# (C++) code integration tutorial

Version 1

Dezyne version 2.3.3

May 17th, 2017

# Introduction

## Learning goals

After following this tutorial, you will be able to:

* Integrate every kind of event type in Dezyne with handwritten (hereafter native) code
* Navigate through the essential areas of generated code
* Create a native environment for the generated code to run in
* Understand what the Dezyne runtime is and how it should be used within your own application
* Compile an application consisting of native and generated code (in C++)

## Intended audience and prerequisites

This tutorial assumes that you are familiar with the basics of the Dezyne modeling language and have some affinity with C++, Linux and the Raspberry Pi. It is assumed you are somewhat familiar with software designed from an event-based approach (<https://en.wikipedia.org/wiki/Event-driven_programming>).

The examples shown throughout this tutorial will be using the Eclipse Dezyne plug-in, although Dezyne can also be used as a stand-alone command line client. It is assumed that you have a Dezyne-equipped Eclipse running on your system before starting this tutorial.

Throughout the tutorial, we will be using various iterations of the Alarm System that can be found in the Dezyne introductory tutorial (<https://www.verum.com/supportitem/tutorial/>). Where applicable, a starting point of Dezyne models required for a step in the tutorial will be provided on Github.

Furthermore, throughout the tutorial you will find references to lambda expressions and polymorphism/inheritance in C++. If you are unfamiliar with these concepts, you may find the following resources useful:

* Lambda expressions: <http://www.cprogramming.com/c++11/c++11-lambda-closures.html>
* Polymorphism/inheritance: <http://www.cplusplus.com/doc/tutorial/polymorphism/>

## Platform choice

This tutorial will target the Raspberry Pi as hardware platform. The Pi can run a Linux distribution that is compatible with C++11 features that are used by generated C++ Dezyne code. With some extra components, it is also capable of driving some basic GPIO. The Raspberry Pi that was used to create this tutorial is a Raspberry Pi Model 2 B running Raspbian 8 (Jessie). The g++ version of the Raspberry Pi is 4.9.2.

The components used are as follows:

|  |  |
| --- | --- |
| Component | Quantity |
| Raspberry Pi | 1 |
| RGB Led | 1 |
| 220 Ohm resistor | 3 |
| Hardware button | 1 |
| Wiring | 1 |

It is, of course, not absolutely necessary to build an actual system that performs all of the tasks. You could also just write stubs for the hardware that will be called upon and run the application on your own system. During the tutorial we will consider actual hardware implementations, but most of the information will still apply if you are using stubs.

Table of Contents

[(C++) code integration tutorial 1](#_Toc490822057)

[Introduction 1](#_Toc490822058)

[Learning goals 1](#_Toc490822059)

[Intended audience and prerequisites 1](#_Toc490822060)

[Platform choice 2](#_Toc490822061)

[Chapter 1: Using Dezyne in your C++ environment 4](#_Toc490822062)

[Creating the native C++ environment 4](#_Toc490822063)

[Including and configuring the dzn runtime 5](#_Toc490822064)

[Inspecting the generated system 6](#_Toc490822065)

[Chapter 2: Implementing and integrating native components 9](#_Toc490822066)

[Next step in integration: detecting native components 9](#_Toc490822067)

[Implementation of events on interfaces 10](#_Toc490822068)

[Implementing generated ‘skeleton’ components 12](#_Toc490822069)

[Chapter 3: Compiling your application 16](#_Toc490822070)

[Breakdown of example makefile: the core 16](#_Toc490822071)

[Breakdown of example makefile: version control improvements 18](#_Toc490822072)

[Breakdown of example makefile: automating Dezyne features 18](#_Toc490822073)

[Chapter 4: Expanding the AlarmSystem 20](#_Toc490822074)

[Timer and Siren integration 20](#_Toc490822075)

[Handling data: password entry 25](#_Toc490822076)

[Chapter 5: Retrospect on learning goals 28](#_Toc490822077)

[Navigate through the essential areas of generated code 28](#_Toc490822078)

[Create a native environment for the generated code to run in 28](#_Toc490822079)

[Integrate every kind of event type in Dezyne with native code 28](#_Toc490822080)

[Understand what the Dezyne runtime is and how it should be used within your own application 29](#_Toc490822081)

[Compile an application consisting of native and generated code (in C++) 29](#_Toc490822082)

[Chapter 6: Code integration – extra materials 30](#_Toc490822083)

[Using the Dezyne locator to distribute (runtime) objects 30](#_Toc490822084)

[Default runtime objects 30](#_Toc490822085)

[Custom dzn::illegal\_handler 31](#_Toc490822086)

[Alternative logging methods 31](#_Toc490822087)

[Distributing your own data 31](#_Toc490822088)

[Thread safety in the Dezyne execution model 32](#_Toc490822089)

[Making use of private thread scheduling to poll hardware 33](#_Toc490822090)

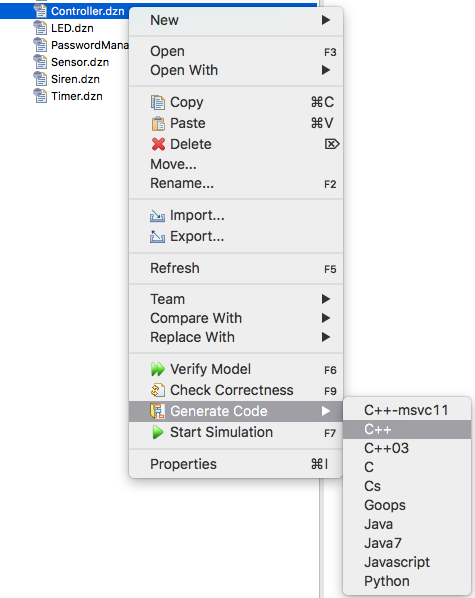
[Chapter 7: Integrating code in other languages 36](#_Toc490822091)

[Integrating code in C# 36](#_Toc490822092)

[Native components in C# 38](#_Toc490822093)

# Chapter 1: Using Dezyne in your C++ environment

## Creating the native C++ environment

The first step to start integrating your Dezyne models in an actual application is to generate code from all the models. This can be done in two ways, namely through the Dezyne application or by use of the command line client. In the Dezyne application, right-click on your \*.dzn files and navigate to the `Generate Code` menu item. There you can select multiple programming languages to generate code for. Select a location where the generated code should be stored and Dezyne will generate code for you. Later on in this tutorial, we will discuss how to automate this process by use of the Dezyne command line client and a makefile in Breakdown of example makefile: automating Dezyne features.

Up until now, we have approached the application as an event-based system. In order to run as a program on the Raspberry, though, we must have some kind of (infinite) loop that will provide the system with events. From here on after, this loop will be referred to as the **event loop**. Let’s start by creating a main function to contain this event loop for our application.

int main(int argc, char\* argv[]) {

  while(true) {

    // Dezyne interaction

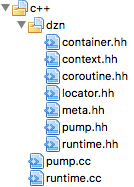
  }

}

The above main function will run until cancelled and each iteration of the while(true) loop will perform some kind of interaction with Dezyne later on. This is the easiest way of facilitating the event loop required by the Dezyne-generated code.

## Including and configuring the dzn runtime

Before the generated code can be used in combination with the event loop you just created, you must also download the Dezyne runtime to use in your project. The runtime can be found in Window  Show Views  Runtime. Once there, select the programming language you want to use and right click to download the associated files. (Just like code generation, downloading the runtime can be done by use of the command line client and a makefile; see Breakdown of example makefile: automating Dezyne features. This approach can be helpful in an environment that makes use of version control.)

The minimum required components of the Dezyne runtime that must be included in your application are the actual runtime and the Dezyne locator. The Dezyne runtime is a library of primitives used by the generated code; the locator is a mechanism to inject dependencies into a Dezyne system and its components. The runtime is an example of such a dependency that can be injected into the locator.

Their respective files that must be included are *runtime.hh* and *locator.hh*, found in the *dzn* subfolder. The inclusion of the runtime files should be done using ‘ <> ’ include tags as opposed to ‘ ”” ’ include tags; this is because of how code is generated from Dezyne models. This has some implications for compilation but those will be covered later in Chapter 3: Compiling your application.

When the files are included you can instantiate the Dezyne locator and runtime objects in your created main. The minimal setup that does all of the above will look as follows:

#include <dzn/runtime.hh>

#include <dzn/locator.hh>

int main(int argc, char\* argv[]) {

  dzn::locator loc;

  dzn::runtime rt;

loc.set(rt);

  while(true) {

    // Dezyne interaction

  }

}

Near the end of this tutorial, in some extra materials you will find a chapter dedicated to an in-depth look at the Dezyne runtime with more information on how you can customize some of the components of the runtime to your liking: Using the Dezyne locator to distribute (runtime) objects.

## Inspecting the generated system

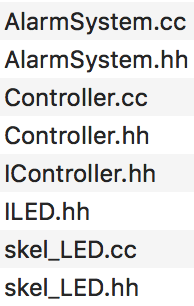
Up until this point you should have completed the following tasks:

* Generated code from your Dezyne models
* Created a main function with an (infinite) loop
* Downloaded and integrated the Dezyne runtime in your main loop

Before we start on writing implementations for the various components of the Alarm System, it is important to have a closer look at the generated code from the System component.

Recall that a System component is used to specify how components within the system interact with one another and how they are connected to the native environment outside of the system. This specification of connection to the outside world is why using a System component is so important; it is where and how your handwritten code will interact with the code generated from verified models.

The iteration of the Alarm System from the introductory tutorial we will be using for this can be found at <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Code_Integration/Ch1_Starting_Point>. If you generate code from these models, you should end up with the following files:



As the name of the System component specified in the Dezyne model is AlarmSystem, the generated files have been named *AlarmSystem.cc* and *AlarmSystem.hh* as well. Let’s start by having a look at *AlarmSystem.hh*. This file contains a definition of a struct AlarmSystem which shares some similarities with the System component as specified in Dezyne, which are highlighted in the snippet below:

struct AlarmSystem

{

  dzn::meta dzn\_meta;

  dzn::runtime& dzn\_rt;

  const dzn::locator& dzn\_locator;

Controller controller;

  LED led;

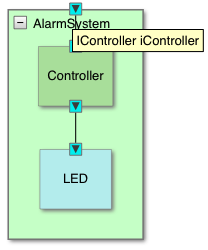
  IController& iController;

  AlarmSystem(const dzn::locator&);

  void check\_bindings() const;

  void dump\_tree(std::ostream& os=std::clog) const;

};

Most notably, it specifies which ports can be accessed on the AlarmSystem object (iController, in this case). **This directly maps to what ports are available on the boundaries of the system.** Another important factor is the definition of the AlarmSystem constructor, which requires a dzn::locator object to be passed as parameter. This is the case for all components and systems generated from Dezyne models, but luckily the System component will take care of distributing the dzn::locator for you! 

The other file that was generated from the System component, *AlarmSystem.cc*, contains the implementation of functions declared in the header file. The constructor generated for the system looks as follows:

AlarmSystem::AlarmSystem(const dzn::locator& dezyne\_locator) : dzn\_meta{"","AlarmSystem",0,0,{},{&controller.dzn\_meta,&led.dzn\_meta},{[this]{iController.check\_bindings();}}}

, dzn\_rt(dezyne\_locator.get<dzn::runtime>())

, dzn\_locator(dezyne\_locator)

, controller(dezyne\_locator)

, led(dezyne\_locator)

, iController(controller.iController)

{

  controller.dzn\_meta.parent = &dzn\_meta;

  controller.dzn\_meta.name = "controller";

  led.dzn\_meta.parent = &dzn\_meta;

  led.dzn\_meta.name = "led";

  connect(led.iLed, controller.iLed);

  dzn::rank(iController.meta.provides.meta, 0);

}

The contents of this file look a lot more daunting, but again you can relate some parts of the implementation to lines in the actual Dezyne model. For instance, the binding of ports that was done by controller.iLed <=> led.iLed in Dezyne can be traced to an invocation of the connect(led.iLed, controller.iLed) function.

In the implementation of the System’s constructor most of the magic happens, as a chain of constructors of components within the system is set off and the dzn::locator container (and as such, the dzn::runtime) is spread across the system. Using a System component for this purpose ensures that every component is using the same runtime, which is **essential** for Dezyne’s functionality. **For every Dezyne application you create, make sure to embed the components in a System component.**

Interaction with the AlarmSystem object in your event loop is how the logic modeled in Dezyne will perform in your fully integrated application. Later in the tutorial you will see how interaction with the System on its exposed ports will work, but to conclude this chapter on preparing the native C++ environment let’s do exactly that; prepare the main loop for use by creating the System object.

Most of the initialization steps were already completed in the earlier draft of your main function; the runtime files were included and the respective objects were created. What remains is to create an object of the AlarmSystem struct type and pass the locator to it. Then, before entering the event loop, it is recommended to invoke check\_bindings() on the System. This is a Dezyne functionality that ensures that all ports have been bound properly. The result will look like this:

#include <dzn/runtime.hh>

#include <dzn/locator.hh>

int main(int argc, char\* argv[]) {

  dzn::locator loc;

  dzn::runtime rt;

  loc.set(rt);

  AlarmSystem as(loc);

  as.check\_bindings();

  while(true) {

    // Dezyne interaction

  }

}

# Chapter 2: Implementing and integrating native components

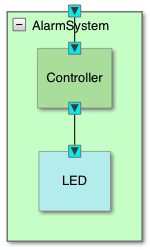
Through the course of this chapter of the tutorial, you will be gradually expanding the C++ environment you prepared in the previous chapter into an application where native code and Dezyne-generated code co-operate. At the end of this chapter you will be able to:

* Identify what must be implemented natively in your Dezyne application
* Integrate two of four kinds of event types known to Dezyne
* Make use of class inheritance to improve the Dezyne integration process

## Next step in integration: detecting native components

Let’s have a look at the situation currently at hand. If all went well, you should have a file which includes your main function, in which Dezyne runtime components were created and an instance of the Dezyne System was created. We have had a brief look at how the System controls its internal components and how to access ports on the System’s boundary.

The next step in the integration process is to implement the functionalities that could not be modeled in Dezyne such as driving hardware and manipulating data. To do this, though, you need to know two things: ***what* needs to be implemented** and ***where* to implement it** to make integration as smooth as possible.



The quickest way to figure out what must be implemented natively is by having a look at the System view in Dezyne. You may have already noticed there are two differently colored components in the AlarmSystem; one is a darker green, whereas the other is blue. These colors can be used to quickly assess whether a component was implemented in Dezyne. A blue component in a system indicates that the component in question was defined as a component without behavior. This should set off some alarm bells (pun very intended) because no behavior means no functionality. This leads us to a first conclusion: **blue components in a System view must be implemented in native code.**

The second indicator of implementation that must be done natively is the presence of ports on the boundary of a System in the System view. Such a port, as you can imagine, means that an interface is provided (or required) but there’s no entity in Dezyne that is making use of the provided interface (or implementing the functionality required by the System). This leads us to the second and last conclusion: **ports on the boundary of a System must be bound in hand-written code**.

So now that we’ve identified *what* needs to be implemented it’s time to look at *where* to do so. In the case of a blue component, a very specific method of implementation must be followed. You may have noticed the generation of *skel\_LED.cc* and *skel\_LED.hh* earlier on during the tutorial. These ‘skeleton’ files are generated for every blue component in a System. In these files, you will find an abstract class with pure virtual function(s) in the ‘skel’ namespace. The implementation of the abstract class **must** be done in files called LED.hh and LED.cc and the struct/class that inherits from skel::LED **must** be named LED. Although this may feel restrictive, this approach has considerable upsides too: the System becomes responsible for the distribution of runtime objects in the component and through the use of abstract classes and pure virtual functions you can verify that your implementation is complete compile-time instead of discovering such things run-time. We will cover the integration process of blue components in more detail later in Implementing generated ‘skeleton’ components.

*In case abstract or pure virtual are unfamiliar terms to you, please refer to the introduction of this tutorial where some links to additional information are given* *(Intended audience and prerequisites)*.

The second case we discussed was ports on System boundaries. The implementation method for this case is not as restrictive. The implementation may be done wherever you like, as long as all events on the interfaces of the unbound ports are bound eventually. This sounds a lot easier than having to deal with abstract classes and pure virtuals, and it is, but it comes with the tradeoff of bearing more responsibilities as developer.

In the next chapter, we will cover the implementation of events on unbound ports.

In general, it is recommended to encapsulate your application’s functionalities as components in a System. It increases the cohesion of the overall application and it makes it much easier to find the whereabouts of hand-written code in respect to the overall system. An added benefit is that all sorts of useful meta-information is automatically generated by Dezyne while your application is running which can help tremendously in tracking the behavior of your application.

## Implementation of events on interfaces

It’s time to put the information we gathered from the System view in the previous chapter to use. To get a feel for interacting with the generated System, let’s start with the ‘easy’ method of integrating handwritten code: binding unbound ports. The port that is currently not bound on the AlarmSystem is of type IController. From the perspective of the System, this is a *provides* port. The IController interface consists of two *in*-events, namely validPincode and sensorTriggered. If you have a look at the generated *IController.hh* file, you can see how this translates to generated code:

struct IController

{

  struct

  {

    std::function<void()> validPincode;

    std::function<void()> sensorTriggered;

  } in;

  struct

  {

  } out;

  // Dezyne meta information

}

For every interface in a Dezyne System, such a file is generated. As you can imagine, *in-*events on an interface can be found in the *in* struct and *out*-events in the *out* struct. These structs contain std::function objects, which can be invoked or assigned to.

In the learning goals, four event types were mentioned. These four types are the cross product of *in-* and *out*-events on *provides* and *requires* ports. There are two distinct integration steps that can be performed to cover the integration of the four event types. The following table serves as an overview of when to apply what integration step:

|  |  |  |
| --- | --- | --- |
| Port type | Event type | User action |
| Provides interface (on top of a component) | In | Call Dezyne function from handwritten code |
|  | Out | Assign handwritten code to system function |
| Requires interface (on the bottom of a component) | In | Assign handwritten code to system function |
|  | Out | Call Dezyne function from handwritten code |

In case of the IController port, a *provides* port, both of its events are *in*-events. Therefore, we must call the Dezyne functions from handwritten code. You may recall that earlier in the tutorial we examined the generated AlarmSystem and noted that its IController port can be accessed on the AlarmSystem object (Inspecting the generated system). Then, on the IController port, we can access events from the *in* and *out* structs. The calling of events should take place in the event loop which you created.

Let’s make the event loop a bit smarter, so we can use it to process some user input. If we then map the user input to triggers for the validPincode and sensorTriggered events, we can control the AlarmSystem. The following code snippet is sufficient for processing simple user input:

  std::string input;

  while(std::cin >> input) { // Loop is executed upon every newline from user input

    if(input.compare("s") == 0) {

      // s == sensorTriggered

    }

    else if(input.compare("v") == 0) {

      // v == validPincode

    }

  }

If you replace the comments with the corresponding function calls on the AlarmSystem object, the integration steps for the IController port are complete. As an exercise, replace the comments with the help of the information in this chapter.

Solution:

int

main(int argc, char\* argv[])

{

  dzn::locator loc;

  dzn::runtime rt;

  loc.set(rt);

  AlarmSystem as(loc);

  as.check\_bindings();

  std::string input;

  while(std::cin >> input) {

    if(input.compare("s") == 0) {

      as.iController.in.sensorTriggered();

    }

    else if(input.compare("v") == 0) {

      as.iController.in.validPincode();

    }

  }

}

At this stage, the AlarmSystem has no unbound ports that contain *provides/out* or *requires/in* events. Later in the tutorial, when we add timer functionality to the AlarmSystem, you will see how this integration step is performed (Timer and Siren integration). First, let’s have a look at implementing the LED component.

## Implementing generated ‘skeleton’ components

To start implementing the LED component’s behavior, create the *LED.cc* and *LED.hh* files. The AlarmSystem you generated from the Dezyne models already includes the *LED.hh* file by name, which is why you are restricted in naming it. In this *LED.hh* file you are also required to define a struct (or class, they are the same in C++) LED that inherits from skel::LED. The LED class definition in *LED.hh* will look like this:

#include "skel\_LED.hh"

class LED : public skel::LED {

}

By inheriting from skel::LED, the LED class is forced to implement all pure virtual functions defined in the base class. The pure virtual functions that are defined in the base class are all events that would normally be handled by the respective component’s behavior. Let’s take a look at the generated skel::LED class in *skel\_LED.hh*:

namespace skel {

  struct LED

  {

    // Dezyne meta information

    ILED iLed;

    LED(const dzn::locator&);

    virtual ~LED();

    // Dezyne meta informaton

    private:

    virtual void iLed\_setGreen() = 0;

    virtual void iLed\_setYellow() = 0;

    virtual void iLed\_setRed() = 0;

    virtual void iLed\_turnOff() = 0;

  };

}

As expected, the events we promised to implement in the ILED interface (setGreen, setYellow, setRed and turnOff) are pure virtual functions in the generated base class. You should add these function declarations to the LED class definition in *LED.hh* as an exercise.

Solution:

#include "skel\_LED.hh"

class LED : public skel::LED {

void iLed\_setGreen();

  void iLed\_setYellow();

  void iLed\_setRed();

  void iLed\_turnOff();

}

Lastly, as the LED class you’ll be creating will be constructed like any other generated Dezyne class, you will need to define a constructor for the LED that accepts a dzn::locator reference as parameter. The final version of the native LED class definition, from the Dezyne perspective, will look like this:

#include "skel\_LED.hh"

class LED : public skel::LED {

void iLed\_setGreen();

  void iLed\_setYellow();

  void iLed\_setRed();

  void iLed\_turnOff();

public:

  LED(const dzn::locator& loc);

}

Note that the constructor must be *public*; this is because the generated AlarmSystem directly calls the constructor of the LED class. The functions it implements do not have to be public; those functions are called through the LED class’ ILED port. The *skel\_LED.hh* file already includes the necessary files from the Dezyne runtime to be able to refer to dzn::locator, so that’s taken care of as well.

With the definition out of the way, what’s left to be done is the implementation of the functions of the LED class. Driving an LED on the Raspberry Pi can be done with the WiringPi library (<http://wiringpi.com>). To make use of WiringPi, its setup function needs to be called and the GPIO pins connected to the LED must be initialized. This is hardware-specific setup that should be performed in the implementation of the LED’s constructor; it will be performed when the constructor is called by the generated AlarmSystem. As an exercise, write an implementation for the constructor using wiringPiSetup() and pinMode() for your hardware setup.

Solution:

#include <wiringPi.h>

#include "LED.hh"

LED::LED(const dzn::locator& loc) {

  wiringPiSetup();

//Pin numbers are declared in LED.hh as PIN\_RED, PIN\_GREEN and PIN\_BLUE for readability

  pinMode(PIN\_RED, OUTPUT);

  pinMode(PIN\_GREEN, OUTPUT);

  pinMode(PIN\_BLUE, OUTPUT);

}

The solution above takes care of all the hardware setup, but that is not all we want to do with the native implementation of a Dezyne component. To fully integrate a native component in the AlarmSystem, you should call also call the constructor of the component’s base class to handle Dezyne related initialization. In C++, this is easily done by modifying the constructor to the following:

LED::LED(const dzn::locator& loc) : skel::LED(loc) {

This modification will call the constructor of LED’s base class skel::LED as well as perform the constructor you implemented in your native component. In skel::LED all of the Dezyne meta information and port binding is handled, so all you need to worry about in the native LED class is the functional behavior of the component.

To implement the setGreen, setYellow, setRed and turnOff functions you can do that like you would any other function. WiringPi provides a digitalWrite() function you can use to turn output on a GPIO pin on or off. All the colors we wish to display on the RGB LED can be created by a mixture of on and off on the three respective pins. Try implementing the remaining functions yourself as an exercise.

Solution:

void LED::iLed\_setGreen() {

  digitalWrite(PIN\_RED, LOW);

  digitalWrite(PIN\_GREEN, HIGH);

  digitalWrite(PIN\_BLUE, LOW);

}

void LED::iLed\_setYellow() {

  digitalWrite(PIN\_RED, HIGH);

  digitalWrite(PIN\_GREEN, HIGH);

  digitalWrite(PIN\_BLUE, LOW);

}

void LED::iLed\_setRed() {

  digitalWrite(PIN\_RED, HIGH);

  digitalWrite(PIN\_GREEN, LOW);

  digitalWrite(PIN\_BLUE, LOW);

}

void LED::iLed\_turnOff() {

  digitalWrite(PIN\_RED, LOW);

  digitalWrite(PIN\_GREEN, LOW);

  digitalWrite(PIN\_BLUE, LOW);

}

In summary: you now have a small but functionally complete burglar alarm program which accepts ‘s’ and ‘v’ commands from the user to denote a sensor trigger and valid password entry. Based on sequences of events, the program will display different colors on an RGB LED denoting whether the system is Unarmed, Armed or Alarming.

In the next chapter, you will find instructions on how to compile this application on a Raspberry Pi target so you can see it function in the real world!

# Chapter 3: Compiling your application

To compile your application, an example makefile has been included. In this chapter, we will explore the features this makefile offers and help you configure it to your environment’s needs. The makefile that was used to compile the AlarmSystem project can be found at <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Code_Integration/Ch3_Makefile>.

The nice thing about this makefile is that it can be re-used for other C++ projects that combine native code with code generated from Dezyne. Additionally, the makefile has gradually grown to facilitate version control. Dzn model files are treated as source code and checked into version control. Dezyne-generated C++ is treated similar to C++-generated object files: it is not checked into version control because it can always be regenerated.

## Breakdown of example makefile: the core

Let’s break the provided makefile down to the core so we can discuss what is actually needed to compile the AlarmSystem application. The makefile assumes you will compile the application on a Raspberry Pi. To accomplish this, you need to transfer the source files and generated files to the Pi filesystem. With all that in place, the following snippet will compile the integrated application on the Pi:

CXX = g++

CXXFLAGS = -std=c++11 -I$(RUNTIME) -I$(SRC) -lwiringPi -lrt

CPPFLAGS = -MMD -MF $(@:%.o=%.d) -MT '$(@:%.o=%.d) $@' -I$(SRC) -I$(RUNTIME)

LDFLAGS = -lpthread

TARGET = dznpi

SRC = ./src

RUNTIME = ./lib

SRCS = $(wildcard $(SRC)/\*.cc)

SRCS += $(wildcard $(RUNTIME)/\*.cc)

OBJS = $(subst .cc,.o,$(SRCS))

all:

make $(TARGET)

$(TARGET): $(OBJS)

$(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS) $(LDFLAGS)

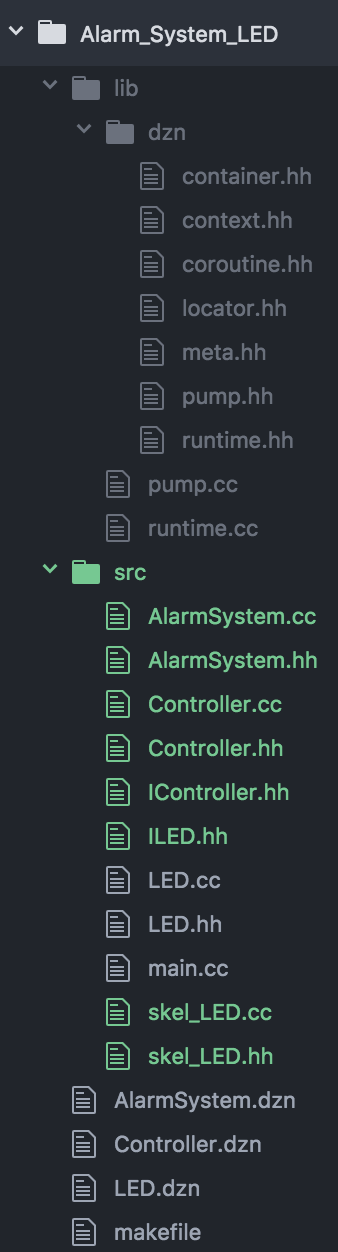
-include $(wildcard $(SRC)/\*.d $(RUNTIME)/\*.d)

This makefile will search for the Dezyne runtime files in the directory pointed at by the RUNTIME variable. The C++ source files of your application should be in the directory pointed at by the SRC variable.

Remember when we first inspected the generated system in Inspecting the generated system? You were told that inclusion of runtime files must be done using ‘ <> ‘ tags and that this would have some implications for compiling the application. These implications are represented in the makefile by the -I$(RUNTIME) -I$(SRC) flags in CXX/CPPFLAGS. This means that for every compilation step, RUNTIME and SRC point to directories that must be searched for included dependencies. With these flags added to your compilation recipe, ‘ <> ‘ tags can be used and the generated code can find its required headers.

The –lwiringPi and –lrt flags are required to be able to make use of the WiringPi library. Due to the –lrt flag, the pthread library must be linked to the executable post compilation; this is done with the LDFLAGS = -lpthread compilation step.

With this makefile and a properly structured filesystem containing your project, you can already compile an application consisting of native and generated Dezyne code. Awesome! In case you are unsure of what your filesystem should look like, you can use the Github repository as inspiration:



With this folder structure, if you type ‘make’ in a terminal window on your Raspberry Pi while inside the same folder the makefile is in, an executable called ‘dznpi’ will be created. If you run the dznpi executable on your Pi and have connected the RGB led to the GPIO pins you defined to be the Red, Green and Blue RGB pins then you should be able to control the dznpi application with sensorTriggered and validPincode events through the command line UI!

## Breakdown of example makefile: version control improvements

If you’ve used version control systems (VCS) before, you have probably encountered conventions for what files are allowed to be stored on the VCS. One strategy that often occurs is to not commit temporary files such as \*.o files to your VCS. To facilitate for this, you can add some clean-up recipes to the makefile like the following:

RM = rm –f

clean:

$(RM) $(OBJS) $(TARGET)

Typing `make clean` on your command line within the respective project folder will remove all of the generated \*.o files as well as the compiled executable. This is a good first step in cleaning up your working directory before committing your local changes to a VCS.

## Breakdown of example makefile: automating Dezyne features

**This improvement requires you to be able to access the dzn client from your command line. To learn how to do this, refer to** [**https://www.verum.com/supportitem/the-dezyne-command-line-tool/**](https://www.verum.com/supportitem/the-dezyne-command-line-tool/) **or follow the instructions on how to set your PATH variable after installation of Dezyne through the Eclipse client.**

In a way, generated code from Dezyne models can also be seen as temporary files, just like the \*.o files from the previous paragraph. The source files for the generated code are your \*.dzn files, and by including the generation of code in a makefile recipe it becomes easy to compile the application with only the makefile and source code in your VCS.

The generation of code from Dezyne models and its removal before committing can both be automated in the makefile. The generated code shares a version dependency with the runtime files, so to avoid any clashes on that aspect the example makefile also has a recipe to download the runtime.

For the generation of C++ code and the downloading of the Dezyne runtime, the following recipes were added:

These additions check the version you’re using against the newest possible version of Dezyne so you are notified if there is a new Dezyne available. The runtime recipe checks if the required folder structure is already present; if it is not, it will be created. Then, it checks Verum’s runtime repository and downloads the respective files for the C++ runtime.

VERSION = 2.3.3

CURRENT := $(shell dzn query | grep '\*' | sed 's,\\* ,,')

ifneq ($(VERSION),$(CURRENT))

$(info current version: $(CURRENT) is not equal to selected version: $(VERSION))

endif

runtime: | $(RUNTIME)/dzn

for f in $(shell dzn ls -R /share/runtime/c++ | sed 's,/c++/,,' | tail -n +3); do \

dzn cat --version=$(VERSION) /share/runtime/c++/$$f > $(RUNTIME)/$$f; \

done

touch $@

$(RUNTIME)/dzn:

mkdir -p $@

generate: $(wildcard \*.dzn)

for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC) --depends=.d $$f; done

touch $@

The generate recipe checks for all \*.dzn files in the root directory of your project where your makefile is located and generates C++ code for all of the interfaces and components in the files. The generated code is stored in the folder specified by the SRC variable that was declared earlier in the makefile; this ensures that native and generated code exist in the same directory. Lastly, the --depends option ensures that extra \*.dzn.d files are generated that provide information about what \*.cc and \*.hh files are generated from their respective \*.dzn files and their dependencies. The information in these files can be used for a final step in the VCS improvements: cleaning up generated code.

clean\_generated:

$(RM) `grep -h dzn $(SRC)/\*.dzn.d | sed -e 's,:.\*,,' -e 's,%,.,g'`

$(RM) $(RUNTIME)

$(RM) runtime generate

The recipe for cleaning up generated code is a bit harder to read, but in essence what it does is it reads the contents of the \*.dzn.d files (that were generated with the --depends option of dzn code generation). In these \*.dzn.d files, the names of generated files can be found; the make recipe clean\_generated then removes all of the files it has found from the contents of the \*.dzn.d files. The recipe also removes the runtime that was downloaded.

This should cover everything that is included in the example makefile. Depending on your preferences, you can choose to include or leave out some of its features; maybe you don’t care about having generated files in your VCS for example. As long as you include the core features discussed in Breakdown of example makefile: the core, you will be able to compile your application consisting of generated and native code.

# Chapter 4: Expanding the AlarmSystem

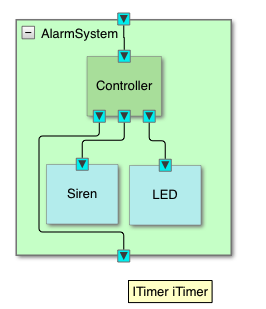
In the previous chapter on code integration, we did not cover all possible integration steps. In this chapter, we will gradually expand the AlarmSystem into the final version that can be found in the introductory tutorial, including an added functionality that lets us explore how data parameters are handled in generated and native code.

After this chapter, you will be able to:

* Integrate all four kinds of event types known to Dezyne
* Understand the role of data within the generated Dezyne framework

## Timer and Siren integration

One integration step you haven’t seen yet is the binding of native code to a System function (i.e. calling native code from Dezyne). This step is performed when you need to integrate an out-event on a provides port or an in-event on a requires port. Recall that the provides/requires is seen from a System perspective.

A good starting point for this integration step is the AlarmSystem with native LED component that you have already fully integrated by now. By adding the Siren and Timer from the introductory tutorial, you can learn how to perform this last type of integration. A starting point containing the relevant Dezyne models for this chapter can be found on <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Code_Integration/Ch4_Siren_Timer>.

Note that in the System component in AlarmSystem.dzn, a requires ITimer was added as opposed to using a component without behavior that provides ITimer. The reasoning behind this is that C++ already has access to a complete Timer implementation on most Linux distributions, including Raspbian which we are running on the Raspberry Pi. In such a case, it might be simpler to integrate the existing implementation as a required port like you will see in this chapter.

You can start off by generating code from the models again, either through the makefile or by use of the Eclipse client. Integrating the Siren should be rather simple after having integrated the LED already. We chose to stub the Siren functionalities with simple console output messages (Siren is activated, Siren is deactivated). Start off by integrating the Siren as an exercise.

Solution:

Siren.hh:

#include "skel\_Siren.hh"

class Siren : public skel::Siren {

public:

  Siren(const dzn::locator& loc);

  void iSiren\_turnOn();

  void iSiren\_turnOff();

};

Siren.cc:

#include <iostream>

#include "Siren.hh"

Siren::Siren(const dzn::locator& loc) : skel::Siren(loc) {

  //no op

}

void Siren::iSiren\_turnOn() {

  std::cout << "SIREN >>ACTIVATED<<" << std::endl;

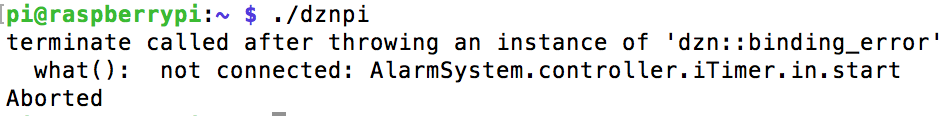
}

void Siren::iSiren\_turnOff() {

  std::cout << "SIREN >>DEACTIVATED<<" << std::endl;

}

After you have implemented and integrated the Siren, you should be able to compile the application again with the provided makefile. However, when you try to run the dznpi executable, the application will abort upon hitting the as.check\_bindings(); line that is processed before entering the event loop with the following error message:



As you can see, check\_bindings() discovered that iTimer’s start function has not been bound. Previously we could not see the effects of the check\_bindings() statement as we did not run the application yet. Now you can clearly see the difference between the integration of a component without behaviour and the integration of an unbound port; try leaving out the declaration and implementation of iSiren\_turnOn in your native Siren implementation. You will discover that you can’t even compile the application, whereas the fact that the iTimer port on the System was not bound couldn’t be discovered until the application was already running.

Now of course, this AlarmSystem application is rather trivial and there are few implications for running the application and hitting the check\_bindings() assert. However, imagine having a much more complex, expensive system that is partly through its initialisation process when hitting the assert- sure, you can implement measures to have the application shut down gracefully but it is a situation you would like to avoid. By using components without behaviour for your native implementations these issues can be found while compiling before the application is ever run. This does not mean you should skip calling check\_bindings() before initiating events on the System, however.

But let’s continue with the integration process. Like we mentioned earlier in this chapter, we can access a complete Timer implementation in C++ on the Raspberry- the alarm (<https://linux.die.net/man/3/alarm>) library. What remains to be done is to bind the events on the iTimer port to the controls of this alarm library. The table that was provided in the Implementation of events on interfaces chapter earlier in the tutorial suggests that the necessary integration step is to assign native code to a System function for in-events and call Dezyne functions from native code for out-events.

A quick recap: a System can expose ports, these ports contain structs for in-events and out-events where the in- and out-events are represented by std::function objects. For iTimer, which is a port of type ITimer, these are the generated structs:

 struct

  {

    std::function<void(long milliseconds)> start;

    std::function<void()> cancel;

  } in;

  struct

  {

    std::function<void()> timeout;

  } out;

For the iTimer port, a timer timeout must result in calling the out.timeout() function. From the manual for the alarm library, we can see that the alarm will generate a SIGALRM signal when the time has elapsed. Handling SIGALRM can be done by implementing a signal handler and binding SIGALRM to this signal handler in C++ as follows:

#include <signal.h>

void sigalrm\_handler(int signal) {

  // signal handling logic

}

// in main():

signal(SIGALRM, sigalrm\_handler);

The signal handler is called when SIGALRM is raised, so when the alarm times out. In this signal handler, you should not fire the timeout event into the Dezyne system immediately; the reasoning behind this has to do with how Dezyne’s execution model works based on certain assumptions. *This will be covered later in Thread safety in the Dezyne execution model. For now, consider it safe to call the Dezyne function in the signal handler, but keep in mind that this is* ***not*** *something you can generally assume.*

To be able to call the timeout function in the signal handler, you need to declare a global std::function object. The AlarmSystem you are interacting with in your event loop has its scope limited to the main() function, but the signal handler is separated from that. You can work around this by declaring a global std::function object and setting its value to the corresponding timeout event on the AlarmSystem’s iTimer port:

#include <iostream>

#include <string>

#include <unistd.h>

#include <signal.h>

#include <dzn/runtime.hh>

#include <dzn/locator.hh>

#include "AlarmSystem.hh"

std::function<void()> timeout;

void sigalrm\_handler(int signal) {

  // For now, call the global timeout() function object within this signal handler

  timeout();

}

int

main(int argc, char\* argv[])

{

  dzn::locator loc;

  dzn::runtime rt;

  loc.set(rt);

  AlarmSystem as(loc);

  as.dzn\_meta.name = "AlarmSystem";

  timeout = as.iTimer.out.timeout;

  signal(SIGALRM, sigalrm\_handler);

  as.check\_bindings();

  std::string input;

  while(std::cin >> input) {

    if(input.compare("s") == 0) {

      as.iController.in.sensorTriggered();

    }

    else if(input.compare("v") == 0) {

      as.iController.in.validPincode();

    }

  }

}

However, we still have not fully bound the iTimer port on the AlarmSystem. The in-events require an implementation in your native code; start(long milliseconds) and cancel() must be bound to their respective counterparts in the alarm implementation. The most efficient way to do this is by using lambda expressions. Starting and cancelling the alarm can be done by the following expressions:

* start(long milliseconds)  [](int ms){ alarm(ms/1000); }
* cancel()  [](){ alarm(0); }

To bind these implementations to their respective events on the iTimer port, a simple assignment will suffice:

#include <iostream>

#include <string>

#include <unistd.h>

#include <signal.h>

#include <dzn/runtime.hh>

#include <dzn/locator.hh>

#include "AlarmSystem.hh"

std::function<void()> timeout;

void sigalrm\_handler(int signal) {

  // For now, call the global timeout() function object within this signal handler

  timeout();

}

int

main(int argc, char\* argv[])

{

  dzn::locator loc;

  dzn::runtime rt;

  loc.set(rt);

  AlarmSystem as(loc);

  as.dzn\_meta.name = "AlarmSystem";

as.iTimer.in.start = [](int ms){ alarm(ms/1000); };

as.iTimer.in.cancel = [](){ alarm(0); };

  timeout = as.iTimer.out.timeout;

  signal(SIGALRM, sigalrm\_handler);

  as.check\_bindings();

  std::string input;

  while(std::cin >> input) {

    if(input.compare("s") == 0) {

      as.iController.in.sensorTriggered();

    }

    else if(input.compare("v") == 0) {

      as.iController.in.validPincode();

    }

  }

}

## Handling data: password entry

The Alarm System you created in the introductory tutorial assumed that the keypad the user entered their password with could assert whether the password was correct. This sounds like an awfully smart component, providing not completely related functionalities in capturing input and assessing the correctness of input. An easy way to split that up would be to separate these two actions; on the IController interface, change the validPassword event to a passwordEntered event with a string parameter and create a component that can be used to assess the validity of a given password.

The new IController interface would look like this:

interface IController {

  in void passwordEntered(String pw);

  in void sensorTriggered();

  behaviour {

    on passwordEntered: {}

    on sensorTriggered: {}

  }

}

The interface for a password management component would look like this:

extern String $std::string$;

interface IPWManager {

in bool verifyPassword(String pw);

behaviour {

on verifyPassword: reply(true);

on verifyPassword: reply(false);

}

}

As checking the validity of a password cannot be done in Dezyne, create a component without behavior that provides the IPWManager interface so this can be implemented in native code. Additionally, extend the behavior of the Controller component to make use of this new PWManager component by replacing the on iController.validPincode(): event statements with on iController.passwordEntered(pw): statements. The parameter to this function, pw, can then be verified with the new IPWManager port on the Controller component as follows:

[state.Unarmed] {

      on iController.passwordEntered(pw): {

      bool valid = iPWManager.verifyPassword(pw);

      if(valid) {

        state = State.Armed;

        iLed.setYellow();

        }

      }

      on iController.sensorTriggered(): {}

      on iTimer.timeout(): illegal;

    }

Implement these changes for the other two state behavior blocks as well (Armed, Alarming). Lastly, update the System component to reflect the new interfaces and components. Re-generate code from the updated Dezyne models and create the necessary files for the native implementation of the password manager.

As you were previously capturing user input to trigger events on the AlarmSystem, some changes will need to be made to how the user input was being processed. Instead of mapping ‘v’ to the validPassword event like we did earlier, the simplest way to accommodate for the change is to treat every user input that is **not** an event trigger as a password entry, like such:

std::string input;

  while(std::cin >> input) {

    if(input.compare("s") == 0) {

      as.iController.in.sensorTriggered();

    }

    else /\*user input counts as password\*/ {

      as.iController.in.passwordEntered(input);

    }

  }

Note that the function prototype in C++ for the passwordEntered event accepts std::string as parameter, just like declared in the Dezyne model by using extern String $std::string$;. By calling the passwordEntered event with the input string, the data object is sent to the AlarmSystem. Within the System, the data object is passed to another component; the PWManager component in this case. Although this data object has no meaning for Dezyne model code, Dezyne can transport such data objects across components. Of course, this is a trivially easy example but it should give you an idea of how data gathered from one component can be handled in another component.

The last thing that needs to be done is the native implementation of the PWManager component’s behaviour. As an exercise, define and implement the PWManager. The correct password should be “Dezyne” and should only be accessible to the PWManager class itself.

Hint: in C++, you can use string::compare or the operator== to compare strings with one another (<http://www.cplusplus.com/reference/string/string/compare/>).

Solution:

PWManager.hh:

#include <string>

#include "skel\_PWManager.hh"

class PWManager : public skel::PWManager {

  const std::string password = "Dezyne";

  bool iPWManager\_verifyPassword(std::string pw);

public:

  PWManager(const dzn::locator& loc);

};

PWManager.cc:

#include "PWManager.hh"

PWManager::PWManager(const dzn::locator& loc) : skel::PWManager(loc) {

  //no op

}

bool PWManager::iPWManager\_verifyPassword(std::string pw) {

  return pw.compare(this->password) == 0;

}

# Chapter 5: Retrospect on learning goals

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

* Navigate through the essential areas of generated code
* Create a native environment for the generated code to run in
* Integrate every kind of event type in Dezyne with native code
* Understand what the Dezyne runtime is and how it should be used within your own application
* Compile an application consisting of native and generated code (in C++)

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

## Navigate through the essential areas of generated code

Throughout the tutorial, code snippets from generated code were provided and key components in the generated code were highlighted. The intent is for you to be comfortable navigating through the code so that you can quickly find (and re-use!) what to implement in an inherited class or how to connect your event loop to the various System ports. In Inspecting the generated system you had a look at generated \*.hh and \*.cc files from a System component and in Implementation of events on interfaces and Implementing generated ‘skeleton’ components you encountered generated code for behaviour components and ports.

## Create a native environment for the generated code to run in

Most importantly, this means properly initializing the runtime objects, supplying an event loop and making sure that files are named correctly. Creating the native C++ environment, Including and configuring the dzn runtime and Next step in integration: detecting native components helped you in the initial steps of generating code, initializing the runtime and event loop and discovering which files need to be created. Chapter 3: Compiling your application provided some recipes to make the overall process easier.

## Integrate every kind of event type in Dezyne with native code

This learning goal covers the bulk of the tutorial, with many different examples on the various integration techniques. In Implementation of events on interfaces, the following table was provided as a reference of when to apply what action:

|  |  |  |
| --- | --- | --- |
| Port type | Event type | User action |
| Provides interface (on top of a component) | In | Call Dezyne function from handwritten code |
|  | Out | Assign handwritten code to system function |
| Requires interface (on the bottom of a component) | In | Assign handwritten code to system function |
|  | Out | Call Dezyne function from handwritten code |

Calling a Dezyne function from handwritten code can be seen in the event loop where sensorTriggered and passwordEntered events were fired as well as the Timer implementation where a timeout event was fired into the System.

Assigning native implementations to System functions was done with lambda expressions for the Timer start and cancel. Wrapping implementations in lambda expressions is an easy way to create std::function objects which Dezyne-generated C++ uses to store event handlers.

An alternative, more cohesive way of providing native implementations is inheritance of skeleton components. This technique requires a bit more work to set up, but when set up properly it is both safer and it becomes easier to track the behavior of your application. Examples for this technique are the LED, Siren and PWManager components.

The idea is generally the same as in the provided table, where assigning native code is replaced by implementing pure virtual functions. For out-events on required ports and in-events on provided ports, the calling of a Dezyne function is performed on the respective port.

## Understand what the Dezyne runtime is and how it should be used within your own application

At this point you know enough about the runtime to successfully use it in your application; the instructions in Including and configuring the dzn runtime cover this. However, the locator and runtime can be used for various other things as well to make your life easier. This information will be discussed in the next chapter, Using the Dezyne locator to distribute (runtime) objects. What is explained there is not required to create a functional application, however there are some techniques that may prove useful for later projects.

## Compile an application consisting of native and generated code (in C++)

Chapter 3: Compiling your application is, obviously, all about this specific learning goal. In this chapter, you can find an example makefile broken down into core features and quality of life improvements. If you followed the tutorial from the start up until this point, you will have noticed that the provided makefile handles expansion of the application very well. You should also be able to re-use the makefile for other Dezyne C++ projects easily.

# Chapter 6: Code integration – extra materials

During the course of this tutorial on Dezyne code integration, you have followed the basic steps to successfully integrate code generated from a Dezyne model. In this chapter, you will find some additional materials regarding more advanced aspects of the code integration process. The following items will be covered:

* Using the Dezyne locator to distribute (runtime) objects
* Thread safety in the Dezyne execution model & thread-safe-shell
  + Making use of private thread scheduling to poll hardware

If you are interested in learning more about features in the Dezyne modeling language, you are recommended to have a look at the next tutorial, which discusses the usage of the external keyword in Dezyne. The current Controller model has a race condition with its usage of the Timer which you cannot find without the usage of external; in the next tutorial you will learn more about increasing the robustness of your Dezyne models.

## Using the Dezyne locator to distribute (runtime) objects

In one of the first chapters of this tutorial, you created instances of dzn::locator and dzn::runtime to be able to construct the generated System and its components. It was mentioned that dzn::locator can be used for more than distributing the runtime; otherwise it wouldn’t make sense to have a separate wrapper, of course.

In this section we will explore some of the default objects stored in the dzn::locator that are used by all Dezyne components as well as discuss how to use the dzn::locator to distribute your own objects throughout a System.

The way the dzn::locator works is it stores all sorts of objects by type. You can use the set() function on a locator object to store an object of a certain type T in the locator, which has two possible outcomes:

* Object of type T did **not** previously exist in the dzn::locator; locator now contains an object of type T
* Object of type T did previously exist in the dzn::locator; previously stored object of type T is overwritten with new object of type T

### Default runtime objects

The dzn::locator creates two objects that are used by all Dezyne components, namely an instance of std::clog (which is an object of type std::ostream) for logging purposes and a default dzn::illegal\_handler:

struct locator

  {

  public:

    locator()

    {

      static illegal\_handler ih;

      set(std::clog).set(ih);

    }

### Custom dzn::illegal\_handler

If you wish to not use the default dzn::illegal\_handler, you can implement your own dzn::illegal\_handler and store it in the locator. The default dzn::illegal\_handler will then be overwritten and if an illegal assert occurs during the runtime of your application, the customized handler will be called. This is especially useful if your application consists of potentially dangerous hardware; if you can no longer guarantee the correctness of runtime software, you **will** want to shut down such hardware gracefully.

An alternative to using the illegal-handler to establish a safe pre-condition to terminating the application would be to write armour components to ensure that native code can never trigger an illegal. Armouring is a technique in Dezyne that can be used to guard control logic from spurious behaviour in native components; see <reference to Armouring documentation> for more information on this subject.

### Alternative logging methods

Trace logging by all Dezyne components is hardwired to use the std::ostream object in the dzn::locator. If you wish to use your own logging rules, for instance if you would like to log to a file instead of logging straight to std::clog, you can do so by overwriting the std::clog object in the dzn::locator.

To do this, you need to create an std::ostream object that uses a file as output buffer; say we want to log to “log.txt”. std::ostream can not directly write to files, it can only write to buffers; so, you need to create a buffer that writes to “log.txt”. This can be done using std::ofstream as follows:

std::ofstream logfile("log.txt");

std::ostream outstream(nullptr);

outstream.rdbuf(logfile.rdbuf());

With std::ofstream, a filestream to “log.txt” is created. The buffer for this filestream can be accessed by calling the rdbuf() member function of logfile. The buffer that the std::ostream writes to can be set using the same rdbuf() function, on the std::ostream object this time. In the end, you will have an std::ostream object that uses a filestream as its output buffer.

If you then use set() with the custom std::ostream object, your Dezyne application will send all of the trace logging to “log.txt” instead of displaying it in the console output. This could be useful for logging remote applications where you don’t have access to console output, for example.

### Distributing your own data

As mentioned in the introduction of this chapter, you can store any object in the dzn::locator, with the limitation that there can only be one object for any given type T in the locator at any point in time. Aside from overwriting runtime objects to provide a custom implementation, you can also use the locator to distribute data objects through the System, for example to be used in native components. Recall that native components are constructed by the System and their constructors require a dzn::locator by reference:

Siren(const dzn::locator& loc);

  Sensor(const dzn::locator& loc);

  LED(const dzn::locator& loc);

  PWManager(const dzn::locator& loc);

Within the implementation of the constructor, you can retrieve objects from the locator that is passed as parameter. This can be useful for initializing hardware. In the example implementations, we used hard-coded values for the GPIO pins. With the dzn::locator, you can distribute an object containing hardware configurations gathered from command line arguments to make the application more dynamic. This would be done by using the set() function to inject a configuration object into the locator, followed by using try\_get() in the constructor of a native component. try\_get() will return a pointer to the object of type T if it is found, or null if it is not found.

Say we have the following configuration struct:

struct HWConfig {

  int GPIO\_Red, GPIO\_Green, GPIO\_Blue;

};

In your main containing the event loop, you could initialize this struct with the use of command line parameters. If you then use set() on your locator with the initialized HWConfig struct, it will be stored in the locator. In the implementation of the LED constructor, you can do the following:

HWConfig \*config = loc.try\_get(HWConfig);

if(config) {

  this->PIN\_RED = config->GPIO\_Red;

  this->PIN\_GREEN = config->GPIO\_Green;

  this->PIN\_BLUE = config->GPIO\_Blue;

}

With this addition, the default GPIO pin definitions will be overwritten if you supplied a HWConfig struct in the dzn::locator. If you didn’t add the struct, try\_get will not be able to find a HWConfig struct and will return 0, so the if-statement will not be entered. If this is the case, you can simply default to preconfigured values.

## Thread safety in the Dezyne execution model

The Dezyne language semantics assumes that at most one thread is active in a component at any time. If you are working in a purely single-threaded environment where no exceptions in the thread execution model of your application exist, this condition is easily met. However, most applications will probably end up containing some thread concurrency. At this point, you need to implement a thread safety mechanism to ensure that data remains valid and no concurrent (thread) access takes place while Dezyne logic is being executed.

The easiest way to do this in Dezyne is to generate a thread-safe-shell for your System component. This thread-safe-shell will wrap its internal components in such a way that every function call is placed in an event queue. This event queue is generated with the thread-safe-shell and ensures that events are handled one-by-one in a private thread. Native code can interact with the System on its ports like normal, but interactions are placed in the event queue and processed by the private thread.

The generation of a thread-safe-shell can be done using the command-line client and the following command:

dzn code -l c++ -s SYSTEM FILE

The above command will generate a thread-safe-shell for the System named SYSTEM that is defined in FILE. For the System, \*.cc and \*.hh files will be generated with some key differences to their counterparts that were generated without the –s option. In \*.hh, the System struct now contains a dzn::pump object. dzn::pump is the object that represents the private thread and the associated event queue. In \*.cc, the constructor for the System now also implements wrapper functions so that incoming events are handled by the dzn::pump thread instead of the calling thread. *(Incoming events should be seen from the perspective of the System, so in-events on a provides port or out-events on a requires port)*

Earlier in the tutorial you were told that calling the timeout() function of the iTimer port is **not** safe by default (Timer and Siren integration) Calling an event from a signal handler is not safe by default. Signal handlers are similar to interrupt-service routines in the sense that they can be called while the process is involved in another function call.

With the default method of generating code from your System component, you run the risk of multiple active threads in the System. By generating a thread-safe-shell variant of the System, however, invoking the timeout event on the iTimer port will place the event in the event queue where it will be handled by the Dezyne private thread. With this change, thread safety is ensured without requiring any changes other than the code generation method. The easiest way to make the change is to add the following rule to the ‘generate’ make recipe:

generate: $(wildcard \*.dzn)

for f in $^; do dzn -v code --version=$(VERSION) -l c++ -o $(SRC) --depends=.d $$f; done

dzn -v code --version=$(VERSION) -l c++ -o $(SRC) -s $(SYSTEMNAME) $(SYSTEMFILE) --depends=.d

touch $@

SYSTEMNAME and SYSTEMFILE are variables that should be defined in the makefile.

## Making use of private thread scheduling to poll hardware

A native component is able to by-pass the dzn::pump when sending events into the System. Consequently, this means it is no longer possible to guarantee the single-thread-active convention within the System. This can be avoided by interacting with the dzn::pump within the System.

To demonstrate how you can interact with the dzn::pump in a native component, the final component of the AlarmSystem will be implemented and integrated: the sensor. Again, a snapshot of the Dezyne models for this stage of the application can be found at <github link>.

For the Sensor, a hardware button was used that needs to be debounced for reliable input readings ([https://www.arduino.cc/en/tutorial/debounce](https://www.arduino.cc/en/tutorial/debounce))). The debouncing algorithm provided in the Arduino tutorial works on a polling basis, which the event loop for the AlarmSystem does not support. On top of that, polling the native Sensor component outside of the safe Dezyne thread can lead to undesirable effects. Both of these challenges can be solved by making use of the dzn::pump event queue in the native Sensor implementation. If you store the logic that requires polling in a separate poll() function and place the poll() event in the queue while the Sensor is active, the private Dezyne thread will handle the polling and thread safety remains intact.

Below you can find a native Sensor implementation that implements the Arduino example debouncing algorithm by the use of dzn::pump to ensure thread safety.

Sensor.hh:

#include "skel\_Sensor.hh"

#include <dzn/pump.hh>

class Sensor : public skel::Sensor {

private:

  dzn::pump& pump;

  bool sensor\_value;

  bool last\_sensor\_value;

  unsigned long last\_dbnc\_time;

  bool polling;

public:

  Sensor(const dzn::locator& loc);

  void iSensor\_turnOn();

  void iSensor\_turnOff();

  void poll();

};

Sensor.cc:

#include <dzn/locator.hh>

#include <dzn/runtime.hh>

#include <wiringPi.h>

#include "Sensor.hh"

const int DEBOUNCE\_TIME = 50;

const int PIN\_SENSOR = 16;

Sensor::Sensor(const dzn::locator& loc) : skel::Sensor(loc), pump(loc.get<dzn::pump>()) {

  wiringPiSetup();

  pinMode(PIN\_SENSOR, INPUT);

  pullUpDnControl(PIN\_SENSOR, PUD\_UP);

  this->sensor\_value = false;

  this->last\_sensor\_value = false;

  this->last\_dbnc\_time = 0;

  this->polling = false;

}

void Sensor::iSensor\_turnOn() {

  this->polling = true;

  this->pump( [&] { this->poll(); } );

}

void Sensor::iSensor\_turnOff() {

  this->polling = false;

}

void Sensor::poll() {

  int new\_sensor\_value = !digitalRead(PIN\_SENSOR);

  if (new\_sensor\_value != this->last\_sensor\_value) {

    this->last\_dbnc\_time = millis();

  }

  if((millis() - last\_dbnc\_time) > DEBOUNCE\_TIME) {

    if(new\_sensor\_value != this->sensor\_value) {

      this->sensor\_value = new\_sensor\_value;

      if(this->sensor\_value) {

        this->pump( [&] { this->iSensor.out.triggered(); } );

      }

    }

  }

  this->last\_sensor\_value = new\_sensor\_value;

  if(this->polling) this->pump( [&] { this->poll(); } );

}

The interactions with the dzn::pump in the System are highlighted; the Sensor class has its own reference to the dzn::pump. This reference is set in the constructor by getting the System dzn::pump from the provided dzn::locator.

The usage of dzn::pump involves providing std::function objects of events that need to be handled. The easy way to do this is by wrapping functions in lambda expressions as can be seen in the above code snippet. Finally, a way to ensure that polling is continued while the Sensor is turned on is to have the poll() method store itself in the event queue recursively.

# Chapter 7: Integrating code in other languages

In this chapter, we will consider the actions denoted in the table of required actions to integrate code in other languages than C++. The following languages will be covered: C#.

For reference, the relevant table is supplied:

|  |  |  |
| --- | --- | --- |
| Port type | Event type | User action |
| Provides interface (on top of a component) | In | Call Dezyne function from native code |
|  | Out | Assign native code to system function |
| Requires interface (on the bottom of a component) | In | Assign native code to system function |
|  | Out | Call Dezyne function from native code |

The Dezyne models that are used in the examples can be found at <https://github.com/VerumSoftwareTools/DezyneTutorial/tree/master/Code_Integration/Ch1_Starting_Point>. Take note that we will be showing the integration steps only; we will not focus on the functional implementation.

## Integrating code in C#

Creating instances of the required runtime libraries is done similar to how it is one in C++. In C#, it is easier in the sense that no additional files must be included on a per-file basis. Preparing the runtime libraries is done as follows:

using System;

namespace MyDznApp {

  class main {

    private dzn.Locator dznLoc;

    private dzn.Runtime dznRt;

    public static void Main() {

      dznLoc = new dzn.Locator();

      dznRt = new dzn.Runtime();

      dznLoc.set(dznRt);

    }

  }

}

Take note that Locator and Runtime are located in the dzn namespace. Depending on the execution semantics of your application, you may choose to define the objects within the scope of Main or outside the scope (as can be seen above).

Creating an object to represent the System generated from Dezyne models requires you to pass the dzn.Locator object you created as parameter:

using System;

namespace MyDznApp {

  class main {

    private dzn.Locator dznLoc;

    private dzn.Runtime dznRt;

    private AlarmSystem as;

    public static void Main() {

      dznLoc = new dzn.Locator();

      dznRt = new dzn.Runtime();

      dznLoc.set(dznRt);

      as = new AlarmSystem(dznLoc);

    }

  }

}

In C#, events are generated as Action or Func objects, the C# variant of function objects. Action is used for void on events; Func is used for on events that have a return value. To reach an event on a port of a component, the following structure is applied in generated C#:

ComponentName.PortName.PortDirection.EventName, where ComponentName is the name of the data object representing the Component, PortName is the name of the Port you are trying to access, PortDirection is either inport or outport depending on the specification and EventName is the name of the on event you are trying to access.

As an example, let’s invoke the validPincode event on the provided iController port of the AlarmSystem (required action for *Provides/in* and *Requires/out*):

using System;

namespace MyDznApp {

  class main {

    private dzn.Locator dznLoc;

    private dzn.Runtime dznRt;

    private AlarmSystem as;

    public static void Main() {

      dznLoc = new dzn.Locator();

      dznRt = new dzn.Runtime();

      dznLoc.set(dznRt);

      as = new AlarmSystem(as);

      as.iController.inport.validPincode();

    }

  }

}

Here, the ComponentName is as; PortName is iController; PortDirection is inport; EventName is validPincode.

Let’s assume the provided IController port has an *out* event void foo(). According to the table in the introduction of this chapter, *Provides/out* and *Requires/in* require you to assign native code to a System function. void foo() will be represented as an Action object, which we must assign native code to. To assign to an Action object, we can use a lambda expression:

using System;

namespace MyDznApp {

  class main {

    private dzn.Locator dznLoc;

    private dzn.Runtime dznRt;

    private AlarmSystem as;

    public static void Main() {

      dznLoc = new dzn.Locator();

      dznRt = new dzn.Runtime();

      dznLoc.set(dznRt);

      as = new AlarmSystem(as);

      as.iController.outport.foo = () => { /\* Native implementation of foo() \*/};

      as.iController.inport.validPincode();

    }

  }

}

For more information on lambda expressions in C#, please refer to <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions>.

## Native components in C#

The interface iLED is provided by the LED component, which is a native component and therefore not generated from your Dezyne models. The generated AlarmSystem System component does refer to an object of type LED, so you will have to make this object yourself. **Take note that this object should not be contained in a namespace; the generated System refers directly to LED.**

A prototype of an LED class that complies to the ILED interface is as follows:

class LED {

  public ILED iLed;

  public LED(dzn.Locator locator, String name = "", dzn.Meta parent = null) {

    iLed = new ILED();

    iLed.inport.setGreen = this.setGreen;

    iLed.inport.setYellow = this.setYellow;

    iLed.inport.setRed = this.setRed;

    iLed.inport.turnOff = this.turnOff;

  }

  public void setGreen() {}

  public void setYellow() {}

  public void setRed() {}

  public void turnOff() {}

}

The LED class has a public member named iLed of type ILED; the name of this member variable has to be the same as the name of the provided interface of the respective component in your Dezyne model. Then, in the constructor the events of ILED that are to be implemented are bound to the implementations within the LED class.