THALAMUS FRAMEWORK v1.1

PROGRAMMING GUIDE

1. INTRODUCTION

Thalamus is a framework for creating modular interactive characters. Its goal is to make it possible to create interactive characters out of a set of different independent modules, each of which has a designated role in the behavior of the character. Unlike most the other interactive character frameworks, there is no specific module to act as the Mind, neither as the Body.

Instead, there can be, for example, a module that deals with text-to-speech, another dealing with recognition of humans using a Microsoft® Kinect®, another acting as a Dialogue Manager, and still other modules controlling the different parts of the character's body (virtual or robotic) or sensors (camera, speech recognition, etc.).

The main advantage of using Thalamus is that by creating these modules once, we can later re-use then with different characters in different scenarios that follow on the same requirements. It also makes it easier to share modules with other colleagues so that all our work can be useful to other people even if they did not use the exact same character, or the exact same scenario, dialogues and interactive goals.

2. THE THALAMUS CHARACTER

A **Character** in Thalamus is a runtime environment in which Thalamus **Modules** (or ***Clients***) reside and interact. The collection of **Modules** that compose the **Character** defines how the **Character** behaves. We can think of a **Character** as a closed environment, as all the messages that travel between **Modules** are internal to the **Character**.

However, a Thalamus **Character** is made to interact with an external environment. In order to achieve this, some of the **Modules** should have some means of communicating with the external environment, and translating all the information into a **Thalamus** message (***Fig. 1***).

Although we think of a Thalamus **Character** as being just an abstract, open-space, in fact, each **Character** is managed by a master node, which we call a **Thalamus Master**.

A Thalamus **Character** is created in the **ThalamusStandalone** application. The **Character** automatically creates a **Thalamus Master**, which will be in standby for **Modules** to connect with it. By default, whenever a **Module** is created, it automatically searches for a **Master** in its own sub-network (via an UDP Broadcasting mechanism) and connects to it (***Fig. 2***).

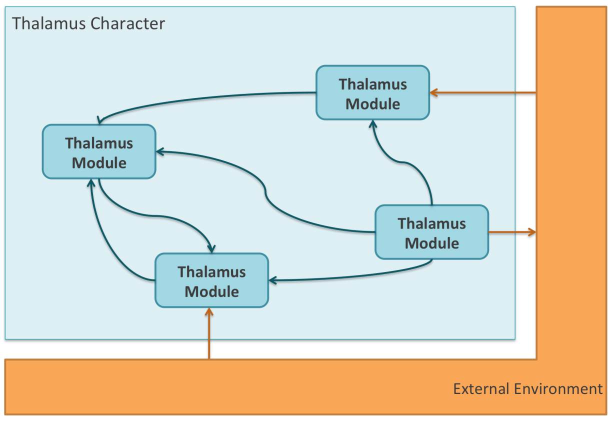


Figure 1. Some Modules receive or send information from the external environment (orange arrows) and then transmit them to other Modules as Thalamus Messages (blue arrows).

All the messages that travel between **Modules** thus actually go through the **Master**. That is important in order to guarantee, in the future, Message/Parameter remapping, or conflict resolution.

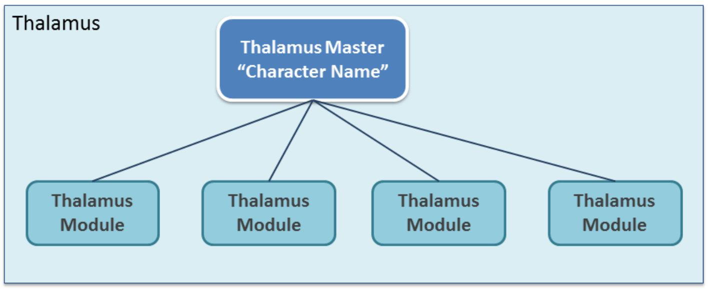


Figure 2. Thalamus Modules automatically search for and connect to the Thalamus Master (which represents the Character). This connection process is completely transparent to the Module programmer.

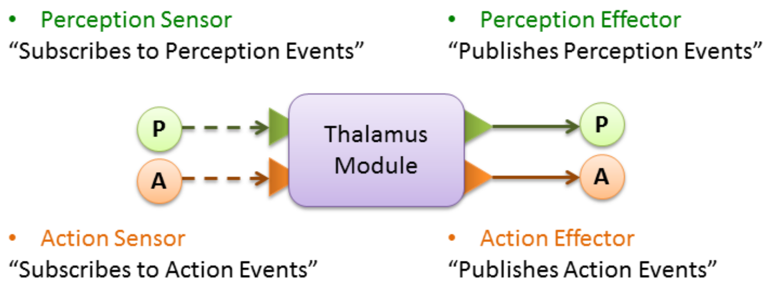
2.1. Thalamus Modules (*aka* Clients)

A **Module** in Thalamus is a program or a piece of a program that we can think of having **Sensors** and **Effectors** for Thalamus **Messages**. Technically it follows a Publication/Subscription mechanism, so a **Sensor** actually corresponds to the subscription (and reception) of a set of **Messages**, while an **Effector** corresponds to the announcement (and publication) of a set of **Messages**. It is important to distinguish these four concepts:

Announcement When the *Module* connects to the *Master*, it announces the *Messages* (both *Perceptions* and *Actions*) that it will be later publishing.
Subscription When the *Module* connects, it subscribes to certain *Messages* that it should later receive.
Publication At any point of its execution, the *Module* can publish any of the *Messages* it announced, so that other *Modules* who subscribed to the same *Message* will receive it.
Reception Whenever another *Module* publishes a *Message* we subscribed to, we will receive it.

A Thalamus **Message** can be a **Perception** or an **Action**. **Perceptions** are meant to be used as a notification of occurrence of an event, and an **Action** is meant to be the request for a module to do something (***Fig. 3***).

A **Module** can both subscribe to **Perceptions** and **Actions**, and publish **Perceptions** and **Actions**.



• Perception Sensor
"Subscribes to Perception Events"
• Perception Effector
"Publishes Perception Events"
P
A
Thalamus Module
P
A
• Action Sensor
"Subscribes to Action Events"
• Action Effector
"Publishes Action Events"

Figure 3.

2.2. MESSAGES

In order to have different **Modules** to communicate, it is necessary to establish some kind of **protocol**. The messaging protocol in Thalamus is very flexible, meaning that at any point, any **Module** can decide to create a new **Message** type, and all the other **Modules** will know about it.

However, on a collaborative development of a **Character** in which different people develop different **Modules**, it is better to have a well-defined set of **Messages**.

In order to support that, a Thalamus **Module** can specify the **Messages** it **Publishes** and **Subscribes** to by inheriting from specified **Interfaces**. These **Message Interfaces** contain one or more **Message types**, which are simply defined by a method header. The parameters of the method are considered to be the message parameters.

Looking at ***Fig. 4***, we can see three M**essage Interfaces** defined within a namespace called ***ExampleMessages***. Each of it defines a collection of messages. Taking as example ***ISomethingEvents***, it inherits from ***Thalamus.IPerception***, meaning that it defines perception messages. Subscribing or announcing this interface implies that the module will be either receiving or publishing messages of the types defined in this interface: ***SomePerceptionA*** and ***SomePerceptionB***. The interface **ISomethingActions** defines a set of actions, as it inherits from ***Thalamus.IAction***. In terms of methodology, Actions and Perceptions work exactly the same. We follow the same steps to subscribe and receive either, and also follow the same steps to publish any of them. They are kept as two different things in accordance to the theoretical **Censys Model**[1], which Thalamus is based on.

[1] *Censys: A Model for Distributed Embodied Cognition*. Ribeiro et al., 2013. http://gaips.inesc-id.pt/gaips/component/gaips/publications/showPublication/8/336

```
namespace ExampleMessages {
        public interface ISomethingEvents : Thalamus.IPerception
        {
                void SomePerceptionA();
                void SomePerceptionB(int someParameter);
        }

        public interface ISomethingActions : Thalamus.IAction
        {
                void DoSomeAction();
                void DoSomeOtherAction(int someParameter, int anotherParameter);
        }

        public interface IOtherEvents : Thalamus.IPerception
        {
                void OtherMessageA();
        }
}
```

Figure 4.

2.3. Creating a Module

Creating a new **Module** is very simple. You just need to create a new class in C# that references the **Thalamus** assembly, and make your class inherit from the ***Thalamus.ThalamusClient*** class as shown in ***Figure 5***.

The constructor in the base class receives one mandatory parameter, which is the name by which your **Module** should identify itself in Thalamus. This name is descriptive only, so there will be no problem in having several **Modules** using the same name – it will only be more difficult to use and debug the **Character**. The name of the **Module** generally follows up on the name of the class you just defined, but you can also make it different if you feel like it.

The second parameter to the constructor is optional. It is a ***boolean*** parameter called ***autoStart***. By default this parameter is ***True***, meaning that your **Module** will automatically connect to Thalamus once the class is instantiated. If for some reason you don't want that, just set this parameter to ***False***, and then call the ***Start()*** method when you want it to connect.

If you create a **Character** in **ThalamusStandalone** and then instantiate this class in an application, it will already connect with Thalamus. However, it does not subscribe to or publish anything, so right now it is useless.

```
…
using Thalamus;

namespace ExampleModule {
        public class ExampleModuleClient : ThalamusClient
        {
                public ExampleModuleClient() : base("ExampleModule")
                { }
        }
}
```

Figure 5.

2.4. Subscribing to and Receiving Messages

In order to subscribe to **Messages**, we must create an empty interface ***IExampleModule*** (***Figure 6***) that contains all the **Message Interfaces** that we want to subscribe to. In this case, we are going to use the **Messages** we previously defined in ***Figure 4***. We then make our class inherit from this interface. This will make the **Module** automatically subscribe to all the messages defined within our interface.

We then need to implement a method to handle each of the messages, or else our code won't even compile. You can write all of the methods by hand, but it is easier to let your IDE do that. Either with Microsoft Visual Studio® or with MonoDevelop®/Xamarin Studio®, you just need to right-click on ***IExampleModule*** in your class definition (not the definition of the interface), and in the ***Refactor*** menu select ***Implement Interface Explicitely***. It is important to keep the implementation explicit, so that the methods will contain the namespaces and interfaces to which they belong.

Whenever our **Module** receives a **Message**, the corresponding **Method** will be called, so you should place all the code you need in order to handle the received message inside those methods.

```
…
using Thalamus;
using ExampleMessages;

namespace ExampleModule {

        public interface IExampleModule :
                                        ISomethingEvents,
                                        IOtherEvents
        {}

        public class ExampleModuleClient : ThalamusClient,
                                                  IExampleModule
        {
                public ExampleModuleClient() : base("ExampleModule")
                { }

                void ExampleMessages.ISomethingEvents.SomePerceptionA()
                {
                        //code
                }
                void ExampleMessages.ISomethingEvents.SomePerceptionB(int someParameter)
                {
                        //code
                }
                void ExampleMessages.IOtherEvents.OtherMessageA()
                {
                        //code
                }
        }
}
```

Figure 6. Red marks new code that was added to the code from Figure 5.

2.5. Announcing Messages

In order to publish **Messages**, we must create another interface ***IExampleModulePublisher*** (***Figure 7***) that contains all the **Message Interfaces** that we will be publishing. In this case, we are also going to use the **Messages** we previously defined in ***Figure 4***. This interface must also inherit from the ***Thalamus.IThalamusPublisher*** in order to be able to be used as a Publisher interface.

Now this time, the interface is going to be used to tell the **Module** how create a Publisher class. The actual Publisher class is created in runtime from our interface. The interface is assigned to our Module in the code added into our Module's constructor.

```
…
using Thalamus;
using ExampleMessages;

namespace ExampleModule {

  public interface IExampleModule :
                                ISomethingEvents,
                                IOtherEvents
  {}

  public interface IExampleModulePublisher :
                                IThalamusPublisher,
                                ISomethingActions
  {}

  public class ExampleModuleClient : ThalamusClient,
                                        IExampleModule
    {
      public ExampleModuleClient() : base("ExampleModule")
      {
        SetPublisher<IExampleModulePublisher>();
      }

      // …
      // same as Figure 6.
      // …
    }
}
```

Figure 7. Red marks new code that was added to the code from Figure 6.

This code adds the ability to publish messages using an object defined in ***ThalamusClient*** that is called **Publisher**. This is a dynamic object[2]. As such, it does not define a set of methods, but instead, accepts calling any method on it during compilation, and only in runtime does it check to see if it actually possesses that method. The only problem about it is that it cannot use IntelliSense for autocomplete, neither verifies syntax as during compile time, the compiler does not know what methods this object might contain in runtime.

A workaround for this is to create a little Publisher class ourselves that encapsulate the calls to the published messages. This way, by using our specific publisher instead of the dynamic one, we can get syntax checking and auto-complete for the publisher while programming.

Our actual publisher class will thus be called ***ExampleModulePublisher***, and will inherit from our publisher interface ***IExampleModulePublisher***. It will keep a reference to the Module's dynamic Publisher, and define methods for all the messages that we will be publishing. Again, instead of writing all the methods, we can just right-click ***IExampleModulePublisher*** and select ***Refactor → Implement Interface*** to have the IDE write the methods automatically. This time we can have an implicit implementation instead of an explicit one.

Now the only thing that each method should do is to call itself on the dynamic publisher object. The whole thing can be seen in ***Figure 8***.

[2] http://msdn.microsoft.com/en-us/library/system.dynamic.dynamicobject.aspx

We defined the ***ExampleModulePublisher*** as a private class inside the ***ExampleModuleClient*** class. However, it can also be defined anywhere else, even in a different file.

```
using Thalamus;
using ExampleMessages;

namespace ExampleModule {

  public interface IExampleModule :
                                 ISomethingEvents,
                                 IOtherEvents
  {}

  public interface IExampleModulePublisher :
                                 IThalamusPublisher,
                                 ISomethingActions
  {}

  public class ExampleModuleClient : ThalamusClient,
                                                IExampleModule
  {
    private class ExampleModulePublisher : IExampleModulePublisher
    {
      dynamic publisher;
      public ExampleModulePublisher(dynamic publisher) {
        this.publisher = publisher;
      }
      public void DoSomeAction() {
        publisher.DoSomeAction();
      }
      public void DoSomeOtherAction(int someParameter, int anotherParameter) {
        publisher.DoSomeOtherAction(someParameter, anotherParameter);
      }
    }

    ExampleModulePublisher examplePublisher;
    public ExampleModuleClient() : base("ExampleModule") {
      SetPublisher<IExampleModulePublisher>();
      examplePublisher = new ExampleModulePublisher(Publisher);
}

    void ExampleMessages.ISomethingEvents.SomePerceptionA() {
        //code
    }
    void ExampleMessages.ISomethingEvents.SomePerceptionB(int someParameter) {
        //code
    }
    void ExampleMessages.IOtherEvents.OtherMessageA() {
        //code
    }
  }
}
```

Figure 8. Red marks new code that was added to the code from Figure 7.

2.1. Publishing Messages

Finally, after we defined our publisher, whenever our Module connects it can announce all the Messages it will be publishing. Now publishing messages is very easy. Because we created the ***ExampleModulePublisher*** class and assigned it to the ***examplePublisher***, we just need to call the message we cant to publish in the ***examplePublisher*** object anywhere in our code:

```
…
    void ExampleMessages.ISomethingEvents.SomePerceptionB(int someParameter) {
      examplePublisher.DoSomeOtherAction(42, someParameter);
    }
…
```