1. Usage of *Elib*

This chapter is aimed at a programmer who wants to start working with an e-Puck using *Elib*. *Elib* offers a lot of tools, which make e-Puck programming easier. The tools are useless without a proper use. This chapter introduces guidelines and numerous examples how to use *Elib* and avoid problems.

On thirteen samples from *TestElib* project we present *Elib* and e-Puck's properties. The samples are written in *C#* and are densely commented. We suggest a programmer, who wants to program his e-Puck, to start with one of the examples from *TestElib* project and modify it according his needs. *Elib* is built robustly and improper use does not destabilise the operating system.

Let us note that this chapter describes the use of *Elib* and its tools, but does not explain basic .Net or *C#* features. The basic knowledge of .Net and *C#* language is required for understanding *Elib* examples. On the other hand .Net *delegates*, an *EventWaitHandle* class and usage of lambda functions in *C#* are shortly introduced. A great part of this chapter is devoted to *IAsyncResult* interface and its usage in *Elib*.

Elib library requires .Net 2.0 on Windows and requires Mono 2.0 and higher on Linux. As an example of an application, which uses *Elib*, we present *Elib Joystick* at the end of this chapter. *Elib Joystick* is a graphical application, which uses Windows Presentation Foundation. It requires .Net 3.5 or higher and is not portable to Linux. *Elib* can control every actuator and get every sensor value with *BTCom*'s version 1.1.3 on e-Puck. *BTCom* is a simple program running on e-Puck. It receives commands from *Elib*, performs the relevant actions and sends confirmation answer back. If we ask for a sensor value, the value is added to the confirmation message.

The source codes are deployed using a solution of Microsoft Visual Studio 2008 (MSVS) for Windows and MonoDevelop solution for Linux. If you use higher version of MSVS or MonoDevelop, then the solution can be easily upgraded.

The contents of this chapter is divided into 6 sections. The first section is devoted to advanced .Net techniques, which are used in *Elib*. The Section 1.2 shortly introduces features of each *Elib's* interfaces and it presents e-Puck's sensors and actuators. The next section goes through samples of implemented behaviours from *TestElib* project and explains the crucial part of the examples. The Section 1.4 presents the *Elib Tools* console application, which is useful during developing a program for e-Puck. Finally the last section sums up the most important guidelines for *Elib* library. It also shortly points out, which algorithms and applications can profit most from *Elib* design and which applications bring problems.

1.1 Advanced .Net techniques

Elib interface requires knowledge of *delegates*. Lambda functions and *EventWaitHandle* class are used in *TestElib's* samples and asynchronous programming is much more comfortable with them. If you are familiar with terms above, their usage in *Elib* will be described in Section 1.3.1. This section describes them in general.

Delegates are .Net wrappers for functions. Every .Net language is strongly typed and even functions are typed in .Net. A delegate has two meanings in .Net. The first meaning is the placeholder type for functions of given type. See the *C#* implementation in Figure 1.1. The placeholder type has name $OkfActuators$. The second meaning is the placeholder itself. In the sample code below the placeholder is called a.

Delegate variable can contain functions, which exactly match delegate definition. In the case of $OkfActuators$ delegate from Figure 1.1 the function must return *void* and it must have one argument of type object. All classes in *C#* are inherited from *Object* class, therefore every object can be passed to this delegate. Object class contains method $ToString()$ and therefore the number as well as the string are printed. Delegate function can contain more functions. On the other hand this feature is not used in *Elib*, because it reduces readability of the code.

The console output of $Example()$ function shows, that the functions are called in the order, in which the functions were added to the delegate variable. The functions have void returning value, because there is no chance how to access the returning value from the first function. Delegates allow the last function to return its value, but it is considered a bad manner, because another function added to delegate can overwrite the returning value of the delegate call.

Let us focus on the function, which is used in the $OkfActuators$ declaration. The function was defined and declared in place. Functions defined in place are called lambda functions and have several advantages. They can omit types of arguments, because they are inferred from delegate definitions. Lambda functions can also directly use variables from the scope of its declaration. The lambda function printed 8 in the enclosed output above, although we have not passed it as an argument to the delegate.

Explenation ($EventWaitHandle$)**.** The $EventWaitHandle$ class

$EventWaitHandler$ class is not tricky itself, but sometimes it is used in multi thread programming, which is usually complicated. An instance of the class is used to synchronise two threads. It is usually used to signal from one thread to another, that some work has been done. In *Elib* there are used two methods per $EventWaitHandle$ instance. Let us suppose we have a thread A running. Let B be the thread, which should perform a long task. The thread A wants after a while to wait until the work of B is finished. Before the job B finishes. A creates $EventWaitHandle$ e with parameters $false$ and $ManualReset$. $False$ parameter sets e to a blocking state. In the blocking state all threads, which have called method $e.WaitOne$, are synchronously blocked in $WaitOne$ method. $ManualReset$ means that the state of $EventWaitHandle$ can be changed only by its methods and $EventWaitHandle$ does not perform any action itself. Let us return back to threads A and B. A creates an instance e of $EventWaitHandle$ with mentioned parameters, passes e to the second thread. Finally A calls $WaitOne$ and blocks on this call. B thread works and after the job is done, it just invokes *Set* method on e. *Set* method releases all threads, which are blocked in $WaitOne$ method. Run the code from the snippet 1.2 to understand it.

```
// Delegate definition in some class e.g Program
delegate void OkfActuators(object data);
void Example(){
  int i=8;
  //declaration of delegate variable and initialization with ↩
     lambda function
  OkfActuators a = new OkfActuators((sth) => { Console.↩
     WriteLine("Lambda f{0},{1}",sth,i); });
  //second function is added to delegate
  a += new OkfActuators(suitableFunction);
  //invocation of 2 functions with string parameter
  a("Hurray!");
  //third function is added to delegate
  ExampleClass c=new ExampleClass();
  a+= new OkfActuators(c.suitableMethod);
  //invocation 3 functions with int parameter.
  a(-333);
}
private static void suitableFunction(object sth) {
  Console.WriteLine("suitable Function {0}",sth);
}
class ExampleClass{
  public void suitableMethod(object sth) {
          Console.WriteLine("suitable method {0}",sth);
  }
}
/
```

Figure 1.1: Definition of a delegate

```
Lambda fHurray!,8
suitable Function Hurray!
Lambda f-333,8
suitable Function -333
suitable method -333
```

EventWaitHandle is used in the function *endBehaviour* described in Section 1.3.1 used in behaviour implementation. Lambda functions can be seen for example in *ConsoleTestSensorsTimeout* function, which presents all sensors from e-Puck. Delegates are used in every command invocation to specify types of callback functions. Callback functions are functions, which are called after finishing an operation.

1.2 Explore *Elib* through examples

There are three public classes in *Elib*. *Sercom*, *Epuck* and *Stamp*. *Stamp* is for time measurement. *Sercom* and *Epuck* are classes, where all the algorithms are located. *Epuck* class uses *Sercom* internally. Let us focus only on *Epuck's* class, because *Epuck*'s basic interface is a specialisation of *Sercom*'s interface for version 1.1.3 of *BTCom*. For more information about interfaces and implementation of *Sercom* and *Epuck* see Section **??**.

Epuck class itself has two interfaces. Let us name them the basic interface and *IAsyncResult* interface. *IAsyncResult* interface is used widely through .Net. We will introduce it in the examples from *TestElib* project. All examples from this section come from *TestElib* project. We suggest reading this chapter with *TestElib* project opened and explore the samples from *TestElib* in detail.

The examples are listed from the simplest to the more complex. First function, which tests the communication between e-Puck and your computer is described, then Section 1.2.1 focuses on starting session using *Epuck*. Later in Section 1.2.2 simple functions describe all e-Puck's sensors and actuators using simple *Epuck's* interface.

Four methods, which invoke different behaviours using *IAsynResult* interface follow in Section **??**. One behaviour implemented by *Epuck's* basic interface is presented. After main section, which covers the behaviours, some "tips and

```
static EventWaitHandle e = null;
static void Athread() {
  Thread t = new Thread(Bthread);
  e = new EventWaitHandle(false, EventResetMode.ManualReset);
  t.Start();
  e.WaitOne();
  Console.WriteLine("Finally someone press the button!!!");
}
static void Bthread() {
  //simulate the work
  Console.Readline();
  e.Set();
}
```

Figure 1.2: Definition of a delegate

tricks" are shown. There is a behaviour in *TestElib*, that emulates *Epuck's* basic interface via *IAsyncResult*. It is depicted in the summary of Chapter ??. We will skip the behaviour, because it is implemented only for theoretical purposes. Apart from minor functions there is an example of image processing in "Tips and Tricks" part and a logging example. Logging *Epuck's* actions is the last example.

Remark. In the following paragraphs an exception means a subclass of *ElibException* if it is not said otherwise.

1.2.1 Set up and disposal of a session

Let us start with a function, which does not use *Elib*. *TestPortTurnAround* opens a given serial port and sends some commands to move and stop, then it ends. It does not require any feedback and throws no exceptions. It is useful to run it, because a lot of errors are not hidden in the code, but in the hardware set up.

Function *startEpuck* returns an instance of *Epuck*, which allows you later to control an e-Puck.

```
return new Epuck(port, "Ada"); //Name it. It is useful for ↩
   logging and debugging.
```

The function throws exception only if other application blocks the port. Operating system remembers the port, which is set up for the communication with e-Puck, therefore no exception occurs if e-Puck is not even turned on and was used before. In such a case *Kof* callback from *Epuck's* basic interface or an exception from *IAsyncResult* are raised after sending a command to e-Puck. Sending a command to e-Puck is done via calling one of *Epuck's* functions. Both *Kof* callback and exception let user know, that a command was not delivered in time. Let us mentioned the closing of session and then we introduce both interfaces.

```
//dispose can take a while (under 500ms)
ada.Dispose();
TextWriter t= ada.LogStream;
t.WriteLine("We are trying to reconnect");
ada.BeginStop(0.1,null,null); //throw an exception
ada=new Epuck(ada.Port,ada.Name);
ada.LogStream=t;
ada.StartLogging();
ada.Stop();//does not throw an exception
```

The function *endEpuckSession* sends a command to stop your robot, but it primarily closes the session by calling *Dispose* method. Dispose method releases serial port for another application. After disposal *LogStream*, *Port* and *Name* are the only properties of *Epuck* instance, which can be accessed. They can be used to reconnection, see above.

1.2.2 Touching e-Pucks's sensors and actuators

ConsoleTestActuatorsTimeout function presents all actuators. Let us choose to control the motors of e-Puck.

```
ada.Motors(1,-1,
  (nth) => { Console.WriteLine("Motors(..) OK "); end = true;
     },
  (nth) => { Console.WriteLine("Motors(..) KO"); end = true;
     },
  null, myto);
wait(0);
ada.PlaySound(3,
// ...and so on
```

Motors method controls the speed of wheels. See code above. Maximum forward speed is +1, backward speed -1. Both motors are controlled at once via first two parameters. Next two parameters are functions, which match *Okfactuators* delegaters, respectively *KofCallback* delegates. Both delegates have only one *object* parameter. In this example *null* value is passed to the lambda function through the fifth argument of *Motors*. The last parameter is the *timeout*, which tells *Epuck* how long it can wait to answer of *BTCom* in seconds. The first delegate is called if the answer from *BTCom* is confirmed before *timeout* has elapsed. *KofCallback* is called otherwise. *ConsoleTestActuatorsTimeout* function wants to present the reaction time of all commands, therefor it does not allow commands to be called asynchronously one after another. Function *wait*(0) forces the current thread to wait until answer is delivered or *timeout* elapsed. Simple synchronization is done via *end* flag.

```
static void wait(int gap) {
  ///<remarks> Simple but unefficient way of waiting. See
     KofOkfWaiting(..) in Behaviour for usage of
     EventWaitHandle.</remarks>
  while (!end) {
    Thread.Sleep(5);
  }
  end = false;
  Console.WriteLine("Ended: {0}",Stamp.Get());
  Thread.Sleep(gap);
  Console.WriteLine("Start: {0}", Stamp.Get());
}
```

ConsoleAsynchronousSensors function asks for sensors too, but does use synchronous waiting only at the end. Run the example and note the huge gab between the computer time after asynchronous calls and the computer time after synchronisation. The time is measured in seconds!

The last function in the introductory section is *GetImage* function. The function creates a window after an image is captured. The *TestElib's* application is blocked until the windows is opened. To see the picture from e-Puck's camera switch from the running *TestElib's* console to the new window.

```
IAsyncResult ar = e.BeginSetCam(40, 40, Zoom.Small, CamMode.↩
   Color, toSetCam, null, null);
e.EndFtion(ar);
ar = e.BeginGetImage(toImg, null, null);
Bitmap bm = e.EndGetImage(ar);
```

ShowImage uses an *IAsyncResult* interface to set the camera and get the picture. Let us shortly introduce the interface on *BeginSetCam* method, which called on an instance *e* of e-Puck. *IAsyncResult* interface allows easily to start asynchronous operation and after that wait for the result. *BeginSetCam* starts the asynchronous operation, which sends the relevant command, and *EndFtion* waits to it. The *EndFtion* can wait for *BeginSetCam* result, because *ar IAsyncResult* instance was created in *BeginSetCam*. Every *IAsyncResult* asynchronous function start with "*Begin*" prefix. If it asks for a sensor value, then it starts with "BegingGet" as you can see on *BeginGetImage*. *BeginGetInfoVersion* and *BeginGetInfoHelp* functions start with "*BeginInfo*" prefix, their return values never change.

The "End" functions differ according to the type, which is returning from their invocation. Sensors do return a value. Most of functions return *int*[] array. They start with "BeginGet" followed by the name of the sensor. However,*BeginGetImage* returns a *Bitmap*. Last type of the functions, which ask for *BTCom's* help and version, return *string*. *EndFtion* can be applied on every *Elib's IAsyncResult's* invocation. In other words *EndFtion* can be used to wait on every "Begin" function, but it is not capable of returning values. Functions *EndGetFtion*, *EndGetImage* and *EndInfoFtion* returns relevant values, but these "Get" functions can be called only in pair with "Begin" functions, which ask for a sensor's value of the same type as a return value from "Get" function.

An *IAsyncResult* "Begin" function has always three arguments. If a function has more than three arguments, the obligatory arguments are the last. Let us look at *BeginSetCam's* arguments. Width and Height of image are first two parameters and together with zoom and colour is specific for *BeginSetCam*. Next comes *timeout*, which tells how long we are willing to wait to the answer. Instead of the *null* values callback and its parameter of *object* type can be specified. If we use $EndFtion(ar)$ we do not need callback.

Callbacks are last not introduced feature of *IAsyncResult*. They are presented in the next section devoted to simple behaviours of e-Puck robot.

1.3 Behaviours

A behaviour is a program, which controls a robot. It can be described as an finite automaton, where the transition from one state to another is based on sensor values. States represent actions, which robots perform.

We present behaviours invoked by functions *Bull*, *Billiard*, *GoAndTurn*, *Go2Light* and *KofGoXcm*. *Bull* behaviour acts like a bull. If there is a red obstacle, robot attacks it. In *Billiard* behaviour e-Pucks goes from an obstacle to an obstacle and if it is very near it turns around. *GoAndTurn* let the robot drive along a square with 15 cm long sides. There are other functions like *goXcm* or *turnAround* available. All mentioned behaviours are implemented using *IAsyncResult*. *KofGoXcm* is the only one behaviour, which is implemented by using simple *Epuck's* interface. It allows robot go almost exactly a given amount of centimeters, even if the connection breaks during the ride. All behaviour used callbacks, because it is a natural way how to switch between states of a behaviour.

Callbacks of "Begin" functions from *IAsyncResult* return *void* and take one *IAsyncResult* instance as an argument. The *state* parameter, which is passed to "Begin" function can be found in *AsyncState* property from *IAsyncResult* argument. In Figure 1.3 *BeginStop* function is invoked on line 3 with *EasyDisposal*(..) callback and *ada* instance as parameter for callback. On line 7 of Figure 1.3 the *ada* parameter is extracted from *IAsyncResult ar*. If we use "BeginGet" function to get a sensor values, then the values are returned by "EndGetFtion" call on *ar* as we can see on the line below.

Callbacks are called always after the operation finish, therefore the *EndFtion*(..) on *IAsyncResult* does not wait. Furthermore no *EventWaitHandle* is created, which results in a better performance if we use callback. For information how this feature is implemented see Section **??**.

Callbacks are invoked in separate threads, which bring complications, if we invoked more than one behaviour from one function as we do in the *Main*(..) function. The problem is that *Epuck* instance gets commands from completely different behaviours at the same time. Fortunately, all behaviours in *TestElib* let the thread with *Main*(..) function wait, until they are finished. After end of one behaviour the next is invoked.

Next we introduce a guideline for implementing the simple behaviour, which introduces how to avoid problems with ending a behaviour. It is very simple. Only one function has to be called with callbacks in each method and ,furthermore, it has to be the last action. It allows you to use *Dispose*(..) method without worrying about accessing instance of *Epuck* from different threads. See the code and mainly the comments in Figure 1.3. We suggest to return to this example after exploring some of the behaviours. All the behaviours use this attitude, but have to care of its ending, because they are invoked from the *Main*(..). See the second comment in *Main* function from Figure 1.3.

1.3.1 *IAsyncResult* Behaviours

Let us finally introduce the *Bull* behaviour. Running this behaviour e-Puck behaves like a bull and bumps into red obstacles. *Bull* behaviour switches between three states. In the first state it travels randomly. After a random time it switches to observation state. In observation state it shoots 4 images from four directions. If the images are red enough it switches to aggressive behaviour and goes to the red obstacle, otherwise it starts a random ride again.

The start function firstly sets e-Puck's camera to desired settings then it invokes the behaviour. *CountDown* is a struct, which wraps the e-Epuck and number of rotation.

```
public static void Bull(Epuck ada) {
  startBehaviour();
  try {
    IAsyncResult ar = ada.BeginSetCam(40,40, Zoom.Small, ↩
       CamMode.Color, toSetCam, null, null);
    ada.EndFtion(ar);
    ada.BeginGetImage(0.5, searchAround, new CountDown(ada, ↩
       3));
    endBehaviour();
  } catch (ElibException e) {
    exceptionOccured_End("Exception was thrown in Bull(..)", ↩
       e);
  }
}
```

The behaviours run until the user presses a key. The waiting is performed in the main thread. The *Bull* behaviour as all other behaviours runs in worker threads. If an exception, which signals undelivered command, is thrown, its message is written to a console and the behaviour ends. Explore code in *exceptionOccured_End* function to see more.

See code below how the behaviour is ended in nice way. It is a tricky part and is not necessary if only one behaviour is implemented. See Figure 1.3 for safe ending of a single behaviour. Important is that *startBehaviour* is called at the

```
static int Main(){
  //some work, Epuck ada created
  ada.BeginGetIR(0.1, EasyDisposal,ada);
  // no code using ada is allowed after BeginStop call
}
static void EasyDisposal1(IAsyncResult ar){
  Epuck ada = (Epuck)ar.AsyncState;
  int[] irSensors = ada.EndGetFtion(ar);
  // synchronous work e.g to stop using EndFtion
  ar=ada.BeginStop(0.1, null, null);
  ada.EndFtion(ar);
  //asynchronous call has to be the last!
  ada.BeginPlaySound(4,SafeDispose, ada);
}
static void SafeDispose(IAsyncResult ar{
  Epuck ada = (Epuck)ar.AsyncState;
  ada.Dispose();
}
```

Figure 1.3: Safe disposal for single behaviour

first line of *Bull* method and *endBehaviour* is called on the last one.

```
static void startBehaviour() {
  endf = false;
  endconfirmed = new EventWaitHandle(false, EventResetMode.↩
     ManualReset);
}
static void endBehaviour() {
  //If you press an key, endf is set to true. ->
  //->All behaviour recursive functions look like: if(!endf)↩
     {..body..}else endconfirmed.Set();
  Console.WriteLine("Press enter to quit..");
  //Behaviours are usually infinite loops, they run and run.
  Console.ReadLine();
  ///<remarks>Blocks invoking next function</remarks>
  endf = true;
  ///Wait on EventWaitHandle until currently running function↩
      is finished.
  endconfirmed.WaitOne((int)(1000 * (toReset + toImg + to)));
  //releasing EventWaitHandle
  endconfirmed.Close();
}
```

1.3.2 *Bull's* internals

The diagram 1.4 shows the states of *Bull*. Three states are named after the functions, which control the robot. State "GetStacked" is the final state and it arises, if the e-Puck is surrounded with obstacles from the front and from behind. The *searchAround* procedure calls itself 4 times if no red obstacle is in front of e-Puck. It uses the struct *CountDown* to count the number of rotations.

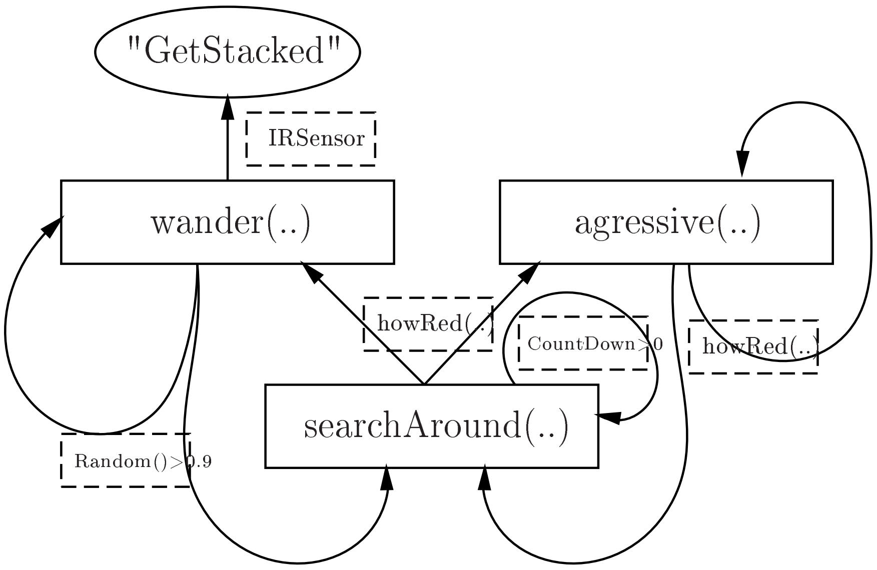

Figure 1.4: Bull behaviour diagram

Aggressive behaviour tries to bump into red obstacles. If it finds out an obstacle not red enough, it starts searching again. After four 90 degrees rotations in *searchAround* are made, the *wander* behaviour is switched on. From *wander* state the behaviour can switch back to *searchAround* or it ends stacked. Let us show how simple the *aggresive* function can be.

```
double redl = 0.2;
IAsyncResult ar = ada.BeginGetImage(toImg, null, null);
Bitmap a = ada.EndGetImage(ar);
howRed(a, out red, out dir);
if (red > redl) {
  ada.BeginMotors(0.4, 0.4, to, agressive, ada);
} else {
  ar = ada.BeginStop(to, null, null);
  ada.EndFtion(ar);
  ada.BeginGetImage(toImg, searchAround, new CountDown(ada, ↩
      3));
}
```

1.3.3 Billiard ball behaviour

The behaviour is invoked by *Billiard* function. The behaviour is rather simple. It goes to a wall and before the wall it stops. Then the robot tries to compute the orientation of the wall it turns around to go in a next direction and than it goes straight to the next obstacle. The behaviour runs without ending until the e-Puck is surrounded by obstacles.

The *rebound* function tries to compute the angle of rebound, but as well the simple *go2wall* function implements a restart of the behaviour. Let us focus on this part. We will show it on *go2wall* function in Figure 1.5

The exception can be throw on only in call of *EndGetFtion* at the beginning and no action in the body was done, so the function can be called again from the catch block without problems.

The problem of this implementation is the logic, not the implementation. The restart of the *go2wall* function does not solve the problem with slow delivery of IR sensors, it only tries and tries again until enter is pressed. The solution will be presented in next behaviours.

Next interesting feature of this example is not calling the *EndFtion* to wait until *BeginMotors* is confirmed. We suppose that if the commands of *BeginGetIR* sensors are delivered, than the *BeginMotors* are delivered too. Furthermore only one command delivered is enough to go to the next obstacle.

1.3.4 Exact movement and *GoAndTurn* behaviour

Exact movement is a very strong feature of e-Puck. E-Puck uses two stepper motors, which allows you to travel exactly 10 cm. The deviation is less than

```
static void go2wall(IAsyncResult ar) {
  //value to decide if an obstacle is near enough
  int frontLimit = 1000;
  //does not throw ElibException
  Epuck ada = (Epuck)ar.AsyncState;
  try {
    // Doesn't create EventWaitHandle because the action has already completed synchronously.
    // Can throw an TimeoutElibException
    int[] ir = ada.EndGetFtion(ar);
    if (ir[0] + ir[7] > frontLimit)
      ada.BeginStop(to, rebound, ada);
    else {
      //Does not use EndFtion, it saves the EventWaitHandle. We suppose, that it succeeds now or in next rounds.
      ada.BeginMotors(0.2, 0.2, to, null, null);
      // The BeginGetIR command is enqueued in the same momment as BeginMotors,
      therefor double timeout is used.
      ada.BeginGetIR(2 * to, go2wall, ada);
    }
  } catch (ElibException e) {
    Console.WriteLine("Billiard restarted in go2wall, because of exception:\n" + e.Message);
    // Invokes go2wall function again. It needs to be invoked by BeginGetIR command, because it expects ar with IR values.
    ada.BeginGetIR(to, go2wall, ada);
  }
}
```

Figure 1.5:

one millimeter. E-Puck has also encoders, which allow you to measure the distance of travel with the same resolution. On the other hand, communication over Bluetooth brings problems, which are typical for common electric motors. Usual motors have an inertia and they do not stop immediately and they do not reach the desired speed at once. The same problem is with *Elib* and Bluetooth communication. The commands have a delay. A common problem is also with a flat tyre, because one wheel has bigger perimeter than the other. Programmers of e-Puck do not have to solve these problems due to good design of e-Puck.

GoAndTurn is the only behaviour, which does not run in an infinite loop. It just goes around a perimeter of square with side of 15 cm. It is based on *goXmiliseconds* function, which lets the robot ride with given speed for a specified time. Functions *TurnAround* and *goXcm* are simply built on *goXmiliseconds*. The *square* function from *GoAndTurn* behaviour simply combines the functions. Let us introduce *goXmiliseconds* function.

```
static void goXmiliseconds(Epuck e, double L, double R, int ↩
    milisec, double addto) {
  int x = e.Working;
  if (x != 0)
    throw new ElibException("It would be extremely inaccurate↩
        , if commands are still waiting to be sent.");
  IAsyncResult ar = e.BeginMotors(L, R, addto, null, null);
  AsyncResultNoResult pom = (AsyncResultNoResult)ar;
  pom.Name += "goXms";
  e.EndFtion(ar);
  ar = e.BeginStop(addto, null, null);
  e.EndFtion(ar);
}
```

Figure 1.6: Function goXmiliseconds

The function needs to have no queued commands in *Epuck* instance, because it supposes that sending takes almost no time and any waiting makes it very inaccurate. The function sends a command to motors, then waits a given time and then stops. The idea behind the little heuristic is that every commands sending takes the same time and also the commands transferred to e-Puck and from e-Puck are equally fast.

More interesting feature presented here is extracting an *AsyncNoResult* class from *IAsyncResult* interface. For more information see Section **??**. From user's point of view it is good to know, that *Name* property can be changed. *Name* property is used in logging. Its default value is the name of "Begin" function, which created the instance of *IAsyncResult*. The *Name* attribute is a feature of *Elib* and is not defined in *IAsyncResult*, therefore the cast is needed for accessing it.

1.3.5 Restarting *Go2light* behaviour

Robot following the light is a typical robotic task. *Go2Light* behaviour uses only one recursive function, which implements the behaviour. The following example also presents a guideline how to cope with unreliable connection in an infinite behaviour.

```

static void recGotoLight(IAsyncResult ar) {
  Epuck ada = (Epuck)ar.AsyncState;
  try {
    ar = ada.BeginGetLight(to, null, null);
    int[] light = ada.EndGetFtion(ar);
    //..omitted part of source code: Debugging printouts
    if (diff_fb > 0) {
      if (light[2] < light[5]) {
        Console.WriteLine("turn around right {0}", diff_lr);
        ar = ada.BeginMotors(speed, 0, to, recGotoLight, ada)↩
            ;
      } else {
      //..omitted part of source code: the lighst is on front↩
          left, back right,...
      }
    }
    //there is no need to repeat EndFtion in the branches
    ada.EndFtion(ar);
  } catch (TimeoutElibException) {
    exceptionOccured_Restart(ada);
  }
}
```

Figure 1.7: Function *recGotoLight* function

See the clear structure of *recGotoLight* function is in source code in Behaviours.cs file. Let us explore the interesting *exceptionOccured_Restart* function in Figure 1.8. The function ends the session with real e-Puck and closes a serial port by calling *Dispose* method. After the disposal, the commands can not be sent to e-Puck and we have to create a new connection. We use the same parameters. Optionally, here is the place to ask the user of behaviour to change a port name. The same approacapproachh is used in *KofGoXcm*, which will be presented below.

1.3.6 Behaviour with return implemented via *Epuck's* basic interface

The behaviour is invoked by *KofGoXcm* function. The robot goes specified amount of centimeters. If the connection breaks during the behaviour, the behaviour ask the user to repair the connection and to keep e-Puck running. If the connection is repaired successfully then e-Puck goes to the destination, where it should have ended before the connection failure.

The behaviour uses *Epuck's* basic interface, which has been introduced in Section 1.2.2. It has some design consequences. The example has two implementations, in fact two behaviours. The behaviour, which has a good connection at disposal, and the behaviour, which has to deal with a broken connection. We also keep the guideline from Section 1.3. It says, that the asynchronous call should be made only once at the end of a function. It is necessary to avoid parallelism and problems with ending of the behaviour.

In summary, the amount of functions grows rapidly, because we can not easily synchronously wait to answers from e-Puck and furthermore we have to implement two kinds of functions: Kof and Okf callbacks. The implementation of $okf(..)$ callbacks is straightforward. We will focus on a function, which is called to stop the e-Puck after it travelled the required distance.

Let us remind of the *Epuck* basic interface. In Figure 1.9 class *RobotAndTime* is used. As you can see, the structure of the code does not differ from *IAsyncResul*. First the necessary cast is performed in order to extract *RobotAndTime* class, then the logic is implemented and in the end the commands to e-Puck are sent. The Kof and Okf commands use the same state argument for passing data, e.g. *RobotAndTime* instance. The state argument together with timeout are obligatory and are located at the end of functions. The Okf callback for a sensor command has additional first argument. The first argument is used to return sensor values. For example the callback for *GetIR* command looks like *void OkGetIR(int[] values, object state)*;.

The logic of this example is very interesting. After five unsuccessful attempts to stop, the *reconnect* function prompts the user to repair the connection. If the

```
static void exceptionOccured_Restart(Epuck ada) {
  //Reconnect again to e—Puck
  ada.Dispose();
  ada = new Epuck(ada.Port, ada.Name);
  restarts−−;
  if (restarts >= 0) {
    Console.WriteLine("Remaining " + restarts.ToString() + " ↩
       restart(s). Press enter to continue");
    ada.BeginStop(to, recGotoLight, ada);
  } else {
    Console.WriteLine("End of Go2Light, because all " + ↩
       restarts_startingValue.ToString() + "restarts have ↩
       been used.");
    Console.WriteLine("Behaviour has finished. Press enter to↩
        perform next actions");
  }
}
```

Figure 1.8: Function *exceptionOccured_Restart* function

```
static void stopKof(object robotAndTime) {
  Console.WriteLine("stopKof was called.");
  RobotAndTime x = (RobotAndTime)robotAndTime;
  x.stopKof++;
  if (x.stopKof > 5) {
    if (!reconnect(x))
      return;
    else
      x.stopKof = 0;
  }
  double time = Stamp.Get() - x.StartTime;
  if (travelled(time, x.Speed) < (x.Cm + 0.05))
    x.E.Stop(stopOkf, stopKof, x, to);
  else {
    Console.WriteLine("We missed the destination spot, we ↩
       return back, try to repair the connection.");
    x.Cm = travelled(x.StartTime, x.Speed) - x.Cm;
    x.Speed = -x.Speed;
    x.StartTime = Stamp.Get();
    x.E.Motors(x.Speed, x.Speed, goOkf, goKof, x, to);
  }
}

class RobotAndTime {
  // Lot of parts omitted!!
  // The thread.Safe read and write are missing!!
  public double StartTime;
  double cm;
  double speed;
  public volatile int stopKof = 0;
  public volatile int goKof = 0;
}
```

Figure 1.9: *Kof* callback *stopKof*

user refuse to continue, no other function is invoked, and the behaviour is ended. Otherwise, the actions for going to the desired place are executed. If the robot does not drive away too far, the *Stop* function is called again. The *Stop* function calls *stopKof* function as its *Kof* callback. If the robot travels too long then it has to go in the opposite direction. The change of the direction is done in the second branch of the *if* command.

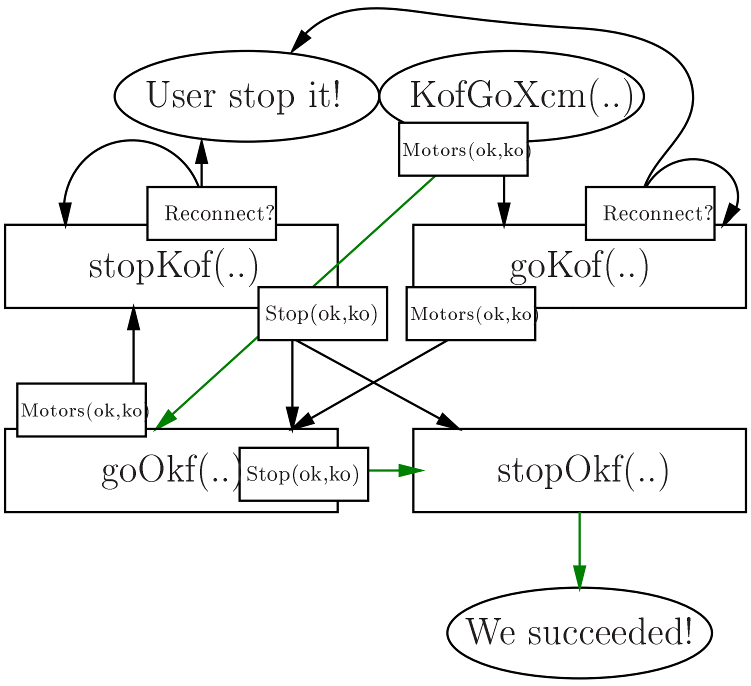

Figure 1.10: *KofGoXcm* basic *Epuck's* interface implementation

1.3.7 Logging of *Epuck* actions and image processing

In the file *TipsAndTricks.cs* there is located a simple example of image processing of a picture from e-Puck's camera. The example is invoked by the function *ShowProcessedImage* and it displays a window, where a black and white image is processed. The white is instead of red colour and black represents the other colours. A nice test of this function is to place the e-Puck's box with the e-Puck logo in front of the camera.

The function *LoggingExample* uses the function *ConsoleTestSensorsTimeout*, which has been presented at the beginning of this section in 1.2.2. It runs the function twice, each time with different timeout. The logging to file specified in *name* property is turned on. Try to run the function. Let us see the body of the function.

```
ada.LogStream = new StreamWriter(new FileStream(name, ↩
   FileMode.OpenOrCreate, FileAccess.Write));
ada.StartLogging();
for (i = 0; i < 2; ++i) {
  double to=1.0 / (i + 1);
  ada.WriteToLogStream("ConsoleTestSensors with timeout :"+to↩
     .ToString());
  ConsoleTestSensorsTimeout(ada, to);
}
ada.StopLogging();
```

Logging to a file is really simple. In needs only stream, where the log is written. After the start of logging all actions that send commands to e-Puck, performed on *ada* instance of *Epuck* are logged to file on path *name*. *WriteToLogStream* method inserts a commented line to the log file.

1.4 Elib tools (et)

Elib offers a standalone command line application, which parses the log from *Elib*. It is meant to be the base tool for a programmer, who wants to statistically analyse a log from *Epuck* class.

Et can be used only from the command line and parses the log row by row. It reads the log file, performs a chosen action and writes the result to the output file. If user does not specify the file it writes or read from, respectively, to command line. Elib tools supports three operation. Two of them process numeric data. The first operation counts an average of a selected column from a log file. The next operation sorts out rows, which have in specified column values from a given range. The last implemented operation separates rows, which have in specified column a value from a given collection of words. Elib tools application skips the rows beginning with '#', which is in *Elib* by default a comment in the log file. Let us look at the usage. The most important command is the second row, which invokes the help file. The help file introduces a complete usage of et command. Let us remark, that this invocation was performed in PowerShell on Windows. On Linux you need to add "mono" before every program, which was compiled for .Net or Mono.

```
>>et -s sourcelog.txt -d output.txt Avg 0
>>et -h
et (Elib Tools) HELP FILE
...here the help file continues
```

Design of *Elib Tools*

The purpose of *Elib Tools* is to be a simple tool for parsing a log file from *Elib*. The application is written in order to be as robust as possible. Wrong arguments do not throw any exception, but the application does no action.

The main contribution of *Elib Tools* application is its design, which can be easily extended. The application can be easily changed according to specific needs of a programmer. The programmer needs to modify *parseArguments* argument function. As you can see, the *parseArguments* function returns only patches of the source and the destination file and *Action* object.

```
Action action = parseArguments(args, out source, out destination);
```

Action object is an abstract class, which is introduced later on in this section. It provides an interface for performing operation to every row. If the new functionality is added, it could be implemented only by deriving a class from *Action* and by overriding two functions. Let us describe an example of Avg command. Lets look first at *Action* class.

```
abstract class Action {
  const string Comment = "#";
  protected int column;
  protected TextReader r;
  public TextReader R { get { return r; } set { r = value; } }
  protected TextWriter w;
  public TextWriter W { get { return w; } set { w = value; } }
  protected char[] sep;
  public char[] Separators { get { return sep; } set { sep = value; } }
  public Action(int Column) {
    column = Column;
  }
  public void DoAction() {
    string line = null;
    while ((line = r.ReadLine()) != null) {
      if (!line.StartsWith(Comment))
        LineAction(line);
    }
    LastAction();
  }
  protected virtual void LastAction() { /*usually does nothing and is called after all lines are processed*/}
  protected abstract void LineAction(string line);
}
```

Log from

Figure 1.11: *Action* abstract class for performing row operation

Now let us focus, on *Average* class, which after instantiation computes the average by calling a method *DoAction*() inherited from *Action* class.

Average class computes average step by step in order to avoid overflow of a variable in parsing long input files. The overflow of a variable is postponed by computing average in every step from the previous step, because the typical implementation by sum all the values and after then diving it with its amount leads to overflow of the sum much more quickly.

```
class Average : Action {
  long count;
  double avg;
  public Average(int Column)
  : base(Column) {
    count = 0;
    avg = 0;
  }
  protected override void LineAction(string line) {
    count++;
    avg *= (count - 1) / (double)count;
    try {
      avg += (Double.Parse((line.Split(sep)[column])) / count↩
          );
    } catch (FormatException e) {
      Console.WriteLine("Avg has to be done from Integer. ↩
          Error: " + e.Message);
      avg = double.NaN;
    }
    }
    protected override void LastAction() {
    w.WriteLine("{0:F8}", avg);
  }
}
```

Figure 1.12: *Average* class

1.5 Purpose of *Elib* and its properties

In this chapter we have presented *Elib* and the guidelines, which are convenient to follow using *Elib*. This section follows up to contents of the previous sections and introduces a view from, which was the *Elib* created.

Elib library was designed in order to help students of mobile robotics with controlling e-Puck by their programs. *Elib* extends possibilities of e-Puck processor, which can perform a very limited range of algorithms due too its low performance. Moreover *Elib* provides a user with numerous examples. The examples are commented and can be easily modified. On the other hand, a program, which use *Elib*, gives up of direct control over robot, which can be observed the best on *GoAndTurn* behaviour in section 1.3.4.

We have presented a few of simple behaviours, which control e-Puck, but much more can be done with *Elib*. *Elib* can be used easily for controlling more than one e-Puck at once by creating more sessions using different instances of *Epuck* class. Programs, which need more than 8 KB of memory on e-Puck such as genetic programs or neural networks, can easily control e-Puck over Bluetooth.

The real challenge is processing a picture from e-Puck, because the dsPic processor of e-Puck is not sufficient and also *Elib* needs to wait quite a lot of time for a picture. On the other hand, *Bull* behaviour successfully use the camera to grab the picture and also a graphical application *Elib Joystick* shoots the pictures with a camera.

Elib Joystick is a graphical application, which makes all sensors and actuators of e-Puck accessible in one window. It also supports capturing of an image, which is presented enlarged to the user. The graphical application runs in Single Thread Apartment on Windows, which means that the controls of the window can be accessed only from the main thread. In order to update the sensors *Elib Joystick* use either *EndFtion* or a *Dispatcher*. *Dispatcher* is a class, which allows access the controls from different thread. It is specific according the technology, which is used to run the graphical part of application. *Elib Joystick* uses Windows Presentation Foundation and uses a dispatcher to capturing image. The other operation, which has timeout 0.1 s, access the controls synchronously using either *EndFtion* or *EndGetFtion*. See section 1.3.1 for information about these functions. *Elib Joystick* is introduced in appendix **??**.

Elib can access all sensors of e-Puck. See the chapter **??** for more information about e-Puck's sensors and actuators. However the camera is not used in the full resolution, because the e-Puck processor has no place, where to store the captured image. The *Elib* captures only the amplitude of sounds from e-Pucks microphones. The frequency of a sound can be computed on dsPic processor using Fast Fourier Transformation, but the *BTCom*1.1.3, which is used by *Elib*, is not supported.

To conclude *Elib* offers almost all sensors of e-Puck in full quality. It also controls all of e-Puck's actuators. Furthermore, it offers much more comfort than a programming e-Puck's processor directly. Read section 1.4 about the advantages of remote control. Last but least, the *Elib* can be used from all .Net languages including *C#*, *VisualBasic*, *F#*, which is a functional language of .Net based

OCaml, or *IronPython*, which is a .Net implementation of Python. On Mono runtime, which runs both on Linux and Windows, the *C#* language can be used. The *Elib* is compatible with .Net 2.0[**?**] and higher and with Mono[**?**] 2.0 and higher.