DevOps Model RealTime External C++ Integration

Author: Anders Ek
HCL

This document describes how to integrate code generated using DevOps Model RealTime with, from Model RealTime point of view, external code.

The document was last updated for Model RealTime 11.2.

Introduction

In many situations it is convenient to integrate the state machine oriented code generated from Model RealTime with C++ code produced using other tools, hand coded components or 3rd party libraries. In this document we will describe the different aspects and options available.

From an integration point of view there are two fairly orthogonal aspects:

- Build/file level integration
- Functional code level integration

The build/file level integration is concerned with how you can organize the files and projects and how you can integrate the code generation, linking and compilation of these files and projects.

The functional code level integration deals with the C++ details on how code in Model RealTime generated components can invoke and be invoked by code in external components.

We will in this document cover both aspects.

Build/File Integration

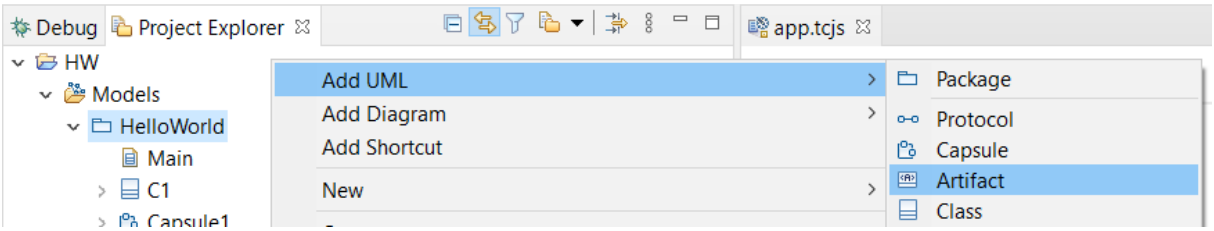
From an overall point of view Model RealTime provides three recommended schemes to manage handwritten C++ code together with Model RealTime generated C++ code:

- File artifacts
- External CDT projects
- External components

File Artifacts

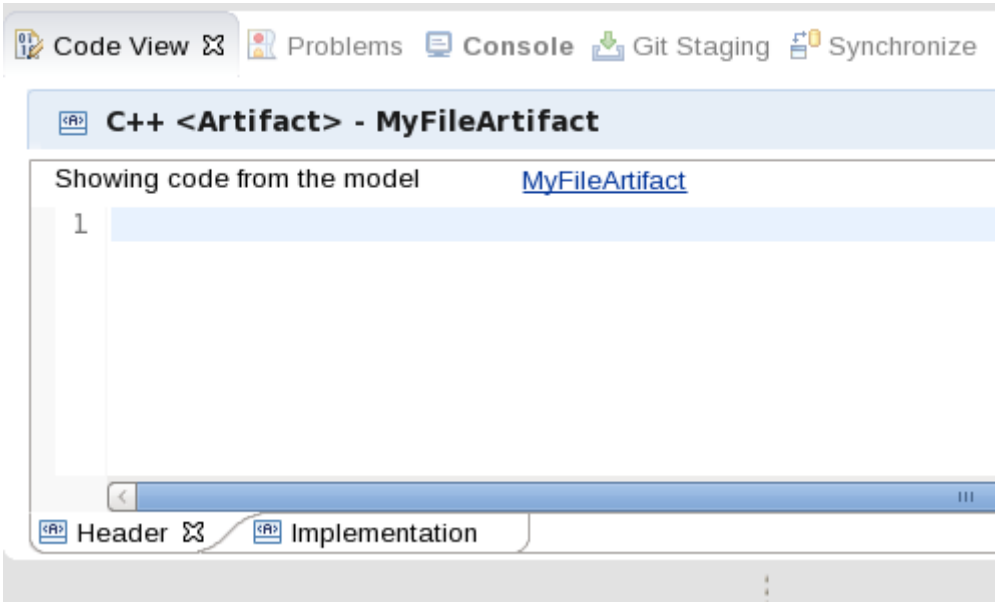
File artifacts are intended to solve the problem of how to handle C++ code that for some reason does not make sense to describe using classes or capsules, but that also is not large enough to make sense to handle as separate projects. So, it provides an easy-to-use mechanism to write some C++ code in an Model RealTime model project. The code will be stored inside the model, just as any other code snippet that you edit with the Code view or Code editor.

You create a file artifact as any other model element in Model RealTime e.g. using the **Add UML** context menu command in the Project Explorer.



When generating code for the project the file artifact will lead to one implementation and one header file similar to e.g. regular classes or capsules. You can edit the contents of these files either in the model, using the code view or code editor, or directly in the generated files and then synchronize back your changes to the model.

If you use the code view you will have both the header and implementation available as two tabs in the view.



It is worth noting that if using file artifacts, then strictly speaking the code is not external to the Model RealTime model project. It will be part of the model project itself and the files will be generated together with all other generated code.

External CDT Projects

The integration with general CDT projects is intended to cover the situation where you have a substantial part of your code base in plain C++ code instead of as part of the Model RealTime model. One example of when this makes sense is if you develop an application that in addition to the real-time Model RealTime part also contains a GUI component developed using a GUI framework.

The general idea behind this integration is to use the external CDT project to create a library that is integrated into the Model RealTime model using external library transformation configurations. The external library transformation configuration makes it possible to compile and link the CDT project as part of the Model RealTime build process.

See the section “External Libraries” in the document ”Building C++ Applications with Model RealTime” for details on how to create and configure an external library transformation configuration.

External Components

In most situations your application will not only contain the code you have written yourself, but also 3rd party components that are maintained by someone else and where you even may get the components as pre-compiled libraries and header files.

There are two ways to integrate such a component with your Model RealTime model project:

- Using an external library transformation configuration
- Directly specify the library in the transformation configuration you are using to build the Model RealTime project

The choice is mainly a matter of personal taste. Using an external library transformation configuration is slightly more heavy-weight, but will give model level visibility to the library and the details of the integration with the 3rd party library is captured in one location in the model, making it easier to maintain.

Directly including the header file information in relevant model element include sections and the linking information in your transformation configuration is simple and quick, but less visible to readers of your model and, if the library is used in many places, also slightly more difficult to maintain.

The details of how to use an external library transformation configuration for this scenario is described in the section ”Precompiled Libraries” in the ”Building C++ Applications with Model RealTime” document.

If you want to use the direct inclusion method you will need to

- modify the ”User Libraries” field of the ”Target Configuration” tab of your transformation configuration to include the precompiled library in the linking phase of the build;
- specify the path to where the include files of the component are found in the ”Inclusion paths” field of the ”Target Configuration” tab, and
- specify the inclusion of the header file of the component in relevant code snippets. A common way is to include the library header file in the ”Header Preface” or “Implementation Preface” code snippets for the capsule, class or other entity that is using the library.

Functional Code Level Integration

The functional code level integration is simply the way to transfer data and/or control between the external C++ code and the Model RealTime generated code. This is useful to decompose into two common sub cases:

- Simple non-threaded API calls
- Thread communication

Simple Non-Threaded API Calls

A very common situation is where there is a need to use utility functions / data types defined in external C++ code from Model RealTime generated code. The external code may e.g. de-

fine data types that are used as the type of capsule/class attributes and the external functions are called from transitions or operations in the Model RealTime model.

As long as the external data types are used locally in one Model RealTime capsule this does not cause any issues and external data types can be used just as if they were defined in the model.

Thread Communication

A more complex situation is when the external code defines run-time threads that will run in parallel with the Model RealTime defined threads. In this situation care must be taken to make sure the data transfer between the external threads and the Model RealTime spawned threads are safe.
From an overall point of view any C++ style thread communication mechanism can be used to accomplish this, however due to how the Model RealTime target run-time system works some features are worthwhile to be aware of.

A common problem in this context is when code in an external thread needs to trigger a transition in an Model RealTime capsule and as part of this transfer some data.

Two features of the Model RealTime run time system are useful in this respect:

- External ports
- Proxy capsules

External ports provide a simple mechanism to trigger a transition in a model from an external thread.

Proxy capsules are capsules that expose a functional API with thread safe functions to trigger events in the Model RealTime model by sending messages to internal ports.

Two variants of proxy capsules can be distinguished:

- A wrapper capsule that wraps an entire top level Model RealTime capsule including all its ports in a functional API
- Port-level proxy capsules that can be inserted anywhere there is a need to expose a port in the model to external C++ threads.

External Ports

An external port is simply a port of an Model RealTime capsule that is typed by the External protocol. This protocol has two interesting operations:

- raise()
- enable()

They are intended to be used the following way:

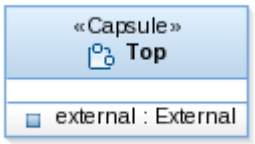
- A capsule that will interface with an external thread should have a port typed by the External protocol.

- At an appropriate point in time, the capsule gives the external thread access to its external port. When the capsule is ready to receive a message from the external thread it calls the enable() function on the port.
- Whenever the external code wants to send an event to the capsule it can invoke the raise() function on the port. The external code must be prepared for the situation that the call will fail due to that the capsule is not ready at that particular point in time to receive the event. If that happens it can choose to wait a little and then try again.
- In the capsule a transition with External::event as trigger will be triggered when the call to raise() succeeds. The external port should be re-enabled (typically before the triggered transition exits) to allow it to be triggered again.

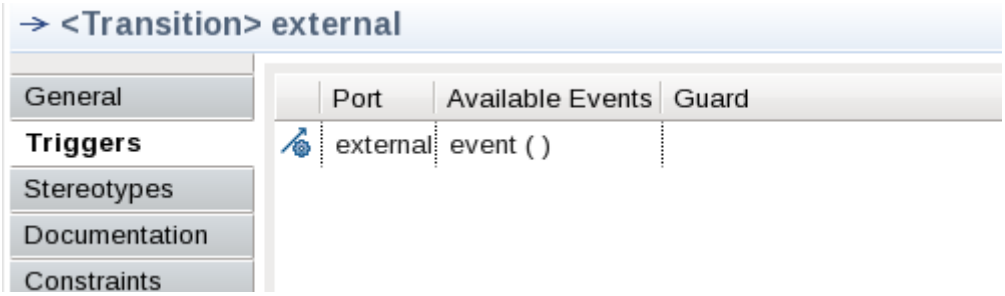
Its important to note that the external port needs to be enabled by the capsule before it can be used to trigger a transition. Also note that when the raise() function is called on an external port it will automatically be disabled. So the transition that is triggered must enable it again before another raise call can succeed. This scheme ensures that it is the capsule that fully decides when it is ready to accept an external event, and external code can never force it to do so. For example, the capsule may want to give priority to some internal events, and only handle external events at times it finds appropriate.

As an example assume that we have an external thread that needs to trigger a transition in an Model RealTime capsule.

The capsule to be triggered by the external code should have a port typed by External.



The capsule should also have transitions defined that are triggered by ”event” from External:



The port is enabled, for example by placing the following code in the initial transition of the capsule:

```
external.enable();
```

Assuming that ”port” is a reference to the port on the capsule, then the following code external to the Model RealTime model will trigger the transition:

```
if (port->raise()) {
	// The transition will be triggered
} else {
	// The port was not enabled
}
```

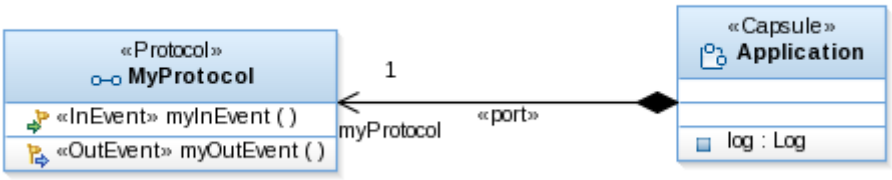
It is possible to pass a data object with the external event. This works in the same way as when you pass data with a timer event. Specify the data and its type descriptor as arguments to the raise function. For example:

```
int i = 42;
port.raise(&i, &RTType_int);
```

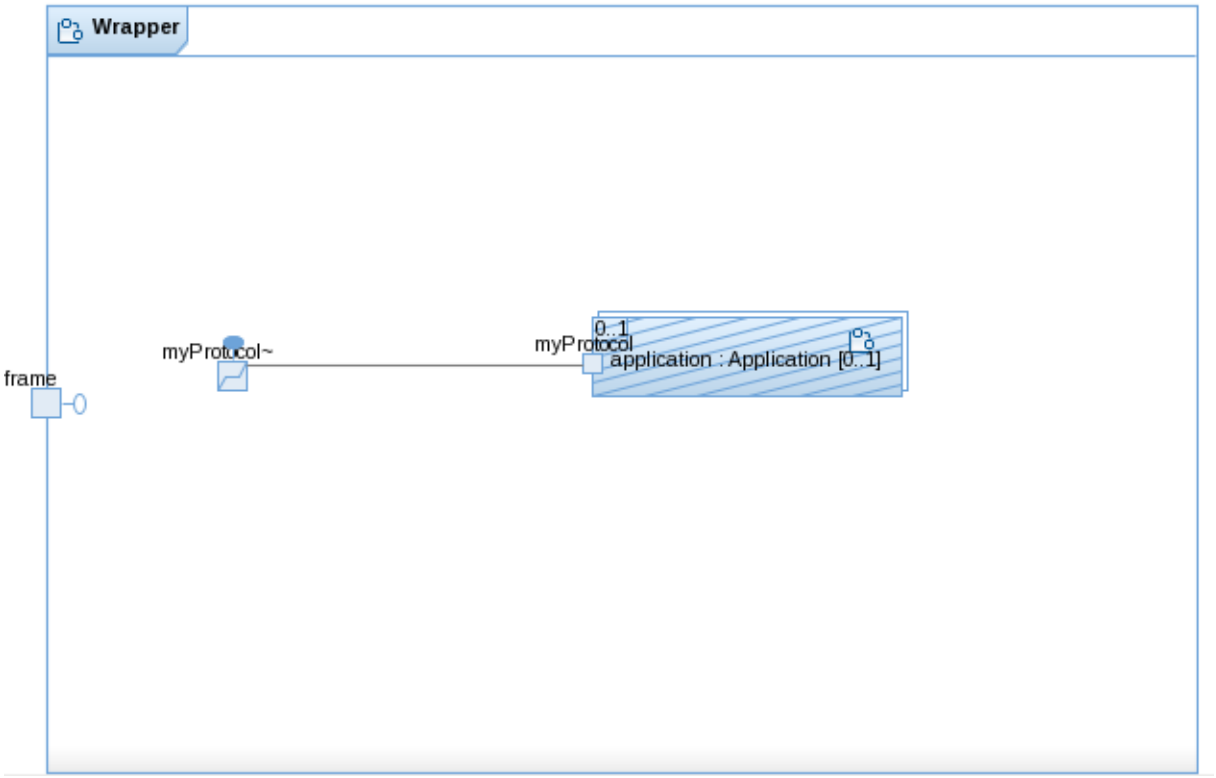
External ports also provide a data area where the external code can store any number of data objects. The target run-time system ensures thread-safe access to this data area. Using this data area is more efficient if a large number of data objects need to be communicated from the external thread to the capsule. Read more about this in the section ”External Port Service” in the ”RT Services Library” document.

Wrapper Capsules

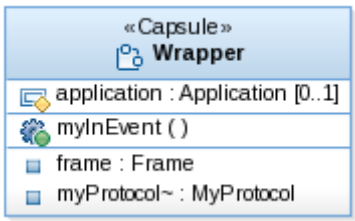
As an example consider a model with a capsule ”Application” that implements a protocol ”MyProtocol” via a port ”myProtocol”.



If we want to expose the ”myProtocol” port using the wrapper capsule strategy we can create a wrapper capsule ”Wrapper” that contains an instance of the ”Application” capsule and that has ports connecting to the visible ports of the ”Application” capsule. In our case it is only the myProtocol port but it can easily be extended to any number of ports.



In addition, the ”Wrapper” capsule also has one operation for each event to be sent to the ”Application” capsule. In our case only the myInEvent operation.



These operations are called from external C++ code to send the corresponding event. Assuming ”capsule” is a pointer to the Wrapper capsule instance the following code will send the event.

```
capsule->myInEvent();
```

The implementation of the myInEvent operation is very simple. It simply sends the corresponding message to the relevant port. In our example:

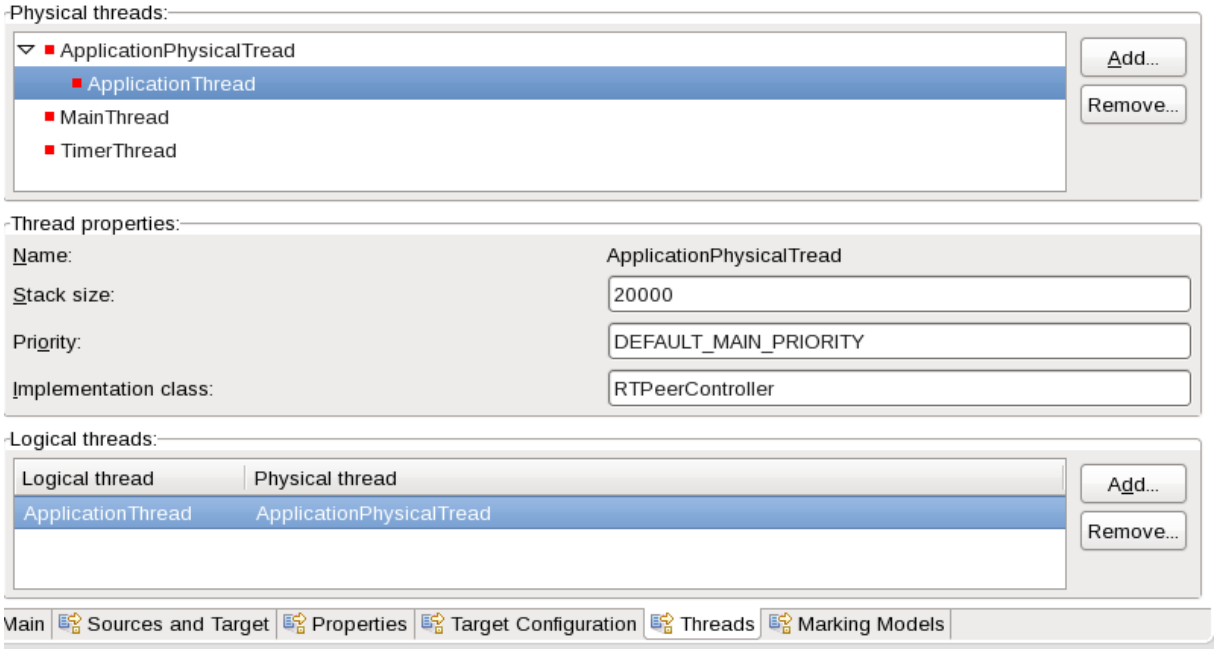
```
myProtocol.myInEvent().send();
```

To make sure that the communication is thread-safe the Wrapper capsule needs to execute in a separate thread. This is accomplished by:

- defining a separate thread for the ”Application” capsule instance, and

- instantiating the ”Application” capsule in the thread when creating the ”Application” instance.

The thread definition is done in the transformation configuration, in the ”Threads” tab.



As you can see we have created one physical thread ”ApplicationPhysicalThread” that will correspond to a real thread in run time and one logical thread ”ApplicationThread” that defines a name to use in code to reference the thread. See the section ”Threads Tab” in the ”Building C++ Applications with Model RealTime” for more information on how to define threads.

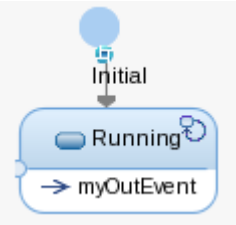
To make the ”Application” instance run in the designated thread we need to create it dynamically and in the incarnation statement define the thread to be used. This creation can easily be done in the initial transition of the ”Wrapper” capsule.

```
RTActorId capsule_id;
capsule_id = frame.incarnate(
             application,
             Application,
             (const void *) 0, // initialization data
             (const RTObject_class *) 0, // type descriptor
             ApplicationThread, // logical thread
             0 // replication index
);
if( ! capsule_id.isValid() ) {
      context()->perror("Incarnation of Application failed: ");
      context()->abort();
}
```

See the section ”Frame Service” in the ”RT Services Library” for more information about creation of capsule instances.

To handle outgoing messages we can create one transition for each outgoing event in the state machine of the Wrapper capsule. In the code for the transition we can use whatever is necessary to trigger an action in the external threads.

In our example we only have one outgoing event, so we will only have one transition.

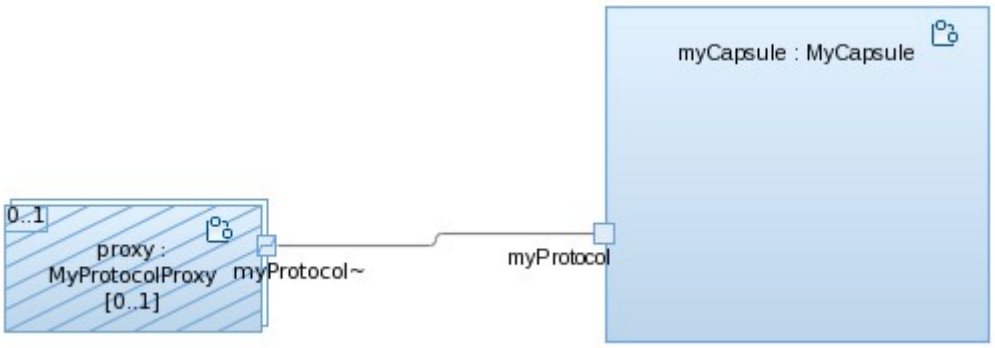


Port Level Proxy Capsules

Port level proxy capsules is an alternative to wrapper capsules that is particularly useful if the ports to be exposed are not on the top level of the Model RealTime capsule hierarchy. The idea is to create proxy capsules that handle specific protocols and incarnate them whenever there is a need to access a port from external C++ threads.

If applying this to the very simple example from the previous section we would now have a proxy capsule handling the ”MyProtocol” protocol.

Whenever we now have the need to expose a port typed by MyProtocol we can instantiate a ”MyProtocolProxy” capsule.



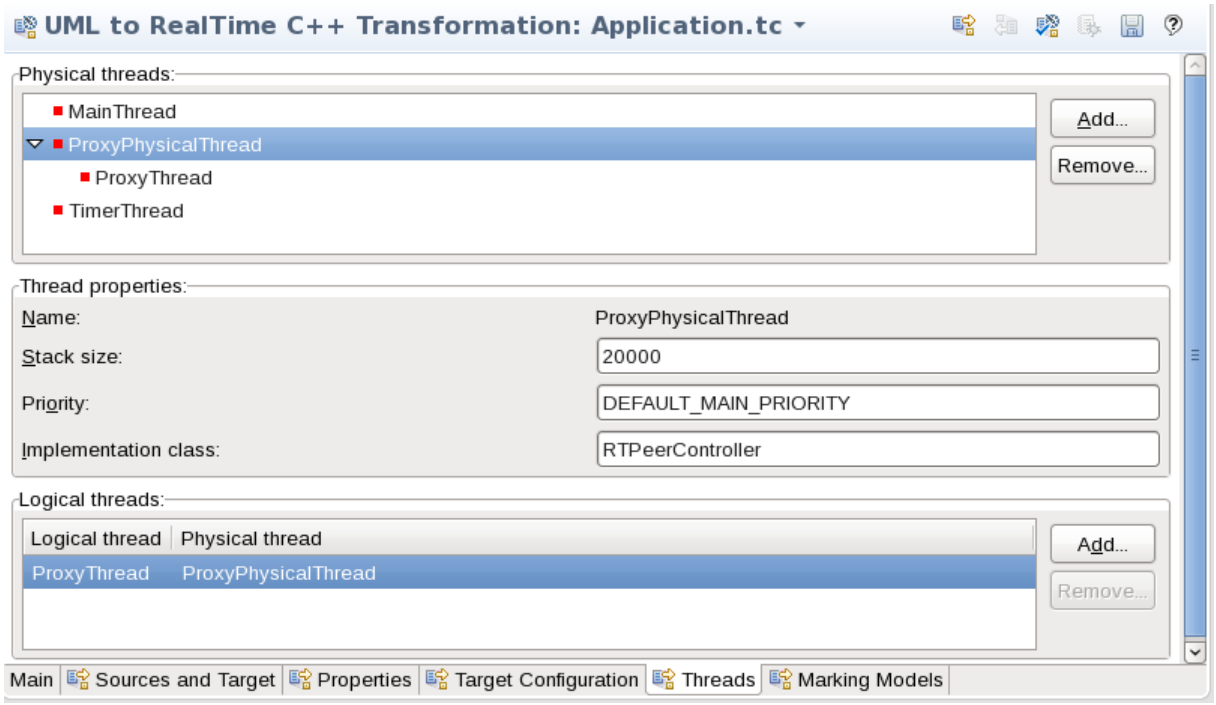
Code in external threads can now (provided it has a pointer to the proxy capsule) send messages to the port similar to how it is done in the Wrapper capsule example above.

```
proxy->myInEvent();
```

The implementation of the operations are also done similar to the Wrapper capsule example above. In our example:

```
myProtocol.myInEvent().send();
```

One difference compared to the Wrapper example is that the application itself would now most likely run in the Main thread, and instead we would create a separate thread where all proxy capsules would execute.



So, the code to create a proxy capsule could for example be the following:

```
RTActorId proxy_id;
proxy_id = frame.incarnate(
             proxy,
             MyProtocolProxy,
             (const void *) 0, // initialization data
             (const RTObject_class *) 0, // type descriptor
             ProxyThread, // logical thread
             0 // replication index
);
if( ! proxy_id.isValid() ) {
       context()->perror("Incarnation of MyProtocolProxy failed: ");
       context()->abort();
}
```

We can handle outgoing transitions using the same method as for the Wrapper example. We simply create a transitions in the state machines of the proxy capsules to handle the events.