# ‘Blocking’ tutorial

Version 2

Dezyne version 2.4.0

June 16th, 2017

# Introduction

## Learning goals

After following this tutorial, you will be able to:

* Understand what ‘blocking’ allows you to do in Dezyne
* Simplify an ‘external’ implementation with usage of ‘blocking’
* Identify when ‘blocking’ may be used and when it may **not** be used
* Reason about design decisions when using the ‘blocking’ keyword

## Intended audience and prerequisites

During this tutorial, we will reconsider an implementation that handles possibly delayed communication over an ‘external’ port from the previous tutorial. It is extremely helpful, but not required to have followed that tutorial before starting this one.

The behaviour described with the ‘blocking’ keyword requires you to be able to reason about your application in terms of active threads and their calling context. In the tutorial, various examples will be discussed but it helps if you are familiar with the concepts.

Usage of ‘blocking’ requires the System it is contained in to be generated with a thread-safe-shell. For more information on the thread-safe-shell, please refer to <https://www.verum.com/supportitem/code-integration-extra-materials/> and <https://www.verum.com/supportitem/thread-safe-shell/>.

## Platform choice

The platform choice from previous tutorials remains unchanged. Raspbian with g++ 4.9.2 on the Raspberry Pi supports all language requirements for using a thread-safe-shell and ‘blocking’ in C++11.

Table of Contents

[‘Blocking’ tutorial 1](#_Toc491418019)

[Introduction 1](#_Toc491418020)

[Learning goals 1](#_Toc491418021)

[Intended audience and prerequisites 1](#_Toc491418022)

[Platform choice 1](#_Toc491418023)

[Chapter 1: What can ‘blocking’ be used for? 3](#_Toc491418024)

[Chapter 2: Simplifying an ‘external’ implementation using ‘blocking’ 5](#_Toc491418025)

[Changing RobustTimer to provide a synchronous Timer interface 5](#_Toc491418026)

[Chapter 3: ‘blocking’ behaviour in runtime context 8](#_Toc491418027)

[Runtime behaviour 8](#_Toc491418028)

[Implications of ‘blocking’ for Dezyne applications 9](#_Toc491418029)

[Chapter 4: Retrospect on learning goals 11](#_Toc491418030)

[Understand what ‘blocking’ allows you to do in Dezyne 11](#_Toc491418031)

[Simplify an ‘external’ implementation with usage of ‘blocking’ 11](#_Toc491418032)

[Identify when ‘blocking’ may be used and when it may not be used 11](#_Toc491418033)

[Reason about design decisions when using the ‘blocking’ keyword 11](#_Toc491418034)

# Chapter 1: What can ‘blocking’ be used for?

For the examples in this chapter, a snapshot containing Dezyne models and C++ source code is available on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Blocking/Ch1_Starting_Point>.

In the tutorial on the usage of the ‘external’ keyword, we successfully implemented a RobustTimer component that maps an ‘external’ requires ITimer port to a regular provides ITimer port. In order to fully capture the possible behaviours that occur when considering the port to be ‘external’, an extra out-event was added to mark the end of the cancellation transaction:

|  |  |
| --- | --- |
| Before | After |
| interface ITimer {    extern long\_integer $long$;    enum State { Idle, Running, Stopping };    in void start(long\_integer milliseconds);    in void cancel();    out void timeout();    behaviour {      State state = State.Idle;      [state.Idle] {        on start: state = State.Running;        on cancel: { }      }      [state.Running] {        on start: illegal;        on cancel: state = State.Idle;        on inevitable: {          state = State.Idle;          timeout;        }      }    }  } | interface ITimer {    extern long\_integer $long$;    enum State { Idle, Running, Stopping };    in void start(long\_integer milliseconds);    in void cancel();    out void timeout();    out void cancelled();    behaviour {      State state = State.Idle;      [state.Idle] {        on start: state = State.Running;        on cancel: { cancelled; }      }      [state.Running] {        on start: illegal;        on cancel: state = State.Stopping;        on inevitable: {          state = State.Idle;          timeout;        }      }      [state.Stopping] {        on start: illegal;        on cancel: illegal;        on inevitable: {          state = State.Idle;          cancelled;        }      }    }  } |

This addition allowed the RobustTimer component to determine whether the ‘external’ requires port was free of any delayed timeout events. This was required to ensure that a new Timer could safely be started after cancelling an older one.

However, this change also had implications for other components that require ITimer ports. Originally, one of the reasons to create the RobustTimer component was to hide the complexity paired with ‘external’ behaviours from the Controller component. Ultimately, due to the changes made to the ITimer interface to accommodate for extra behaviours, we still had to make changes to the Controller component. Handling the asynchronous cancelled event had to be added to the behaviour of the Controller.

This phenomenon where changes lower down in the chain of command influence top-level components is what we will refer to as **contagious asynchronicity**. Because of asynchronous behaviour that was added to one of the lower components in the System, components higher up in the System are forced to accommodate for this asynchronous behaviour as well.

This is not always a bad thing- sometimes it is actually desired to propagate such events upwards so that perceived state of a subcomponent is updated. However, there are also cases where this additional information on the state of a subcomponent is irrelevant and as such, you do not want to deal with the added asynchronous behaviour of components. **This is where ‘blocking’ comes into play**; using ‘blocking’, it becomes possible to convert certain asynchronous behaviour patterns to synchronous versions.

In the case of the RobustTimer, ‘blocking’ will allow you to implement the cancellation process in such a way that it appears synchronous over the provides port even though it is asynchronous with respect to its requires port. In the next chapter, you will learn how to perform such an implementation.

# Chapter 2: Simplifying an ‘external’ implementation using ‘blocking’

In this chapter, we will take a look at describing an asynchronous process as if it were synchronous. We will do this by use of the ‘blocking’ keyword. The asynchronous process we will be considering is the cancelling of RobustTimer. In case you hadn’t had a look at the snapshot containing Dezyne models and C++ source code already, it is available on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Blocking/Ch1_Starting_Point>.

## Changing RobustTimer to provide a synchronous Timer interface

The goal is to make RobustTimer easier to use with respect to its provided port. We will still want the newer version of the ITimer interface to fully describe the behaviour of the ‘external’ requires port, but we want to translate this to a simplified version of ITimer. The original ITimer interface will suffice. As an exercise, perform the following tasks:

* rename the new ITimer interface to AsyncITimer
* reinstate the “old” ITimer interface as ITimer
* change the RobustTimer to requires ‘external’ AsyncITimer and provides ITimer

Solution: see <https://github.com/VerumSoftwareTools/DezyneTutorial/blob/master/Blocking/Ch2_Solution/Timer.dzn>

These changes will break several parts of the application, but let’s focus on RobustTimer only for now. The cancellation process was quite nasty; a cancel event on its provided ITimer port started the process, at which point the provided ITimer port was unusable until the cancelled event from ext\_iTimer came in. By separating the provides and requires interfaces, you are no longer forced to describe this period of unusability in the provides interface. From the provides port point of view, the cancellation process is synchronous again; all that remains is to make sure the RobustTimer component is verifiably correct with the new provides/requires ports.

The beginning of the cancellation process is the cancel event on the provided iTimer; the mark that denotes the end of the process is the cancelled event from the required ext\_iTimer. ‘blocking’ allows you to postpone the return of an incoming event from a provides port until an event comes in from a **different** port; this sounds exactly like the scenario of the cancellation process.

In RobustTimer, there are two locations where the cancellation process is ended, both currently marked by the iTimer.cancelled() event left over from the previous iteration of RobustTimer. If we were to denote the iTimer.cancel() event as ‘blocking’, the ending of the transaction will be in the same locations as the current iTimer.cancelled() events:

component RobustTimer {

  provides ITimer iTimer;

  requires external AsyncITimer ext\_iTimer;

  behaviour {

    enum State { Idle, Running, Stopping };

    State state = State.Idle;

    on iTimer.start(ms): {

      [state.Idle] {

        ext\_iTimer.start(ms);

        state = State.Running;

      }

    }

    blocking on iTimer.cancel(): {

      [state.Running] {

        ext\_iTimer.cancel();

        state = State.Stopping;

      }

      [state.Idle] { iTimer.reply(); }

    }

    on ext\_iTimer.timeout(): {

      [state.Running] {

        iTimer.timeout();

        state = State.Idle;

      }

      [state.Stopping] { /\* discard \*/ }

    }

    on ext\_iTimer.cancelled(): {

      [state.Stopping] {

        state = State.Idle;

        iTimer.reply();

      }

    }

  }

}

Note that the transaction endpoints are represented as port.reply() events. An important thing to keep in mind is that with ‘blocking’, you block the calling thread of the event until the blockage is lifted- which is done with the port.reply() event. The modified RobustTimer posted above is verifiably correct and will display the following behaviour:

On iTimer.cancel, the iTimer port is blocked. If the RobustTimer is Running, the cancel event will be sent to ext\_iTimer and RobustTimer will transition to the Stopping state. In the Stopping state, the asynchronous ext\_iTimer.cancelled event will release the blocked iTimer port. If RobustTimer was Idle upon receiving iTimer.cancel, the blocked iTimer port is immediately released.

This immediate release of the blocked iTimer port hints at a possible optimization; if a port is released instantly then it doesn’t really have to be blocked at all. By nesting the ‘blocking’ keyword deeper into the [guard]-on event scope you can avoid this unnecessary blockage:

component RobustTimer {

  provides ITimer iTimer;

  requires external AsyncITimer ext\_iTimer;

  behaviour {

    enum State { Idle, Running, Stopping };

    State state = State.Idle;

    on iTimer.start(ms): {

      [state.Idle] {

        ext\_iTimer.start(ms);

        state = State.Running;

      }

    }

    on iTimer.cancel(): {

      blocking [state.Running] {

        ext\_iTimer.cancel();

        state = State.Stopping;

      }

      [state.Idle] { }

    }

    on ext\_iTimer.timeout(): {

      [state.Running] {

        iTimer.timeout();

        state = State.Idle;

      }

      [state.Stopping] { /\* discard \*/ }

    }

    on ext\_iTimer.cancelled(): {

      [state.Stopping] {

        state = State.Idle;

        iTimer.reply();

      }

    }

  }

}

With the changes highlighted above, the ‘blocking’ keyword is nested deeper and affects fewer behaviour traces; the net behaviour is the same. Note that when on-event behaviour is **not** marked as blocking, a void port.reply() event is not allowed; hence, [state.Idle] now shows as no-op instead of port.reply(). The continuation is implicit when the on-event is not ‘blocking’.

From the perspective of the iTimer port, the cancellation process is now synchronous; a cancel event sent through the iTimer port will not return until the cancellation process has been completed. RobustTimer can still make use of the asynchronous specification of AsyncITimer to facilitate the cancellation handshake. Therefore, the asynchronous behaviour has successfully been made synchronous by using the ‘blocking’ keyword.

With the changes made to the provides interface of RobustTimer, you should also repair the Controller and AlarmSystem components. The Rearming state in the Controller can be removed as the cancellation process is synchronized. In AlarmSystem, some port types will have to be renamed to reflect the new interface names. As an exercise, fix the models so they are free of errors again.

Solution: the final versions of the models in the AlarmSystem application can be found on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Blocking/Ch2_Final>

Unlike in the ‘external’ tutorial, no native source code will have to be changed. All of the simplifications that were made are in Dezyne models and the native components and ports on the boundary of the System remain unchanged. The only requirement to be able to use ‘blocking’ is that your System is generated with a thread-safe-shell (and for now, only C++ is supported). The generation of a thread-safe-shell is included in the makefile, therefore the AlarmSystem application is fully functional again.

Now that we have implemented a simplification of the asynchronous ITimer interface, let’s consider some implications this will have for the runtime behaviour of the AlarmSystem application in the next chapter.

# Chapter 3: ‘blocking’ behaviour in runtime context

Although ‘blocking’ has allowed you to make the Controller model much simpler, ‘blocking’ may not always be as useful or runtime semantics might prevent you from using ‘blocking’ at all in an application. In this chapter, we will consider the runtime behaviour of the changes we made to RobustTimer and consider some limitations to the usage of ‘blocking’.

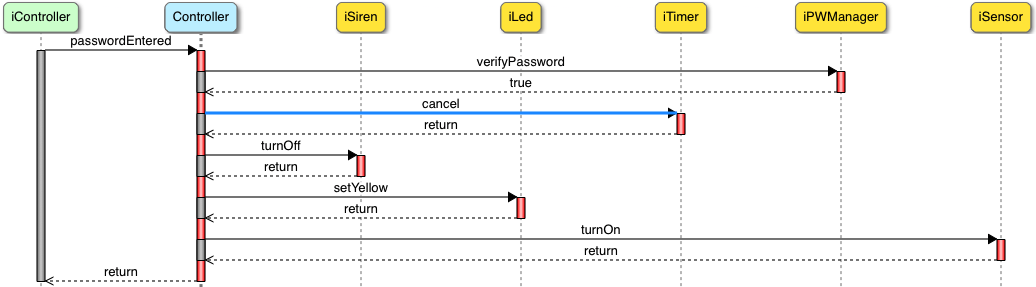
## Runtime behaviour

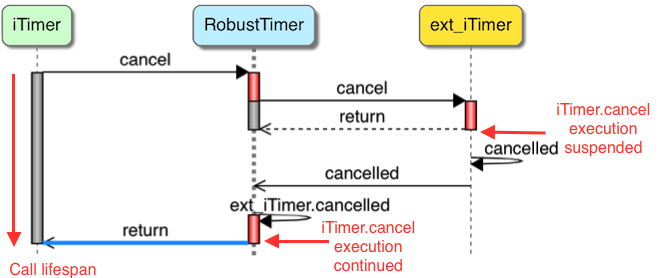
*For simplicity’s sake, we will consider two active threads in this analysis: the* ***main thread****, running the event loop, and the* ***Dezyne private thread****, consuming events from dzn::pump.*

The behaviour we described as ‘blocking’ is the cancellation process of the Timer. This process is started when in the Alarming state, a valid password is entered. Password entry is done through the passwordEntered event on the IController port of the AlarmSystem. This is where the call stack begins: invoking passwordEntered from the main thread.

Because AlarmSystem is generated with a thread-safe-shell, invoking the passwordEntered event is done with the dzn::shell functionality from the Dezyne runtime libraries. This means the main thread will be blocked until the Dezyne private thread has completed processing the event. **Important:** this is *not* because of ‘blocking’! This behaviour is fully because of the thread-safe-shell.

Once the event is scheduled in dzn::pump, at some point in time it will be picked up by the Dezyne private thread. Part of the processing of the passwordEntered event is the ‘blocking’ iTimer.cancel call:



The processing of the passwordEntered event is done on the Dezyne private thread, but remember that the main thread is blocked until the related *return* statement is sent (signaling that the event has been fully processed). Now, when the Dezyne private thread invokes the iTimer.cancel event, it will encounter ‘blocking’ as the implementation of iTimer.cancel in RobustTimer is ‘blocking’:

The Dezyne runtime libraries provide support for ‘blocking’ in the sense that an execution on the Dezyne private thread can be suspended and released again, which ‘blocking’ makes use of. Invoking a ‘blocking’ event suspends the execution of the current event from the dzn::pump and starts execution of the next available event. This subtle implementation in the Dezyne runtime is how the blocked execution can be released again; the release will occur from another event scheduled in dzn::pump. In the figure above, this other event is ext\_iTimer.cancelled. When the execution is released, the Dezyne private thread can continue processing the iTimer.cancel event which eventually returns and both threads are released again.

## Implications of ‘blocking’ for Dezyne applications

Obviously, a big concern regarding whether you can use ‘blocking’ in your Dezyne models is that your native code running on the main thread must not suffer from being blocked. The Sensor polling implementation could just as easily have been handled in the event loop instead of through dzn::pump; however, then the debouncing algorithm could suffer from not polling consistently. The current event loop only handles the entering of passwords; this is so trivial that blockage is not a concern. However, when more execution takes place on the main thread you will have to make sure that it is allowed to be postponed until the blocking call returns.

A second concern is that the execution semantics of your platform of choice must support some way to schedule events outside of normal main thread activity. If the main thread is blocked and the private Dezyne thread is waiting for events in dzn::pump, the only way to continue is to have some sort of activity outside of the two existing threads. This activity can then release the Dezyne private thread, which can release the main thread again. Examples of such activity are Interrupt Service Routines (ISRs) or raising a signal like in the native Timer implementation.

Another consideration you should make is whether the asynchronous process you’re simplifying with ‘blocking’ even benefits from it. RobustTimer could be simplified even more by having a ‘blocking’ start event that returns upon timeout. This would make the components requiring ITimer even simpler, perhaps, but it would no longer be possible to cancel the timer.

In a way, we even removed functionality from Controller by synchronizing the cancellation process; at the end of the ‘external’ tutorial, we implemented a “queue” to allow passwords entered during the cancellation to be stored and handled when the process ends. By hiding this process from Controller, such interactions are no longer necessary. Sometimes you can make clever use of the asynchronous period and if that is the case, you will have to consider whether the simplified models outweigh the clever usage.

All in all, there are some hard and some soft constraints to using the ‘blocking’ keyword: hard constraints are the execution model on your main thread and whether it can handle being blocked for an arbitrary amount of time; soft constraints mostly consist of design decisions and whether you can make use of the asynchronous process or not.

# Chapter 4: Retrospect on learning goals

In the introduction of this tutorial, the following learning goals were mentioned:

* Understand what ‘blocking’ allows you to do in Dezyne
* Simplify an ‘external’ implementation with usage of ‘blocking’
* Identify when ‘blocking’ may be used and when it may **not** be used
* Reason about design decisions when using the ‘blocking’ keyword

In this chapter, you will find a quick summary on each of the learning goals and where they were discussed in the tutorial.

## Understand what ‘blocking’ allows you to do in Dezyne

Using ‘blocking’, it becomes possible to describe certain asynchronous behaviour patterns as if they were synchronous. By hiding the asynchronicity of a process, components higher up in a system are simplified. This was discussed in Chapter 1: What can ‘blocking’ be used for?

## Simplify an ‘external’ implementation with usage of ‘blocking’

In Changing RobustTimer to provide a synchronous Timer interface, changes were made to RobustTimer to provide a blocking synchronous cancel event. This event starts an asynchronous process in RobustTimer where the end of the process signals the return of the ‘blocking’ event.

## Identify when ‘blocking’ may be used and when it may **not** be used

Following an analysis of a ‘blocking’ process in Runtime behaviour, some hard constraints were identified in Implications of ‘blocking’ for Dezyne applications: the main thread must not suffer from being blocked and the target platform must support some way to provide activity when the main thread is blocked.

## Reason about design decisions when using the ‘blocking’ keyword

Besides hard constraints, Implications of ‘blocking’ for Dezyne applications also provides example situations where the usage of ‘blocking’ may or may not be warranted. Considerations to take into account are whether the period of asynchronicity can be used for other purposes. Due to contagious asynchronicity, other components may be simplified by usage of ‘blocking’, which could have implications for the layer of a system you wish to implement ‘blocking’ in.