PRAGMADEV STUDIO

TUTORIAL

PRAGMADEV
modeling and testing tools

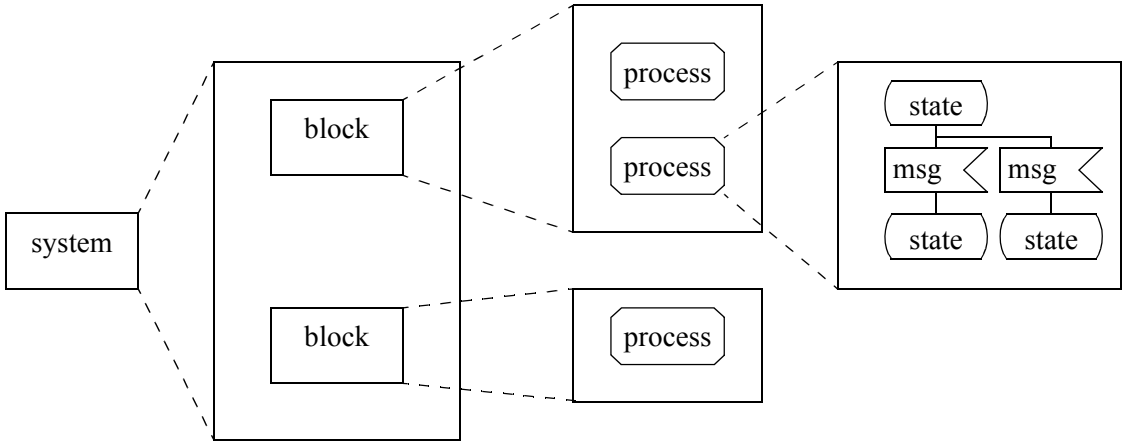
1 - Introduction

Before starting this tutorial, it is important to understand the basic concepts used in PragmaDev Studio.These concepts derive from the two languages supported by PragmaDev Studio, **SDL** and **SDL-RT**:

- SDL stands for **Specification and Description Language**. SDL is a graphical, object-oriented, formal language defined by the International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) as recommendation Z100. The language is intended for the specification of complex, event-driven, real-time and interactive applications involving many concurrent activities that communicate using discrete signals.
- SDL-RT stands for **Specification and Description Language - Real Time**. It is a mix of SDL with another graphical language, UML, and of a textual language, C. It retains the graphical abstraction brought by SDL while keeping the precision of traditional techniques in real-time and embedded software development and making simpler the re-use of legacy code by using natively the C language. The object-orientation is also pushed a step further by using the UML diagrams.

The underlying concepts of both languages are the same: the overall application to develop is called the **system**. Anything that is outside the system is called the **environment**. The system itself is described via four complementary and consistent views:

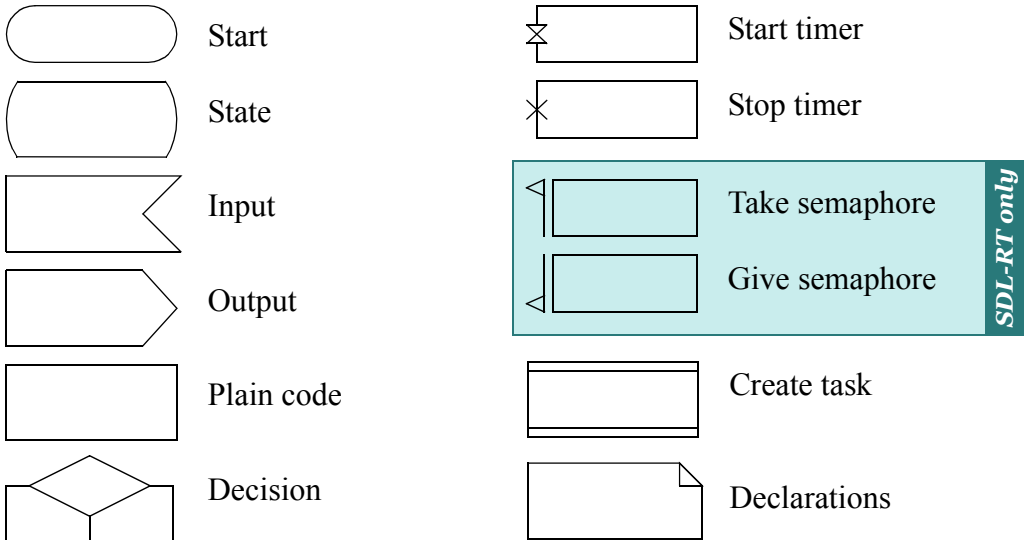
- *Architecture*
  A system can be decomposed in functional **blocks**. A block can be further decomposed in sub-blocks and so on until the functionality of the final blocks are simple enough. A block then fulfils its functionality with one or more **processes**, communicating with each other via **messages** (also called **signals**). A process is basically a task and has an implicit message queue to receive messages from other tasks. There is no need to define it. A block has no direct implementation in the final application; it is a matter of organizing and structuring the application. Blocks and process are called **agents**.



- *Communication*

Agents exchange messages through **channels**. Messages going through channels are listed to define the interface between the agents. When it comes to final code on the target, channels have no direct implementation; they are only used for structuring the software and defining the interfaces.

- *Behavior*
  A process behavior is described graphically as a finite state machine. Internal process states, events (messages), decisions, timer manipulations, semaphore manipulations have specific symbols briefly summarized below necessary to understand the following tutorial:



  SDL or SDL-RT procedures can be called within the process behavior description. In SDL-RT, C functions can be called as well; SDL also allows the call of C function via external operators or procedures.

- *Data and syntax*
  This is where SDL and SDL-RT differ the most:
  - In SDL, data is defined via **ADT** (**Abstract Data Types**), using specific concepts and notations. The data manipulation has also a specific syntax, derived from languages such as Pascal.
  - In SDL-RT, the C language is used to define and manipulate data, making things more familiar to developers.

Another SDL-RT specificity is the integration of UML use case and class diagrams for less time-critical parts of the system. Objects can be associated to processes or blocks and used in the behavioral parts of the processes.

In both SDL and SDL-RT models, PragmaDev Studio also integrates the Message Sequence Chart dynamic view. On such a diagram, time flows from top to bottom. Lifelines represent agents, semaphores or objects and key events are represented. The diagram emphasizes the sequence in which the events occur.

The two languages have each their specific domain of usage:

- SDL will be mainly used during the specification phase. Being formal, it also allows more possibilities of verification and testing.

- SDL-RT will be mainly used in the development phase, since it is closer to the hardware on which the software will eventually run.

In addition to the full features PragmaDev Studio, there are also two variants of the tool that focus on each of this languages: **PragmaDev Specifier** for SDL modelling, and **PragmaDev Developer** for SDL-RT modeling.

Would you need any extra information on the diagrams and their meaning, the following references may be used:

- For SDL, the SDL Forum web-site has many tutorials and presentations: *http://www.sdl-forum.org/*
- For SDL-RT, the reference manual is available in PragmaDev Studio via the *Help / SDL-RT reference* menu. This manual is also available on the SDL-RT web-site: *http://www.sdl-rt.org*

2 - A simple system

The system we have chosen is simple enough to be written from scratch but rich enough to pinpoint the basics of SDL-RT and SDL. It is a very basic phone system composed of a central and of several phones. When the phones are created, the central gives them an automatically computed phone number. When a user takes a phone to call another one, the phone asks the central the id of the phone to be called identified by its phone number. The caller sends directly a call request to the distant phone. For simplicity sake the distant phone automatically answers.

This tutorial is divided into two parts, depending on the tool you will be using, or on the modelling language if you're using PragmaDev Studio:

- The tutorial for PragmaDev Specifier describes SDL modelling and starts on page 7.
- The tutorial for PragmaDev Developer describes SDL-RT modelling and starts on page 62.

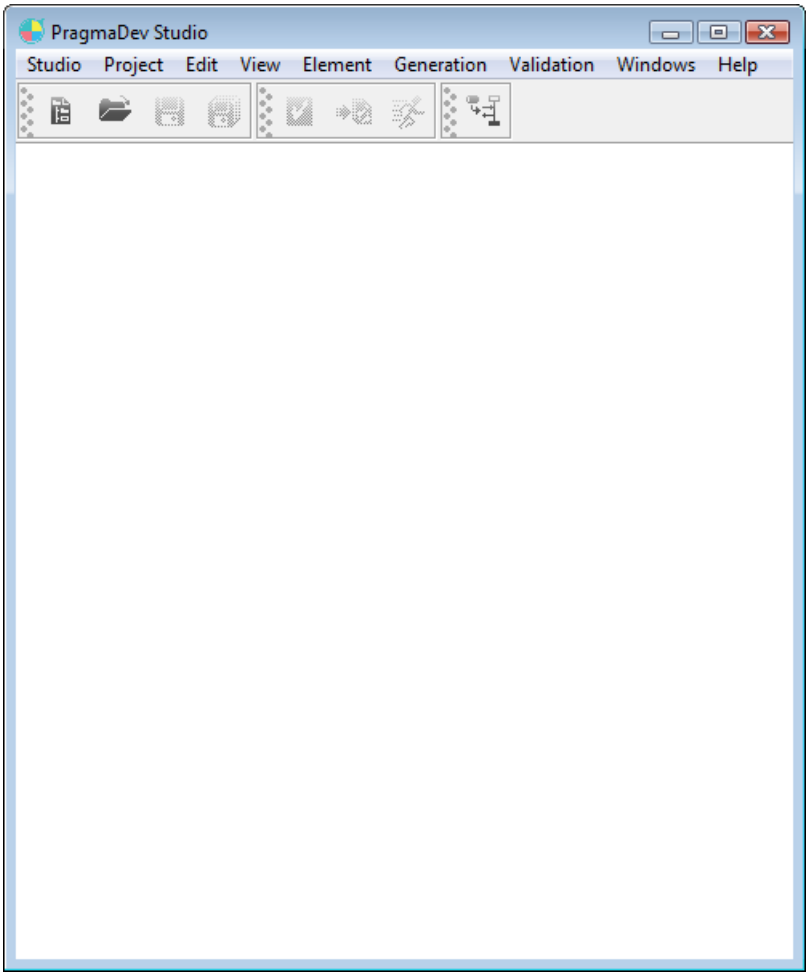
The last part of this tutorial describes the automatic documentation generation (page 112). It is based on the SDL system but that can be applied to the SDL-RT example, and therefore will work in both variants of the tool.

If you do not want to design the example, you can find a complete project of this system in the examples under "Specifier/Tutorial" and "Developer/Tutorial".

3 - PragmaDev Specifier Tutorial

3.1 - Organization

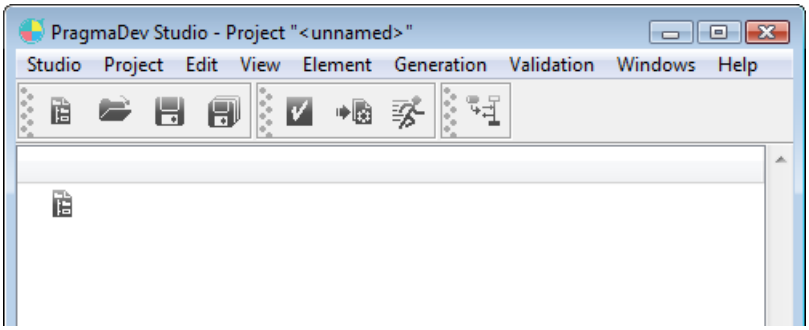
Let's get our hands on the tool! Start *PragmaDev Specifier* (or *PragmaDev Studio* if this will be the application you will be using). The window that appears is called the Project manager:



The Project manager window

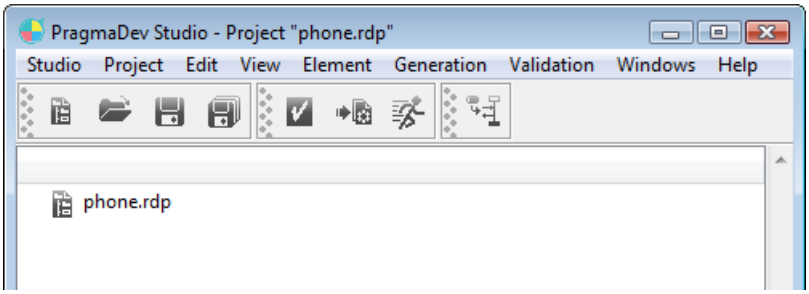
The project manager gathers all the files needed in the project. First let's create a new project with the *New project* button:

The Project manager displays an empty project:



An empty new project

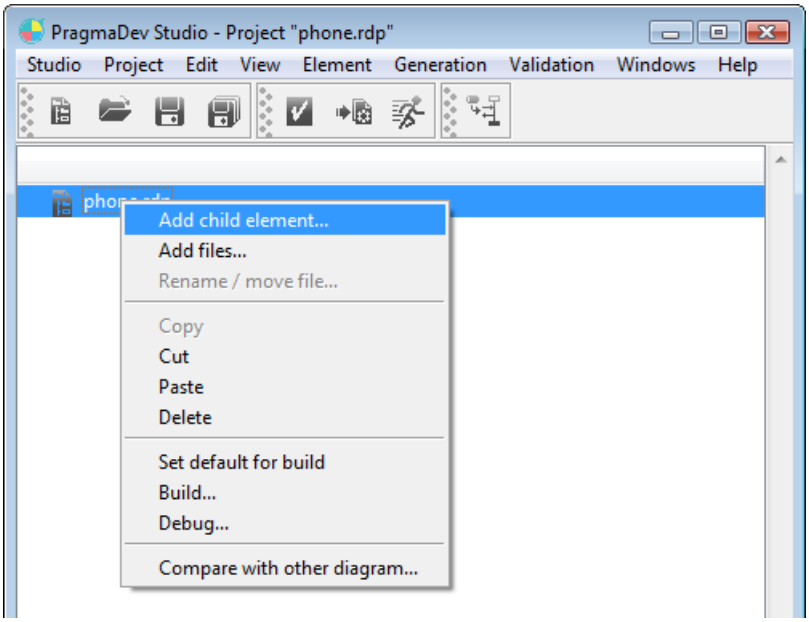
Let's save it straight away as "phone" and put it in a dedicated directory:



Phone empty project

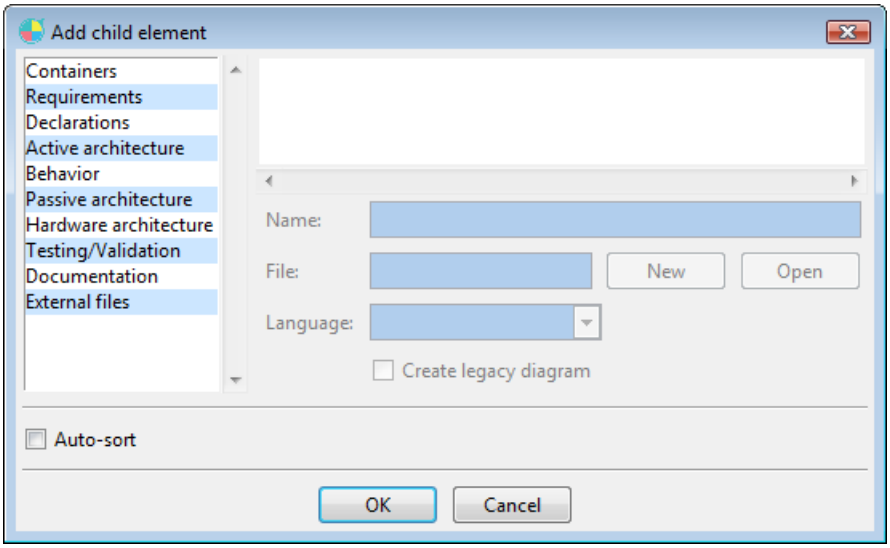
3.2 - Requirements

Let us express the requirements of our system with a Message Sequence Chart (MSC). To add an MSC, select the project, and click on the right mouse button. A contextual menu will appear:



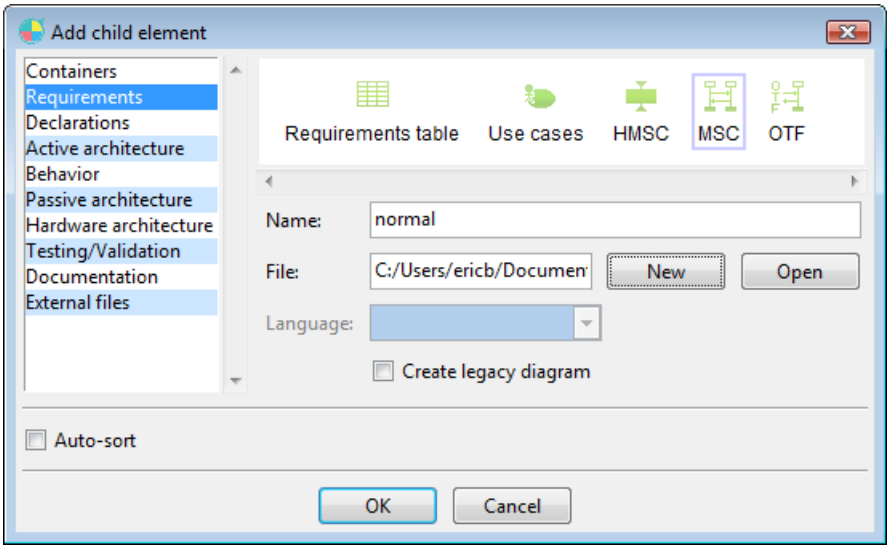
Add components to the project

Select *Add child element* and the following window will appear:



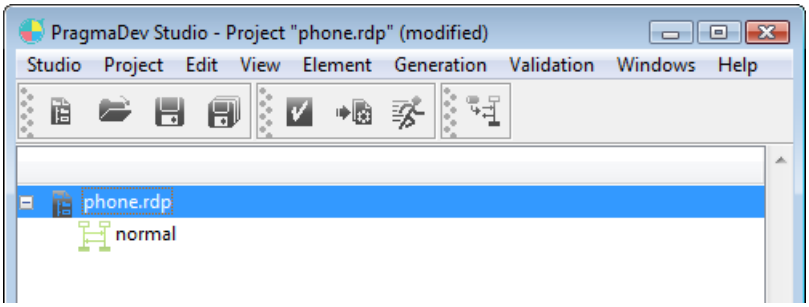
The add component window

In *Requirements*, select *MSC* element and click on the *New* button. Go to the directory where your project is and type in "normal" with no extension. Click on save and you will get the following window:



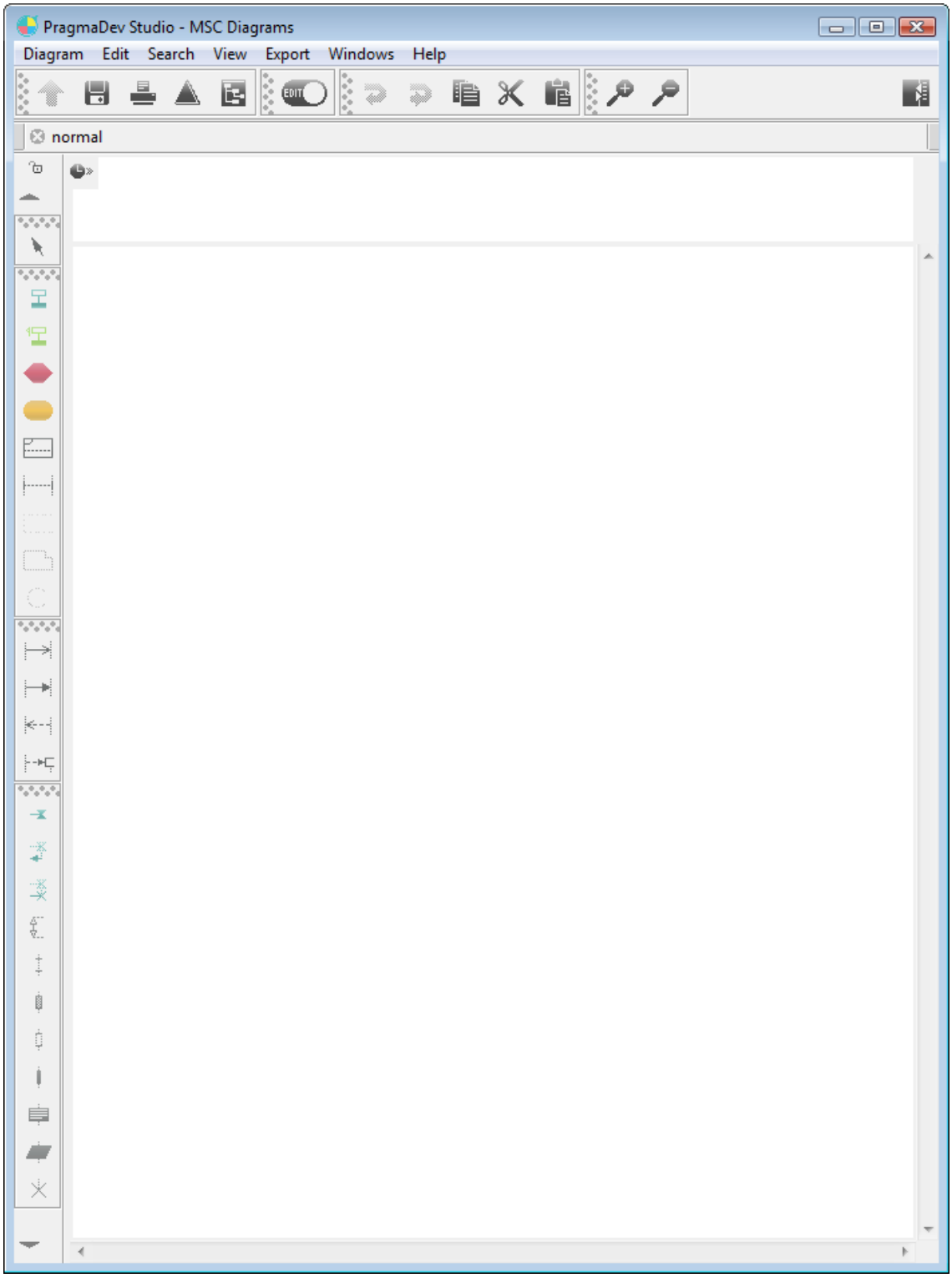
Completed Add component window

Click Ok and the "normal" MSC appears in the "phone" project:



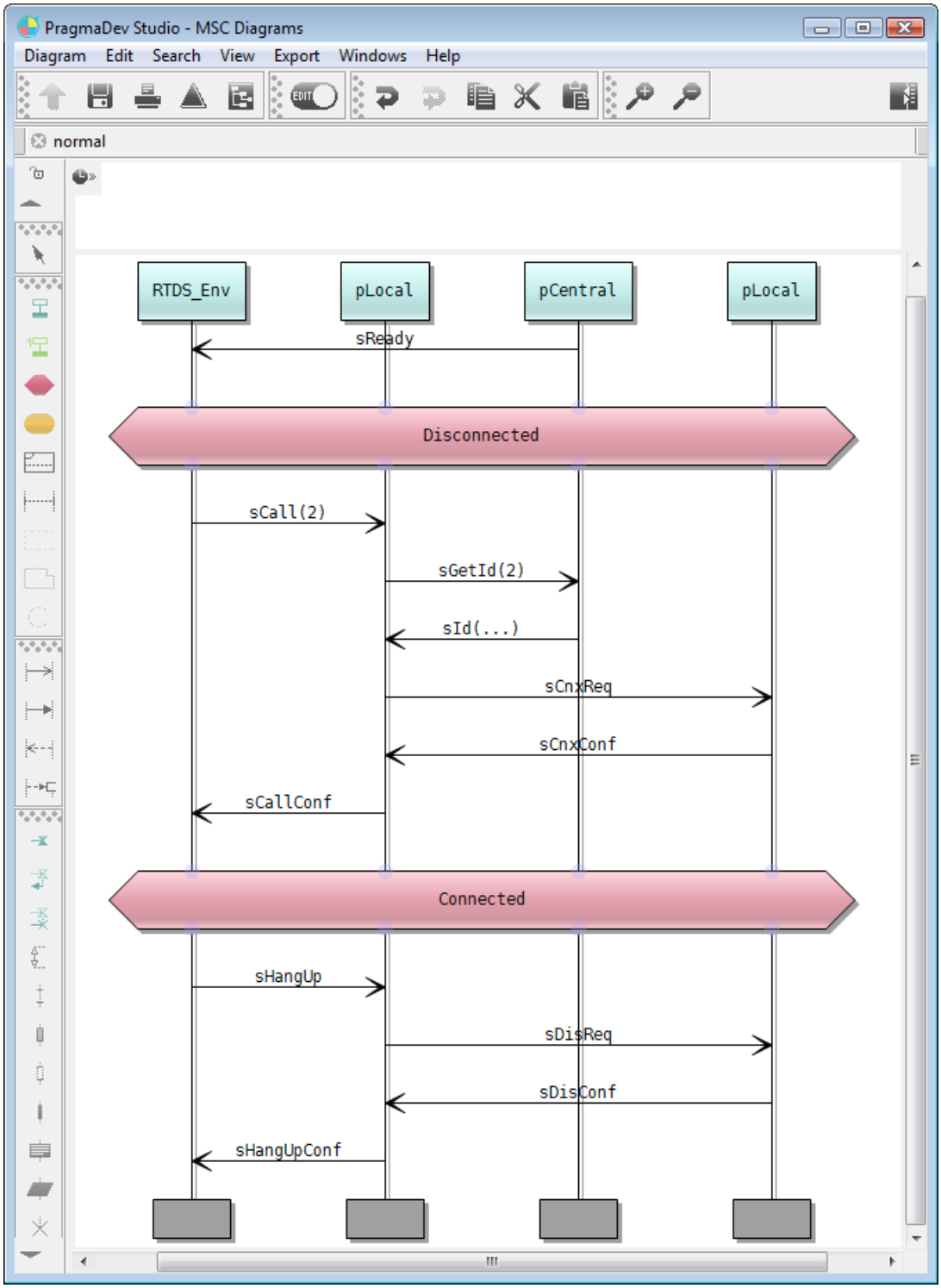
"normal" MSC in "phone" project

Double click on the MSC name or icon to open it. The MSC editor opens:



The MSC editor

Draw the following to express the requirements of our phone system:



The "normal" MSC

You will have to use the tool bar on the left. If you have any problem refer to the user's manual.

This MSC basically says the following:

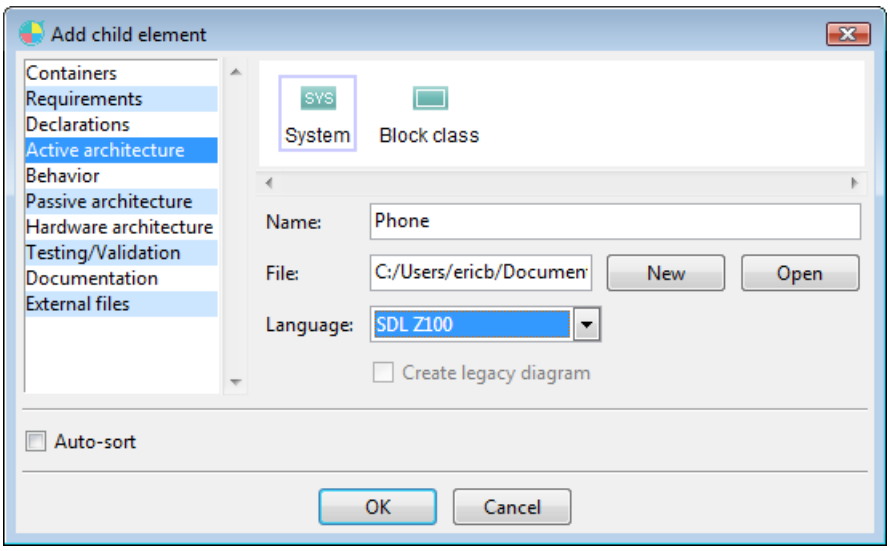
- `pCentral` indicates the system has been initialized and is ready

- The initial global state is `Disconnected`
- The user represented as the environment (`RTDS_Env`) makes a request on the first phone `pLocal` to call the phone with the number `2`
- The first `pLocal` asks the central the queue id of the phone with number `2`
- The first `pLocal` uses the id to send a connect request (`sCnxReq`) to the second `pLocal`
- The second `pLocal` being disconnected, it confirms the connection (`sCnxConf`)
- The first `pLocal` tells the environment the call has succeeded
- The global system state is then considered `Connected`
- The user hangs up
- The first `pLocal` sends a disconnection request (`sDisReq`) to the second `pLocal`
- The second `pLocal` confirms disconnection (`sDisConf`) back to the first `pLocal`
- The first `pLocal` tells the environment the disconnection is confirmed
- The overall final state is back to `Disconnected`

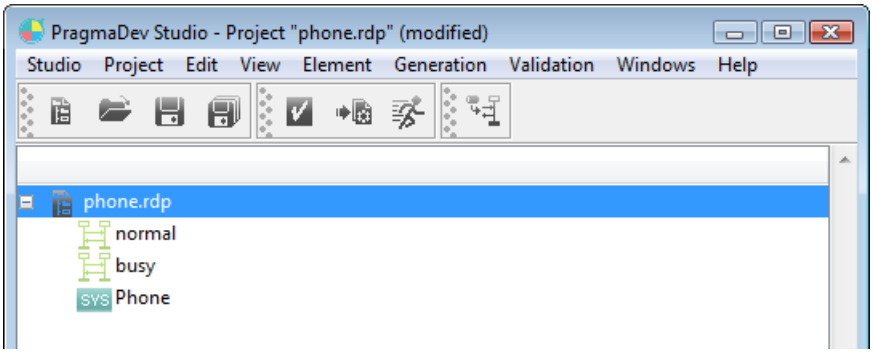
You can write some other MSCs to get clearer ideas on what you want to do. Note the instances represented on the MSC can be any type of agent. Somehow you are roughly defining the first architectural elements. You can copy from the SDL/Tutorial example the "normal" and "busy" MSCs in the project to complete the description.

3.3 - Design

Let us now specify and design the system. As for creating an MSC, select *Add child element* on the project, then the *Active architecture* category in the dialog, and select *System* component. Note that the dialog will this time require to set a modelling language for the diagram. Choose *SDL Z100*:

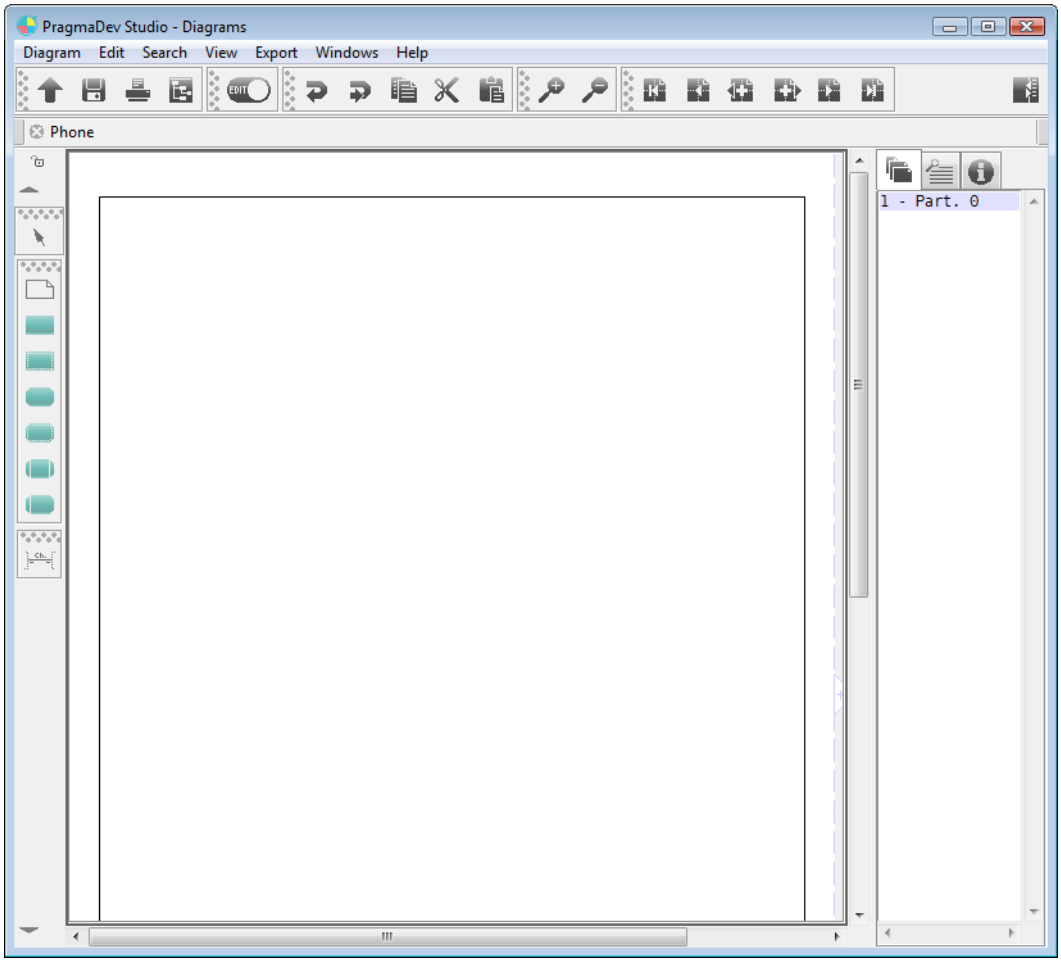


After validating the dialog, the system appears in the project:



"Phone" SDL system in the "Phone" project

Double click on the system name or icon to open the system diagram in the SDL editor:



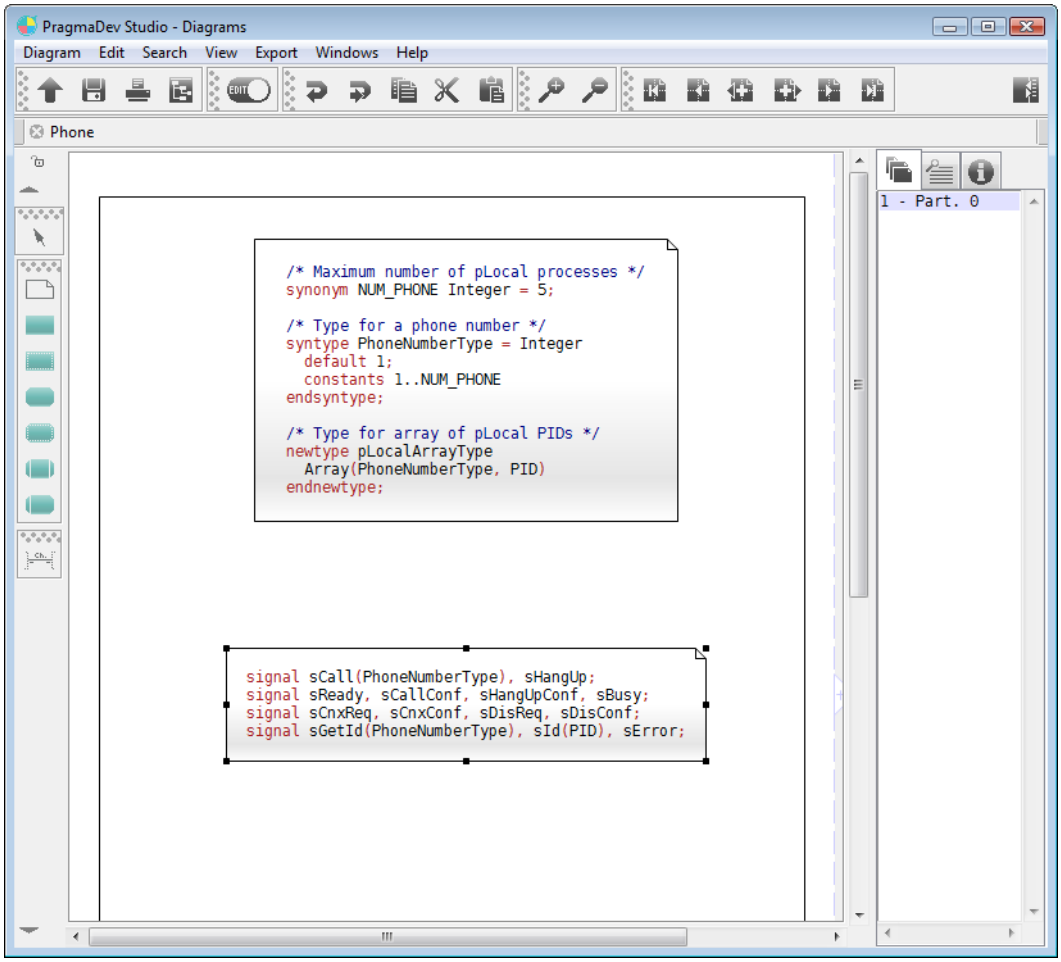
The SDL editor

The system will be divided into two main parts:

- The definition of the data types and messages we will use.
- The architecture in terms of processes.

To avoid mixing things, we will use partitions in the diagram. A partition is just a means to separate different kind of contents within a diagram; it is just a group of pages that can contain any symbol allowed in the diagram.

PragmaDev Studio has created the first partition for us, so let's use it to declare the types and messages we'll need:



Declarations in the "Phone" system

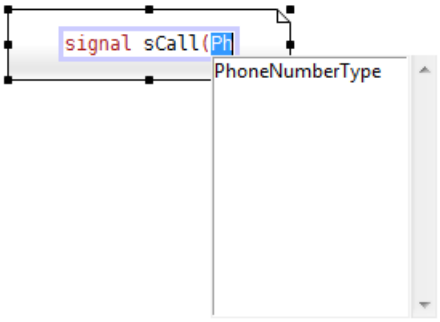
The top declaration text box declares the data and types we need:

- The `synonym` declaration declares a constant for the maximum number of phones;
- The `syntype` declaration declares a special type for the phone. This is basically an integer restricted to be between `1` and the maximum number of phones;
- The `newtype` declaration actually declares the type for the array of phone processes; the index is a phone number, and the value is a `PID`, which is a basic type in SDL, just as `Integer`.

The second declaration text box declares the signals that will be used in the system. They are mainly the ones we used in the MSC we created earlier, plus a few ones for error conditions. Three of the signals we declare have parameters: `sCall`, `sGetId` and `sId`. Note the `sCall` and `sGetId` signals use the `PhoneNumberType` type we've declared above.

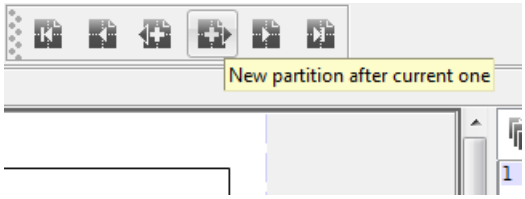
PragmaDev Specifier will offer to auto-complete your code as you type. So for example, when declaraing the `sCall` signal, if you start to type the signal parameter type, a list will

appear under your text cursor listing all known types that start with the text you've already typed:



You can select one of the choices by using the up and down arrow keys, or by clicking on it in the list. Note that entities defined in a symbol are known only when the symbol has been validated, and if its syntax is correct.

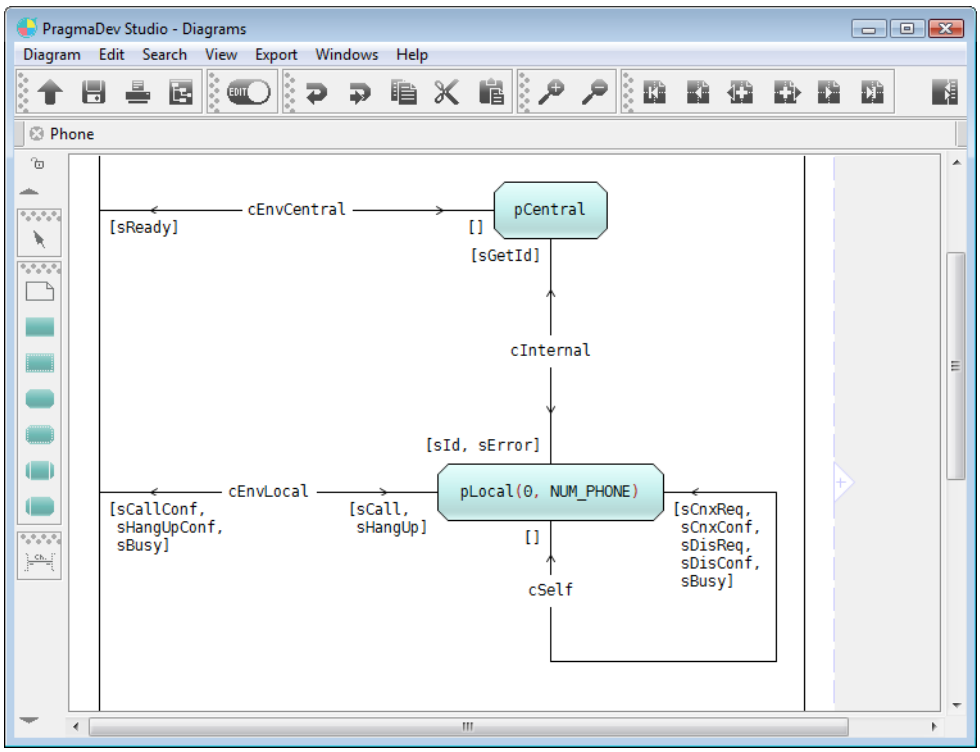
Now let's design the architecture of the system. As we said, we'll use another partition, so let's create it first, using the new partition button in the partition tool-bar:



A new empty partition appears. What you've already done is of course not lost: you can go back to it using the other buttons in the partition tool-bar.

The system being very simple it will not require any block decomposition. The central will be a process as well as the phones. All the phones have the same behavior so they will be several instances of the same process. The phone system is therefore made of two pro-

cesses. For better legibility their name will be prefixed with a "p" because they are processes:



phone system view

Notes:

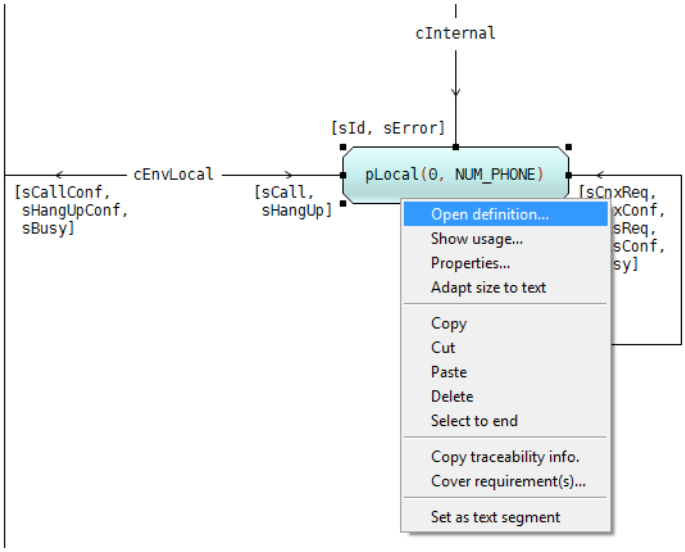
- This architecture is not strictly correct in regular SDL, since processes should not appear directly at system level. But PragmaDev Studio allows it, so let's keep things simple.
- To draw the `cSelf` channel keep the shift key down and click where the channel should break. To change the position of the channel name, click on a segment, right-click on it and select *Set as text segment* in the contextual menu.
- Signals and the `NUM_PHONE` constant have already been declared, and process names appear in the MSC. So their names will be auto-completed by PragmaDev Studio.

Since `pCentral` is making the link between the `pLocal` processes and considering the number of phones can be modified, `pCentral` will create all instances of `pLocal`. To represent that, the name `pLocal` is followed by the initial number of instances and the maximum number of instances we defined in the `synonym` in the other partition.

Messages to be exchanged between the processes are listed in the channels . To specify the incoming and outgoing messages in the diagram double click on the "`[]`" and type in between the square brackets. The channel going to the outer frame is implicitly connected to the environment. In the above example the channel `cEnvLocal` connects `pLocal` to the environment and defines `sCall` and `sHangUp` as incoming messages and `sCallConf, sBusy` and `sHangUpConf` as outgoing messages. The channel `cEnvCentral` connects `pCentral` to the environment and defines `sReady` as an outgoing message. The

`cSelf` channel has been created to represent messages exchanged between the different instances of `pLocal`.

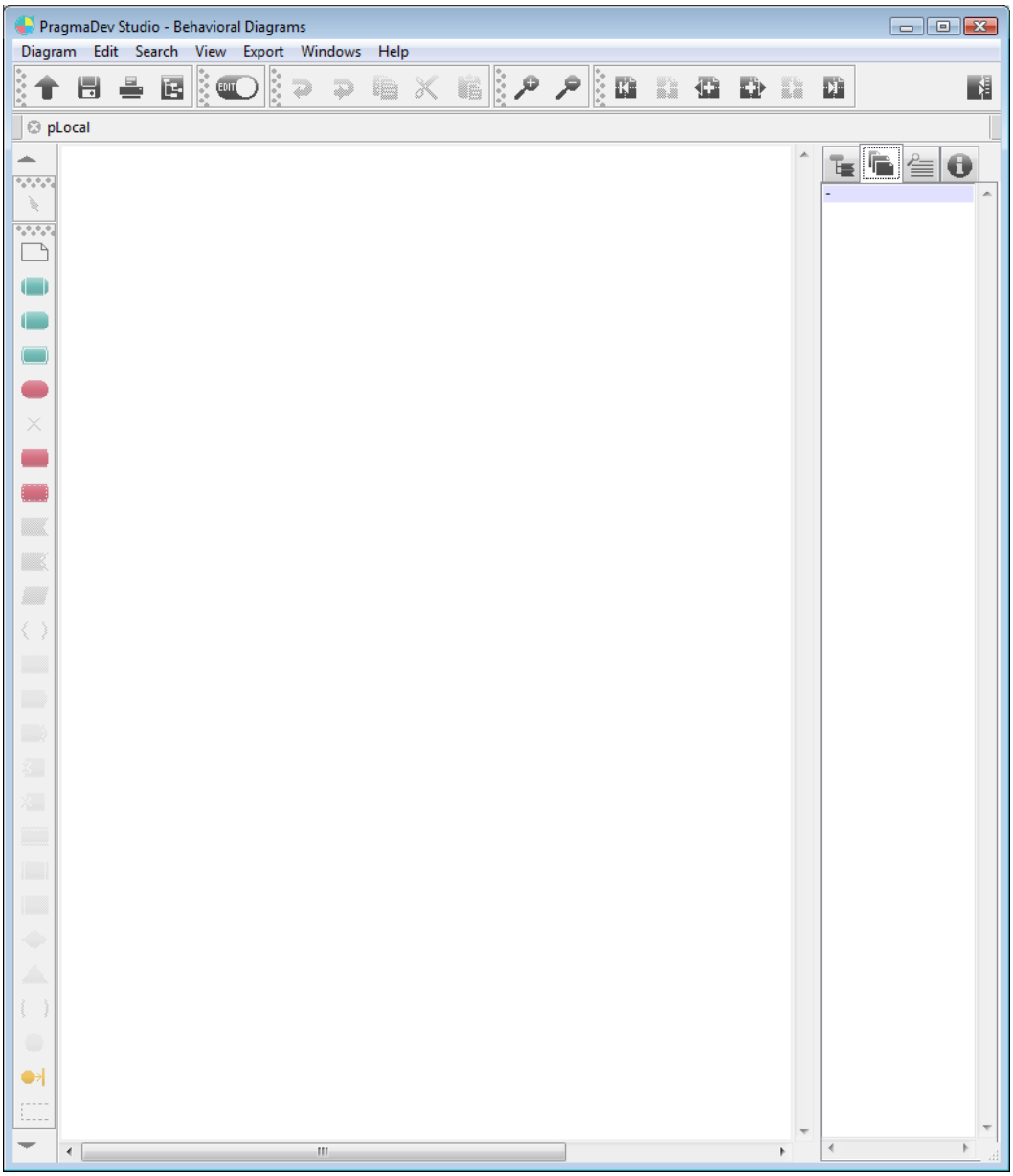
Select `pLocal` and click on the right mouse button to open the process definition, or simply click on the button that appears when you hover the mouse pointer over it:



Contextual menu

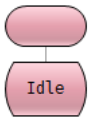
Since the process is not in the project, it will be asked what type of element must be added. Keep the pre-selected options, *Process* element in *Behavior* and click OK.

A new window opens, showing the process definition. As for the system, the first partition has been automatically created:



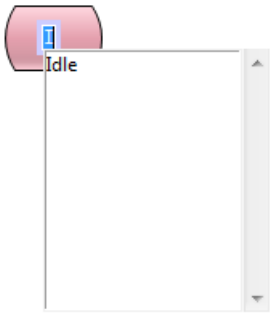
The process behavior description in SDL editor

The first thing to design is the start transition. It is what the process will do as soon as it is created. In the case of `pLocal` process, we do nothing:



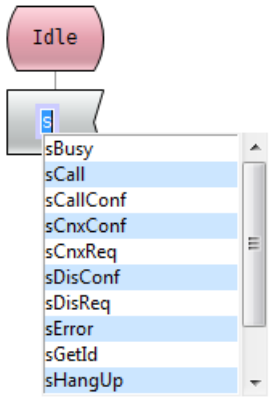
That transition means that once the process is started it will go to state `Idle`. Place a start symbol , keep it selected and click on the state symbol in the tool bar . The state symbol is automatically inserted and connected after the start symbol. An internal data

dictionary is updated on the fly to ease the writing of the process behavior. First create the `Idle` state definition: click on the State icon and put it in your diagram:



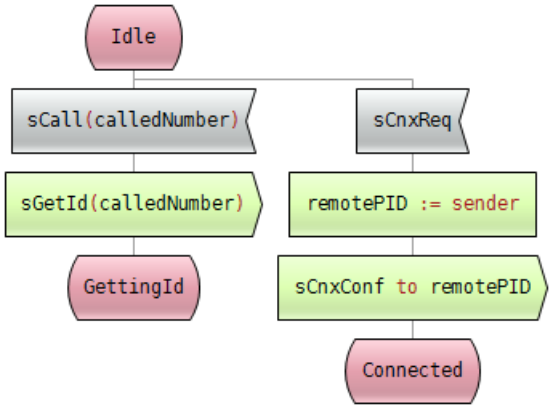
The state name is in edit mode so you can directly type `Idle` in it; but you may also use auto-completion to list the available choices for the state symbol.

Once the state has been defined, click on the input symbol in the tool bar and the input message symbol will be automatically inserted below the state symbol. When you start typing, a list of all available messages will appear:



Select the `sCall` message and complete it with the correct parameter. This facility is context sensitive and works for almost everything: SDL keywords, agents, channels, signals, states, types, variables, timers, ... You can now finish the state description by yourself as explained below.

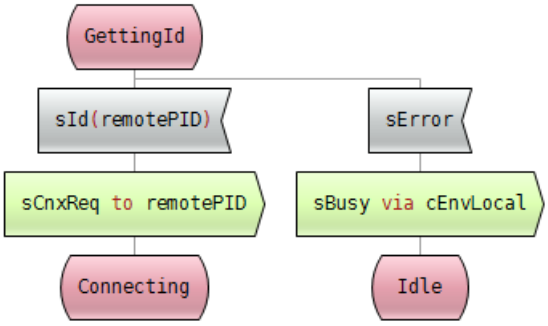
Considering the requirements described earlier, the `pLocal` process can either be asked to make a call by the operator or receive a call from another phone. The `Idle` state can therefore receive two types of messages described below:



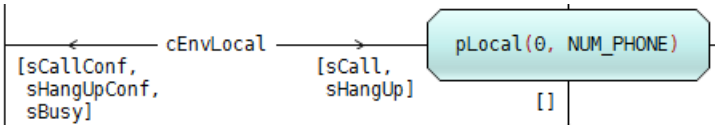
When receiving `sCnxReq` message, it will reply `sCnxConf` to the `sCnxReq` sender. `To` and `sender` are SDL keywords in the output symbol. The `sender` id is stored in the `remotePid` variable. The process then goes to `Connected` state.

If asked to make a call, the phone number to call needs to be retrieved. To do so, a variable of the correct type is given as parameter of the receiving message. It will be assigned when this message is received. Since `pLocal` has no idea how to address a phone number it asks the central process the process id of the called `pLocal` with the `sGetId` message. The `calledNumber` variable is re-used as is. No receiver is specified since the receiver process is completely determined by the system architecture. We may however have used `TO pCentral` to specify it, or even `to parent` since `pLocal` was created by `pCentral`. The process then goes to `GettingId` state, waiting for the central to answer.

Once the pid of the remote phone is received from the central, it is stored in a local variable and the connection request message `sCnxReq` is sent. The process then goes into state `Connecting`. If the pid of the receiver was not found, the `sError` signal is received. A `sBusy` message is sent back to inform the user and the process goes back into state `Idle`:

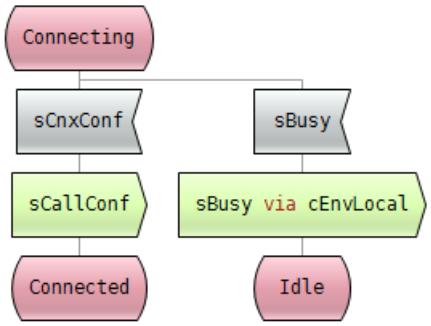


Note the receiver for the `sBusy` message is specified by using `VIA cEnvLocal`. It means that the signal will be sent to the process at the other end of the channel cEnvLocal, connected to the process pLocal in the system diagram:



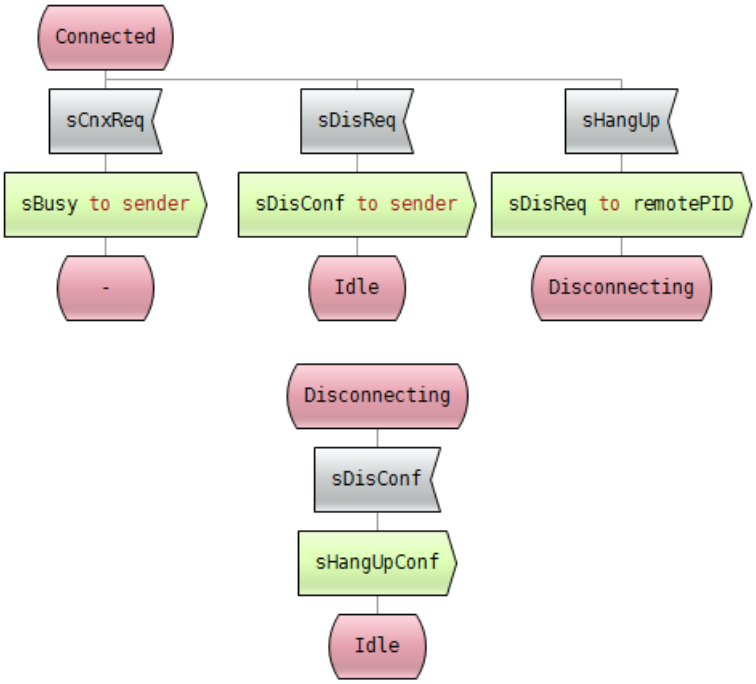
Since this channel is connected to the system's external frame, the signal will go to the environment. Please note specifying a receiver for `sBusy` is necessary here, since this signal may be sent not only to the environment, but also to the other `pLocal` processes in the system. If a receiver is not specified, the SDL semantics is to choose randomly a receiver among the available ones, so the signal may have been received by the wrong process.

Once the connection request has been sent, the remote process is either available and replies `sCnxConf`, or not available and replies `sBusy`:

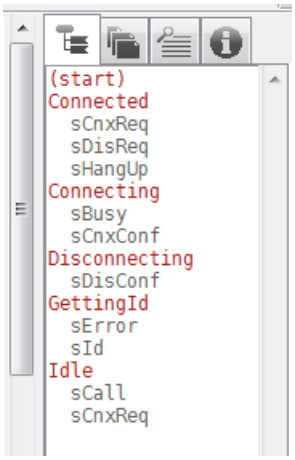


Depending on the answer the resulting state is different.

Now that you have understood the basics of the finite state machine you can complete the process behavior:



As the description is done, the browsing window on the right side is updated allowing to quickly jump to a transition: just click on the transition. This is especially useful when the system gets big.



It is now time to declare variables in our process. To do so, the text symbol in the process behavior diagram is used. The declarations are introduced via the keyword `DCL`, followed by a list of couples `<variable name> <variable type>`, with an optional default value:

```
dcl
  remotePID PID := NULL,
  calledNumber Integer;
```

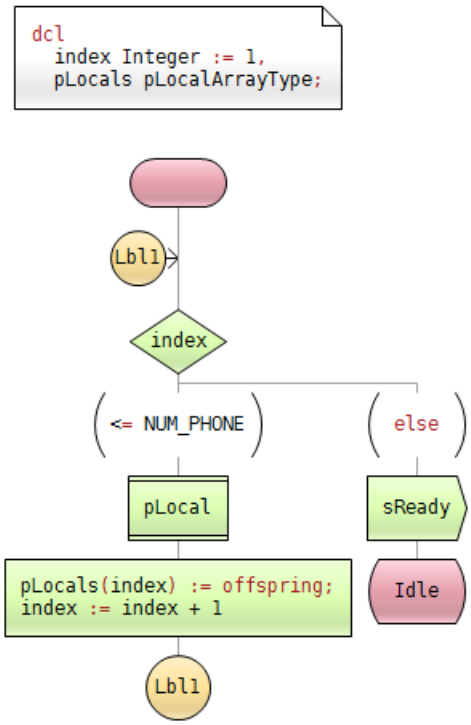
Note the type of the variable used in the input and output symbols for `sCall` and `sGetId` are not strictly the ones appearing in the definition: the signal declares a `PhoneNumber-Type`, but we use a regular `Integer`. This is no problem as long as the `Integer` satisfies the conditions set on the `PhoneNumberType` type.

Let's have a look at process `pCentral` now. It must do the following things:

- At startup, it creates all instances of `pLocal` and gives them a new phone number.
- When asked for a phone number, it sends back the pid for the corresponding process.

Go to the system diagram `Phone`, and open `pCentral` (via the contextual menu or the button). Since the process is not in the project, it will ask if it should be added. Answer *Yes* and a pre-filled *Add child element* window with the process name appears. Click *OK*

and the process definition window appears. Let's first write the needed declarations and the initial transition:

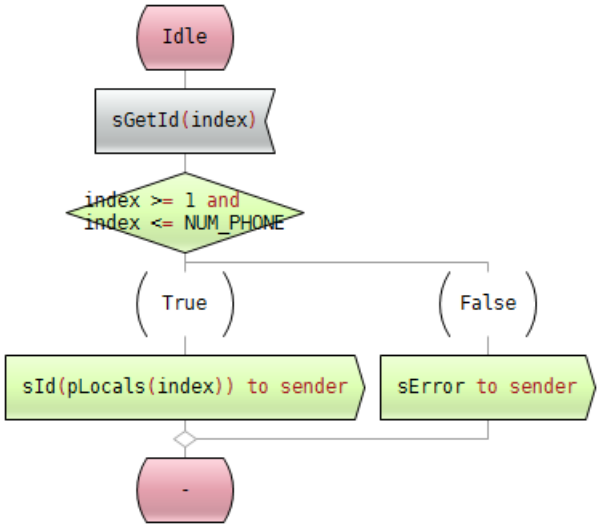


The variables include an index which will be used as the phone number for created `pLocal`'s, and an array mapping the phone number to the pid.

The initial transition creates all instances of `pLocal` within a loop testing `index <= NUM_PHONE`. Each time the loop is executed the `pLocal` process is created and its process id (`offsrping` keyword for the parent process) is stored in the `pLocals` array, using the phone number as index.

After the `pLocal` processes creation, the `sReady` signal is sent to the environment to indicate initialization is finished and the process goes to state `Idle`.

Note we have voluntarily introduced an error by typing `";"` instead of `","` at the end of the first line in the lowest block of code to later show how to analyze the errors.



The only request that can be received by `pCentral` process is `sGetId`. The phone number to reach is the parameter passed to the signal, which we will receive in the `index` variable. The process id of the phone is extracted from the array and sent back directly to the sender of the `sGetId` message (SDL keyword `sender`). If the phone number is out of range, an error message is sent back to the sender.

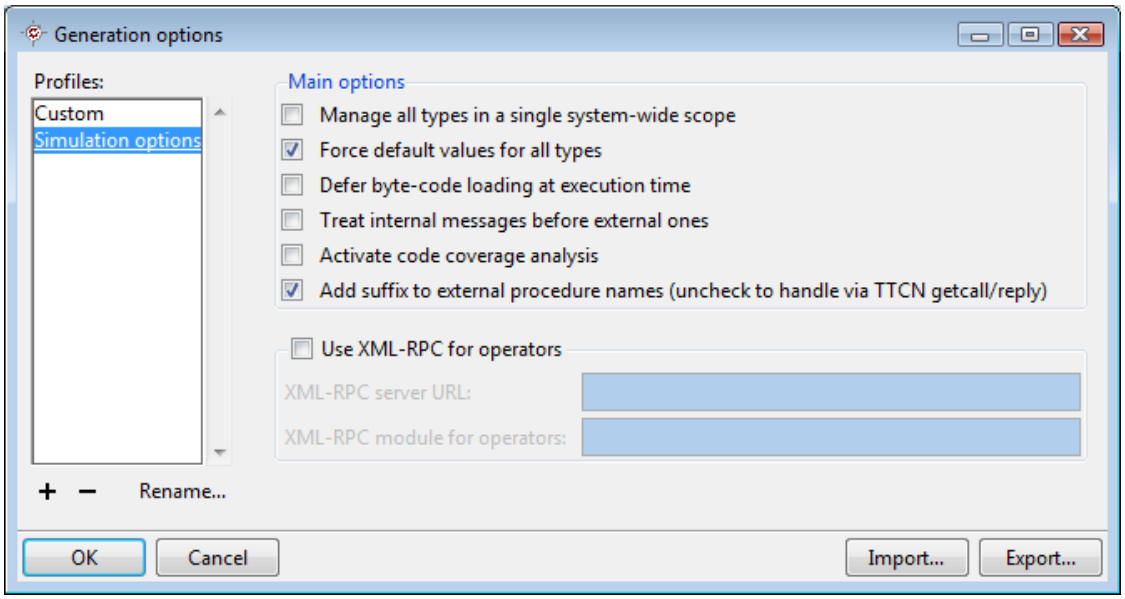
3.4 - Simulating the system

Now that the system has been designed, we'll debug it using PragmaDev Studio's SDL simulator. The simulation process is divided into two main phases:

- First, the code for all transitions in all system's processes is transformed into a language called SDL byte-code. This language is easy to generate from the process description but far easier to interpret than plain SDL instructions. This language is used only internally and has no direct external representation.
- The generated byte-code is then executed based on a scheduling managed by PragmaDev Studio. The simulation conforms to the SDL semantics: all transitions are considered to be executed in no time and cannot be interrupted.

3.4.1 Simulation options

The Simulation options and the code generation options are edited via the *Generation / Options...* menu. By default a valid simulation profile is listed as well as an empty code generation profile:



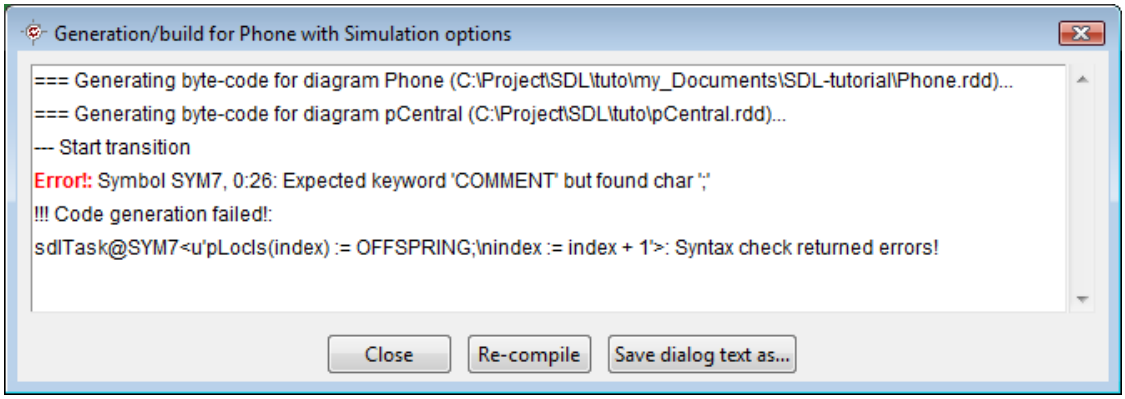
As the simulation profile is the only valid debug profile, it will be used by PragmaDev Studio by default.

3.4.2 Byte-code generation

Select the Phone system in the project manager and click on the *Execute* quick button in the tool bar: 

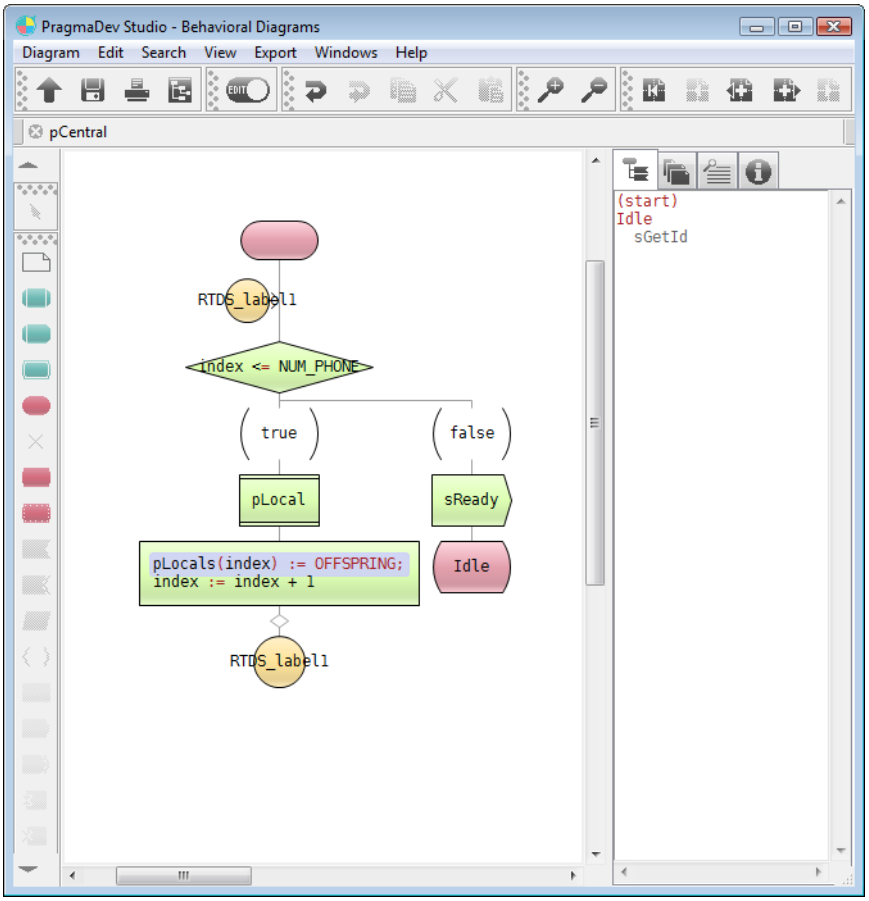
A log window opens and displays the actions performed by the byte-code generator. Before actually generating anything, PragmaDev Studio performs a global syntax and semantics check on the written code. So any basic error such as typing mistakes or misspelling in variable names will be reported during this phase.

Since we introduced an error in process `pCentral`, this is what appears in the byte-code generator log window:

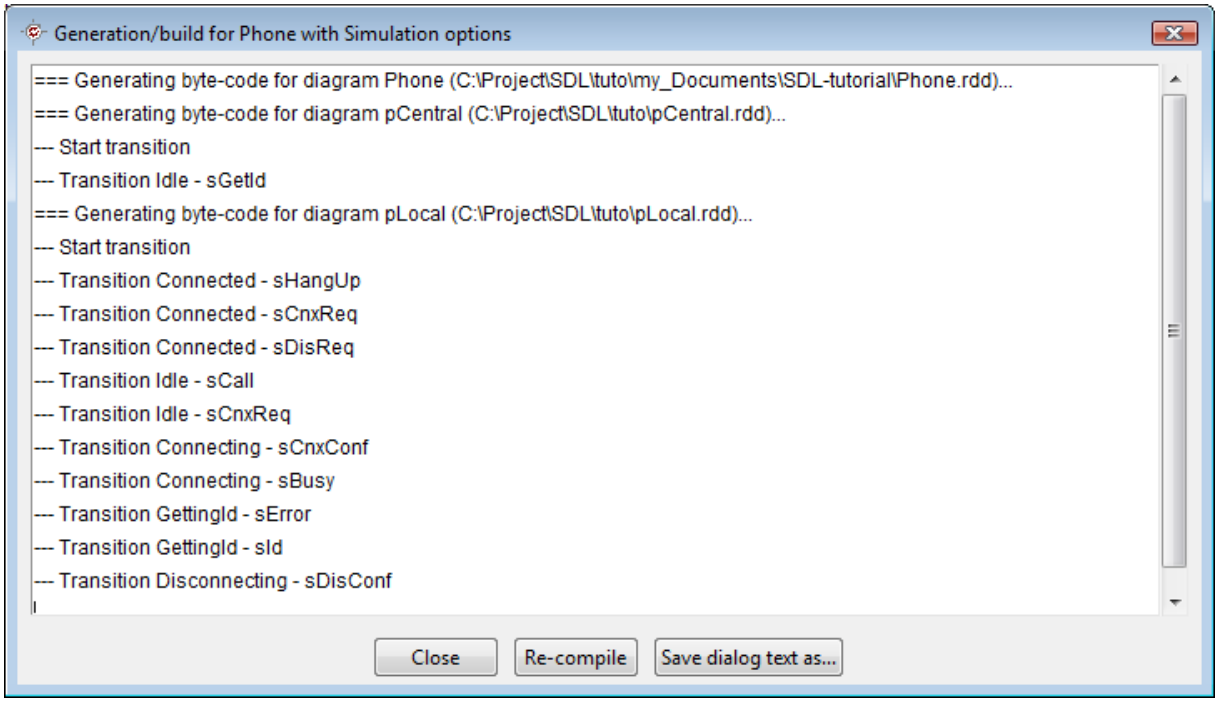


The byte-code generation started at system level, then went down in `pCentral`. During the generation for `pCentral`'s start transition, the inversion between ";" and "," was encountered. So the generation stopped and this error message was displayed.

Double clicking on the error automatically opens the SDL editor and selects the symbol where the error occurred:

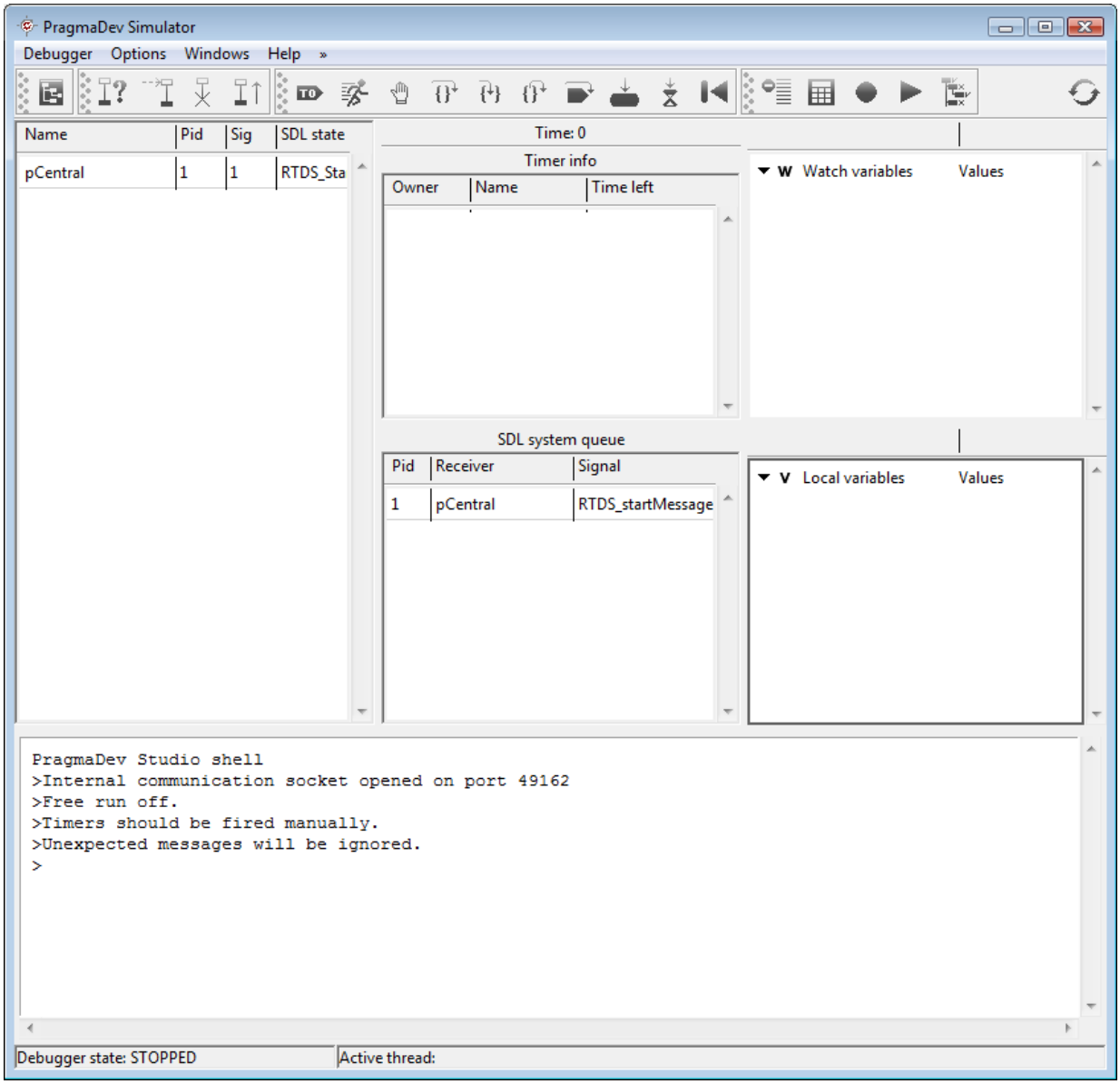


Once the error have been corrected the log window should look like this:



3.4.3 The SDL simulator

Once the byte-code generation is over, the SDL simulator window opens automatically:



The SDL simulator window

The simulator window shows a global state of the running system in terms of:

- Running processes
- Sent messages
- Started timers
- Local variables when in the context of a running process
- Watched variables, allowing to see the value of any variable at any time

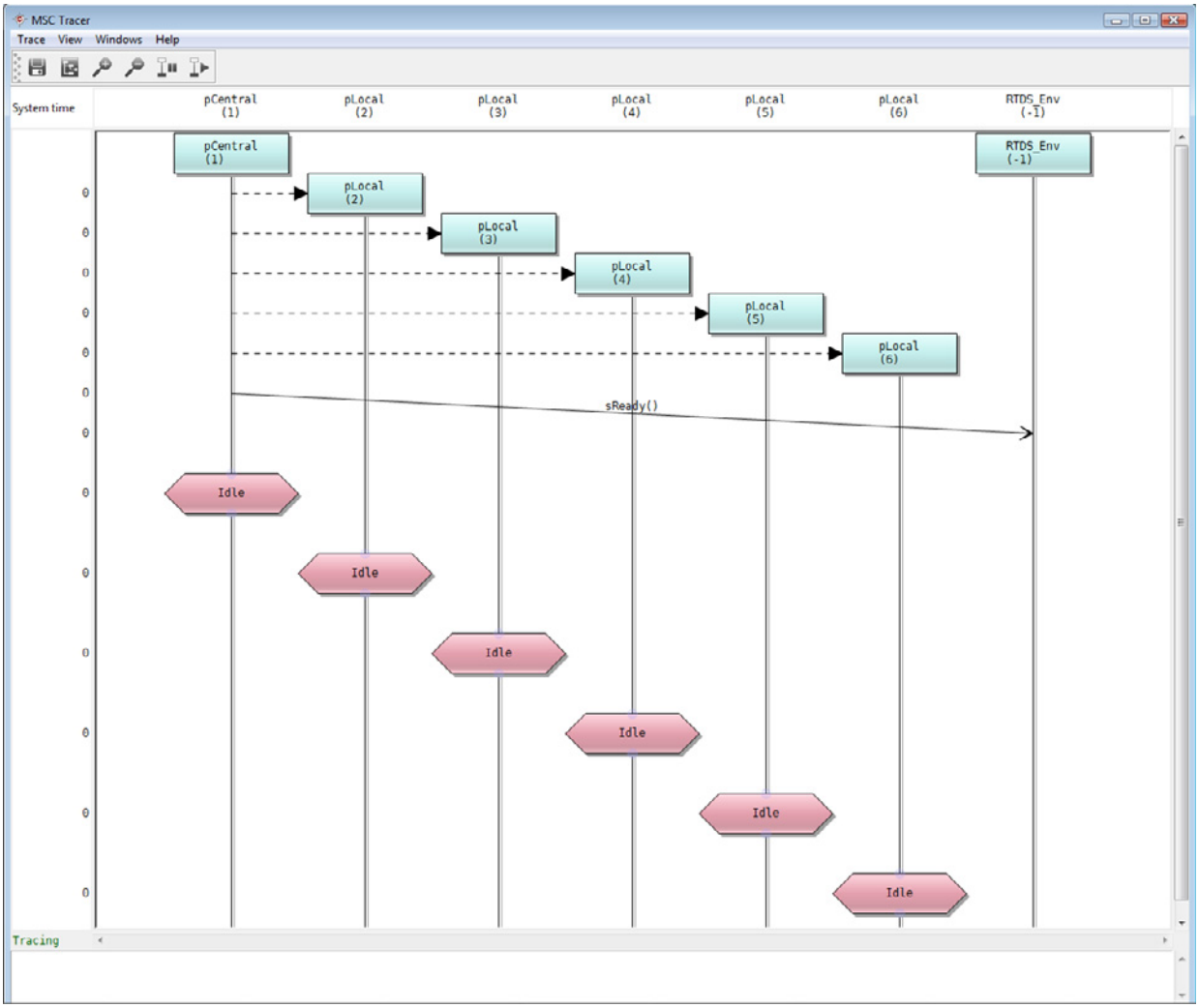
The lower part of the window is a shell where actions taking place in the system will be reported.

Let's first run a MSC trace so that we can see graphically what is happening in the system.

Click on the *Start MSC trace* quick button: A *MSC Tracer* window appears.

Now let's actually start the system by clicking on the *Run the system* quick button:

Let the system run until all `pLocal` processes are created by `pCentral` and their start transition executed:



Note you can detach the execution buttons bar by dragging it away from its header (the zone looking like this:):

Detached execution buttons toolbar

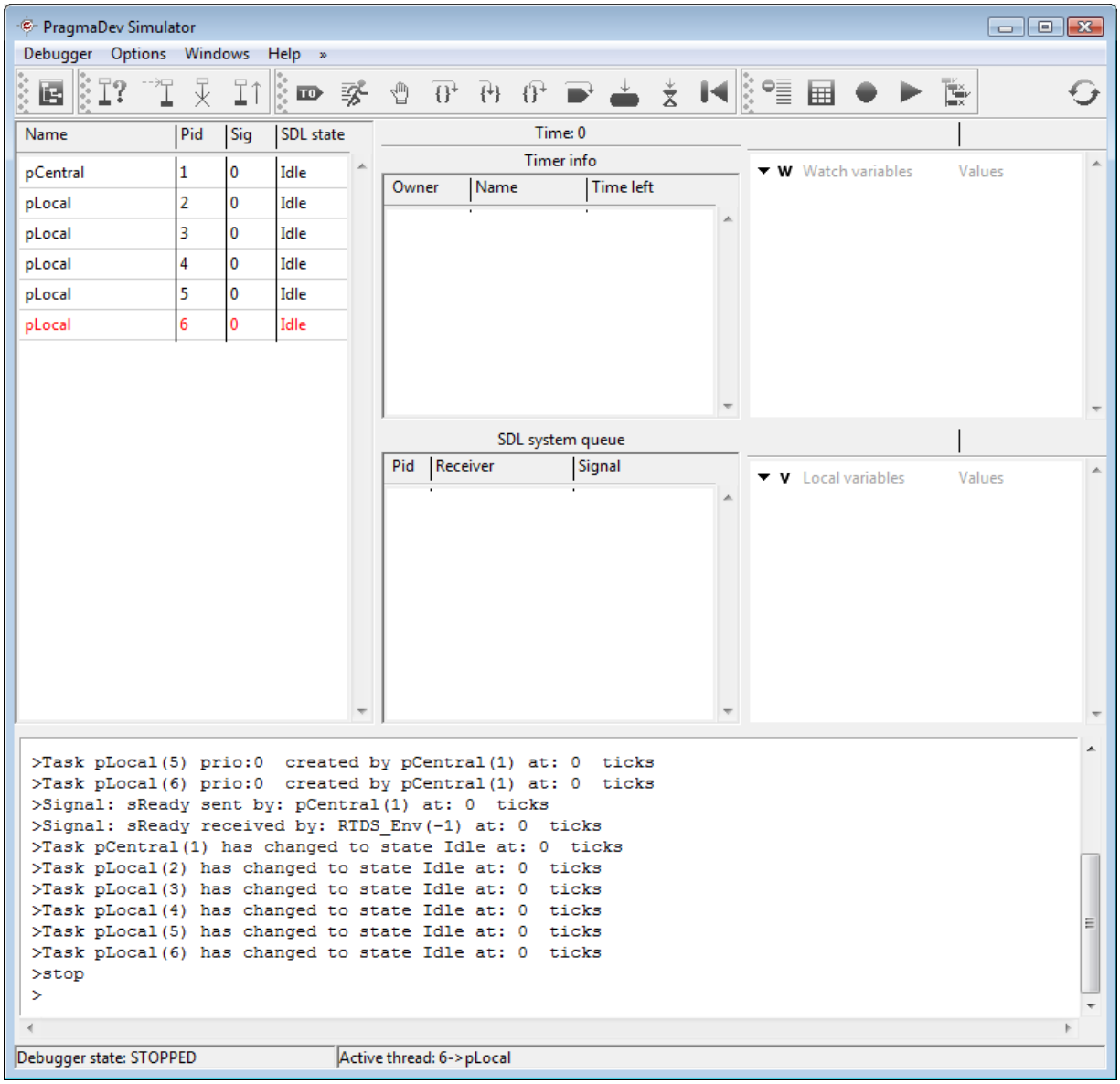
The environment is represented by the pseudo-process `RTDS_Env`. This is not really a process as it does not appear in the list of running processes in the simulator window and

has no code associated. It is only used to trace messages sent from and received by the environment.

Process `pCentral` dynamically creates 5 instances of `pLocal`, sends the `sReady` message to the environment and goes to state `Idle`. Each `pLocal` instance then go to state `Idle`. On the left is the value of the system time. According to SDL semantics, all start transitions executed in no time, so the system time is still 0 after all processes have started.

Click on the *Stop* button to break execution:

The SDL simulator window shows the list of all running processes, displaying for each one its name, process id, number of messages in its message queue and SDL state:

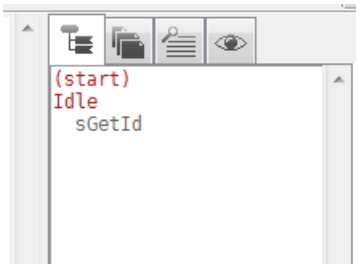


SDL simulator window

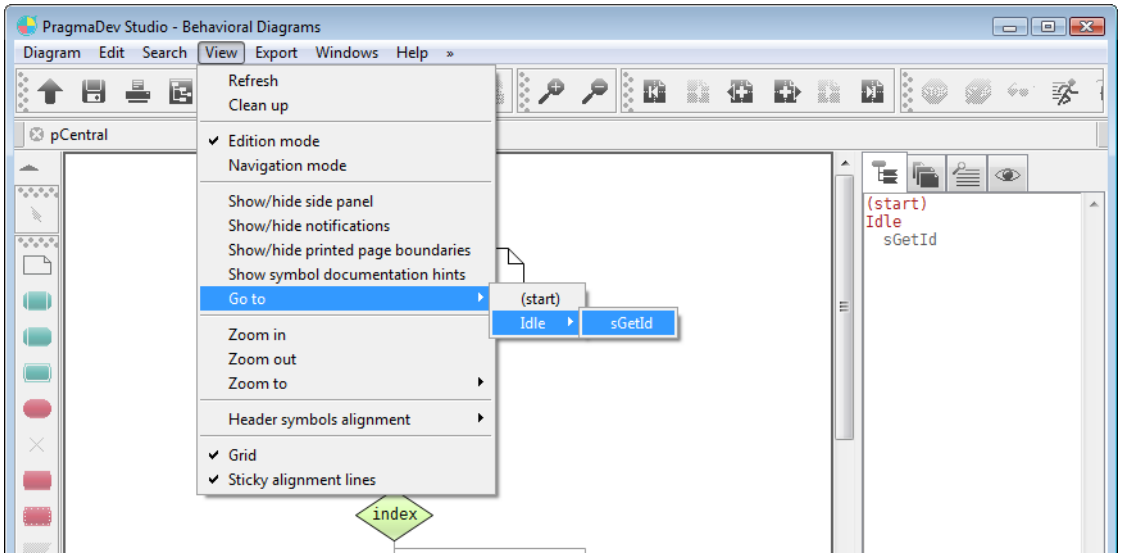
Now let's put a breakpoint in process `pCentral`:

- Open `pCentral` from the project manager.

- Go to the transition for signal `sGetId` in state `Idle` using the state / message browser:

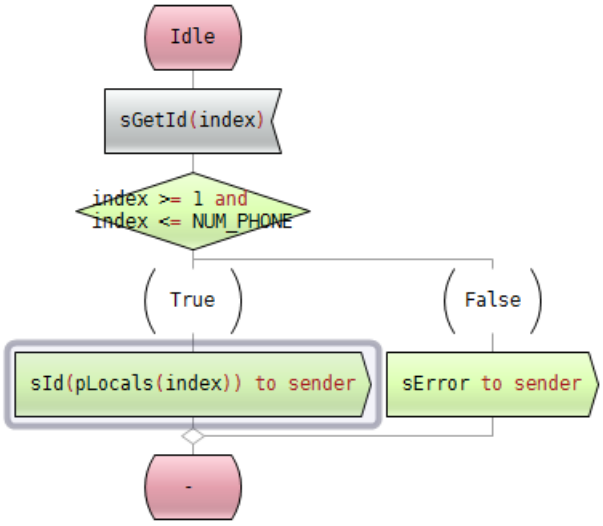


or the *View* menu:

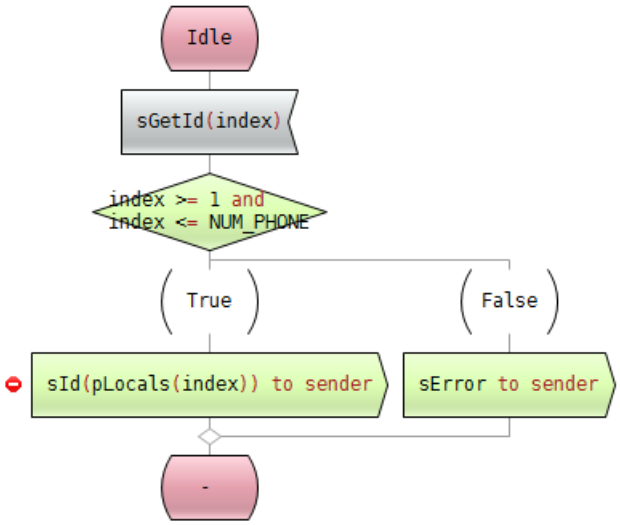


The state / message browser and the *View / Go to* menu allow to quickly navigate among the transitions defined in the process. Selecting the transition will automatically open the partition where the transition is and scroll to the corresponding signal input symbol.

- Click on the signal output symbol just after the decision's `True` branch:

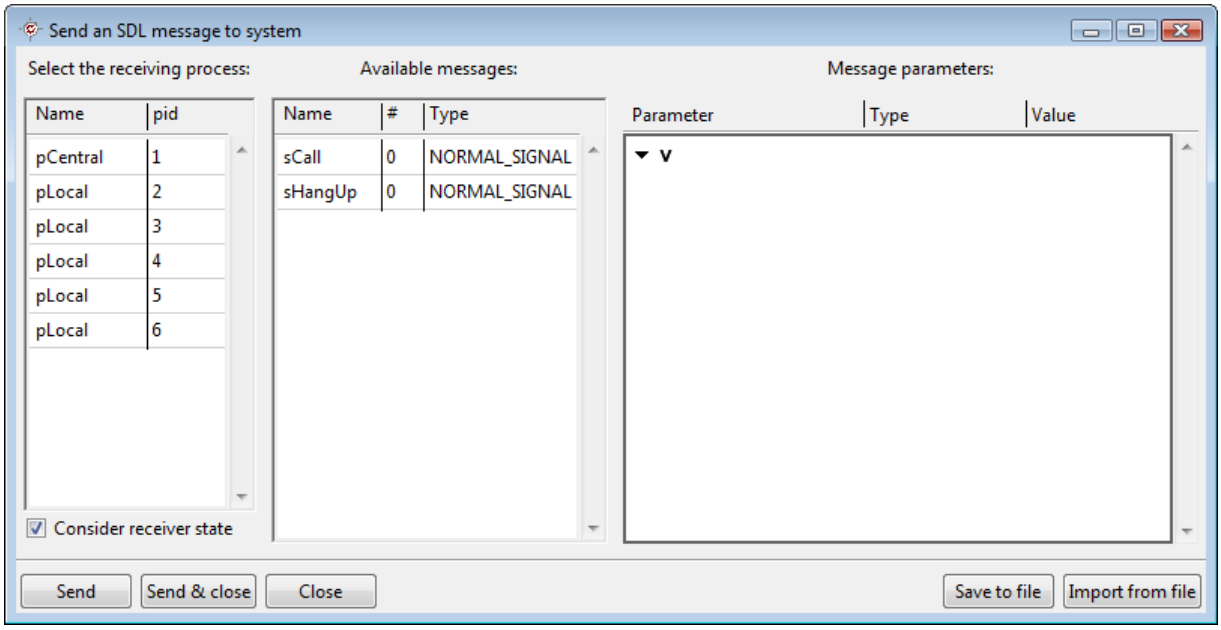


- Click on quick-button STOP or go to *Debug / Set breakpoint* menu in the *SDL editor*. A breakpoint symbol is displayed on the side of the selected symbol:



We will now simulate an incoming message from a user:

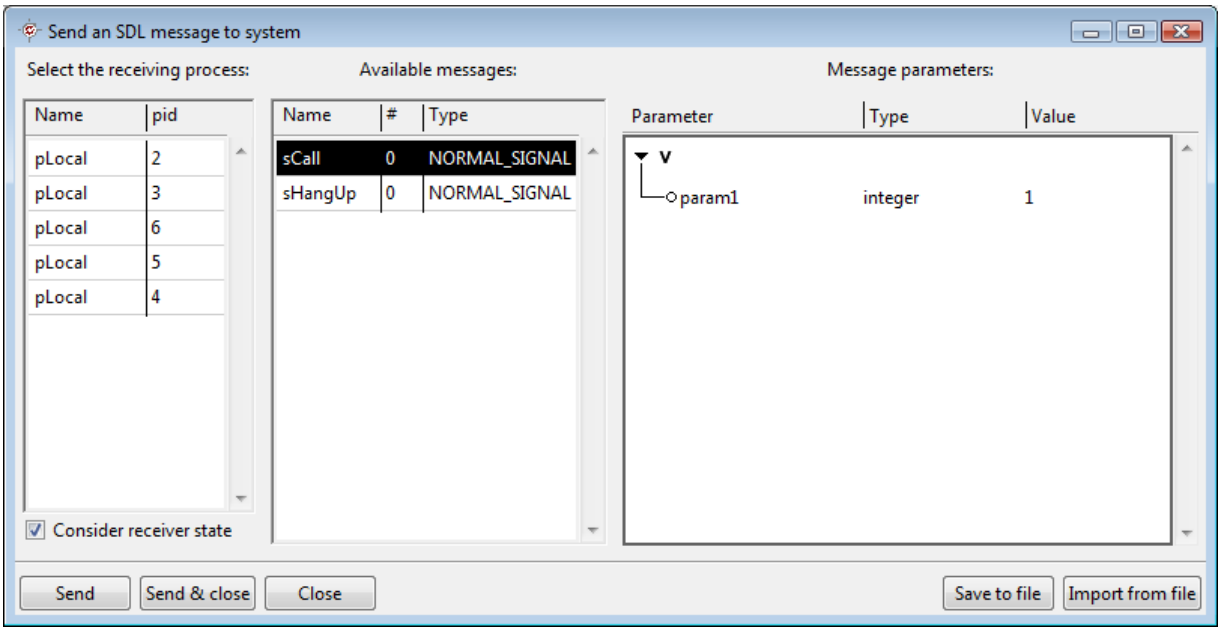
- Go to the *SDL Simulator* and click on "Send an SDL message to the running system" quick-button TO
- The *Send an SDL message* window shows up:



Send an SDL message window

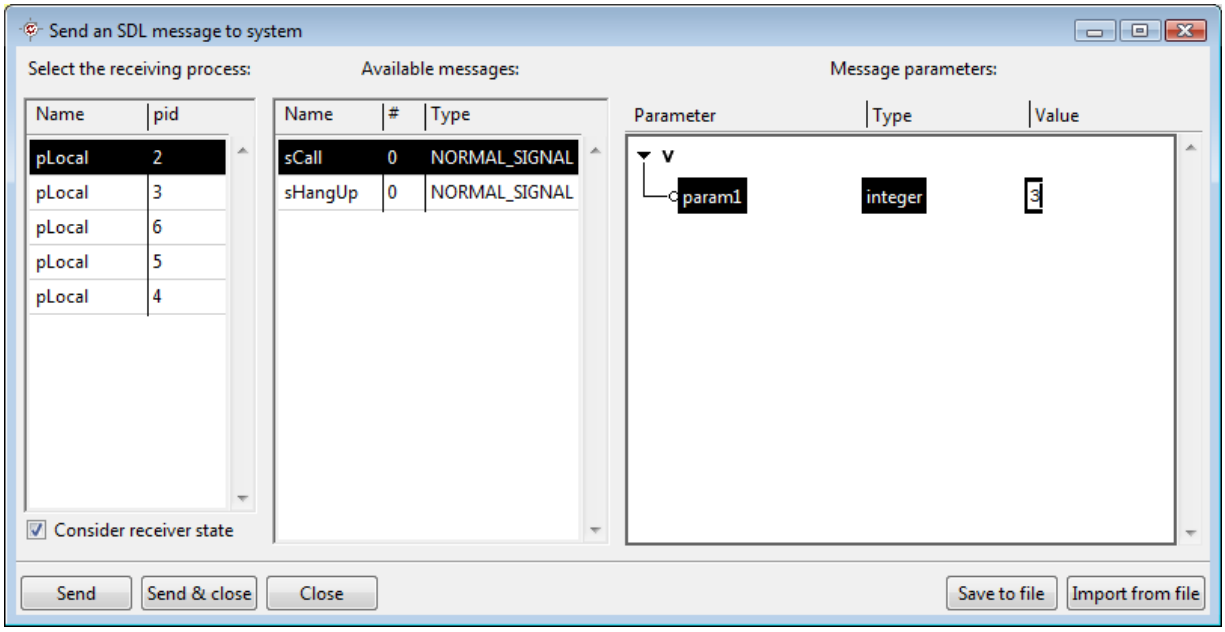
On the left are listed all possible receiving processes, in the middle all possible messages, i.e. all messages used in the SDL system, and on the right the value of the parameters associated with the selected message. Clicking on either the mes-

sage or the receiver will restrict the other list to show only the consistent choices. Here, we want to send a `sCall` signal, so let's select this signal in the list:



Since process `pCentral` cannot receive signal `sCall`, it disappears from the list of available receivers.

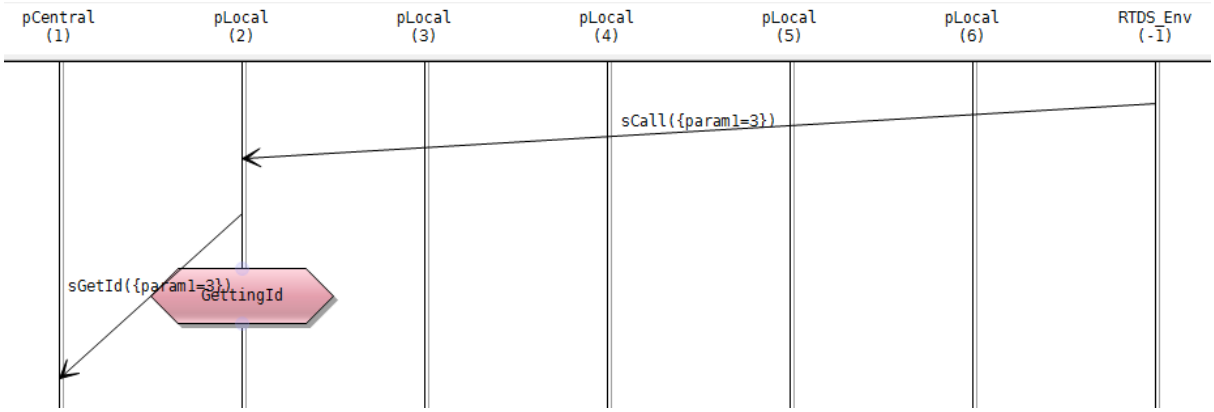
- Now let's select the signal receiver and input the called phone number, which should be passed as a parameter to the `sCall` signal:



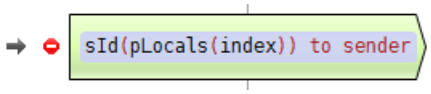
The signal parameters are described in the right part of the signal send window. Double click on the parameter to edit its value and hit <*Enter*>.

- Click the *Send & close* button.
- Resume system execution by clicking the *Run* button in SDL simulator window.

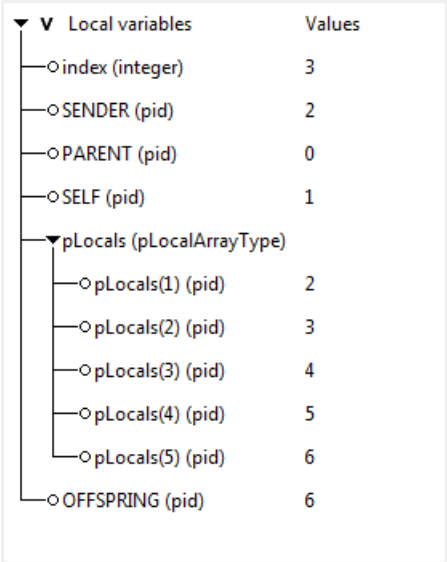
- The following actions appear in the MSC trace:



- When the breakpoint is hit: the SDL editor then pops up and displays the symbol where the execution has stopped:

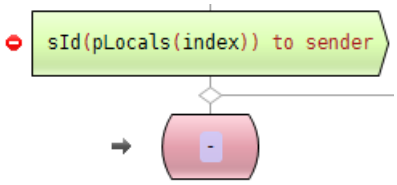


- Since we are in the context of a running process, local variables are automatically displayed in the SDL simulator window. All complex variables such as structs or arrays can be expanded to show their contents. Here are the local variables with the `pLocals` array expanded:

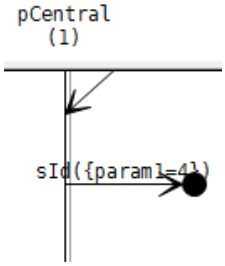


You can see the value for `index` is 3, so the value sent with the `sId` message will be the pid stored at index 3 in `pLocals`, i.e. 4.

- You can also execute instructions line by line in the symbols by using the *Flat step* quick-button: 



(NB: you may have to click on the button twice to go to the next state)
The signal send has been done, as shown in the MSC trace:

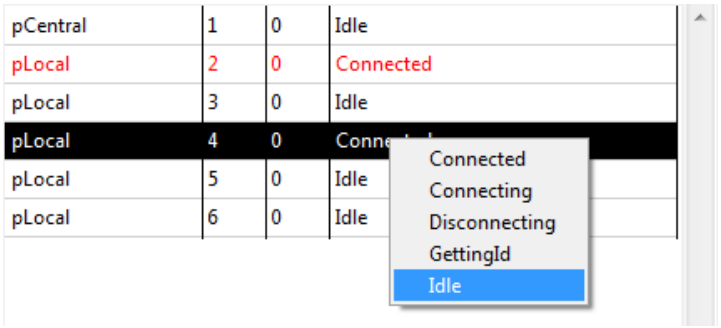


- Let's now finish the system execution by pressing the *Run* button once more.
- When the signal `sCallConf` has been received by `RTDS_Env` in the MSC trace, stop system execution with button 

The SDL states for all running processes are updated in the simulator window:

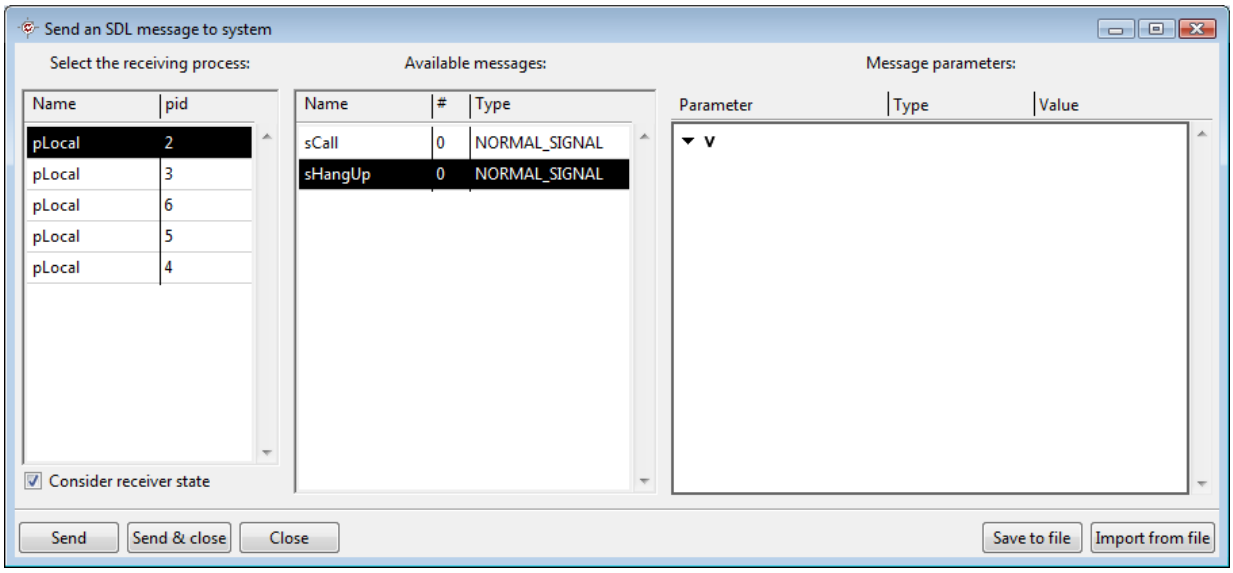
Name	Pid	Sig	SDL state
pCentral	1	0	Idle
pLocal	2	0	Connected
pLocal	3	0	Idle
pLocal	4	0	Connected
pLocal	5	0	Idle
pLocal	6	0	Idle

- The SDL state of a process can be dynamically changed using the contextual menu in the process list:



Some caution is required with this feature, since it may have unexpected results on the system behavior…

- We will now disconnect the two connected `pLocal` processes by sending another signal. So press TO once more:



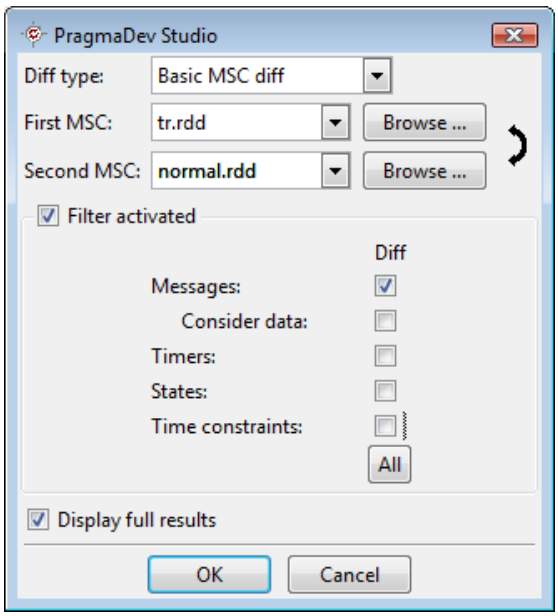
  This will send a `sHangUp` signal to the first `pLocal` (the receiver for our sCall message) with no parameters.
- Send the message with the *Send & close* button.
- We saw that stepping could be done at code line level. There are other step levels including:
  - Step at SDL event level with button
    This button will step one SDL event at a time. Click on it while looking at the MSC trace; you'll see that each time a SDL event happens (signal send, signal receive, process creation, timer start, and so on...), the system execution stops just after the event.
  - Step at transition level with button
    This button will execute a whole transition and stop just after its end (usually the next state symbol). Click on it while looking at the MSC trace; you'll see the active process execute all actions in current transition up to the state change, and the system execution stops.

3.4.4 Verifying the behavior

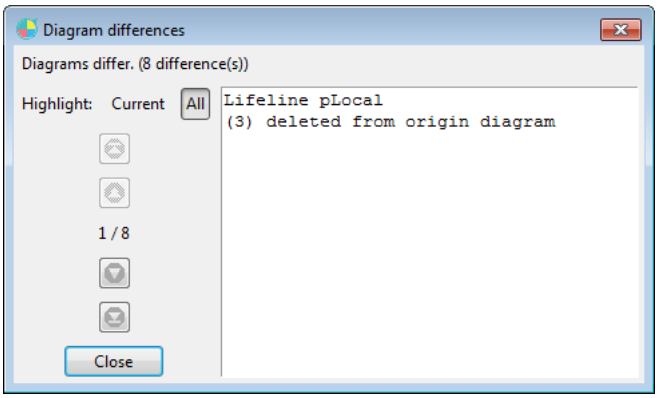
We will now check if the behavior is the one we expected in the first place. To do so we will use the MSC diff feature.

- Make sure the execution is over by clicking button a last time. Then go to the MSC trace window, save the trace and close it.
- Close the simulator window.
- In the project manager, open the trace diagram.

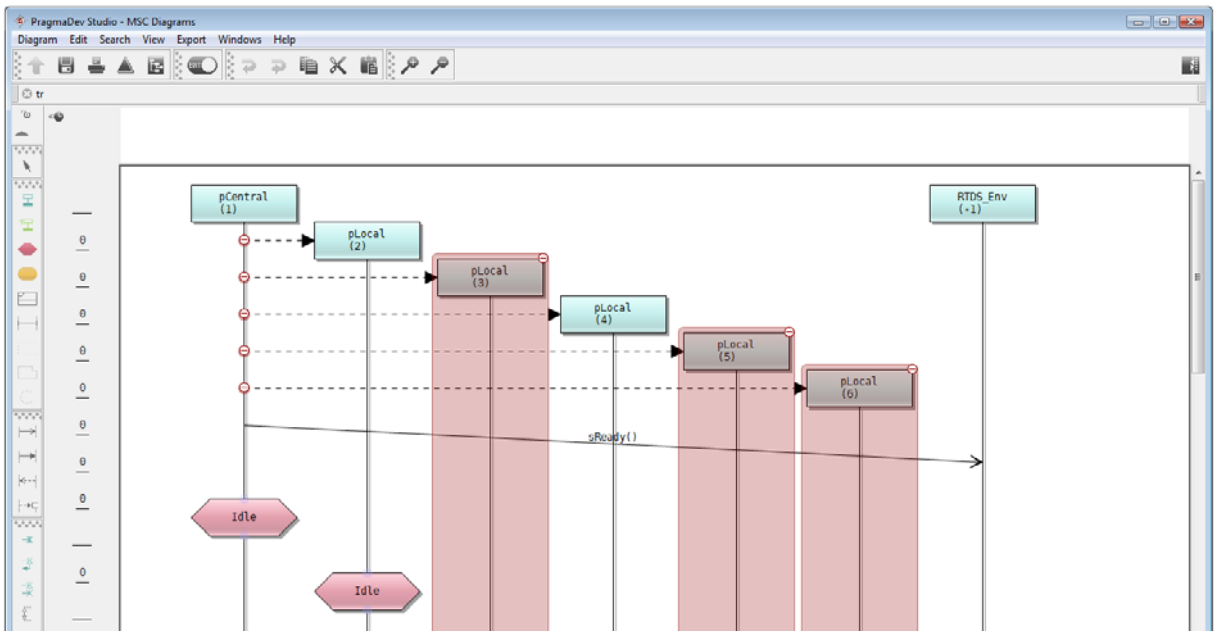
- Go to the *Diagram / Compare with other diagram...* menu to get the MSC Diff configuration window and set it up as described below:



The first MSC is the trace and the second is the `normal` scenario we described in the first place. Since the `normal` MSC was not supposed to be thoroughly detailed we will only show and compare messages without considering their parameters.
Click *OK*; the following window appears:



It allows navigation through the differences between the MSCs. By selecting *All*, then all the diffrences will be shown in the diagram:



The only differences between the MSCs are the dynamic task creation of the `pLocal` instances. After that the exchange of messages are the same between the dynamic trace and the specification. The SDL system therefore conforms to the `normal` MSC specification.

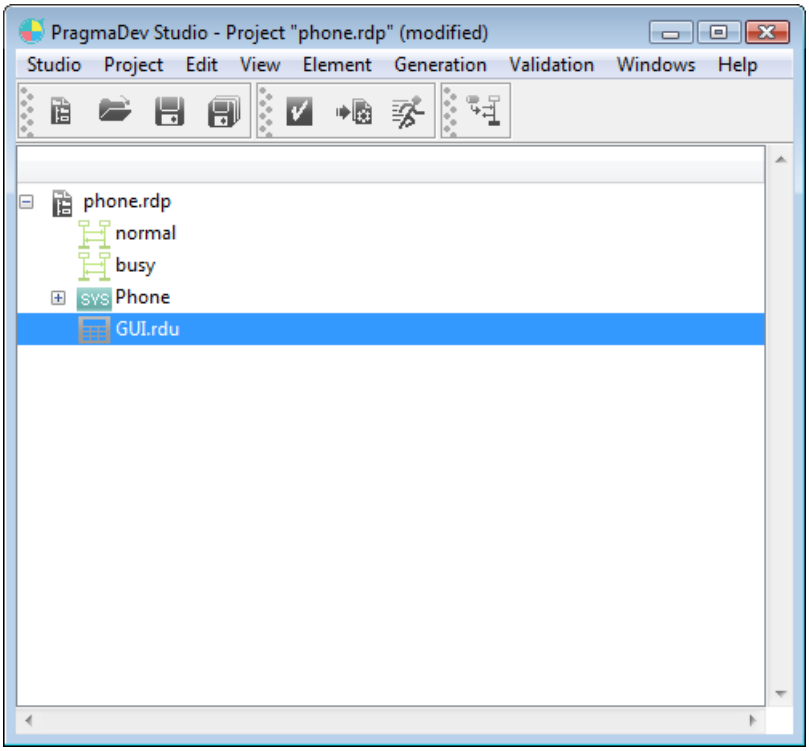
This is the end of this very simple SDL simulation session. There are many areas that have not been covered, such as timers, procedures, external operators, system queue manipulations, watched variables, and so on... You may discover all these features yourself using the examples delivered in PragmaDev Studio distribution or by designing your own system.

3.5 - Prototyping GUI

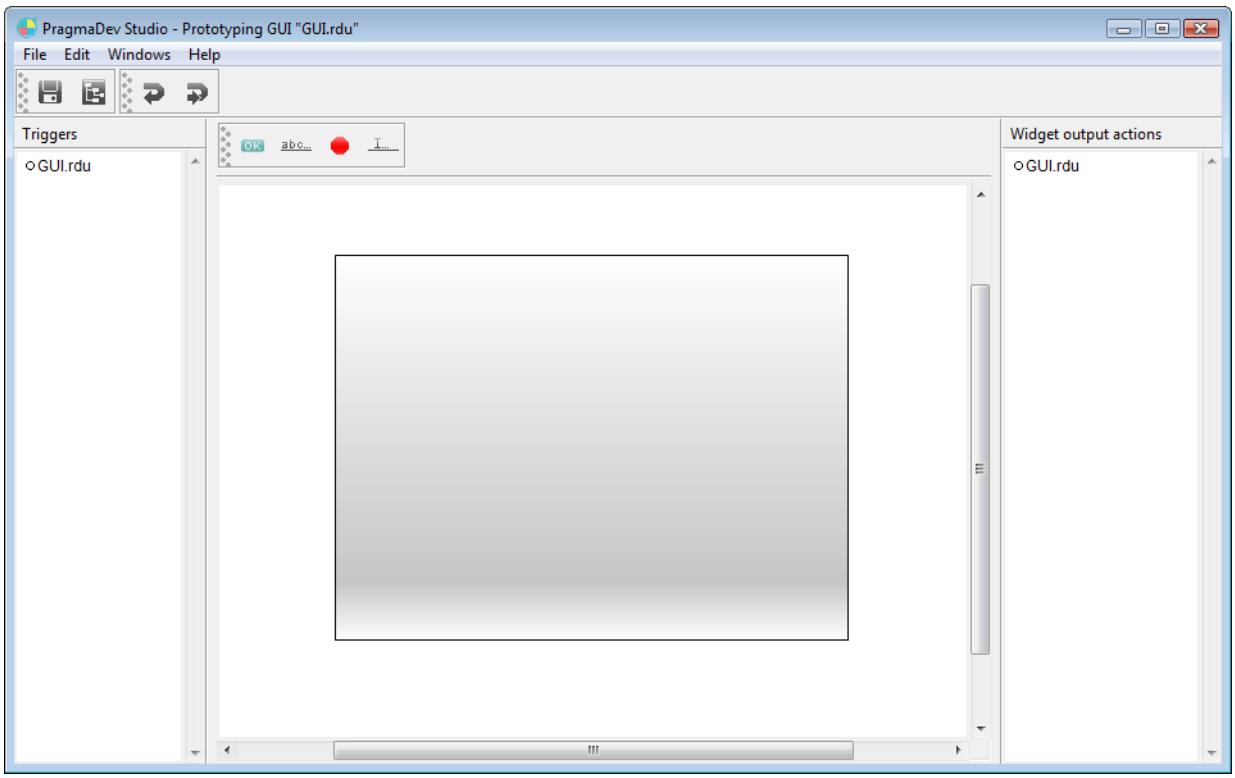
PragmaDev Studio has a built in support to design simple prototyping interface to ease testing. We will build a very simple one for our phone system to demonstrate its capabilities.

3.5.1 GUI editor

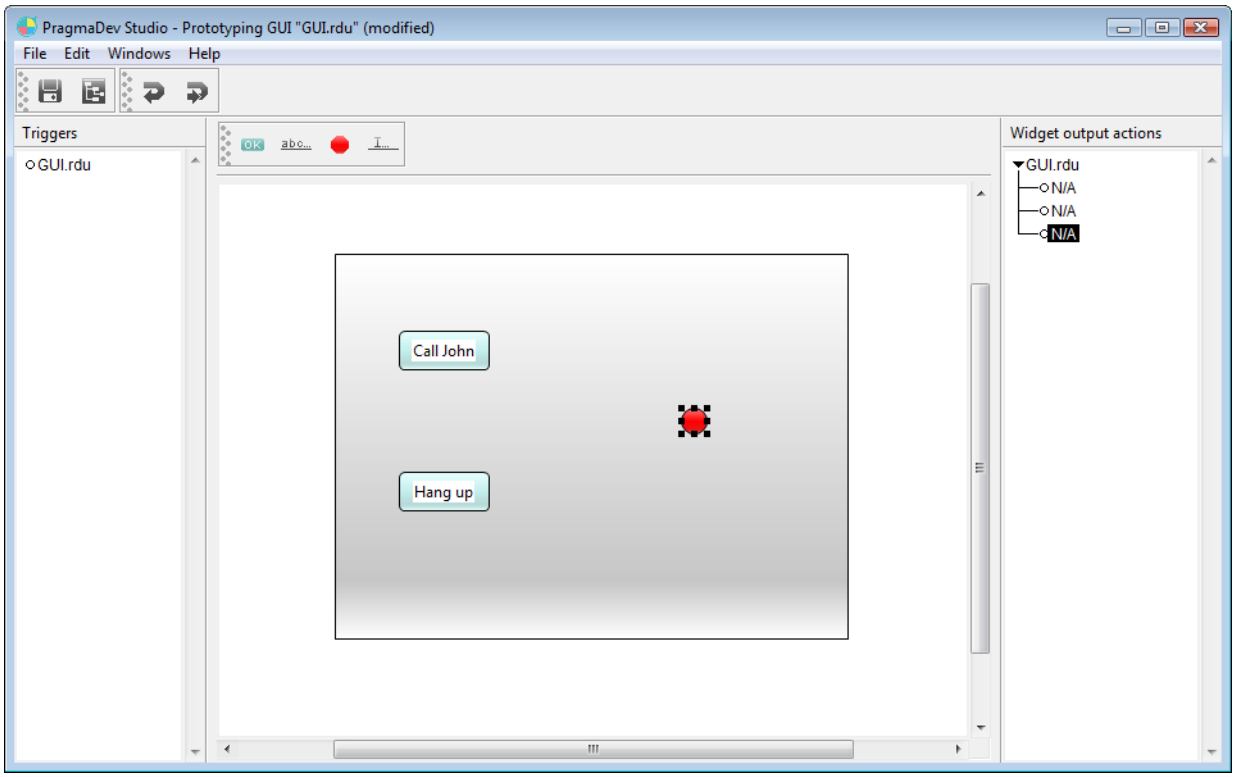
Add a Prototyping GUI node in the project (category *Testing / Validation* in the *Add child element* dialog) and open it:



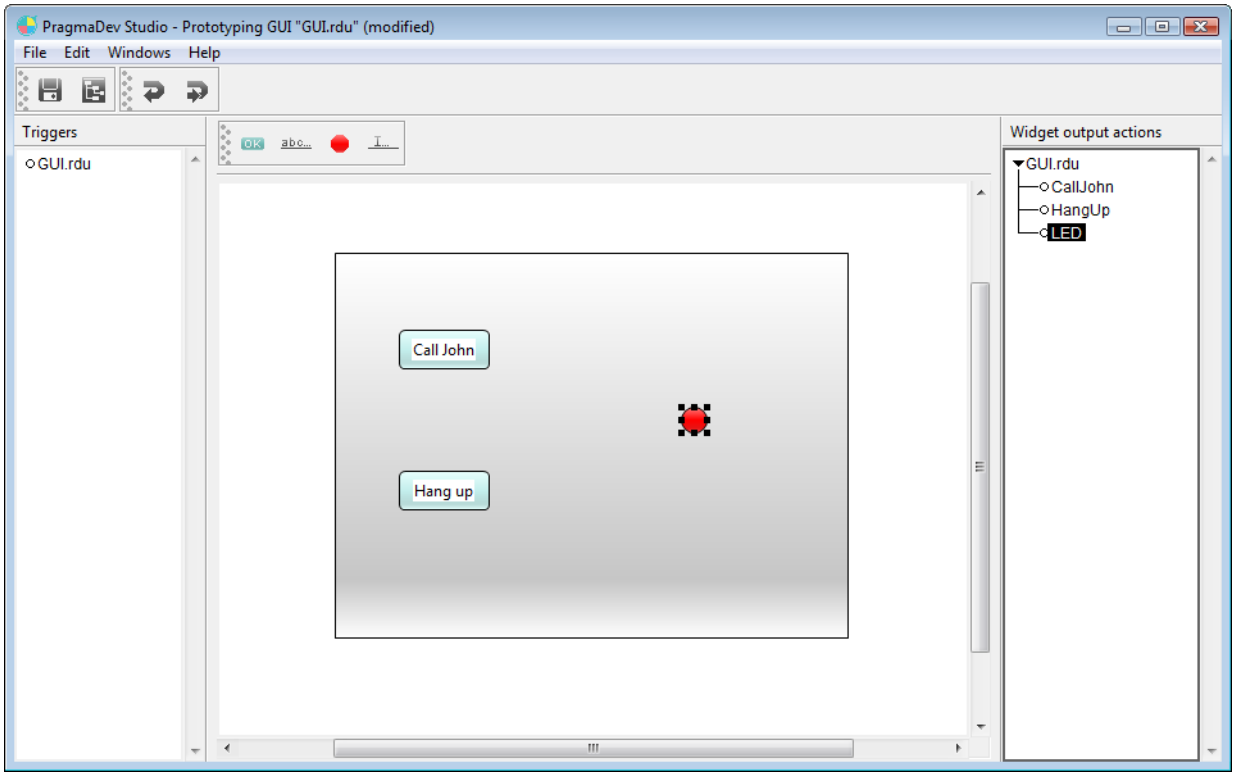
The left panel contains the incoming triggers for the GUI, the central panel the GUI itself, and the right panel the outgoing message from the GUI:



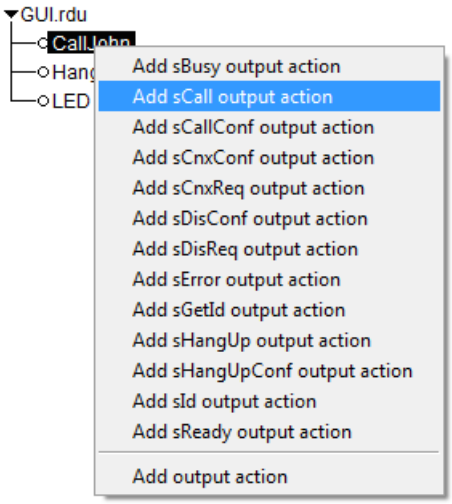
Let's add 2 buttons and one LED:



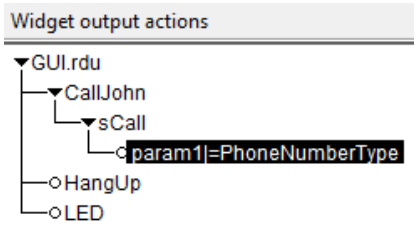
Change their display value in the central panel and their widget name in the right panel in order to recognize them:



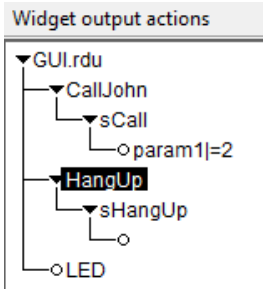
Let's say that when the user clicks on the "Call John" button, the GUI sends an `sCall` message with parameter set to "2". Select the CallJohn widget on the right panel and right click:



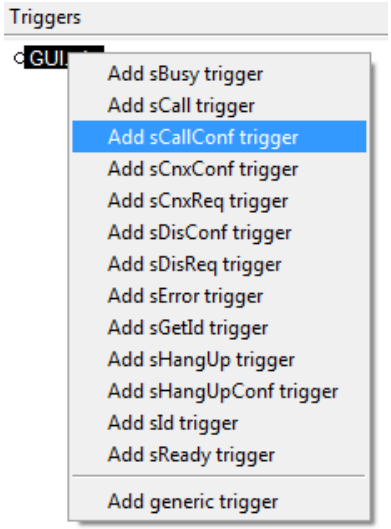
All the available messages in the system are then listed. Select `sCall` and expand the created sub-tree. The parameters are listed with their corresponding type:



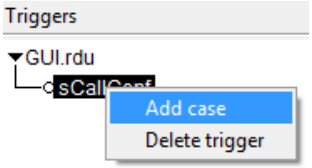
Let's say the parameter value is '2' and let's send `sHangUp` without any parameter when clicking on Hangup:



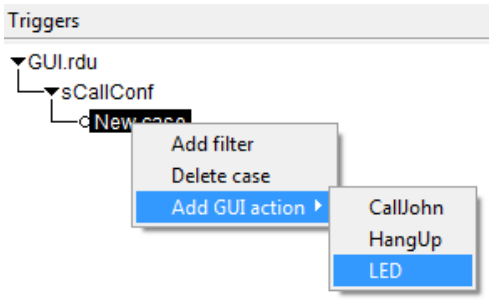
On the left panel now, we will add a new trigger. A trigger performs some action on the widgets whenever a message is sent out of the system. Select the top of the tree and right click to get a list of all the possible triggers:



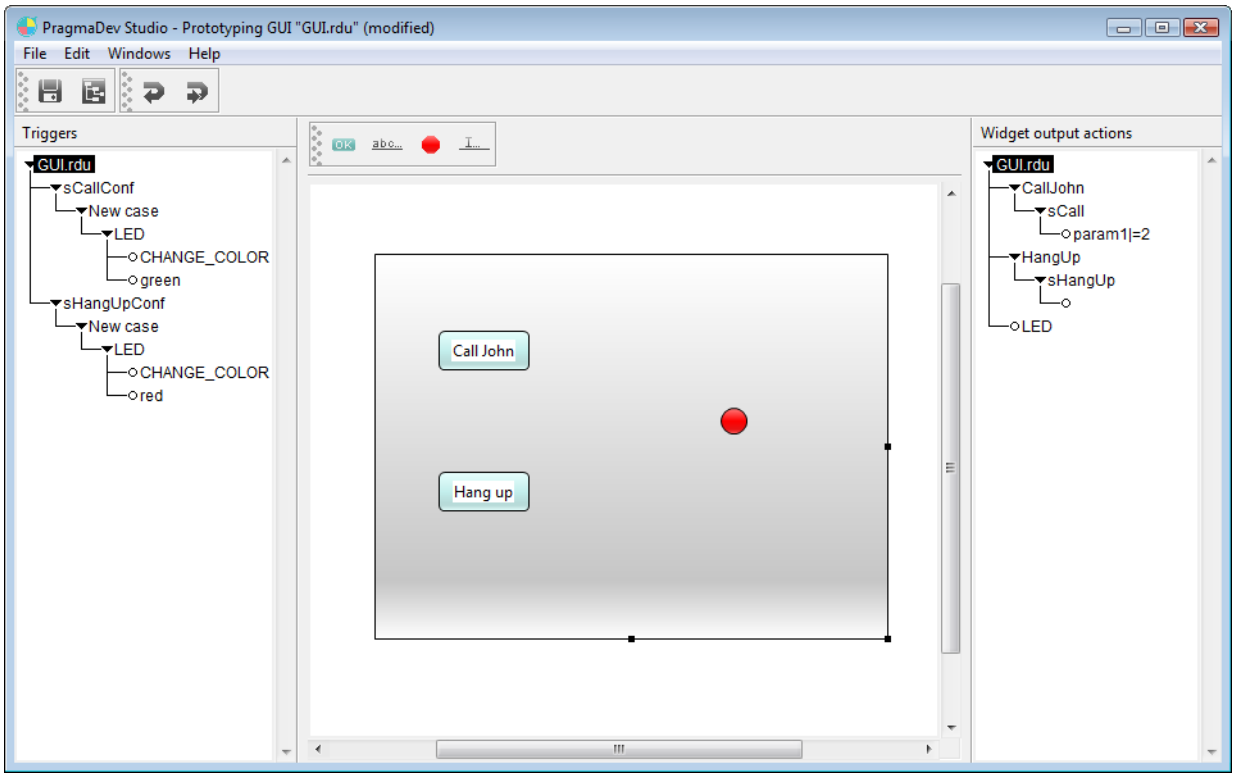
Let's add the `sCallConf` trigger. When a trigger is received by the GUI, a case with a set of filters is verified. Let's add a new case:



In our case we won't put any filter, we will just change the color of the LED:



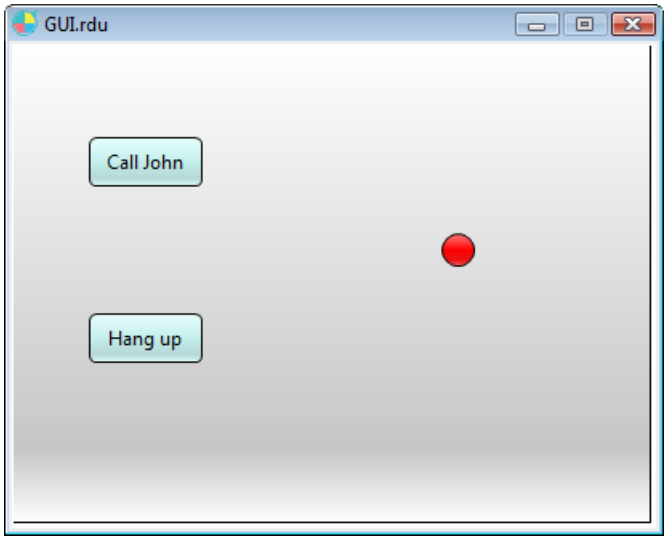
The default action is `DISPLAY`; you'll have to right click on it to change it to `CHANGE_COLOR`. The node in the tree under the action specifies the new color for the widget. It is possible to directly name the basic colors, otherwise the RGB hexa code can be used (e.g #FF8000 is this color). Let's put the LED back to red when we receive a `HangUp` confirmation and we're done:



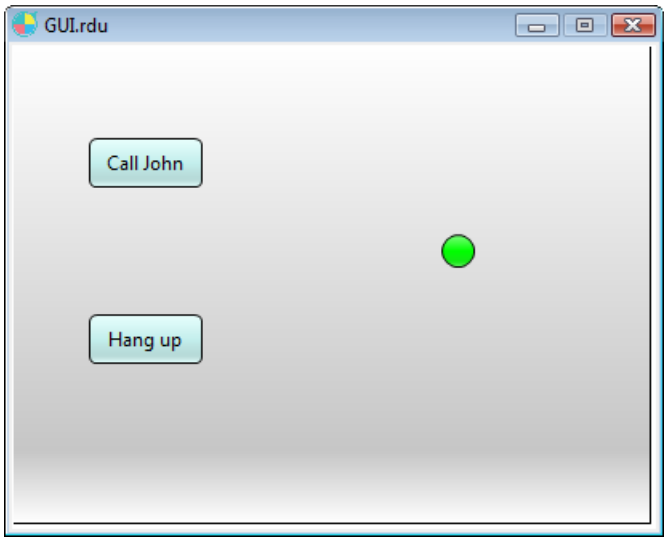
3.5.2 Simulation

Let's start the Simulator again and click on the Start prototyping GUI quick button: 

The GUI will start and connect automatically to the system:



Start an MSC trace and run the system. Click on the "Call John" button, that should send the `sCall` message with parameter value set to `2`, the `sCallConf` should be received by the GUI, and the LED should be set to green:



In practice, this is not a good example because there are a lot of different pLocal processes that could receive the messages sent by the GUI so the receiver is randomly selected.

For a more advanced GUI, please have a look at the AccessControl system in the Specifier example directory.

3.6 - Conclusion

During this tutorial we have been through:

- SDL,
- Project manager,
- SDL architecture & behavioral editors,
- MSC editor,
- SDL simulation including three stepping modes:
  - SDL code line,
  - SDL event,
  - transition,
- Conformance checking,
- Prototyping GUI.

As a result, you saw that SDL is perfectly suited to describe high-level specifications for real time systems. Its complete description of architecture and behavior, including the behavior itself with the Abstract Data Types, allows you to fully describe your system independently from any technical requirement such as the type of target, the RTOS or even the implementation language you'll use.

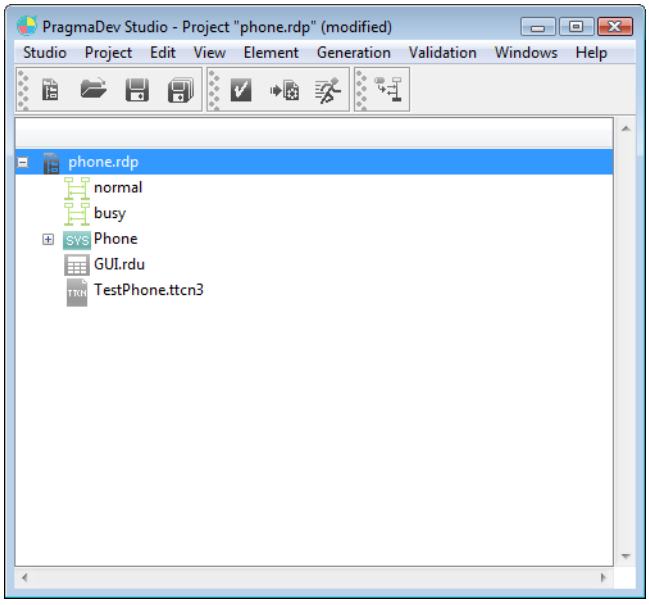
The following chapter will explore some PragmaDev Studio specific features such as testing and code generation.

4 - PragmaDev Studio

This chapter follows the PragmaDev Specifier Tutorial part. Please follow the previous chapter before continuing.

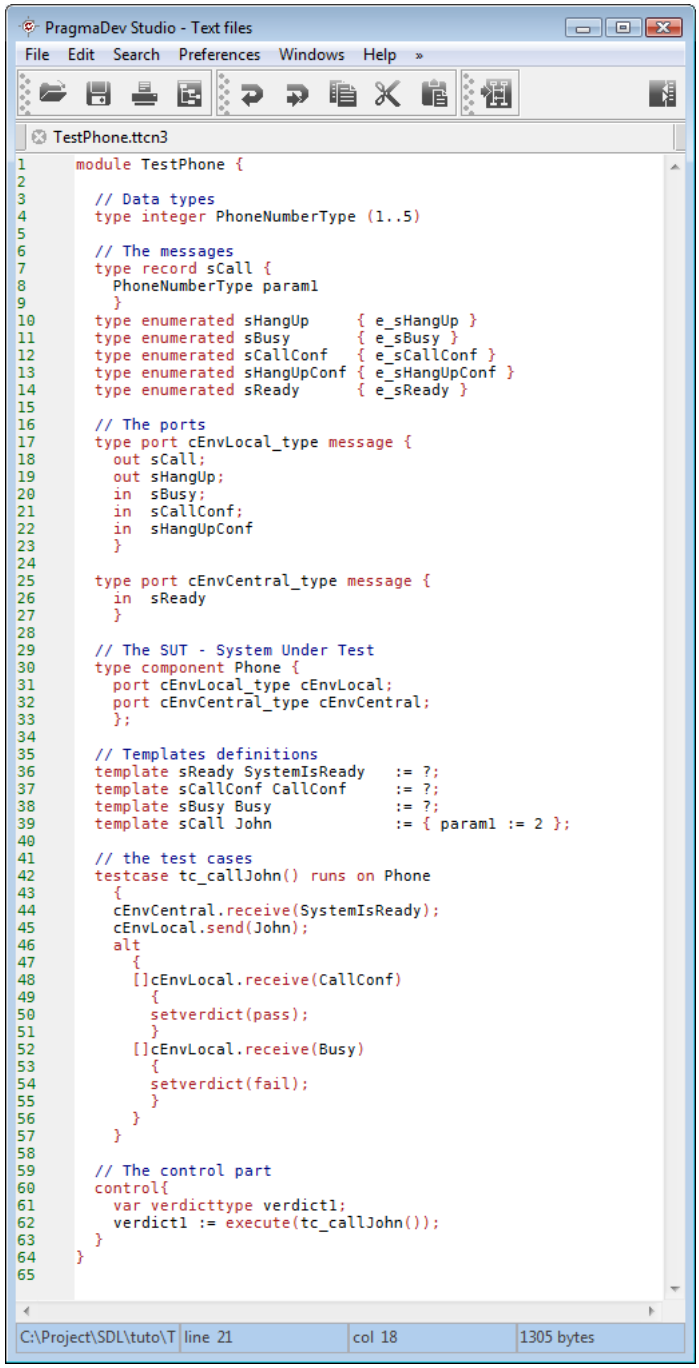
4.1 - Testing

PragmaDev Studio supports TTCN-3 standard testing language for edition and simulation. We will build up a small test case and run it on the phone system we have just designed. Let's add a TTCN-3 compoment to the project (category *Testing / Validation* in the *Add child element* dialog) and name it TestPhone:



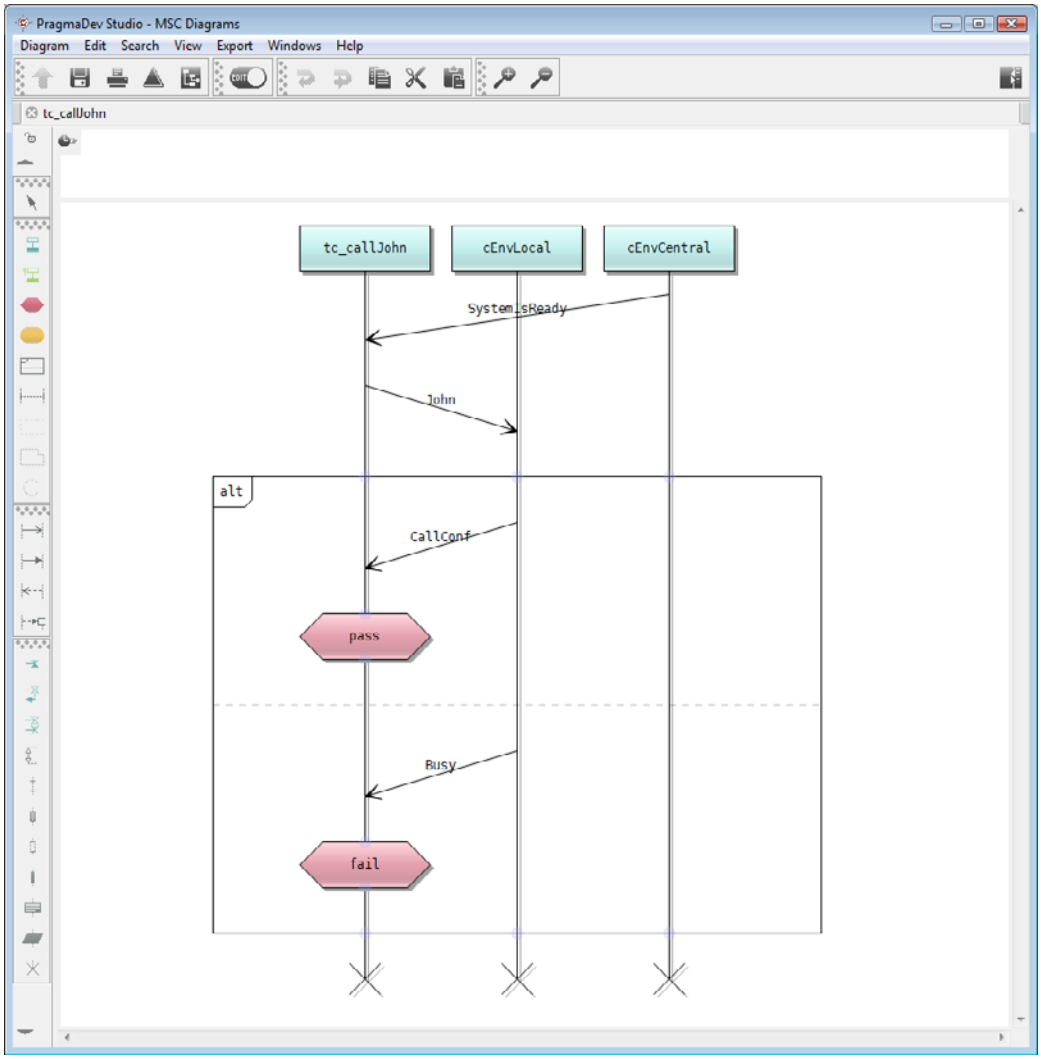
4.1.1 Test case

The text editor recognizes the TTCN-3 syntax so all the keyword will be highlighted. Here is the test suite we will explain in the following paragraphs:




```
module TestPhone {

  // Data types
  type integer PhoneNumberType (1..5)

  // The messages
  type record sCall {
    PhoneNumberType param1
    }
  type enumerated sHangUp     { e_sHangUp }
  type enumerated sBusy       { e_sBusy }
  type enumerated sCallConf   { e_sCallConf }
  type enumerated sHangUpConf { e_sHangUpConf }
  type enumerated sReady      { e_sReady }

  // The ports
  type port cEnvLocal_type message {
    out sCall;
    out sHangUp;
    in  sBusy;
    in  sCallConf;
    in  sHangUpConf
    }

  type port cEnvCentral_type message {
    in  sReady
    }

  // The SUT - System Under Test
  type component Phone {
    port cEnvLocal_type cEnvLocal;
    port cEnvCentral_type cEnvCentral;
    };

  // Templates definitions
  template sReady SystemIsReady   := ?;
  template sCallConf CallConf     := ?;
  template sBusy Busy             := ?;
  template sCall John             := { param1 := 2 };

  // the test cases
  testcase tc_callJohn() runs on Phone
    {
    cEnvCentral.receive(SystemIsReady);
    cEnvLocal.send(John);
    alt
      {
      []cEnvLocal.receive(CallConf)
        {
        setverdict(pass);
        }
      []cEnvLocal.receive(Busy)
        {
        setverdict(fail);
        }
      }
    }

  // The control part
  control{
    var verdicttype verdict1;
    verdict1 := execute(tc_callJohn());
  }
}
```

To see a MSC representation of the TTCN-3 tescase behaviour, click on the `View graphical representation` button :



This MSC representation is not editable.

Since TTCN aims at testing complex systems, it is strongly structured. We first need to define the data types we will be using in our test case, define the interfaces with the system, and the value templates that will be exchanged.

4.1.1.1 Declarations

The messages exchanged between the system and the environment are the ones listed in the channels connect to the frame of the system. Most of the message exchanged with the system have no parameters except sCall. sCall takes an integer sub-type as a parameter we will re-define here:

```
// Data types
type integer PhoneNumberType (1..5)
```

There is no message or signal specific type in TTCN, if the message has parameters it is defined as a record, if it has no parameters, it is defined as an enumerated with a single possible value:

```
// The messages
type record sCall {
  PhoneNumberType param1
  }
type enumerated sHangUp     { e_sHangUp }
type enumerated sBusy       { e_sBusy }
type enumerated sCallConf   { e_sCallConf }
type enumerated sHangUpConf { e_sHangUpConf }
type enumerated sReady      { e_sReady }
```

4.1.1.2 Ports

TTCN-3 can test asynchronous systems, synchronous systems, or a combination of both. In our tutorial example only asynchronous messages are exchanged with the system. We will define a port for each channel in the system representing the 2 interfaces:

```
// The ports
type port cEnvLocal_type message {
  out sCall;
  out sHangUp;
  in  sBusy;
  in  sCallConf;
  in  sHangUpConf
  }

type port cEnvCentral_type message {
  in  sReady
  }
```

We will now define the component we will be testing, that is the system itself. The name of the component must be the name of the SDL system.

```
// The SUT - System Under Test
type component Phone {
  port cEnvLocal_type cEnvLocal;
  port cEnvCentral_type cEnvCentral;
  };
```

4.1.1.3 Templates

When exchanging messages with the system, the values of the parameters of the messages must be pre-defined. These values are called templates. Templates are used to both:

- define the values of the outgoing messages parameters,
- to verify the values of the received messages parameters are correct.

In our phone example, the messages coming from the system do not have any parameter, only the sCall message has one parameter. Still we need to define templates for all the message we will exchange with the system.

```
// Templates definitions
template sReady SystemIsReady   := ?;
template sCallConf CallConf     := ?;
template sBusy Busy             := ?;
```

```
  template sCall John             := { param1 := 2 };
```

The John template will set the parameter of sCall to '2'.

4.1.1.4 Core test case

The core test case is the execution part, the scenario itself. To make it simple, we will first wait until the system sends the ready message on the cEnvCentral port, then we will call John and wait for the answer. An alternative is created: the call is confirmed and we will consider the test pass, or John is busy and we will consider the test fail:

```
// the test cases
  testcase tc_callJohn() runs on Phone
    {
    cEnvCentral.receive(SystemIsReady);
    cEnvLocal.send(John);
    alt
      {
      []cEnvLocal.receive(CallConf)
        {
        setverdict(pass);
        }
      []cEnvLocal.receive(Busy)
        {
        setverdict(fail);
        }
      }
    }
```

4.1.1.5 Control part

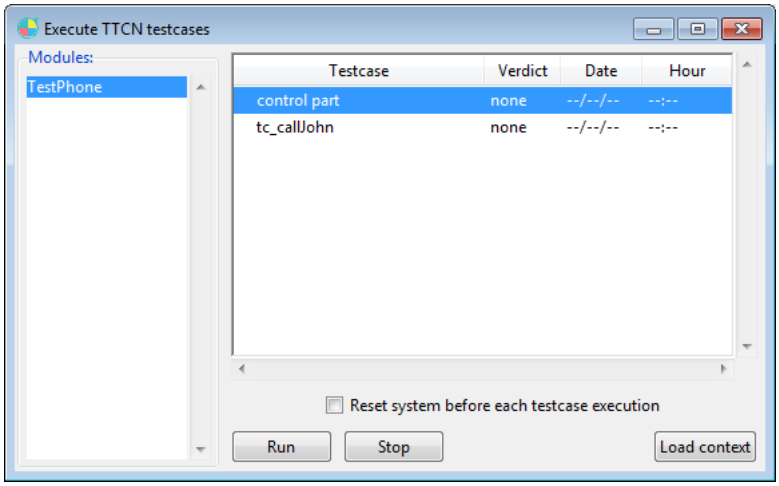
The control part is what will be executed, this is where you combine which test case you would like to run on the system. Please note there might be a different verdict for each test case. This won't stop the execution of the control part. In our tiny example we will just run the unique control part we wrote:

```
  // The control part
  control{
    var verdicttype verdict1;
    verdict1 := execute(tc_callJohn());
  }
```

We're done with our test case, let's now run the test on the system and see what it does.

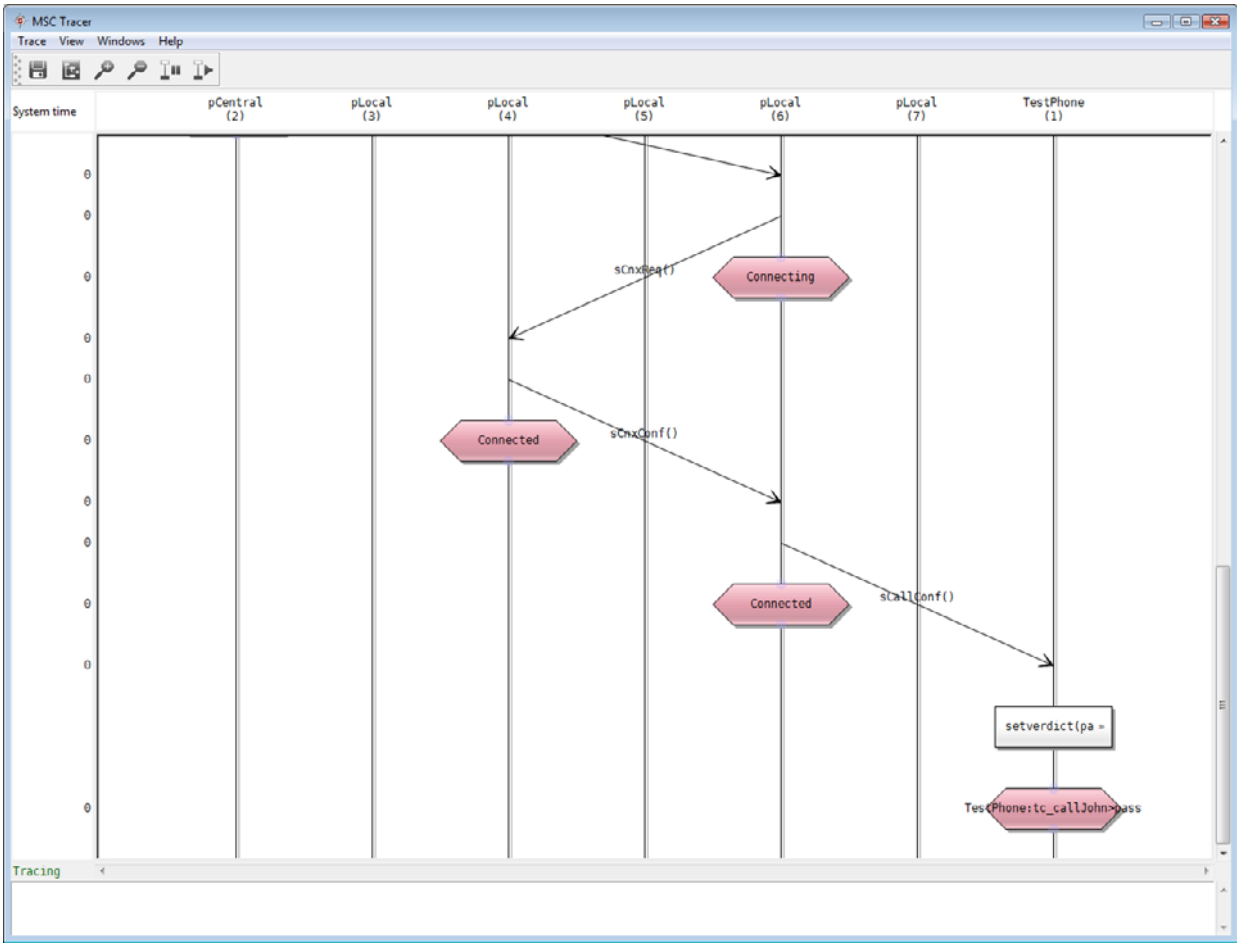
4.1.2 Simulation against the SDL system

Select the test suite in the project manager and click on the Debug button. In addition to the PragmaDev Simulator, the following window will open:



Please note it is possible to set breakpoints in the test case as well as in the SDL system.

Start an MSC trace, select the control part, and Run. The scenario will execute by itself and the verdict is displayed in the shell and in the MSC trace:



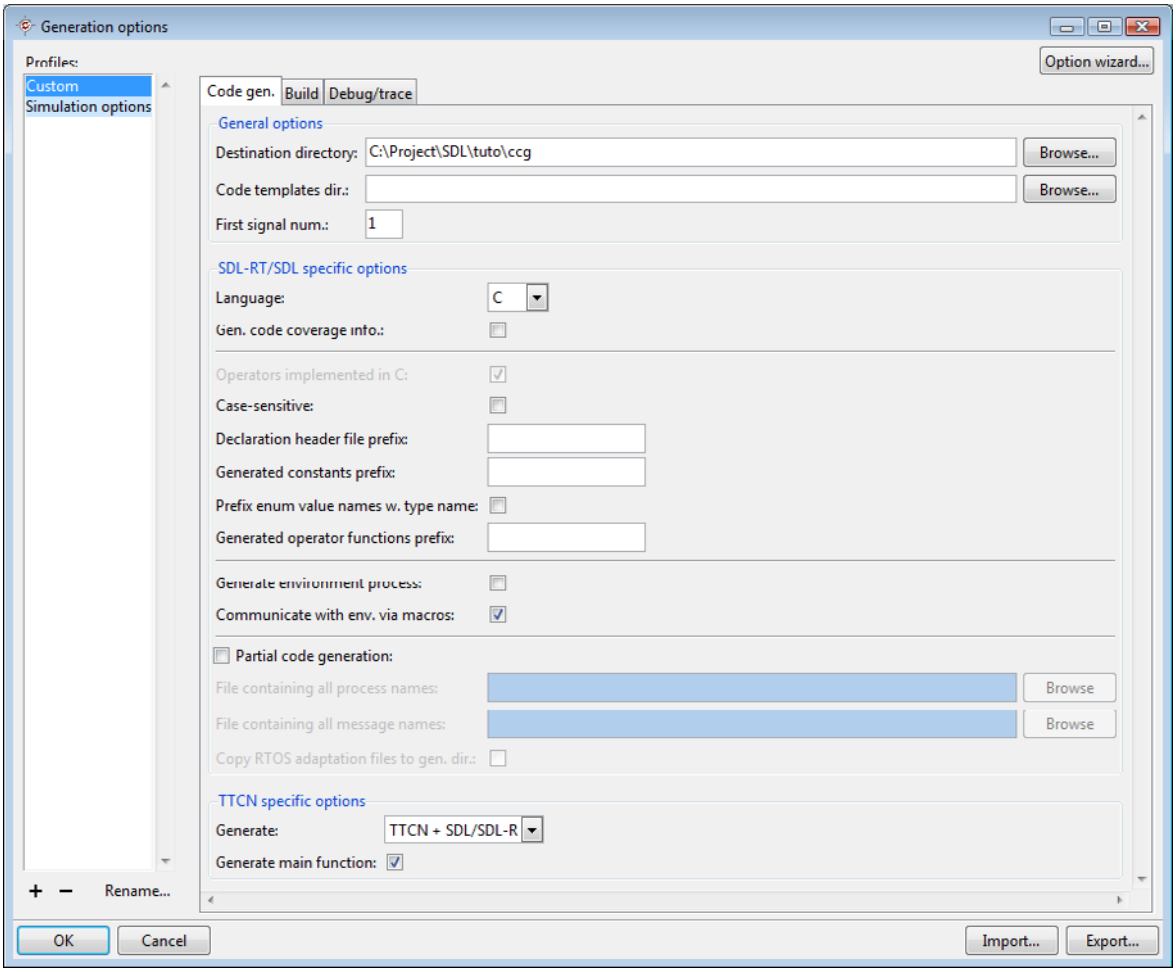
Because there are several instances of pLocal it might happen that the test case calls itself. In that case, run the scenario again for the test to pass.

4.2 - Code generation

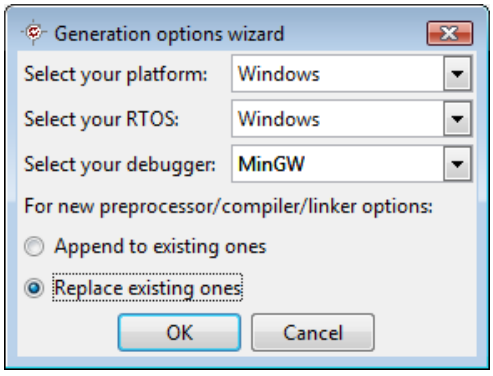
4.2.1 Code generation options

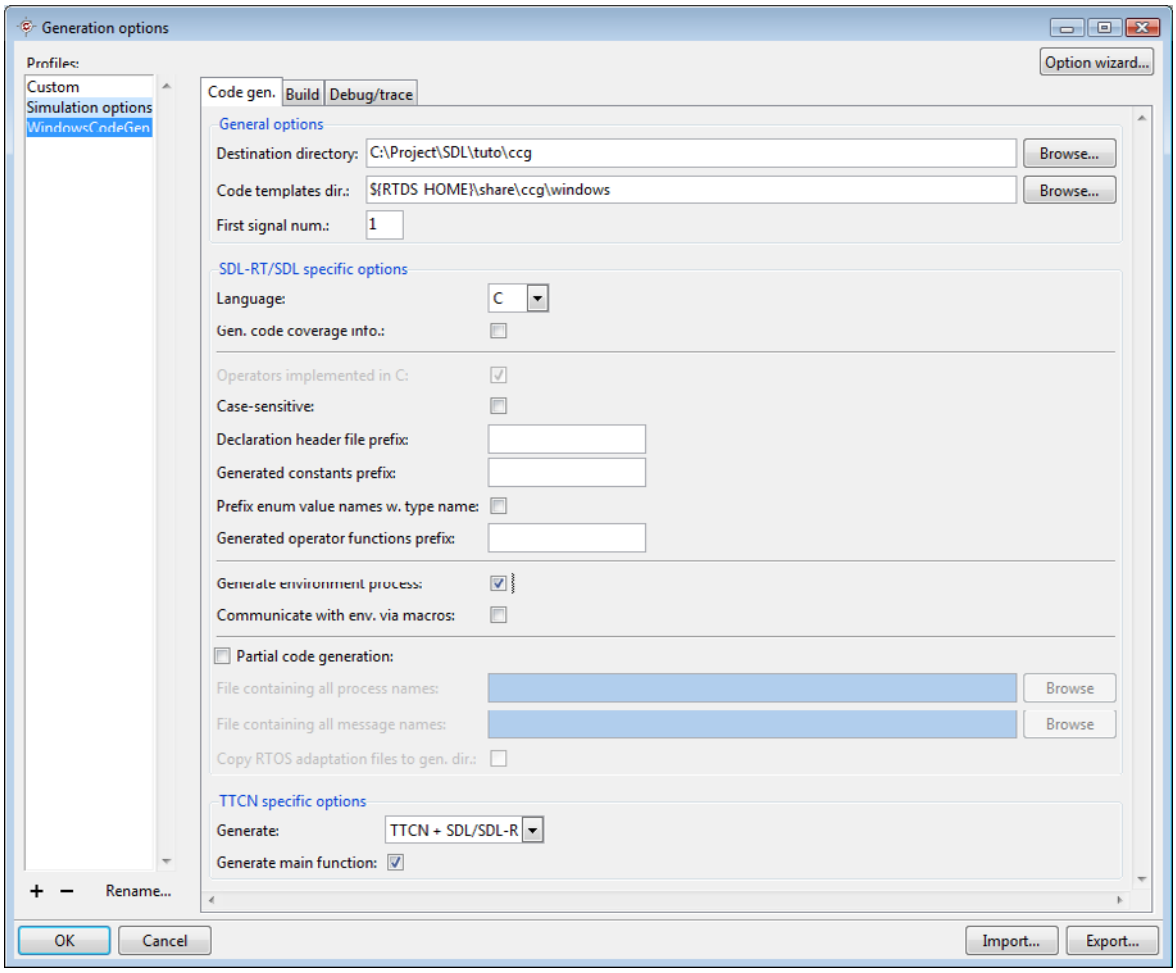
It is also possible to generate C code out of the SDL system in order to implement it on a real target. For this chapter, it is necessary to have a supported debugger installed such as gdb or MinGW.

The generation profiles are listed in *Generation / Options...*

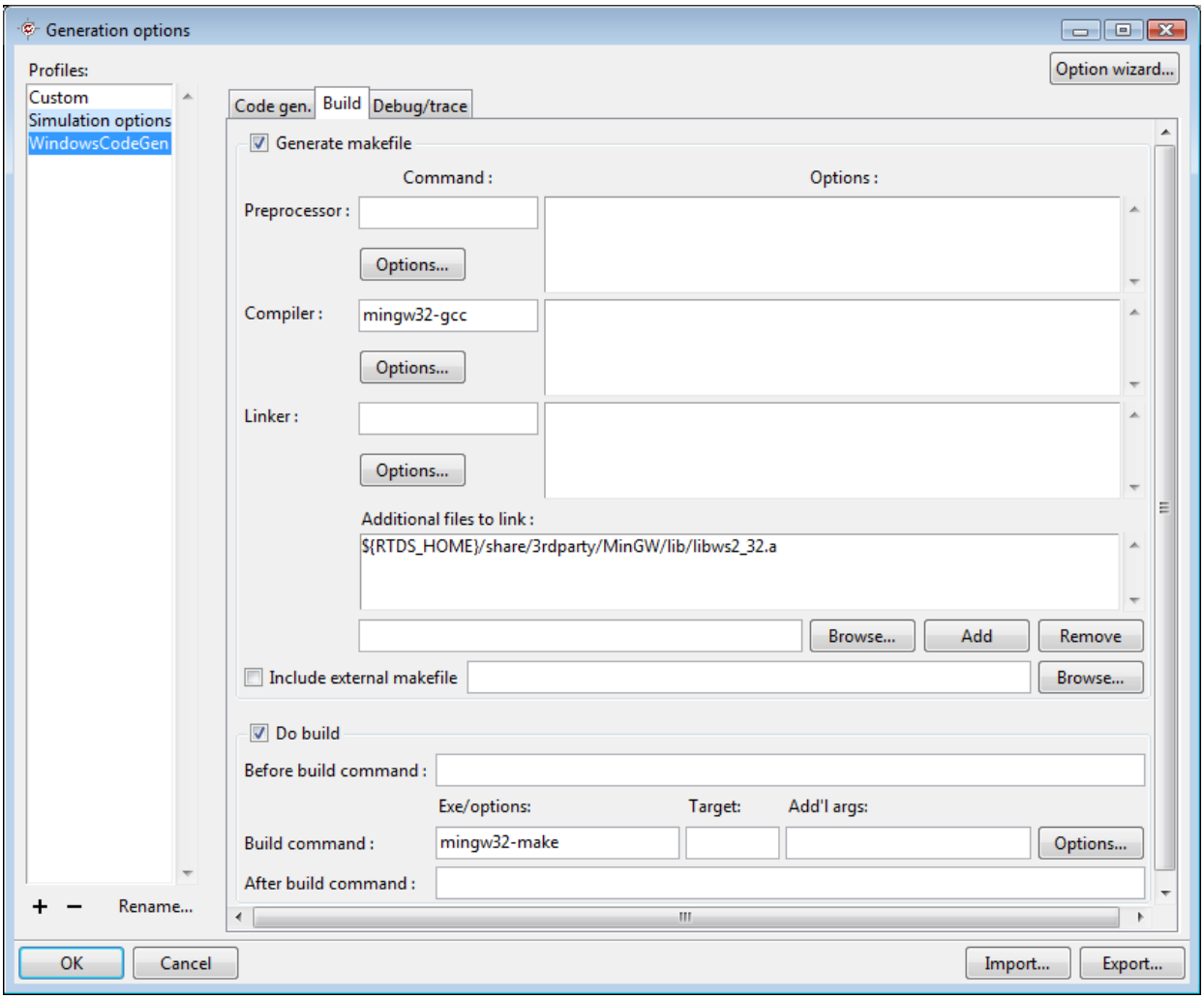


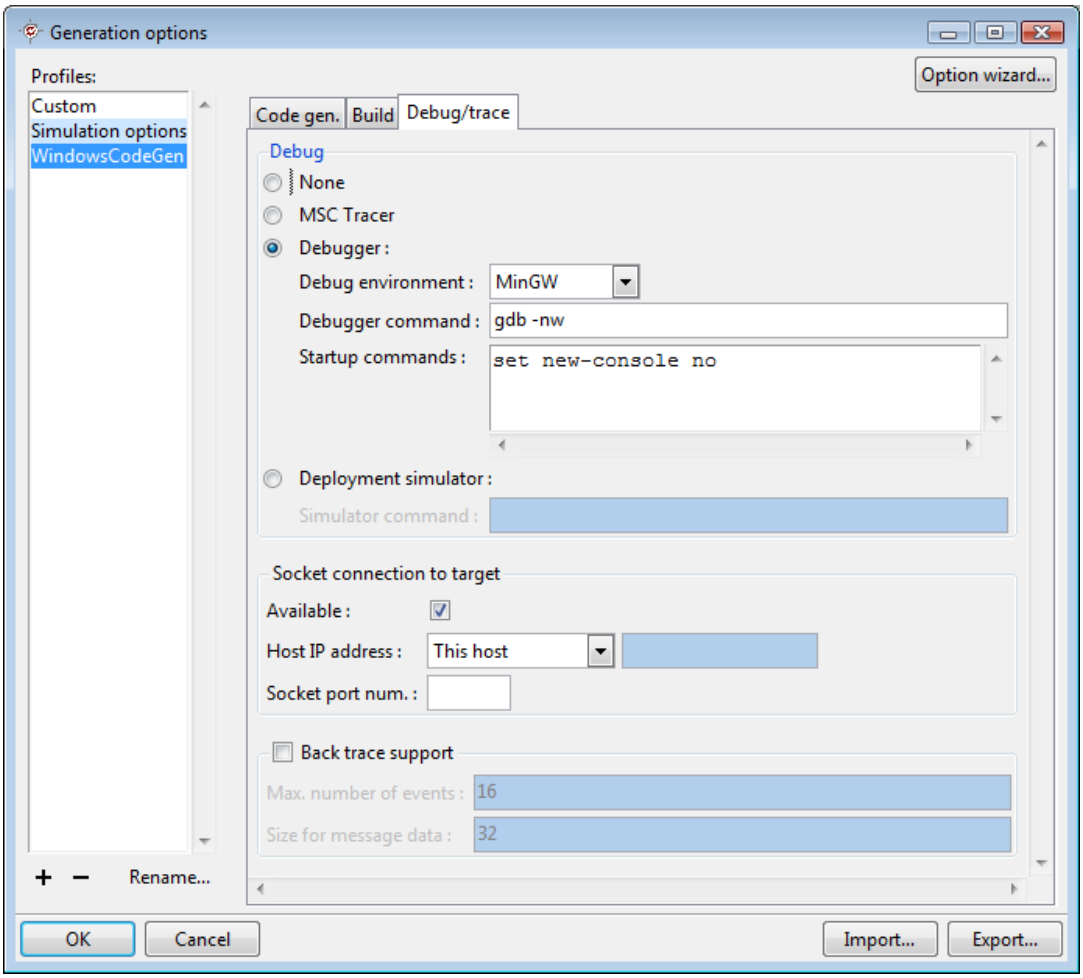
The directory where the C files will be generated is automatically set to the ccg subdirectory under the project directory. Now we'll use the *Option wizard...* to quickly create a valid profile such as described below:




Generation options
Profiles:
Custom
Simulation options
WindowsCodeGen
Option wizard...
Code gen.
Build
Debug/trace
General options
Destination directory:
C:\Project\SDL\tuto\ccg
Browse...
Code templates dir.:
${RTDS_HOME}\share\ccg\windows
Browse...
First signal num.:
1
SDL-RT/SDL specific options
Language:
C
Gen. code coverage info.:
Operators implemented in C:
Case-sensitive:
Declaration header file prefix:
Generated constants prefix:
Prefix enum value names w. type name:
Generated operator functions prefix:
Generate environment process:
Communicate with env. via macros:
Partial code generation:
File containing all process names:
Browse
File containing all message names:
Browse
Copy RTOS adaptation files to gen. dir.:
TTCN specific options
Generate:
TTCN + SDL/SDL-R
Generate main function:
+
−
Rename...
OK
Cancel
Import...
Export...

Make sure the "Generate environment process" box is checked.

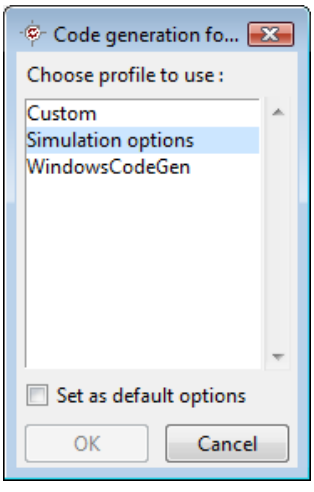




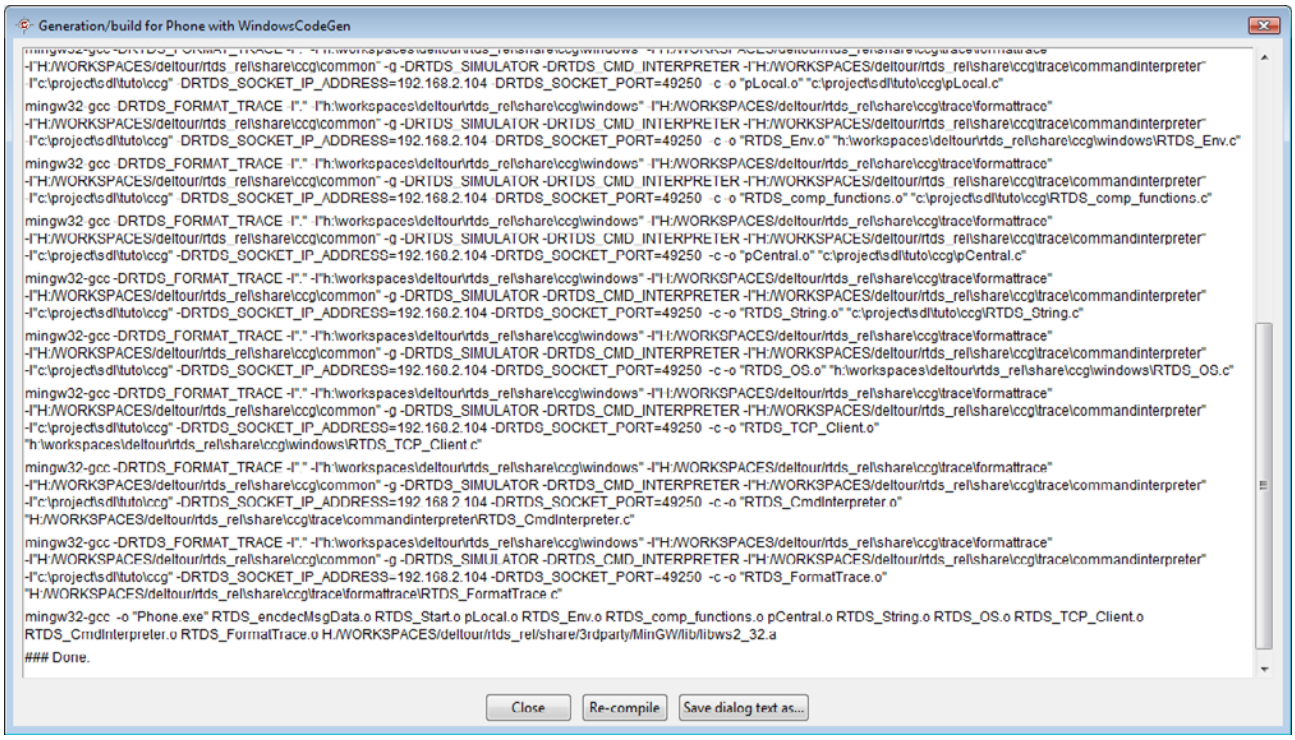
4.2.2 Graphical debugging

Once the SDL debug profile is properly defined select the SDL system in the *Project manager* and click on the *Debug* quick button in the tool bar: 

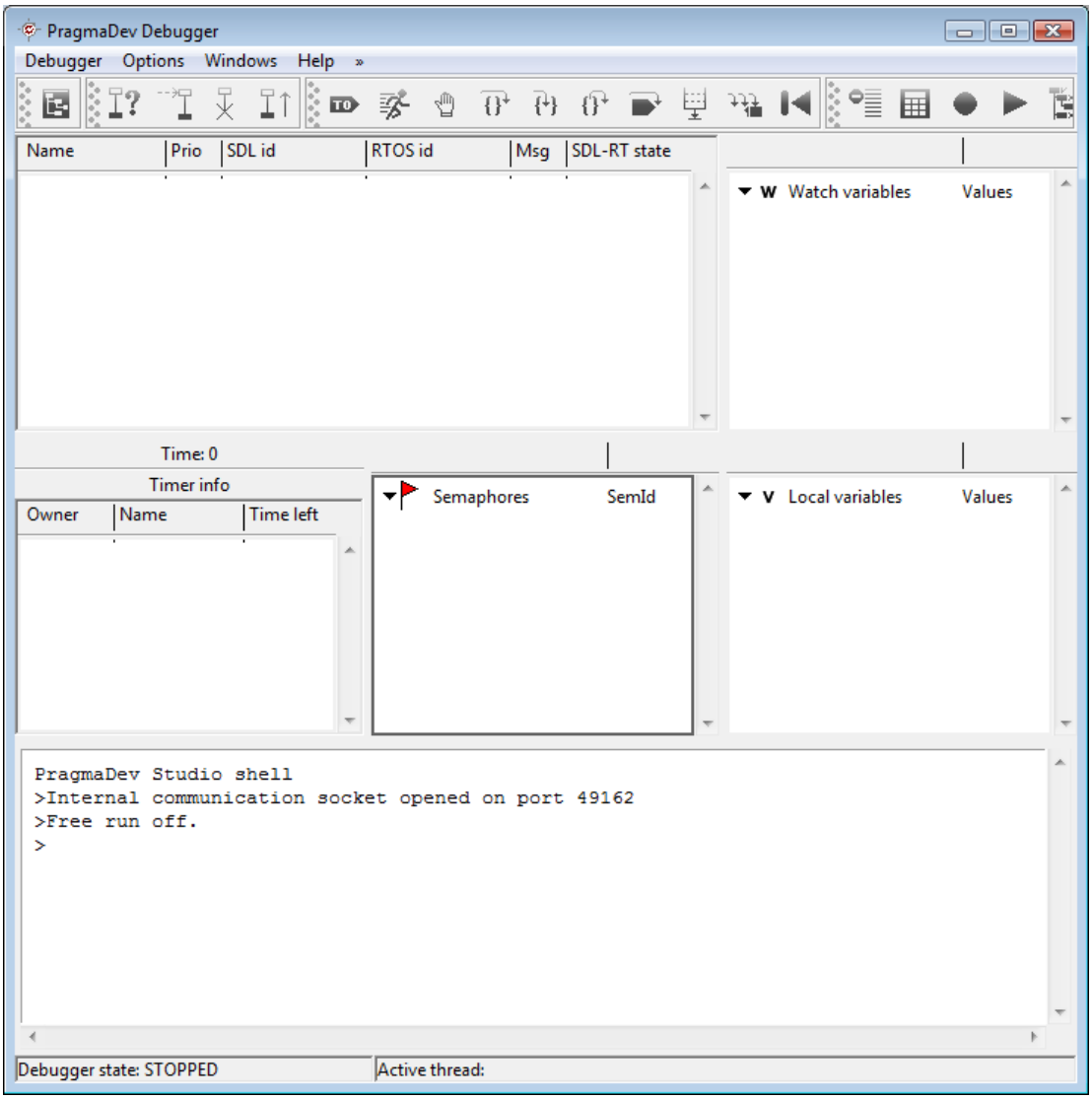
Since several execution profiles are defined, a window pops up asking for the profile you want to work with:



Select *C code generation* and click *Ok*. The code will be generated, compiled, and the debugger will be started automatically :



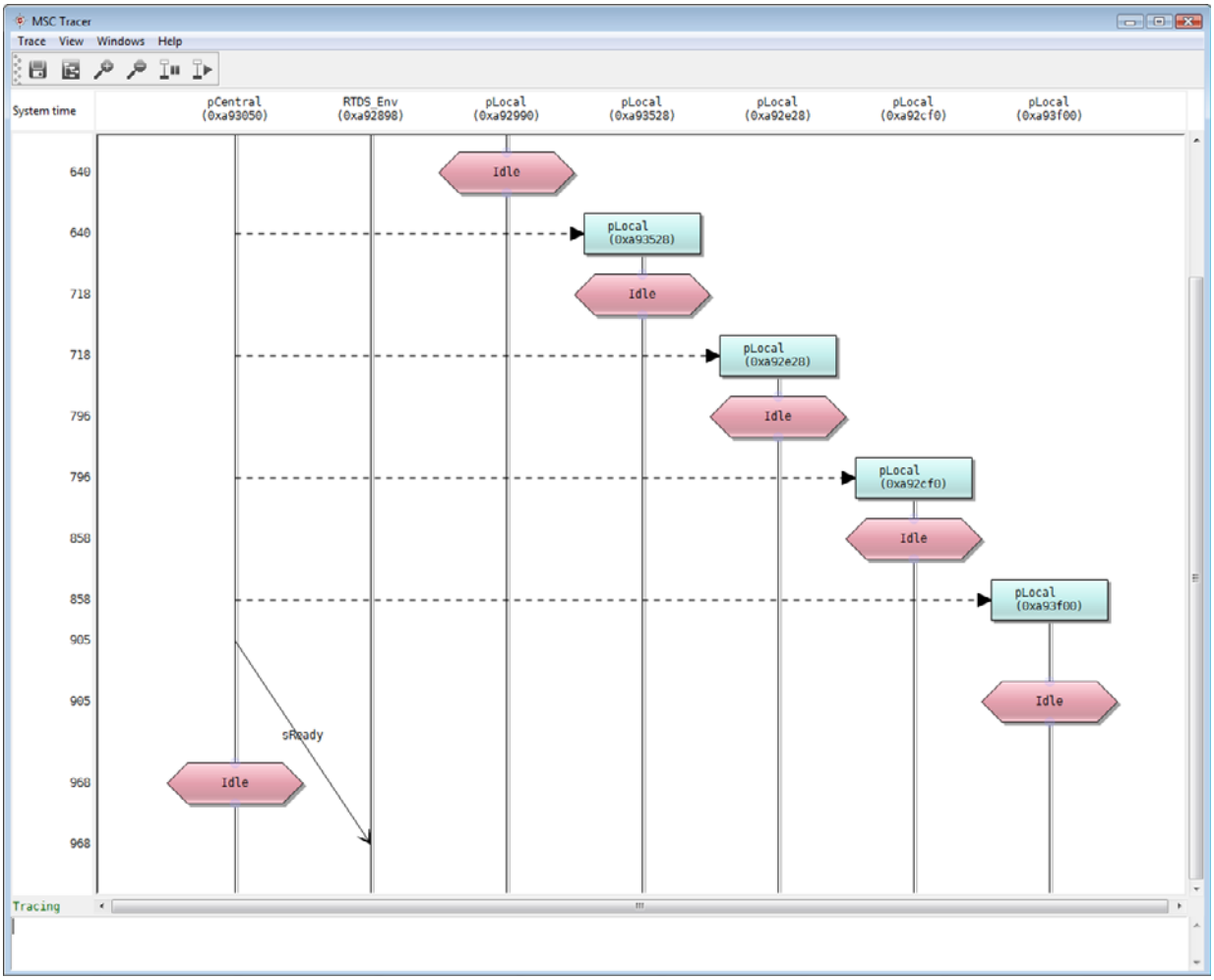
The debugger interface looks pretty much like the simulator one :



Click on the *Start MSC trace* quick button: 

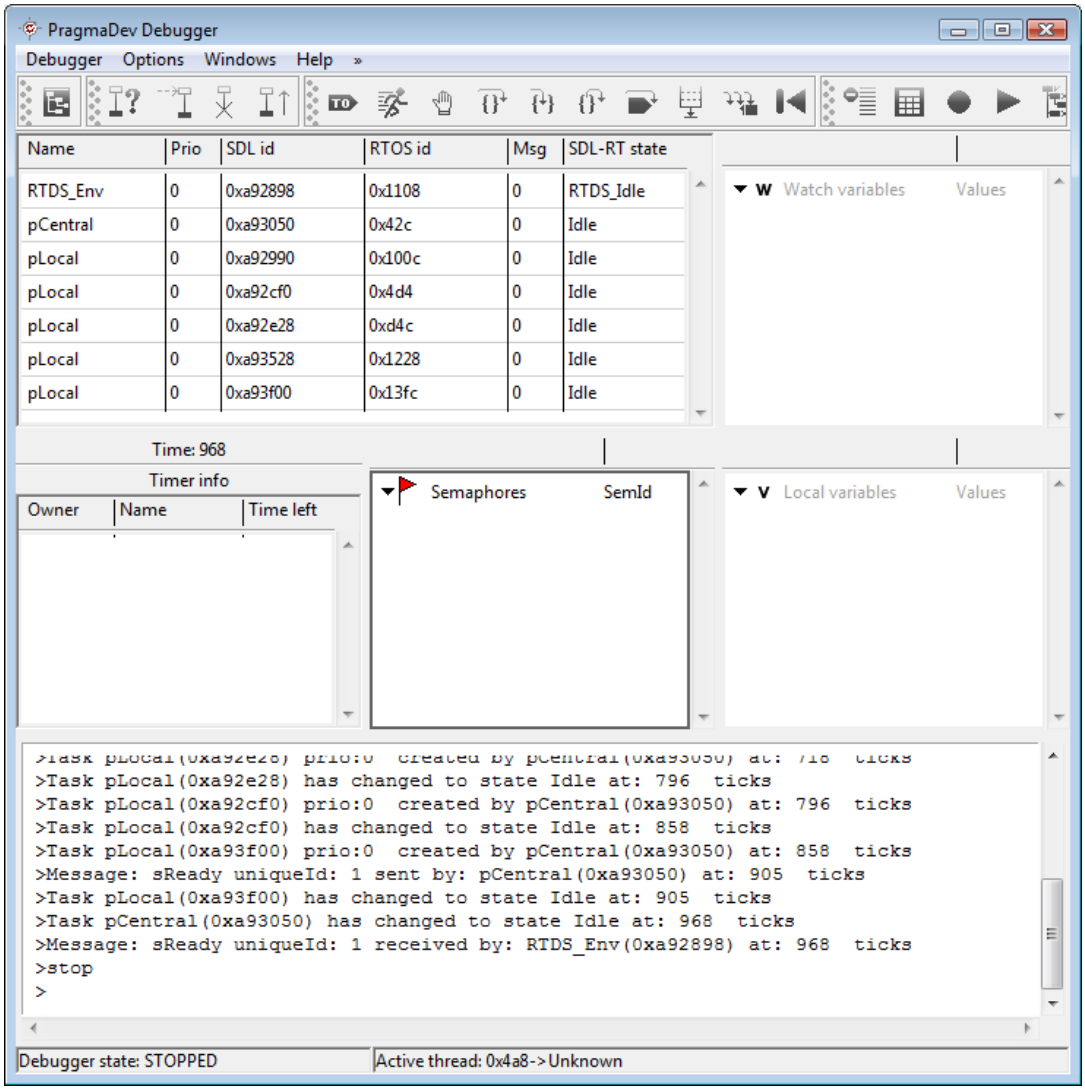
An *MSC Tracer* window appears. Let's start the system; click on 'run' quick button: 

The MSC Trace will look like the one below :



Click on the Stop button to break execution:

The SDL debugger window shows the list of processes with their names, priority, process id, queue id, number of messages in their respective message queues, SDL internal state as we defined in the diagrams, and the RTOS internal system state if available.



As with the SDL Simulator, it is possible to set breakpoints, view the value of variables, send messages... Would you like to know more about graphical debugging of C code, we strongly suggest to go through the PragmaDev Developer tutorial as the debugging features are the same.

4.3 - Conclusion

During this tutorial we have been through:

- Test of an SDL system,
- C code generation from an SDL system.

When it comes to design, the system will actually be implemented on a given target, so additional requirements will have to be taken into account:

- Use of legacy code and/or external libraries, usually written in C with no way to manipulate SDL's high-level abstract data types;
- Support for additional concepts, such as semaphores or pointers, unneeded in an SDL description, but usually required in real-time systems.

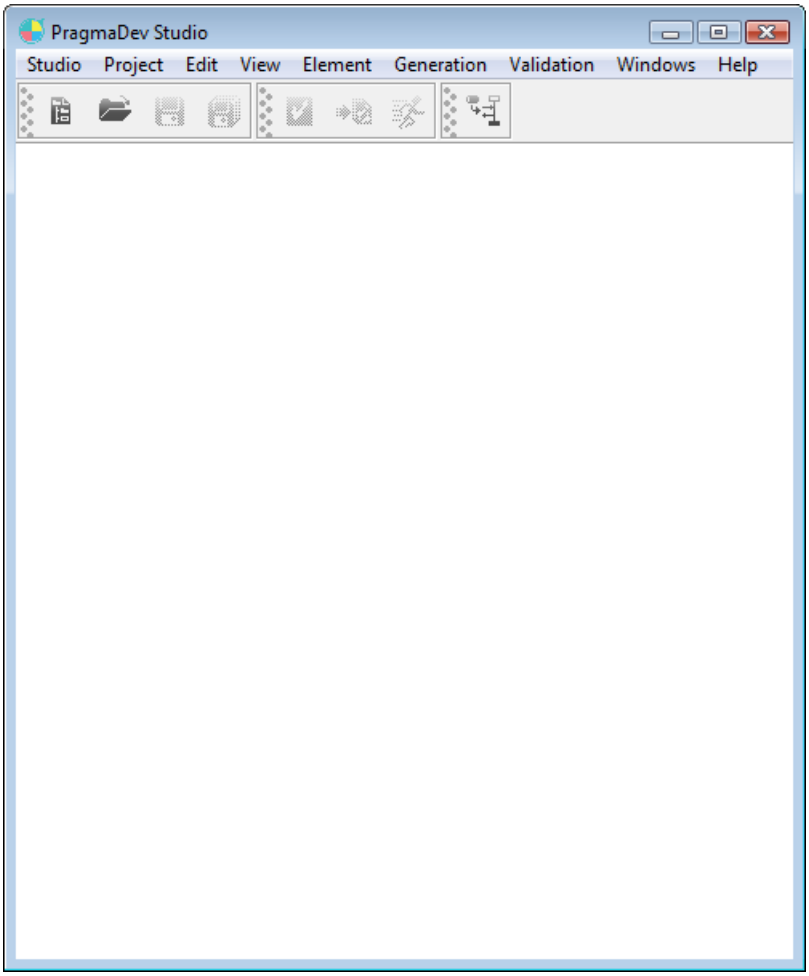
This is the aim of the SDL-RT language, which keeps the graphical description used in SDL but introduces all missing concepts required for system low-level design.

So let's move on to the PragmaDev Developer tutorial!

5 - PragmaDev Developer Tutorial

5.1 - Organization

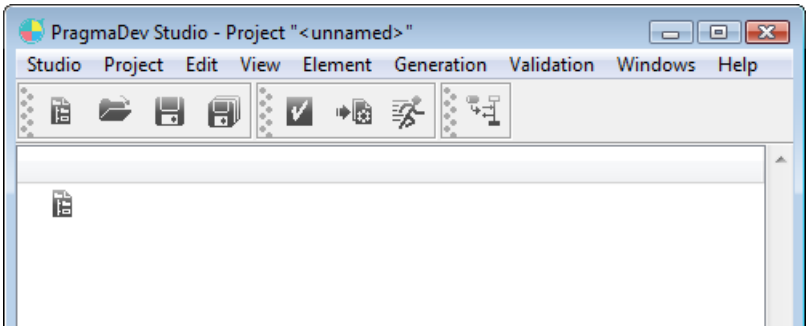
Let's get our hands on the tool ! Start *PragmaDev Developer* (or *PragmaDev Studio* if this will be the application you will be using).. The window that appears is called the Project manager:



The Project manager window

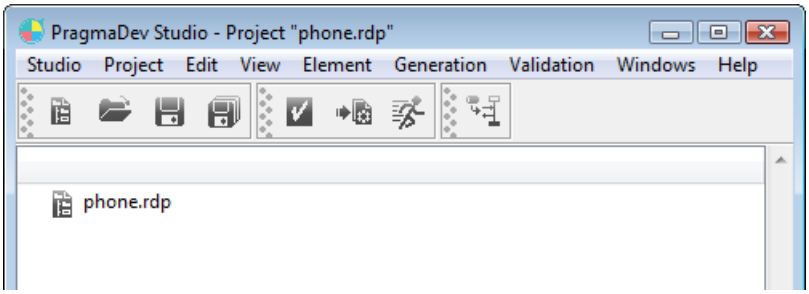
The project manager gathers all the files needed in the project. First let's create a new project with the *New project* button:

The Project manager displays an empty project:



An empty new project

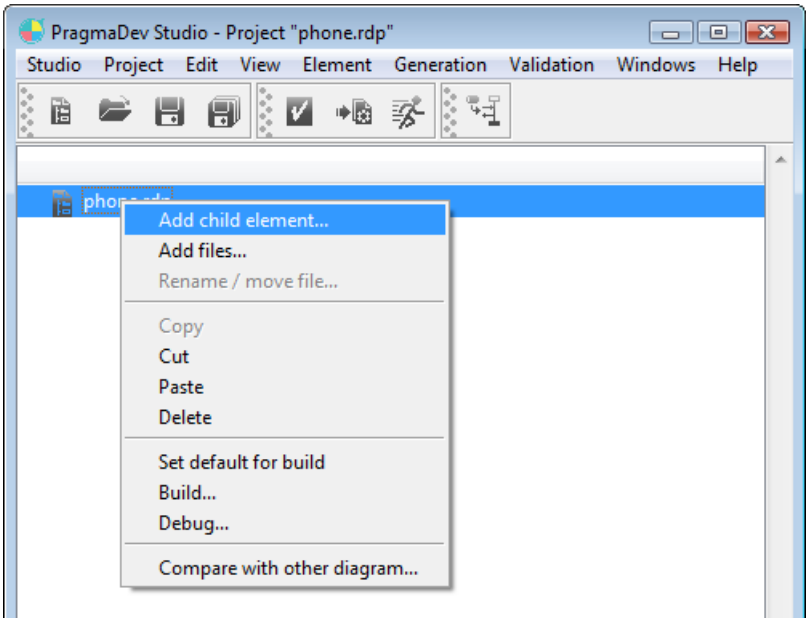
Let's save it straight away as "phone" and put it in a dedicated directory:



phone empty project

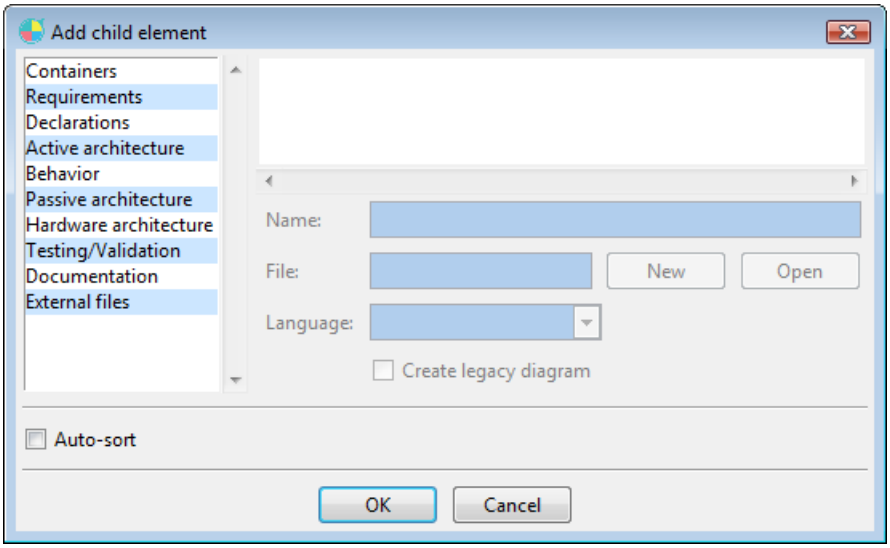
5.2 - Requirements

Let us express the requirements of our system as MSC. To add an MSC, select the project, and click on the right mouse button. A contextual menu will appear:



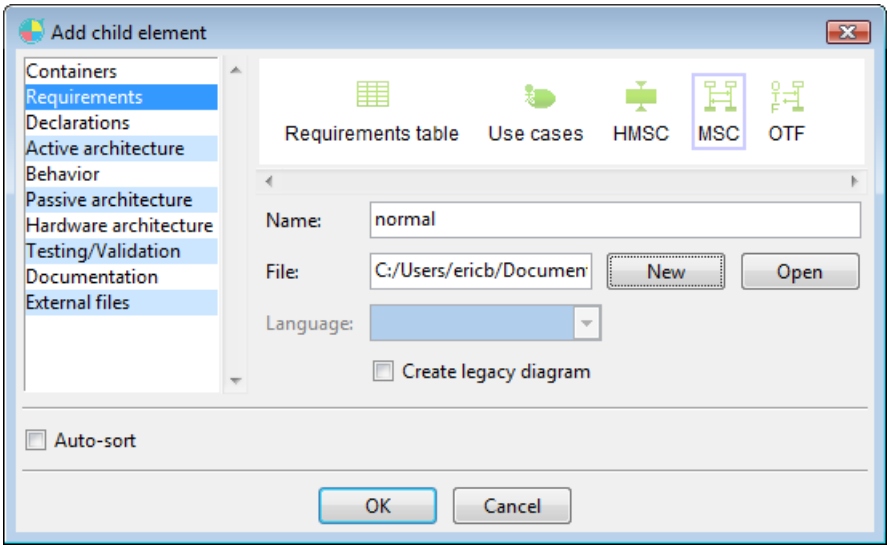
Add components to the project

Select *Add child element* and the following window will appear:



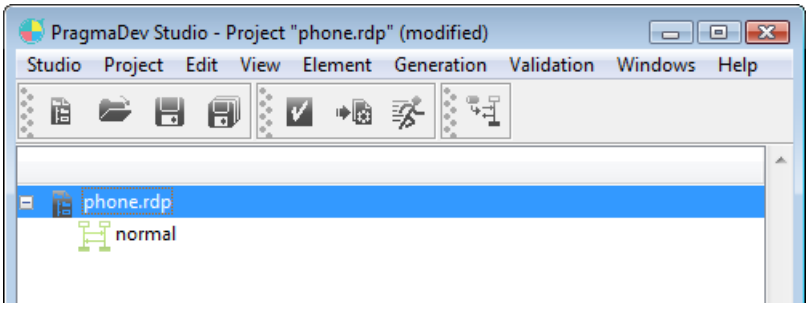
The add component window

In *Requirements* category, select MSC component and click on the *New* button. Go to the directory where your project is and type in "normal" with no extension. Click on save and you will get the following window:



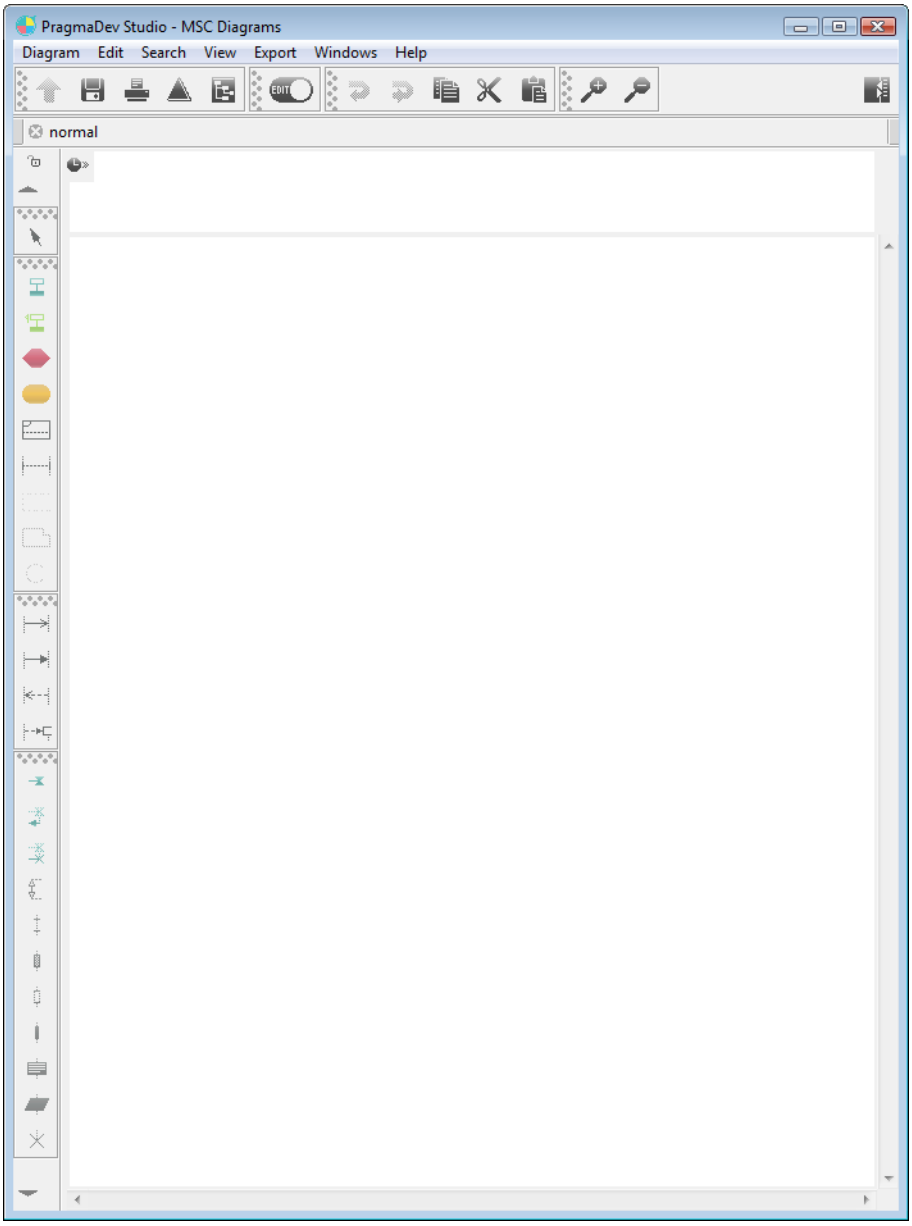
Completed Add component window

Click Ok and the "normal" MSC appears in the "phone" project:



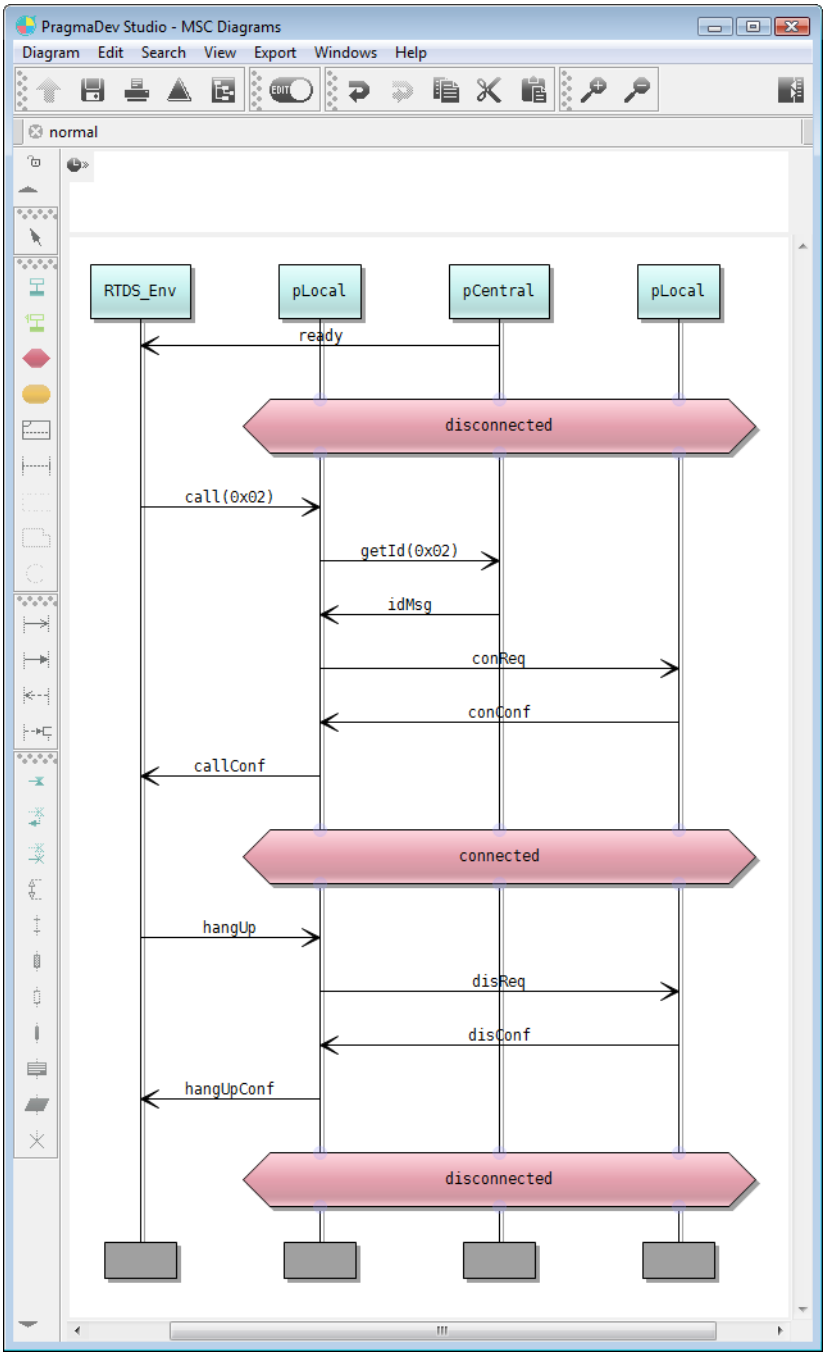
"normal" MSC in "phone" project

Double click on the MSC name or icon to open it. The MSC editor opens:



The MSC editor

Draw the following to express the requirements of our phone system:



The "normal" MSC

This MSC basically says the following:

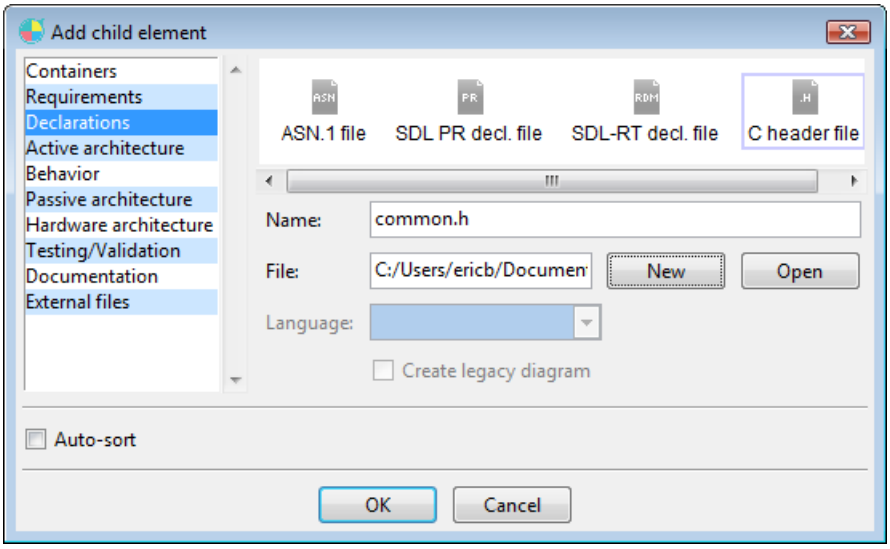
- `pCentral` indicates the system has been initialized and is ready
- The initial global state is `disconnected`
- The user represented as the environment (`RTDS_Env`) makes a request on the first phone `pLocal` to call the phone with the number `0x02`
- The first `pLocal` asks the central the queue id of the phone with number `0x02`

- The first `pLocal` uses the id to send a connect request (`conReq`) to the second `pLocal`
- The second `pLocal` being disconnected, it confirms the connection (`conConf`)
- The first `pLocal` tells the environment the call has succeeded
- The global system state is then considered `connected`
- The user hangs up
- The first `pLocal` sends a disconnection request (`disReq`) to the second `pLocal`
- The second `pLocal` confirms disconnection (`disConf`) back to the first `pLocal`
- The first `pLocal` tells the environment the disconnection is confirmed
- The overall final state is `disconnected`

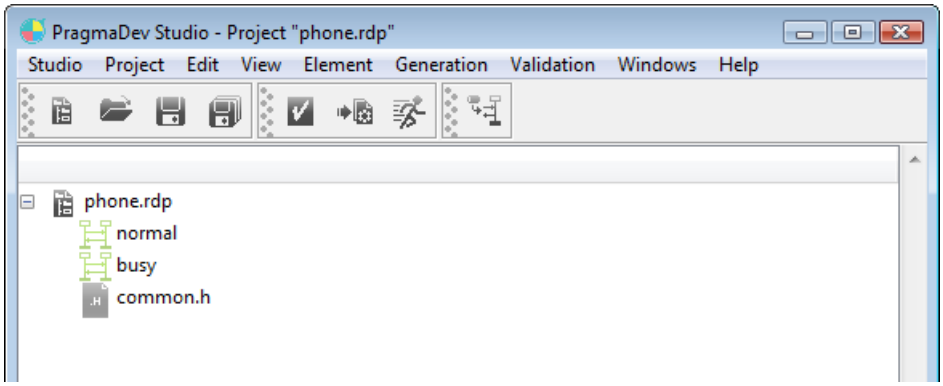
You can write some other MSCs to get clearer ideas on what you want to do. Note the instances represented on the MSC can be any type of agent or semaphore. Somehow you are roughly defining the first architectural elements. You can copy from the phone example "normal" and "busy" MSCs in the project to complete the description.

5.3 - Design

Let us now specify and design the system. As for creating an MSC, select the project and add a C header file component:



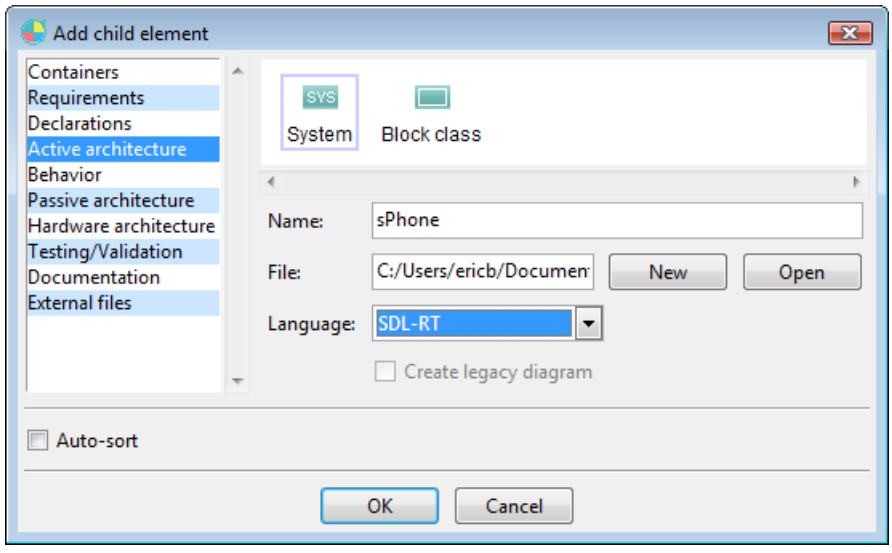
Project will be like:

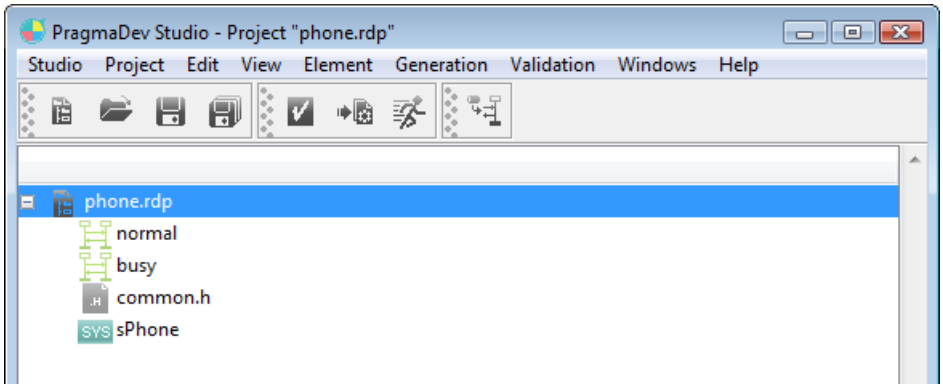


This header file will contain all type and macro definitions to use in the whole system. What we need is the number of phones that will be created and a type for the phone number. So open common.h type its contents:

```
/* Number of phones to create */
#define NUM_PHONE 5
```

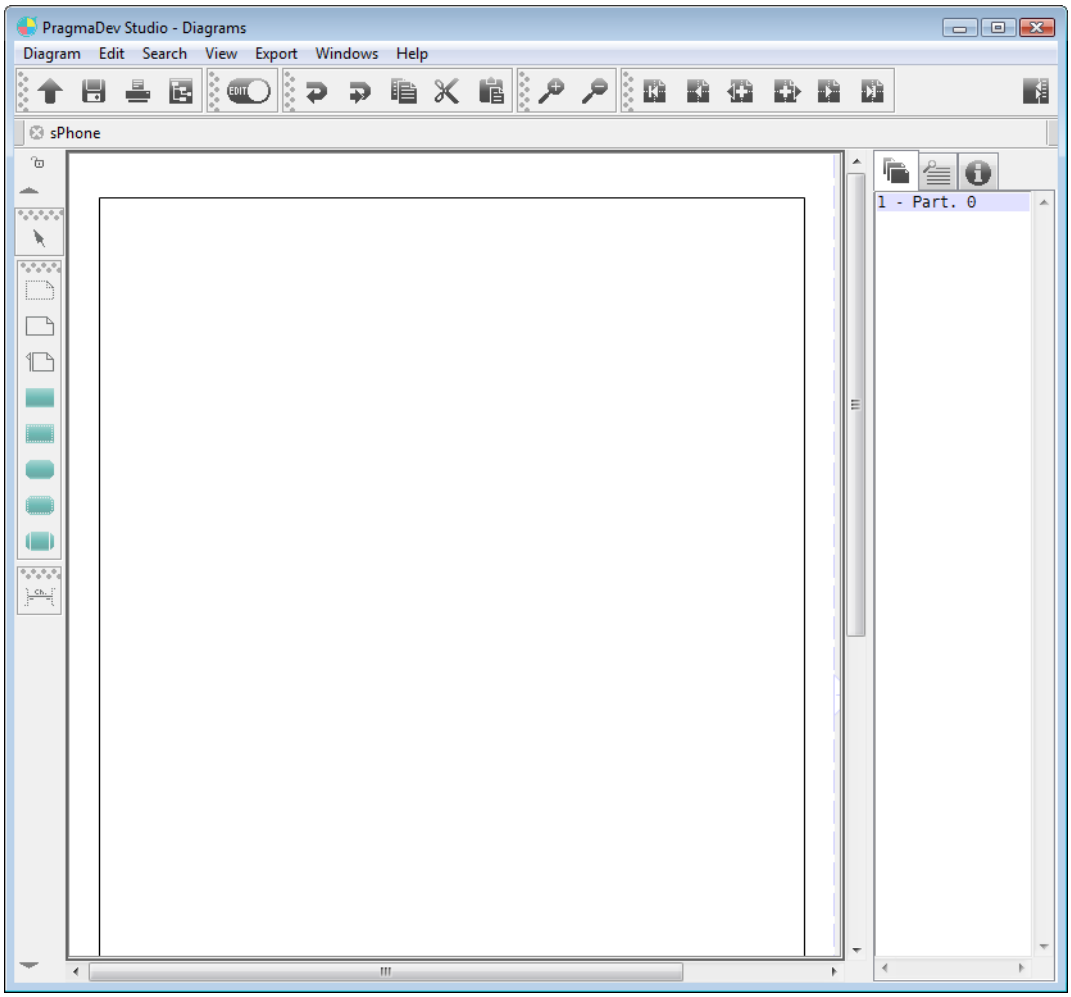
Now, let us actually design the system itself. Select the project and add a System component. Make sure to select SDL-RT as the language for the system:





"phone" SDL-RT system in the "phone" project

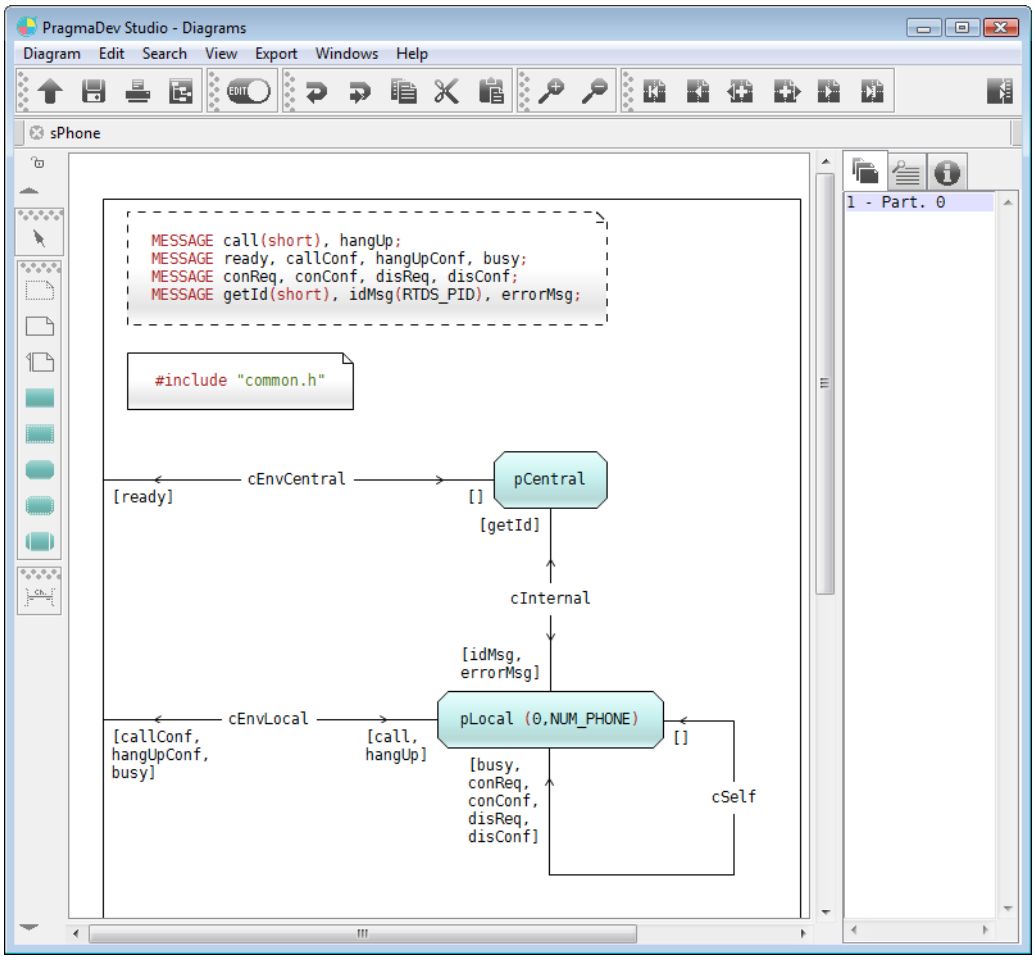
Double click on the system name or icon to open the system diagram in the SDL-RT editor:



The SDL-RT editor

The system being very simple it will not require any block decomposition. The central will be a process as well as the phones. All the phones have the same behavior so they will be several instance of the same process. The phone system is therefore made of two processes. For better legibility their name will be prefixed by a 'p' because they are processes.

Note: to draw the cSelf channel keep the shift key down and click where the channel should break.



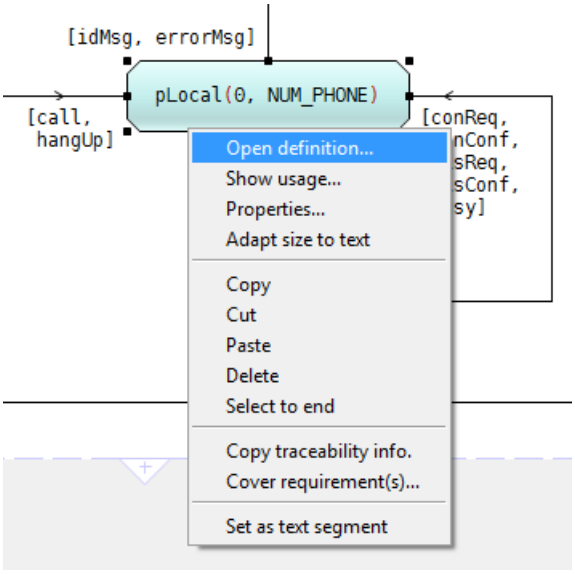
phone system view

Since pCentral is making the link between the pLocal processes and considering the number of phones can be modified, pCentral will create all instances of pLocal. To represent that, the name pLocal is followed by the initial number of instances and the maximum number of instances. Since the maximum number of instances is defined in common.h, we include it in a text box.

Messages to be exchanged between the processes are defined in the additional heading symbol and listed in the channels . To specify the incoming and outgoing messages in the diagram, click on the "[]" and type in between the square brackets. The channel going to the outer frame is implicitly connected to the environment. In the above example the channel cEnvLocal connects pLocal to the environment and defines call and hangUp as incoming messages and callConf, busy and hangUpConf as outgoing messages. The channel cEnvCentral connects pCentral to the environment and defines ready as an outgoing message. The cSelf channel has been created to represent messages exchanged between the different instances of pLocal.

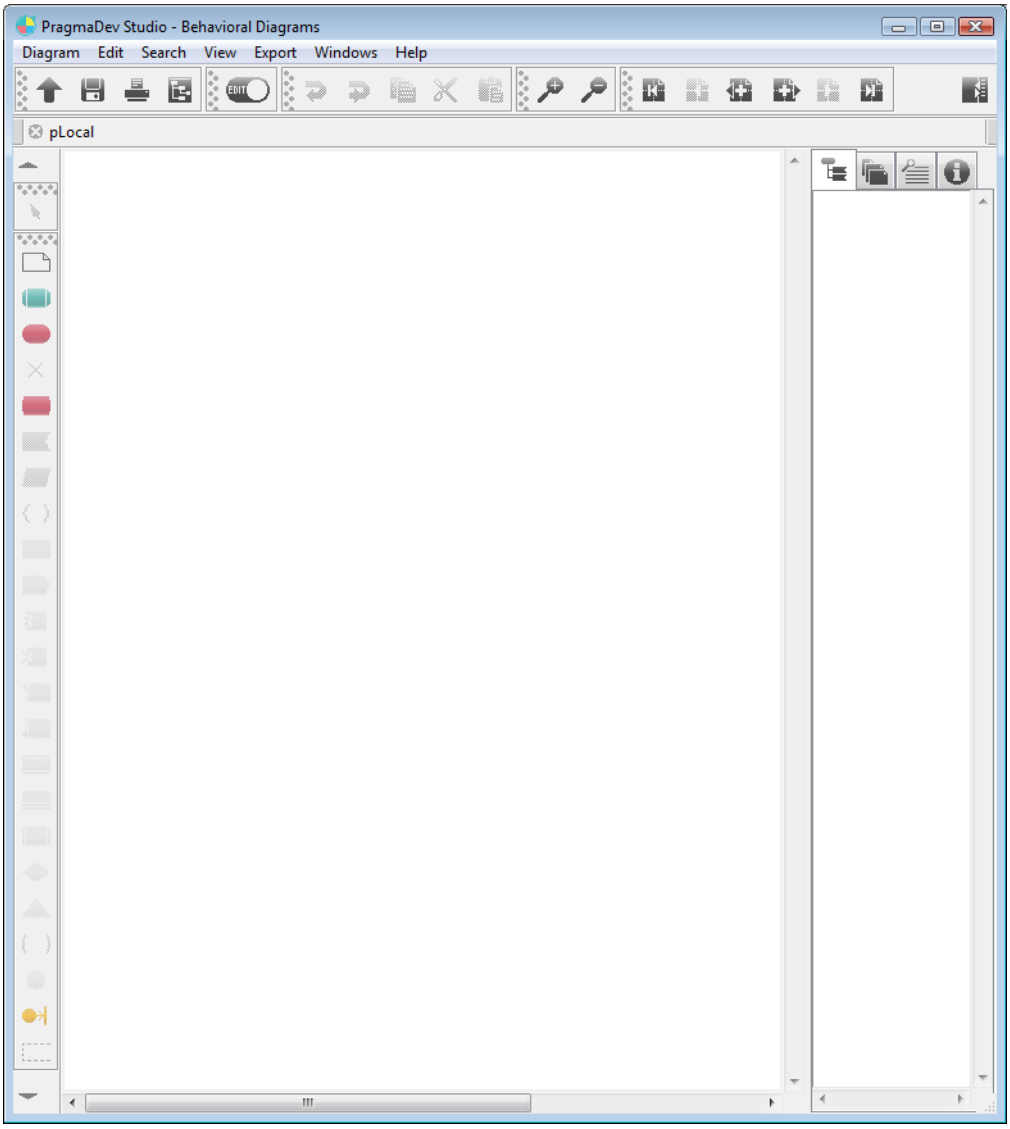
In the message definitions, only the messages call, getId and idMsg have parameters. In our example process ids will be stored, so in order to design an RTOS independent model the RTDS_PID type will be used. During code generation this type will be mapped to the real RTOS data type.

Select `pLocal` and click on the right mouse button to open the process definition, or simply click on the button that appears near the symbol when hovering the mouse pointer over it:



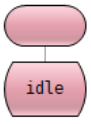
Contextual menu

Since the process is not in the project it will ask if it should be added. Answer yes and an *Add child element* dialog pre-filled with the process name appears. Click OK and you end up in a new windows showing the process definition.



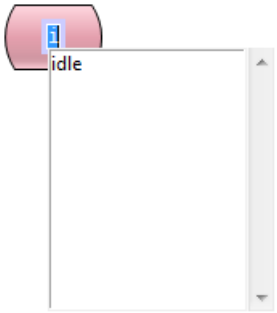
The process behavior description in SDL-RT editor

The first thing to design is the start transition. It is what the process will do as soon as it is created. In the case of pLocal process we do nothing:



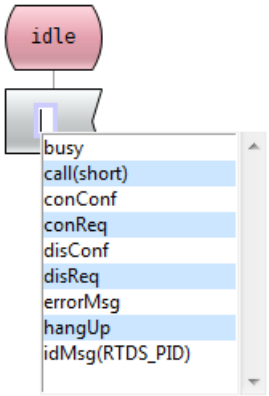
That transition means that once the process is started it will go to state `idle`. Note you can use automatic insertion in the editor: place a start symbol , keep it selected and click on the state symbol in the tool bar . The state symbol is automatically inserted and connected after the start symbol.

An internal data dictionary is updated on the fly to ease the writing of the process behavior. First create the `idle` state definition: click on the State icon and put it at the top of your page:



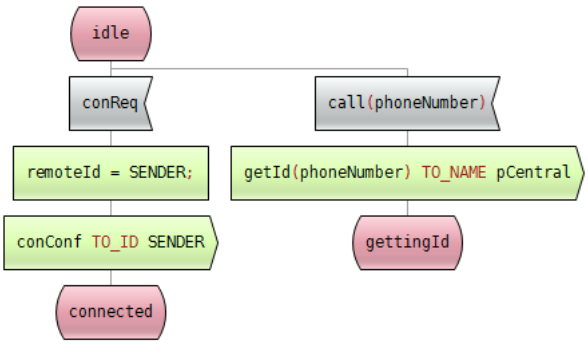
The state name is in edit mode so you can directly type `idle` in it. Note that as soon as you start typing, PragmaDev Developer will display a list of choices for the state name, listing all those starting with the text you have already typed. To select one of the names, use the up and down arrow keys to browse the list, or click on it. Here, the list has only one entry because the `idle` state is the only one that has been defined so far.

Once the state has been defined, click on the input symbol  in the tool bar and the input message symbol will be automatically inserted below the state symbol. As for the state, the list of possible messages will appear as soon as you start typing. You can also press the F8 key to display the list of all possible messages:



Select the `call` message and complete it with the parameter as described below. This facility is context sensitive and works for messages, states, semaphores, and timers. You can now finish the state description by yourself as explained below.

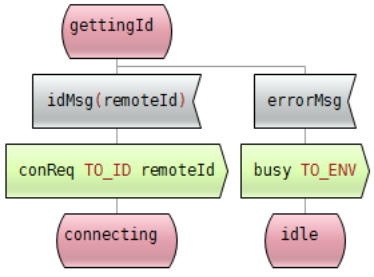
Considering the requirements described earlier the pLocal process can either be asked to make a call by the operator or receive a call from another phone. The idle state can therefore receive two types of messages described below:



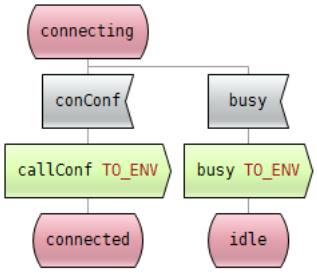
When receiving `conReq` message, it will reply `conConf` to the `conReq` sender. `TO_ID` and `SENDER` are SDL-RT keywords in the output symbol. The `SENDER` id is stored in `remoteId` variable. The process then goes to `connected` state.

If asked to make a call, the phone number to call needs to be retrieved. To do so, a variable of the correct type is given as parameter of the receiving message. It will be assigned when this message is received. Since pLocal has no idea how to address a phone number it asks the central process the receiver queue id with the `getId` message. The `phoneNumber` variable is re-used as is and the `TO_NAME` SDL-RT keyword is used to specify the receiver. Note `TO_ID PARENT` could have been used since the central process is the current process's parent. The process then goes to `gettingId` state waiting for the central to answer. Note also that the memory allocated for `phoneNumber` memory will be freed by the receiver of the `getId` message.

Once the queue id of the remote phone is received from central, it is first stored in a local variable and the connection request message `conReq` is sent. The process then goes to `connecting` state. If the pid of the receiver was not found, the `errorMsg` message is received. The process tells the user (environment: `TO_ENV`) and goes back to `idle` state:

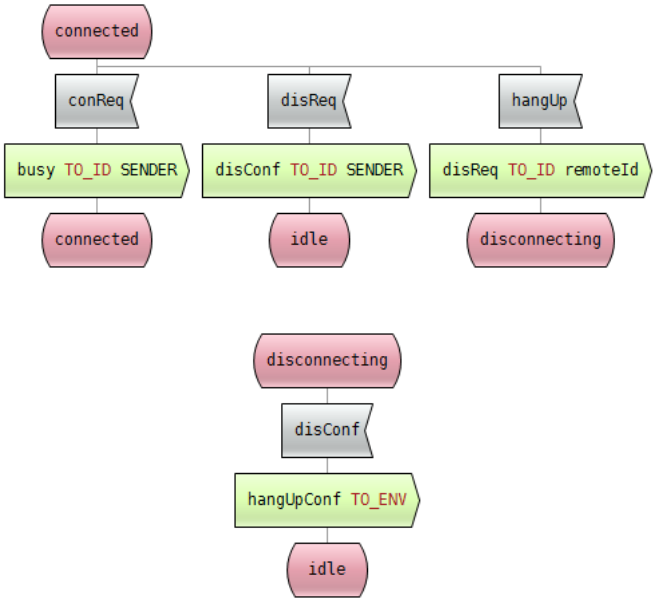


The remote process is either available and replies `conConf`, or not available and replies `busy`:

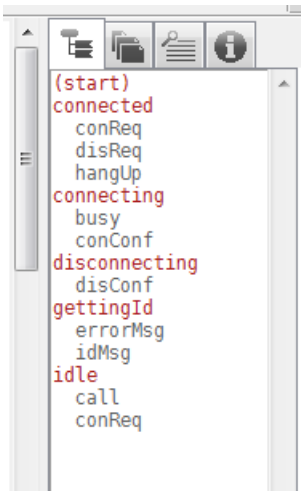


Depending on the answer the resulting state is different.

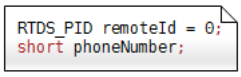
Now that you have understood the basics of the finite state machine you can complete the process behavior:



As the description is done, the browsing window on the right side is updated allowing to quickly jump to a transition: just click on the transition. This is especially useful when the system gets big.



It is now time to declare variables in our process. To do so the text symbol in the process behavior diagram is used with standard C declarations in it:



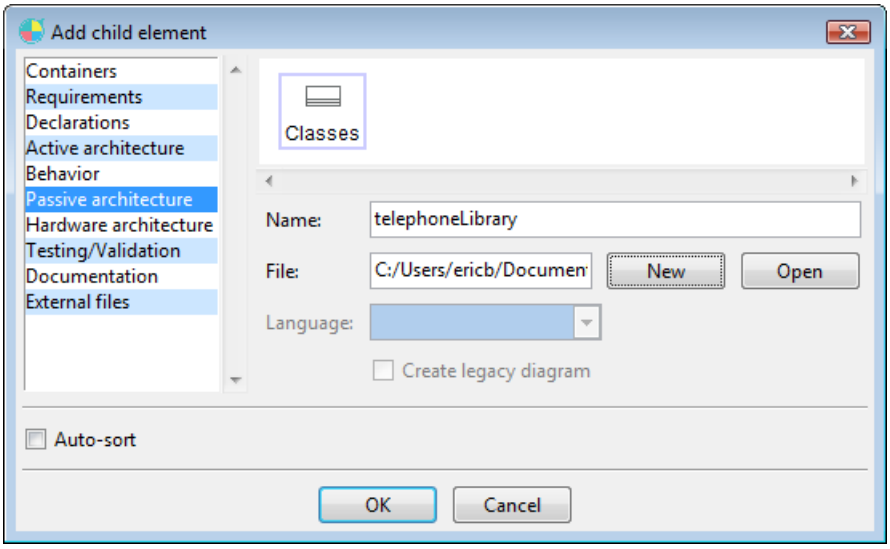
Note the types of the variables used in the input and output symbols: `short` for `call` and `getId` and `RTDS_PID` for `idMsg`. `RTDS_PID` is the type for a process identifier; It is mapped to the corresponding RTOS-dependent data type.

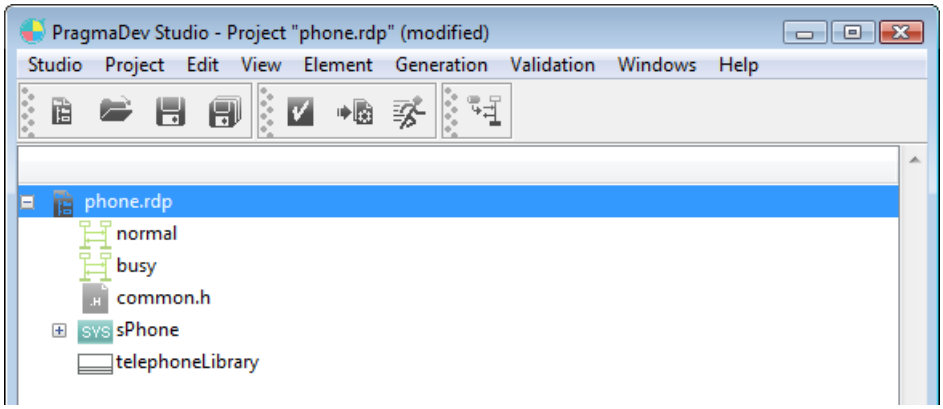
Let's have a look at process pCentral now. It must do the following things:

- at startup, it creates all instances of pLocal and gives them a new phone number;
- when asked for a phone number, it sends back the queue id for the corresponding process.

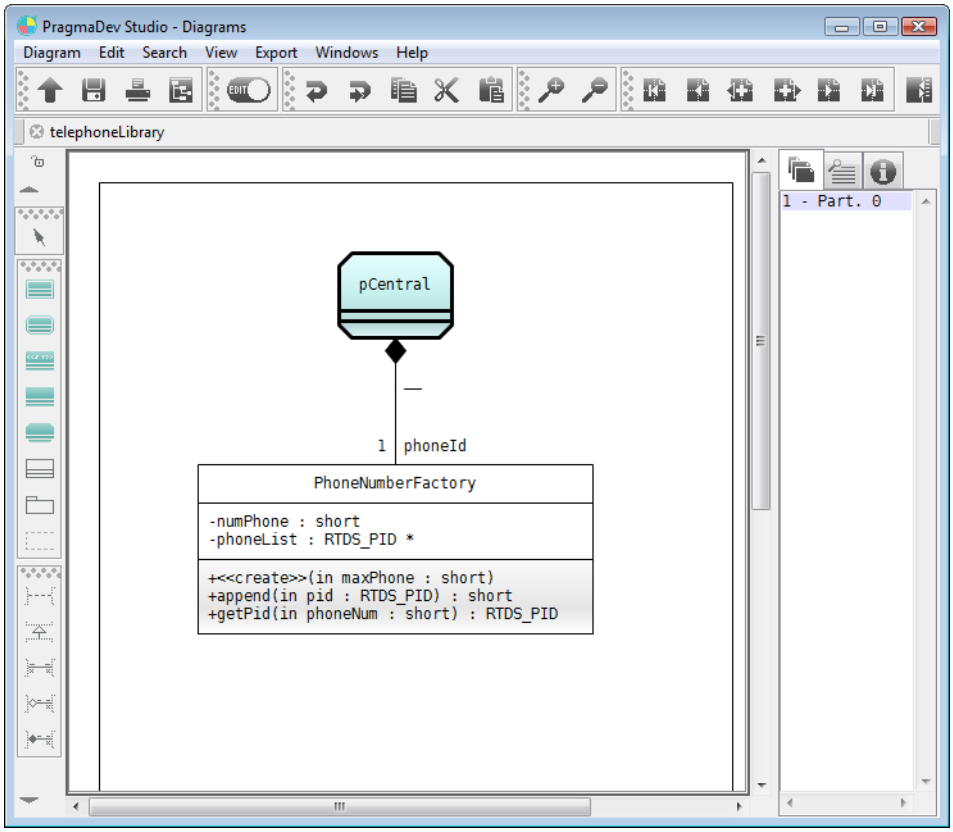
To avoid mixing the code managing the phone numbers with the code managing the processes, let's decide we'll use a class associated to `pCentral` that will take care of the phone numbers.

The first thing to do is to tell pCentral that it should use a class. So, let's go back to the project manager with the  quick button and let's create a class diagram named `telephoneLibrary` in the project:





Double-click on the class diagram's name to open it and let's define a class associated to pCentral:



In a class diagram, the process `pCentral` is represented as an active class with a "graphical stereotype": the class symbol has bold borders and looks like a process symbol in a block diagram. Note the symbol used is a process symbol, not a process class symbol. That means the symbol is a direct reference to the process declared in the SDL system.

The class `PhoneNumberFactory` is the class we will use to manage the phone numbers. Its interface is quite simple:

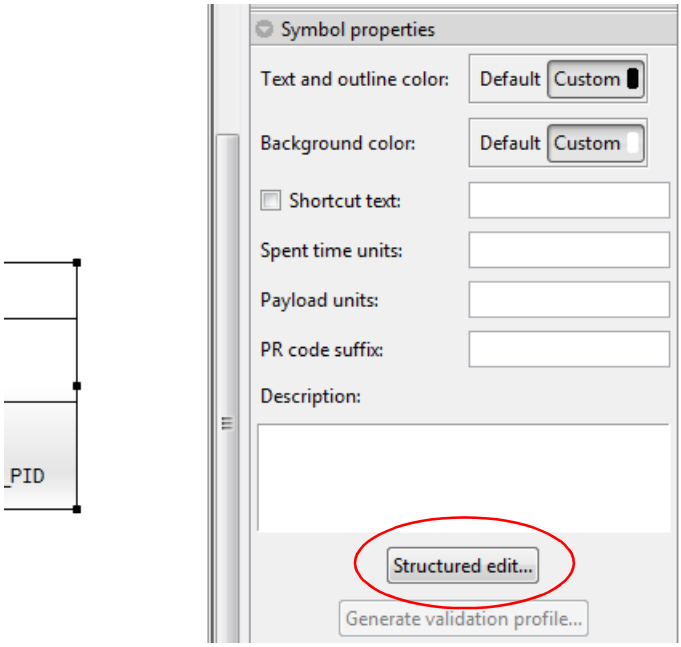
- Its constructor (named `<<create>>` in the symbol) will just initialize all internal data, it takes the maximum number of phones to manage;
- The `append` method will add a new phone to be managed.
- The `getPid` method will return the process id for a given phone number.

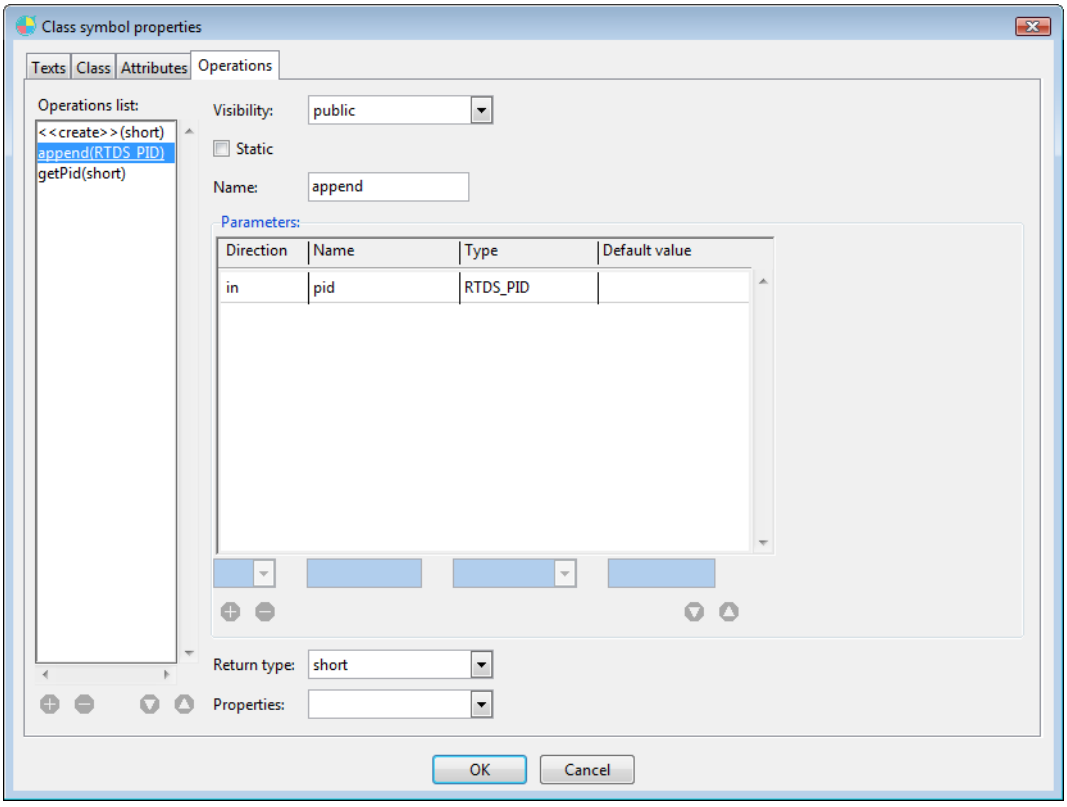
The class `PhoneNumberFactory` will also have 2 private attributes:

- `numPhone` is the next available phone number;
- `phoneList` is a pointer on a process id. The process ids will actually be stored in an array.

Since there will only be one instance of `PhoneNumberFactory` used only in `pCentral`, we can make the instance a part of the process via a composition with a cardinality set to 1. The role name `phoneId` will identify the instance of `PhoneNumberFactory` in `pCentral`. That means `phoneId` does not need to be declared in the SDL diagram; it is implicitly declared.

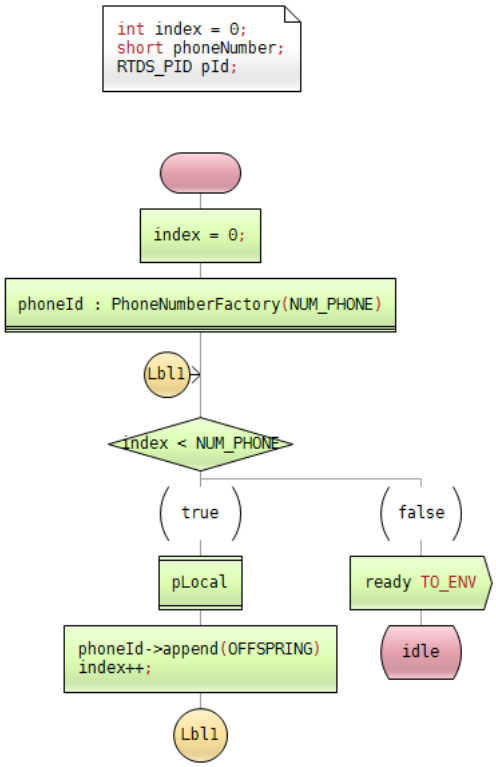
To fill in the attributes and operations, you can either type the text, or select the class, click on the right mouse button and select *Properties*. The symbol properties will appear in the zone at the right of the diagram editor. There, you can click the *Structured edit* button, which will open a dialog allowing to add all the attributes and operations in a graphical way:





Now we have our class to manage the phone numbers. Of course, it isn't complete yet, since we didn't write any actual code for the methods. But its interface is fully defined, so we can go back to our `pCentral` process.

Go to the system diagram `phone`, and double click on `pCentral`. Since the process is not in the project it will ask if it should be added. Answer *yes* and a pre-filled save window with the process name appears. Click *OK* and the process definition window appears. First, its initial transition:

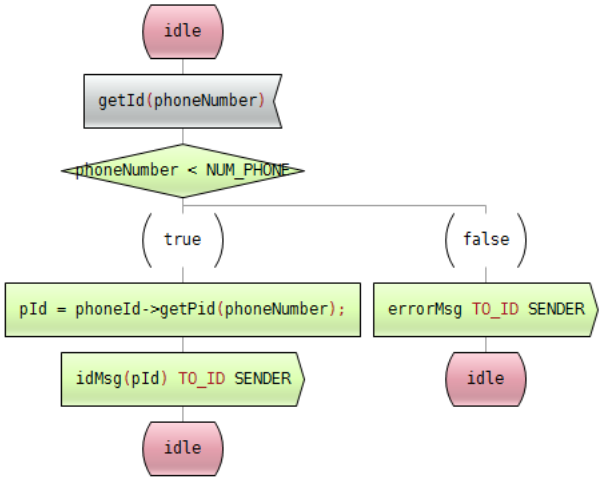


The first thing to do is to create the instance of `PhoneNumberFactory` we will use. This is done via an object initialization symbol, where the instance `phoneId` (the name we set in the role in the class diagram) is created as an instance of `PhoneNumberFactory`. The class constructor takes the maximum number of phones to handle as a parameter. Again, because of the association between `PhoneNumberFactory` and `pCentral` classes, `phoneId` is implicitly declared in `pCentral` SDL behavior diagram.

Then, all instances of `pLocal` are created in a loop testing `index < NUM_PHONE`. Each time the loop is executed the `pLocal` process is created and its process id (`OFFSPRING` keyword for the parent process) is stored in `phoneId` object via the `append` method.

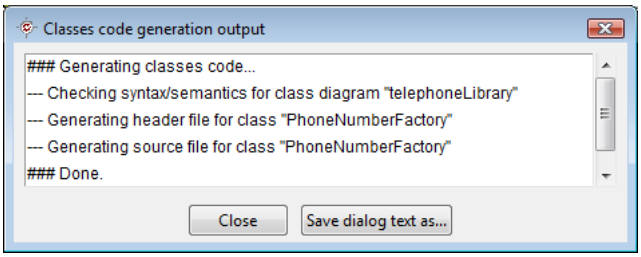
After the `pLocal` processes creation, the `ready` message is sent to the environment to indicate initialization is finished and the process goes to state idle.

Note we have intentionally introduced a C syntax error by forgetting ";" in the lowest block of code to later show how to analyze the compiler errors.

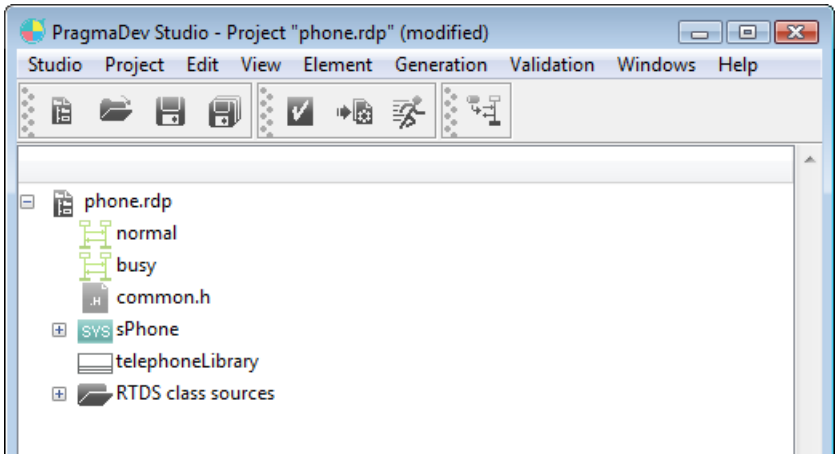


The only request that can be received by `pCentral` process is `getId`. The phone number to reach is the `GetId` parameter (`phoneNumber`). The process id of the phone is extracted with the `getPid` method. When received, the index corresponding to the received phone number is searched via the `getPid` method on the `PhoneNumberFactory` instance. The answer is sent to `SENDER` (SDL-RT keyword in output). If the phone number is out of range, an error message is sent back to the sender.

Now, let's go back to our class `PhoneNumberFactory`: we could write the .h and .cpp files directly, but PragmaDev Studio can help. So go back to the project manager, select the phone system, and select *Generate classes code...* in the *Generation* menu. A log window will then list the operations made during the code generation, which should run without errors.

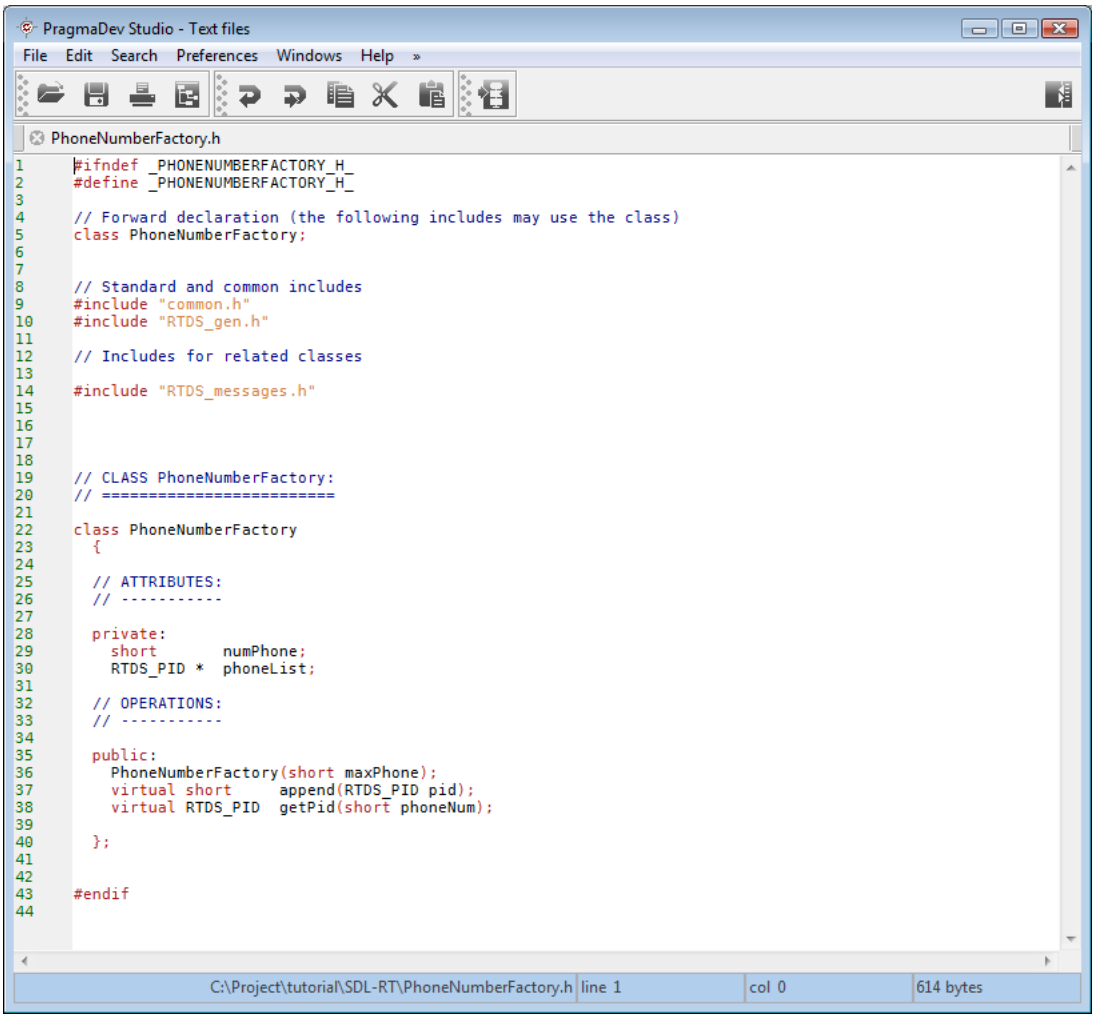


Now let's close the log window and go back to the project manager:



`RTDS class sources` package has been automatically created and contains C++ code generated from the classes defined in the project.

`RTDS class sources` should contain the `.h` and `.cpp` files for the class `PhoneNumberFactory`. Open `PhoneNumberFactory.h`:

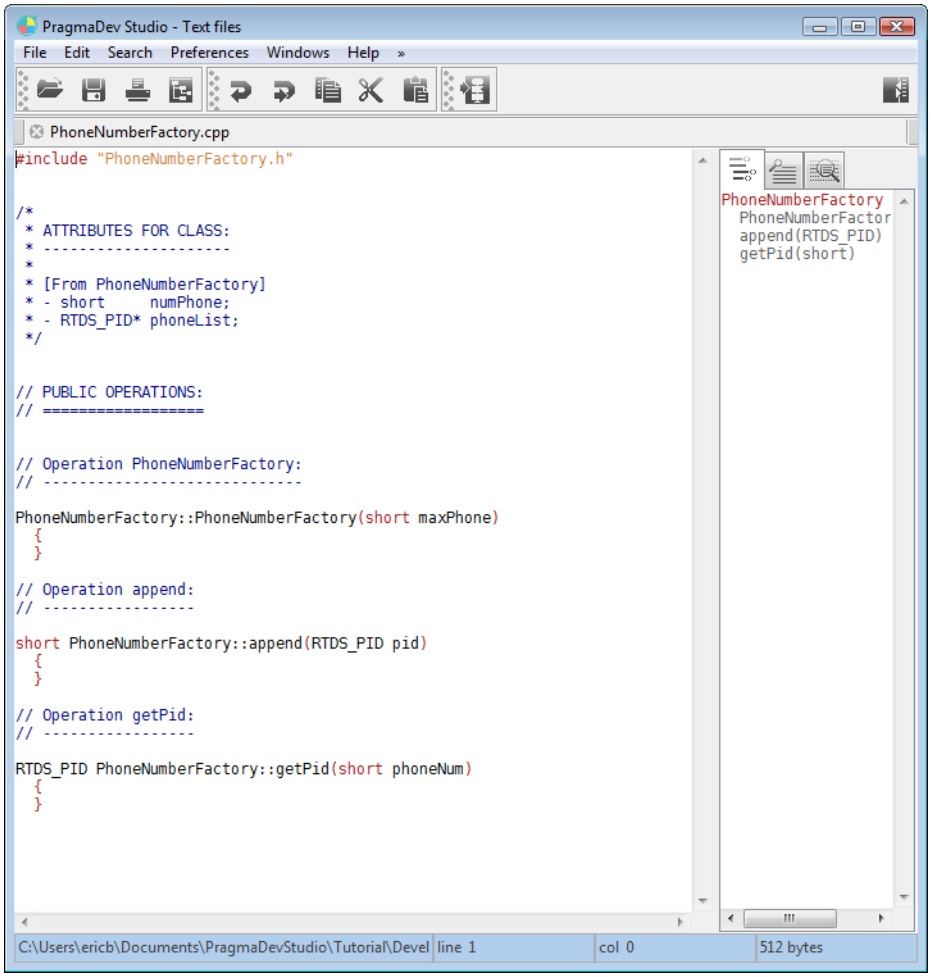


Since the file `common.h` has been defined at the project level, it is supposed to be needed everywhere in the project. So it has been automatically included in the generated header

file. RTDS_gen.h is a generated file containing declarations specific to the system. The class definition then contains all attributes and operations we entered in the class diagram.

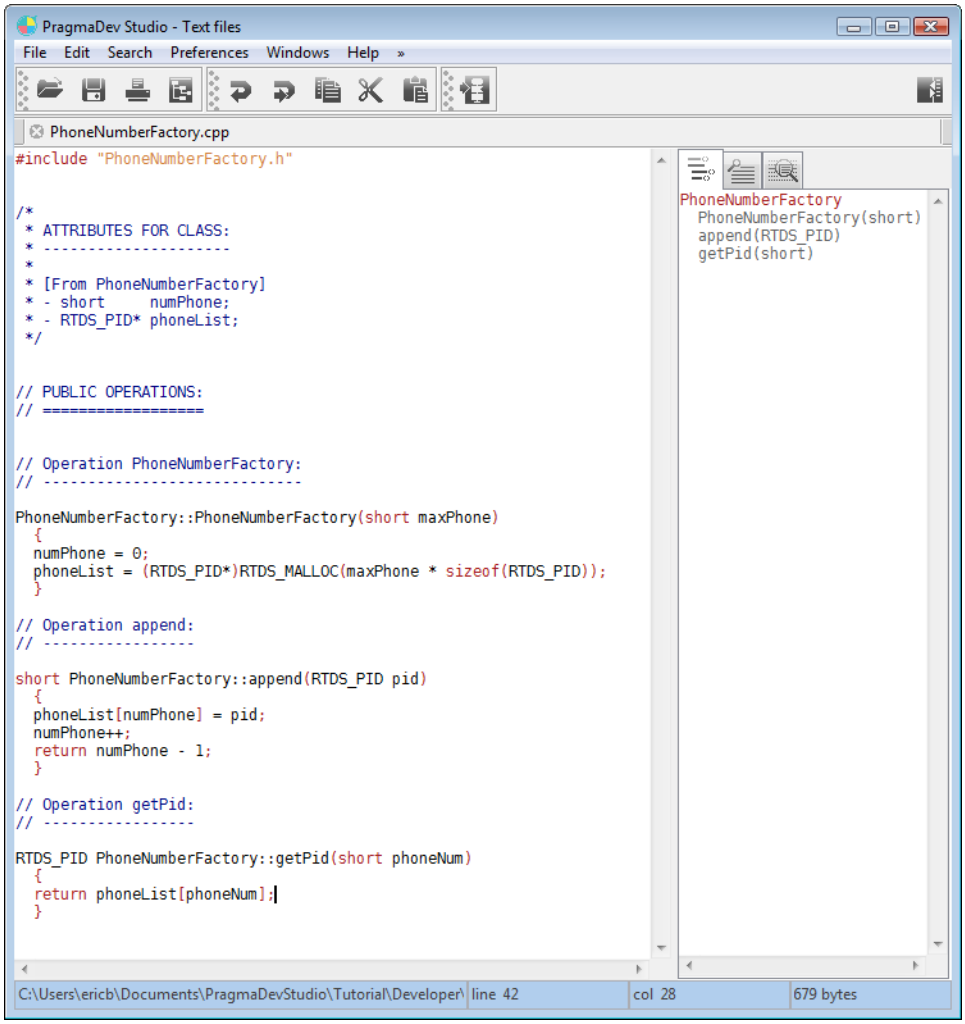
Please note this header file must not be modified manually: it will be re-generated each time a code generation is made.

Now open PhoneNumberFactory.cpp:



Skeletons for the constructor and the two methods defined for PhoneNumberFactory have been generated. Note the C++ code browsing window on the right to quickly access operation definitions. The generated code also includes as a comment the attributes definitions.

Now you can enter the code for the methods:



`numPhone` represents the next available phone number. The append method stores the process id in the `phoneList` array with index `numPhone`.

Please note that once the `.cpp` file exists, it will not be overwritten by the next code generation. So the code you've written will be kept as long as you don't manually erase the file.

5.4 - Running the system

In the current release, execution and debug of the system can be done using:

- Wind River Tornado environment on Windows or Solaris, or
- Posix and gdb integration on Linux or Solaris, or
- Win32 and gdb or MinGW integration on Windows, or
- CMX RTX and Tasking Cross View Pro debugger on Windows, or
- OSE and gdb debugger on Windows, or
- Nucleus and gdb debugger on Windows.

It is important to understand integration is done at two different levels:

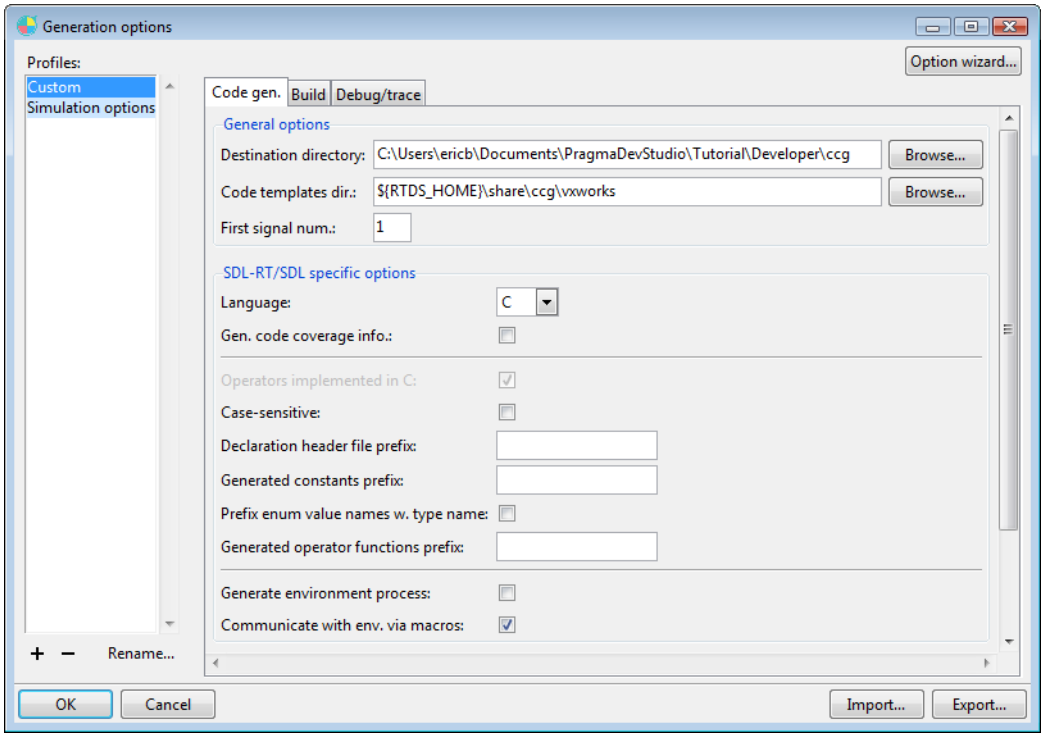
- RTOS integration
  The generated code is based on C macros that are defined in the "Code template directory" to call the corresponding RTOS system primitives. Currently there is a directory for:
  - VxWorks,
  - Win32,
  - Posix,
  - CMX RTX,
  - OSE Delta,
  - OSE Epsilon
  - ThreadX,
  - uITRON 3,
  - uITRON 4,
  - Nucleus.
- Debugger integration
  To be able to trace execution, set breakpoints and view variables, the SDL-RT debugger is interfaced with a C debugging environment. Depending on the C debugger functionalities there might be differences in the SDL-RT debugger. The available C debugger interfaces are:
  - Tornado
  - gdb (Gnu debugger)
  - MinGW (Minimalist GNU for Windows)
  - Tasking Cross View Pro
    Tasking integration has one major restriction: it is not possible to send an SDL-RT message to the running system from the debugger.
  - XRAY
    As with *Tasking* integration: it is not possible to send an SDL-RT message to the running system from the debugger.
  - Multi 2000
    As with Tasking and XRAY integrations, it is not possible to send an SDL-RT message to the running system from the debugger.

The rest of the tutorial will use your host environment as a target (windows or posix integration) and gdb as a debugger.

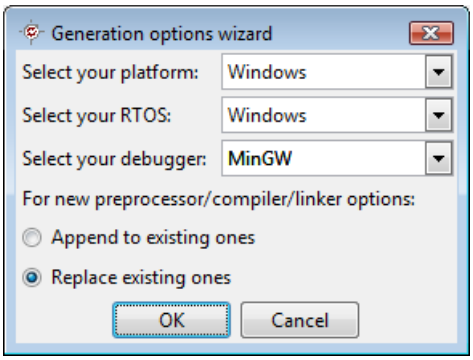
Please note win32 and posix integrations use a socket to communicate with the host. The default port set to 49250 but it can be modified in the *Socket port num* field of the corresponding generation profile.

5.4.1 Generation profile

Now that the system is designed, let's debug it with the SDL-RT debugger. To do so we will need to generate code from the SDL-RT system. This requires to define a set of generation options, which is done via the *Generation / Options...* menu:



Rename the default empty profile and use the *Options wizard* to quickly set up a working profile:

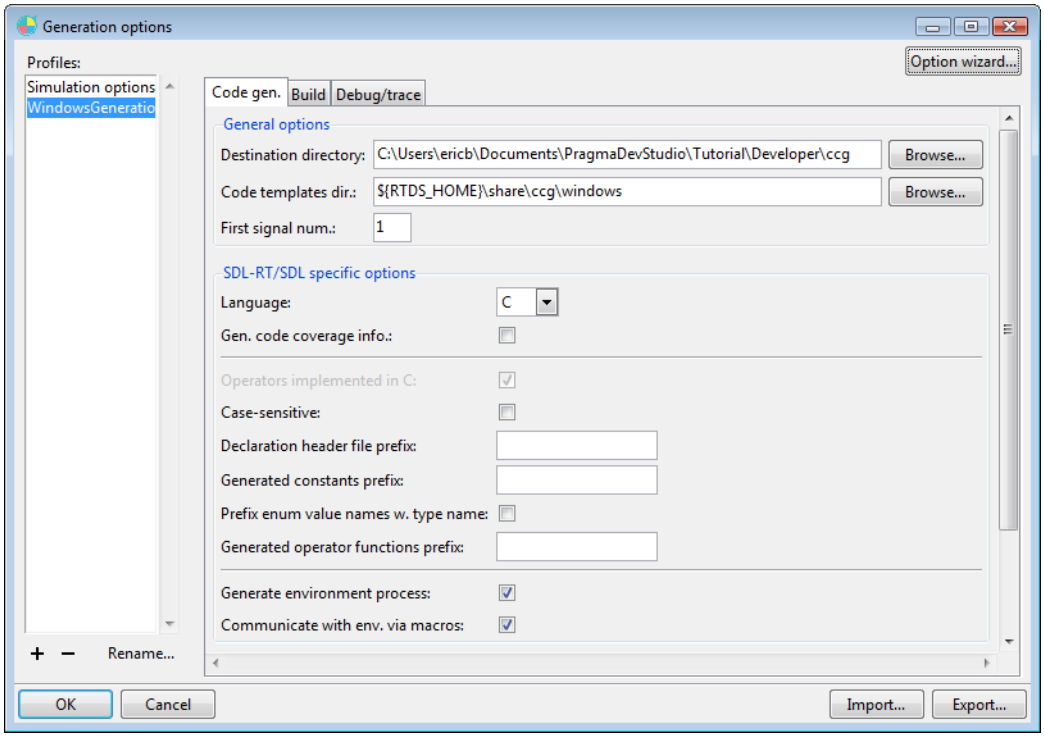


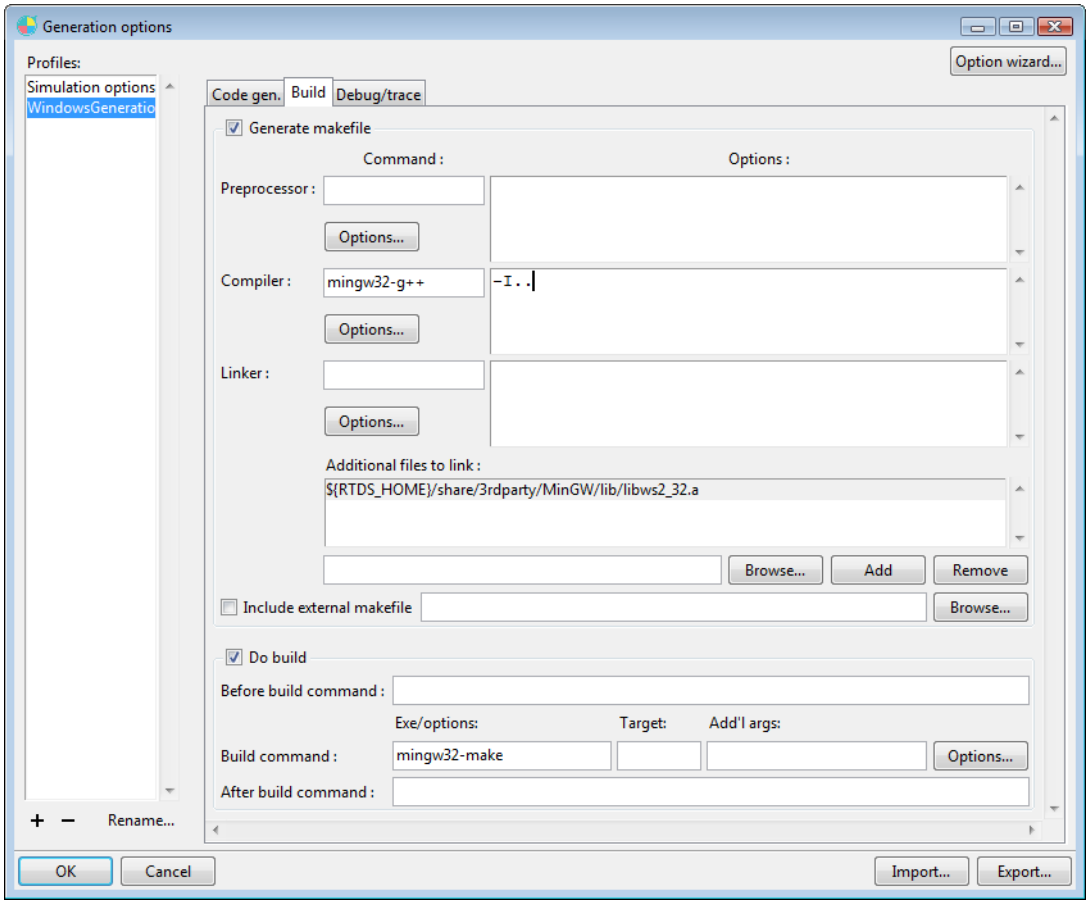
You'll have to check that everything is setup properly, and also to change a few things in the created options:

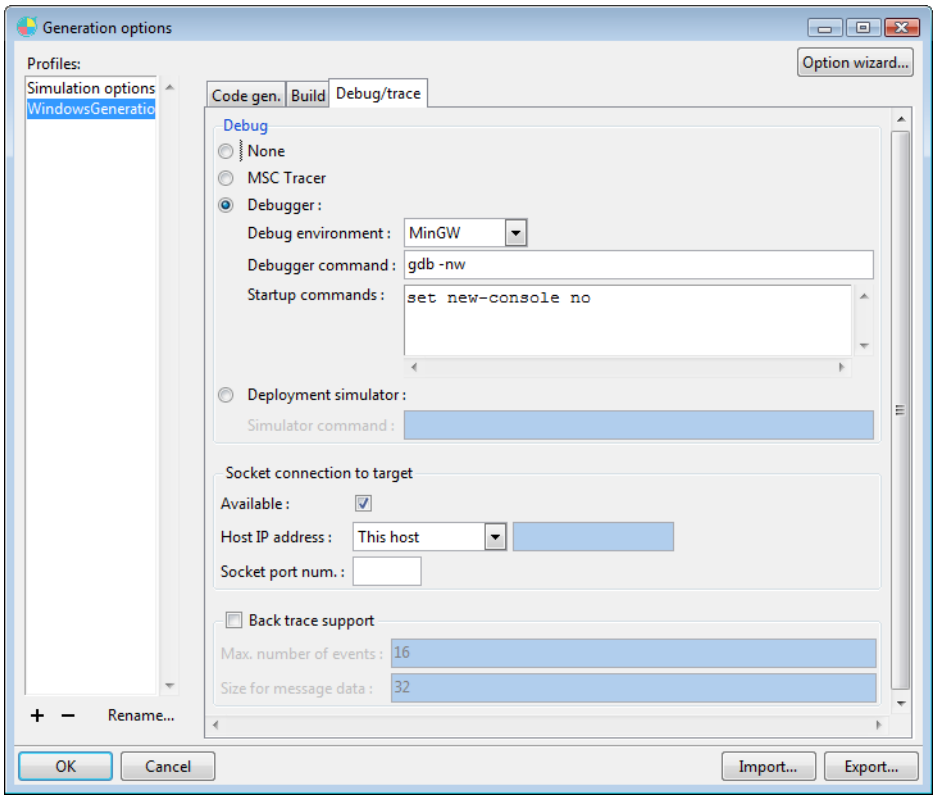
- The destination directory should be set to `ccg`. This directory is relative to the project directory, and will be created if it doesn't exist yet, so no need to create it explicitely.

- Since we're using C++, the compiler has to be changed, the default one being a C compiler (in the build tab). The correct value is:
  `mingw32-g++`
- Make sure the C++ include path option includes the upper directory because the `common.h` file is in the project directory and the generated C files are in `ccg`. So the compiler options must include:
  `-I..`

Here is how the 3 tabs should look like in the final set of options:





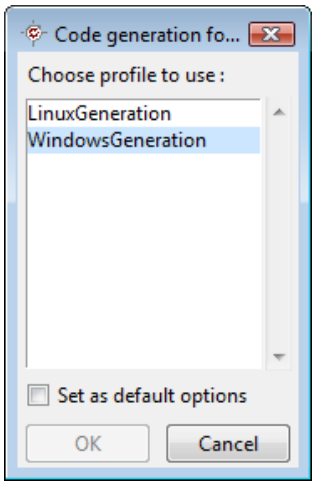


Gnu example on Windows platform

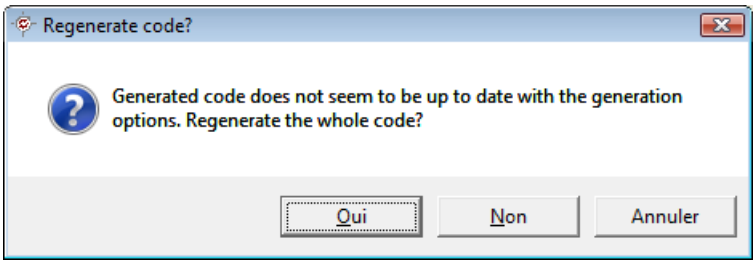
5.4.2 Compilation errors

Once the SDL-RT debug profile is properly defined select the phone system in the *Project manager* and click on the *Debug* quick button in the tool bar: 

If several execution profiles are defined, as in the examples, a window pops up asking for the profile you want to work with:

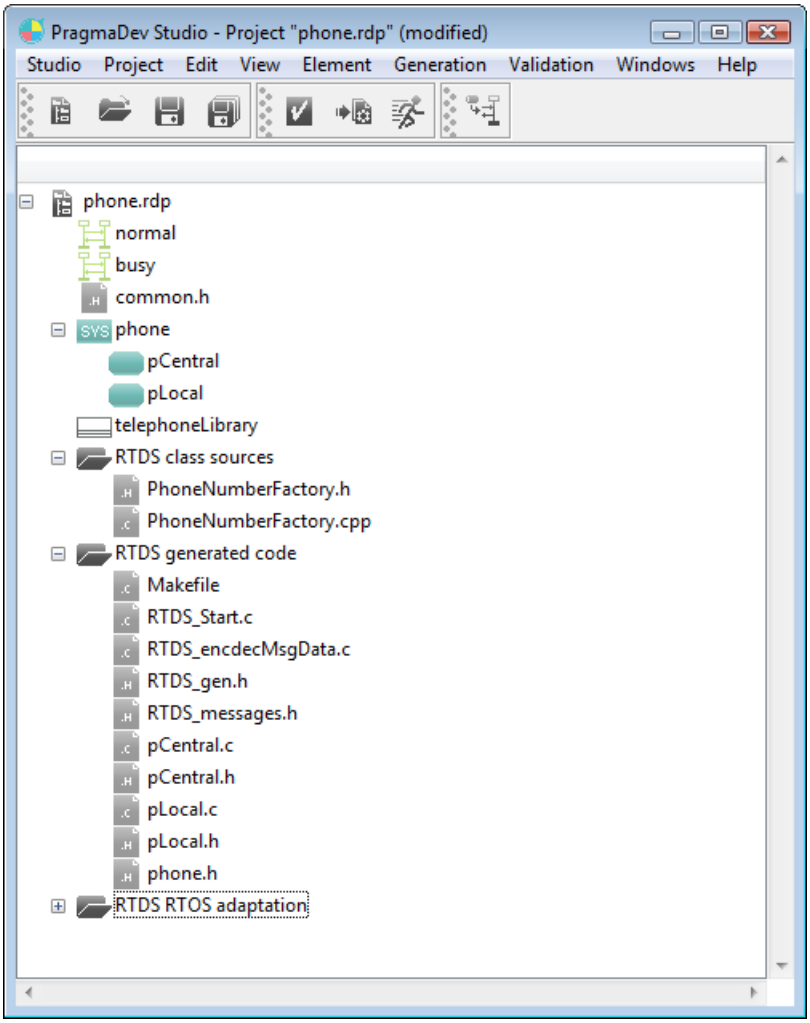


PragmaDev Studio will compare the dates of the generated C/C++ files with the dates of the project, the diagram, the preferences. If the generated C file are not up to date the following window will pop up to confirm the code should be generated again or not.



This is very useful with large projects to avoid long compilations.

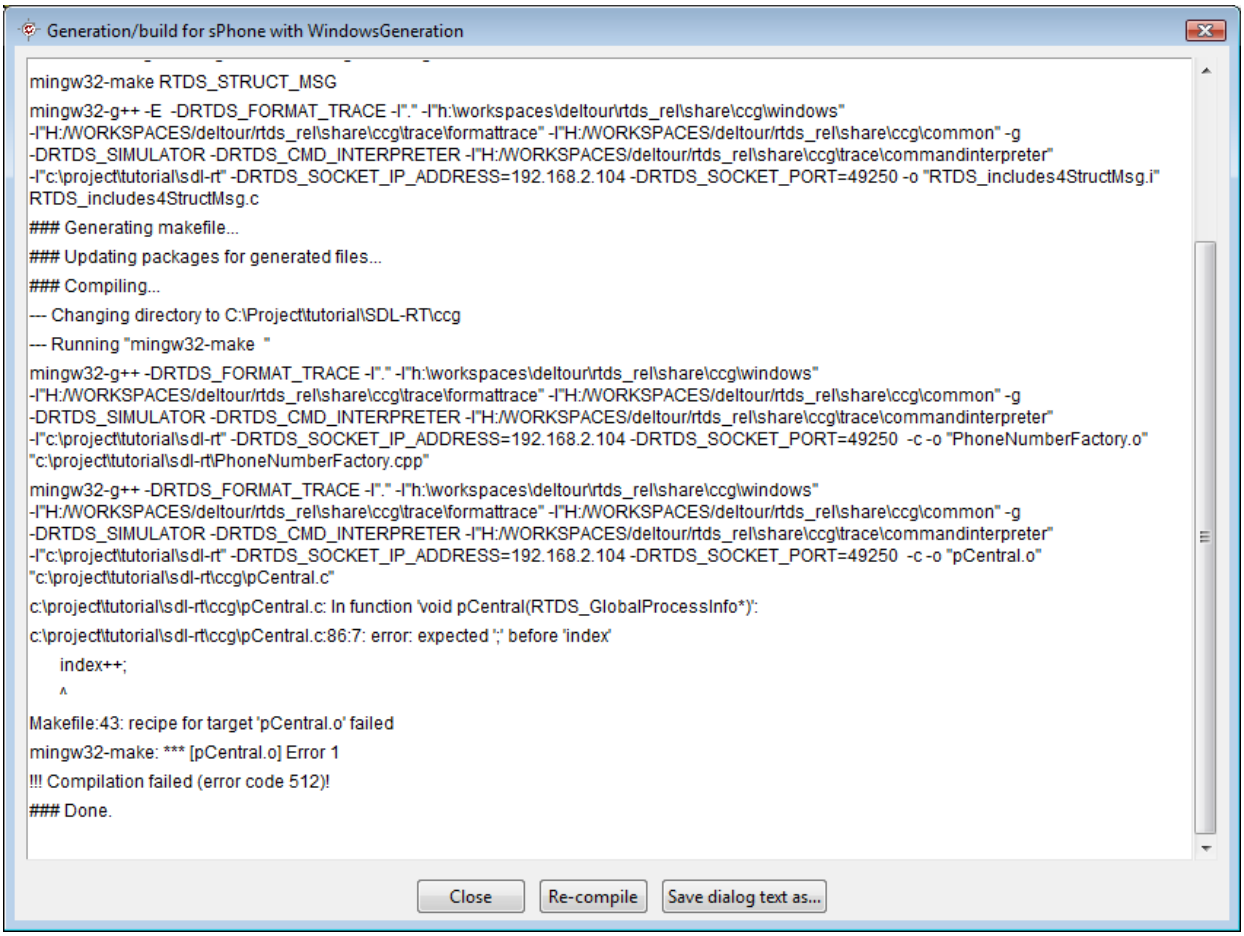
The package `RTDS generated code` is automatically created in the Project manager window that will contain all the generated C files.



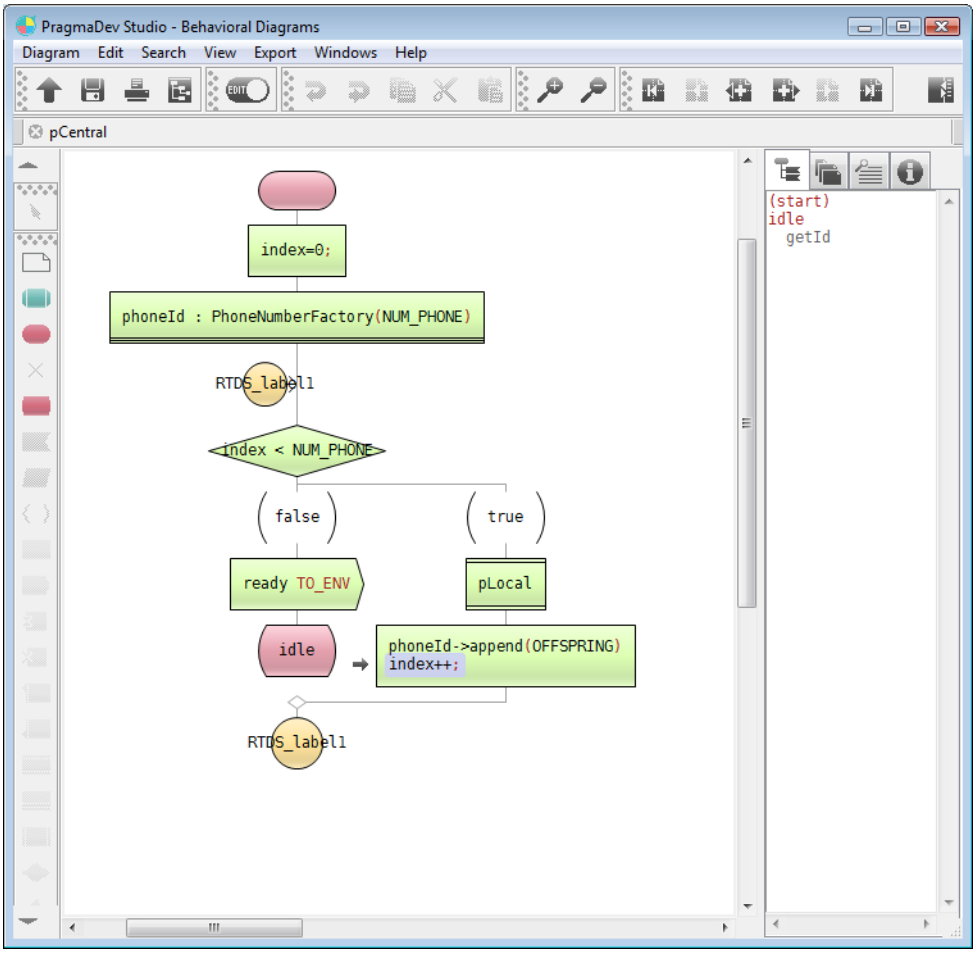
The package `RTDS RTOS adaptation` is automatically created in the Project manager window that will contain all the C files needed to adapt to the selected RTOS. These files are actually in the code templates directory defined in the generation profile. These files are normally not needed but since they are part of the build process they must be visible.

The package `RTDS class sources` is also re-generated, but all the existing `.cpp` files are left as is.

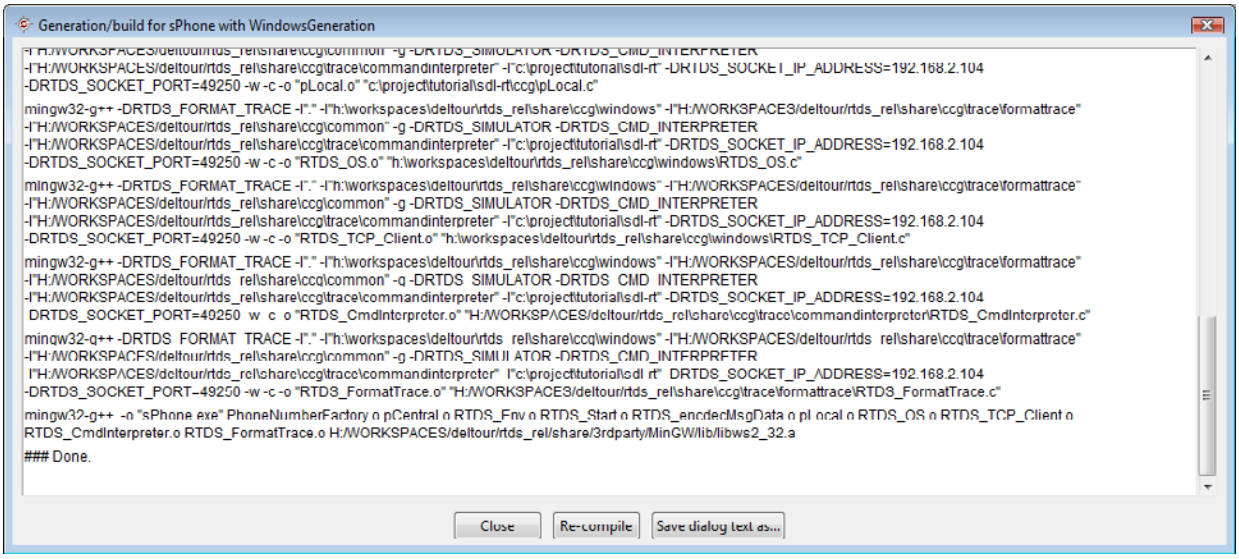
Syntaxic verification, semantic verification, code generation, and compilation starts: Let's consider an error occurred while designing `pCentral` process. The compiler will complain in the *Generation / compilation output* window:



Double click on the desired warning or error to automatically open the SDL-RT editor on the error (please note this is only available with gcc based compilers):

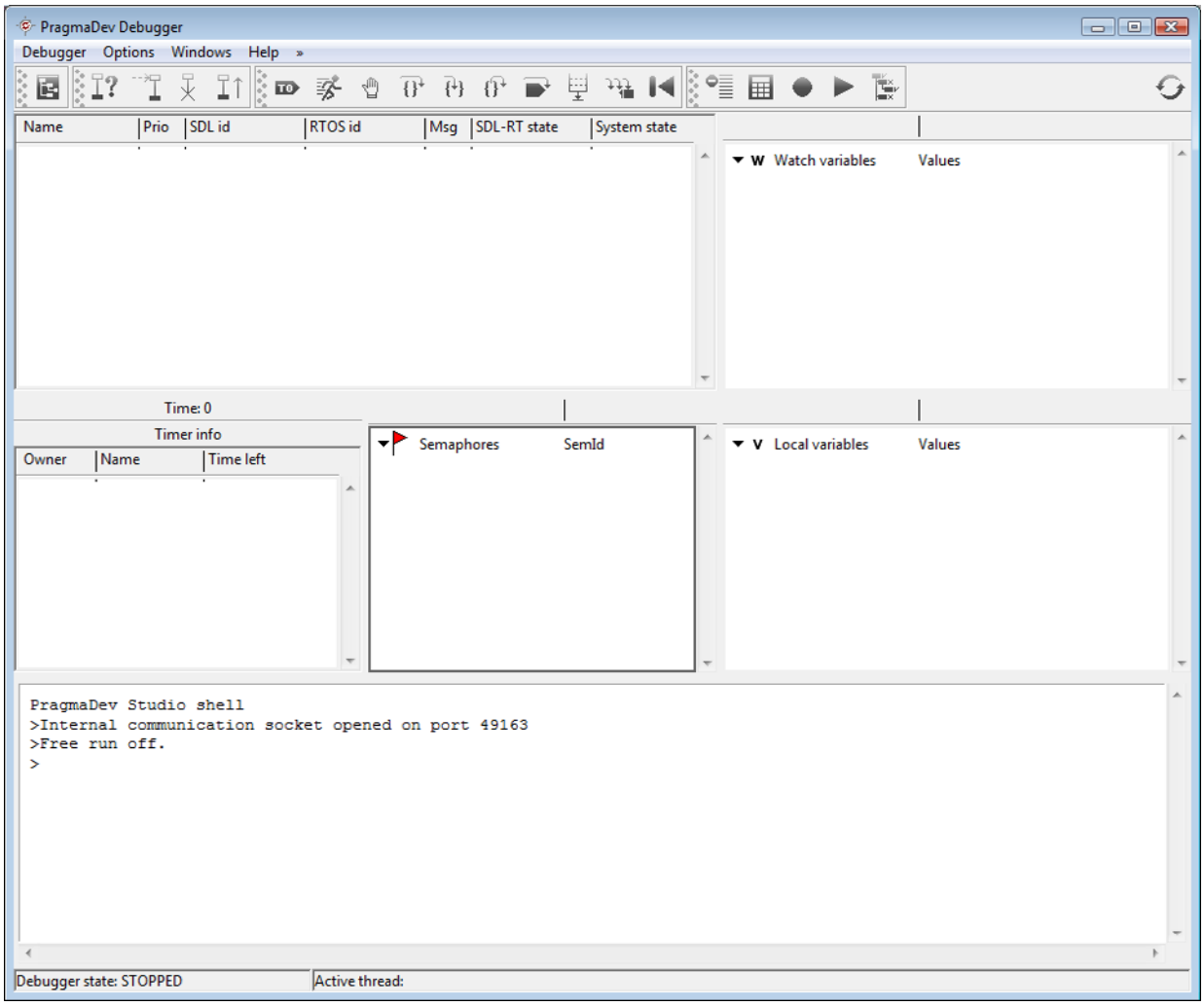


Once the error have been corrected the *Generation / compilation output* window should look like this:



5.4.3 The SDL-RT debugger

When launched, the SDL-RT debugger will automatically start and initialize the underlying gdb environment.The *SDL-RT debugger* window is started automatically:



The SDL-RT debugger window

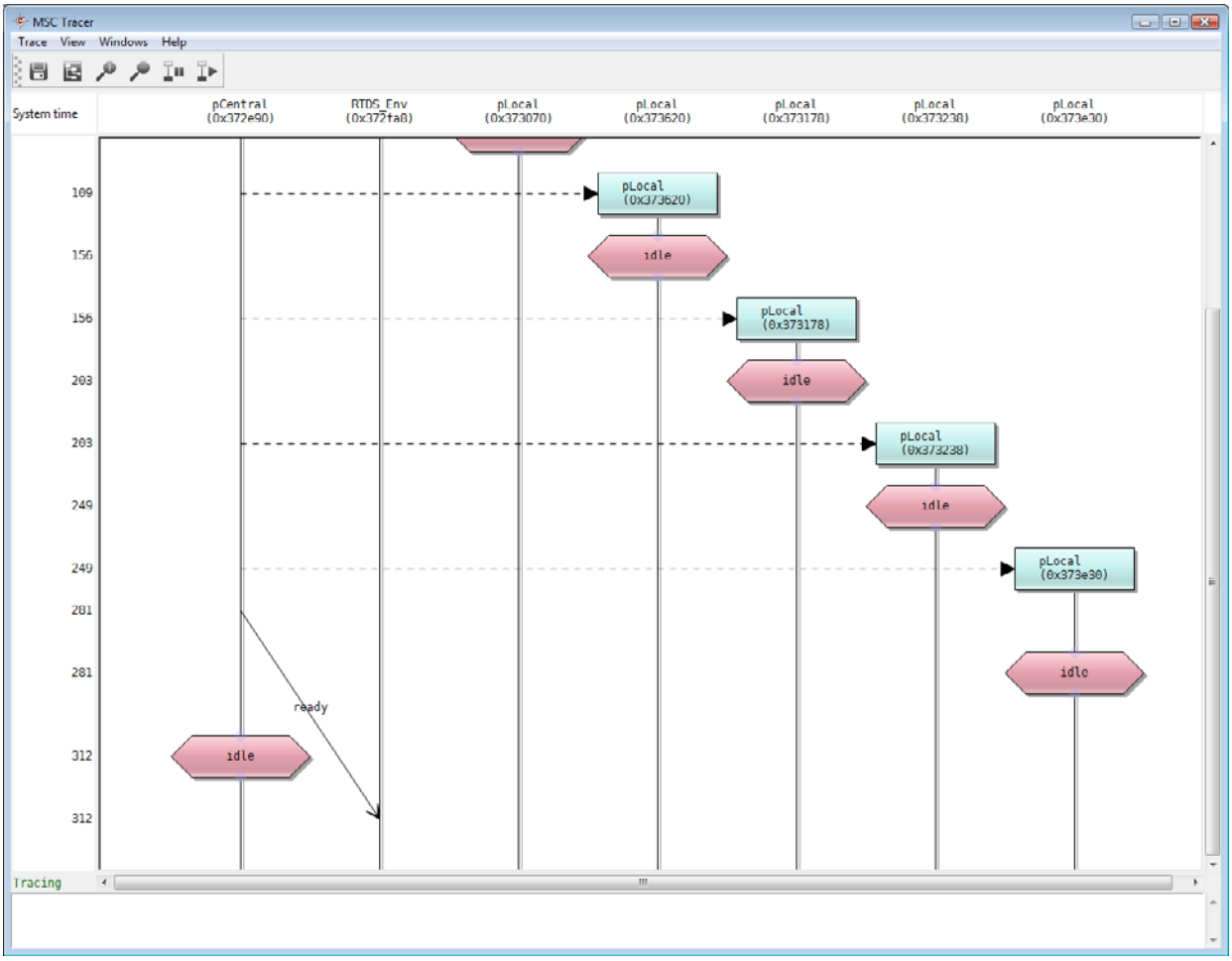
The SDL-RT debugger is basically a debugger with graphical integration. This window provides snapshots of the overall system.

First we want an MSC trace to see what is happening in the system. Click on the *Start MSC trace* quick button: 

An *MSC Tracer* window appears. Note it is not an MSC editor window; the MSC Tracer has been optimized for performance and the displayed trace can not be edited.

Let's start the system; click on 'run' quick button: 

Let the system run until all `pLocal` processes are created by `pCentral` and their start transition executed:



MSC trace

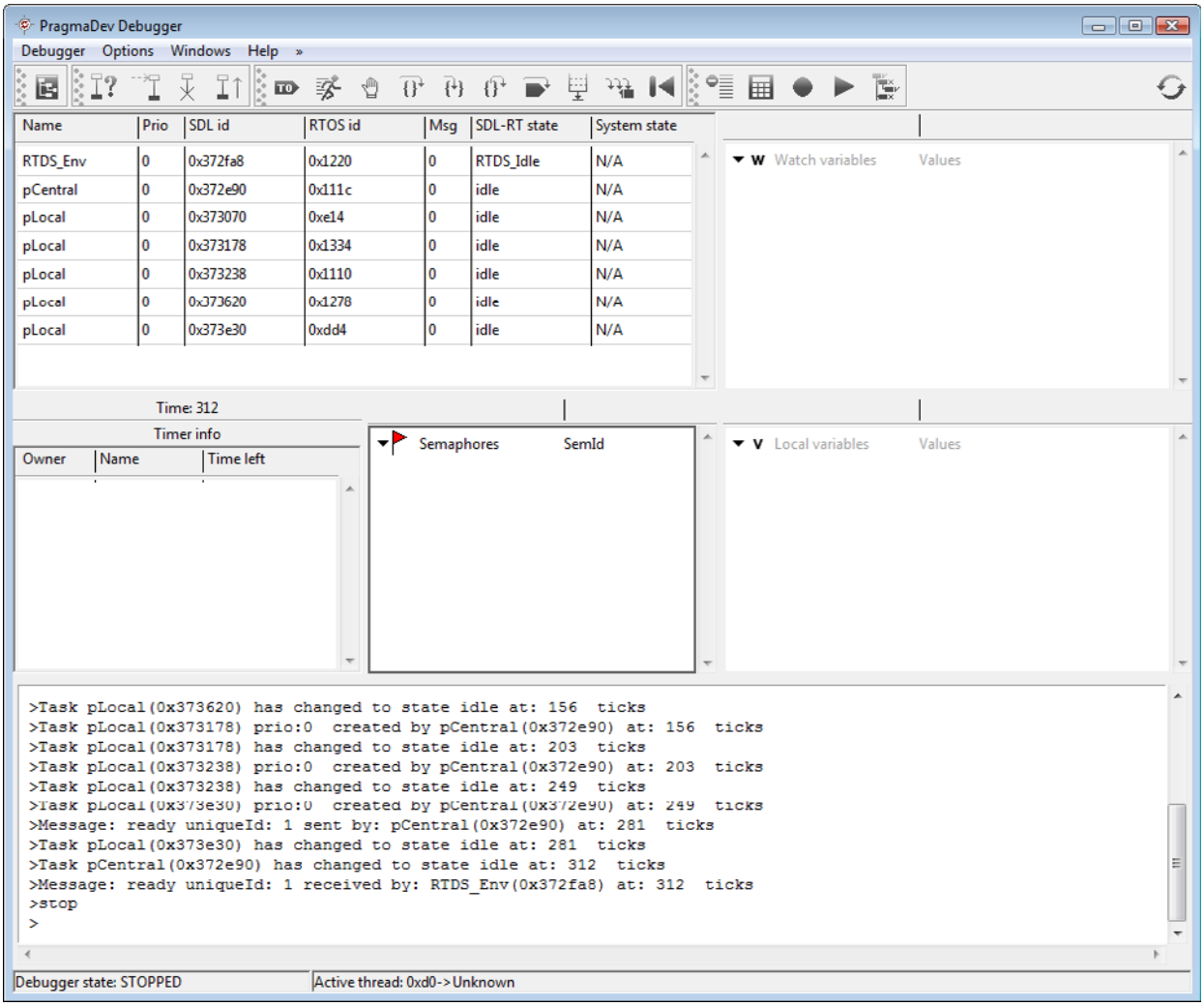
Note you can detach the execution button bar by dragging it away from the debugger window by its header (the zone looking like this:).

The environment is represented by `RTDS_Env` process. It is automatically generated by PragmaDev Studio when debugging to represent all external modules. When generating target code it will of course disappear.

Process `pCentral` dynamically creates 5 instances of `pLocal`, sends `ready` message to the environment and goes to `idle` state. Each `pLocal` instance then go to `idle` state. On the left is the value of the system time.

Click on the Stop button to break execution:

The SDL-RT debugger window shows the list of processes with their names (all the same in our case), priority, process id, queue id, number of messages in their respective message queues, SDL-RT internal state as we defined in the diagrams, and the RTOS internal system state if available.



SDL-RT debugger window

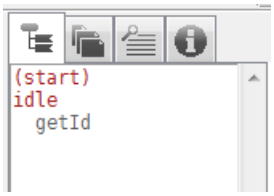
It can also display information regarding semaphores and timers, local variables and watch variables. The SDL-RT debugger shell gives a textual trace of the events displayed in the MSC.

We are now going to set a breakpoint, simulate a user using one phone to call another one and step in process `pCentral`.

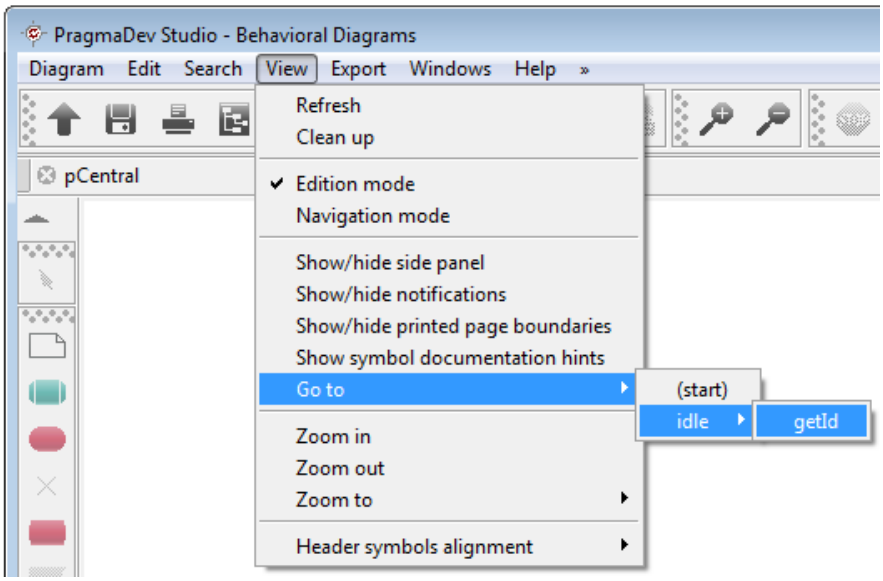
Let's put a breakpoint in pCentral process. To do so:

- Open `pCentral` process from the *Project manager*. Double-clicking on its name in the running instances list in the debugger window works too.

- Go to transition getId in state idle using the *transition browser* window on the right:

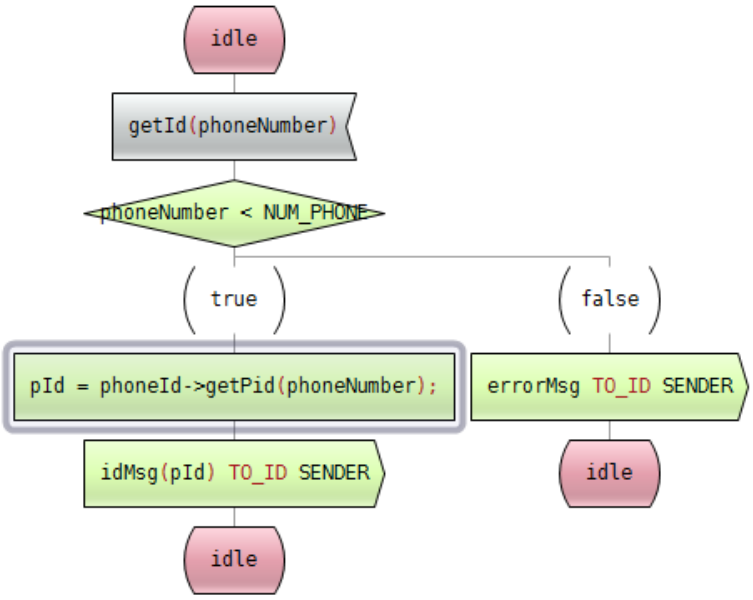


or the view menu:

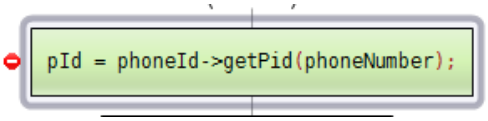


The *transition browser* of the *View/Go to* menu will list all SDL-RT states and all transitions in each state to quickly navigate through the system. In our case there is only one state and one transition.

- click on the symbol just after the decision:



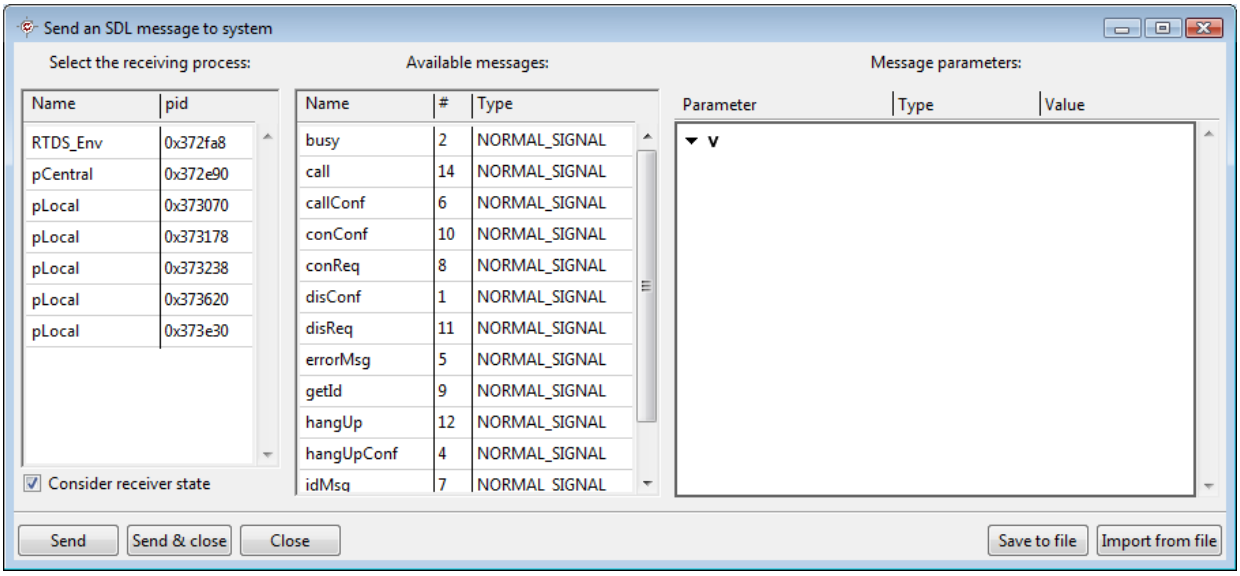
- click on  or go to *Debug / Set breakpoint* menu in the *SDL-RT editor*. A breakpoint symbol will be displayed on the side of the selected symbol:



We will now simulate an incoming message from a user. Please note this feature is not available in the *Tasking* integration nor in the *XRAY* integration because these C debuggers can neither execute function calls on target nor simulate interrupts.

- Go to the *SDL-RT debugger* and click on "Send an SDL message to the running system" quick-button 

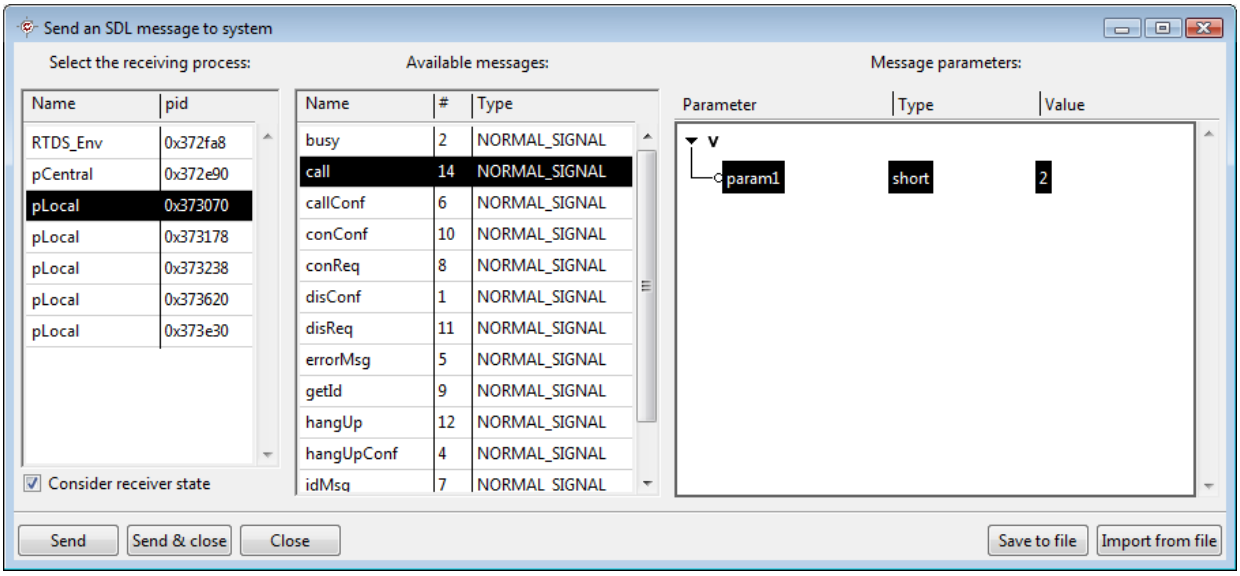
- The *Send an SDL message* window shows up:



Send an SDL message window

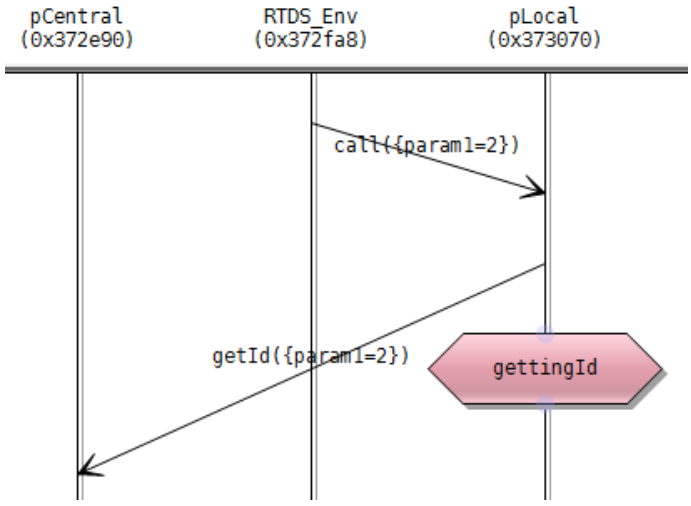
On the left are listed all possible receiving processes, in the middle all possible messages, i.e. all messages used in the SDL-RT system, and on the right the value of the parameters associated with the selected message.

- Prepare the message to be sent:

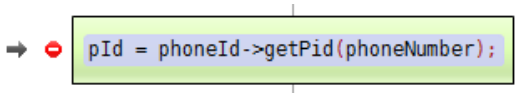


The message to be sent is `call`, with the parameter value 2. We will choose the first listed instance of `pLocal` as the receiver. You might want to write down it's PID, since we will send it another message later.
NB: it may happen that the first listed instance of `pLocal` has been assigned the phone number 2, in which case the following will not work, as the phone will try to call itself. If this happens, just send again a new `call` message to another `pLocal` instance.

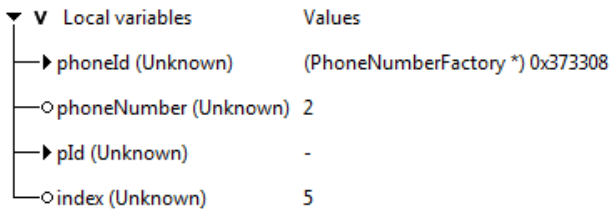
- Click on *Send & close* button.
- Click on *Run* in the *SDL-RT debugger* window,
- The following trace appears in the MSC:



- The SDL-RT editor pops up where the breakpoint was set with the break line in yellow:



The values of the process local variables are automatically displayed in the SDL-RT debugger. For example the `phoneNumber` variable value is 2:

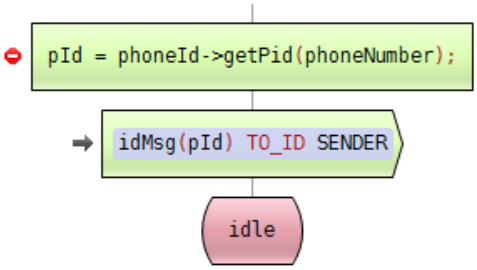


Local variables values

- Click on *Auto step until next graphical symbol* button:

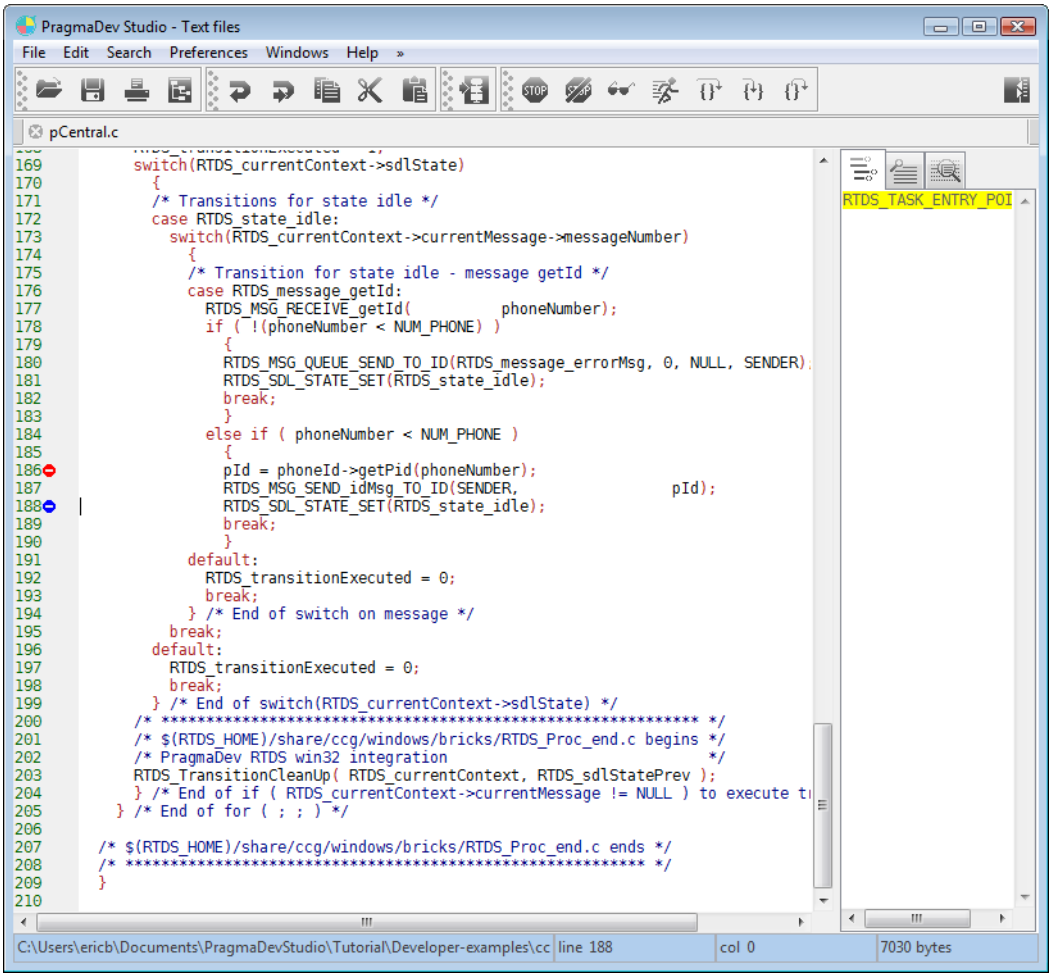
That quick button will actually step in the C code until the line in the C file is generated from a graphical symbol. In our case it will only step once.

- The SDL-RT has moved to the next symbol:



- Click on *Step over* button:

- This will step in the C code as any normal C debugger. The text editor opens and displays the next line to execute in the generated C file. Have a look at the generated code to see how legible it is.




```
169       switch(RTDS_currentContext->sdlState)
170         {
171         /* Transitions for state idle */
172         case RTDS_state_idle:
173           switch(RTDS_currentContext->currentMessage->messageNumber)
174             {
175             /* Transition for state idle - message getId */
176             case RTDS_message_getId:
177               RTDS_MSG_RECEIVE_getId(          phoneNumber);
178               if ( !(phoneNumber < NUM_PHONE) )
179                 {
180                 RTDS_MSG_QUEUE_SEND_TO_ID(RTDS_message_errorMsg, 0, NULL, SENDER);
181                 RTDS_SDL_STATE_SET(RTDS_state_idle);
182                 break;
183                 }
184               else if ( phoneNumber < NUM_PHONE )
185                 {
186                 pId = phoneId->getPid(phoneNumber);
187                 RTDS_MSG_SEND_idMsg_TO_ID(SENDER,                pId);
188                 RTDS_SDL_STATE_SET(RTDS_state_idle);
189                 break;
190                 }
191             default:
192               RTDS_transitionExecuted = 0;
193               break;
194             } /* End of switch on message */
195           break;
196         default:
197           RTDS_transitionExecuted = 0;
198           break;
199         } /* End of switch(RTDS_currentContext->sdlState) */
200       /* ********************************************************** */
201       /* $(RTDS_HOME)/share/ccg/windows/bricks/RTDS_Proc_end.c begins */
202       /* PragmaDev RTDS win32 integration                           */
203       RTDS_TransitionCleanUp( RTDS_currentContext, RTDS_sdlStatePrev );
204       } /* End of if ( RTDS_currentContext->currentMessage != NULL ) to execute t
205     } /* End of for ( ; ; ) */
206
207   /* $(RTDS_HOME)/share/ccg/windows/bricks/RTDS_Proc_end.c ends */
208   /* ********************************************************** */
209   }
210
```

Note it is possible to switch from the SDL source to the generated C source back and forth with the *Search / Go to SDL symbol* and *Search / Go to generated source* menus.

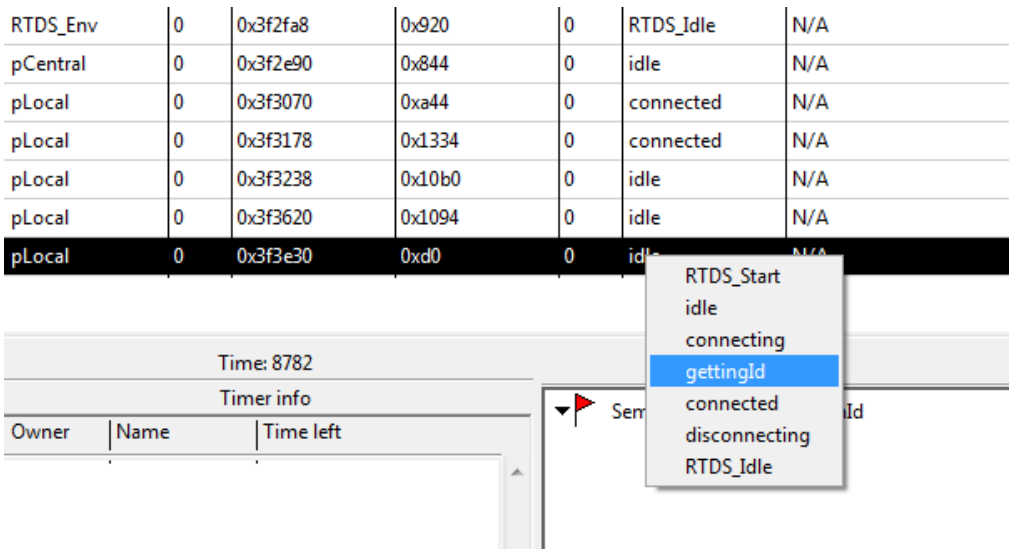
- Let the system finish its job: click on Run button.

- Stop the system once it has finished execution:  . The process list is updated with their new states:

Name	Prio	SDL id	RTOS id	Msg	SDL-RT state	System state
RTDS_Env	0	0x3f2fa8	0x920	0	RTDS_Idle	N/A
pCentral	0	0x3f2e90	0x844	0	idle	N/A
pLocal	0	0x3f3070	0xa44	0	connected	N/A
pLocal	0	0x3f3178	0x1334	0	connected	N/A
pLocal	0	0x3f3238	0x10b0	0	idle	N/A
pLocal	0	0x3f3620	0x1094	0	idle	N/A
pLocal	0	0x3f3e30	0xd0	0	idle	N/A

Two instances of the `pLocal` process are connected.

- It is also possible to change the state of a process. To do so right click on the state you want to change and a drop down menu will list all possible states. Let's change the state of the last instance of `pLocal` to `gettingId` for example:

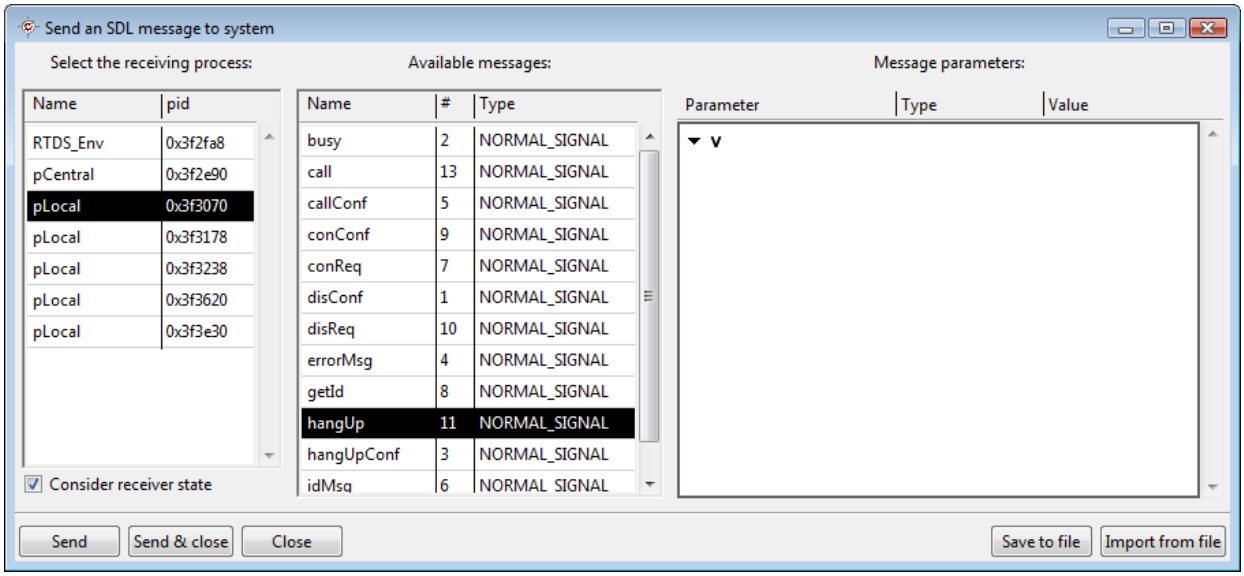


The state is changed on the target and the process list will refresh:

Name	Prio	SDL id	RTOS id	Msg	SDL-RT state	System state
RTDS_Env	0	0x3f2fa8	0x920	0	RTDS_Idle	N/A
pCentral	0	0x3f2e90	0x844	0	idle	N/A
pLocal	0	0x3f3070	0xa44	0	connected	N/A
pLocal	0	0x3f3178	0x1334	0	connected	N/A
pLocal	0	0x3f3238	0x10b0	0	idle	N/A
pLocal	0	0x3f3620	0x1094	0	idle	N/A
pLocal	0	0x3f3e30	0xd0	0	gettingId	N/A

It is recommended to be cautious when changing a task state that way...

- We will now disconnect the 2 instances of `pLocal`. Click on the Send button TO :



  Select the `hangUp` message with the receiver set to the same `pLocal` instance as you selected as the receiver of the `call` message earlier (check with its PID). Then click on Send & close.

- We will now use the key SDL-RT event stepping button: 

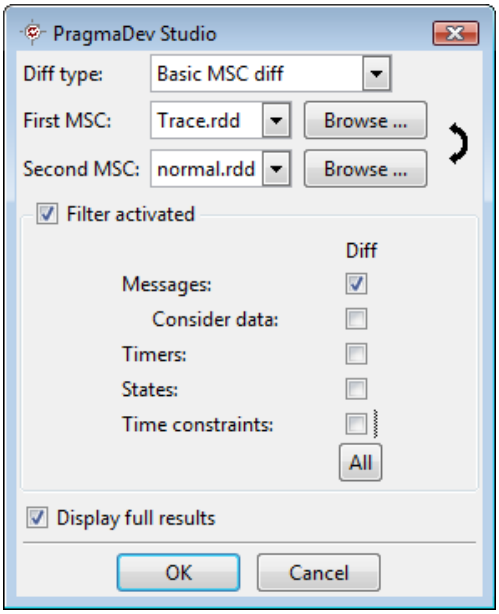
  This feature runs the system until the next SDL-RT event such as sending or receiving a message, changing SDL-RT state, starting or cancelling a timer... Click on the button and you will see the system executing until the next SDL-RT event and then stop. This is a very nice feature when debugging for the first time.

5.4.4 Verifying the behavior

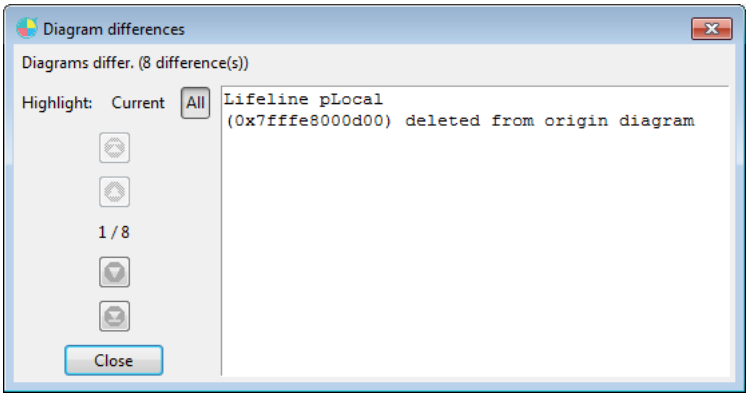
We will now check if the behavior is the one we expected in the first place. To do so we will use the MSC diff feature.

- Once the execution is finished, close the MSC Tracer and save the trace.
- Go to the Project manager and open the trace.

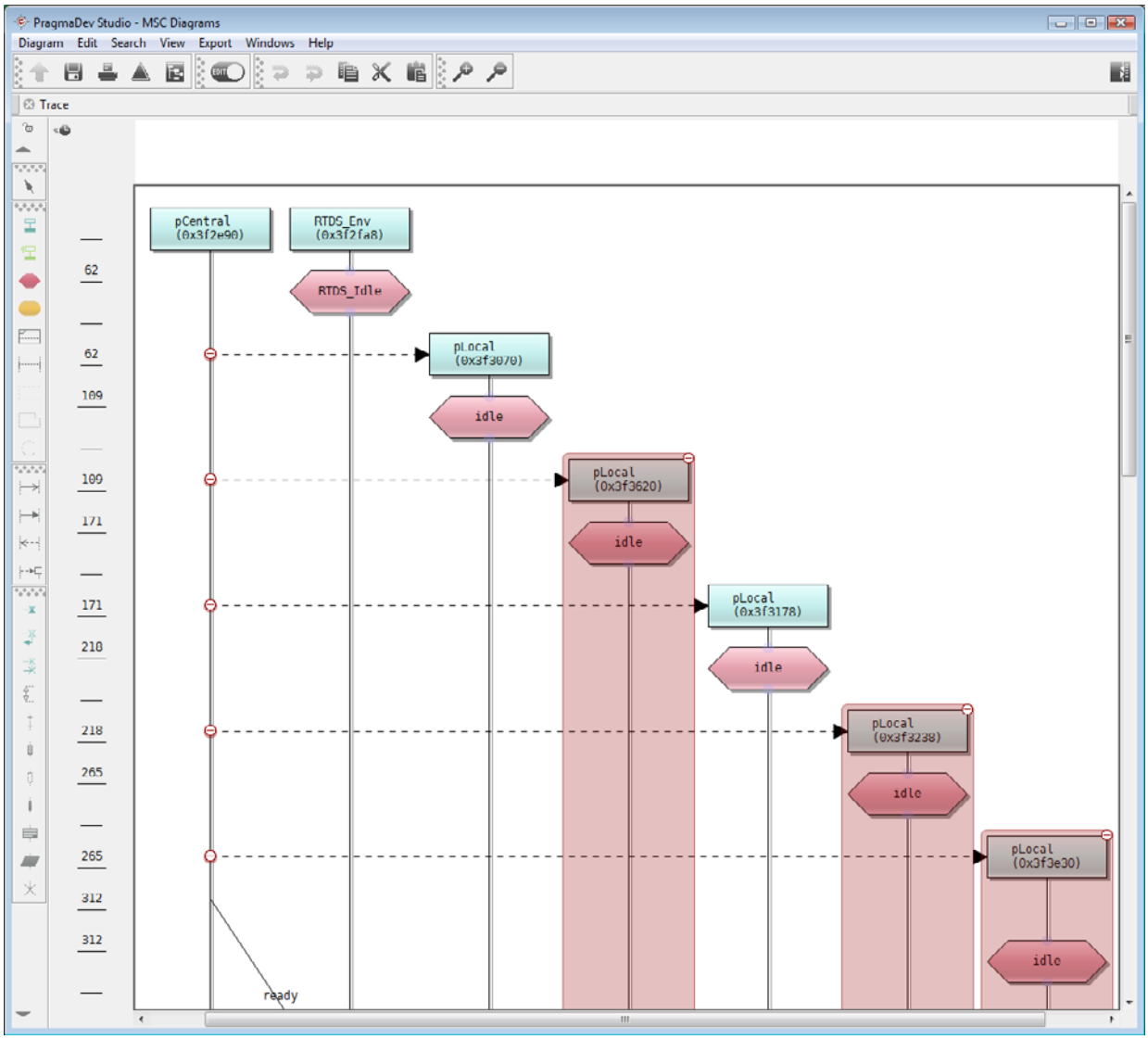
- Go to the *Diagram / Compare with other diagram...* menu to get the MSC Diff configuration window and set it up as described below:



The first MSC is the execution trace and the second MSC is the normal MSC we have written in the first place. Since normal MSC was not supposed to be thoroughly detailed we will only show and compare messages without considering their parameters. Click *Ok* and you should get the following:



Here you can navigate through the differences, and by selecting *All*, then all of them will be shown in the diagram:



The differences between the MSCs are the dynamic task creation of the `pLocal` instances. After that the exchange of messages are the same between the dynamic trace and the specification. The SDL-RT system is therefore conform to the normal MSC specification.

You are done with a very simple SDL-RT debugging session. If you want more, do your own system or run the examples delivered in the distribution to see how to manipulate:

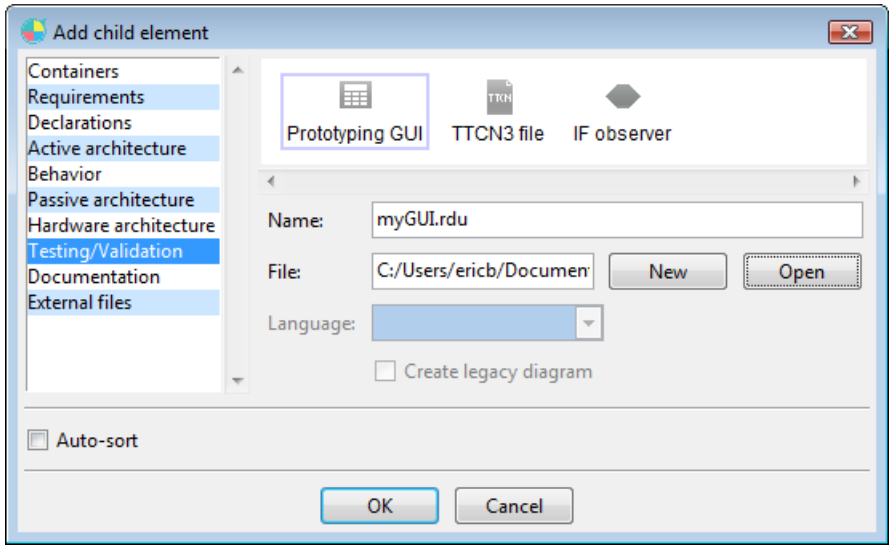
- timers,
- semaphores,
- external C header files,
- global variables.

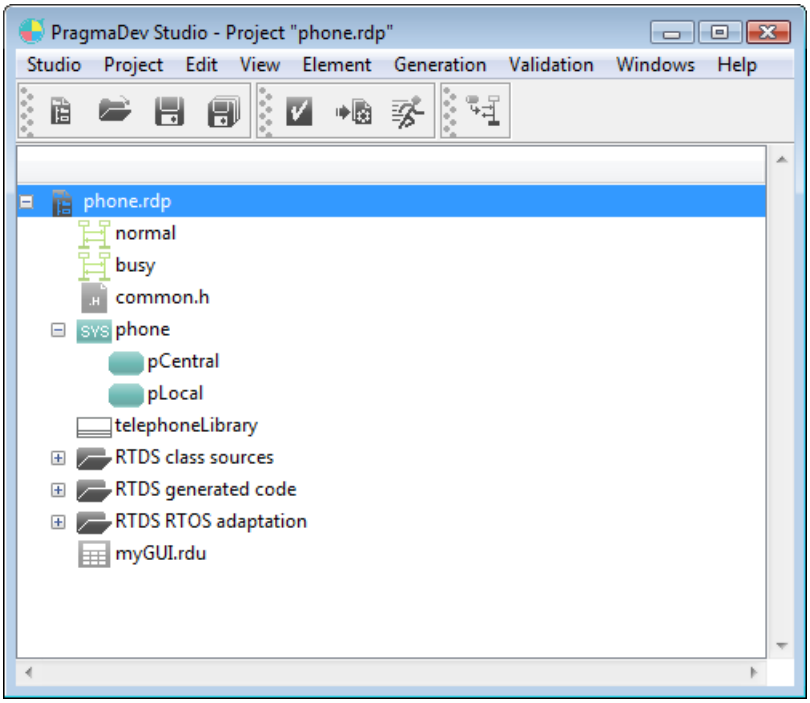
5.5 - Prototyping GUI

PragmaDev Studio has a built in support to design simple prototyping interface to ease testing. We will build a very simple one for our phone system to demonstrate its capabilities.

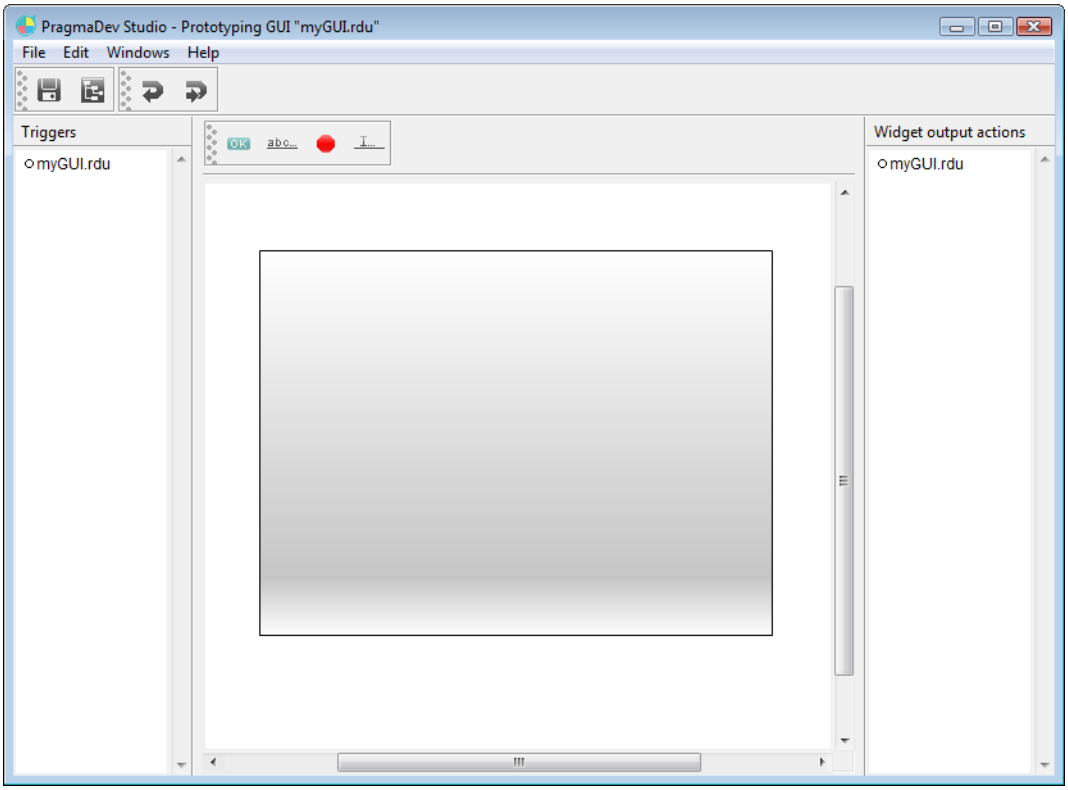
5.5.1 GUI editor

Add a Prototyping GUI node in the project and open it:

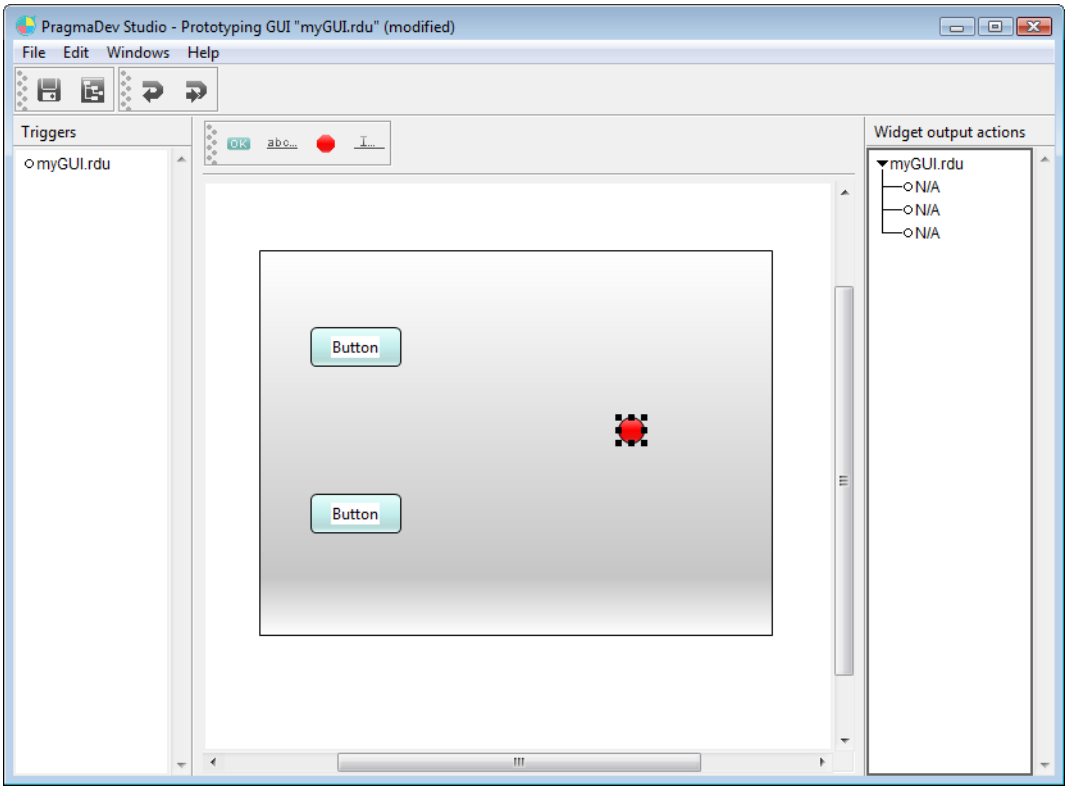




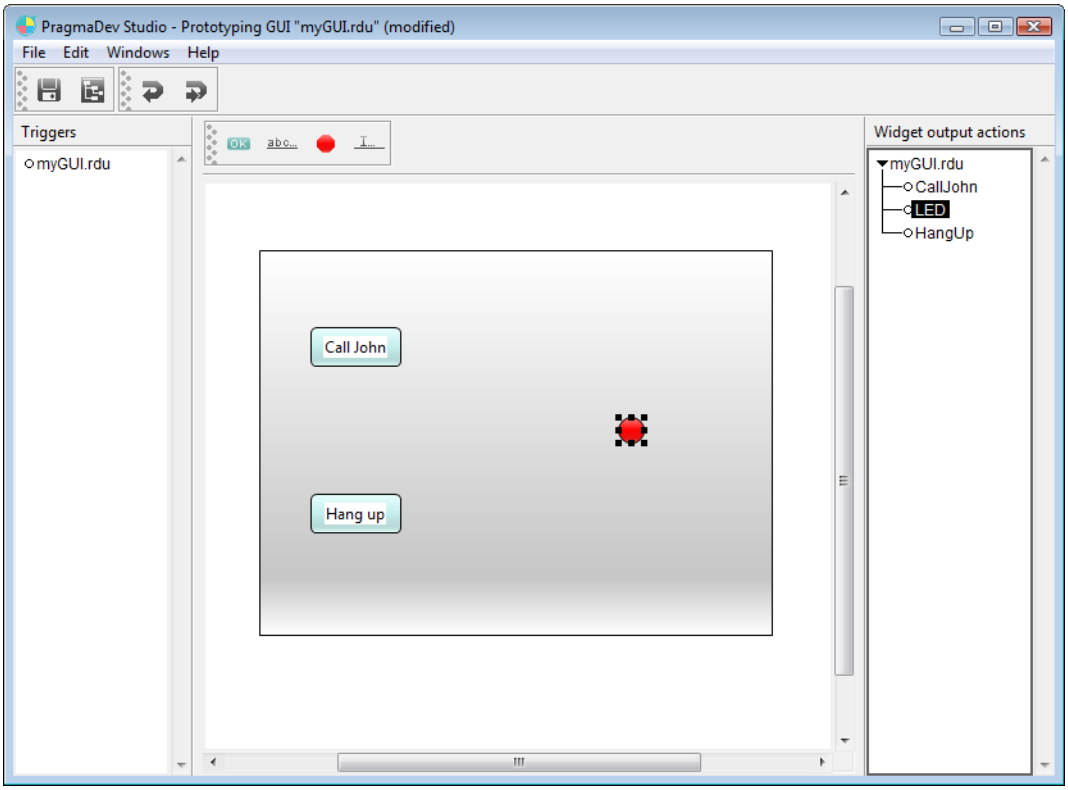
The left panel contains the incoming triggers for the GUI, the central panel the GUI itself, and the right panel the outgoing message from the GUI:



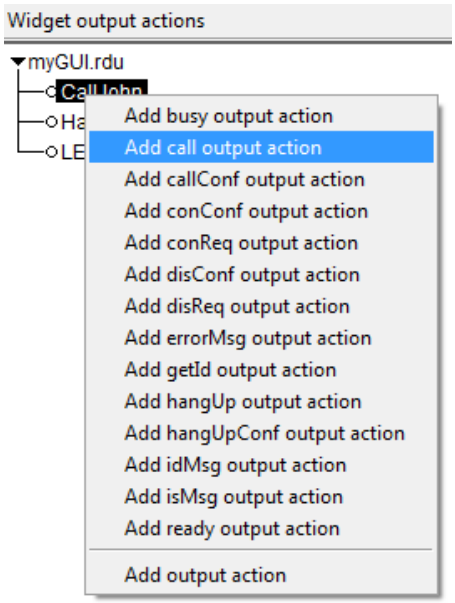
Let's add 2 buttons and one LED:



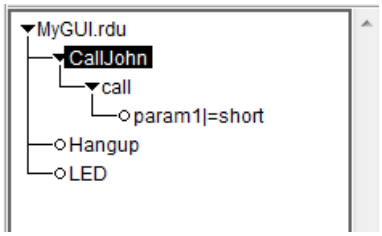
Change their display value in the central panel and their widget name in the right panel in order to recognize them:



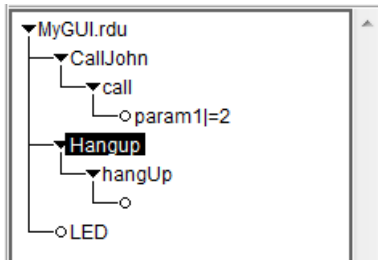
Let's say that when the user clicks on the "Call John" button, the GUI sends a `call` message with parameter set to "2". Select the CallJohn widget on the right panel and right click:



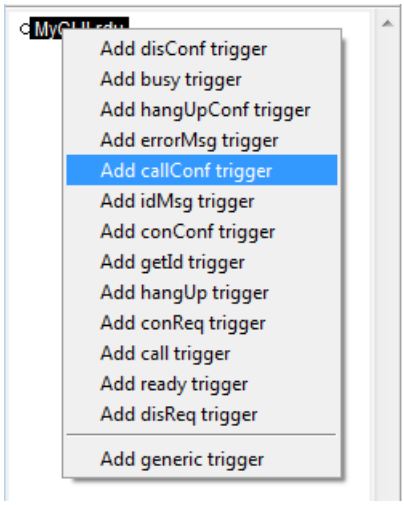
All the available messages in the system are then listed. Select `call` and expand the created sub-tree. The parameters are listed with their corresponding type:



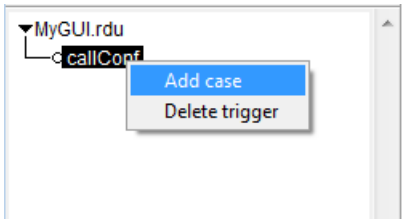
Let's say the parameter value is '2' and let's send sHangUp without any parameter when clicking on Hangup:



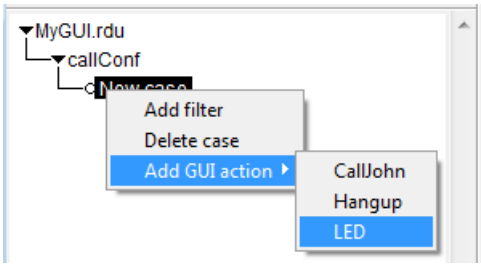
On the left panel now, we will add a new trigger. A trigger performs some action on the widgets whenever a message is sent out of the system.. Select the top of the tree and right click to get a list of all the possible triggers:



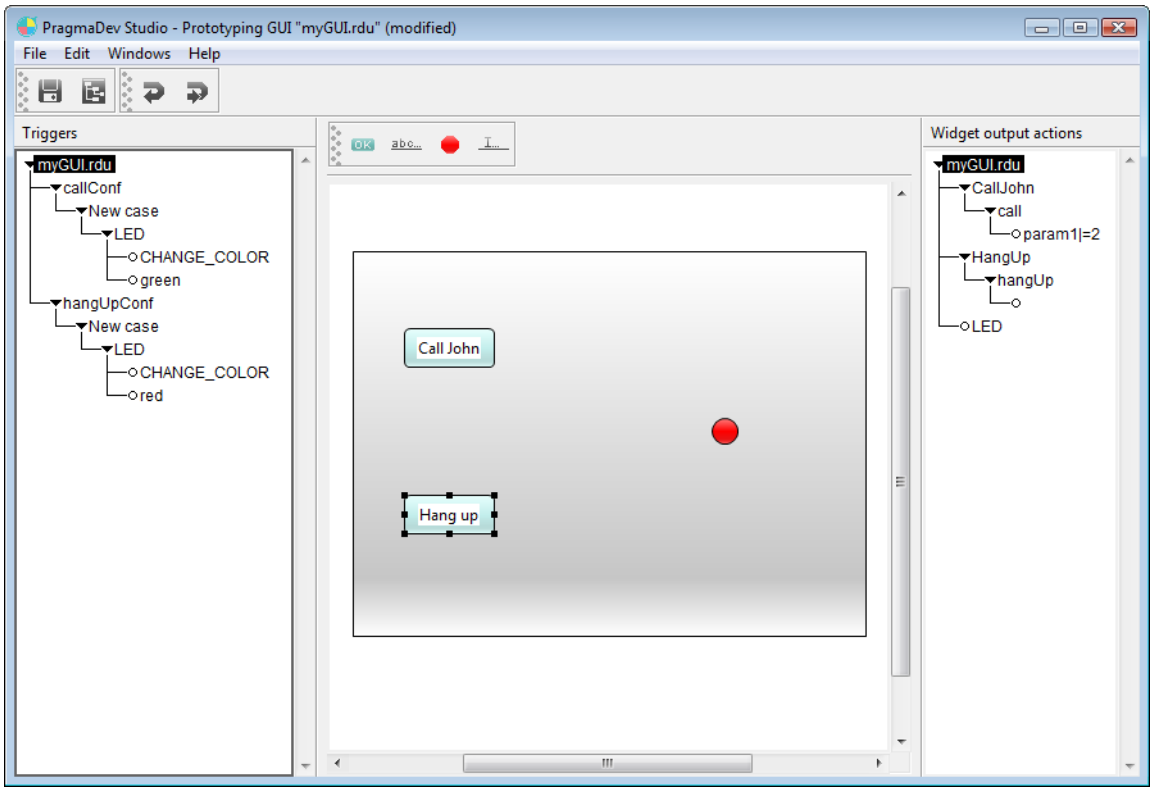
Let's add the `callConf` trigger. When a trigger is received by the GUI, a case with a set of filters is verified. Let's add a new case:



In our case we won't put any filter, we will just change the color of the LED:



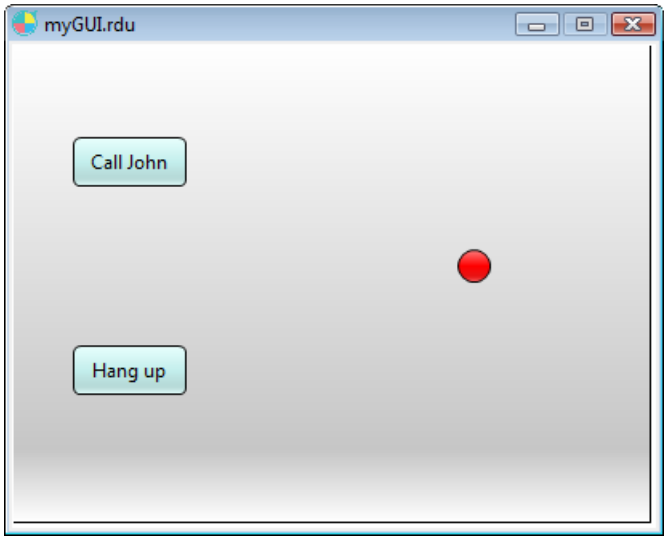
The default action is `DISPLAY`; you'll have to right click on it to change it to `CHANGE_COLOR`. The node in the tree under the action specifies the new color for the widget. It is possible to directly name the basic colors, otherwise the RGB hexa code can be used (e.g #FF8000 is this color). Let's put the LED back to red when we receive a `HangUp` confirmation and we're done:



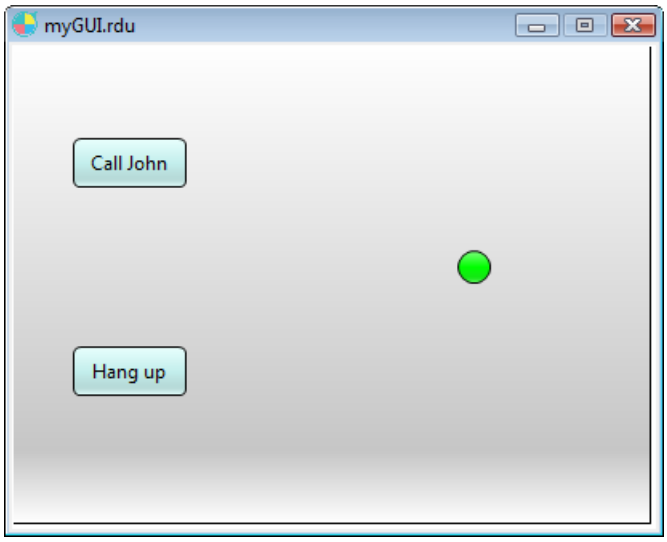
5.5.2 Simulation

Let's start the Debugger again and click on the Start prototyping GUI quick button: 

The GUI will start and connect automatically to the system:



Start an MSC trace and run the system. Click on the "Call John" button, that should send the `call` message with parameter value set to `2`, the `callConf` should be received by the GUI, and the LED should be set to green:



For a more advanced GUI, please have a look at the AccessControl system in the Developer example directory.

5.6 - Conclusion

During this tutorial we have been through the basics of the following:

- SDL-RT;
- Project manager;
- SDL-RT editor;
- MSC editor;
- Code generation;

- SDL-RT debug including the three stepping modes:
  - SDL-RT key event,
  - SDL-RT graphical,
  - textual;
- Conformance checking;
- Prototyping GUI.

As a result you saw SDL-RT has the preciseness of C language with the graphical abstraction of SDL and UML perfectly suited to real time systems showing key concepts such as tasks, semaphores, timers, messages in a single consistent development environment.

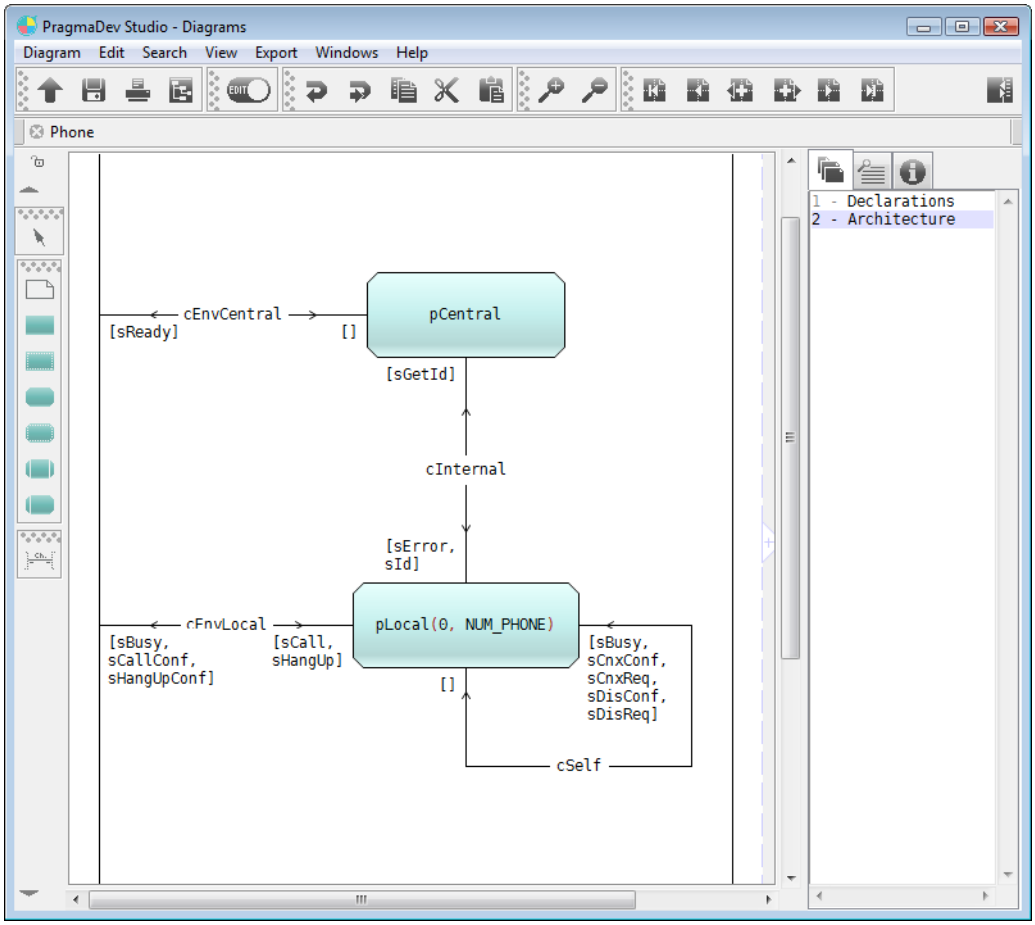
Now it is time for you to work on a real real time system!

6 - Automatic documentation generation

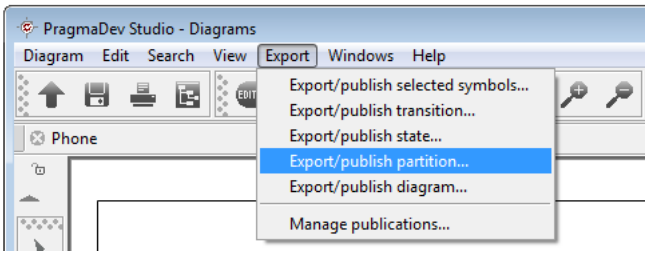
Let's now have a look at how to document our system. PragmaDev Studio has an automatic documentation generator that generates OpenDocument (OpenOffice), RTF (Microsoft Word), HTML, and SGML documents. In this tutorial we will generate an OpenDocument as it is very similar to generating an RTF document.The basic idea is to document while you are modeling your system in SDL, and when you are done just generate the documentation for a word processor or a browser.

6.1 - Publications

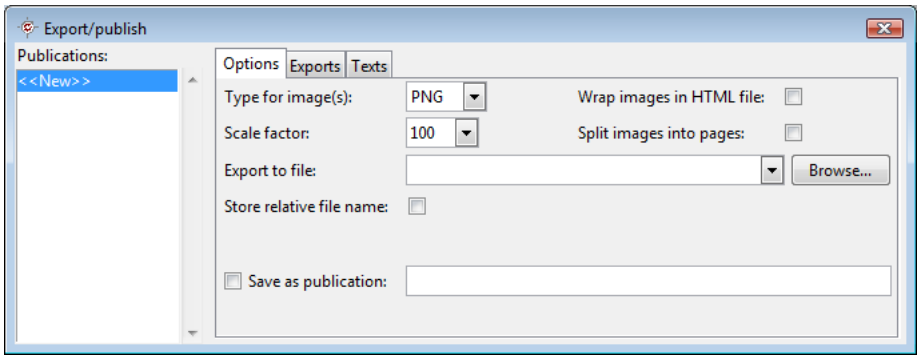
Let's first define what is important to document in our system and put it in a publication. For example let's consider the architecture of the system should be further documented:



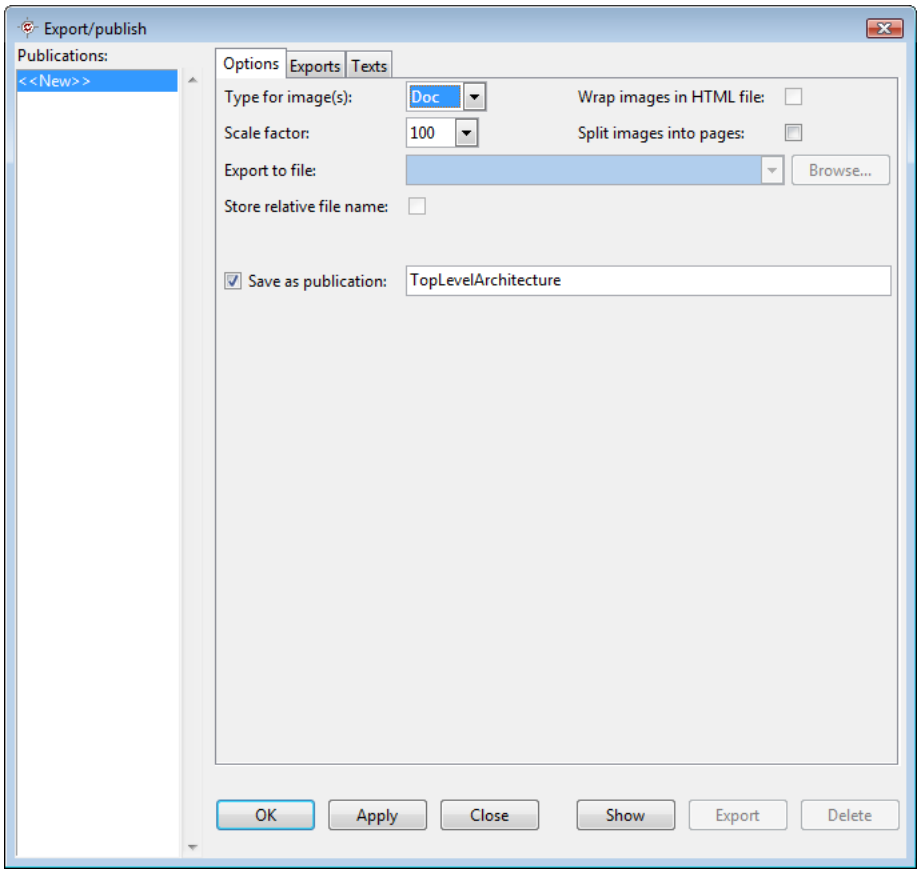
Get to the corresponding diagram and go to the *Export* menu. In that we will export the whole partition, to do so select *Export/publish partition…*



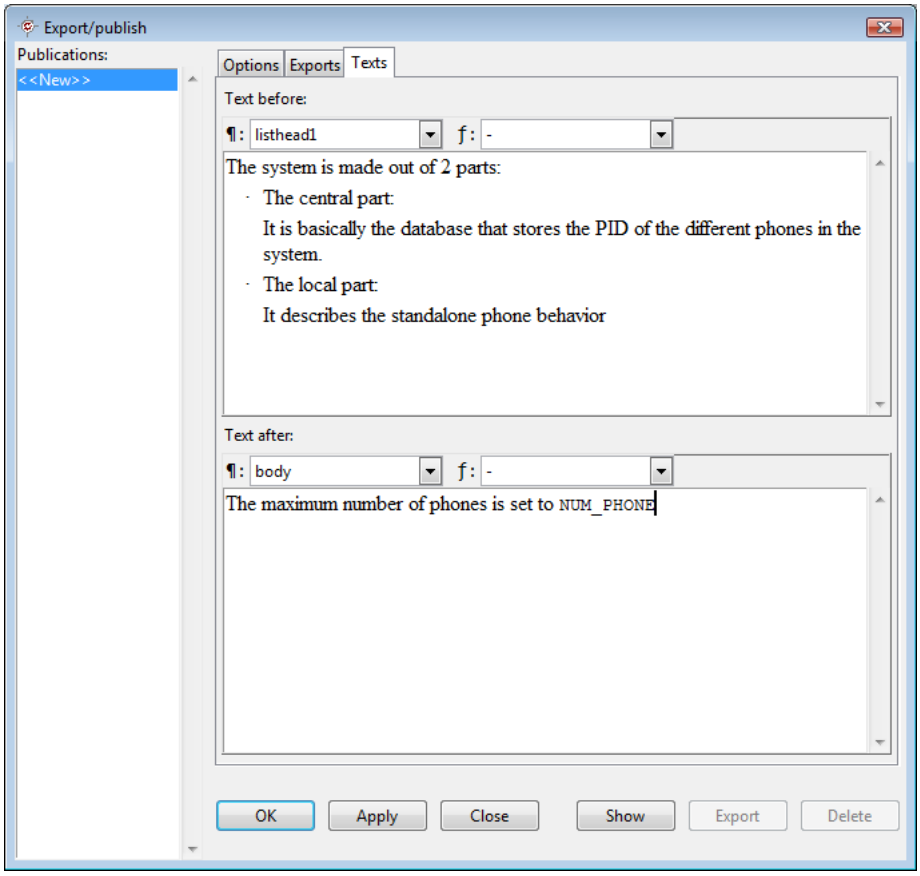
The *Export/publish* window opens:



Let's give our publication a name, the *Doc* type, and check the *Save as publication* box:



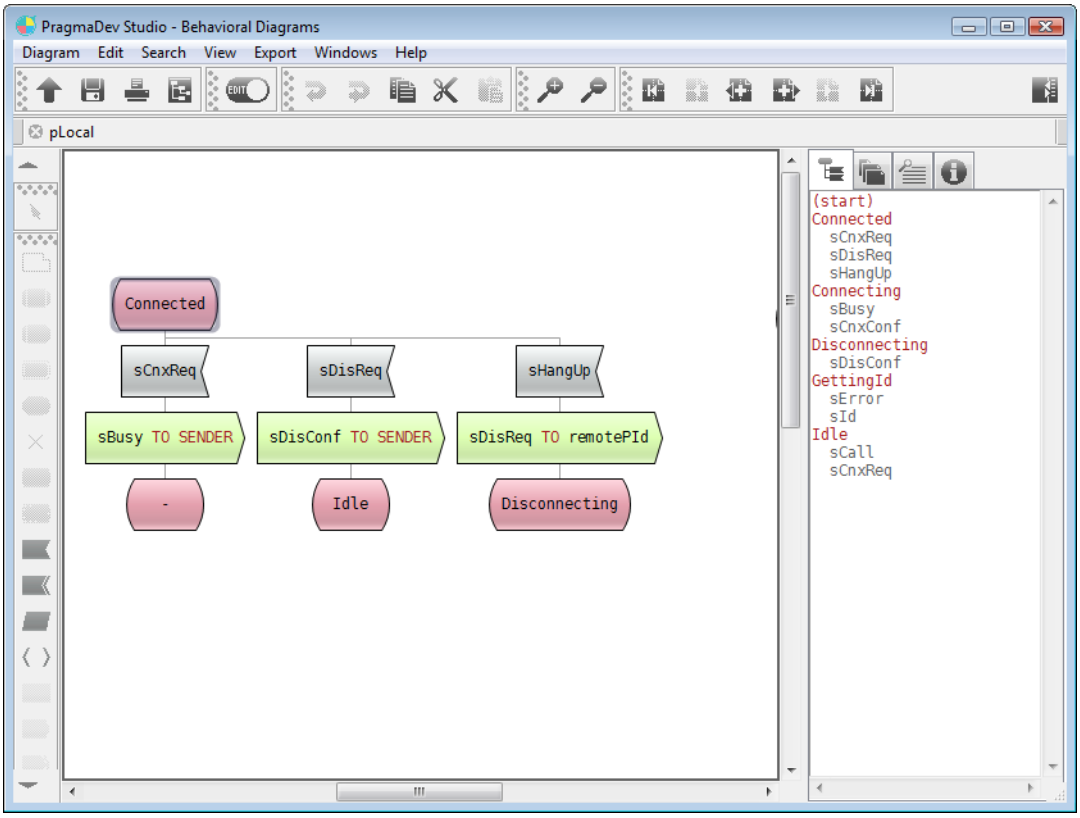
Select the *Texts* tab. The two editors here allow you to type the texts that will be generated before and after the diagram when generating the documentation. Let's type a few words to document the diagram with the pre-defined paragraph and character styles:



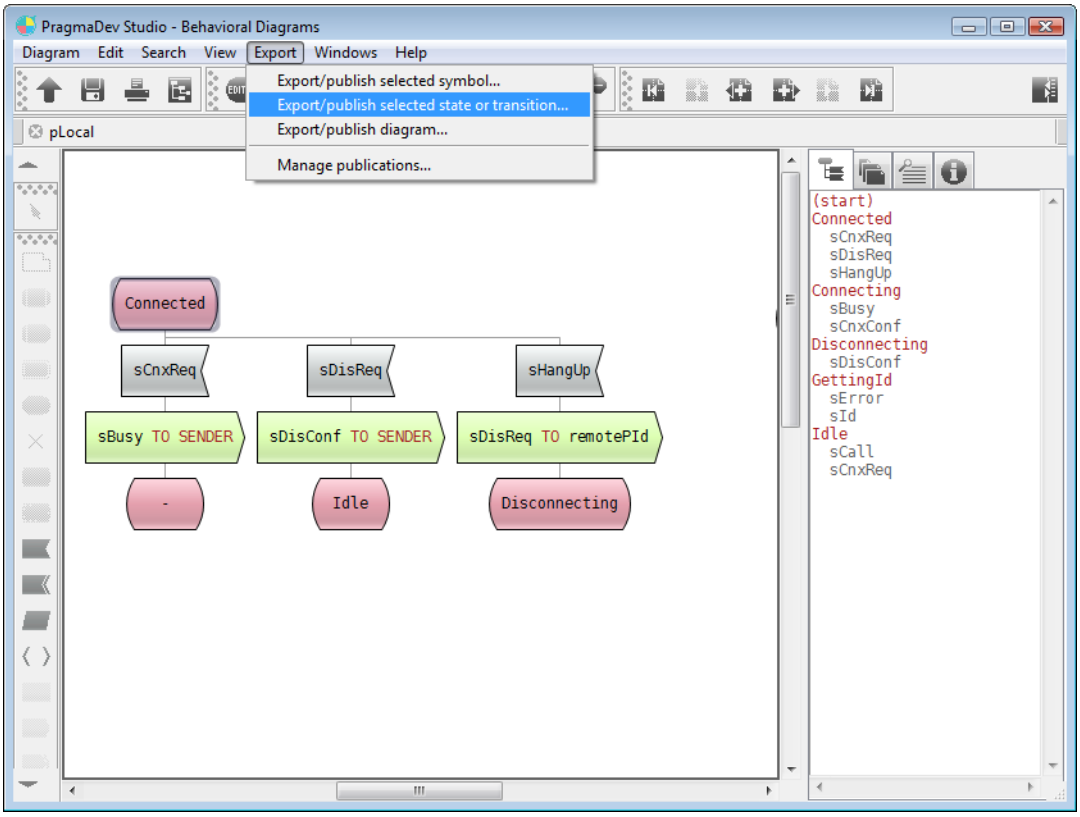
We want the NUM_PHONE identifier to appear as code, and also to be listed in the document index. To do so, select the text, and apply to it the `code-index-entry` character style that is listed in the "`ƒ:`" selector above the text.

When done click *OK*.

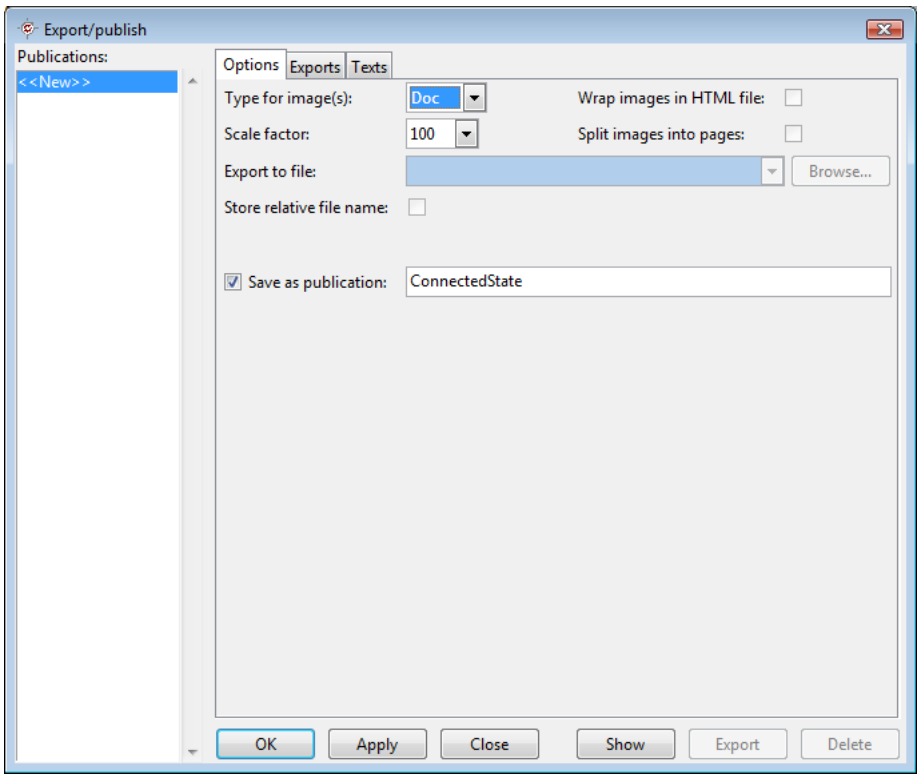
Let's now document a transition. Open the `pLocal` process and go to the `Connected` state:

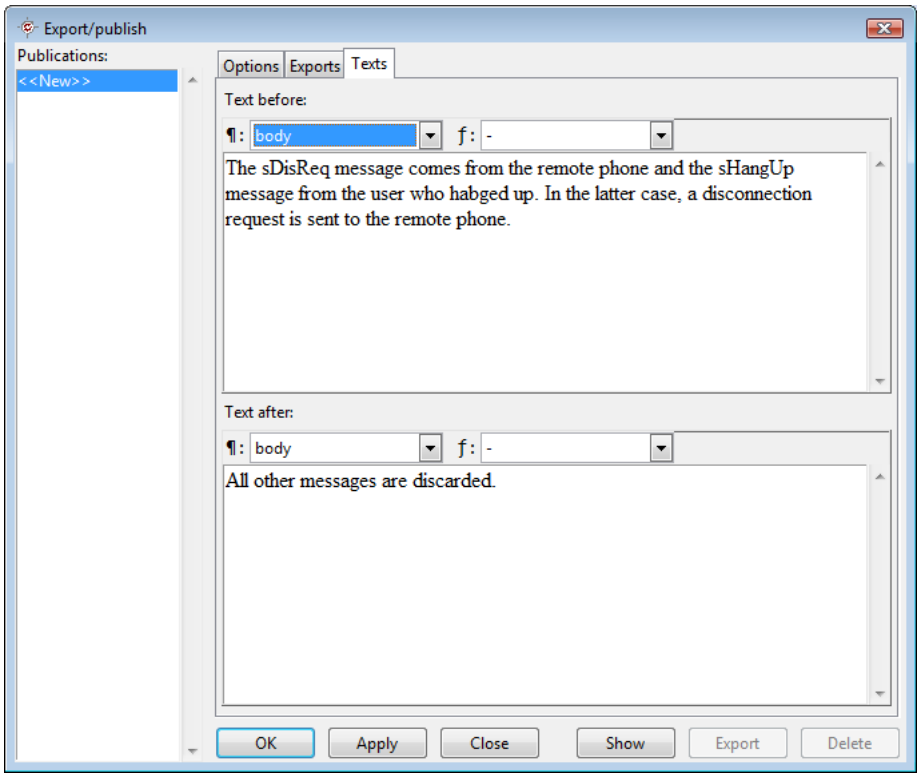


Go to the *Export* menu and select *Export/publish selected state or transition...*:



This will export the state with all the connected inputs as a publication. Let's document the state:

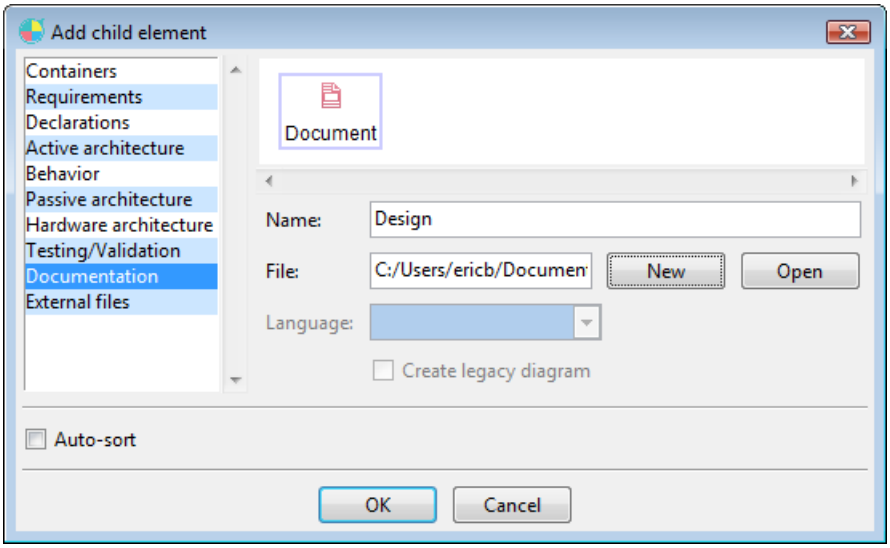


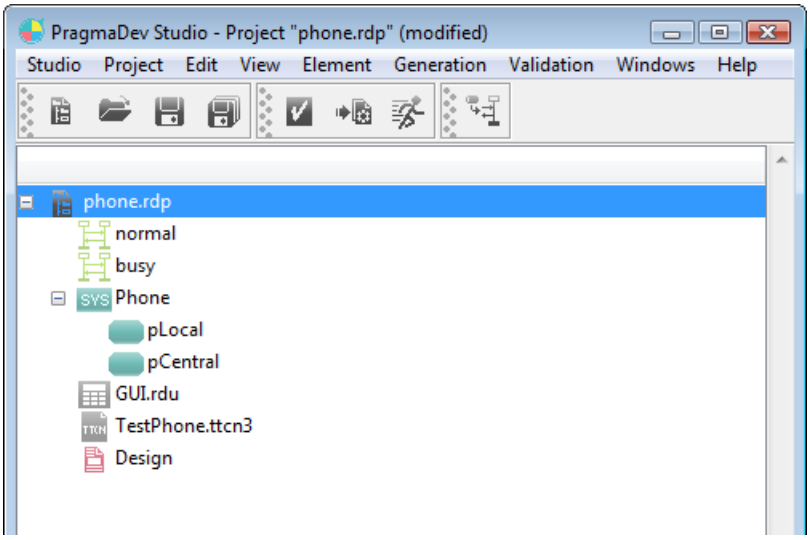


Please note the publications are saved within the diagram so it is important to also save the diagrams.

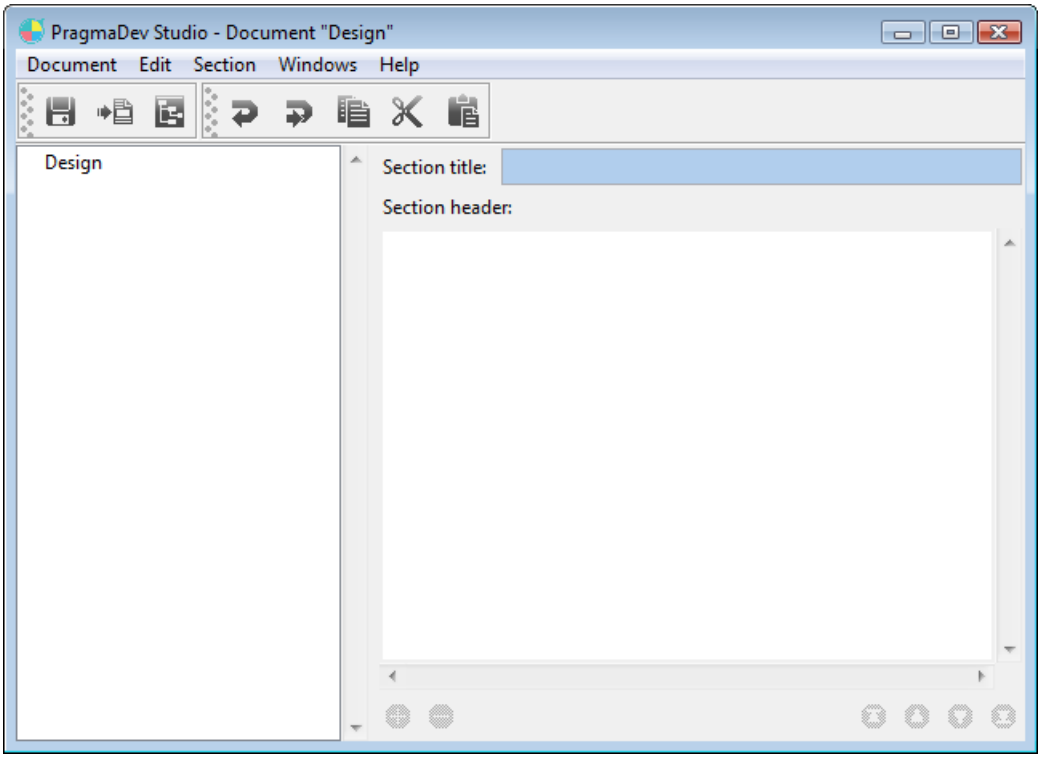
6.2 - Documentation

Let's now go back the *Project manager* and create a new item of type *Document*:

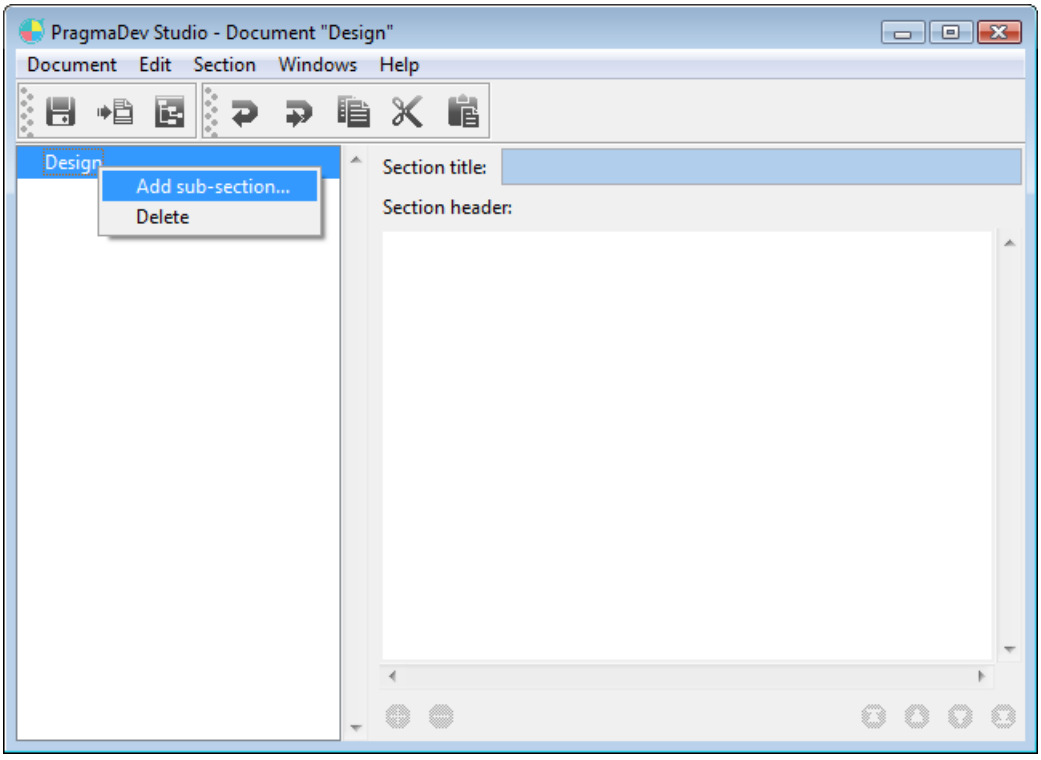




Let's open the *Document* `Design`:

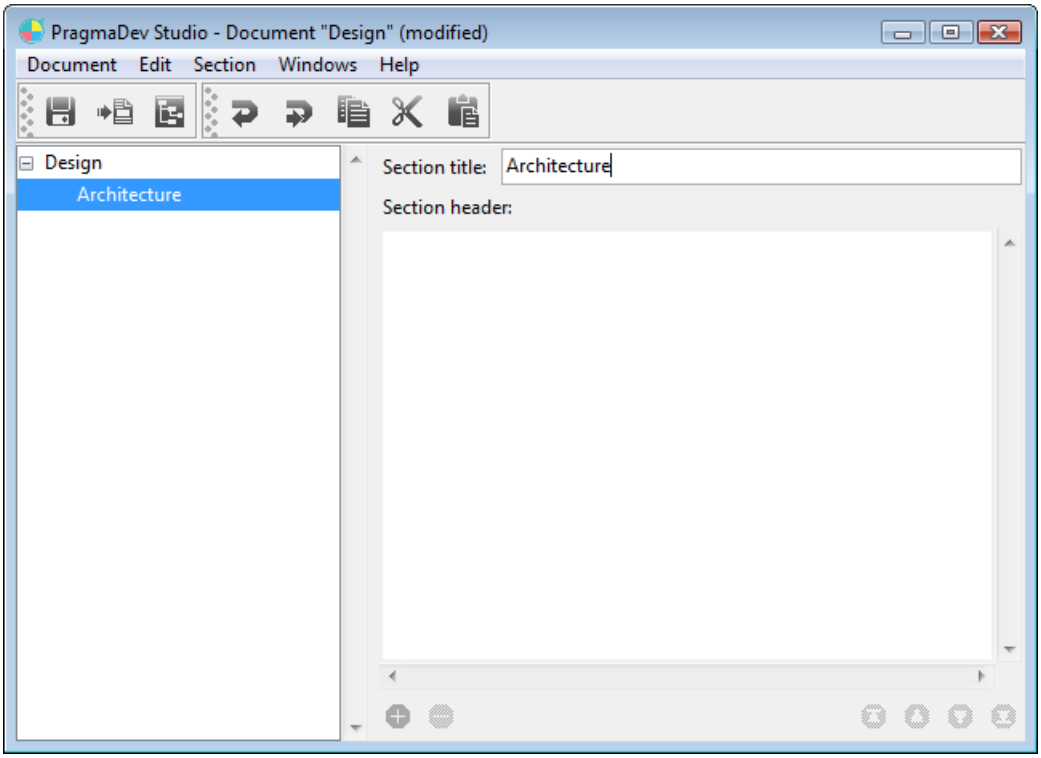


Select MyDocumentation and right-click to add a sub-section:

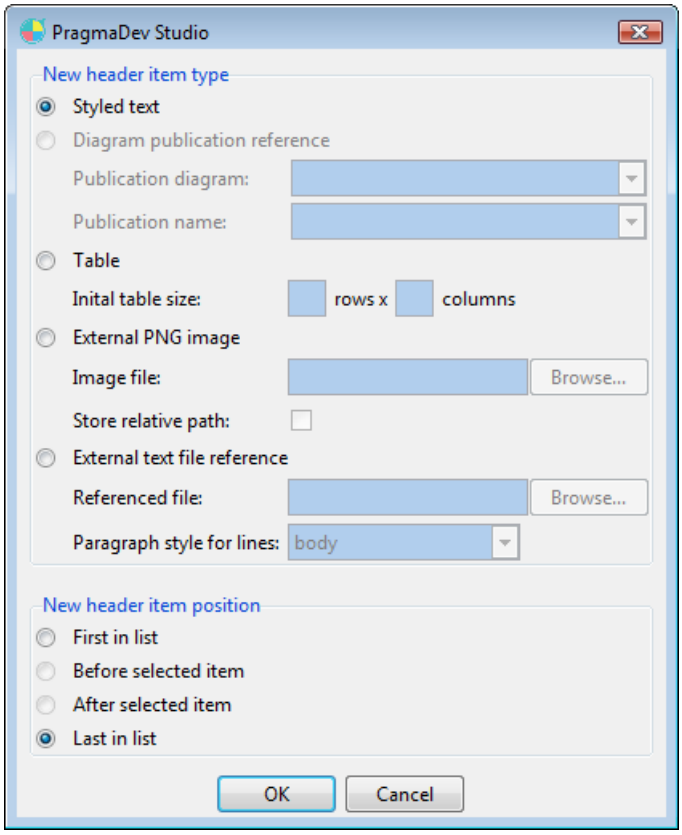


This sub-section will actually be the first chapter of the generated documentation. Adding other sub-sections at the same will generate other chapters. Adding sub-sections to

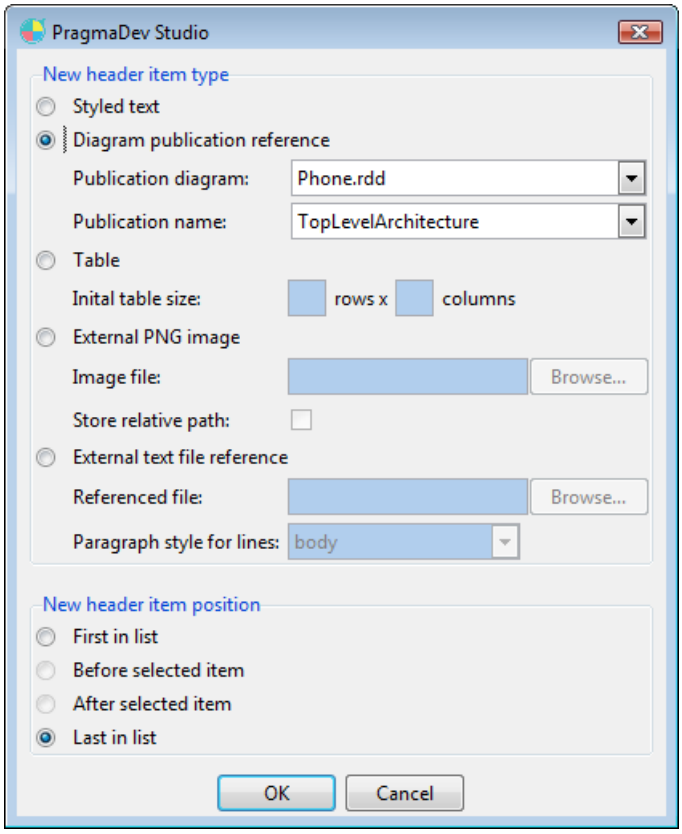
this section will generate sub-chapters. Let's type the section title and add some contents to the section with the + button:



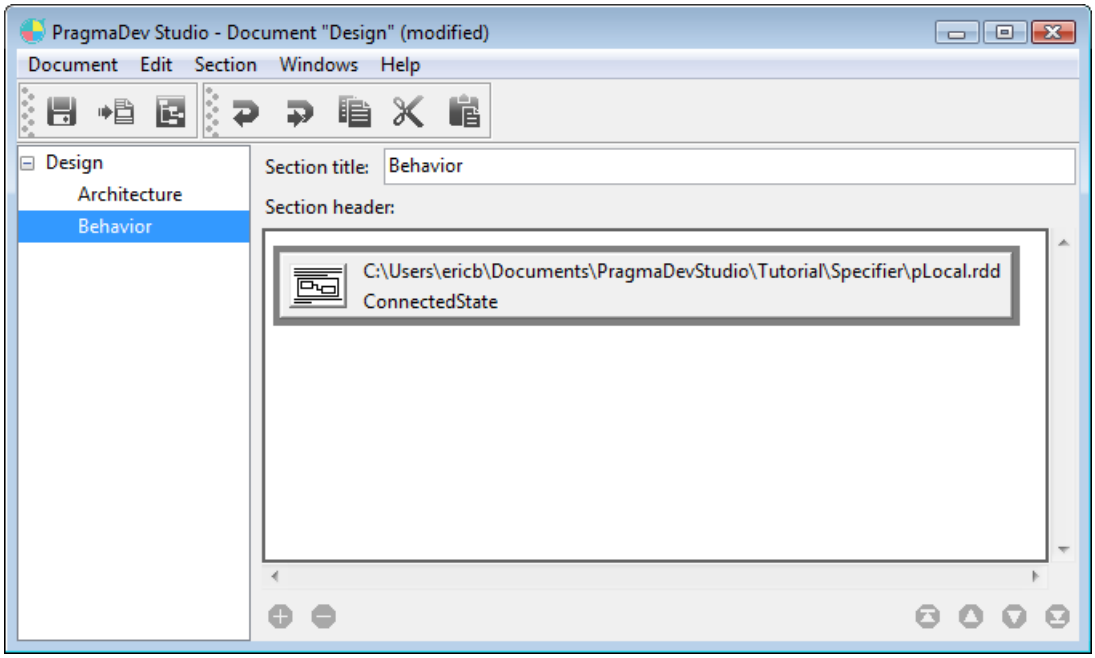
The following window will pop up:



The *Styled* text allows to insert plain text in the documentation. In order to have the *Diagram publication reference* available, some diagrams containing publications must be opened. So the top level architecture diagram and the `pLocal` process should be open in order to move on. The window should then look like this:

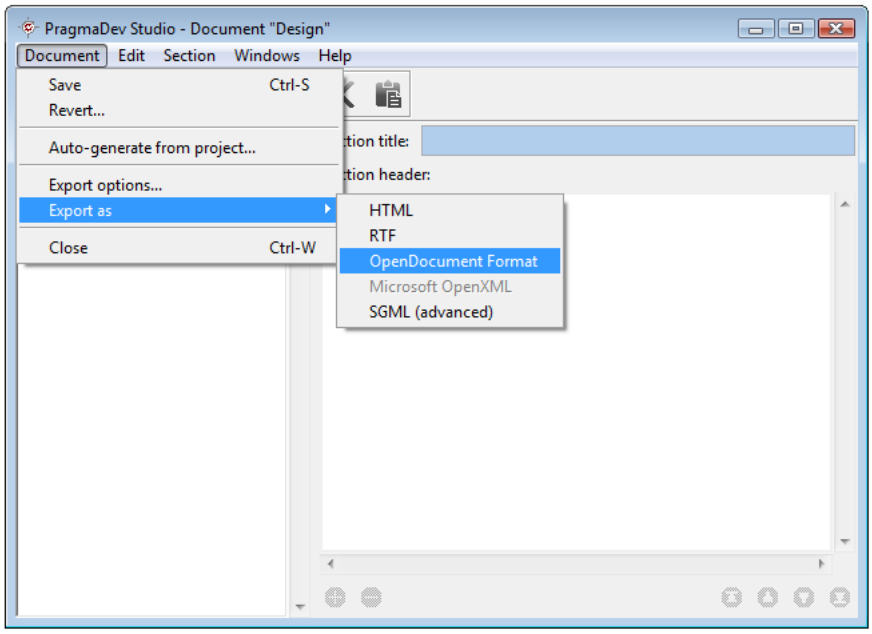


Select the appropriate *Publication diagram* and *Publication name* to insert in the *Document*. Then create another sub-section in the document with the other publication:



6.3 - Automatic generation

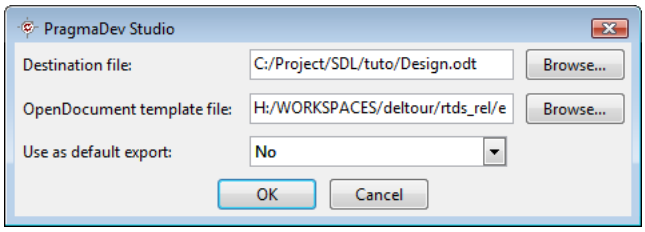
Go to *Document / Export as / Open Document Format*:



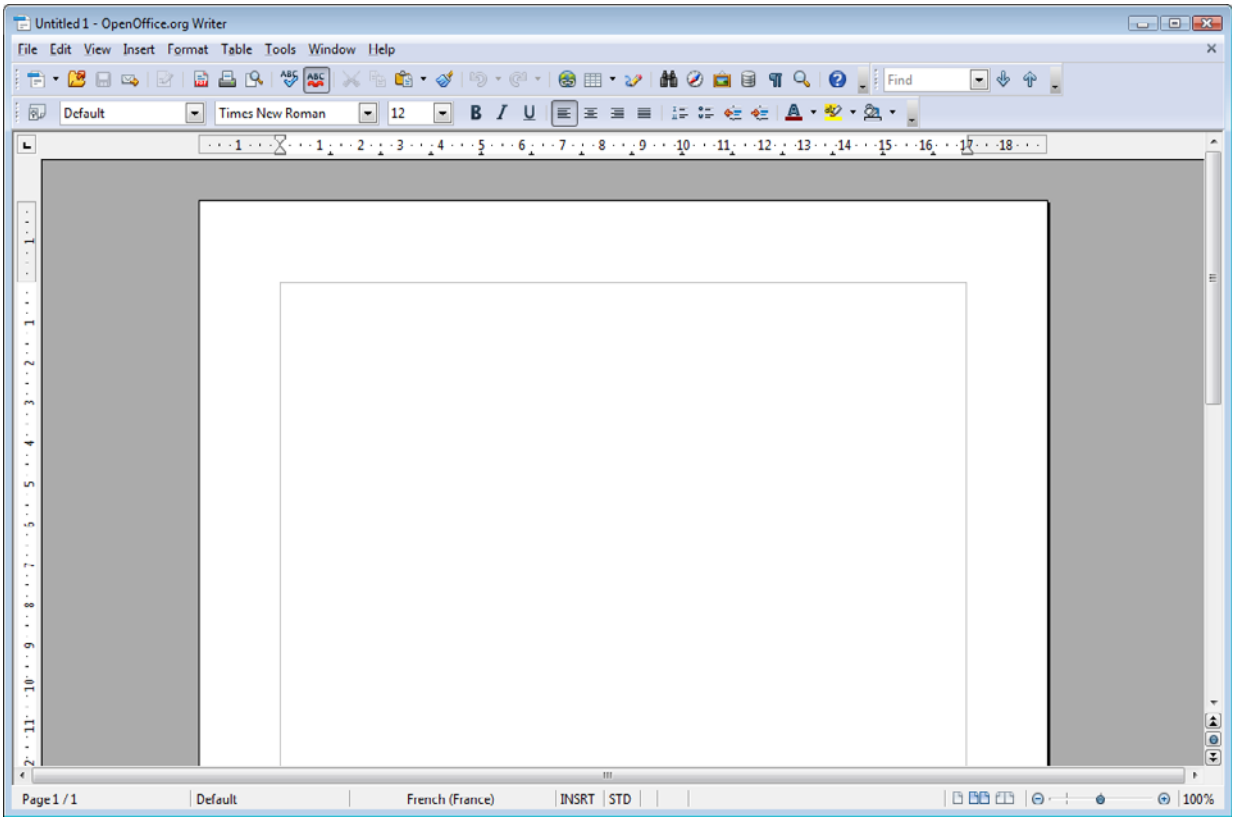
An OpenDocument is actually a zip file that contains several files among which:

- one is the document itself as an XML file,
- one describes the styles used in the document also as an XML file.

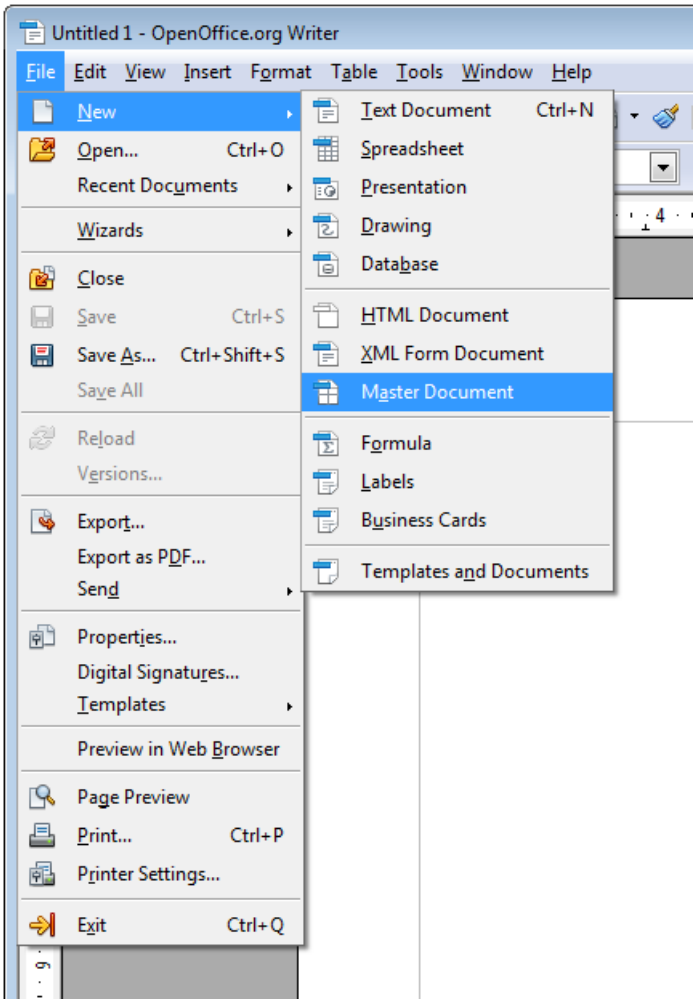
In order to generate the full OpenDocument zip file, PragmaDev Studio requires a template file. We suggest you use the one that is provided in the AccessControl example directory:



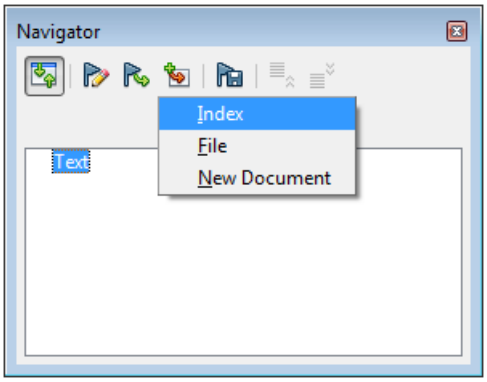
Now we will create an OpenDocument container for our generated document in which we can set a title, introduce a table of contents, and an index. To do so, let's start OpenOffice :

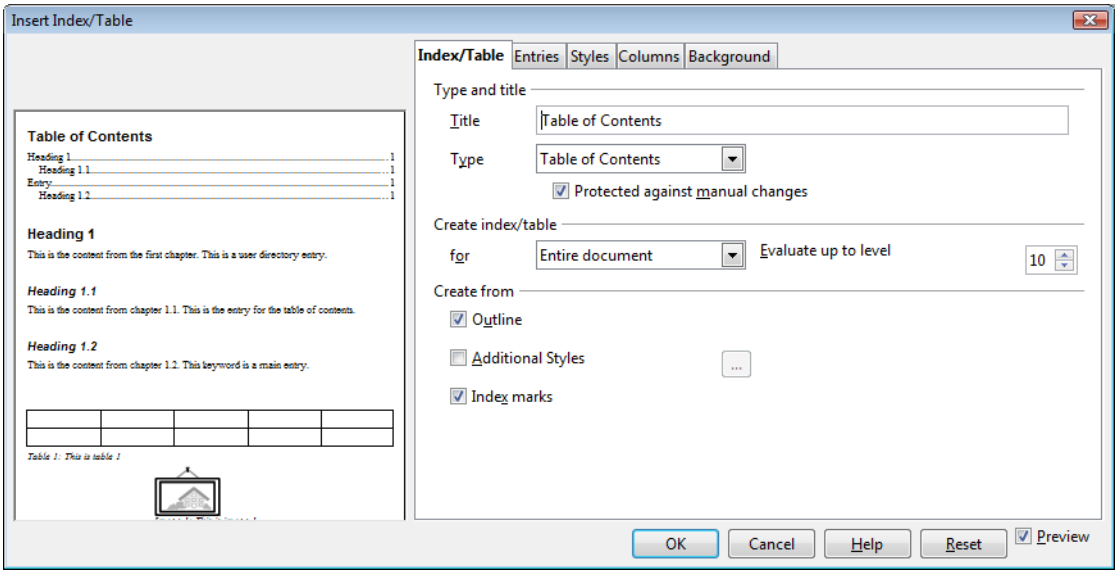


Create a new Master document with *File / New / Master document* menu:

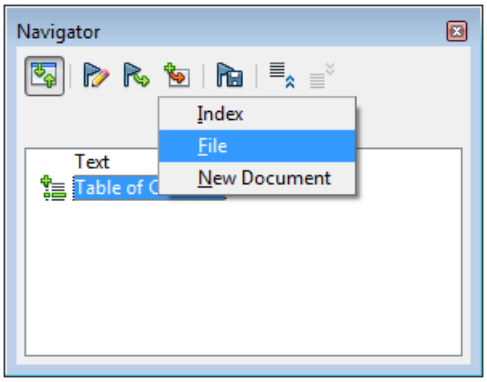


Type the title, insert a page break, and insert a table of content with the *Navigator* window:

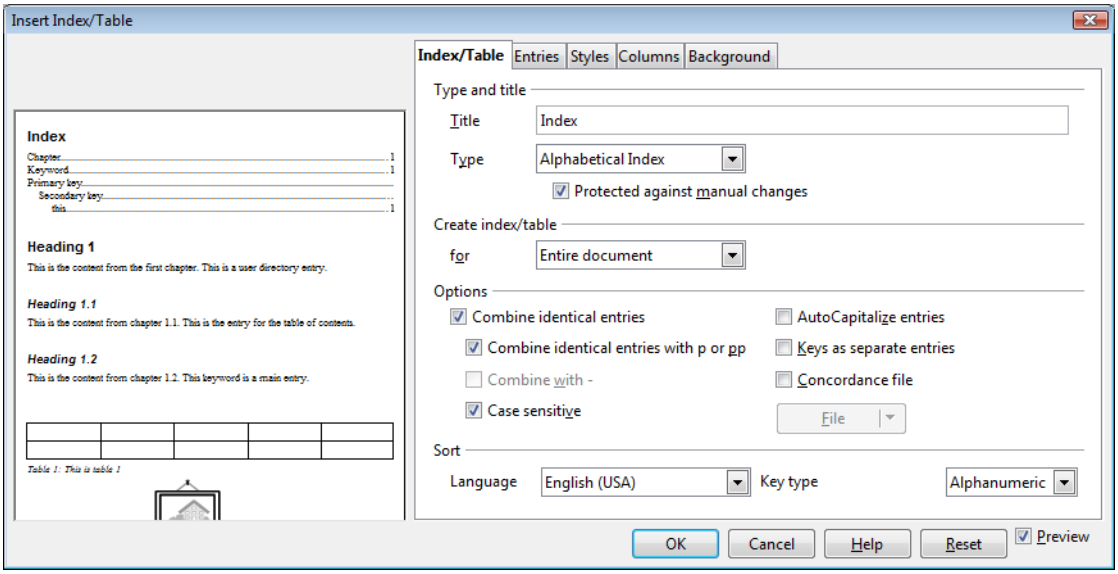




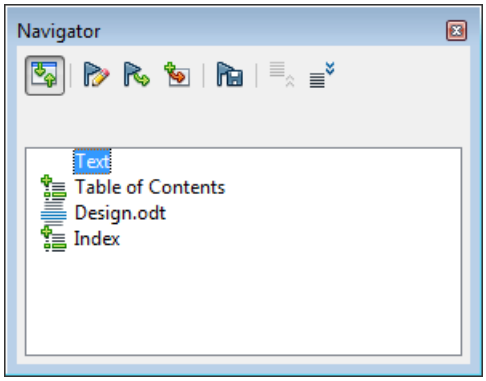
In the same Navigator window, insert the OpenDocument we have just generated:



And insert an index:

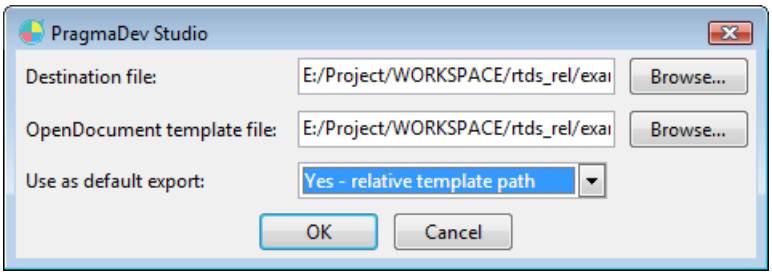


It is possible to drag and drop sections in the Navigator window to get the right order:

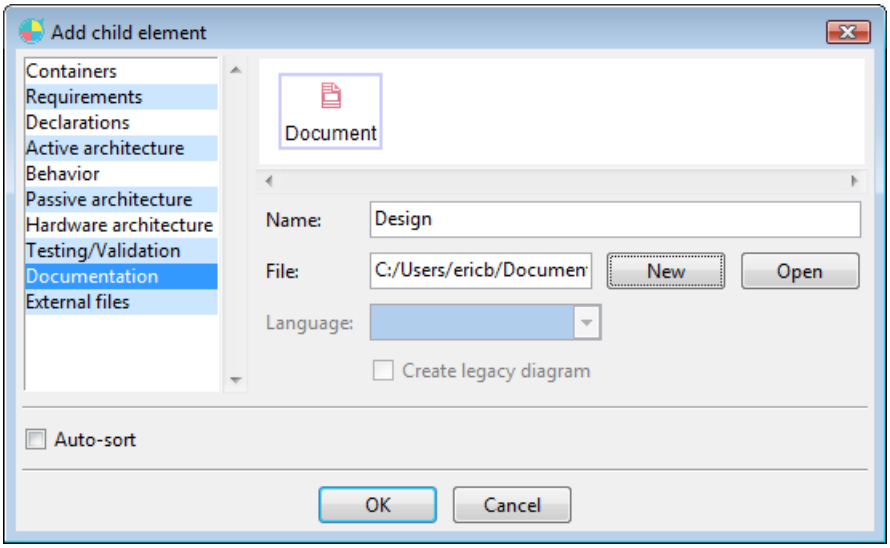


It is possible to add other sections or other external documents in the master document.

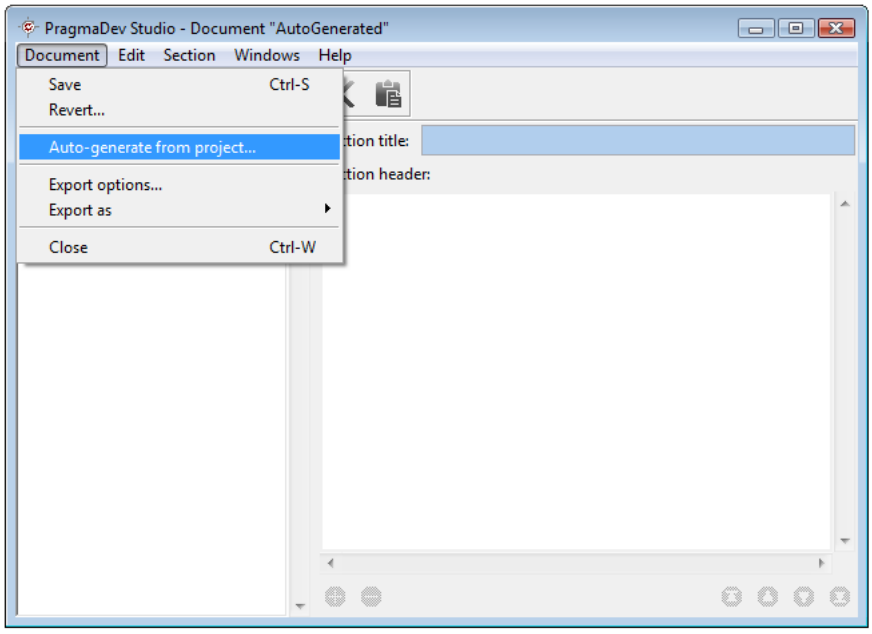
When you further document your system, to update the generated documentation: open your Pragmadev Studio document, export it as an OpenDocument, replacing the existing one, and that's it! The OpenOffice master document will be updated by itself as well as its table of contents and index and ready to be printed. You can also set the OpenDocument export as the default one and just press the button in the document editor:



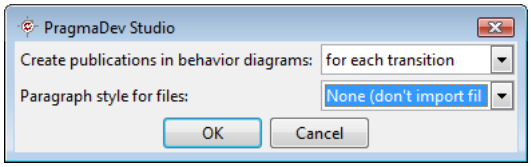
Please note it is also possible to automatically generate all publications and a document based on the project architecture. To do so, create a new document:



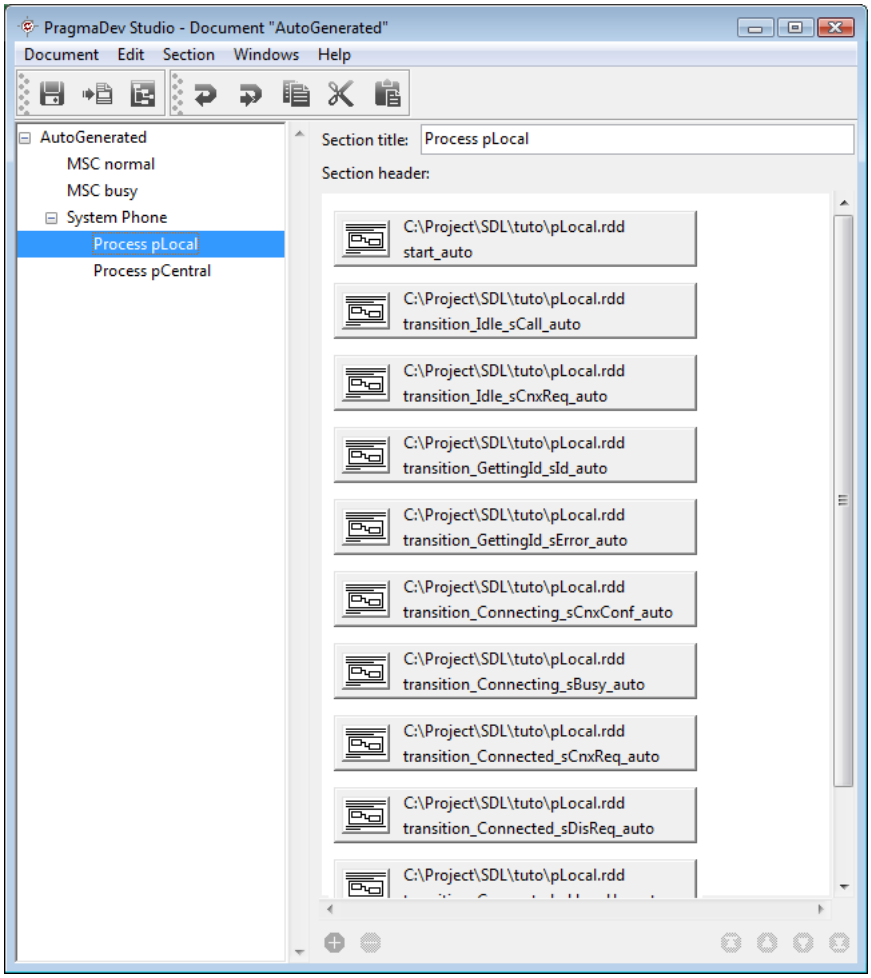
Open it and go to the Document / Auto-generate from project... menu:



Select the level for each publication:



And a full document is generated:



If you already have exisiting publications at the proper level for the auto-generated document, they will be reused. If a publication is missing, it will be created. You can add explanation texts before and/or after the created publications after creation.