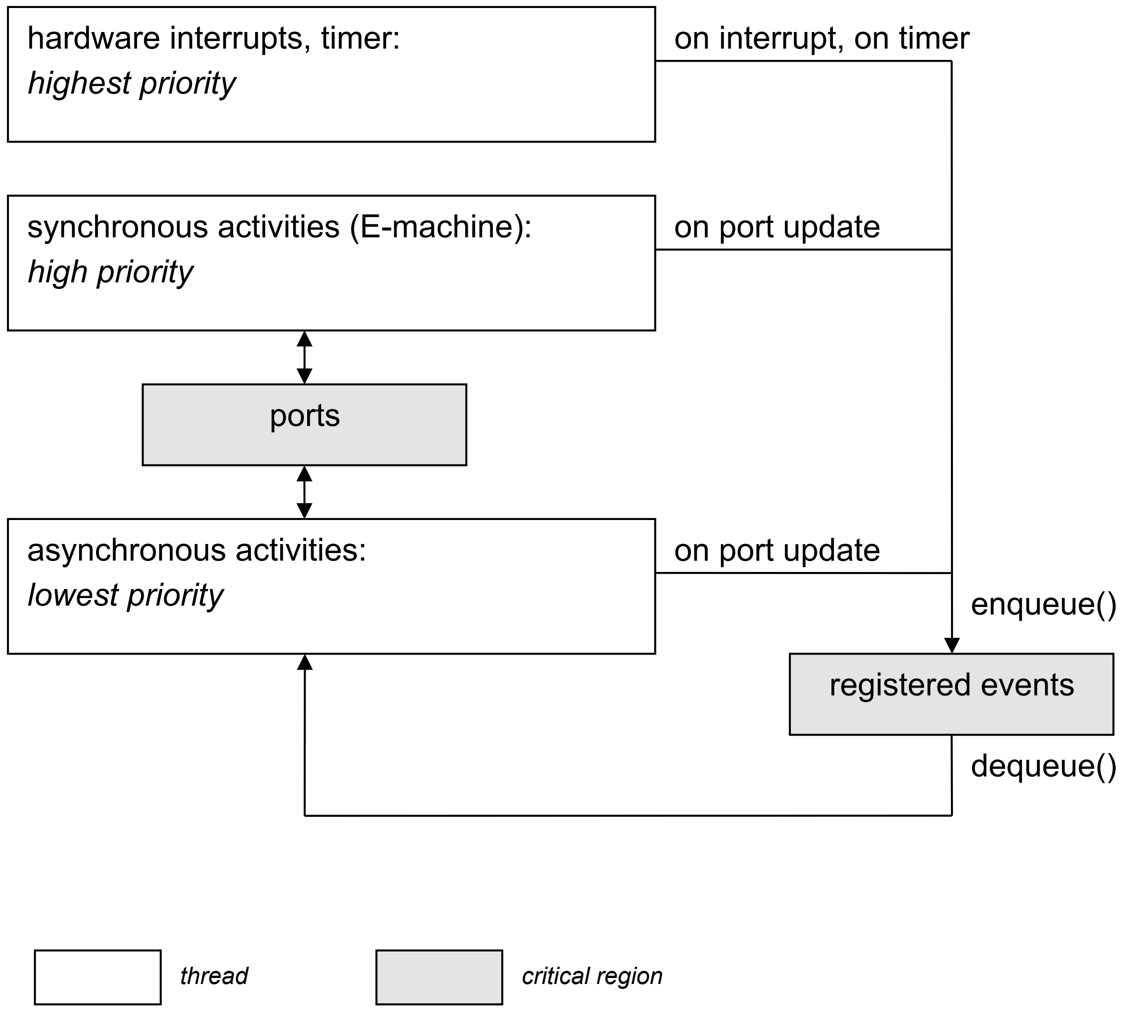
- hardware and software interrupts
- periodic asynchronous timers
- port updates

Use a registry for later execution of the async activities.

Parameter passing occurs at execution time.

Registry functions as a priority queue.

Threads and Critical Regions



Synchronization Requirements

- Async activities don‘t preempt anything.
- E-machine may preempt async activities.
- Hardware interrupts (incl. timers) may preempt everything incl. other hardware interrupts.
- We need a very robust thread safe registry.
- We need a very efficient `enqueue` operation
  - for serving hardware interrupts quickly
  - for efficient synchronous port update triggers
- `dequeue` is done asynchronously and may be slower.

Event Registry

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

- `enqueue` sets the *pending* flag; constant time
- `dequeue` searches for the highest priority event and clears the *pending* flag
- triggering a pending event is a no-op.

Synchronizing Input Port Reading

- reading of input ports for async activities must be **atomic**
- i.e. must not be interrupted by the E-machine
- only one async event is processed at a time
- we use a global flag that signals E-machine execution
- we clear this flag before input port reading
- we set this flag in every E-machine step
- we repeat the reading until there is no interruption

Synchronizing Output Port Writing

- writing output ports after async task invocation must be **atomic**
- but may be interrupted by the E-machine
- observation: output port writing is **idempotent**
- we can re-execute it atomically in the E-machine
- only one async event is processed at a time
- therefore we register the function (*termination driver*) that does the output port writing in a global variable
- the E-machine tests for the existence of a registered termination driver and re-executes it

Example Asynchronous TDL Activity

```
// Radio control data is received by an interrupt service routine.
// Once all channels have been received the data is passed into the
// TDL world by raising the software interrupt RCInterrupt.

module CaptureRC {

    import Types;
    import Kalman;

    public type Command = …;

    public output Command cmd;
    public output Kalman.State targetState;

    public task getRcData {
       uses doGetRcData(cmd, targetState);
    }

    asynchronous {
       [interrupt = RCInterrupt, priority = 1] getRcData();
    }
}
```

Other Extensions

- Module level output ports
- Structured user defined types
- Adaptations in TDL tool chain and Simulink integration
- VisualAnalyzer
- Incremental communication scheduling for FIBEX
- FlexRay Startup-Protocol
- FlexRay Configuration Editor and Checker
- OSEK platform support
- Combined Comm/Task-Scheduling + Genetic Alg.
- 2-step E-machine for Simulink