ReaGeniX™

Programmer 2.0

A Real-Time Application Generator

User's Manual

14.4.2000

OBP Research Oy

Teknologiantie 10 D, 90570 OULU, Finland

Tel. +358 8 551 5540

http://personal.inet.fi/business/obp/index.htm

ReaGeniX Programmer User’s Manual

Our policy is continuous improvement. This can cause slight differences between appearances of actual symbols in the documentation and in the distributed software.

TABLE OF CONTENTS

DOCUMENT CONVENTIONS

This manual uses the following typographic conventions.

Example of convention	Description
`duration_max, boolean`	In syntax, words in bold indicate language specific keywords.
`duration_max, boolean`	In text, this font is used for language specific keywords and names referring to examples.
`duration_max, boolean`	In text, this font is used for language specific examples.
parameter file	In text, italic letters are used for defined terms, and occasionally for emphasis.
parameter_file	In syntax, italic letters indicate placeholders for information you supply.
[*name*]	In syntax, items inside square brackets are optional.
name **|** *name*	In syntax, a vertical bar indicates a mandatory choice between two or more items. You must choose one of the items unless all of the items also are enclosed in square brackets.
{*name*}	In syntax, items inside braces are repeated at least once.

1. INTRODUCTION

1.1 Background

Embedded software is an essential part of modern electronic products, such as mobile phones, lifts and domestic applications. Embedded software is often called real-time software because it has a continuous interaction with its environment and it often has strict response time requirements.

The behaviour of real-time programs differ from the sequential programs and therefore the design and testing of such software require special languages and tools to avoid reliability problems.

1.2 Purpose of manual

The purpose of this manual is to give detailed instructions how to create real-time applications by using ReaGeniX Modelling Language and Visual Programmer.

The *ReaGeniX Modelling Language* (RML) is aimed for modelling and implementation of real-time systems. The models may range from abstract prototypes to the implementation. The *ReaGeniX Visual Programmer* is a tool for compiling of such models to the executable prototypes or to the final production code.

The ReaGeniX Modelling Language is a visual language, which is based on communicating hierarchical *state machines*. Each state machine has a well defined *interface* describing a *component,* which can be freely instantiated in higher level components (i.e. context-free components).

The *ReaGeniX Visual Programmer* translates the RML components to C/C++ program modules. The computation and data types used in RML are written in C/C++ using a predefined macro library and they are used as such in the produced code.

The best performance of the code produced by the ReaGeniX Visual Programmer can be achieved by the use of *ReaGOS* real-time operating system. However, the produced code is independent of operating systems and therefore it can be used with the most real-time operating systems.

1.3 Scope of Application

The ReaGeniX Visual Programmer can be used to produce code for hard real-time and embedded computer systems. The application areas include, but are not limited to: control, measurement, sequence control, process simulation, and data communications.

ReaGeniX Visual Programmer can be used in quick prototyping, demonstration, validation of specifications and intermediate designs, and final implementation.

1.4 Components of ReaGeniX Visual Programmer

ReaGeniX Visual Programmer consists of following parts:

Visiopath`\Solutions\Reagenix\`**`reagenix.vss`** - a Visio stencil for ReaGeniX Modelling Language

Visiopath`\Solutions\Reagenix\`**`Rgx3w.vst or Rgx4h.vst`** - templates which use the `reagenix.vss`

Visiopath`\Solutions\Reagenix\`**`reagenix.exe`** - a Visio add-on, which reads the diagram information from Visio -tool using OLE2 interface into pagename.vis file. The ReaGenix add-on should reside in one of Visi add-on paths (see Installation Guide for editing *visiopath*`\System\visio.ini`), which calls `rgxbat.bat` after conversion.

Rgxpath`\Visio5\gener\`**`rgxmsg.exe`** - a Windows executable for viewing the ReaGeniX warnings and error messages (file `pagename.err`). Opens the diagram page in Visio and shows the object. The error message is referring by selecting it. Works best if to be assosiated in Windows with `.err` files.

Rgxpath`\Visio5\gener\`**`rgxbat.exe -`** a Windows executable file called by `reagenix.exe` add-on. Parameters for batch file are:

1. The directory where the Visio model to be generated is.
2. Page name of generated diagram (used for file names).

Rgxpath`\Visio5\gener\`**`vis2rgx.exe`** - a MSDOS executable converting visio format diagram information into ReaGeniX (`.rgx`) format (run by `rgxbat.exe`).

Rgxpath`\Visio5\gener\`**`rgx2c.exe`** - a MSDOS executable generating C - code from ReaGeniX (`.rgx`) format (run by `rgxbat.exe`).

Visiopath`\Solutions\R`*`eagenix`*`\`**`reagenix.hlp`** – a shape help for reagenix.vss.

2. GETTING STARTED

2.1 Modelling an Automatic Door Controller

The problem:

> You should design an automatic controller for a market door. You have a door machinery and a radar. The door machinery is controlled by a two state electrical relay (a flag for now on). If the flag is high, the servo drives the door open. If the flag is low, the servo drives the door closed. The radar output is also a flag. If the radar senses something within its range, the output flag is high, otherwise low.
>
> If the radar senses something, the door must be driven open. The door is allowed to close after 3 seconds has elapsed after the last indication of something within the range of the radar.

A state machine is the solution for a simple sequence control problem like this.

State machines are the basic building blocks of a ReaGeniX system description. A state machine reacts to *events* with computation, control, output, and updating internal and external data. An event may be considered something happening outside of the system, e.g. closing a door or elapsing of a predetermined waiting time. However, a state machine sees an event as arrival of data or a signal or firing of an internal timer and it can react to all of them. A *state-transition diagram* is a picture of a state machine.

The main parts of a state machine are states and transitions. The states are drawn as boxes and the transitions are drawn as arrows in a state transition diagram. In a state machine, one of the states is the *current state* at a particular point of time. The possible responses of the state machine are defined by the transitions leading from the current state.

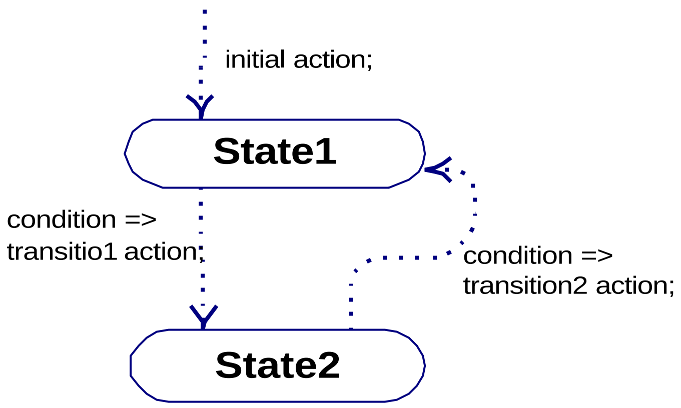


Figure 1: A state transition diagram.

A transition leading from the current state may be fired, if the *condition* of the transition holds. When a transition is fired, the state, which the transition leads to, becomes the new current state. Moreover, the *actions* of the transition are executed. The condition and action are defined textually and the condition is separated from action with an arrow (=>) .

It is convenient to visualise the operation of a state machine moving a paper clip on a state transition diagram. The paper clip is moved from the old current state to the new current state following the arrow of the fired transition.

An action is a piece of program written in C-language. It may perform any computation, output data, modify stores both internal and external to the state machine, and control other components.

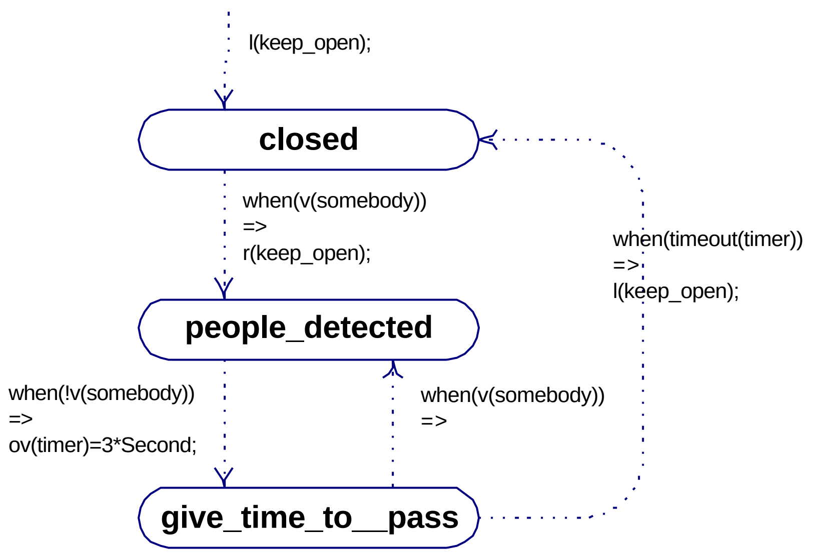


Figure 2: Automatic door state transition diagram .

In Figure 2, there is a state-transition diagram for automatic door control. Here we can see three states and four transitions. One of the transition arrows starts from nowhere. It is the *initial transition*. The initial transition defines what is done when the state machine is started.

A new state transition diagram is created with Visio tool using the stencil provided by ReaGeniX delivery. It is assumed here that the reader has the basic knowledge of using Visio Tool. Figure 3 shows the state transition diagram in Visio tool. The symbols in stencil can be seen on the left. The symbol is imported in the diagram by drag and drop.

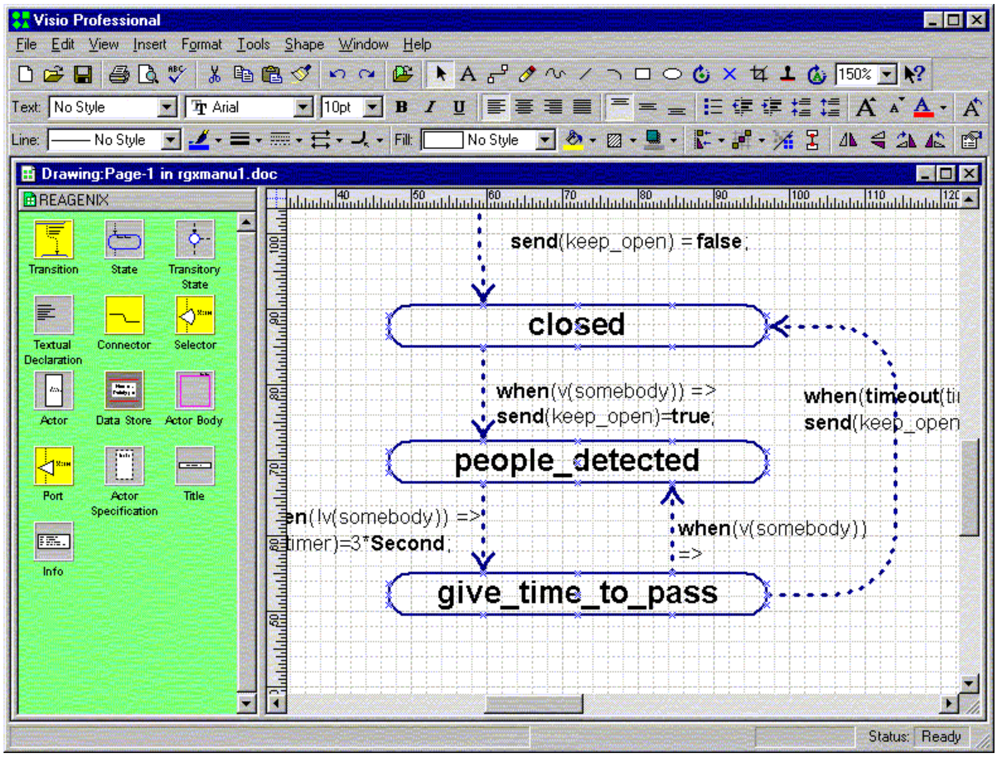
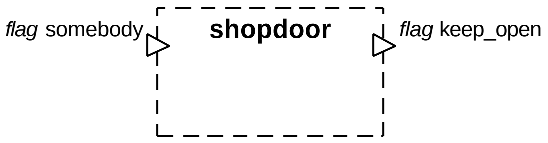


Figure 3: A state transition diagram in Visio.

A diagram in ReaGeniX for Visio is drawn in separate page. The name of page is important for code generation purposes and a page name of maximum 8 letters is recommended. The symbols of the ReaGeniX stencil usually have a right mouse button menu which contains the most common operations for the symbol. Usually one of these is editing the custom properties, which are asked every time a new symbol instance is created into diagram.

In addition to states and transitions, a complete state transition diagram includes a component body, a component specification, and local variable declarations. A component body is represented by a rectangle around the state machine. A component specification is a broken-line box. A local variable declaration represented by a textual declaration. An example of complete model is presented in Figure 4.

Every diagram must have exactly one component body symbol, which names the component type that the diagram describes (here `shopdoor`). A component specification defines all the connections between the new component type and its environment. A textual declaration is used to define the local variables whose value is to be retained between transitions. The timers are also declared using textual declaration.



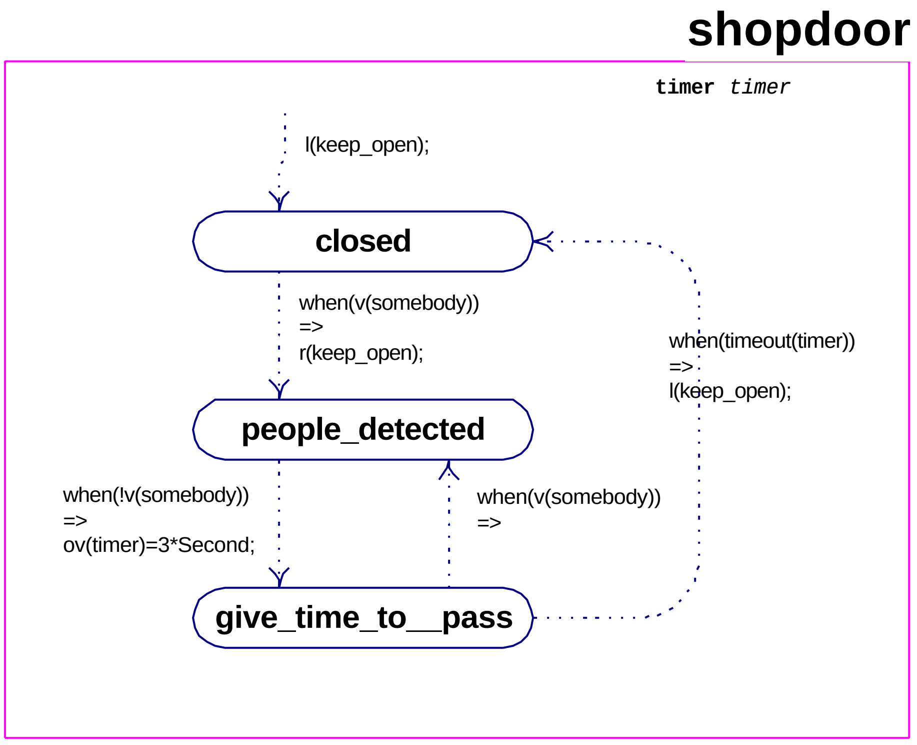


Figure 4: Automatic door control diagram for ReaGeniX.

In the shopdoor component specification there is one *input port* (a triangle pointing inwards), a flag called `somebody` which tells, whether somebody is seen near to the door. A flag may have Boolean values *true* and *false*. Here it is assumed that the value *false* means that nobody is seen and the value *true* means that somebody is seen.

To avoid conflicts with standard Windows header files ReaGeniX uses names `R_boolean`, `R_false`, and `R_true` for Boolean type and values.

There is also one *output port* (a triangle pointing outwards) a flag called `keep_open`. It is used to command the physical door either open or closed. Here it is assumed that the value `R_false` commands the door to close or keep closed and the value `R_true` commands the door to open or keep open.

One timer `trail_timer` is declared. A *timer* is a variable whose value is automatically decrement in pace of real-time.

When the door controller is *enabled* (= switched on), the initial transition is fired and its action, the statement `send(keep_open)=R_false;` is executed. The statement sends the value `R_false` to the output port to keep the door initially closed.

Therefore, the controller is first in the state `closed`. From this state, only one transition is possible. If somebody appears in the detection range, the condition `when(v(somebody))` causes the transition to fire. A person in the detection range is seen by the state machine so that flag `somebody` raises to `R_true`. The operator `v()` access the value of the port and the condition operator `when()` reacts to the value `R_true` of the argument. Consequently the action `send(keep_open)=R_true;` causes the door to open. Finally, the machine is settled in the state `people_detected`.

When the people has passed clear of the detection range, the expression `!v(somebody)` becomes `R_true` and the next transition is fired. `ov(trail_timer)=3*Second;` sets timer to fire in 3 seconds. The physical door is not affected at this moment.

The machine can wait maximum of 3 seconds in the state `give_time_to_pass`. If somebody appears in the detection range during the wait the machine returns to `people_detected`, otherwise the timer fires, `timeout(trail_timer)` becomes `R_true`, the door is commanded to close, and the machine returns to the state `closed`.

The diagram of the automatic door controller is found from the distribution media as `exampl01\shopdoor.vsd`.

2.2 Compilation

To store the generated code make a working directory, called *workpath*.

Check that ReaGeniX Visual Programmer is in the **Tools-Run Add-on** menu. If it is not you must modify the file `Visio.ini`

`AddonsPath=` *rgxpath*`\Visio\add-on`

To generate code from Visio Diagram select command from menu **Tools-Run Add on** -**ReaGeniX** (see Visio documentation for details of running add-ons).

The ReaGeniX Visual Programmer produced files **`.C`** and **`.H`** for each diagram:

`.C` file contains the program code.

`.H` file contains the data declarations and function prototypes.

The base of filenames is defined by name of the Visio page of the diagram.

ReaGeniX uses macro library `REACTIME.H` and a parameter file `REACTIME.P`, both of which are found in *rgxpath*`\INCLUDE`. These files must be included in the main program. No special run-time libraries are needed.

A simple test environment is provided with the ReaGeniX Visual Programmer to execute the generated program. The instructions ara provided in Appendix 1.

3. REAGENIX COMPONENT

3.1 Actor component specification

A component specification describes the interface of a new component type. The interface consists of communication channels defined by named ports. In ReaGeniX language, all components are actors so the name actor and component are used without distinction. An actor specification symbol is a rectangle into which the port symbols are snapped to.

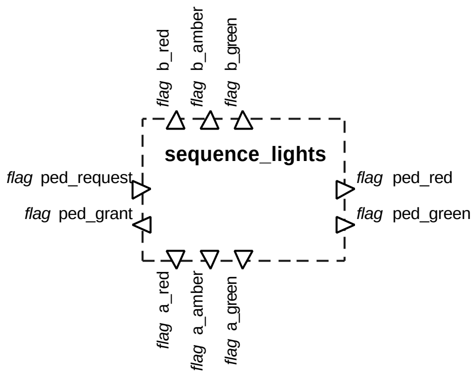


Figure 5: An actor component specification.

3.2 Ports

A port defines the direction, name, class and datatype of a communication channel in actor component interface. A communication consists normally of associated data value and event.

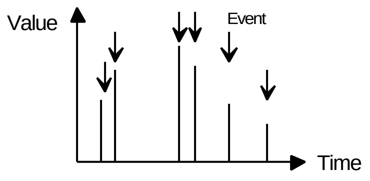
An actor component can respond to an event in a communication channel in its interface. A data associated with communication can be used as a basis of calculations inside component. A component can read the data associated with communication channel when the component is activated by an event, but a change in data without event cannot trigger a response. The event that activated the component need not always be in the same communication channel from which the data is read.

Depending on the event time behaviour of data in communication channel the basic communication mechanisms are divided into basic classes (or connector types):

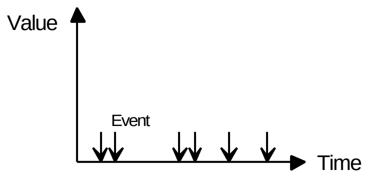
- Time Discrete communication
- Time Continuous communication
- Store
- Control

3.2.1 Time Discrete Communication

A discrete communication passes messages from the sender to the receiver. The data is passed, if the receiver component is waiting for the data at the moment of transmission, otherwise, the data is lost. The message and the accompanying event are not buffered in any way. The sender is not affected by the successful reception or the loss of data. The datatype of data value is specified when declaring a port of type discrete communication.

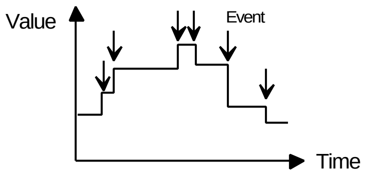


A special case of discrete communication is a *signal*, it has no data value associated to the event.

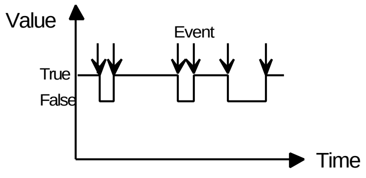


3.2.2 Time Continuous Communication

A continuos communication has a value at any time. It retains its value until a new value is sent. The receiver can read its value whenever it needs to. An event is associated to the change of the value so, that the receiver can immediately react to the new value.

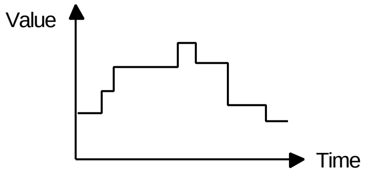


A special type of continuous data is a flag whose datatype is Boolean and need not to be specified in declaration.



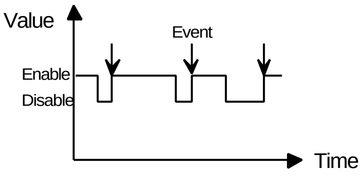
3.2.3 Shared Data Store

A shared data store can be accessed from several components. It retains its value until changed. No event is associated to the change of the value. It is impossible to immediately react to the change of the value, without other kinds of signalling.



3.2.4 Control (Enable)

Controls are used to switch a component on and off. Controls cannot be branched or merged.



Switching a component off means immediate killing of the component with complete loss of data values and state information within the killed component. However, data in continuous communications and stores **outside** the killed component retain their values.

Switching a component on means full reset of the component. The data in stores and continuous communications inside the component are set to initial values and initial transitions of the state machines are executed.

3.3 Port symbol

A declaration of port consist of:

[*direction*] *connector_type portname*[*index_name*][**:** *datatype*]

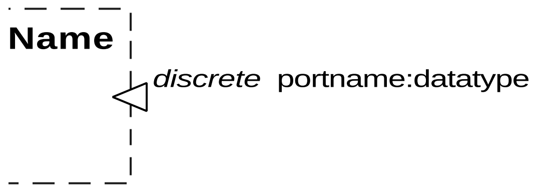


Figure 6: A port.

A port symbol is a small triangle snapped to component specification symbol (Figure 6)

direction of port is shown visually by the direction of triangle. The direction of port can be changed from **right button** menu and from **custom properties**:

Triangle head to the component	for input port
Triangle head from the component	for input port

connector_type is selected from symbol specific **custom property** window (right button menu):

`discrete`	for time discrete communication
`continuous`	for time continuous communication
`store`	for stores
`signal`	shorthand for discrete communication with no data value
`flag`	shorthand for continuous Boolean valued communication
`control`	for component enable/disable

port name identifies the name of the port that must be unique (local) for this component. *datatype* is used for discrete and continuous connections. *datatype* is either a C standard datatype or it is declared in an included C header file (with `typedef`). *port name* and *datatype* are written in **text property** of port symbol.

For C -datatypes used in port declarations a diagram must have a *textual declaration* symbol (see ch. 0) where the name of header file where to find the datatype is told, e.g.:

```
include ctypes.h
```

A port can be *indexed*. A indexed port consists of ports with the same connector type and datatype who are separated by port index. An indexed port has a separate event and data value for each index. Note that a port with C -array type has only one event. *index name* must be declared with *textual declaration* symbol (see ch. 0) before it can be used in port declarations in a diagram, e.g.:

```
index I :< ntrans
```

defines index `I` so that its range is `0..ntrans-1`. The syntax of an index definition clause is

index_variable **`:<`** *repeat_count*

where *index_variable* is a legal C variable name and *repeat_count* is a constant expression that can be evaluated during C compilation. The *repeat_count* may contain symbols defined in an included header file. The definition makes the lower bound of the *index_variable* to be 0 and the upper bound to be (*repeat_count* -1) inclusive.

3.4 Component body

An actor component body symbol is used to describe the implementation of a component type. The actor body symbol is used to draw a border around a state transition or an architecture diagram in ReaGeniX language.

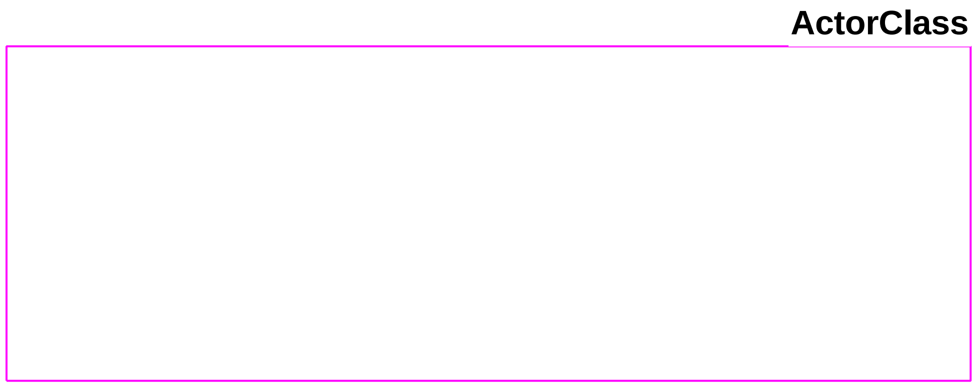


Figure 7: An actor body symbol.

In ReaGeniX diagrams, the component type symbol is used to represent the interface of component for connections to the outside world. The ReaGeniX state transition diagrams and architecture diagrams are drawn inside the component body symbol. The name of component type is written in text custom property of actor body symbol. Usually ReaGeniX diagram has an actor body and actor specification, which have a same name.

4. APPLICATION DESCRIPTIONS BY REAGENIX

The application descriptions made by ReaGeniX consist of two major diagram types, architecture diagrams (AD) and state-transition diagrams (STD). A ReaGeniX diagram always defines a new *component type*. It is thus possible to form a *library* of *re-usable components*.

State-transition diagrams describe the dynamics and the computation of the system. A state transition diagram describes an elementary sequential process called a *state machine*.

Architecture diagrams describe the structure of a system. An architecture diagram describes a higher level process composed of *concurrent* lower level processes called *actors* some of them might be described as state machines. An actor symbol in architecture diagram defines a *component instance*. There may be several component instances of the same component type in a system

The hierarchy of architecture diagrams may be arbitrarily deep. Actually, the depth may be limited by the C-compiler used to compile the results of the ReaGeniX Programmer.

The processes are connected together by communication mechanisms. The communication mechanisms available in the ReaGeniX language are

- shared data stores
- discrete communication including signals
- continuous communication including flags
- control (enabling/disabling a component)

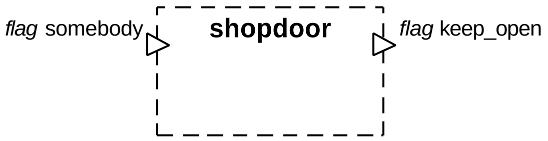
Discrete and continuous communication and control carry an *event*. It means that a state machine can immediately react to

- change of the value of a continuous input
- new data (even if the same value) in a discrete input
- enabling via an control

Moreover, a ReaGeniX component can react to a timeout using a timer concept. A change of the value of a store cannot be immediately reacted to. The value of a store can be tested in a reaction to some event e.g. to a timeout of a timer. However, high frequency polling of a store results computationally heavy implementation and should be used with discretion.

5. STATE TRANSITION DIAGRAM

A state transition diagram defines a state machine. State machine reacts to *events* with computation, control, output, and updating internal and external data. An event may be considered something happening outside of the system, e.g. closing a door or elapsing of a predetermined waiting time. However, a state machine sees an event as arrival of data, a signal, or firing of an internal timer, and it can react to them all.



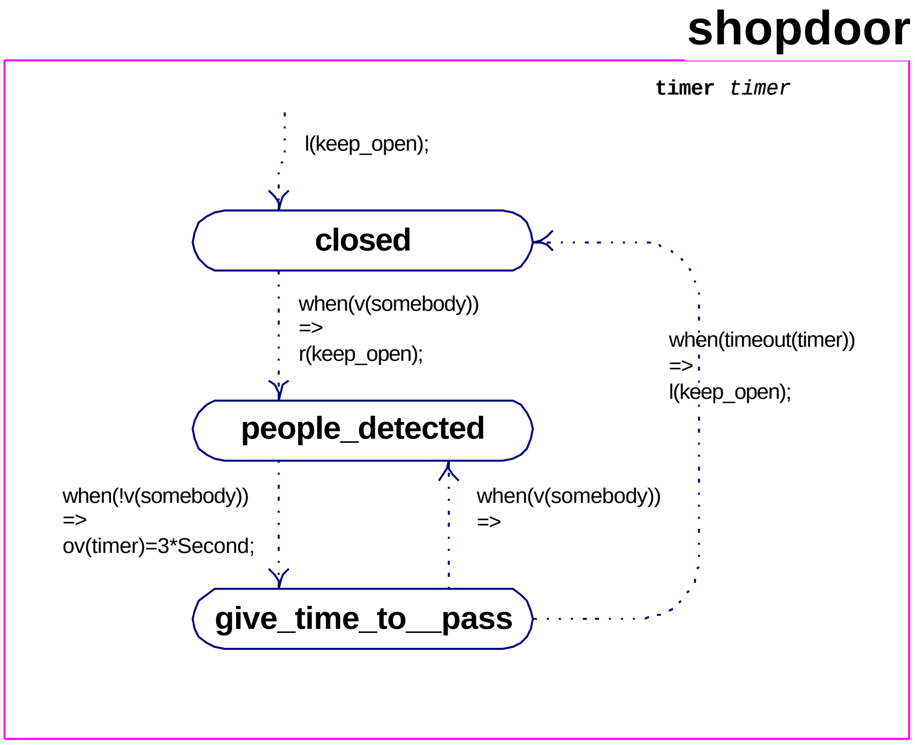


Figure 8: A State Transition Diagram.

A state transition diagram consists of states, transitions, timers and local stores.

5.1 Local Declarations

A textual declaration of form:

`store` *storename*`:`*datatype*[`:=`*initializer*]`;`

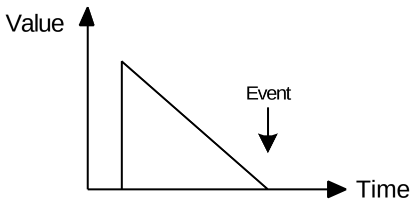
declares a local state variable in state transition diagram. Each time the state machine is reset, the initializer value is assigned to the variable. The initializer can be omitted, if the type has a default initializer, or the type is explicitly uninitialized. See section *initial values*.

A textual declaration of form:

`timer` *timername*[`:=`*initializer*]`;`

declares a timer. A timer is initialised like a store. The default initializer is 0.

A timer is almost like a store of type duration whose value decreases with passing of time. A timer produces an event when its value reaches zero.



5.2 State

A state is used to define waiting. In a state, a state machine can wait for an external event or a condition to hold. The different states of a state machine represent different ways to react to the same kind of events. States are drawn as boxes in a state-transition diagram. The states are given descriptive C names.

If a state machine is enabled, one of the states is called *current state*.

5.2.1 Transition node

Transitory states are used if a transition should branch to several states based on Boolean conditions, or several transitions should join, or a combination of both is needed.

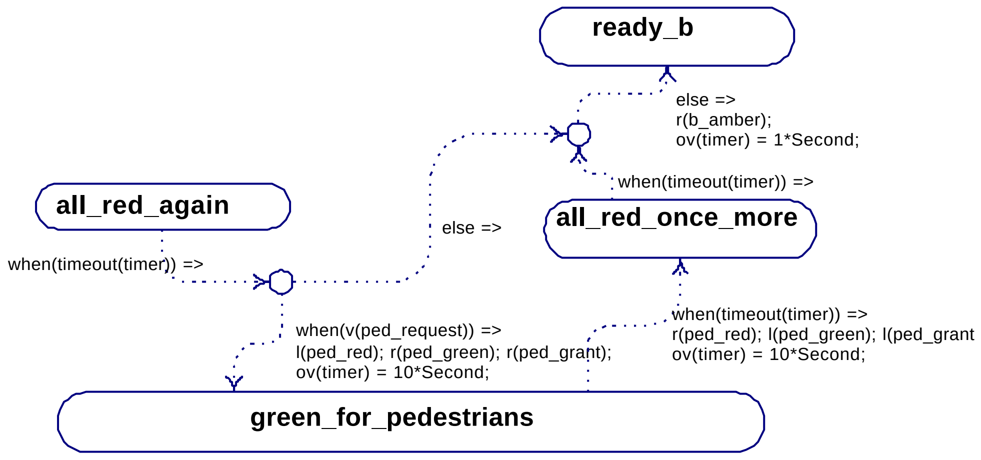


Figure 9: Example of transitory states.

No waiting happens in a transitory state. One of the transitions leaving a transitory state is the *default transition*. The default transition is executed, if no other transition can be immediately executed. A transitory state has no name.

The conditions leaving from a transitory state are evaluated in arbitrary order (not in random order). The first transition with a satisfied condition is executed. If none holds, the "else" branch is executed.

5.3 Transition

The transitions describe the responses of a state machine. A transition leading from the current state may be fired, if the *condition* of the transition holds. When a transition is fired the *action* of the transition is executed and the state, which the transition is leading to, becomes the new current state.

The transitions between states are drawn as arrows annotated with text. The syntax of text is:

condition **=>** *action*

If the ‘=>‘ is missing the whole text is assumed describe an action.

NOTE: Because C -macros are used in transition condition and actions for accessing the ports and local data the ReaGeniX Programmer cannot currently check if the operation is valid for the object. An operation using value field of signal type port results in C compiler error: *field .pv is not defined*. Similarly, an operation for event field for store results in C compiler error *field .ev is not defined*.

5.3.1 Initial transition

A state machine must have at least one initial transition. An initial transition defines the first operation and the first current state of a state machine. The initial transition has no condition, if it is the only one. If the start operation should depend on some condition, one of the initial transitions must have an **`else`** condition and the others have **`when`** conditions. The evaluation order is the same as in a transitory state.

5.3.2 Transition condition

A condition defines when a transition leaving the current state is fired. C programming language with macro extensions is used in transition conditions. It is possible to call external functions from an action if the header file for C - module is included into the diagram.

`on(`*port_name***`)`**

An **`on`**-condition is used to receive a discrete communication or a signal. The value of the received discrete communication is available in this transition only.

`when(`*boolean_expression***`)`**

A **`when`**-condition is used to wait some condition to hold. C syntax is used for the expression. The expression can test values of arriving continuous flows (e.g. flags) and time-outs of timers, **only**. When testing continuous or discrete flow a v(portname) must be used to access the value field in expression

`when(v(`*portname***`))`**

See section *25 Transition* for how to refer to values of the ports and local state variables in expressions.

`on(`*portname***`)`** {**`or_on(`***portname***`)`**}

The above construct is used for waiting any of several signals. No discrimination between signals is possible.

`on(`*portname***`) and_when(`***boolean_expression***`)`**

The above construct allows extra conditions for arrival of data. There are no limitations for the expression. The expression **can** test the value of the arriving data.

5.3.2.1 Transition condition from a transitory state

`when(`*boolean_expression***`)`**

A when condition allows branching to this transition, if the expression is true. C syntax is used for the expression. There are no limitations for the expression.

`else`

`else` is used to mark exactly one (default) transition from a transitory state. This branch is taken, if none of the conditions hold.

A condition is not needed if there is only one transition leaving a transitory state.

5.3.3 Transition action

An action defines the computation and output of the transition. C programming language with macro extensions is used in actions. It is possible to call external functions from an action if the header file for C -module is included into the diagram.

An action is not allowed to wait anything. Actions with some kind of waiting: - OS-calls for delays, receiving messages, reading data from devices, waiting some condition to change in a tight loop - are erroneous. Such actions cause the whole system to occasionally halt permanently or temporarily. All waiting must be organised to states. OS interactions should be organised to task main routines calling the ReaGeniX generated modules.

Following C -macros are used to access the **values** of ports, and local stores and timers.

Table 1. Available access macros and their definitions

Available access macros	**Definition**
`v(`*portname***`)`**	Access to the value field of a port
`v(`*portname***`)[`***index***`]`**	Access to an element of an array type value field
v(*portname***).***member_name*	Access to a member of a structure type value field
`v(`*portname***`).`***method_name***(...)**	Access to a method of a C++ -object type value field
`&v(`*portname***`);`**	Access to a pointer to the value field
`ov(`*store***|***timer name***`);`**	Access to a value field of local data (store or timer) Accessing elements and members of local data can be done with ov() as in v().
`timeout(`*timername***`);`**	Access to a timer. It can be used also in actions it has the value `R_true`, if the time assigned to the timer has been elapsed, otherwise `R_false`.

Table 2. Available output macros and their definitions

Available output macros	Definition
`send(`*portname***`)=`**expression**`;`**	Sending the value of the expression to a port.
`emit(`*portname***`);`**	If the port value field is already assigned with v(portname) .
`s(`*portname***`);`**	Send a signal.
`r(`*portname***`);`**	Raise a flag. It is equivalent to send(portname)=R_true;
`l(`*portname***`);`**	Lower a flag. It is equivalent to send(portname)=R_false;.
`e(`*portname***`);`**	Reset and enable a component controlled via a port.
`d(`*portname***`);`**	Disable a component controlled via port.

5.4 Indexing in state transition diagrams

5.4.1 Operations on indexed objects

Index expressions, which refer to elements indexed ports, timers, or stores, are **inside** of the macro parenthesis. Normal C -indexes referring to elements of data arrays are outside of the macro parenthesis.

Examples:

```
send(p[5][k_pp])=ov(x[k_pp]);
```

There is a special loop construct to make processing of the whole range of an index variable easier and safer in transition action:

```
for_each(index_variable)
    loop body
end_for_each(index_variable)
```

The *loop body* is repeated with each value of the *index variable*.

5.4.2 Indexed transition

A state machine can receive from an indexed port using an indexed transition. During the execution of transition action an index variable used in a transition condition has the value of index which made the transition condition to trigger. The value is valid until the end of the transition.

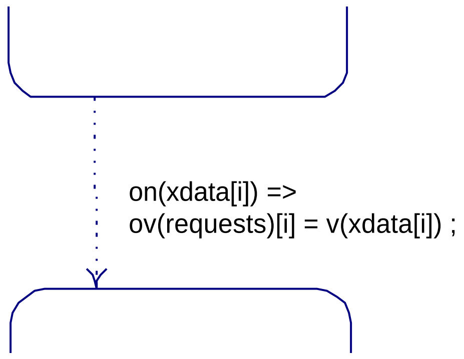


Figure 10: An indexed transition.

It also is possible to test indexed timers and indexed continuous connections.

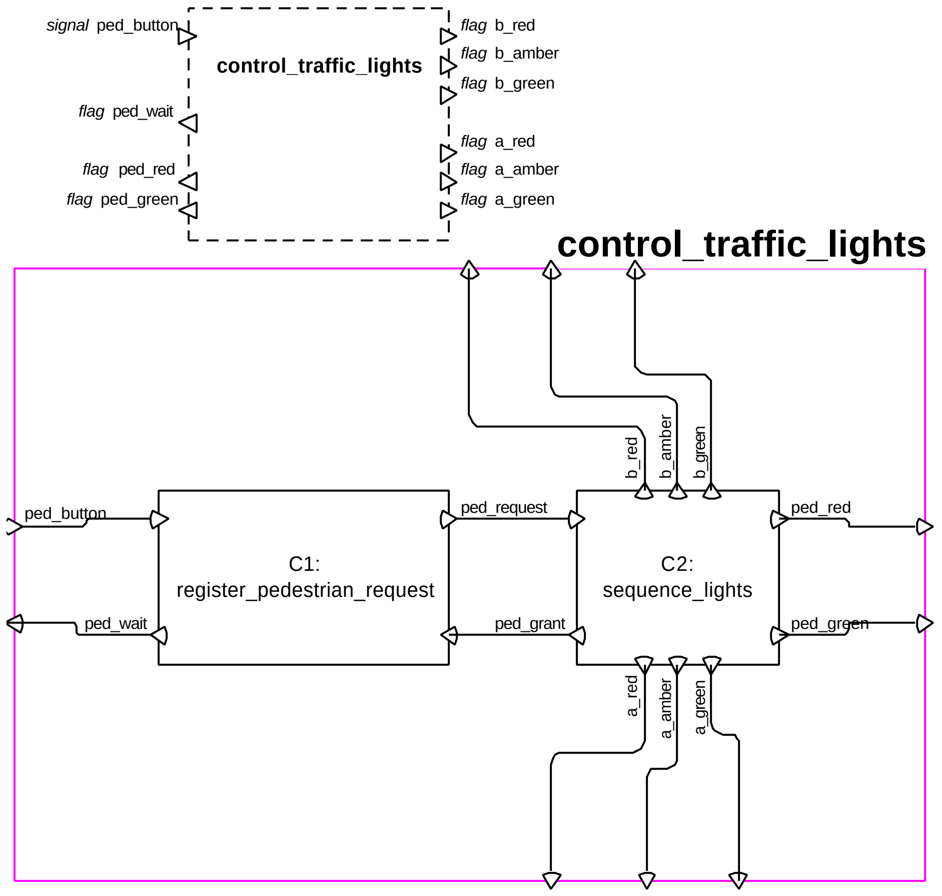
Examples:

```
    when(timeout(delay[path_index]))
    when((!v(drive_enabled[i]))&&v(drive_out[i]))
```

An indexed transition is enabled, if there is an integer value which is within the range and which makes the transition condition to hold. When the transition is fired the index variable is assigned a value, which makes the condition to hold. If there are several possible index values, the choice is not determined by the ReaGeniX language.

6. ARCHITECTURE DIAGRAM

An architecture diagram combines concurrent subsystems (*subcomponents* for now on) to larger and more capable systems.



```
include xtlight2.h
include xtlight3.h
```

Figure 11: An Architecture Diagram.

Because a component type of a subcomponent can be described, also with an architecture diagram a ReaGeniX model is a hierarchical description of the system. The hierarchy of architecture diagrams may be arbitrarily deep (Figure 12). Actually, the depth may be limited by the C-compiler used to compile the results of the ReaGeniX Visual Programmer.

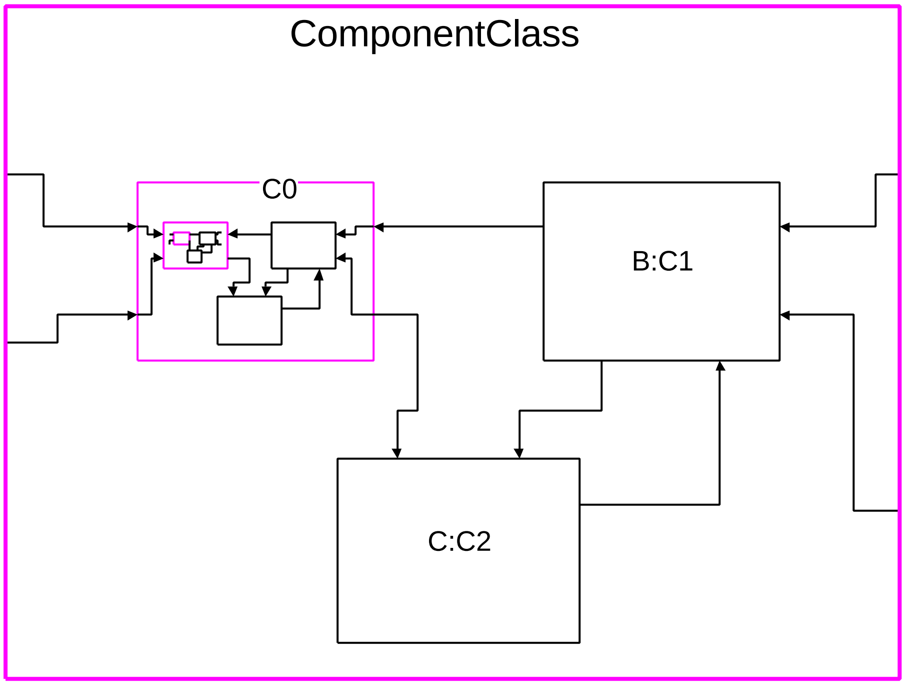


Figure 12: Component hierarchy in ReaGeniX models.

6.1 Subcomponents

The label of a subcomponent determines the names of the *subcomponent instance* and the *component type*. The name of the subcomponent instance is used to refer to the specific component instance. The name of the component type refers to the declaration of the subcomponent.

The name of a component instance is used e.g. by the other components enabling or disabling the component or by a testing environment to identify the location of interest.

The name of the component type is the name of the actor class, the data type, or the object class.

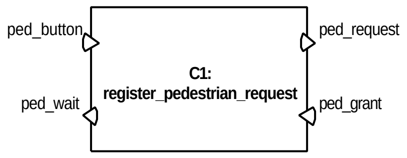


Figure 13: An actor component instance with selectors.

The names of the subcomponent instance and the subcomponent type can be different especially in following cases:

- Library components are applied.
- There are several instances of the same component type.
- There is a conceptual difference between the inside mechanics and the application of the component.

The syntax used, if the names of the instance and type are different is:

instance_name`:`*component_type*

Examples:

```
keep_fuel_level:PID_control
remove_ripples:f728
monitor_pre_heater_current:monitor_current
```

If the names of the subcomponent instance and the subcomponent type are the same, the simple syntax is used:

component_name

NOTE: In ReaGeniX for Visio, you must specify the header files of component types used in the architecture diagram. This is done using the include textual declaration. Forgetting this will produce a C compiler error of type: *R__componenttypename is undefined*.

6.1.1 Selector

Selector symbol resembles the port symbol. A selector symbol is snapped into an actor, store or component body (see Figure 13). A selector can be changed into port and vice versa.

The syntax of a selector text property is:

componentname [*cindexexpr*] *portname*[*pindexexpr*]`:`*datatype*

[`:=`*initializer*]

- Selectors in a connection may have different names. This is needed often when a subcomponent has different type and instance names.
- If selector has no text, name is sought from the object at other end of connector.
- If datatype is missing, datatype is sought from the other end of connector.
- An initializer is needed for a continuous connection. The initializer can currently only be defined in selector symbol. **This will be changed in later versions.** The value of a continuous connection is reset to the initial value when the component containing the connection is enabled or re-enabled. See section *Initial Values.*
- Only one initializer is allowed for one connection (selector-connector-selector).
- No initializers are allowed for input/output selectors (connected to component body symbol).

The text of port or selector of type control is simply:

portname

6.2 Connections

A connection between two selectors is made with connector symbol. Control ports may not be merged or branched with selectors. A control connection can be connected to a component instance without a selector. This means that the control is used for controlling (enable or disable) the component.

NOTE: A continuous connection between two subcomponents needs an initializer in either selector (this is not consistent and will be changed later). The value of a continuous connection is reset to the initial value when the component containing the connection is enabled or re-enabled.

6.3 Stores

6.3.1 Store declaration

Store in component is declared with store symbol. Textual declaration could also be used as in state transition diagrams, but it could not be accessed by any other component. The text property of a store has the following syntax:

name`:`*type*[`:=`*initializer*]

Initialzer can be omitted, only if there is a default initializer associated to the type, or the type is explicitly uninitialized.

The value of a store is reset to the initial value when the component containing the store is enabled or re-enabled.

6.3.2 Accessing a store

A store is usually connected without selector. The connector_type of store is store and direction is out.

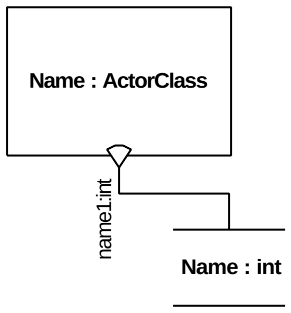


Figure 14: Accessing a store

If an element of an array or a member of a structure is to be accessed by a subcomponent, a selector connected to store is used. A selector expression is an index expression or a name of a member of a structure or a combination of several of both kinds. The member name should be preceded with a ".". The selector may be followed by the type.

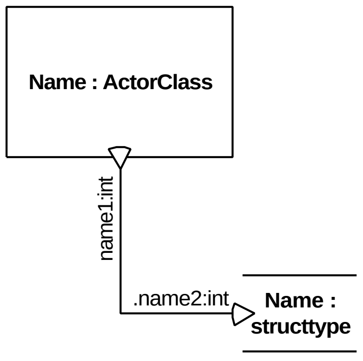


Figure 15: Accessing a part of a store

Examples:

```
[i]
.position
[iservo].position.variance:float
.corrections[x][y].bias
```

6.3.3 Scope of names

A name of actor or store has a diagram wide scope. A name at a selector connected to component body symbol has a diagram wide scope.

A name in a subcomponent selector has a scope limited to the interface of the subcomponent instance. There might be selectors of the same name connected to different component instances but having nothing else common but the name.

Examples:

```
fuel_level:float:=EMPTY_LEVEL
front_positioning:positioning_command
```

6.4 Indexing in architecture diagrams

Indexing a component is a shorthand notation to specify a multitude of similar objects. An indexed item behaves like several independent items.

Special *index variables* are needed for indexing. Any component except a store can be made indexed. Indexing is a **powerful tool** to create complicated connections between components.

The primary reasons to use indexing are

- to avoid drawing many similar objects
- to get clearer picture
- to make the design easier to reconfigure and re-use
- copy-paste + renaming increases the size of specification and the code and is a source of possible errors

Without indexing it would be very impractical to specify tens of parallel independent processes with same behavioural description.

Indexed ports are useful in following situations

- Requests and responses to and from a server component. This makes it possible for the server to use port indexes as client numbers. The server receives indexed ports using *indexed transition*s.
- Data can be distributed to or collected from the elements of an *indexed component*.
- Commands and reports from and to a central controller. This makes it possible for the controller to use port indexes to keep track on subsystems.

6.4.1 Indexed subcomponents

To define an indexed subcomponent the text property of component is of format

name{**[***index _variable***]**}**:***component_type*

Examples:

```
     maintain_display[Row][Column]
     maintain_reactor_temperature[Reactor_Id]:
          temp_controller
```

Indexed components are used when many similar but independent controls or monitoring processes are needed. It is easy to connect processes with different indexes to different ports. Moreover, it is easy to give each process a different set of parameters.

To give different set of parameters to processes with different index values you can connect a different element of a C -array store to each process. Give initial values for all using one array type initializer.

6.4.2 Connections to indexed ports

When connecting indexed ports the corresponding selector text is of form:

portname{**[***index_expression***]** }**:***datatype*

Note:

- An index expression may be constant or may contain one or more index variables.
- Total number of connections is the product of ranges of different index variables found in expressions.

6.4.3 Connections to indexed subcomponents

When a connection is connected to an indexed subcomponent, the index of the subcomponent must be specified in the beginning of the selector text.

[*component_index _expression***]***name***:***type*

Note:

- A component index expression may be constant or may contain one or more index variables.
- It is possible to connect indexed connections to indexed subcomponents.
- The range of the index expression does not need to cover all the bubble instances. Other selectors with the same instance name but different index expressions may cover the gaps. However, it is important, that each connection is connected exactly once to each bubble instance.

Examples:

```
    [0]alarm[j]:flag
    [Row][Column]new_character:char
    [Machine*N_SPINNERS+Spinner]thread:speed
```

6.4.4 What is really connected to which and how

A distinct connection is created for each combination of the possible values of the index variables found in the index expressions at the selectors at both ends of a connection (selector-connector-selector). The same index variable means the same value in each of the expressions. The values of different index variables are independent.

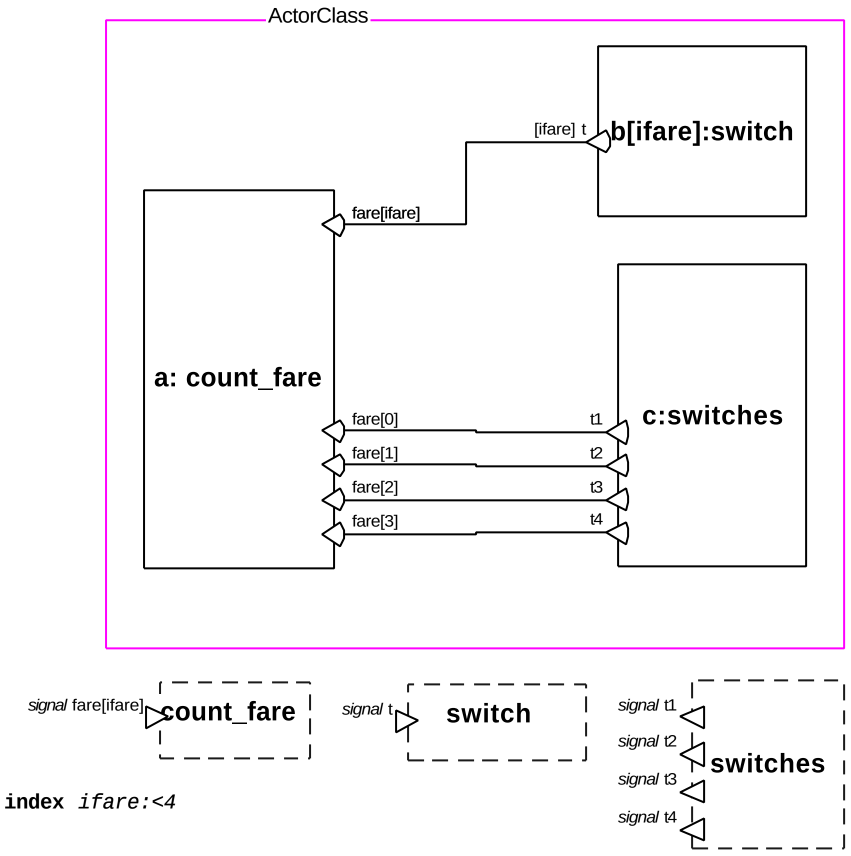


Figure 16: Examples of indexing.

In Figure 16, the component specifications are shown at the bottom.

First there is a connection from a to b where and indexed port is connected to an indexed component instance. Each element of the `fare` port is connected to an non-indexed port t of the indexed component b with same index as the port index. E.g. `fare[0]` of `a` is connected to `t` of `b[0]`, `fare[1]` of `a` is connected to `t` of `b[1]` and so on. Here the index `ifare` in both selectors of the connection binds the index expressions together.

In connections from `a` to `c` the indexed port is connected into separate ports in component `c`.

7. INITIALISING SYSTEM VARIABLES

The purpose of initial values is to make the initial state of a system deterministic.

Each store and continuous connection must be given explicit or implicit initial value or it must be explicitly left uninitialized.

7.1 explicit initialisation

The initial value can be written directly to store, port or selector symbols both component and state-transition diagrams:

name`:`*type*`:=`*initializer*

Initializer is a expression or macro applicable as an initializer in C.

7.2 implicit initialisation

An implicit initialisation is connected to a data type. The initializer is written to a header file included into diagram:

`#define` *type*_**`init`** *initializer*

7.3 explicit uninitialisation

It is practical not to define an initial value for large data structures. They are better to be initialised by the program. For such large data types put a definition:

`#define` *type*_**`uninitialized`**

to a suitable header file to be included in the diagram.

8. MODELLING EXAMPLES

8.1 A simple test environment

It is advisable to test a system somehow before it is connected to real power machinery. Professional testing methodologies are outside the scope of this manual. However, you may gain confidence to the operation of the system, or you may find errors, using a simple interactive simulation environment provided with the ReaGeniX Visual Programmer.

To run the diagram in an interactive simulated environment, make a working directory. Let's call it *workpath* for now on.

Copy all files from the directory `\EXAMPLES\EXAMPL01` of the ReaGeniX directory a to *workpath*.

Generate code from Visio Diagram (see ch.2.2).

Check the file `MODULE.T` that controls generation of the testing environment. It should contain the line

```
#include "shopdoor.h"
```

`MODULE.T` must include the header file of the main module of the subsystem to be tested.

Let's use *rgxpath* for now on to denote the path to the ReaGeniX compiler and the associated files e.g. `C:\REAGENIX`.

Compile the following files

`SHOPDOOR.C`

rgxpath`\SUPPORT\TEST.C`

Those files include macro library `REACTIME.H` and a parameter file `REACTIME.P`, both of which are found in *rgxpath*`\INCLUDE`.

Link the results to file `SHOPDOOR.EXE`. No special run-time libraries are needed.

For Turbo-C 2.01, the following procedure is applicable:

1 Check that the project file `SHOPDOOR.P20` contains the following:

`XDOORCTR.C`

rgxpath`\SUPPORT\TEST.C`

2 Start Turbo C
`TC XDOORCTR`

3 Select **Options-Directories-Include directories**. Append "`;`*rgxpath*`\INCLUDE`" to the end of include directory list. (Using cursor keys first retains the original list.)

4 Select **Project-Project name**. Select `XDOORCTR.PRJ` as the project file.

5 Select **Project-Auto dependencies On**.

6 Select **Options-Save options** for further use. Save options to local directory in order not to interfere other activities.

7 Select **Run** (or Ctrl-F9).

Start the tester with command

`SHOPDOOR`

Not needed, if you started it directly from Turbo-C.

Now, the display should look like this:

```
 keep_open                                low

 shop_door                                disabled

             --- total time = 0.000000 s ---
 e = exit test    c = configure    g = GO
 i = show inputs  s = show states  o = show outputs
 r = RESET        t = time lapse

 1 somebody                               low

choice :
```

Let's start and reset the system with command
`r`

The system responds by resetting the output keep_open to low state and going to the closed state.

```
 keep_open                                low*
```

```
  shop_door                                closed

              --- total time = 0.000000 s ---
  e = exit test    c = configure    g = GO
  i = show inputs  s = show states  o = show outputs
  r = RESET        t = time lapse

  1 somebody                               low

 choice :
```

Simulate a person entering in the detection range. Toggle flag `somebody` by typing

```
1
```

The response should look like this:

```
  keep_open                                high*

  shop_door                                people_detected

              --- total time = 0.000000 s ---
  e = exit test    c = configure    g = GO
  i = show inputs  s = show states  o = show outputs
  r = RESET        t = time lapse

  1 somebody                               high

 choice :
```

You can see the new value of the input flag, and that the system has initiated opening the door by driving the output flag `keep_open` to `high` state. The system is now in the `people_detected` state.

Let's assume that the person enters the door and goes out of the detection range. Drop somebody by giving

```
1
```

again.

```
  keep_open                                high

  shop_door                                give_time_to_pass
```

```
            --- total time = 0.000000 s ---
e = exit test    c = configure    g = GO
i = show inputs  s = show states  o = show outputs
r = RESET        t = time lapse

1 somebody                              low

choice :
```

The system is transferred to the `give_time_to_pass` state, `keep_open` is still `high`.

Let the time pass by giving

g

It advances the simulated time until the next timeout occurs in the system.

```
keep_open                               low*

shop_door                               closed

            --- total time = 3.000000 s ---
e = exit test    c = configure    g = GO
i = show inputs  s = show states  o = show outputs
r = RESET        t = time lapse

1 somebody                              low

choice :
```

It required 3 seconds before the system started closing the door. The person had time enough to pull his/hers heels from the door opening. Now we are at the initial state again.

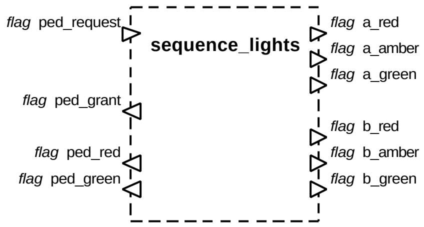
Hit

e

when you want to exit.

As an exercise, you could try to add a signal close door button into model. Pressing the button closes the door immediately if there is the door is clear.

8.2 A Compound System: a Traffic Light Controller



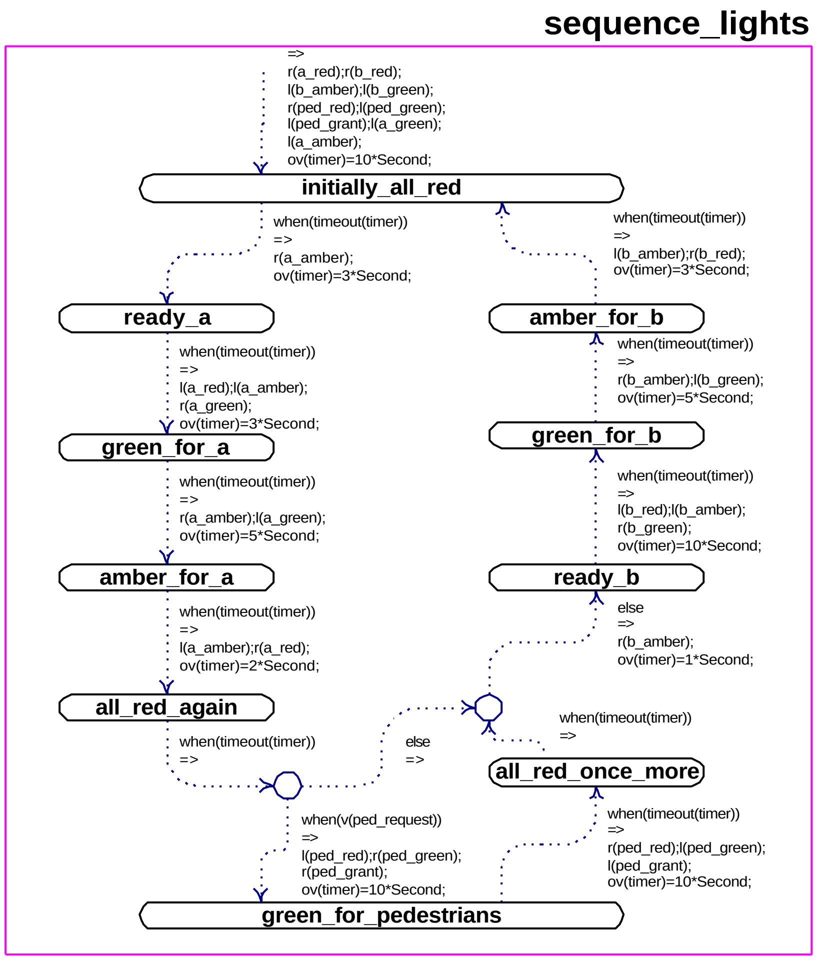
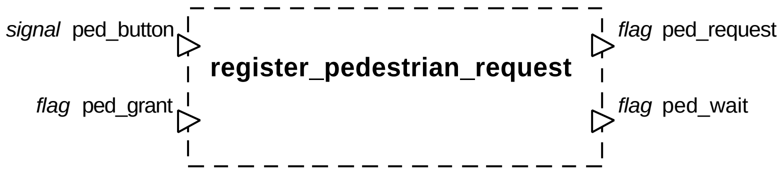

sequence_lights
=> r(a_red);r(b_red); l(b_amber);l(b_green); r(ped_red);l(ped_green); l(ped_grant);l(a_green); l(a_amber); ov(timer)=10*Second;
initially_all_red
when(timeout(timer)) => r(a_amber); ov(timer)=3*Second;
when(timeout(timer)) => l(b_amber);r(b_red); ov(timer)=3*Second;
ready_a
amber_for_b
when(timeout(timer)) => l(a_red);l(a_amber); r(a_green); ov(timer)=3*Second;
when(timeout(timer)) => r(b_amber);l(b_green); ov(timer)=5*Second;
green_for_a
green_for_b
when(timeout(timer)) => r(a_amber);l(a_green); ov(timer)=5*Second;
when(timeout(timer)) => l(b_red);l(b_amber); r(b_green); ov(timer)=10*Second;
amber_for_a
ready_b
when(timeout(timer)) => l(a_amber);r(a_red); ov(timer)=2*Second;
else => r(b_amber); ov(timer)=1*Second;
all_red_again
when(timeout(timer)) =>
when(timeout(timer)) =>
else =>
all_red_once_more
when(v(ped_request)) => l(ped_red);r(ped_green); r(ped_grant); ov(timer)=10*Second;
when(timeout(timer)) => r(ped_red);l(ped_green); l(ped_grant); ov(timer)=10*Second;
green_for_pedestrians

Figure 17: A Traffic Light Sequencer.

The basic sequence of traffic lights is presented in Figure 17. It uses a set of flags to control the lights. The lights are assumed to operate so that the value false means dark and the value true means light. The crossing directions are called a and b. The opposite directions are signalled equally. The pedestrians of both directions are shown green only, when the cars of both directions are shown red.



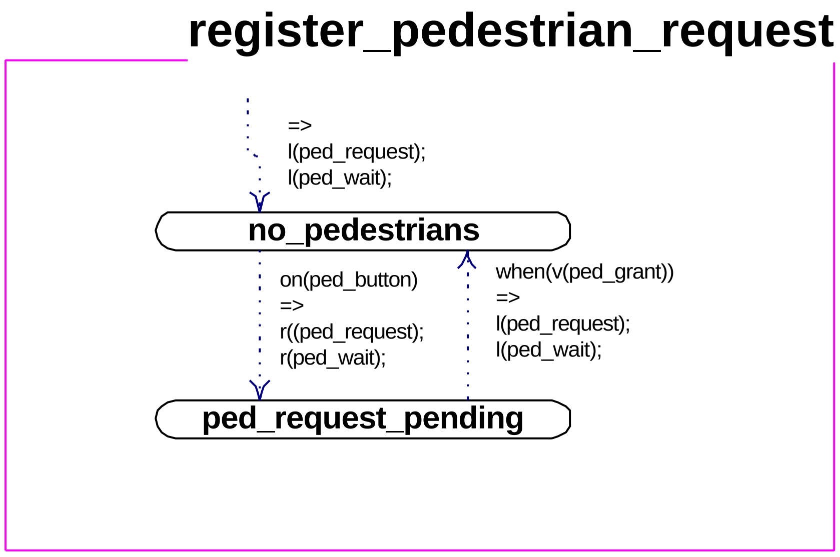
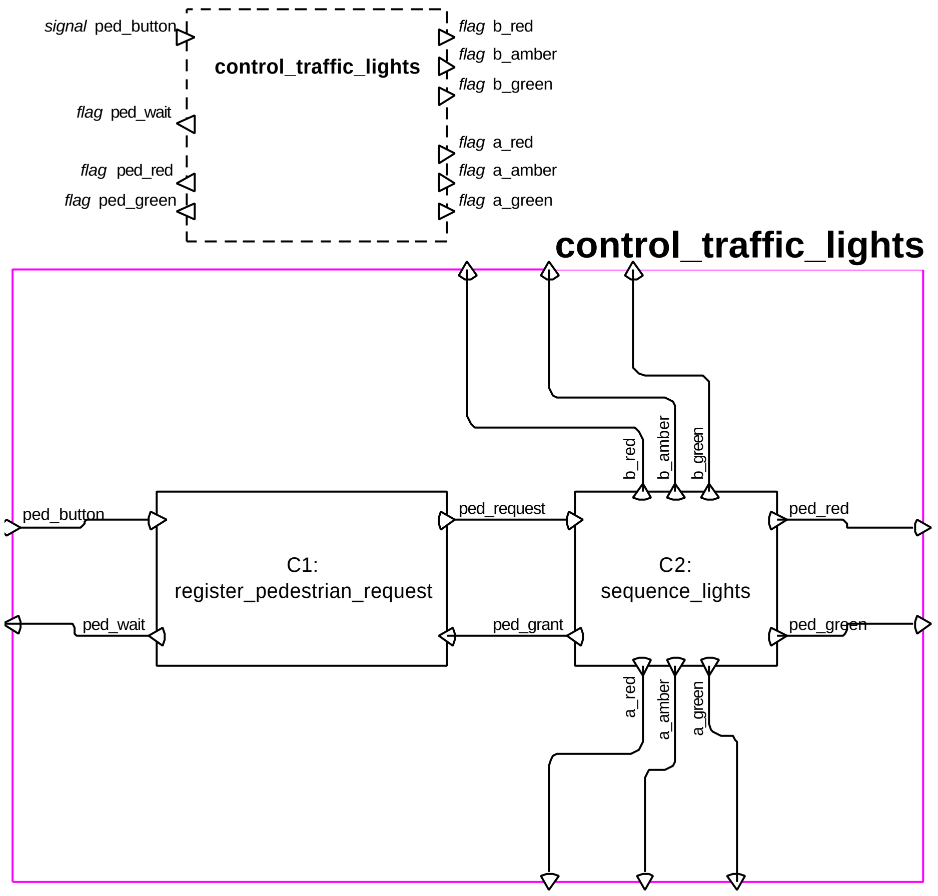


Figure 18: A Pedestrian Button Registration.

Registration of pedestrian passing requests is presented in Figure 18. A signal `ped_button` is received every time a pedestrian presses a request button. The flag `ped_grant` is high, when the pedestrians are allowed to pass. The logic is arranged so, that `ped_button` raises the request flag `ped_request` and lights the wait light by raising `ped_wait` . The wait light is lit until the pedestrians are let to pass. If the button is pressed while there is a green light for pedestrians, a virtual flash of the wait light takes place. However, the flash does not take *real* time, it cannot be seen, and it does not stress the power electronics or the light bulb.



```
include xtlight2.h
include xtlight3.h
```

Figure 19: A Traffic Light Controller.

Figure 19 shows an architecture diagram. It combines the state machines and makes them a system (a system is just a new component type).

As in state transition diagrams the diagram needs a component body symbol and component specification symbol for the new component type defined by the diagram.

The state machines are represented by component (an actor) instances whose label consists of instance name and component type name.

A connection of ports of the intercommunicating components is drawn using two selectors and a connector. Selectors (sector symbols) which are snapped into component instances select ports to be connected. A connector symbol connects the matching selectors together. The direction of a selector is presented by the direction of sectors sharp corner and it must match the direction of the selected port.

A connection of a port of a component to the port of the external interface is drawn using two selectors and a connector. Selectors (sector symbols) select ports to be connected. One selector is snapped to the component and another is snapped to the component body symbol framing the diagram. A connector symbol connects the matching selectors together. The direction of a selector is presented by the direction of sectors sharp corner and it must match the direction of the selected port.

The actor components in an architecture diagram represent independent, concurrent subprocesses. Those subprocesses communicate by discrete and continuous communication or shared stores. A signal is a special type of discrete communication who has no data value. A flag is a special case of continuous communication whose datatype is R_boolean. In addition, direct control by switching another subprocess on or of is possible. **Other kinds of inter-process communication are not recommended** (global variables, OS calls, etc.)**.**

To follow the subsequent walkthrough example you may take paper copies of the diagrams and use paper clips on the diagrams to show the current states and flags being high.

Lets consider the case when green is shown to the b direction. Moreover, nobody has pressed the button since last green for pedestrians. The flag `b_green` is high (true) and all the others are down (false). Now, a pedestrian presses a button. Let's assume that pressing the button down, rather than releasing it, causes the signal `ped_button` to be injected to the system. From the architecture diagram in the figure 19 the signal goes to the bubble `register_pedestrian_request`.

The state machine `register_pedestrian_request` in the Figure 18 is in the state `no_pedestrians` since button has not yet been pressed. Now, the signal `ped_button` triggers the transition to `ped_request_pending`. The transition lits the wait light and raises the `ped_request` flag.

The state machine `sequence_lights` in Figure 17 is showing green light to b direction in the state `green_for_b`. It does not immediately respond to the new value of the flag.

The machine `sequence_lights` steps thru the light sequence controlled by the `timer` until it has shown enough all reds in the state `all_red_again`. When the time is out and the machine leaves that state, a decision is made depending on the value of the flag `ped_request`. Because the flag is high, the path to the state `green_for_pedestrians` is taken, `ped_green` is lit, and `ped_grant` is raised.

The state machine `register_pedestrian_request` is still in the state `ped_request_pending`. It responds immediately to raising of the `ped_grant` by dropping the `ped_request` and turning off the wait light.

New pedestrian requests cannot be registered until red is lit for pedestrians again.

The light sequence continues controlled only by the timer until the `sequence_lights` is leaving the state `all_red_again` again.

8.3 Component instancing: Ping-Pong

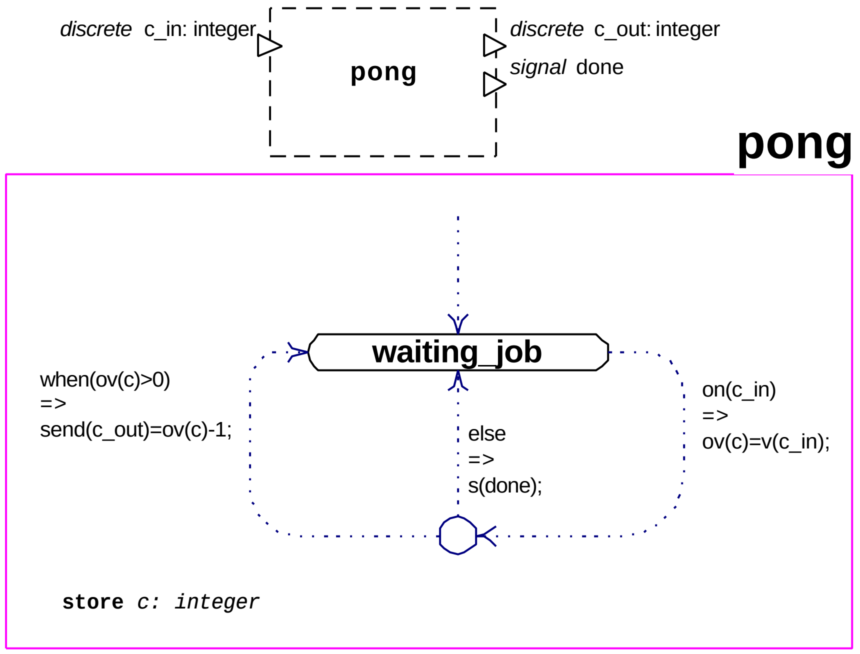


Figure 20: Pong

In Figure 20 is presented a transition diagram of subcomponent pong. Its struction is very simple. Pong gets in integer value called c_in and if c_in is more than zero it sends out integer value called c_out but if c_in is zero, it sends signal called done.

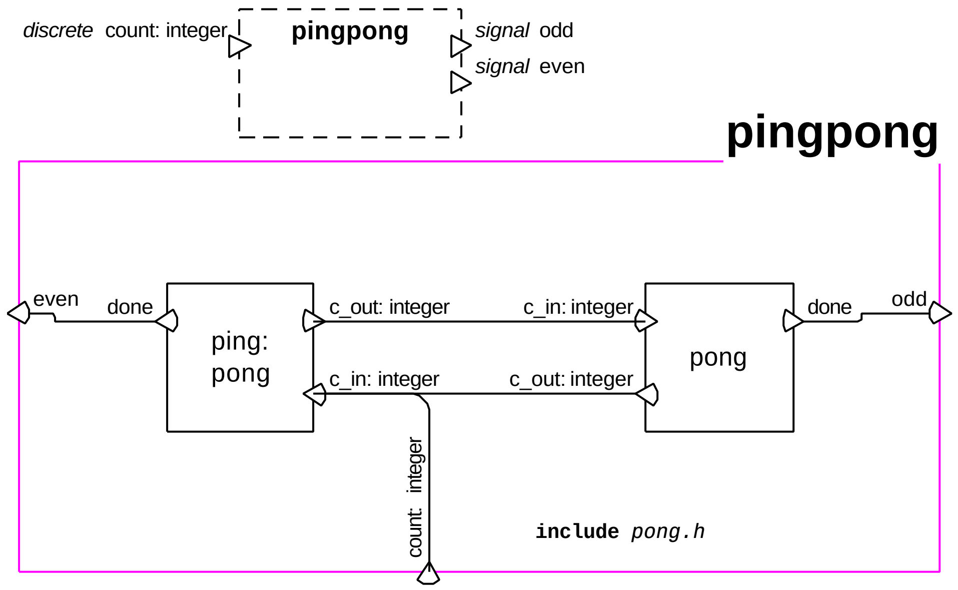


Figure 21: A Pingpong

Figure 21 shows an architecture diagram Pingpong. In the diagram, there are two subcomponents named ping: pong and pong. The pong could be also named pong: pong because the first pong means a name of component and the pong after colon a type of component. We can see that subcomponents have same names of inputs and outputs. Both subcomponents have also samekind inside structure and same operations. So both ping:pong and pong are a same type. A component named ping is same type as pong so it points to the type of pong

If components are a same type, there is no use to make two samekind pictures by different names. The best way is point to the type of component as in the Pingpong diagram in subcomponents name ping:pong have done.

8.4 Data Processing: a Weighing System

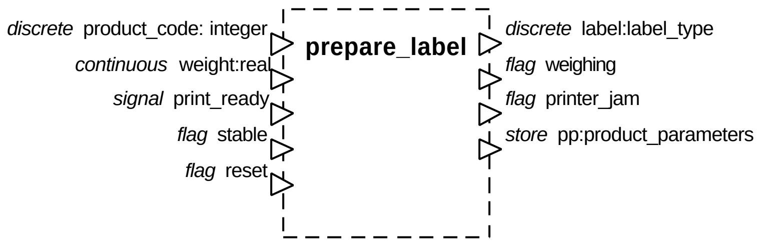
In data processing operations data is received from discrete and continuous communication inputs, accessed from stores, new data is computed from previous values, and the results are stored to stores or sent via outputs for further processing.

Data processing is frequently initiated by data arriving via a *discrete communication*. A discrete communication passes messages, which can contain a data value. The receiver should either use the value immediately or store it for further use. Each message is considered meaningful even if it contains the same value as the previous one.

The arrival of data from a discrete communication is detected by an macro `on(`*portname*`)` operator in the beginning of a transition condition. The data value of communication is accessed by a `v(`*portname*`)` operator. The `v()` operator can be used both in conditions and actions. Moreover, `v()` can be used to assign a value to an output communication.

Data stores may be internal or external in respect to a system component. Both store data for further use. External data stores may be shared with other system components. The data value of an external store is accessed by a `v(`*store*`)` ("value") operator. The data value of a local store is accessed by an `ov(`*store*`)` ("own value") operator. Both operators can be used both in conditions and in actions. The `v()` and `ov()` operators work both in reading the value and assigning a new value.

Output data is either stored to an external store or it is sent via a data flow. If the structure of the data of an output flow is elementary, it is convenient to use the `send(`*flow*`)=`*expression*`;` type statement to send a new value to a flow. If the data is a structure, new values are assigned to members using `v(`*flow*`).`*member*`=`*expression*`;` type statements and finally the data is released by `emit(`*flow*`);` for distribution.



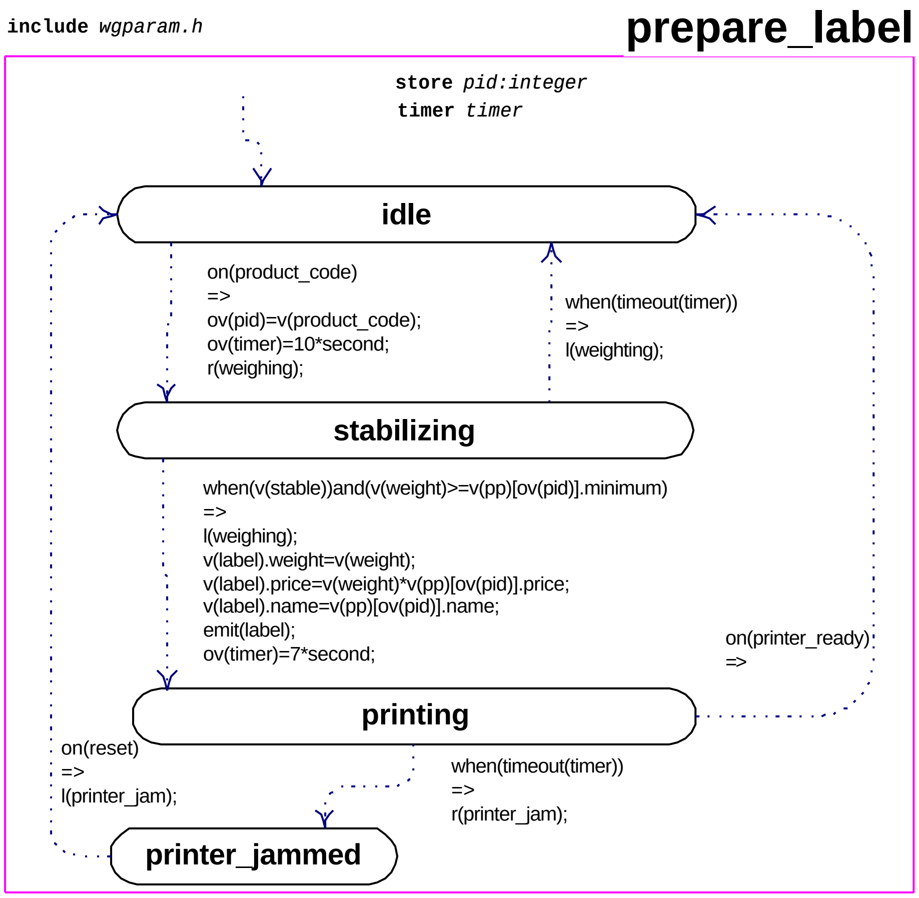


Figure 22: Label Printing Logic

The label printing logic of an automatic grocery store scale is presented in Figure 22. The operation is launced upon arrival of product code data. The product code is stored in a local variable `pid` for further use using statement

```
     ov(pid) = v(product_code);
```

If the weight reading stabilizes within 10 seconds the price is computed using statement

```
     v(label).price = v(weight)*v(pp)[ov(pid)].price;
```

and the label data is sent for printing. The complex expression above (boldface) selects the member `price` from the `pid`'th element of the array `pp`.

In above example the `label` structure is filled member by member. When the whole data structure is completed, it is sent by

```
     emit(label);
```

If you forget to acknowledge printing by the `print_ready` signal, the system will go to the printer_jammed state and requires the `reset` signal.

8.5 Connecting to Real-Time and Real-World

This example is specific to IBM style PC and MS-DOS environments and Turbo/Borland C run-time library. Other environments may require modifications to the example and the support files.

The simplest way to connect a system generated by the ReaGeniX is to use a polling (non-pre-emptive) real-time environment main.c. The main.c is a minimum environment, which provides time for timers and an `Idle` signal for polling peripherial devices.

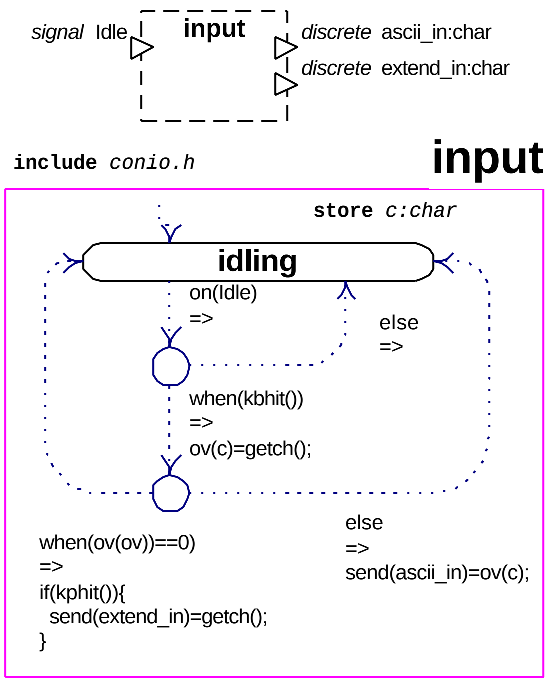


Figure 23: Real-Time Keyboard Input

In Figure 23 you can see how the keyboard input buffer is polled each time the processor becomes idle. If a key has been hit, a BIOS interrupt handler has stored the character to the keyboard input buffer. If there is a character in the keyboard input buffer, the `kbhit()` returns `1` (nonzero) and the `getch()` returns the character code. For non-ASCII keys (e.g. ALT-codes and function keys) a sequence (zero, function code) is returned.

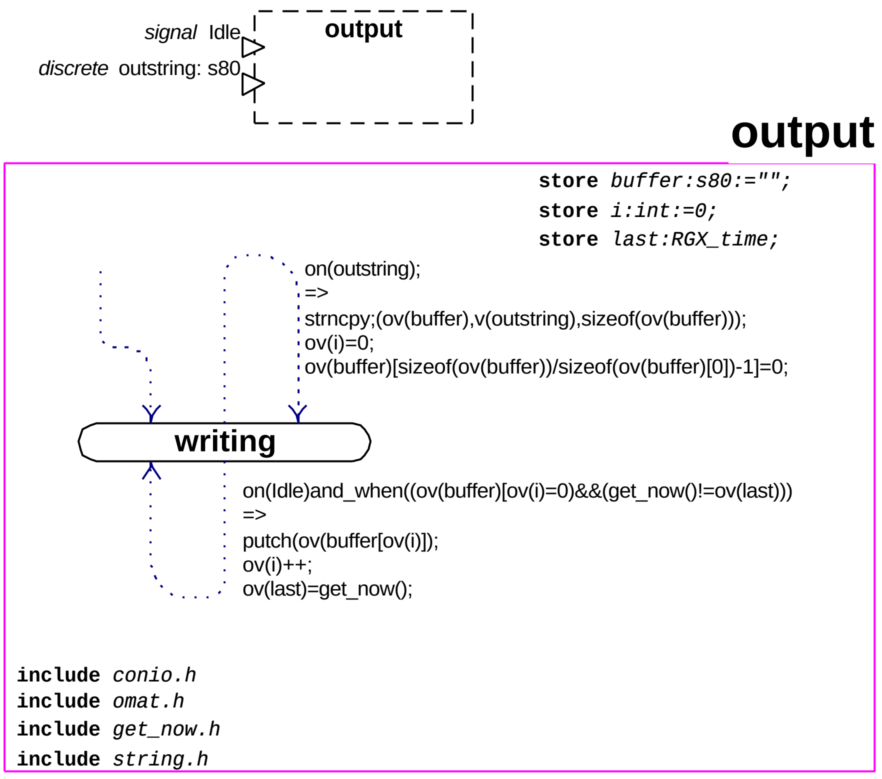


Figure 24: Simulated Character Based Output

Figure 24 shows how to control the output rate using an external condition with `Idle` signal. In the example the external condition is `get_now()!=ov(last)`. This gives output at the same rate than the system time is updated (18.2 Hz in PC). In a real application, the test may be upon transmitter-buffer-not-busy etc.

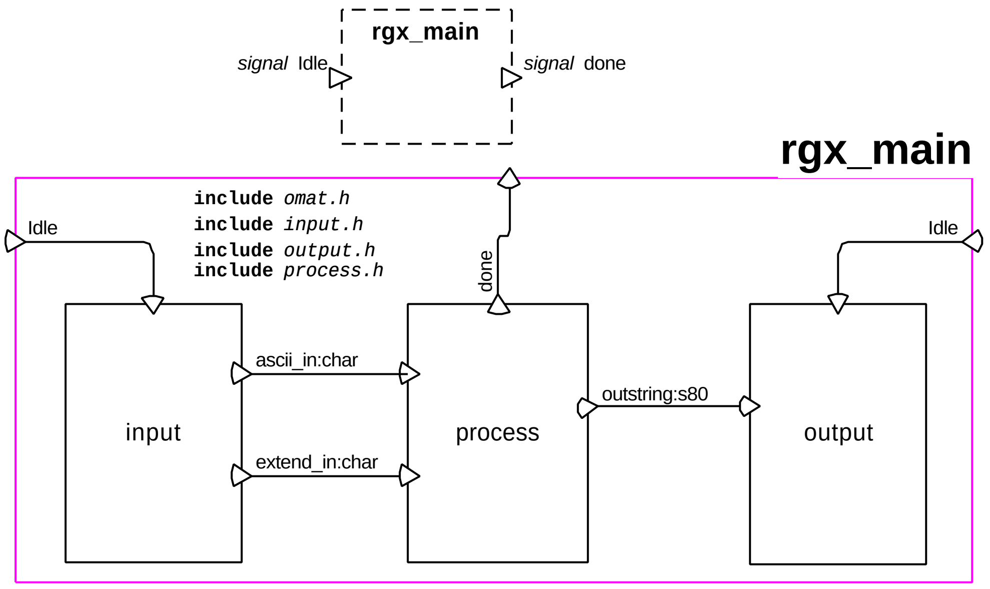


Figure 25: A Simple Stand-Alone Real-Time System

Figure 25 shows the overall organization of the example system. The environment sends in the `Idle` signal each time the processor is found to be idle. The application terminates by sending `Done`.

8.6 Indexing: A Mass Storage Multiplexer

The problem:

> A mass storage driver accepts requests via a dataflow. When the request has been processed, the driver sends the response via another dataflow. A request is discarded if the handler is processing the previous request.
>
> The mass storage multiplexer allows several clients to share the mass storage. The interface between a client and the mutiplexer is similar to the interface between the multiplexer and the driver so that it is possible, without modifications, to connect a client to the driver either directly or via the multiplexer.

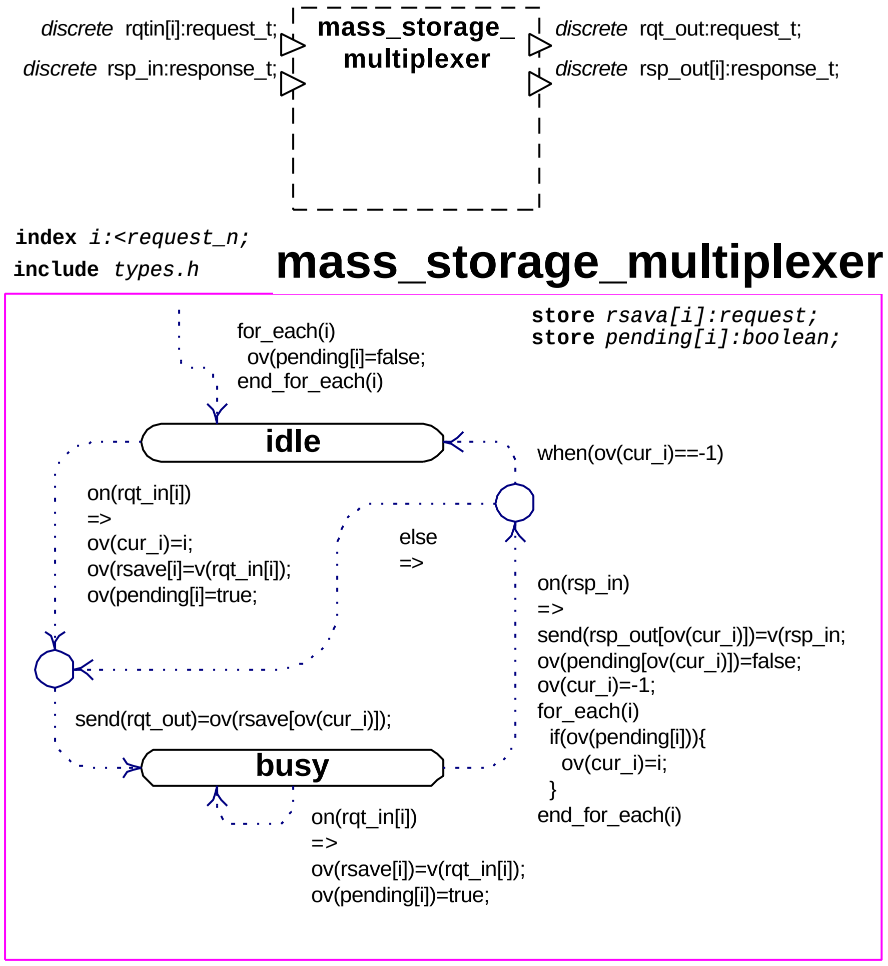


Figure 26: The Mass Storage Multiplexer

The multiplexer accepts requests from the indexed dataflow `rqt_in`. It sends the response to the indexed dataflow `rsp_out` with the same index value. If the mass storage driver is idle, the multiplexer passes the request immediately to `rqt_out`. Responses from the driver arrive from the dataflow `rsp_in`. The multiplexer immediately passes the response to the `rsp_out` with the proper index.

If there is an unprocessed request when the response arrives from the driver, it is sent to the driver. Unprocessed requests are stored in the internal store `rsave`. The selection of the next request is simple but relatively unfair in this example. It is, however, easy to modify the solution for more sophisticated resource scheduling algorithm.

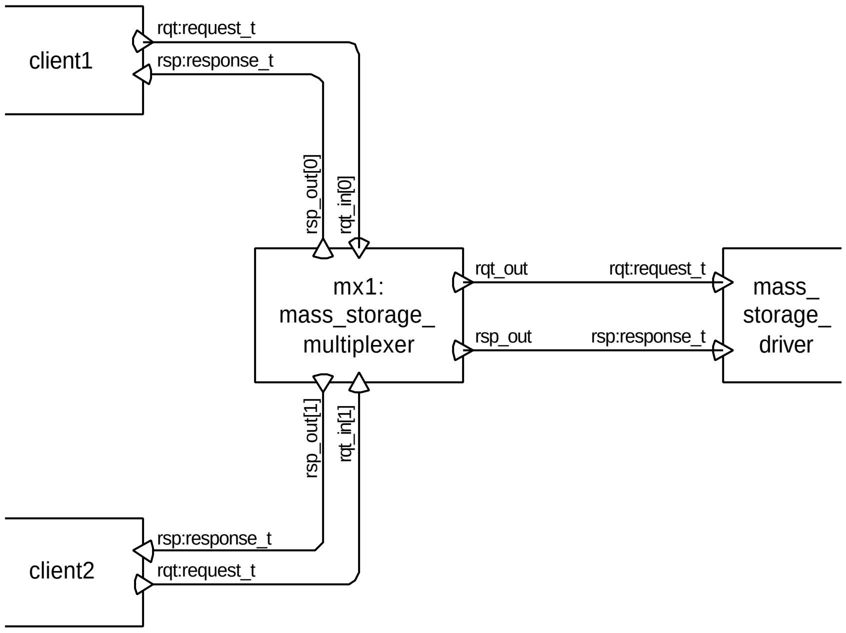


Figure 27: Examples of Indexed Connections

This diagram connects two clients to mass storage driver via a mass storage multiplexer. A client sends requests via the `rqt` flow connector and accepts responses via the `rsp` connector. The request and the response flows of a client are connected to the mass storage multiplexer using same index values in the `rqt_in` and `rsp_out` connectors of the multiplexer.

9. APPENDIX 1

9.1 ReaGeniX components for Prosa tool

The following files are common with the Prosa version:

*Rgxpath***reagenix.scy** - ReaGeniX licence file.

*Rgxpath***Include** - ReaGeniX C - header files including reactime.h.

*Rgxpath***Support** - ReaGeniX support C -modules including test.c and main.c.

*Rgxpath***Examples** - Some modelling examples with TurboC 2.0 project and configuration files. Examples of generated code after macro preprosessing are included.

The Prosa version 1.4 of ReaGeniX generator including ReaGOS is in directory:

*Rgxpath***Prosa**