Robox Motion control Documentation

Release 1.0.0

Abed

Aug 17, 2018

CHAPTER 1

Robox S.p.A.

Robox a company started in 1975, designs and manufatures electronic controllers, programming languages, development environments for robotics and motion control systems.

Its broad range of products permit to deal with any applications, from the simplest ones (one or two controlled axes), to the most sophisticated ones (dozens of controlled axes) thanks to the availability of architectures which can be “modular”, “compact” or even integrated in brushless drives. Innovation and quality have been Robox’s main goals since the very beginning.

Innovation has always been pursued keeping in mind the global reliability (present and future) of the product.

Quality has always been ensured by appropriate design choices and an accurate selection of materials. The respect for cogency and the continous improvement of the Quality Management System ensure the achievement of the mentioned objectives

`Robox product catalog` contain complete informations about Robox controllers, drives, HMI, softwares and packages and libaries (G-code, Robot kinematics, PLCopen, etc.). Download and check the catalog before going on with this documentation.

Note: The purpose of this document is to show the use of ROBOX products. It is a kind of tutorial. Even if a lot of examples deal with the basics, the purpose of this tutorial is not to teach how to use Microsoft windows neither how to learn to program from the begining. Previous knowlodge of the basics of any programming language (know how to write a simple program, e.g. 10 lines of instructions) and the use of Linux or Windows operating systems (changing ip address) is assumed.

Note: The appendix Fundamental of automation cover the basics and principles of industrial controllers (PLC), sensors and actuators.

Note: Obvious steps are not listed in the tutorial. e.g. when we say connect the controller to the computer it si obvious that you have to power it on, plug the ethernet cable I don't know where, etc. Otherwise I encourge you to watch **The Lego Movie**

1.1 Documentation

This documentation will show the use of Robox follwing products:

- RP1, RP2 : Robox compact controllers
- RDE : Robox Development Environment
- RDT : Robox Display tools (HMI)
- IMD : Robox Integrated Drive

1.1.1 Overview

Robox RP1 and RP2 are 2 compact motion controllers. They can be programmed in Ladder, in R3 language (Robox Strucuted text) and in C++. The IDE is called RDE (Robox Integrated Environment).

In this part we will see Robox IDE for motion control called RDE, RTE that is the real-time operating system of Robox and the commissioning of AGV using its already existing software written in R3 and Object block (C language).

Overview

Compact Controllers

Contoller programs are stored on memory card. RP1 use a compact flash memory and RP2 use a microsd memory. Robox realtime operating system called RTE usually is present in the memory card in the folder `/f@`. If not present on the memory card the binary file, rte_platform_name_version.bin, can be downloaded from Robox website. The memory is provided by Robox together with the licence, present in the folder `KEY`, and the last version of RTE.

Note: **ppc-ge** is for RP1, **arm-a9** is for RP2. RP1 MCU is a PowerPC freescale. RP2 MCU is an Intel ARM9 dual core

---

For technical specification, CAD and electrical drawings check Robox website.

On the website you can find other Robox controllers.

Fig. 1: RP1: 8 Genral purpose tasks, 32 RULEs (realtime motion tasks)
Up to 32 interpolated axes driven by Ethercat or CanOpen. Compact flash, RS232, RS485, Profibus, Ethercat, 2 CanOpen, Ethernet/IP, Integrated IO, Native interface to Pheonix Axioline IO

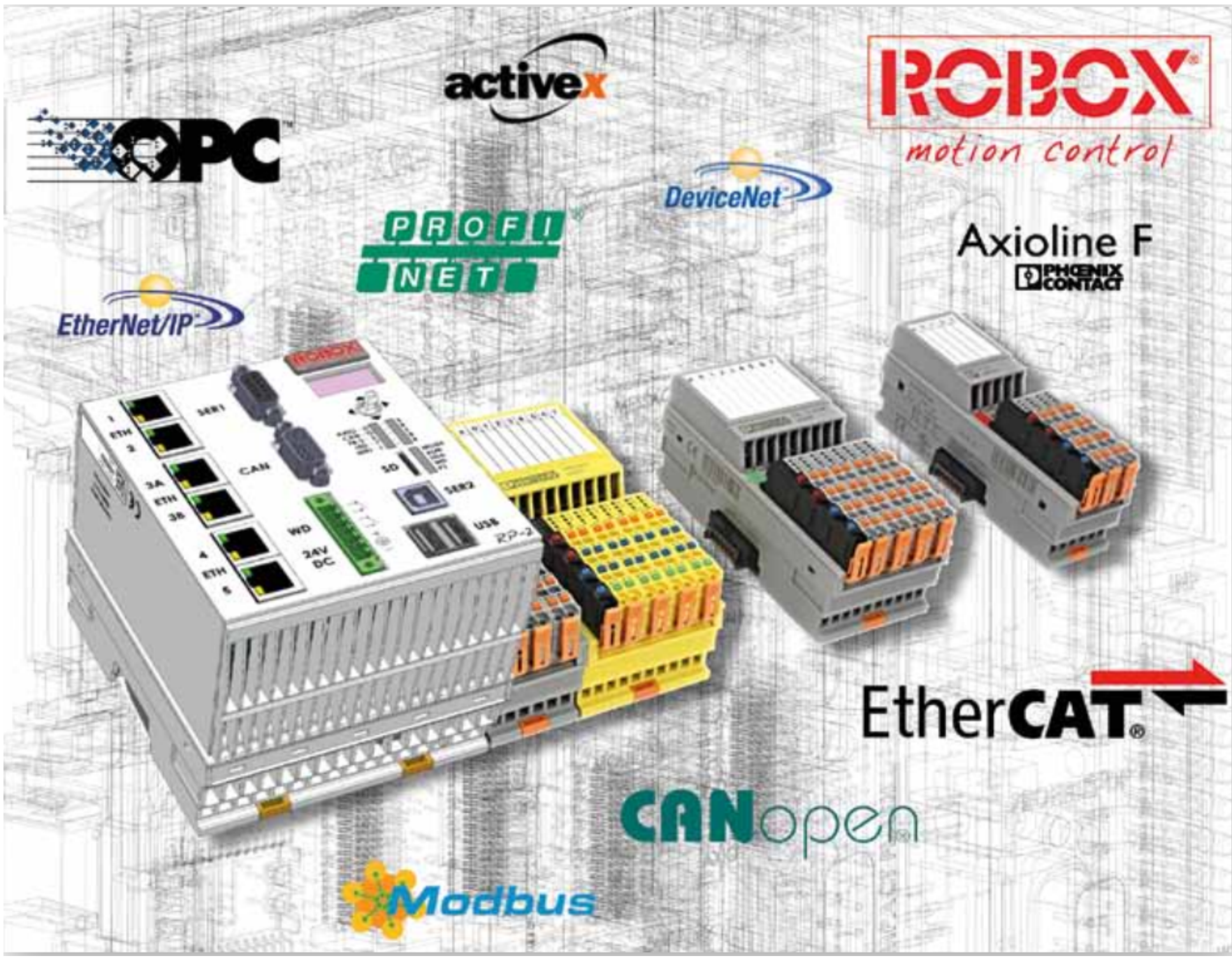


Fig. 2: RP2: 8 Genral purpose tasks, 32 RULEs (realtime motion tasks)
Up to 250 interpolated axes driven by Ethercat or CanOpen. MicroSd, RS232, Ethercat, CanOpen, Ethernet/IP, ProfiNEt, Native interface to Pheonix Axioline IO, 2 USB type A, 1 USB port type B, WiFi (internal webserver)

Controller program

Robox realtime operating system RTE, basically can execute until 8 tasks in time sharing (round robin) and one high priority periodic task called `Rule`. The maximum (recommended) frequency of the `Rule` is 200Hz(5ms) for RP1 and 1000Hz (1ms) for RP2.

Taks usually are used for general purpose control like a PLC, and the `RULE` is used for motion control. RTE can execute until 32 Rules.

Compared to Siemens PLC, tasks can be like OB1. So we can have until 8 OB1 executed in parallel. And Rules can be compared to an interrupt OB e.g. OB35, fixed time interrupt.

Note: Task1 is executed automatically by RTE. Taks1 have to call others task and rules in the initialization phase. This can be done with the instruction `mt_en(task_number)`

In the following images show how tasks are executed in principle. Schedulars differ from one operating system to another.

Note: More about RTE scheduler, Multitasking and RULEs will be discussed in the chapter related to motion control.

Controllers memory

Robox controllers have a memory card where RTE (Real Time Extended) and RPE (Robot Path Executor) binary file are saved, together with the program files, configuration files, etc.

They contain also a retentive memory, which dimension depend on the controller type. Non volatile registers (nvr, nvrr, nvsr) and retentive user defined structures are retained in this memory. You can make a bakup of the values of the retentive memory and save them into a formatted text file usally with extension `.stp`.

The dimesion of each type of register can be determined in the project configuration, as we will see in RDE chapter. Remember that registers are indexes of memory areas, like **Merkers** in Siemens PLCs, and can be Integer (nvr), Real (nvrr) and String (nvsr). Robox controllers have also the same types of registers but volatiles (r, rr, sr).

Safety

1.1.2 RDE - Robox Development Environment

First step

In order to getting started with the controller, we need a memory card where we have to copy `RTE` and some configuration files. A new memory card will have the folder `KEY` that contain Robox licence and the folder `/f@` with the last version of RTE. The RTE binary file can also be downloaded from Robox website.

Note: Don't delete the folder **KEY** from the memory card. I contain the licence.

After the installation of `RDE`, `RCE` and `Icmap`, we need to copy in the installation directory, usually Robox, the license in order to compile programs. The license is provided by Robox.

Note: `Microsoft c++ redistributable 2010 and 2015 x86` must be installed on your computer

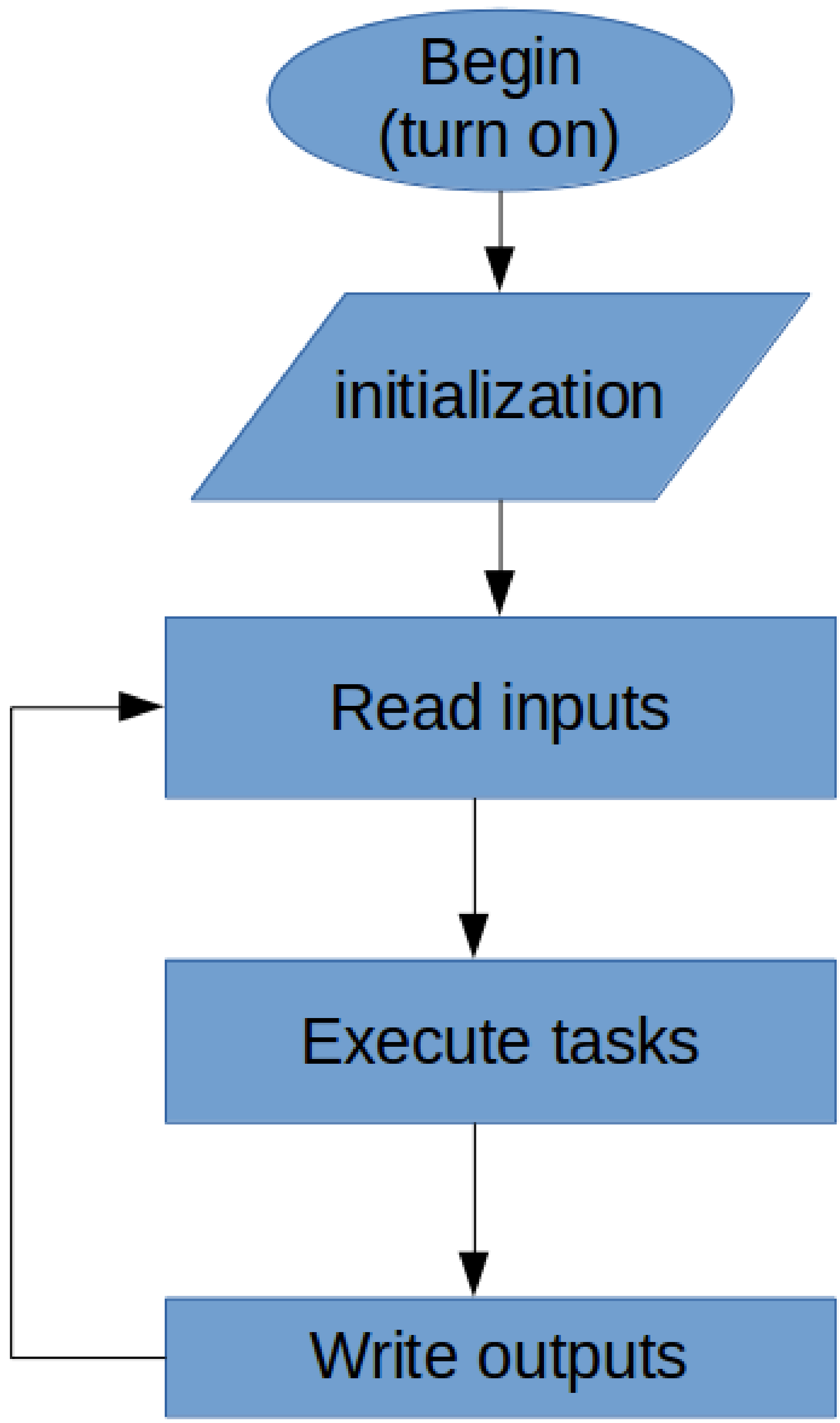


Fig. 3: One task execution

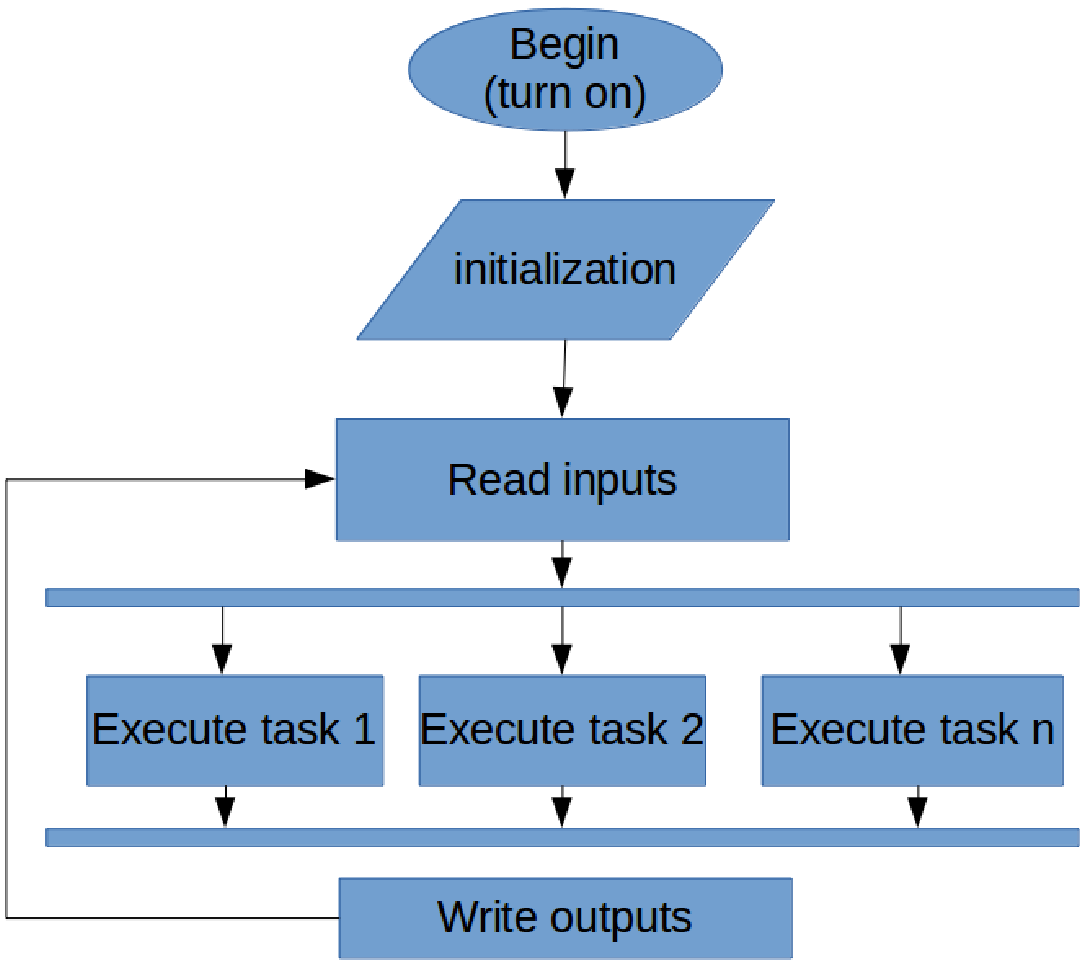


Fig. 4: More than one task

Note: RDE is the IDE of Robox, RCE is R3 and OB compiler, ICmap is the AGV's maps compiler. ICmap can be installed only if we use AGV, otherwise is not needed.

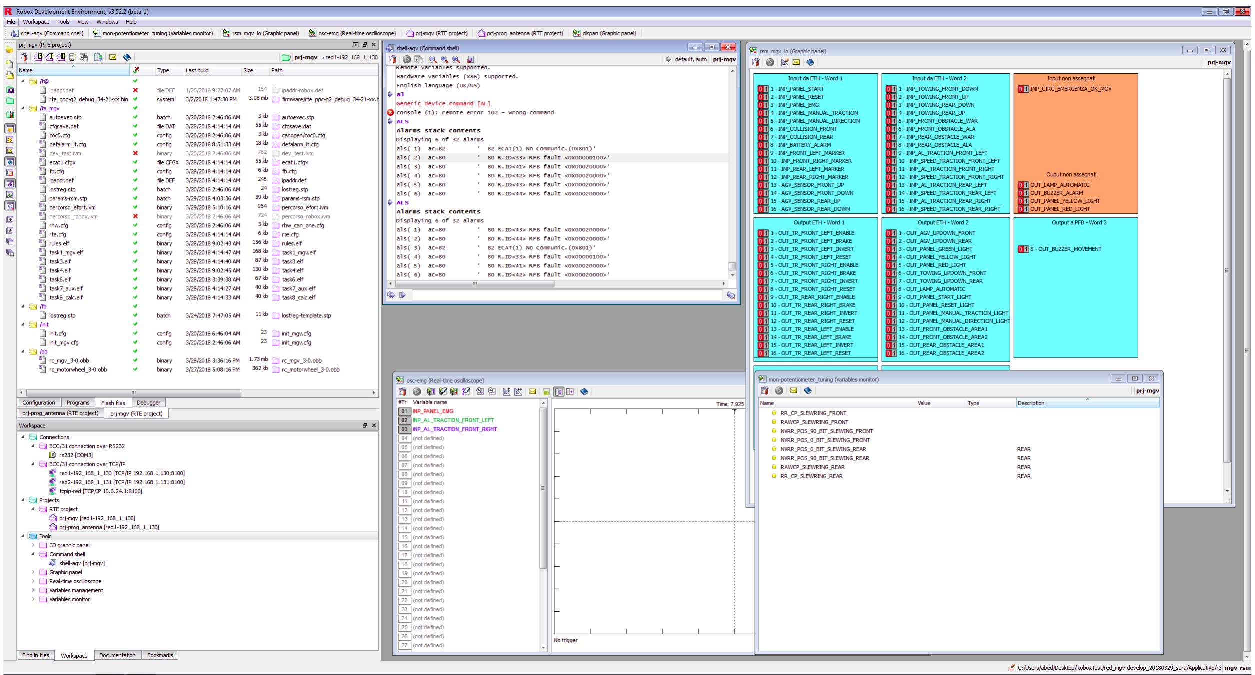


Fig. 5: RDE main windows

New RTE project

RDE like others IDEs (Eclipse, Atom, Visual studio code, etc.) use workspaces. One workspace may contain more than one project. So before creating a new RTE project, a workspace have to be created.

In the menu bar, the workspace menu, allow to open, create and manage workspaces, also to access the predefined examples. We can create as many workspaces as we want. Usually one project or machine program have its own workspace.

The ***New RDE workspace and RTE project*** **animation illustrate step by step how:**

- Create a new space
- Create a connection to a controller
- Create a new project
- Choose the compiler

Fig. 6: New RDE workspace and RTE project

RTE project was created, and a connection. The connection should have the same ip address of the controller.

The following animation illustrate how a R3 program and an Object block can be created in an RTE project.

Note: Remember to save the workspace after any modification: Workspace –> Save workspace

Fig. 7: R3 program and Object block (OB)

Tasks and Rules are R3 programs. When the keyword `$TASK n` where `n` is the task number, e.g. `$TASK 1`, the R3 program become a task. If the keyword `$rule` is used the R3 program become a RULE.

An RTE system files can have different folders. This demo use the default folders:

- f@ : RTE binary file
- fa : R3 programs and configuration files
- ob : Oject block compiled files

Fig. 8: Memory card default folders

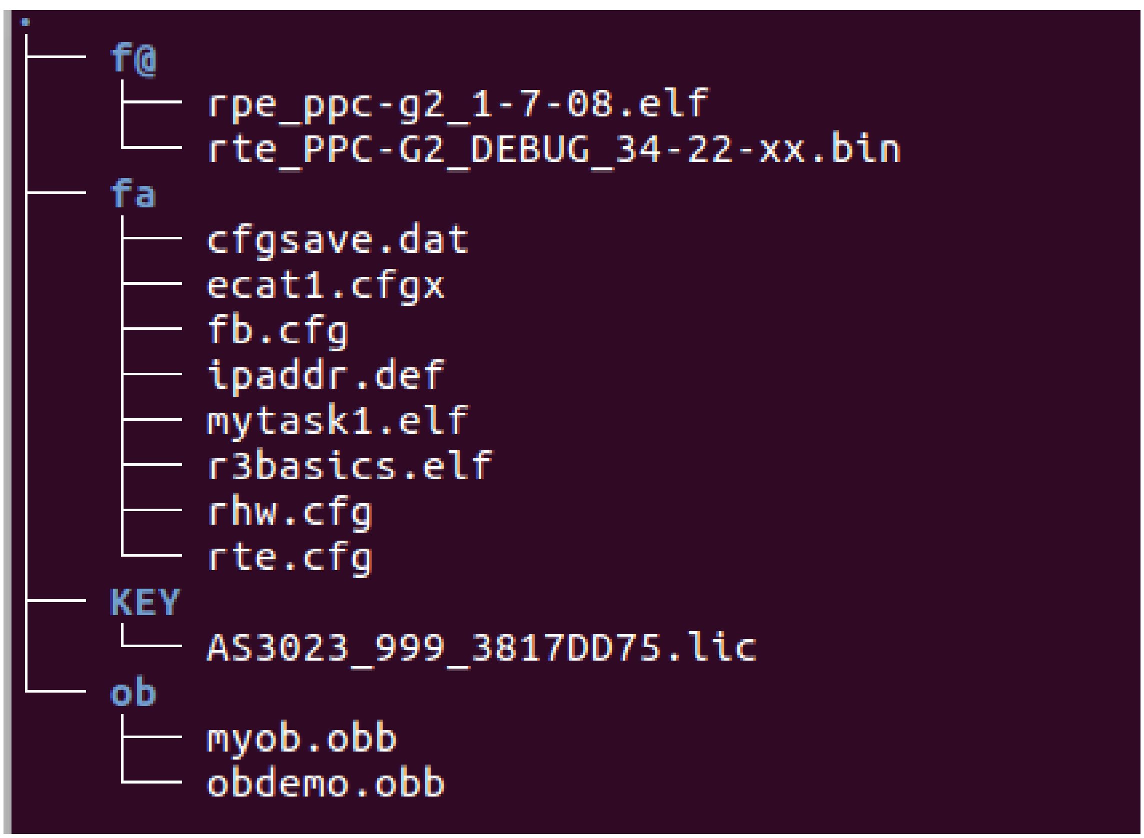


Fig. 9: Memory card default folders tree

Note: An RTE project can have until eight R3 programs as TASKS and one R3 program as RULE. This doesn't mean we have one rule.

```
Demo used in this chapter
```

RDE basics

Flash image

After the creation of a new project, the memory card should be prepared with the right folders and files, *Memory card default folders tree*. The RTE binary file can be downloaded from Robox website. Project files and programs can be created from RDE.

Mainly it is enough to have on the memory card the hardware configuration file and the ip address file, in order to connect to the controller from RDE. Anyway, when a new project is created is more conveniente to create the `project image` from RDE and copy them to the memory card.

Connect the memory card to your computer using a flash card reader if using RP1 or a microSd card reader if using RP2. And copy the generated folders (the image) to the memory card following the procedure showed in the animation.

The following animation show the procedure to prepare the first project image.

Fig. 10: First setup
Prepare the flash card the first time

The image is generated in any folder that you want, but it is better to create a folder in the working workspace called `CF_image` and generate the image files in it. When the image files are generated, they will be copied into the memory card.

Note: Don't copy the or replace the /f@ folder. The f@ should contain the necessary files.

When the memory card contains all the necessary files and folders, any modification to the project can be downloaded to the controller directly from RDE, and the controller is ready to communicate with RDE. Plug the ethernet cable on the second ethernet port of e.g. RP1.

Tools

We already see how to create a new project, create a connection to the controller, and copy the main files and folders to the memory card. In order to see if the controller can communicate with the computer we can use the windows command `ping` or linux command `nmap`.

Our goal is to use RDE, so we will see how to use RDE tools in order to connect to the controller. RDE have different tools: console (like linux terminal or windows prompt), oscillosope, graphic panels, etc.

The scope of this section is to show how to use tools to monitor variables. Don't care now about R3 syntax, even if the code should be clear if you know another programming language. Keep in mind that we will monitor some variables which value is changing.

Connect the controller to the computer via an ehternet cable, turn it on (give power) and be sure that your computer have the same ip class address. If the controller have e.g. 192.168.1.130 ip address the computer should have 192.168.1.xxx where `xxx` is a number different from the ip address of the controller, in this case 130.

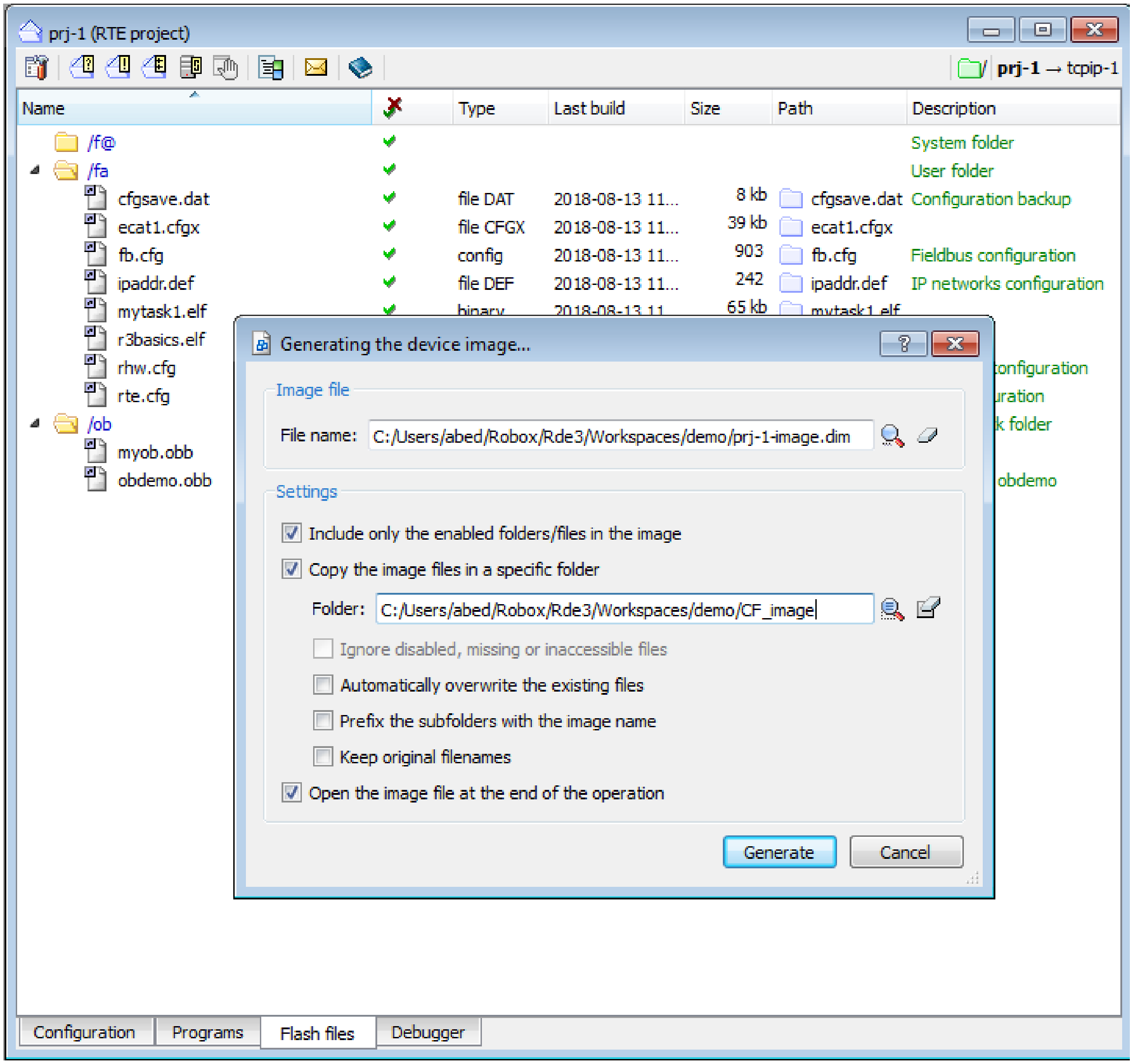


Fig. 11: Flash image files

Console

The following animation show how we can *create a console*, connect it to the controller and use some commands.

Notice that the connection name is the project, not the connection to controller. It is conveniente to connect the project to controller connection, and all other tools to the project.

Fig. 12: create a console

The project can be compiled and download to the controller, using the button `Make project` or `Rebuild project`. You can select wich folders you want to download to the controller. Usually we download the mofied folders.

Note: Don't select f@ folder when building the project.

Custom `console commands` can be created using Robox X-script language.

Variable monitor

In the following animation we will modify the R3 program in order to create a **one second timer**, then build and download the project to the controller. We will create also a variable monitor to show the value of the timer variable.

Fig. 13: Modify program and download to controller. Create a variable monitor.

Graphic panel

The next modification we will create a graphic panel with one button and one textbox. The button will be related to a variable or register to stop or run the timer. The textbox is used to show the value of the timer.

Fig. 14: graphic panel
Create a graphic panel (HMI)

Oscilloscope

In order to illustare the use of an oscilloscope we modify the R3 program in order to generate a sinusoidal wave.

3D graphic panel

Create a *3D graphic panel*, where we will show a box that move along the `y` axis in an alternate motion.

3D graphic panels can be cutomized using X-script language, see X-script chapter for more informations.

Fig. 15: Oscilloscope
Create a sinusoidal wave and monitor it in an oscillosope.

Fig. 16: 3D graphic panel

Important folders and files

In the memory card are present a lot of files and folders. Some of them are important to know what they are, others no. In the documentation of RDE you can find an explanation of those files.

RDE will generate automaticcaly files in the default folders. So for now don't care too much about them. First confidence with the use of RDE should be gained, then advanced concepts will be invetigated.

Tools

From the workspace we can access the tools provided by `RDE`. Different kinds of tools are provided to debug and monitor the software: panels, oscilloscope, variable monitors and command shell, etc. Some of them we have already see.

In the following image we can see some tool created and present in the workspace. We can notice that this workspace have two projects.

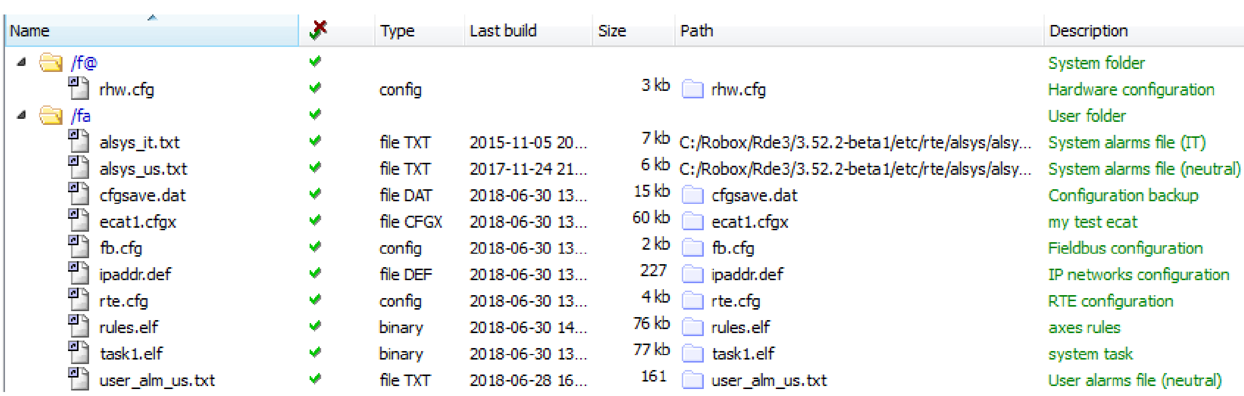
Connections

In order to connect a project or a tool to a controller, we need to create a connection. If we connect to the controller via ethernet cable, as we already see previously, we need to create a tcp/ip connection with the ip address of the controller.

If we connect to the controller via serial cable, we need to create a serial connection.

Command shell

The shell allow to interact with the controller via shell commands and device commands. The most important commands for debugging are `sysinfo` to get information about the controller, `als` to get the list of alarms in the stack and `mreport` to get a report about the activities of the controller, the result is a log menu that can be exported to text file..



Name		Type	Last build	Size	Path	Description
/f@						System folder
rhw.cfg		config		3 kb	rhw.cfg	Hardware configuration
/fa						User folder
alsys_it.txt		file TXT	2015-11-05 20...	7 kb	C:/Robox/Rde3/3.52.2-beta1/etc/rte/alsys/alsy...	System alarms file (IT)
alsys_us.txt		file TXT	2017-11-24 21...	6 kb	C:/Robox/Rde3/3.52.2-beta1/etc/rte/alsys/alsy...	System alarms file (neutral)
cfgsave.dat		file DAT	2018-06-30 13...	15 kb	cfgsave.dat	Configuration backup
ecat1.cfgx		file CFGX	2018-06-30 13...	60 kb	ecat1.cfgx	my test ecat
fb.cfg		config	2018-06-30 13...	2 kb	fb.cfg	Fieldbus configuration
ipaddr.def		file DEF	2018-06-30 13...	227	ipaddr.def	IP networks configuration
rte.cfg		config	2018-06-30 13...	4 kb	rte.cfg	RTE configuration
rules.elf		binary	2018-06-30 14...	76 kb	rules.elf	axes rules
task1.elf		binary	2018-06-30 13...	77 kb	task1.elf	system task
user_alm_us.txt		file TXT	2018-06-28 16...	161	user_alm_us.txt	User alarms file (neutral)

Fig. 17: Files in flash. Auto generated files

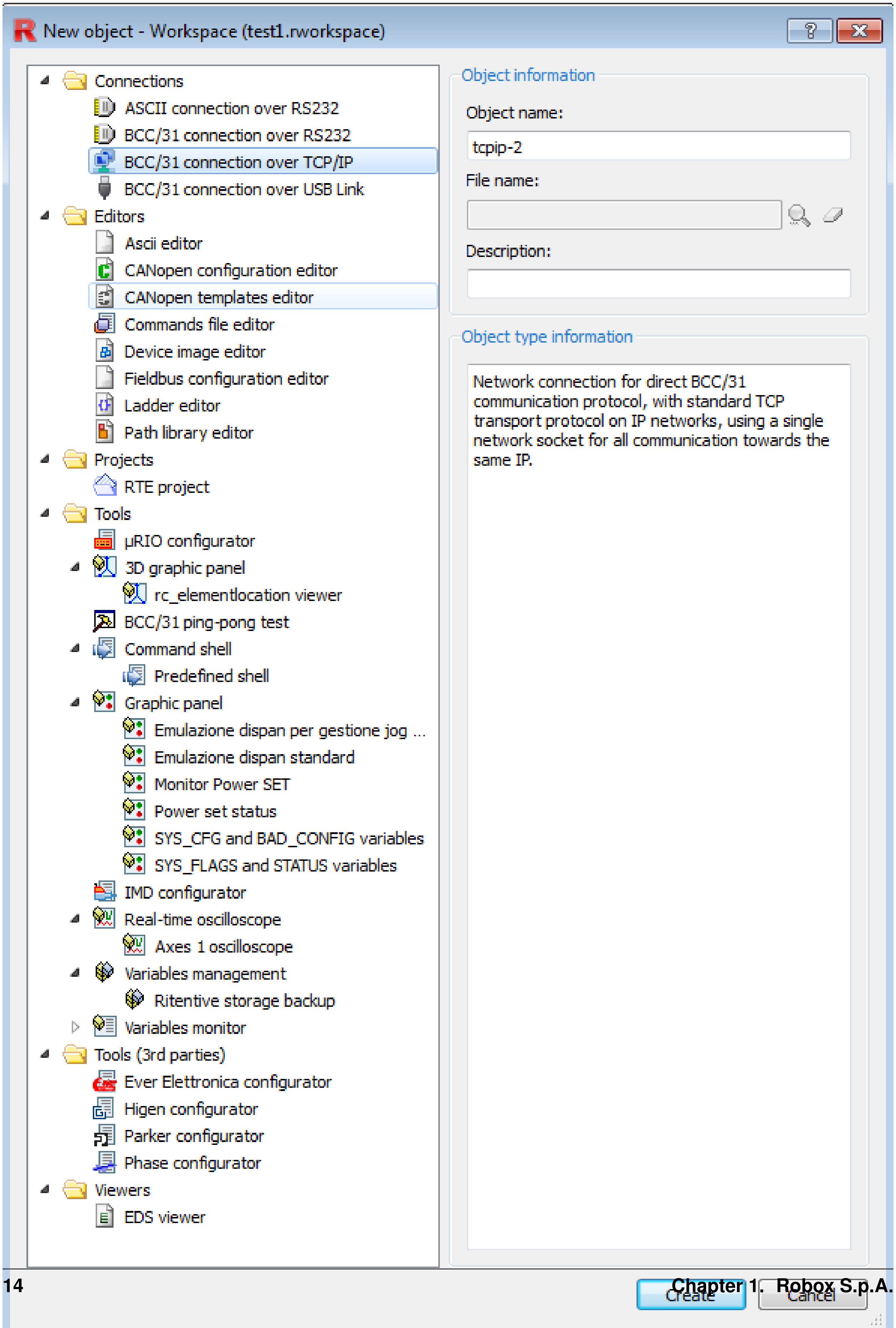


Fig. 18: Objects available in a workspace

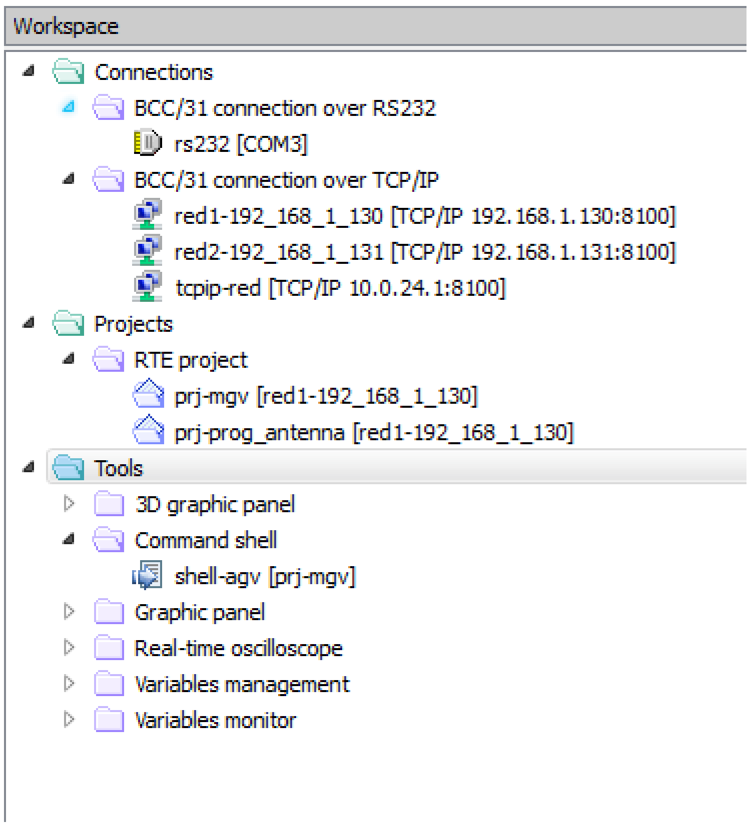


Fig. 19: Some tools in the workspace

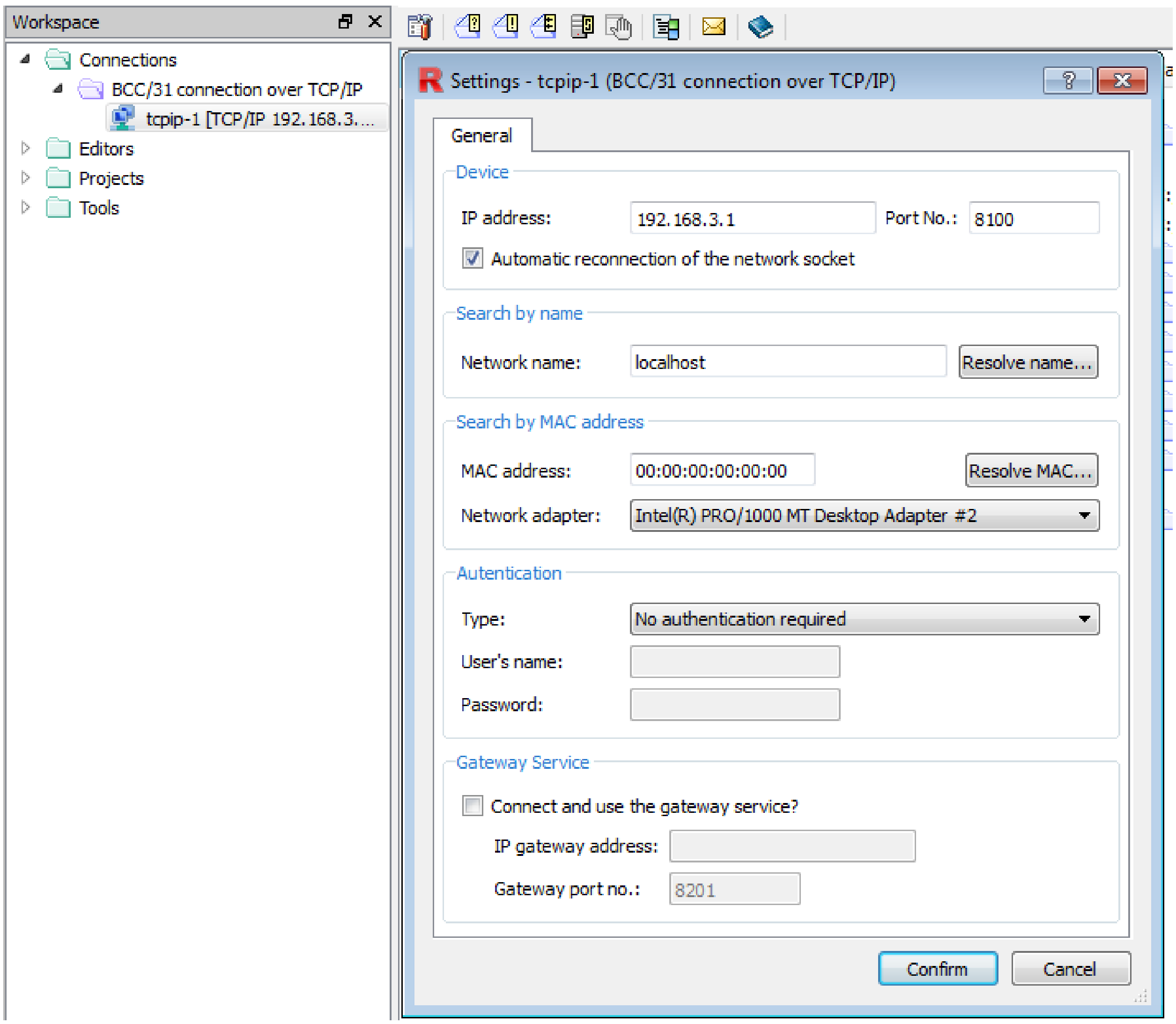


Fig. 20: Tcp connection

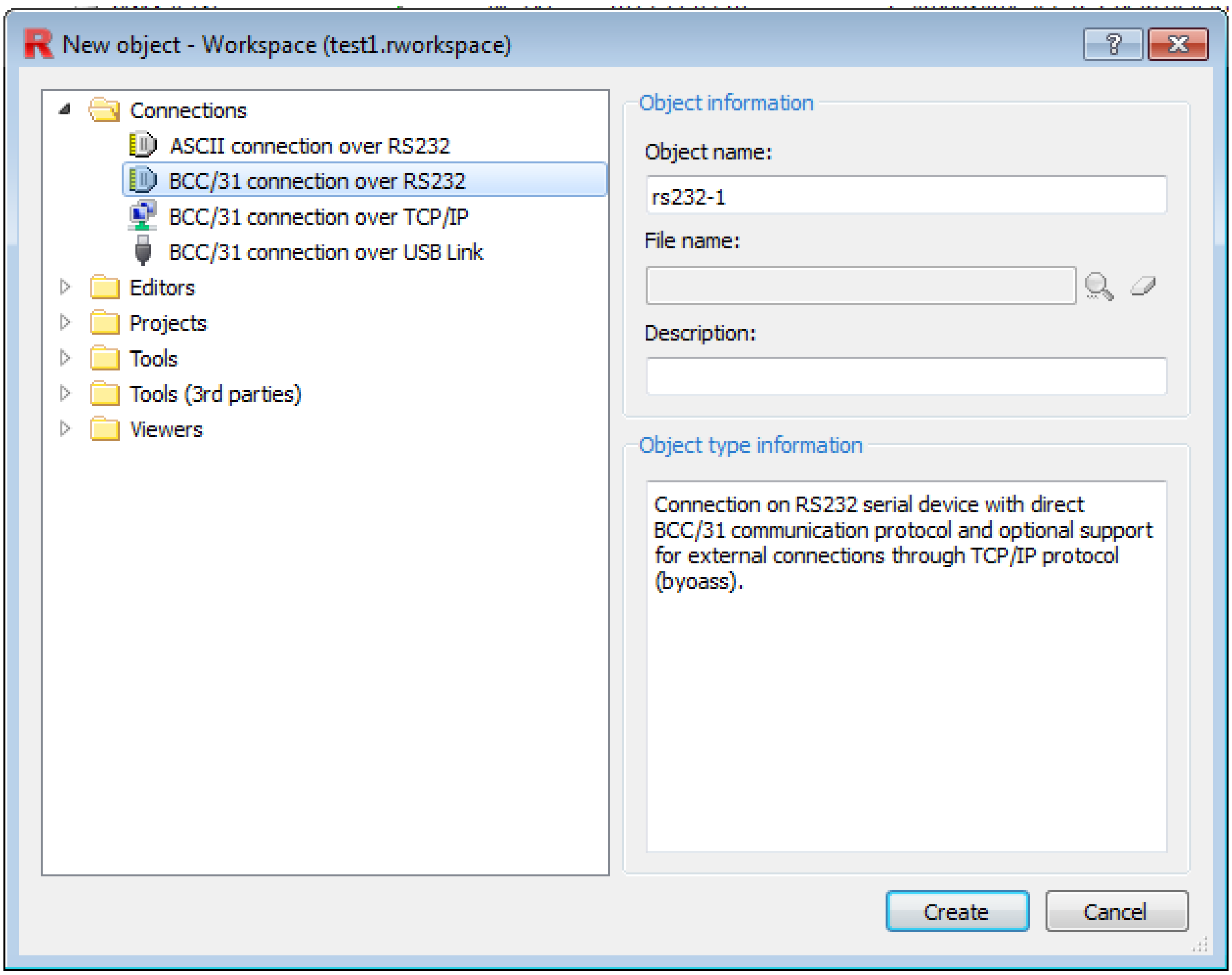


Fig. 21: New object. Serial connection

We can make shortcuts to the most used commands. Click the mouse right button and go to `set quick commands` in order to define shortcuts. A list of defined shortcuts is available from the function keys `[F1-F12]` and from the action menu accessible from the mouse right click.

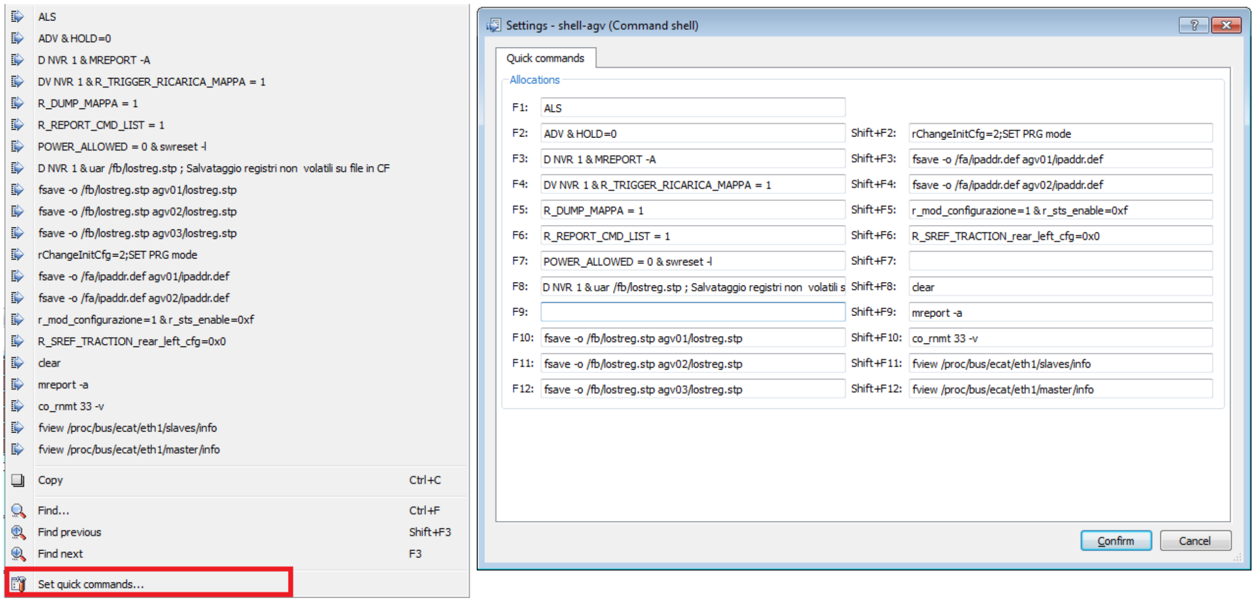


Fig. 22: Command shell: Quick commands

There are different types of commands, some types to manage variables others to manage the flash card other the device. A list of commands is available in the official documentation.

We will see some of the most used commands divided by category. Several commands can be used alone or with options. More than one command can be sent together by using the `\&` operator. Take a look at *Command shell: Quick commands* in order to see the usage and syntax of some commands.

Variable management

- `DV`: Display variable value. The dv command allow us to monitor the value of variables e.g. `dv nvr 1` display the value of the register `nvr(1)`.
- `SV`: Set variable value
- `FV`: Force variable value
- `RV`: Release variable value

Device management

- `adv` Resets the device alarm
- `sysinfo` Get information on connected device.
- `mreport` It displays the events log. the option `-a` display all reports. Other options are available in order to filter the report.
- `als` It displays the contents of the alarms stack.
- `swreset` Request for software reset.

- `uar` Opens a file present in the flash card and refreshes the assignments to R, NVR, RR, NVRR, SR and NVSR with the current values but leaves the comment lines unchanged.

Flash management

- `fsave` Save file from flash.
- `fview` view a file from the flash.

Example of use

- `nvr 1 5` Set the value of nvr register 1 to 5, equivalent to `sv nvr 1 5`
- `nvr 4.2 1` Set the bit 2 of nvr register 4 to 1
- `d inp_w 100`
- `d inp 1`
- `d nvr 1`
- `d nvr 2.3`
- `d nvr 1 5` Displays 5 registers starting from 1
- `d nvr 1 5 -v` Displays 5 registers starting from 1 with their index
- `f_inp 300` Force logical state of input 300
- `uar /fb/lostreg.stp` Save the value of register in the file `lostreg.stp`

Bus configuration

Physical IO (Input-Ouput) are mapped into the memory of the controller, in the so called process image. IO are updated at the begining or the end of the periodic task (Rule). The cycle time is too short, about 5 ms, so it have no importance when it is update. So let's suppose that the controller read the physical input `pin_w` at the begining of the periodic task and write the to the designated memory `inp_w`, and read of the output memory `out_w` and write to the physical output `pout_w`.

In the program the process image or logical IO are used, rarely physical IO are used in a program. IO memory area is an index area represented by 2 big arrays of words (16 bits), one for inputs `inp_w` and one for outputs `out_w`. In IEC 61131-3 these are represented as `%IW` and `%QW`. IO memory can be accessed also by single bit, using the 2 arrays `inp(bit_index)` and `out(bit_index).`

The indexes begin from `1` NOT from `0`, e.g. input word 2 is `inp_w(2)` and the first bit of the word is `inp_w(2).0` the correspond to `inp(17).`

Axioline

Robox controllers support natively Phoenix Axioline bus.

In the following animation we will add to the harware configuration one Phoenix Digital IO module (8 Digital inputs and 8 Digital outputs) and one analog module (2 analog inputs and 2 analog outputs) as shown in the previous pictures..

In the animation we choose automatic memory addressing, we can find the addresses in the flash file `rhw.cfg` :

Fig. 23: RP1 and Phoenix Axioline IO

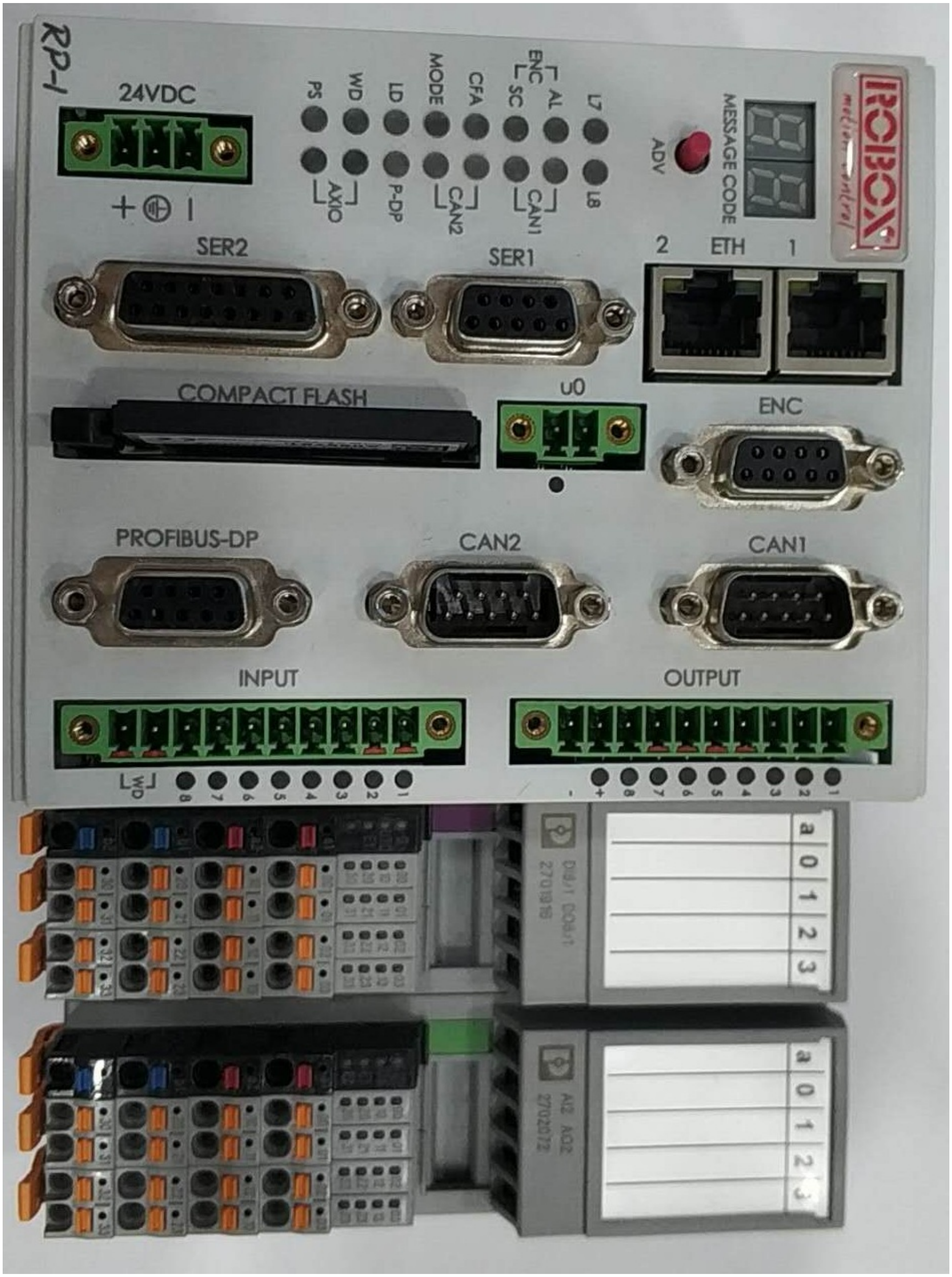


Fig. 24: RP1 and Phoenix Axioline one digital IO module and one analog IO module

Fig. 25: Axioline configuration. Add one Pheonix Digital IO module and one analog IO module

```
IW 36 SLOT 3.01              ; AXL_F_DI8_1_DO8_1_1H (First 8 input group)
IW 37 SLOT 4.01              ; AXL_F_AI2_AO2_1H (Analog input channel 1)
IW 38 SLOT 4.02              ; AXL_F_AI2_AO2_1H (Analog input channel 2)

OW 36 SLOT 3.01              ; AXL_F_DI8_1_DO8_1_1H (First 8 output group)
OW 37 SLOT 4.01              ; AXL_F_AI2_AO2_1H (Analog output channel 1)
OW 38 SLOT 4.02              ; AXL_F_AI2_AO2_1H (Analog output channel 2)
```

The first physical input can be read on the address `inp_w(36).0` and the the first digital output can be written to `out_w(36).0`. We have also 2 analog inputs and 2 analog outputs. We can read the value of the first analog input from the address `inp_w(37)` e.g. `rawTemperatura = inp_w(37)` and write to the second analog output in this way e.g. `out_w(38) = rawSpeed`.

Ethercat

In this section we show how to create an Ethercat bus configuration file. We will use Wago Ethercat modules.

Fig. 26: Ethercat configuration. Wago ethercat modules, one 16DI and one 16DO

After the creation of the Ethercat configuration with a bus coupler and 2 IO modules, we need to configure the input-output variables as shown:

Fig. 27: Ethercat gloval variable configuration

In this configuration we will assign manually IO addresses. We have one 16 digital input Wago module and one 16 digital output wago module. Even if each module is 16 bits, wago map each 8 bit on a word. So will have one 2 words for each module. As the animation, we assign the first 8 inputs of the module the address 300 and the first 8 outputs the address 300. Then we proceed incrementally. So the first 16DI module will be mapped to `inp_w(300)` and `inp_w(301)`.

CanOpen

Profibus

1.1.3 R3

Overview

R3 is similar to IEC 61131-3 ST language (Strucuted Text). It is used by Robox controllers. The syntax is simlar to C, Pascal and basic languages.

Note: R3 language is **NOT** case sensitive.

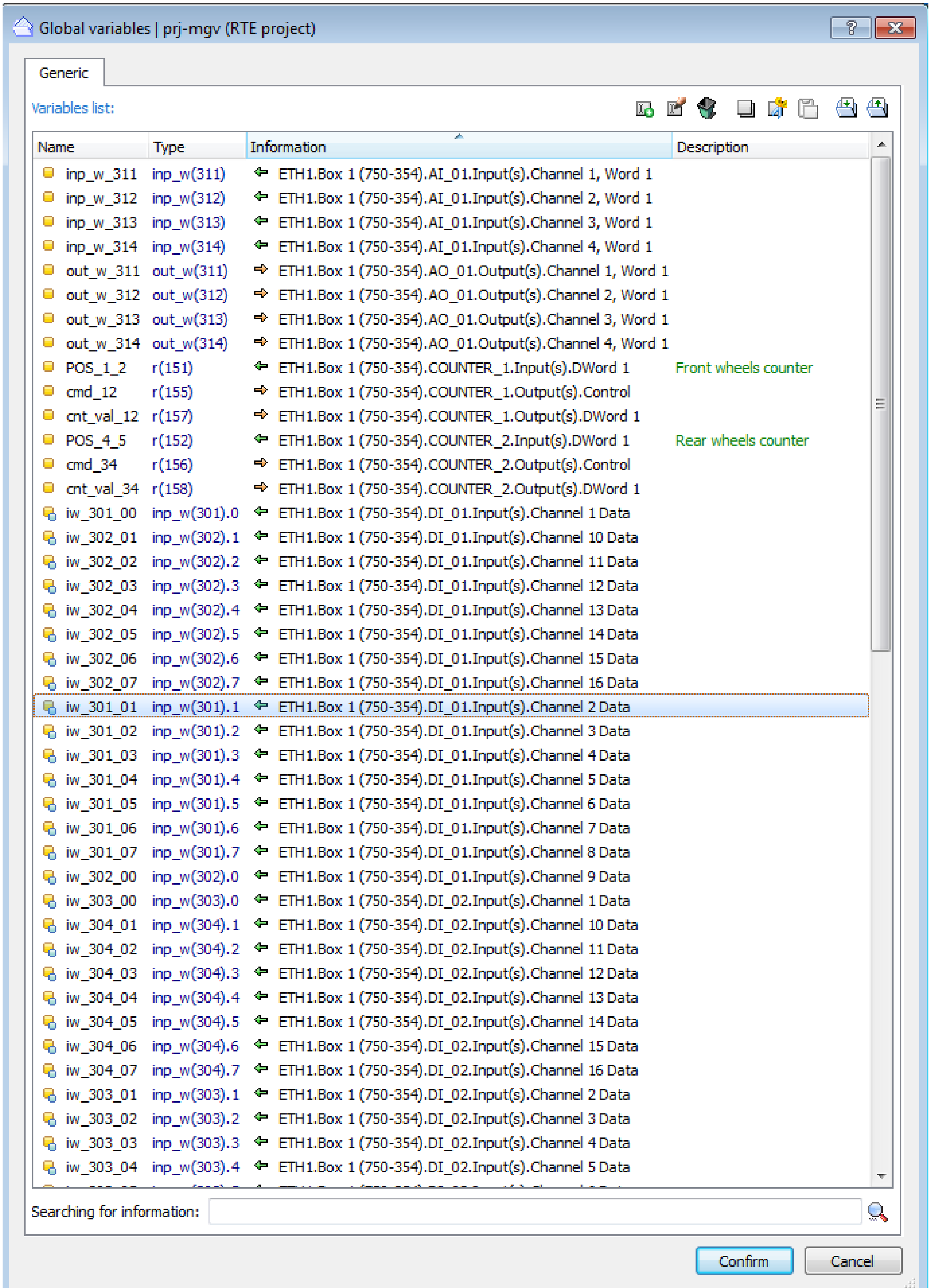


Fig. 28: Global variable example

Program structure

An R3 program can be a task or a rule file. The code of a program can be written in one file or divided in several files that can be included in R3 program using the keyword `$include` followed by the file name. Usually the file to be included have `.i3` extension.

The rule case is a little bit different, and will discussed later. A task is like a C prrogram, is executed from the begining until the end. If no infinite loop is used, once the task reach the end, its execution is terminated. Usually a task have to be executed cyclically, for this reason after the initialisation, the code is written inside an infinite loop i.e. ‘‘ while(1) { code }‘‘. R3 provide `__MAIN_LOOP__` block that is equivalent to the infinite loop.

```
$include filetobeincluded.i3

; variable declaration

; initialization

__MAIN_LOOP__
  ; this is an infinite loop
  ; write your code here

END_MAIN_LOOP
```

Basic syntax

Variables and types

Fundamental types :

```
bool

I8
I16
I32
INT

U8
U16
U32

float
real

char
string
```

Some strucutre or complex types:

```
STRUCT
STRUCTP
TIMER
COUNTER
```

Some example of motion related types

```
STRU_MVTO
STRU_CAM
```

Here some example on how to use. The syntax id the same for all types `type varname`.

```
INT intVar
REAL position

INT vv[10]
REAL vr [5][2]

STRING description
STIRNG desc[5]

intVar = 10
position = 20.0
```

We can access the single be of a varible by using the dot operator, e.g. we want to access bit 5 of the varible pippo:

```
int pippo = 0x20
pippo.5 = 1 ; assign value 1 to bit 5 of the varible pippo
```

Constants

Usually we create constants, to avoid to use numbers, to make our program more readable. Constants are created suing the keyword `LIT`, e.g. `LIT MONDAY 1`, `LIT TUESDAY 2`. As a convention constants are written in capital letters.

R3 doesn't have the enumeration type, pay attention when constructing enumeration with constants to keep different numerical values to different costants of the same category, e.g. `MONDAY` and `SUNDAY` should have different numerical values, they can't have both of them the value `1`. If they have the same value, this will be a programming error, not a syntax error.

Operators

```
; assignment
=


;
+ - * /

;
AND OR NOT XOR

;
> < =
<> >= <=

; bitwise
R_AND R_NOT R_OR R_XOR

; string concatanation
#
```

Control flow

As any programming language, usual control flow statements are :

```
if (contition)
  ;
elseif (condition)
  ;
else
  ;
end_if

_if (condition)
  ; one statement
_else
  ; one statement
```

```
for(initialisation, conidtion , update)
  ;
end_for

for( i=0, i < 10, i=i+1)
  ; code
end_for

for(,1,)
  ; infinite loop
end_for
```

```
select (var)
  case 1
    ; code
    break

  case 2
    ; code
    break

  default
    ; code

end_select
```

```
while(condition)
  ;
end_while

while (i < 10)
  i=i+1
end_while

while (1)
  ; inifite loop
end_while
```

```
do
  ;
end_do_while(condition)
```

```
__main_loop__
  ; infinite loop

end_main_loop
```

```
I32 cond_val (condition, I32 val_if_true, I32 val_if_false)

i = cond_val (b=2, 10, 20)

; this equivalent to
if (b=2)
 i = 10
else
 i = 20
end_if
```

In the documention and in the example shown before can be found their syntax.

ALIAS

An **alias** is a more undertandable or more clear alternative to a varible or to a function. In R3 can be used to give a name to a register or to an input or a memory. The keyword `LIT` is used, like for constants. For example in an R3 we can write `r(3) = 100`, it is correct be the meaning of `r(3)` is not clear.

If we writte:

```
LTI Position r(3)
position = 100
```

it will be clear that the varible we are dealing with, is a position. We can give different alias to the same register.

Let's suppose that `r(10)` is a mask where every bit represent something. We can use the dot operator to access the singular bits. e.g. `r(10).4`. Of course it will more clear if we give a name to number `4`.

```
LIT DriveStatusWord r(10)
LIT DRIVE_READY 0
LIT DRIVE_RUN   1
LIT DRIVE_ALARM 4

if  ( DriveStatusWord.DRIVE_ALARM )
  ; do something
end_if
```

Data structure

Data strucutres could be Arrays, Sruct and OBs. In R3 documentation we can find predefined structures and OB that main are related to motion control.

We can also define our own structures and OBs.

Modular programming

Tasks, functions and Object blocks can be used to make the program modular and esay to debug.

RDE allow us to create maximum 9 R3 programs (files) divided in one Rule program (one file) and eight taks (8 files). It allow also to create other files that can be included in tasks and rule files. We can write our functions, variable delcation, IO mapping, registers aliases (using LIT) in different files, usually with extension `.i3` and include them in the disered task using the keyword `$include filename.i3`

Scope rules

As any programming language variables have a scope. They could be local or global variables. Registers, IOs and predefined variables are global, and they can be written and read from any task. Also variables that are aliases to registers and IO are global.

Variables could be local to a function or local to a task.

```
$task 1

int b  ; local to this task, it can't be seen by other tasks
int c  ; local to this task
int i ; local to task

__MAIN_LOOP__

 ; code

 val =2  ; it will give compilation error. this is not delcared in the task

END_MAIN_LOOP

function

 int val ; local to funtion
 int i ; local to funtion, it is not the same as the one delcared in the TASK

 b = 2 ; this is delcared in the task, it can be used also in the funtion.

 ; code
end_fun
```

Variables could be also public and can be shared between tasks. If a variable is delared as `public` in task 1, and `extern` in task 2, it can be written and read in task 1, and only read in task 2.

```
 $task 1
  public int val  ; public variable. can be read an written by this task. I can be only read in other tasks where the keyword extern is used.

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 $task 2
  extern int val  ; external variable can only be read

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

$task 3
  int val ; this is local to task 3
```

I you want to read and write a variable delcared as `extern`, the keyword `$WRITE_ON_EXTERN` should be add to the task where the variable is delared as `extern`

```
$task 1
 public int val  ; public variable. can be read an written by this task. I can be␣
↪only read in other tasks where the keyword extern is used.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

$task 2

 $WRITE_ON_EXTERN  ; if this keyword is present all varibles delared as extern become␣
↪also writable by this task
 extern int val    ; external variable can only be read
```

Example

The purpose of the following code is to illustrate the syntax of R3. The whole code have no meaning by itself.

```
$TASK 2

$include incfile.i3

; this is a comment

; STRUCT definition
STRUCT stPoint
    REAL x
    REAL y
        REAL z
        INT n
END_STRUCT

; variable of type stPoint
stPoint myPoint1
stPoint myPoint2

int a         ; 32 bit signed variable
a=2                 ; variable initilization

int n

real time        ; 8 bytes floating point

real tim

bool c                ; bool variable

; LIT keyword used as alias to registers, inputs and outputs
LIT sinf rr(1)
LIT inpValve        inp_w(200)

; LIT can be used also to define constants
lit THIS_IS_CONSTANT        2

time = tfb
```

(continues on next page)

(continued from previous page)

```
tim =tfb

LIT operation r(10)
operation = 0

; array of 5 int
int arrBuffer[5]


; infinite loop
__MAIN_LOOP__

        if (tfb > time +1)
                _if ( r(2).0 )
                        r(1) = r(1) + 1
                time = tfb
        end_if

        if (tfb > tim +0.005)
                sinf = sin(2*3.14/2 * tim)
                tim = tfb
        end_if

        ; Object block use
        obdemoist.b =true
        c= obdemoist.readonlyvar

        ; call a function
        call thisIsFunction()

        if ( a > 10 )
                n = 100
        elseif ( a < 5 AND a > 0)
                n = 10
        else
                n = -1
        end_if

        int i
        for (i = 0, i < 22, i=i+2)
                n = n + i
                if (n > 10)
                        continue
                elseif ( n= 100)
                        break
                end_if
        end_for

        i =0
        while (i < 10)
                i=i+1
                n= i +2
                _if (n = 10) ; this _if have only one instruction that belong to it
                        break
        end_while

        real distance
```

(continues on next page)

(continued from previous page)

```
        distance = getDistance(myPoint1, myPoint2)


END_MAIN_LOOP

function thisIsFunction()
        ; string concatenation
        sr(1) = "it's" # " eight"
end_fun

; funtion
function testFunc()

        select (operation)
                case 0
                        ; do something
                        break

                case 1
                        ; do somethingelse
                        break
                default
                        ; do somethingelseelse
                        operation = 0
        endselect

end_fun

; this function return a real value
function real getDistance(stPoint p1, stPoint p2)
        ; euclidean distance
        return sqrt( pow((p2.x -p1.x),2) +  pow((p2.y -p1.y),2) + pow((p2.z -p1.z),2)␣
↪)
end_fun
```

```
Basic syntax of R3 language
```

Predefined variables

A full list of the **predefined variables** can be found in **Documentaion –> Programming languages –> R3 language –> Predefined variables**

Input-Output

Registers

Regisers are arrays of prealloced memories. The dimension can be defined by the user, *Register dimension*.

Register dimension show different types of registers and their allocation in memory.

RESOURCES Input/Output	#	Read/Write	Retentive	Default	U.M.	Bit access Permission	Description
inp		R	No	0	bool	-	Input digital channel
out		RW	No	0	bool	-	Output digital channel
inp_w		R	No	0	msk	Yes	Input word (16 bit)
out_w		RW	No	0	msk	Yes	Output word (16 bit)
pinp		RW	No	0	bool	-	Physical input channel state (the state can NOT be forced)
pout		RW	No	0	bool	-	Physical output channel state (the state can be forced)
pinp_w		RW	No	0	msk	Yes	Physical input word state (the state can NOT be forced)
pout_w		RW	No	0	msk	Yes	Physical output word state (the state can be forced)

Fig. 29: IO predefined variables

GLOBAL REGISTERS	#	Read/Write	Retentive	Default	U.M.	Bit access Permission	Description
r		RW	No	0	k	Yes	32-bit integer register
rr		RW	No	0	k	No	64-bit real register
sr		RW	No	0	k	No	128 byte string register
nvr		RW	Yes	0	k	Yes	32-bit retentive integer register
nvrr		RW	Yes	0	k	No	64-bit retentive real register
nvsr		RW	Yes	0	k	No	128-byte retentive string register
als		R	No	0	k	No	Alarm stack
aln		R	No	0	k	No	Alarm stack
am		R	No	0	k	Yes	Alarm mask
p_ip		R	No	0	k	No	IP buffer of the previous counts
p_iv		R	No	0	k	No	IV buffer of the previous counts

Fig. 30: Registers predefined variables

Axis parameters

The following variables are arrays of 32 elements. The array index correspond to the axis index. For example `cp(2)` is the current position of `Axis number 2`.

- `kbit2unit` Bit-unit conversion factor.
- `cp` Axis current position [unit]
- `cv` Axis current velocity [unit/s]
- `ca` Axis current acceleration [unit/s^2]
- `ip` Axis ideal position [unit]
- `iv` Axis ideal velocity [unit/s]
- `ia` Axis ideal acceleration [unit/s^2]
- `sref` speed reference.
- `pro_gai` position loop proportional gain
- `kff` feed forward factor
- `epos` position error when the position loops are closed with a predefined formula
- `fr` feed rate. This variable contains a factor ranges from 0 to 1. If a min value of 0 is programmed, the variable fr will be set = 0. If a value >1 is programmed, the variable fr will be set = 1.

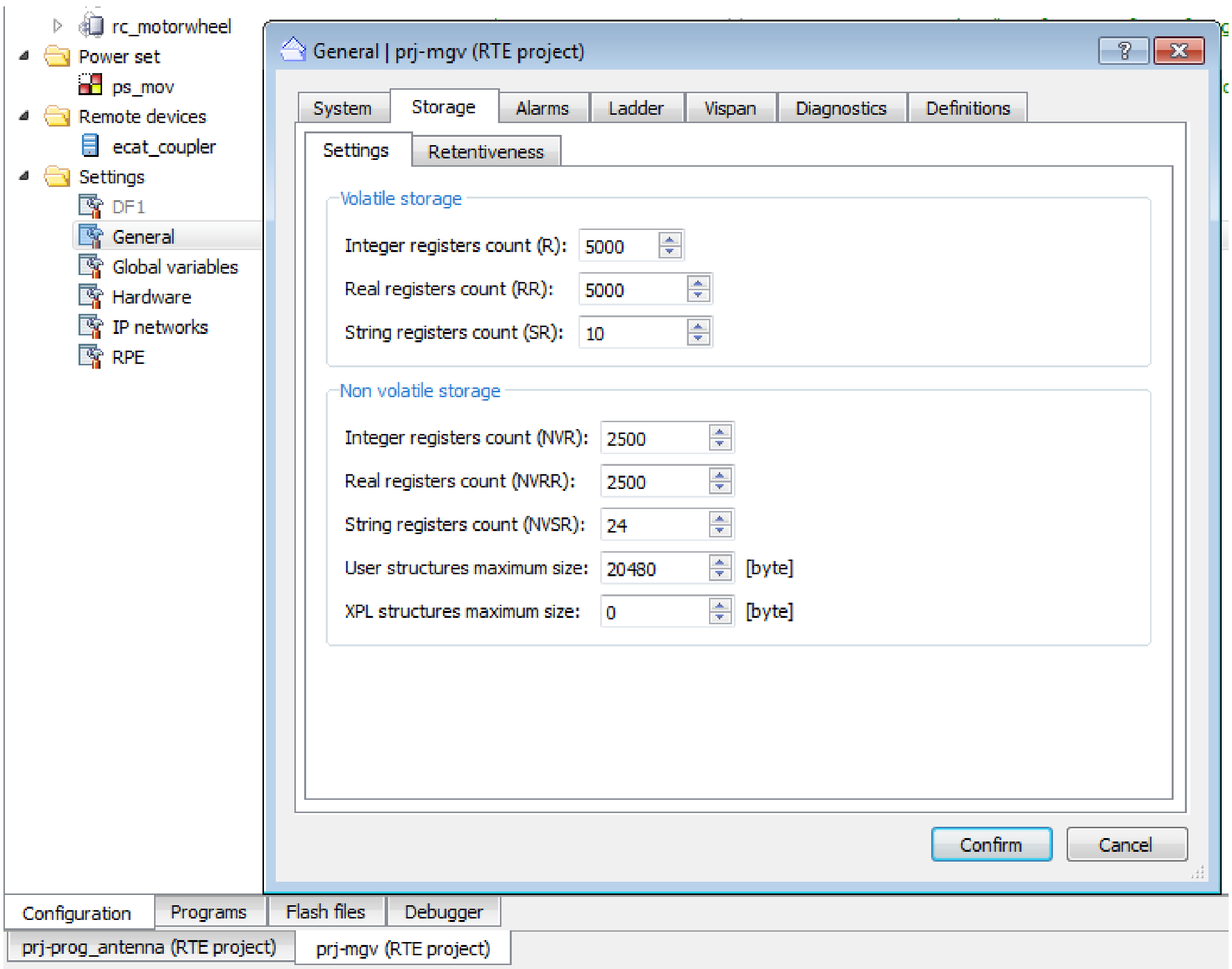


Fig. 31: Register dimension

1.1.4 R3 demos

All `demos:` of this chapter can be found in one workspace and one project.

Demo 1: Analog input

Le't connect an analog temperature sensor to the analog module of Pheonix. Check RDE chapter for information about how to configure IOs. The temperature sensor have linear relationship between the tension (V) and the temperature. The analog input have an internal 16 bit ADC (Analog to Digital Converter). The data type of the converted value is 16 bit (15bit + sign) Tension value is mapped from `[0V;10V]` to `[0,30000]`. Usually the max value `7FFF` is more than `10V`.

The linear relationship between the signal and its phycical value is represented as:

$m = \frac{(y_1 - y_0)}{(x_1 - x_0)}$

$y = m(x - x_0) + y_0$

In our case the `y` will be the temperature and the `x` will be the digitalized value. To build a linear relationship we need two point (x_0, y_0) and (x_1, y_1). From the datasheet of the temperatura sensor we obtain the curve of the sensor.

Let's suppose that:

- 0V (AI=0) is $0°C$
- 10V (AI=30000) is $100°C$

`The following is the code implemennted in R3:`

```
$TASK 1

; Analog sensor connected to first analog channel of pheonix module
LIT temperaturaAI inp_w(36)        ; Temperatura analog input

; temperature values are saved in non volatile real regisers
LIT temp0                nvrr(1)        ; temperature
LIT temp1                nvrr(2)        ; temperature
; scaling values. saved in non volatile integer registers
LIT temp0_ai        nvr(3)        ; analog value corresponding to temp0 degree
LIT temp1_ai        nvr(4)        ; analog value corresponding to temp1 degree

temp0 = 0.0
temp1=100.0

temp0_ai = 0
temp1_ai = 30000

real temperature = 0.0

__main_loop__

        ; scaling equation, linear relationship between temperature and analog input
        ; consult the datasheet of the analog module
                ; 0V --> 0x00 (0)
                ; 10V --> 0x7530 (30000)

        ; Temperatura sensor
                ; Range -45degree (1V) ~ 125 degree (10V)
```

(continues on next page)

I/O data		0 V ... 10 V
hex	dec	V
8001	Overrange (output)	+10.837
8001	Overrange (input)	> +10.837
7FFF ... 7F01		+10.837
7F00	32512	+10.837
7530	30000	+10.0
3A98	15000	+5.0
0001	1	+333.33 µV
0000	0	0
FFFF	-1	0
C568	-15000	0
8AD0	-30000	0
8100	-32512	0
80FF ... 8000*	(Output)	Hold last value
8080	Under-range (output)	0
8080	Under-range (input)	0

Fig. 32: From datasheet of Pheonix AI module

(continued from previous page)

```
        temperature = ((temp1-temp0) / (temp1_ai - temp0_ai) ) * (temperaturaAI -
↪temp0_ai) + temp0

        ; short task, we add a waiting instruction
        dwell(0.2)        ; wait for 0.2 seconds
end_main_loop
```

Demo 2 : Using functions

In this section will show how to use functions. We will modify the temperature example, we create `TASK2`. First we create a function that represent a linear relationship between two variables `linearmap()`. Then we will call it in the main loop. In this example we will map the analog input into a register in order to be able to simulate it, as we don't have the phycial sensor.

Note: remmeber to execute `task2` form `task1`, by adding the instruction `mt_en(2)`

Note: We can force the value of `inp_w` in order to debug the program.

`The following is the code implemennted in R3:`

```
$TASK 2

; Analog sensor connected to first analog channel of pheonix module
LIT temperatureAI r(1)        ; Temperatura analog input

; temperature values are saved in non volatile real regisers
LIT temp0                nvrr(1)        ; temperature
LIT temp1                nvrr(2)        ; temperature
; scaling values. saved in non volatile integer registers
LIT temp0_ai        nvr(3)        ; analog value corresponding to temp0 degree
LIT temp1_ai        nvr(4)        ; analog value corresponding to temp1 degree

temp0 = 0.0
temp1=100.0

temp0_ai = 0
temp1_ai = 30000

LIT temperature rr(1)


__main_loop__

        temperature = maplinear(temperatureAI, temp0_ai, temp0, temp1_ai, temp1)
        dwell(0.2)
end_main_loop

function real maplinear(int x, int x0, real y0, int x1, real y1)
        real m = (y1-y0)/(x1-x0)
        return m*(x - x0) + y0
end_fun
```

Demo 3 : Cylinder

In this demo we will illustrate the use of functions, and include files.

Remember that the code of included files, at compilation time are merged with the main file. It means the keyword `$include filename.i3` is replaced by its content.

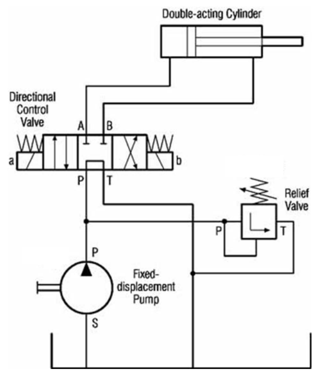


Fig. 33: Hydraulic double acting cyclinder, 3 state electrovalve

Demo 4: State machine

1.1.5 Object block

Object block is a C++ class, it is another option to write program in RDE. An OB is the equivalent of a Funtion Block (FB) in the IEC 61131-3, this mean that an OB have a static memory that is conserved between different calls of the OB. It is dieerent from the concept of a Function (FC).

An OB is composed from a header file (.h) and a source file (.cpp) like like any `C++ class`, in addition to these classic files, `RDE` use the `obs file` to describe the `interface` of the Object block. In the obs file, public fields and methods are defined.

OB

Create a new OB

The following animation, *Object block creation*, show step by step how to create and deploy a new OB. The main steps are shown and explained also in the static images below.

Fig. 34: Object block creation
Create new Object block and an instance of it

In rte project, right click and add *new Object block*. A folder have to be selected for the compiled file, usually `/ob`. If the folder `ob` dosen't exist add it in the flash memory before creating the Oject Block, see section files and folders.

Insert the name of th OB class and the description. The description will be shown in the description colounm in the RTE project. Usually this field is brief. Select the Flash folder, usually `/ob`, where the compiled OB (.obb) will be saved. The check box **Automatic generation** should checked, otherwise not all files will be generated.

In the following image the result of the creation of an OB is shown:

After the creation of a new object block we will obtain 4 files:

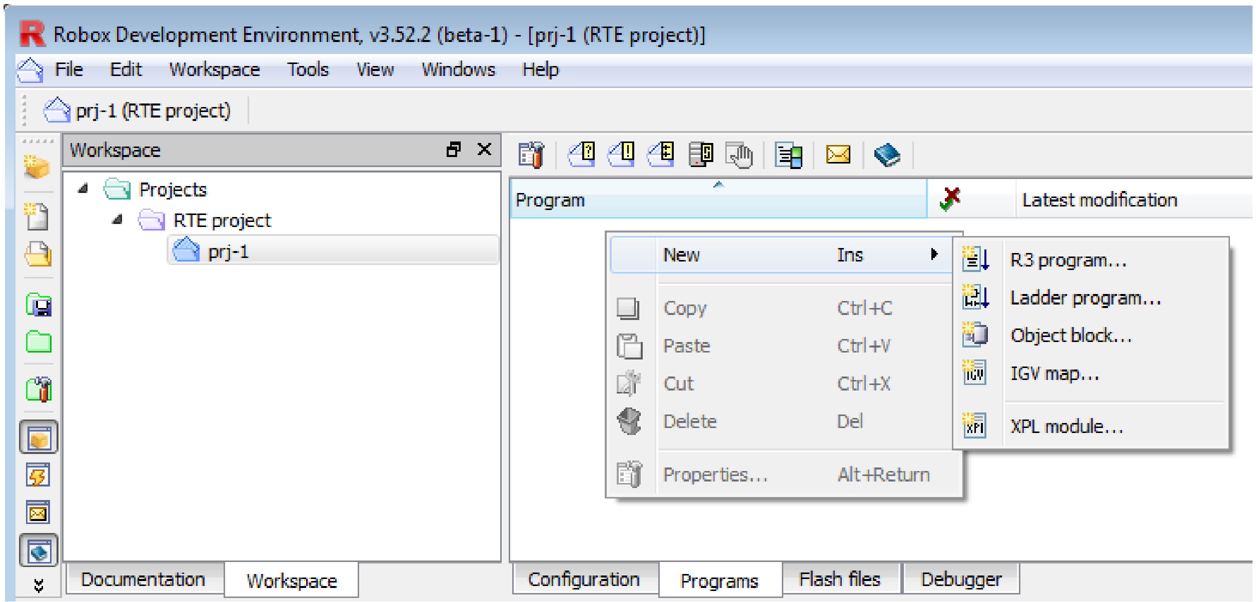


Fig. 35: new Object block
Create new Object block. right click in the tab program of an RTE project

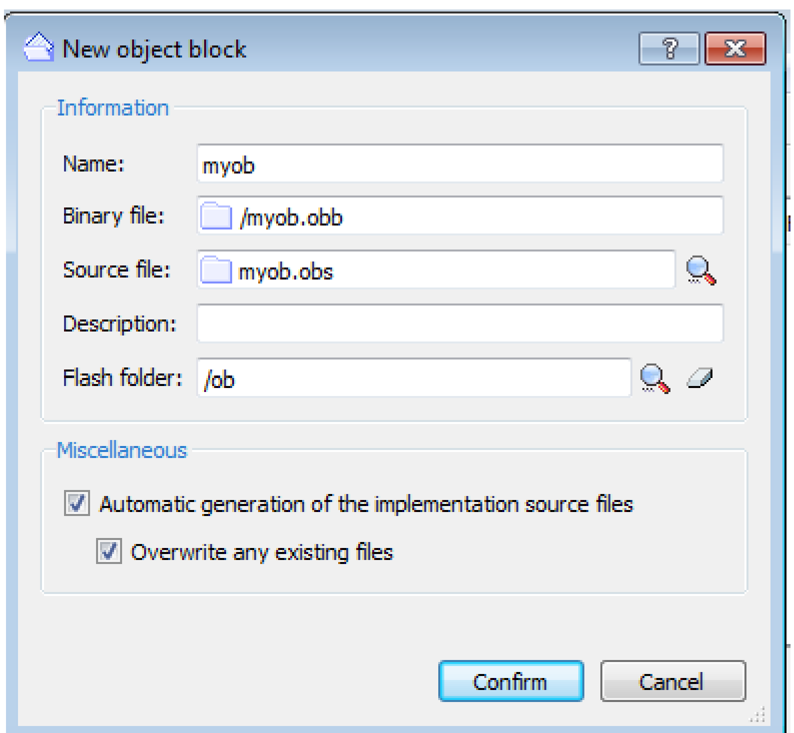


Fig. 36: Write the OB name, select the folder of destination and check at least the first option

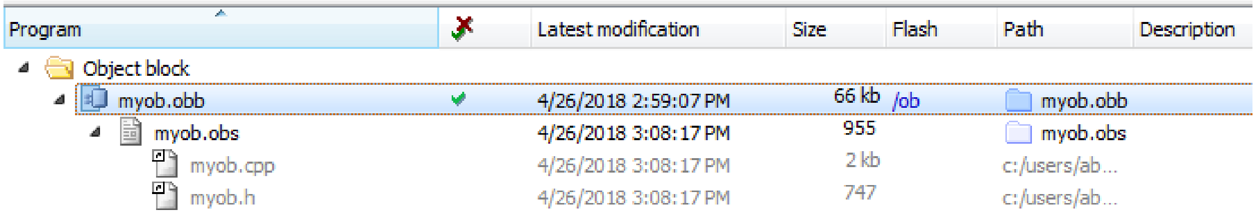


Fig. 37: Object block structure files

- obs : object block interface file
- h : C++ header
- cpp : C++ source
- obb : Object block binary file (compiled file), that can found in the /ob folder in the Flash files.

Fig. *Obs*, *Header* and *Source* show the auto generated files. As we can see the header and the source files have the structure of a classic C++ class with class name, class constructor and destructor.

Deploy an OB

As any object oreinted language, a class have to be instantiated before using it. In the configuration tab of an RTE project, right click Object block and add *OB Class or OB Instance*. A class could have more than one instance. An `OB` is similar to an `FB (Function block)` in `PLC` programming.

OB basics

As any class of an object oriented language, an Object block have `methods` (functions) and `fields` (variables). Public methods and fields that can be accessed from an R3 program should be written in the obs file respectively in the methods and properties blocks.

Properties could be only of simple C++ types: BOOL, I8, I16, I32, U8, U16, U32, INT, FLOAT, REAL, CHAR, could not be of struct type.

Note: Properties name should be lower case, capital letters generates compilation errors.

The source file where the code is implemented is written in the block `implementation`. An OB can be implemented in more than on source file.

When an OB inherit from another OB, and we want to ovveride a property or a method the keyword `virtual` is used in the declaration.

Using an OB in R3

Suppose we have the class `obCylinder` and its instance `cylinder_right`. Let's suppose the OB have the methods `opencyl()` and `closecyl()`, and 2 readonly properties `cyl_opened` and `cyl_closed` and 2 not readonly properties `cmd_open` and `cmd_close`. We can call in R3 the methods as we call them in C++ using the dot operator: `cylinder_right.opencyl()`. We can access properties using also the dot operator for reading or writing: `bool cyl_closed = cylinder_right.cyl_closed` or `if(cylinder_right.cyl_opened)` or `cylinder_right.cmd_open= TRUE` and `cylinder_right.cmd_close = FALSE`.

myob.cpp | myob.obs | myob.h

```
;===========================================================================
;
; ROBOX SpA
; Via Sempione 82, Castelletto Ticino, ITALY
; +390331922086
; http://www.robox.it
;
;---------------------------------------------------------------------------
; Job number     :
; Title          : Class MYOB project
; Platform       : RTE
; Generator      : Robox Development Environment, v3.52.2 (beta-1)
;===========================================================================

define DEBUG_MYOB        ; Enable DEBUG for the class

object_block myob

    ; General object block information
    title
    version 1.0.0
    info

    end_info

    ; Class structures
    structures
    end_structures

    ; Class properties
    properties
        ; Use 'ro' data modifier for read-only properties
        ;     'ba' data modifier for bit access enabled properties
    end_properties

    ; Class methods
    methods
    end_methods

    ; Implementations
    implementation
        source "myob.cpp"
    end_implementation

end_block

```

Fig. 38: Obs
Auto-generated OBS file

myob.cpp | myob.obs | myob.h

```
//================================================================
//
// ROBOX SpA
// Via Sempione 82, Castelletto Ticino, ITALY
// +390331922086
// http://www.robox.it
//
//----------------------------------------------------------------
// Job number   :
// Title        : Class MYOB declaration
// Platform     : RTE
// Generator    : Robox Development Environment, v3.52.2 (beta-1)
//================================================================

#ifndef __MYOB_H__
#define __MYOB_H__

#include <myob_base.h>

/*
 * Class MYOB declaration.
 *
 */
class myob: public myob_base
{
public:
    /* Class constructor */
    myob(rObOptions *opts);

    /* Class destructor */
    virtual ~myob();
};

#endif // __MYOB_H__

```

Fig. 39: Header
Auto-generated C++ header

myob.cpp | myob.obs | myob.h

```cpp
//==================================================================
//
// ROBOX SpA
// Via Sempione 82, Castelletto Ticino, ITALY
// +390331922086
// http://www.robox.it
//
//------------------------------------------------------------------
// Job number    :
// Title         : Class MYOB implementation
// Platform      : RTE
// Generator     : Robox Development Environment, v3.52.2 (beta-1)
//==================================================================

#include <ob/reprintf.h>
#include "myob.h"

OB_FACTORY (myob)
    OB_INSTANCE (myob)
OB_ENDFACTORY

//==================================================================
// myob
//==================================================================

myob::myob(rObOptions *opts): myob_base(opts)
{
#ifdef DEBUG_MYOB
    reprintf("myob %p: created", this);
#endif

    // TODO: initialization code here
}

//------------------------------------------------------------------

myob::~myob()
{
    // TODO: termination code here

#ifdef DEBUG_MYOB
    reprintf("myob %p: destroyed", this);
#endif
}

```

Fig. 40: Source
Auto-generated C++ source

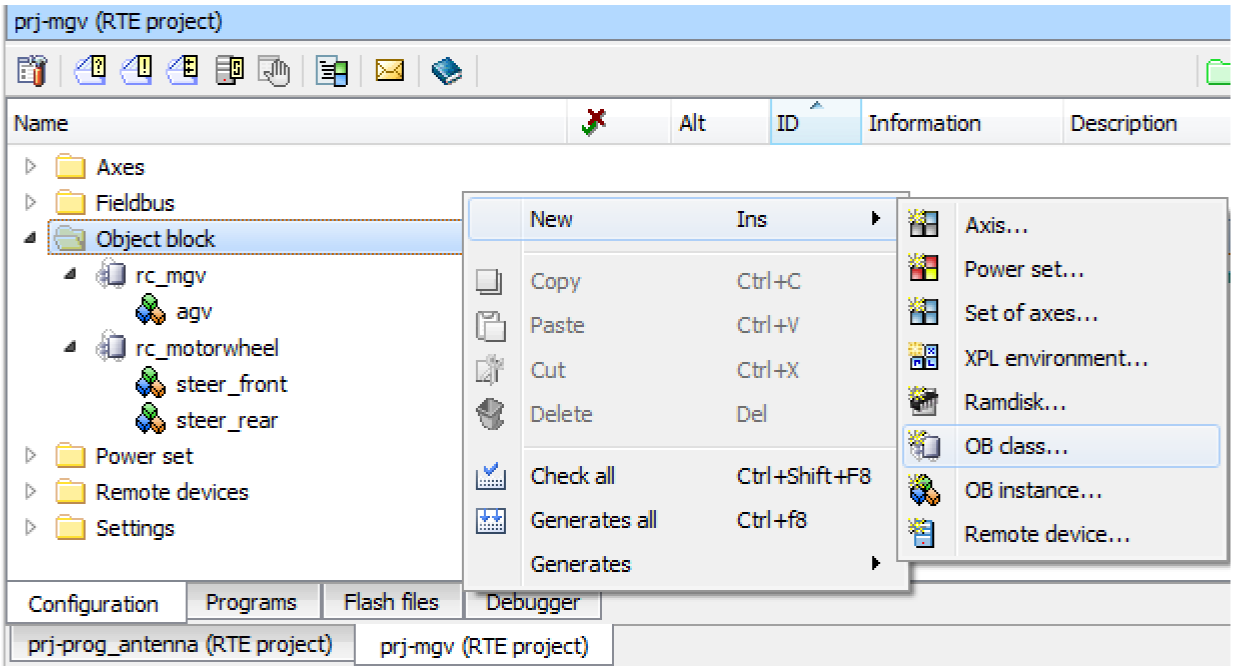


Fig. 41: OB Class or OB Instance
Add a class than add an instance. In the figure we can see 2 classes : `rc_mgv` and `rc_motorwheel`, and one instance of the first class and two instances of the second one

If we defined a structure in the obs file we can use it to define a variable of that type (stucture type) in R3.

OB Predefined example

In menu file, workspace, specials, predefined examples, we can find the example **OB: Use and OB implementation**. This example provide the source code an OB, `rc_belt`, that handle a belt, a rule and task1 implementation.

The Class `rc_belt` is an OB that can be find in the Object Block library, this OB inherit from the class `rc_belt_base`. The example use another OB from the standard library, `rc_axis`, without providing its source code.

Refer to the official Object Block documentation for more informations about OB classes.

In the obs file of `rc_belt`, *Obs example file*, we can see the interface of the Class, how to use another class by importing it, define inputs and outputs and some methods.

Note: Input and outputs deffer only with the keyword `ro`. When an property is declared as read only behave like an output only like the output of a Function block, otherwise behave like an input-output like an inputoutput of a Function block.

The OB is implemented in two C++ source files. In this OB, 2 classes were defined. The class `rc_belt`, that inherit from `rc_belt_base`, and the class `RCBelt`. The OB main class is the one written in the `OB_FACTORY` block

```
OB_FACTORY(rc_belt)
        OB_INSTANCE(rc_belt)
OB_ENDFACTORY
```

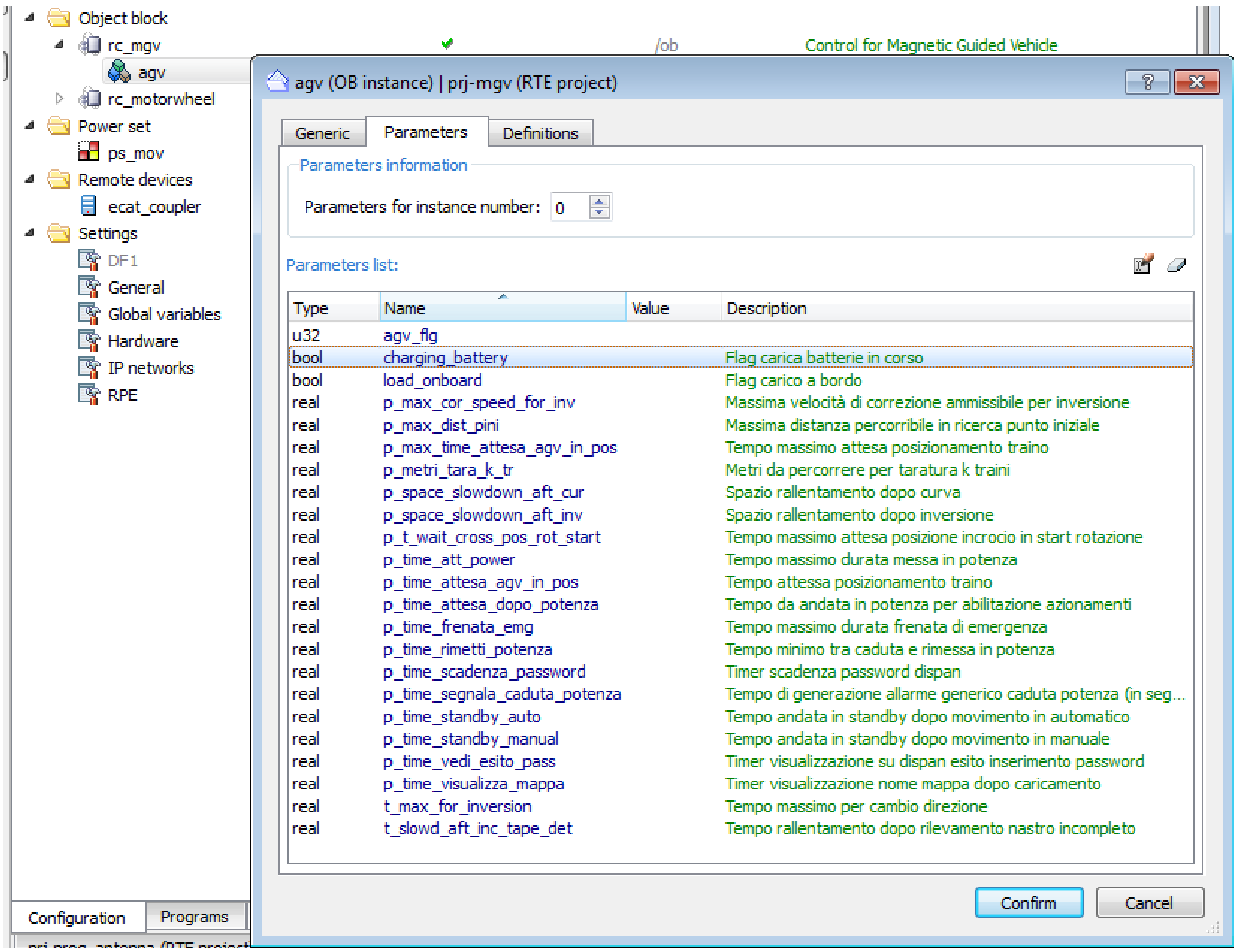
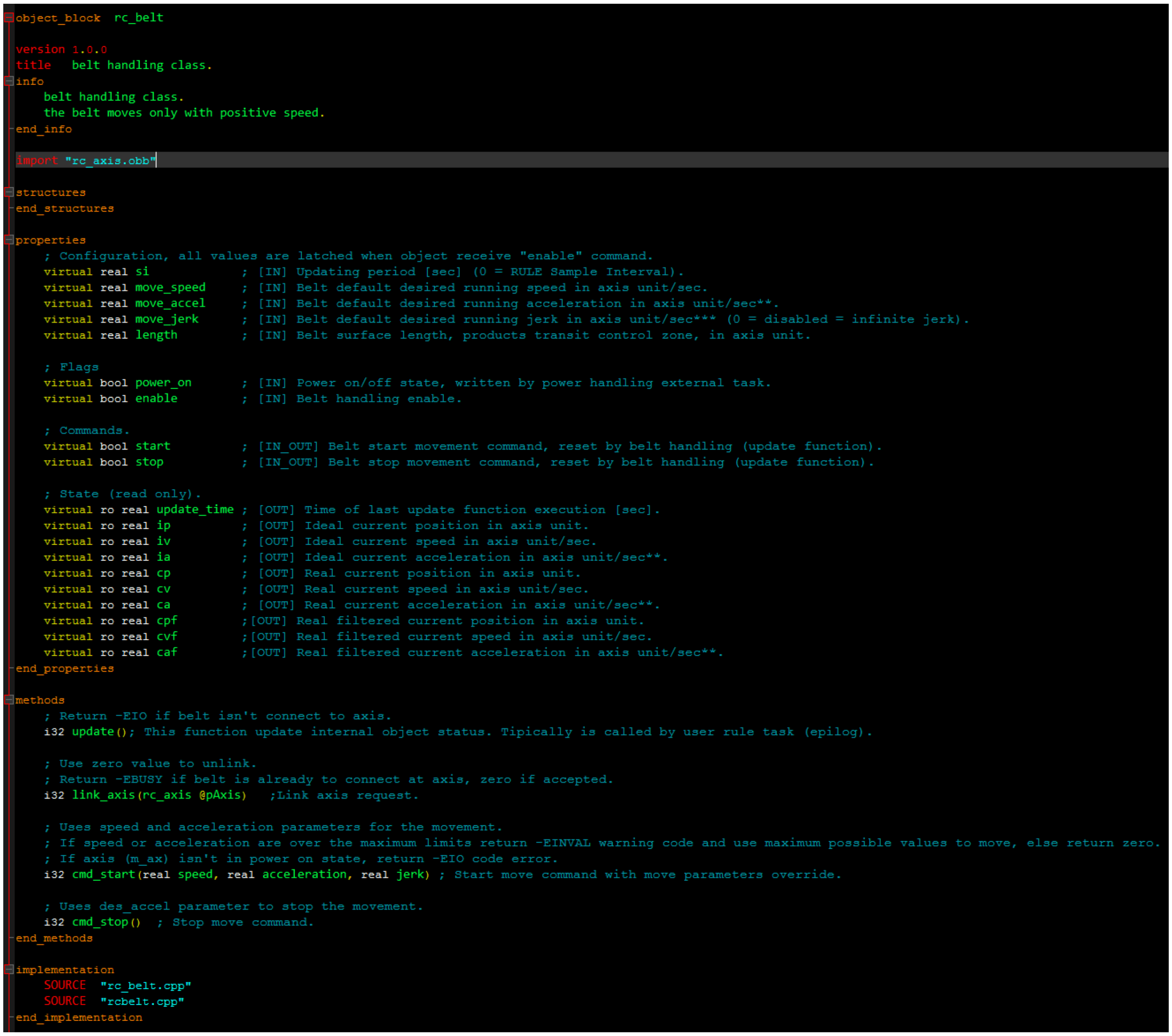


Fig. 42: Object block instance parameters.

In the column `Value` we can initialize the variables. To keep the program easy to read, it is better to initialize OB properties in R3. Note that properties declared as `ro` (read-only) are not shown here.



```
object_block  rc_belt

version 1.0.0
title   belt handling class.
info
    belt handling class.
    the belt moves only with positive speed.
end_info

import "rc_axis.obb"

structures
end_structures

properties
    ; Configuration, all values are latched when object receive "enable" command.
    virtual real si            ; [IN] Updating period [sec] (0 = RULE Sample Interval).
    virtual real move_speed    ; [IN] Belt default desired running speed in axis unit/sec.
    virtual real move_accel    ; [IN] Belt default desired running acceleration in axis unit/sec**.
    virtual real move_jerk     ; [IN] Belt default desired running jerk in axis unit/sec*** (0 = disabled = infinite jerk).
    virtual real length        ; [IN] Belt surface length, products transit control zone, in axis unit.

    ; Flags
    virtual bool power_on      ; [IN] Power on/off state, written by power handling external task.
    virtual bool enable        ; [IN] Belt handling enable.

    ; Commands.
    virtual bool start         ; [IN_OUT] Belt start movement command, reset by belt handling (update function).
    virtual bool stop          ; [IN_OUT] Belt stop movement command, reset by belt handling (update function).

    ; State (read only).
    virtual ro real update_time ; [OUT] Time of last update function execution [sec].
    virtual ro real ip         ; [OUT] Ideal current position in axis unit.
    virtual ro real iv         ; [OUT] Ideal current speed in axis unit/sec.
    virtual ro real ia         ; [OUT] Ideal current acceleration in axis unit/sec**.
    virtual ro real cp         ; [OUT] Real current position in axis unit.
    virtual ro real cv         ; [OUT] Real current speed in axis unit/sec.
    virtual ro real ca         ; [OUT] Real current acceleration in axis unit/sec**.
    virtual ro real cpf        ;[OUT] Real filtered current position in axis unit.
    virtual ro real cvf        ;[OUT] Real filtered current speed in axis unit/sec.
    virtual ro real caf        ;[OUT] Real filtered current acceleration in axis unit/sec**.
end_properties

methods
    ; Return -EIO if belt isn't connect to axis.
    i32 update(); This function update internal object status. Tipically is called by user rule task (epilog).

    ; Use zero value to unlink.
    ; Return -EBUSY if belt is already to connect at axis, zero if accepted.
    i32 link_axis(rc_axis @pAxis)   ;Link axis request.

    ; Uses speed and acceleration parameters for the movement.
    ; If speed or acceleration are over the maximum limits return -EINVAL warning code and use maximum possible values to move, else return zero.
    ; If axis (m_ax) isn't in power on state, return -EIO code error.
    i32 cmd_start(real speed, real acceleration, real jerk) ; Start move command with move parameters override.

    ; Uses des_accel parameter to stop the movement.
    i32 cmd_stop()  ; Stop move command.
end_methods

implementation
    SOURCE  "rc_belt.cpp"
    SOURCE  "rcbelt.cpp"
end_implementation
```

Fig. 43: Obs example file
OB example that use another OB from the standard library. The code in implemented in 2 source files. Example taken from the predefined examples of RDE.

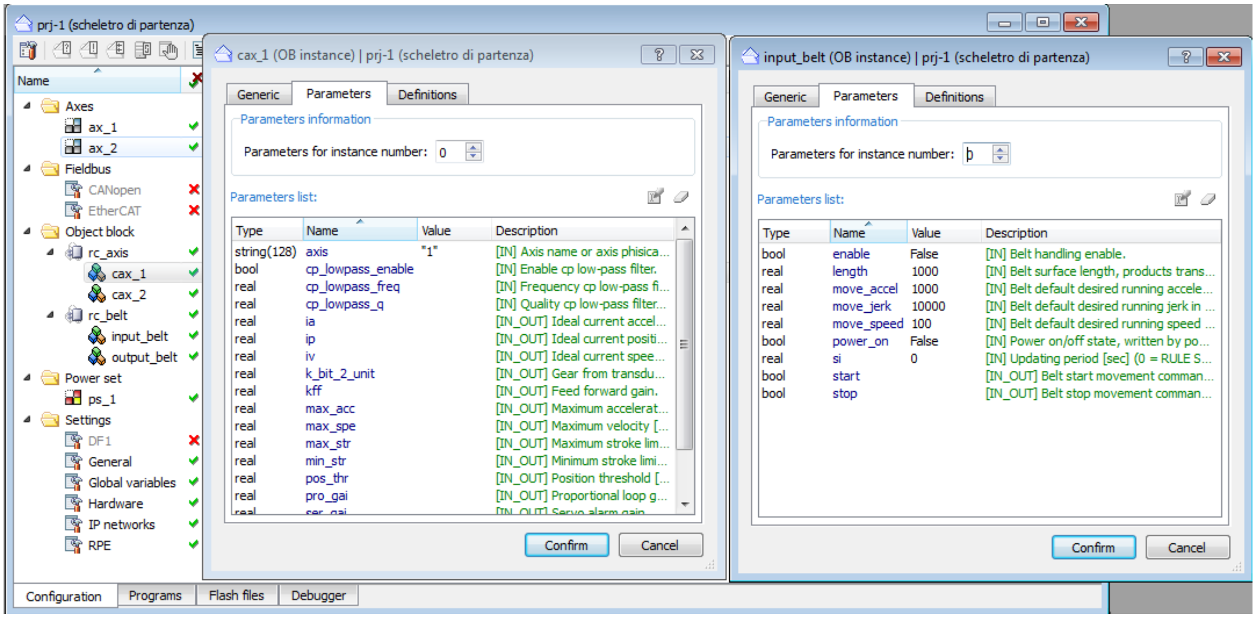


Fig. 44: OB interface
OB: Use and OB implementation, predefined example

1.1.6 OB demo

Cylinder

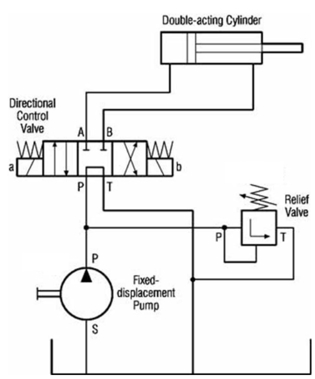


Fig. 45: Hydraulic double acting cyclinder, 3 state electrovalve

1.1.7 X-script

X-script can be used to extend RDE, create shell commands, write AGV scripts, make animation in the 3d graphic panel. User interfaces can be designed in **Qt designer** then deloyed with x-script application.

X-script have some limited object oriented abilities. When it is compiled it generate a byte code, than can be executed by the XVM (X-script virtual machine).

Its syntax is similar to C, pascal and basic. The official documentation provide quite fair explanation of the basic syntax.

The VMI documentation can be found in every tool that can use the X-script language:

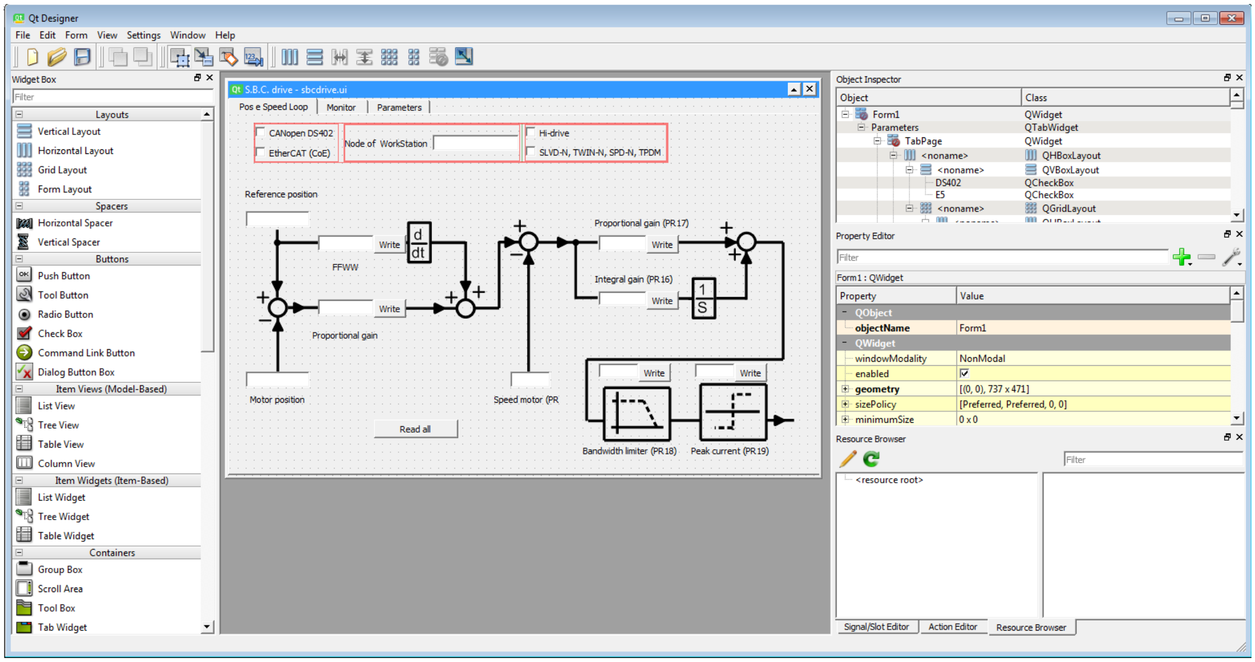


Fig. 46: user interface
User interface example designed with Qt, and implemented in X-script, in order to configure the parameter of a third party drive.

- Command Shell
- 3D graphic panel
- AgvManager
- etc.

Basic syntax

Listing 1: Fundamental data types

```
int, uint, long ; 32 bit
int16, uint16, short, ushort ; 16 bit

char, uchar, byte, bool ; 8 bit

real ; 64 bit
float  ; 32 bit

string ; strings are terminated wit /0 like C

handle (uint)
color (uint)

timeout (real)
```

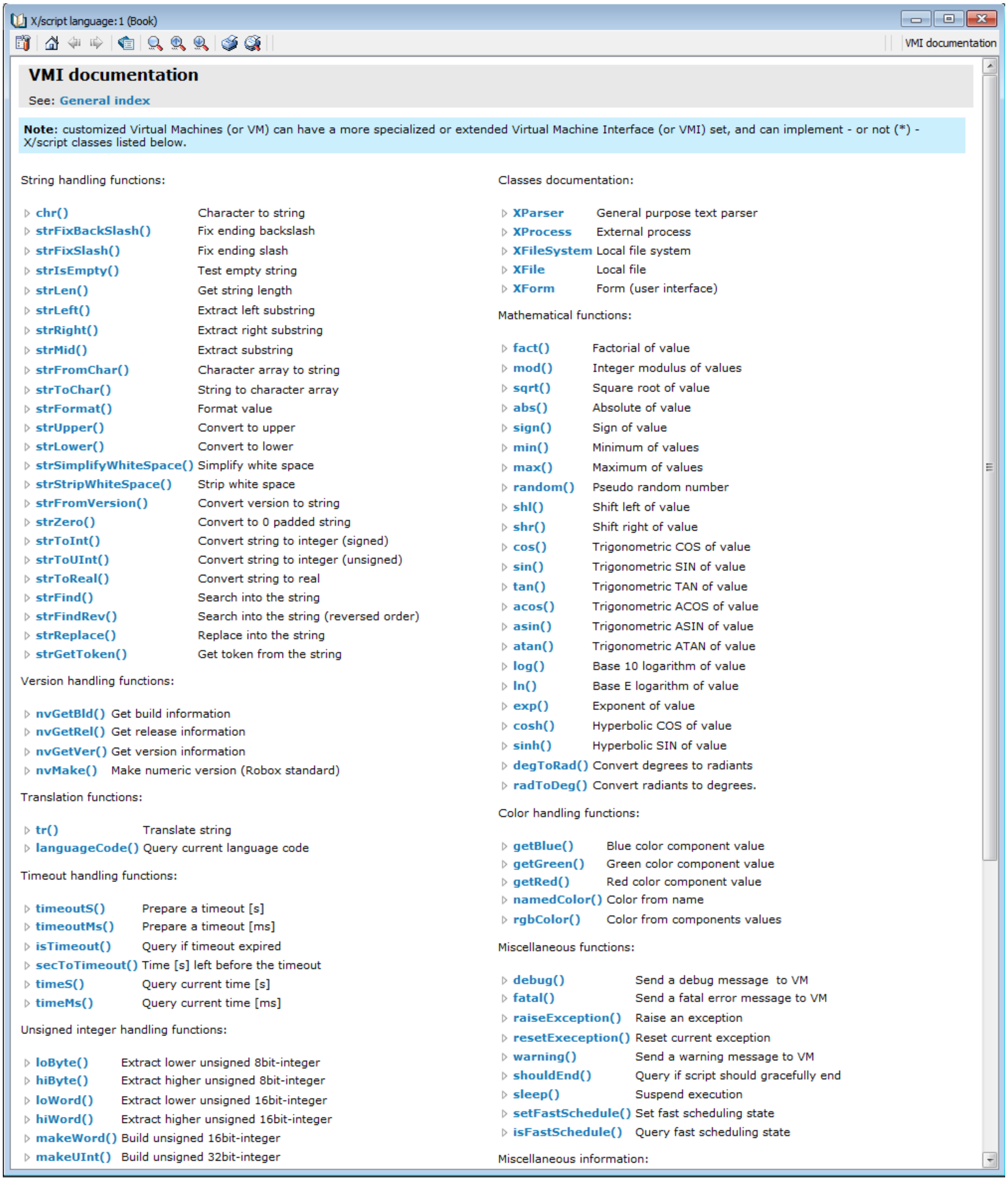


Fig. 47: X-script VMI documentation

Documentation –> Programming languages – > X/script language –> VMI documentation

Listing 2: Control flow

```
if(condition)
else
end or endif

while(condition)
end or endwhile

do
end condition

for(init, cond, update)
endfor

select(var)
 case 2
   ;
   break
 default
   ;
   break
 end or endselect
```

Functions

A function is declared using the keyword `code` and `end` or `endcode` :

```
code functionName()
  ; function body
end

code function2() : int
  int res
  ; function body
  return res
end

code func3(uint par, uint i = 0)
  ; function body
end
```

If a function is implemented in a file after another function that use it, the keyword `forward should be used`. It is like in `C` a function prototype should be provided.

```
forward func2(int)

code func1()
  func2(10)
end

code func2(int c)
  ; function body
endcode
```

Objects

X-scripts objects are like Classes, in order to use them they should be instanciated. An object is declared using the keyword `object` and `endobject` or `end`. First an object interface, header should be provided, then the implementation. Can be done in the same file. An object have also a contructor method.

```
object obClass

 code constructor()

 int var

 code method1()
 code method2(int):bool
endobject

code obClass.contructor()
 ; constructor implementation code
end

code obclass.method1()
 ; method implementation
end
```

Objects are used as classed, can be instanciated. Properties and methods can be accessed via the dot operator.

3D graphic panel

To create a 3D graphic panel in the workspace right click then: New object –> editors –> 3D graphic panel.

3D graphic panels can be customized using X-script language. An example can be found in **Workspace –> specials –> predefined examples – > R3/OB:rc_rod_crank Demo** and in **Workspace –> specials –> predefined examples – > OB: Element location**

Shell commands file

To create new shell commands in the workspace right click then: New object –> editors –> commands file editor. A file with extension `.shc` will be created. Shell commands are implemented using X-script language.

Every shell command should have at least the `execute()` and the `help()` function.

```
code execute (CMDLINE @cl): BOOL
  ; TODO: code for execution of command
  return true
end

code help (): BOOL
  ; TODO: code to request help, like print() o invokeHelp()
  return true
end
```

A *user interface* `ui` can be desinged in `Qt designer` and used in the command shell.

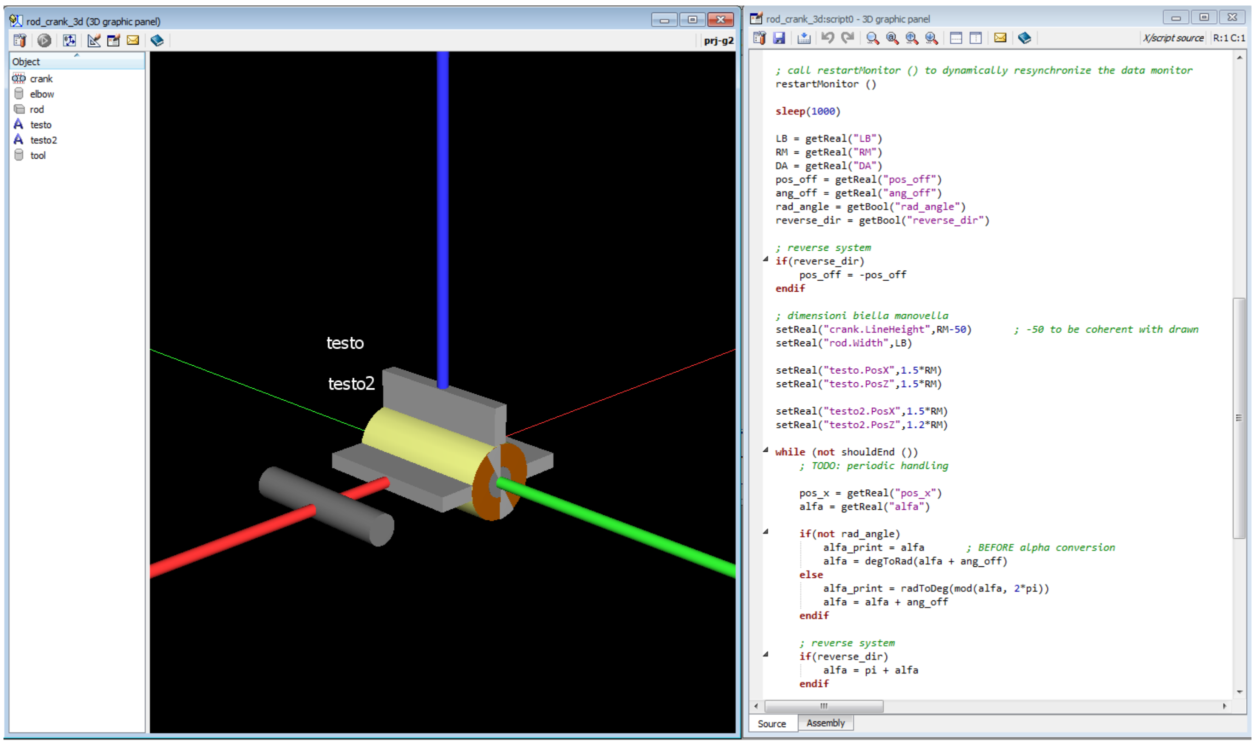


Fig. 48: Customization of a 3D graphic panel with X-script

Fig. 49: Example of Shell commands implemented in X-script

AGV

AGV's plant logic, dispatching, are implemented in X-script language. The script is compiled by AgvManager, not by RDE. Consult the documentation of AGV for more information.

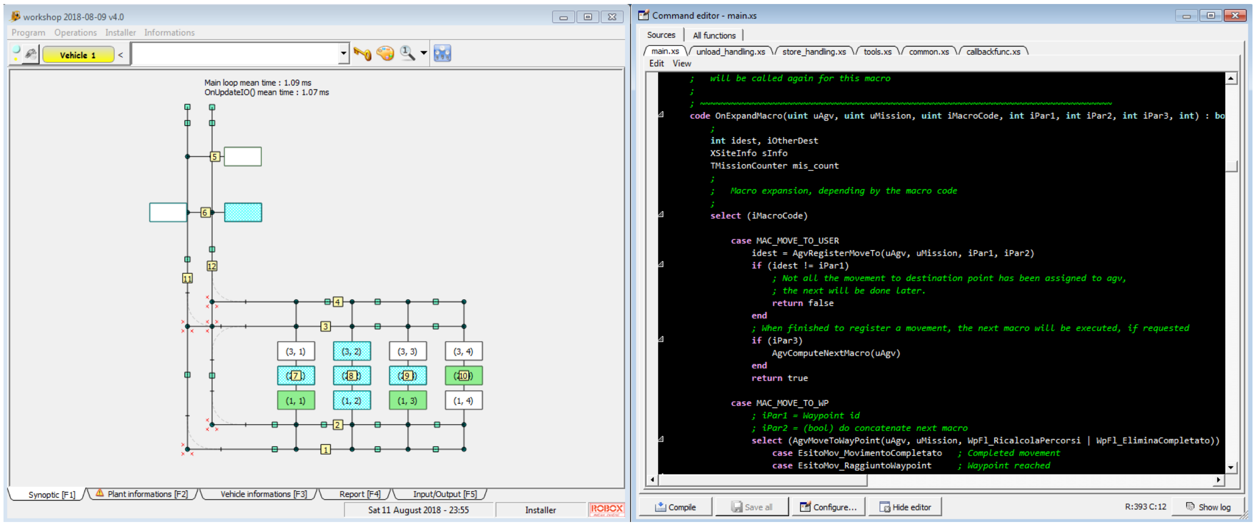


Fig. 50: AGV plant logic in implemented in X-script

1.1.8 X-script 3D Graphic panel

VMI API

The complete projects can be found in the predefined exmples in RDE. We will use Element location and Rod crank predefined examples.

Element location

Date command cource code

```
; ===========================================================================
; ROBOX SpA
; Via Sempione 82, Castelletto Ticino, ITALY
; +390331922086
; http://www.robox.it
; ---------------------------------------------------------------------------
; Script.......:
; Description..: 3D graphic panel customization
; ===========================================================================

; How to use:
; set here the name of rc_elementlocation instance, then save and start the panel
; Note: you must use rc_elementlocation V 1.3.0 or above


$DEFINE EL_NAME "LOV:buffer_belt"
```

(continues on next page)

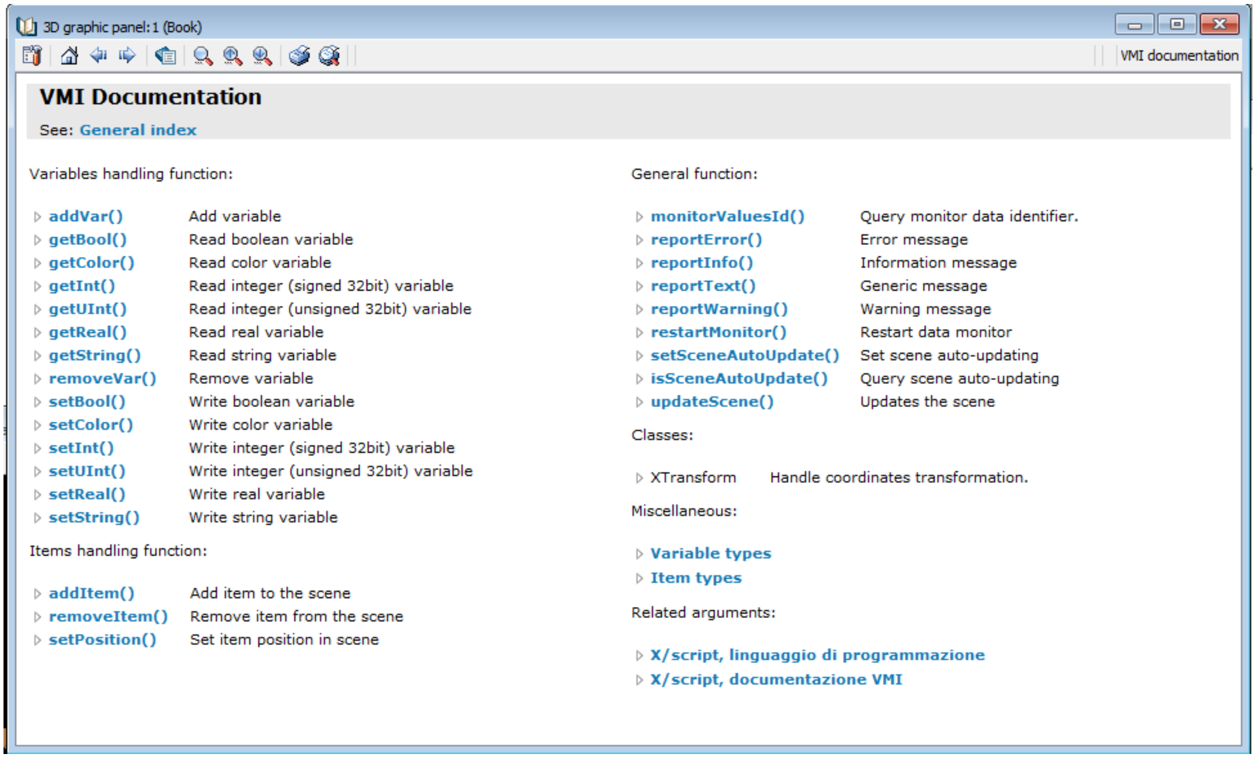


Fig. 51: 3D graphic panel VMI documentations

Documentation –> RDE documention –> 3D graphic panel –> VMI documention

(continued from previous page)

```
; --------------------------------------------------------------------------



addVar(varInt, "elem_nr", EL_NAME + ".NUMBER_OF_ELEMENTS")
addVar(varReal, "el_dim", EL_NAME + ".DISTANCE2END")
addVar(varReal, "passo", EL_NAME + ".VIEW_SCALE")
addVar(varInt, "orig_sin", EL_NAME + ".ORIG_SIN_MARK")
addVar(varBool, "log_inp", EL_NAME + ".LOG_INP")

int elem_nr
real el_dim
real el_offset
int i
int sts
int orig_sin
bool log_inp

restartMonitor ()

sleep(1000)


elem_nr = getInt("elem_nr")
el_dim = getReal("el_dim") * getReal("passo")
```

(continues on next page)

(continued from previous page)

```
el_offset = el_dim * 0.5
orig_sin = getInt("orig_sin")

for (i=0, i<=elem_nr, i=i+1)
        addItem(itemBox, "elem-"+i, "visible="+EL_NAME+".VIEW_PRES["+i+"];posX="+EL_
↪NAME+".VIEW_POS["+i+"];width=0;color=#FFC800;height=50;length=100;offsetX="+(-el_
↪offset)+";offsetZ=25")
        addVar(varReal, "dim-"+i, EL_NAME+".VIEW_DIM["+i+"]")
        addVar(varInt, "sts-"+i, EL_NAME+".VIEW_STS["+i+"]")
end

setReal("piano.width",el_dim + 200)
for (i=1, i<=16, i=i+1)
        setReal("end-"+i+".posX",el_offset)
end
for (i=1, i<=10, i=i+1)
        setReal("start-"+i+".posX",-el_offset)
end


restartMonitor ()



while (not shouldEnd ())

        log_inp = getBool("log_inp")

        for (i=0,i<=elem_nr,i=i+1)
                setReal("elem-"+i+".width",getReal("dim-"+i))
                sts = getInt("sts-"+i)
                if (sts == 0)
                        setString("elem-"+i+".color", "#1F1F1F")
                else
                        if (not (sts & 0x01000000))
                                setString("elem-"+i+".color", "#7FFF7F")
                                if (sts & orig_sin)
                                        log_inp = false
                                end
                        else
                                if (sts & 0xFF)
                                        setString("elem-"+i+".color", "#FF0000")
                                else
                                        setString("elem-"+i+".color", "#FFC800")
                                end
                        end
                end
        end
        setBool("start-7.visible", not log_inp)
        setBool("start-10.visible", log_inp)

end
```

Rod crank

Date command cource code

```
; ==========================================================================
; ROBOX SpA
; Via Sempione 82, Castelletto Ticino, ITALY
; +390331922086
; http://www.robox.it
; --------------------------------------------------------------------------
; Script.......:
; Description..: 3D graphic panel customization
; ==========================================================================

; How to use:
; set here the name of rc_rod_crank OB instance, then save and start the panel

$define rc_sys "biellaman"
;$define rc_sys "ROD_CRANK_INSTANCE_NAME"

; --------------------------------------------------------------------------

; TODO: initialization here
addVar(varReal, "LB", rc_sys + ".rod_len")
addVar(varReal, "RM", rc_sys + ".crank_len")
addVar(varReal, "DA", rc_sys + ".h_pivot")
addVar(varReal, "alfa", rc_sys + ".panel_alfa")
addVar(varReal, "pos_x", rc_sys + ".pos_x")
addVar(varReal, "pos_off", rc_sys + ".pos_offset")
addVar(varReal, "ang_off", rc_sys + ".ang_offset")
addVar(varBool, "rad_angle", rc_sys + ".rad_angle")
addVar(varBool, "reverse_dir", rc_sys + ".reverse_dir")

real rodX
real rodY
real rodAng
real RM
real LB
real DA
real alfa
real beta                ; rod rotation angle an Y axis. Positive sign clockwise␣
↪under RM top
real LBx
real LBy
real pos_x
real pos_off
real ang_off
real alfa_print
bool rad_angle
bool reverse_dir

; call restartMonitor () to dynamically resynchronize the data monitor
restartMonitor ()

sleep(1000)

LB = getReal("LB")
RM = getReal("RM")
```

(continues on next page)

(continued from previous page)

```
DA = getReal("DA")
pos_off = getReal("pos_off")
ang_off = getReal("ang_off")
rad_angle = getBool("rad_angle")
reverse_dir = getBool("reverse_dir")

; reverse system
if(reverse_dir)
        pos_off = -pos_off
endif

; dimensioni biella manovella
setReal("crank.LineHeight",RM-50)                ; -50 to be coherent with drawn
setReal("rod.Width",LB)

setReal("testo.PosX",1.5*RM)
setReal("testo.PosZ",1.5*RM)

setReal("testo2.PosX",1.5*RM)
setReal("testo2.PosZ",1.2*RM)

while (not shouldEnd ())
        ; TODO: periodic handling

        pos_x = getReal("pos_x")
        alfa = getReal("alfa")

        if(not rad_angle)
                alfa_print = alfa                ; BEFORE alpha conversion
                alfa = degToRad(alfa + ang_off)
        else
                alfa_print = radToDeg(mod(alfa, 2*pi))
                alfa = alfa + ang_off
        endif

        ; reverse system
        if(reverse_dir)
                alfa = pi + alfa
        endif

        LBx = pos_x - pos_off - RM*sin(alfa)
        LBy = RM*cos(alfa) - DA

        beta = atan2(LBy, LBx)
        setString("testo.text", strformat("alpha: %.2f deg", alfa_print))
        setString("testo2.text", strformat("pos: %.2f mm" ,pos_x))

        rodX = RM*sin(alfa) + 0.5*LB*cos(beta)
        rodY = RM*cos(alfa) - 0.5*LB*sin(beta)
        rodAng = radToDeg(beta)

        setReal("crank.beltPos",radToDeg(alfa))
        setReal("rod.posX",rodX)
        setReal("rod.posZ",rodY)
        setReal("rod.posB",rodAng)

        setReal("elbow.posX", RM*sin(alfa))
```

(continues on next page)

(continued from previous page)

```
        setReal("elbow.posZ", RM*cos(alfa))

        setReal("tool.posX",pos_x - pos_off)
        setReal("tool.posZ",DA)

        sleep (100) ; 10hz loop
end

; TODO: termination here
```

1.1.9 X-script Command shell

VMI API

Shell commnands are written in X-script language.

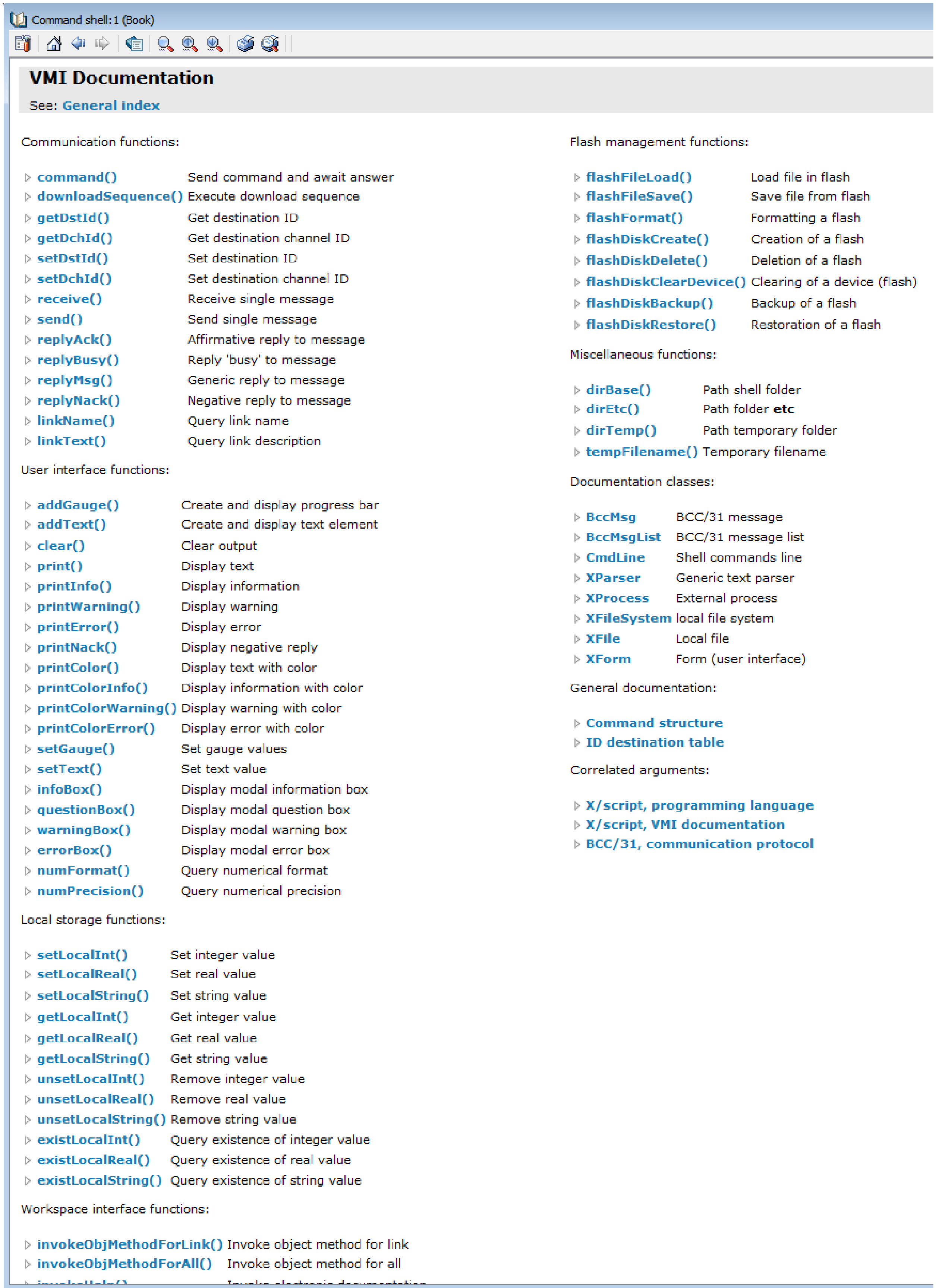
In this chapter we will see 2 commands that use the BCC communication protocol of Robox, in order to communicate with the controller. The 2 commands can be found in the installation folder of RDE.

ALS command

`ALS command source code`

```
; ============================================================================
; ROBOX SpA
; Via Sempione 82, Castelletto Ticino, ITALY
; +390331922086
; http://www.robox.it
; ----------------------------------------------------------------------------
; Script.......: ALS
; Description..: Display alarms stack content
; ============================================================================

code help (): bool
        print ("ALS [-E] [pos]", textBold)
        if (languageCode() == "it")
                print ("Visualizza contenuto stack allarmi.", textItalic)
                print ("Parametri:")
                print ("  -E, informazioni estese")
                print ("  pos, indice dello stack (1-N)")
        else
                print ("Display alarms stack content.", textItalic)
                print ("Parameters:")
                print ("  -E, extended information")
                print ("  pos, stack index (1-N)")
        end
        return true
end

; ----------------------------------------------------------------------------

code printStackPosition (bccmsg @asw)
        string buf
        buf = "als("+  strFormat("%2d",asw.u32(0)) +")  "
```

(continues on next page)


Command shell:1 (Book)

VMI Documentation

See: General index

Communication functions:

command()	Send command and await answer
downloadSequence()	Execute download sequence
getDstId()	Get destination ID
getDchId()	Get destination channel ID
setDstId()	Set destination ID
setDchId()	Set destination channel ID
receive()	Receive single message
send()	Send single message
replyAck()	Affirmative reply to message
replyBusy()	Reply 'busy' to message
replyMsg()	Generic reply to message
replyNack()	Negative reply to message
linkName()	Query link name
linkText()	Query link description

User interface functions:

addGauge()	Create and display progress bar
addText()	Create and display text element
clear()	Clear output
print()	Display text
printInfo()	Display information
printWarning()	Display warning
printError()	Display error
printNack()	Display negative reply
printColor()	Display text with color
printColorInfo()	Display information with color
printColorWarning()	Display warning with color
printColorError()	Display error with color
setGauge()	Set gauge values
setText()	Set text value
infoBox()	Display modal information box
questionBox()	Display modal question box
warningBox()	Display modal warning box
errorBox()	Display modal error box
numFormat()	Query numerical format
numPrecision()	Query numerical precision

Local storage functions:

setLocalInt()	Set integer value
setLocalReal()	Set real value
setLocalString()	Set string value
getLocalInt()	Get integer value
getLocalReal()	Get real value
getLocalString()	Get string value
unsetLocalInt()	Remove integer value
unsetLocalReal()	Remove real value
unsetLocalString()	Remove string value
existLocalInt()	Query existence of integer value
existLocalReal()	Query existence of real value
existLocalString()	Query existence of string value

Workspace interface functions:

invokeObjMethodForLink()	Invoke object method for link
invokeObjMethodForAll()	Invoke object method for all
invokeHelp()	Invoke electronic documentation

Flash management functions:

flashFileLoad()	Load file in flash
flashFileSave()	Save file from flash
flashFormat()	Formatting a flash
flashDiskCreate()	Creation of a flash
flashDiskDelete()	Deletion of a flash
flashDiskClearDevice()	Clearing of a device (flash)
flashDiskBackup()	Backup of a flash
flashDiskRestore()	Restoration of a flash

Miscellaneous functions:

dirBase()	Path shell folder
dirEtc()	Path folder **etc**
dirTemp()	Path temporary folder
tempFilename()	Temporary filename

Documentation classes:

BccMsg	BCC/31 message
BccMsgList	BCC/31 message list
CmdLine	Shell commands line
XParser	Generic text parser
XProcess	External process
XFileSystem	local file system
XFile	Local file
XForm	Form (user interface)

General documentation:

- Command structure
- ID destination table

Correlated arguments:

- X/script, programming language
- X/script, VMI documentation
- BCC/31, communication protocol

Fig. 52: Command shell VMI documentation
Documentation –> RDE documention –> Command shell –> VMI documention

Fig. 53: Example of Shell commands implemented in X-script

(continued from previous page)

```
        buf = buf + " ac=" + strFormat("%-4d",asw.u16(4))
        if (asw.u16(6) != 0)
                buf = buf + " ax=" + strFormat("%-2d",asw.u16(6))
        else
                buf = buf + "       "
        end
        buf = buf + strFormat(" '%s'", asw.str(40))
        print(buf)
end

; ---------------------------------------------------------------------------

code execute (cmdline @cl): bool
        int pos = 0
        bool extInfo = false
        string alsId
        string alsTitle
        bccmsg cmd, asw, msg
        bccmsglist msgs

        ; Imposta task veloce
        setFastSchedule(true)

        ; Titolo
        print (tr ("us=Alarms stack contents^it=Contenuto stack allarmi"), textBold)

        ; Verifica opzioni
        while (cl.isOption ())
                if (strLower (cl.asString()) == "e")
                        extInfo = true
                        cl.next ()
                        continue
                end

                printError (tr ("us=Wrong option -^it=Opzione errata -") + cl.
↪asString ())
                return false
        end

        ; Verifica parametro posizione (opZ)
        if (cl.isInteger ())
                pos = cl.asInt ()
                if (pos < 1)
                        printError (tr("us=Invalid stack index^it=Indice dello stack␣
↪non valido"))
                        return false
                end
                cl.next ()
        end

        ; Ignora parametri extra
        cl.ignoreExtra ()
```

(continues on next page)

(continued from previous page)

```
        ; Richiesta singola posizione
        if (pos)
                        ; Compose and send request command
                cmd.msgcode = AS|520
                cmd.msglen = 8
                cmd.u32(0) = 0                         ; flags
                cmd.u32(4) = pos                 ; posizione
                if (not command (@cmd, @asw))
                        printnack (@asw)
                        return false
                end
                printStackPosition(@asw)
                return true
        end

        ; Richiesta stack completo allarmi
        cmd.msgcode = AS|521
        cmd.msglen = 4
        cmd.u32(0) = 0                         ; flags
        msgs.clear ()
        if (not downloadSequence (@cmd, @asw, @msgs))
                printnack (@asw)
                return false
        end

        ; Composizione titolo
        alsId = strFormat ("0x%08x", asw.u32 (8))
        if (languageCode () == "it")
                alsTitle = "Visualizzazione di " + msgs.count() + " su "+ asw.u32 (4)␣
↪+ " allarmi"
                if (extInfo)
                        alsTitle += ", con ID "+ alsId
                end
        else
                alsTitle = "Displaying " + msgs.count () + " of " + asw.u32 (4) + "␣
↪alarms"
                if (extInfo)
                        alsTitle += ", with ID " + alsId
                end
        end

        ; Stampa contenuto dello stack
        if (msgs.count () > 0)
                print (alsTitle)
                if (msgs.first(@msg))
                        do
                                printStackPosition(@msg)
                        end msgs.next (@msg)
                end

        ; Stampa stack vuoto
        else
                if (extInfo)
                        print (alsTitle)
                end
                print (tr ("us=No alarm in stack^it=Nessun allarme in stack"))
        end
```

(continues on next page)

(continued from previous page)

```
        return true
end
```

DATE command

Date command cource code

```
; ===========================================================================
; ROBOX SpA
; Via Sempione 82, Castelletto Ticino, ITALY
; +390331922086
; http://www.robox.it
; ---------------------------------------------------------------------------
; Script.......: date
; Description..: Show (or set) date for connected device
; ===========================================================================

code help (): bool
        print ("DATE [dd mm yy]", textBold);
        print ("DATE -LSET", textBold);
        if (languageCode() == "it")
                print ("Visualizza (o imposta) la data per il dispositivo.",␣
↪textItalic)
                print ("Parametri:")
                print ("  -LSET, imposta data usando la data locale")
                print ("  dd, giorno (1-31)")
                print ("  mm, mese (1-12)")
                print ("  yy, anno (2003-2100)")
        else
                print ("Display (or set) date for the device.", textItalic)
                print ("Parameters:")
                print ("  -LSET, set date using local date")
                print ("  dd, day (1-31)")
                print ("  mm, month (1-12)")
                print ("  yy, year (2003-2100)")
        end
        print ("")
        print ("DATE -L", textBold);
        if (languageCode() == "it")
                print ("Visualizza data locale.", textItalic)
        else
                print ("Display local date.", textItalic)
        end
        return true
end

; ---------------------------------------------------------------------------

code showDate (): bool

        ; Titolo
        print (tr ("us=Current date&^it=Data corrente"), textBold)
```

(continues on next page)

(continued from previous page)

```
        ; Compose command message
        bccmsg cmd, asw
        cmd.msgcode = AS|503
        if (not command (@cmd, @asw))
                printnack (@asw)
                return false
        end

        ; Show result
        int d, m, y
        d = asw.u8 (3)
        m = asw.u8 (4)
        y = asw.u16 (5)
        print (dayName (d, m, y) + ", " + dateToString (d, m, y))
        return true;
end

; ----------------------------------------------------------------------------

code showLocalDate (): bool
        int d, m, y
        print (tr ("us=Current date (local)^it=Data corrente (locale)"), textBold)
;        print (dateToString (day (), month (), year ()))
        d = day ()
        m = month ()
        y = year ()
        print (dayName (d, m, y) + ", " + dateToString (d, m, y))
        return true;
end

; ----------------------------------------------------------------------------

code setDate (cmdline @cl): bool
        int d, m, y
        bccmsg cmd, asw

        ; Read and check first parameter (DAY)
        if (cl.isInteger ())
                d = cl.asInt ()
                if (d < 1 or d > 31)
                        printError (tr("us=Invalid day (1-31)^it=Giorno non valido (1-
↪31)"))
                        return false
                end
                cl.next ()
        else
                printError (tr("us=Expected a day number^it=Atteso numero giorno "));
                return false
        end

        ; Read and check second parameter(MONTH)
        if (cl.isInteger ())
                m = cl.asInt ()
                if (m < 1 or m > 12)
                        printerror (tr("us=Invalid month (1-12)^it=Mese non valido (1-
↪12)"))
                        return false
```

(continues on next page)

(continued from previous page)

```
                end
                cl.next ()
        else
                printerror (tr("us=Expected a month number^it=Atteso numero mese"));
                return false
        end

        ; Read and check third parameter(YEAR)
        if (cl.isInteger ())
                y = cl.asInt ()
                if (y < 2003 or y > 2100)
                        printerror (tr("us=Invalid year (2003-2100)^it=Anno non␣
↪valido (2003-2100)"))
                        return false
                end
                cl.next ()
        else
                printerror (tr("us=Expected a year number^it=Atteso numero anno"))
                return false
        end
        cl.ignoreExtra ()

        ; Check date validity
        if (not isDateValid (d, m, y))
                printerror (tr("us=Invalid date^it=Data non valida"))
                return false
        end

        ; Prepare and send command
        cmd.msgcode = AS|504
        cmd.msglen = 8
        cmd.u8(0)  = 0
        cmd.u8(1)  = 0
        cmd.u8(2)  = 0
        cmd.u8(3)  = byte(d)
        cmd.u8(4)  = byte(m)
        cmd.u16(5)= word(y)
        cmd.u8(7)  = byte(0x0038)

        ; Send message
        if (not command (@cmd, @asw))
                printnack (@asw)
                return false;
        end
        return true
end

; ---------------------------------------------------------------------------

code setDateFromLocal (): bool
        int d, m, y
        bccmsg cmd, asw

        ; Init variables
        d = day ();
        m = month ();
        y = year ();
```

(continues on next page)

(continued from previous page)

```
        ; Prepare and send command
        cmd.msgcode = AS|504
        cmd.msglen = 8
        cmd.u8(0)  = 0
        cmd.u8(1)  = 0
        cmd.u8(2)  = 0
        cmd.u8(3)  = byte(d)
        cmd.u8(4)  = byte(m)
        cmd.u16(5)= word(y)
        cmd.u8(7)  = byte(0x0038)

        ; Send message
        if (not command (@cmd, @asw))
                printnack (@asw)
                return false;
        end
        return true
end

; ---------------------------------------------------------------------------

code execute (cmdline @cl): bool
        bool result
        bool showLocal = false
        bool setFromLocal = false

        ; Imposta task veloce
        setFastSchedule(true)

        ; Check options
        while (cl.isOption ())
                if (strLower (cl.asString()) == "l")
                        showLocal = true
                        cl.next ()
                        continue
                end
                if (strLower (cl.asString()) == "lset")
                        setFromLocal = true
                        cl.next ()
                        continue
                end
                printError (tr("us=Wrong option -^it=Opzione errata -") + cl.
↪asString())
                return false
        end

        ; Check and launch operation
        if (showLocal)
                cl.ignoreExtra ()
                result = showLocalDate ()
        else
                if (setFromLocal)
                        cl.ignoreExtra ()
                        result = setDateFromLocal ()
                else
                        if (cl.isEol ())
```

(continues on next page)

(continued from previous page)

```
                                                result = showDate ()
                                        else
                                                result = setDate (cl)
                                        end
                                end
                        end

                        return result;
end
```

Using XForm and Qt designer

`The User interface` is designed in **Qt designer**. The extension of the file is `.ui` and it is an `xml` file.

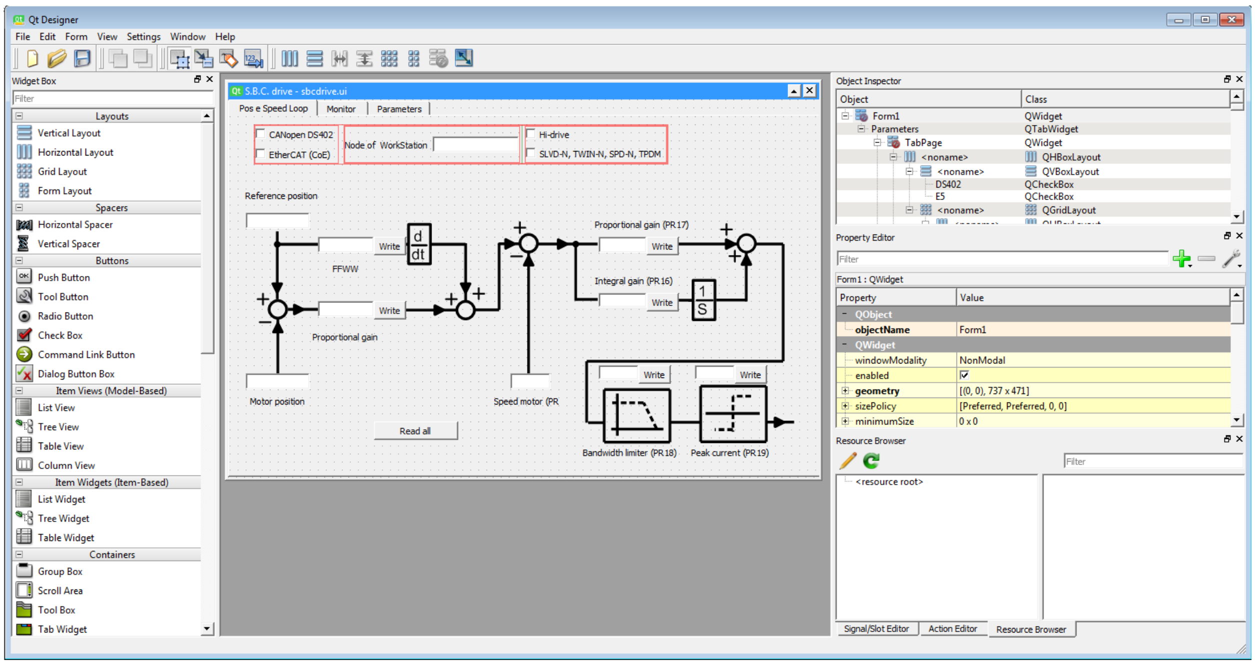


Fig. 54: Qt designer

`This script` use the Class XForm in order to handle the Qt user interface and send configuration parameters to a third party drive.

1.1.10 RDT

1.1.11 Modbus

1.1.12 Basics

RTE scheduler

RTE is a realtime preemtive operating system. The principles of RTE are quite simple. In a one core CPU, an operating system give the illusion to the user that it is executing programs, tasks or processes in parallel (in the same time). Even on a multicore computer, we have this illusion. Supposed our machine have a 4 core cpu, and we are using a webbrowser, a keyboard, a mouse, music player, a word processor, mail client, the OS is executing processes that the user ignore, etc. at the same time, 4 core are not enough to do the job.

Simply RTE allow execution of one motion task called `RULE` and 8 general purpose tasks. The rule have high priority, and it is a periodic task, that execute always at predefined time `si`, sampling interval. And tasks are executed in time sharing, for example 10 instruction from task1, then 10 from task2, and so on until task 8, then back agian to the following 10 instructions of task 1 and so on. This inifinite loop or iteration is interrpted at fixed time `si` by RTE in order to execute the RULE.

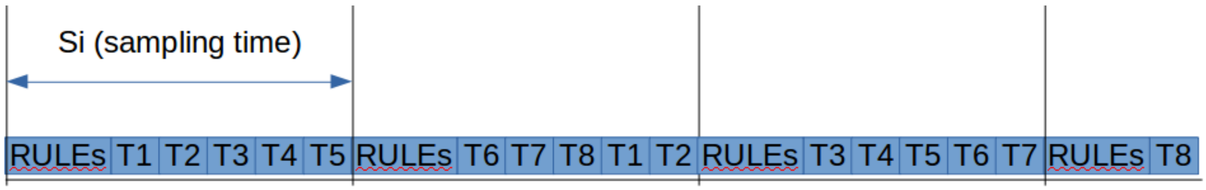


Fig. 55: RTE scheduler. The execution of task depend on the sampling time. But RULEs are always executed periodically with a period equal to *si*'.

Note: The sampling time can be read from the predefined variable `si`. The period frequency can be set in RTE configuration.

RULEs are a more complex concept. There are more then one rule, in RTE they are all executed together in sequence in the same sampling time. If RTE can't execute them in one period, may be there are a lot of instruction and `si` is too short e.g. `0.2ms`, RTE give an alarm and you have to increase `si`. Typical value `si = 5ms` on RP1.

RTE can execute until 32 RULEs plus other 2 special ones called `RULE_PROLOGUE` and `RULE_EPILOGUE`. The sequence can be assinged in R3 program. Toegether with RULEs RTE execute other compenents. But for our purpose, we care only about RULEs.

You can immagine RULEs as different functions that RTE call in the sequence that you tell him. There is only one R3 program with the keyword `$RULES` where rules and other helper funtions are written.

Note: The rule file can have up to 1000 rules, but RTE can execute maximum 32 of them in the same sampling time.

Complete overview on RTE multitasking:

In this chapter we will create a simple demo in order to show how we can configure an axis (drive + motor) and simulate it.

Axis configuration

We will create three projects in one workspace, in order to illustrate different axis configurations:

- Ethercat
- CanOpen
- Analog reference

Each project reside in one folder in the workspace.

Every axis have a unique name and a unique index, that can be used in R3 programs.

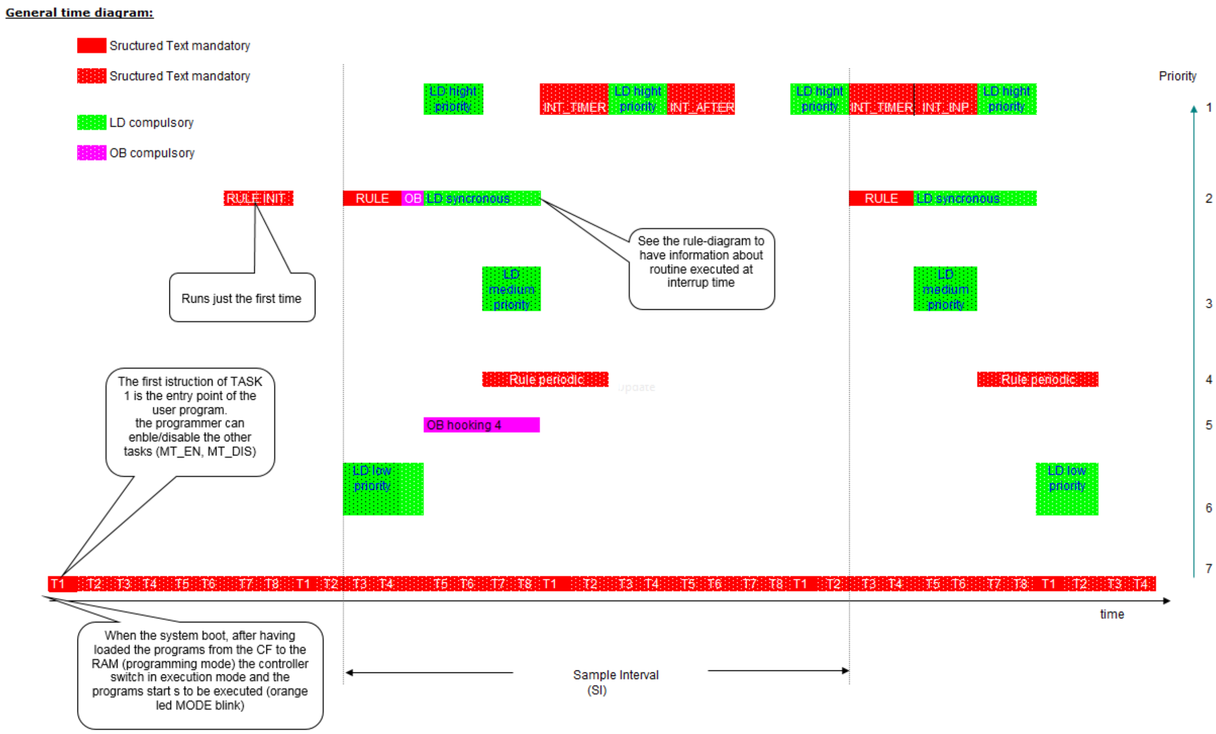


Fig. 56: RTE multitasking

Analog reference

Let's suppose we have a motor drive with analog reference speed control, and and feedback position.

We configure 2 axis, The first axis speed refrence is assigned to a volatile real register `rr(1)` and the feedback position to `rr(11)`. Note that we check `emulated` field in order to be able to emulate the drives. With a real drive, this field should be clear.

Ethercat

When controlling a drive via a bus, e.g. Ethercat, profinet, etc. we use PDO to control the drive.

Important input words via bus:

- Status word : A mask that contain the status of the drive e.g. running, alarm, etc.
- Actual position or Actual speed

Importnat output words:

- Control word : A mask with command to the drive e.g. enable, run, etc.
- target position or target speed.

The role of each bit in the status and control words depend on the drive configuration.

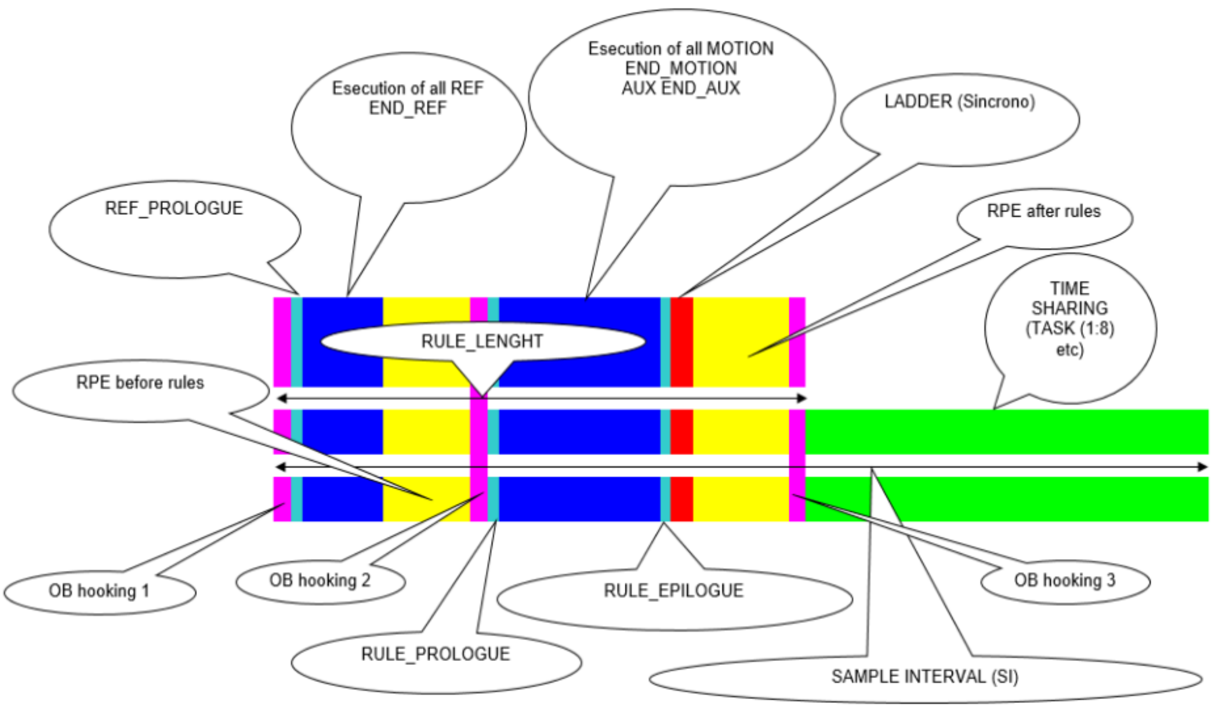
Robox drive: IMD

Robox Integrated Drive

Priority	Task type
7	TASK in BACKGROUND (time-sharing) They are 8 low priority tasks written in the R3 language, used for not time consuming functions (for instance the management of the machine logic). Task 1 ($TASK1) represents the program entry point. This task will enable/disable the other tasks (see the instructions **mt_en** and **mt_dis**). The typical architecture of these tasks consists in an initialization session followed by an endless loop where the different operations/tasks of the controlled machines are performed. The correct evolution of these tasks is ensured by RTE. If any misbehaviour occurs, alarm **9113** User Task(t.s.): reduced freq. <Tname> will be output. Any information on the length and frequency are at the programmer's care (**loop_time**). With device command **ts_per_override** and **ts_nst_override** is possible to display /overridedi default settings
6	**LOW PRIORITY LADDER TASKS** Ladder task whose execution frequency is programmed by the user (1Hz÷2000Hz). Its length can be viewed through the predefined variable **ltl_length**
5	OB service.
4	RULE_PERIODIC RULE executed at a frequency programmed by the user. As the priority of this rule is lower than the one of the other RULES, its execution is subject to jitter, whose max length will be equal to the time required by RTE to execute the tasks with higher priority (RULE + SYNCRONOUS LADDER TASKS + HIGH PRIORITY LADDER TASKS + TASK ON EVENT, iif present).
3	**NORMAL PRIORITY LADDER TASKS** Task written in Ladder language whose execution frequency is programmed by the user (1Hz÷2000Hz). Its length can be viewed through the predefined variable **ltl_length**
2	RULES (fixed frequency functions -interrupt-) Tasks written in R3 language, reserved to the path building, to the descriptions of the links among the different axes and to the execution of the feedback algorithm (loop closure) RTE can execute up to 32 RULES (RC) in the same system interrupt. The selection of the RULES to be executed is done with the instruction **GROUP**, while the execution sequence can be programmed with the instruction **ORDER**. The rules execution frequency can be programmed with the instruction **RULE_FREQ** (in the range 25Hz÷2000Hz). With the CPU P2020 its max frequency is 5000hz. Its length can be viewed through the predefined variable **rule_length** **RULE_INIT** RULE executed just once, before the execution of the other RULES. It is executed the first time the instructions **ORDER** or **GROUP** is invoked. No RULE is active until the GROUP instruction has been executed or the predefined variable **RC** is set. If you wish to execute a **rule_init**, for instance to activate a **rule_prologue or epilogue** even in absence of rules, use the keyword **rule_start_norc**
2	**SYNCHRONOUS LADDER TASK** Task written in Ladder language, executed (if present) together with the RULES. Its length can be viewed through the predefined variable **ltl_length**
1	TASK ON EVENT They are particular tasks written in R3 language with max system priority and which are used to solve some particular requirements. The enabling events are: *Variation edge of a digital input (**INT_INP**) *Set frequency (**INT_TIMER**) *Time delay (**INT_AFTER**) Any RTE running operation is interrupted to handle these events (a typical latency is 40us). Consequently we advise the user that an excessive use of this performace can result in a degradation of the system's normal performance.
1	**HIGH PRIORITY LADDER** Task written in Ladder, whose execution period(PERIODO DI ESECUZIONE) is set by the user (1Hz÷2000Hz). Its length can be viewed through the predefined variable **ltl_length** Any RTE running operation is interrupted to handle these events (a typical latency is 40us). Consequently we advise the user that an excessive use of this performace can result in a degradation of the system's normal performance.

Fig. 57: Priority

Single rule actions:



Function	Description
OB hooking 1	Object Blocks hooked hook1
REF_PROLOGUE	Optional function which, if defined, is first executed when a system interrupt occurs (and threfore it has less gitter with respect to the sample interval SI)
REF	Depending on the active RULES, all the REF, END_REF fields for the axes defined in such rules are executed in order to generate the driving reference (result of the option loop)
RPE	Handling of a group of axes by RPE with selection "before the rules"
OB hooking 2	Object Blocks hooked hook2
RULE_PROLOGUE	Optional function which, if defined, is executed immediately after the REF fields
MOTION, AUX	MOTION Generation of new kinematic variables for the axes (IP or IV or IA) The distinction between MOTION and AUX is only conceptual and is adopted by particularly meticolous programmers
RULE_EPILOGUE	Optional function which, if defined, is executed immediately after the MOTION AUX fields
LADDER SINCRONO	SYNCHRONOUS LADDER TASK. Use the command LAD_STATUS to get information on the execution timing
RPE	Handling of a group of axes by RPE with selection "before the rules"
OB hooking 3	Object Blocks hooked hook3

Fig. 58: Rule execution

Fig. 59: Multiproject workspace

Fig. 60: Analog speed reference

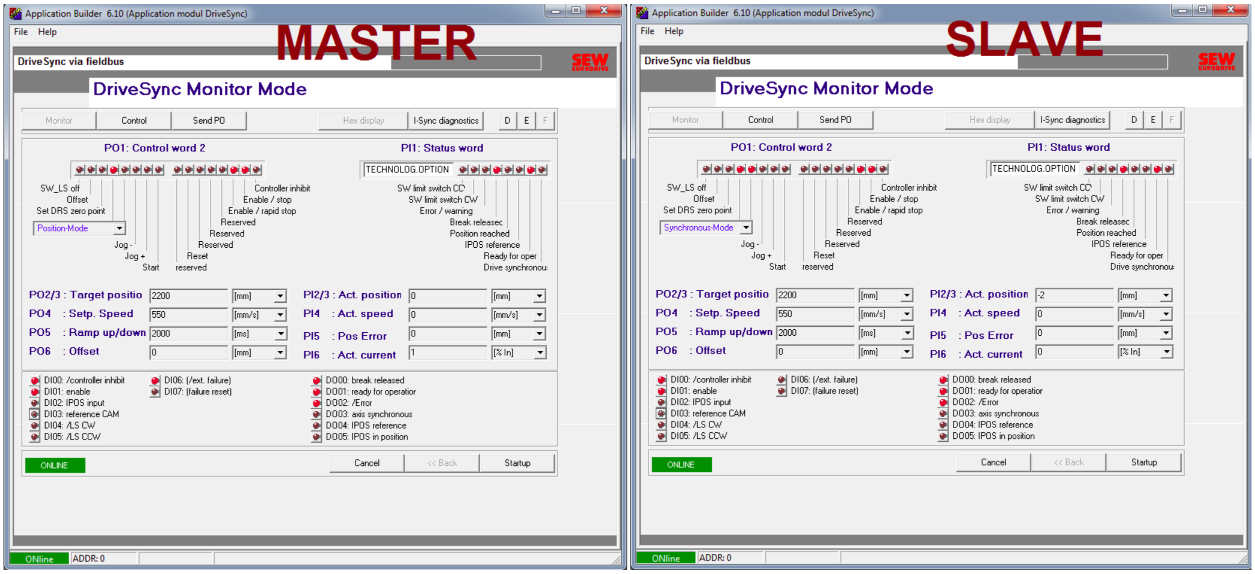


Fig. 61: SEW Status and control word example of 2 different axis configuration.

Fig. 62: Robox IMD20: Ethercat

In order to congirure Robox IMD drives, you can use the predefined example.

CoolDrive

CoolDrive is a chinese drive with Ethercat bus. Download the software **DriveStarter** and the **xml** bus definition from their website.

CanOpen

Powerset

We can imagine the `powerset` as a logical power, for safety purpose, of a set of drives. For example we are controlling a 6-axis anthropomorphic robot and a 3-axis cartesian robot. We can create 2 power sets to group the 6 axis of the first and another one to group the 3-axis of the second one. Of course we can create 9 powersets, one for each axis. Suppose that the 3-axis robot have a problem and need to stop, it is logical to stop all 3 axis.

In the powerset configuration we select the axis, which power is handedled by the powerset the we call `ps`. If axis are controlled via a bus e.g. Canopen over Ethercat, as feedback we choose `CANopen(CAN402)`. Usually even if drives use fieldbus, saftey circuit still exist. Suppose we have an emergency circuit, that is connected to the controller which state can be read in `r(101).0`. In the powerset feedback we add also that register.

If the feedback signals are `HIGH`, the powerset can be energized by the signal that set in the tab requests `POWR_RQ`, power request. In our case, we choose `r(100).0`. Imagine the powerset as a safety realy, if safety condition are met `feedback = true`, the relay contacts can be closed under a request from `POWR_RQ`.

You can find a ready to use graphic panel to monitor the powerset. Later we will see how to use it.

To enable power of different axis, a chain of power in the power set have to be enabled, like a safety chain. There are some predefined variables and funtions that manage axis power. We will some of them in order of chain hierarchy, top-down view:

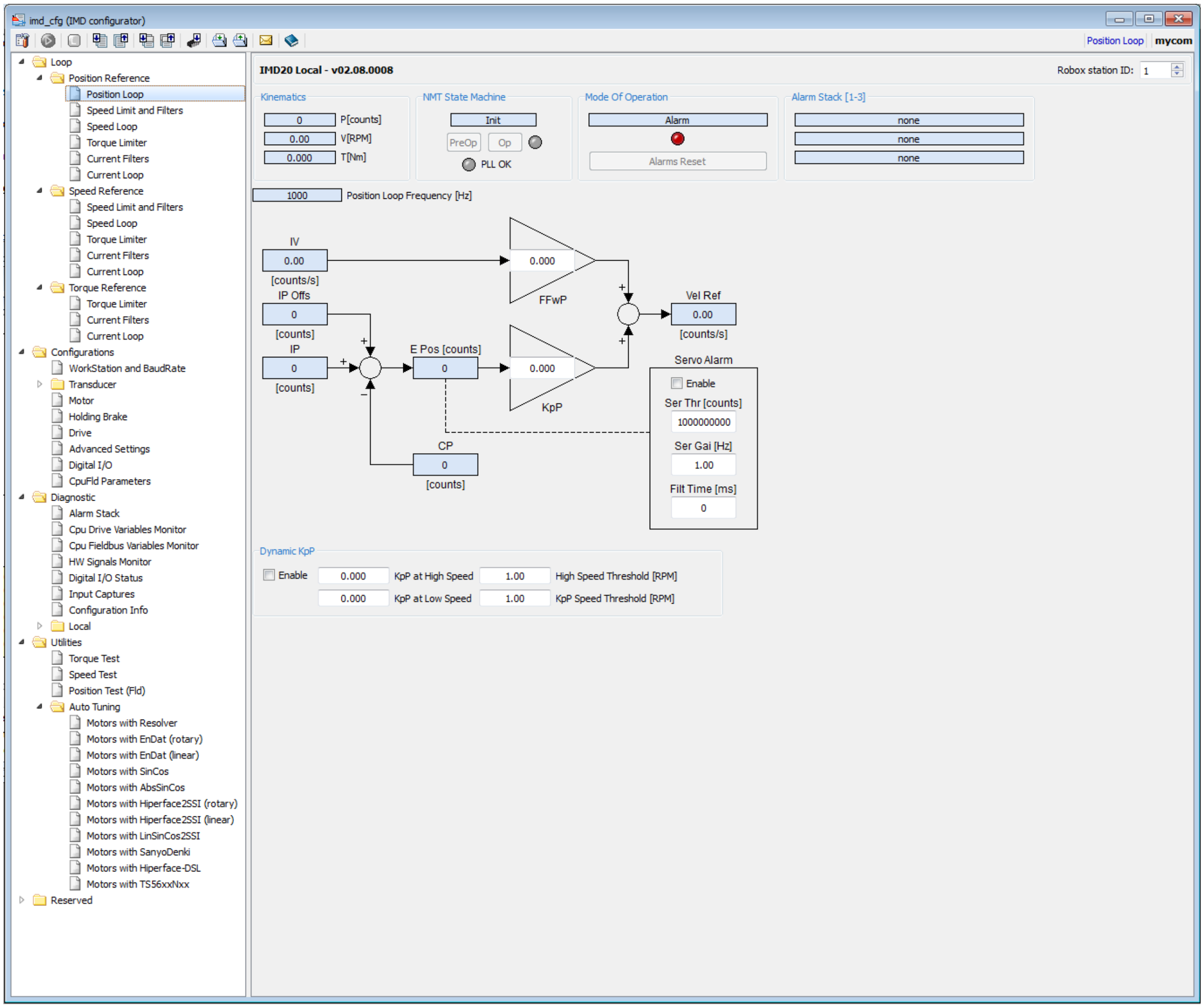


Fig. 63: Robox IMD20: Configurator

Fig. 64: Powerset

- `power_allowed` : Flag that enables all the PowerSets
- `ps_power_enable(POWER_SET ps, I32 flag)` : Enables the PowerSet `ps` if flag == 1.
- `ps_channel_enable(POWER_SET ps, I32 enableMask)`: Enables the drives of the PowerSet `ps`, where the drive relativ bit is 1. e.g. `enableMask=0x05` only axis(1) and axis(3) are enabled.

`POWER_SET` is a `STRUCT` to define a powerset. for example the field `eba` is a flag related to system alram, if it is `true`, it means there are no alram that forbide the power to be energized.

> Consult the documentation in RDE **RTE firmware –> power handling** and related arguments where you can find also a state machine of power handling.

RULEs

We prepare the base project, that let us to complie with errors. Remember that in order to compile a task, at least one instrution should be present.

Rules are similar to function that handle motion instruction. We can use different rules to manage an axis state machine, or we can use one rule where we can write the state machine directly. One RULE can handle up to 32 axis. In the rule body, the required axis are selected. The main strucure of a rule is:

```
RULE number
  axes n[,n,n,...,n]

  ref
    ; optional block of instructions containing the position loop closure algorithm.
  end_ref

  motion
       if(first_time())
          ; solo la prima volta
       endif
       ; block of instruction containing the algorithms to build the ideal trajectory
  end_motion

  aux
    ; optional block containing auxiliary instructions
  end_aux

END_RULE
```

If the rule is active, it is executed one time every period of the RULEs execution, one time every `si`. The `ref` and `aux` blocks are optional. So for now we don't discuss them. Remember the rule number is unique.

Let's write a code to move an axis in jog mode.

```
rule R_JOG
      ; axis 1
      axes(1)

      motion
      jog = bJogPos - bJogNeg ;
    ; MVA_JOG2(I32 ax, I32 direct, REAl speed, REAL accel, REAL decel)
      res = MVA_JOG2(axis_x, jog,      10,               100,        100,  1, 1, 0)
      end_motion

end_rule
```

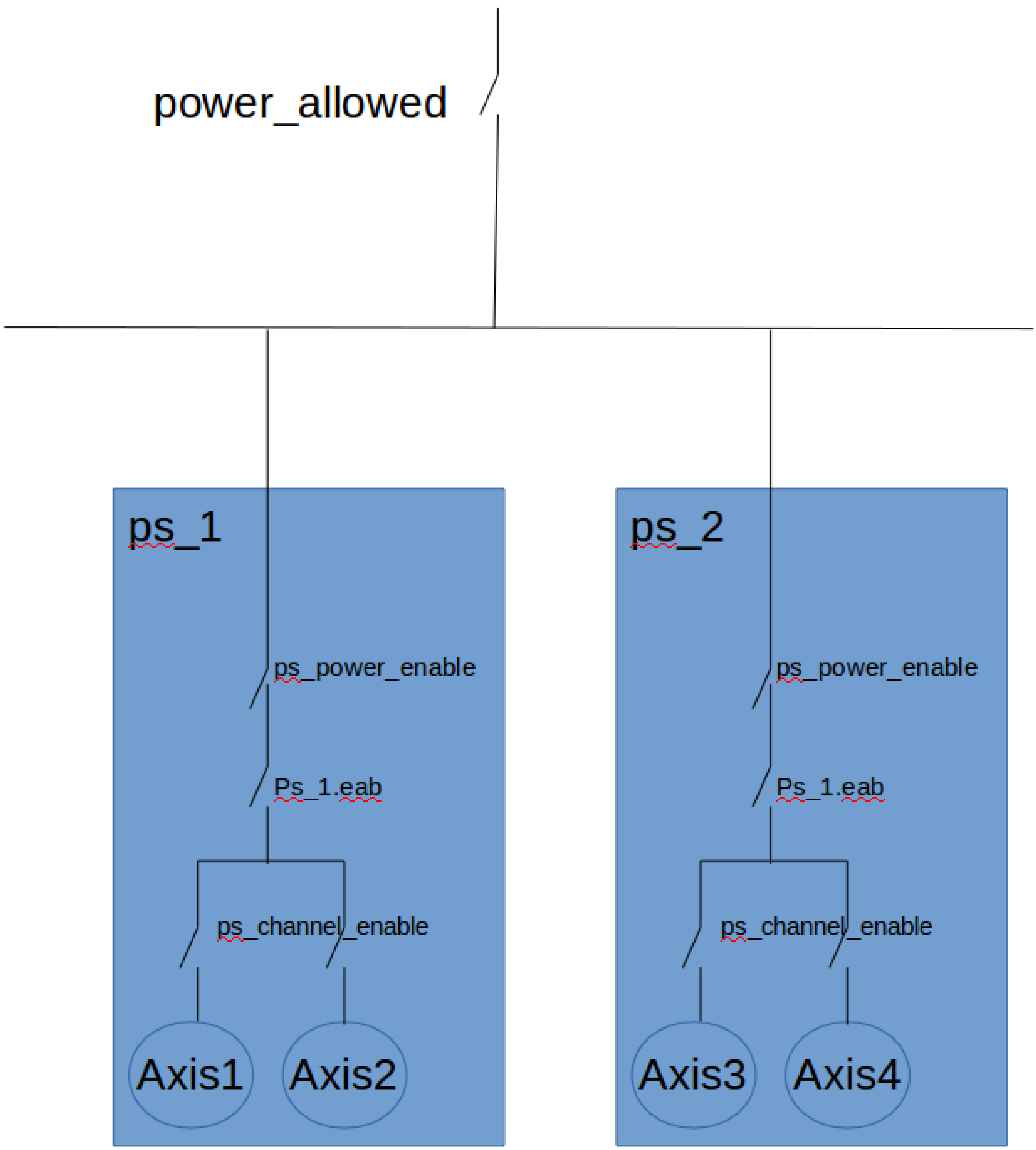


Fig. 65: Powerset enabling chain

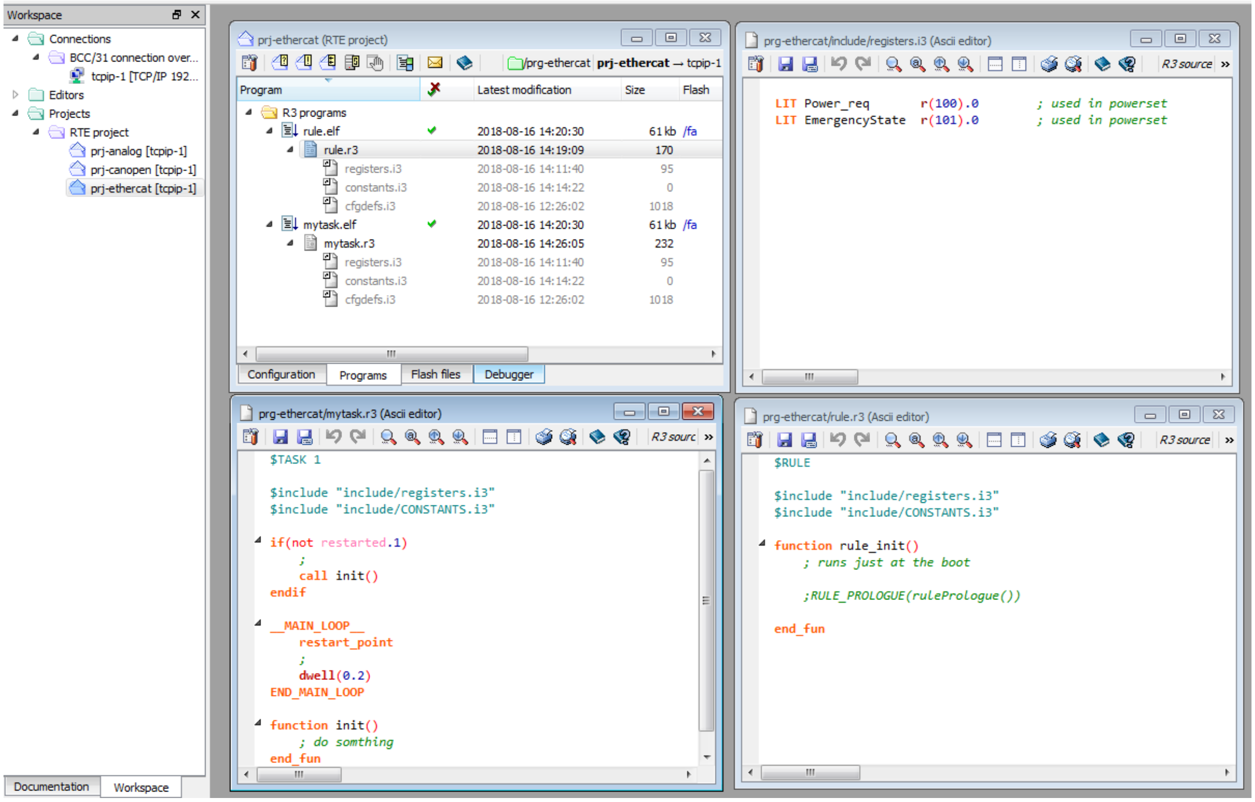


Fig. 66: Powerset enabling chain

Motion instruction can be found in R3 documentation. Check also the predefined variables in R3, related to motion control. This rule when active, handle the motion of axis 1 only, `axis(1)`. When `jog = 0` the axis doesn't move, when `jog=1` it move in positive direction when it is `jog=-1` it move in negative direction.

We can use different RULEs to assign different behavior to the axis. For example one Rule for jog motion, one for homing, one for positioning, etc.

We have two special rules, `RULE_PROLOGUE` and `RULE_EPILOGUE`. `RULE_PROLOGUE` is executed by RTE before the `motion` block of all active rules and `RULE_EPILOGUE` is executed after them.

`rule_init()` is a special function executed by RTE at the first execution to initialize the rule, e.g assign `RULE_PROLOGUE`.

Listing 3: rule_init()

```
function rule_init()
  ; if needed we enable the execution of the
  ; rule_prologue and rule_epilogue in the initialization rule
  rule_prologue(func_prologue)
  rule_epilogue (func_epilogue)
  ; do something
end_fun

function func_prologue
  ; do something
end_fun
```

(continues on next page)

(continued from previous page)

```
function func_epilogue
  ; do something
end_fun
```

First let's define some rule to manage an axis, then define a state machine to assign rules depending on the requirments. We begin to define some constants, to assign a name to rules:

Listing 4: RULE number definition

```
LIT R_POWER_MISSING   1
LIT R_FAST_STOP       2

LIT R_IDLE            3
LIT R_HOMING          4
LIT R_JOG             5
LIT R_POSITIONING     6
LIT R_AUTO            7
```

We will present some of the rules code, the complete code is found in the attached `demo:`.

Listing 5: Missing power rule

```
rule R_POWER_MISSING
      axes(1,2) ; only axis 1 and 2 is managed by this rule
      motion
         res = mva_open_loop(ax_x)
     res = mva_open_loop(ax_y)
      end_motion
end_rule
```

We can write also the axis name in the function `axis()`

```
axis(axis_x, axis_y)
```

The motion funtion `mva_open_loop(axis_number)`, assign the actual position to the ideal position `ip(axis_number) = cp (axis_number)`. This can be done when power is missing, to avoid any gap between the ideal position and the acual one, when the axis is powered and avoid any sudden motion.

Let's suppose we want to stop the axis in a controlled way when emergency circuit is opened, or to some grave error happen. So we have to ramp down the motion of the axis, ramp down the speed to zero, quickly and avoid sudden stop.

Listing 6: Fast stop, speed ramp down to 0

```
rule R_FAST_STOP
      axes(axis_x, axis_y)
      motion
              iv(axis_x) = ramp(iv(axis_x), 0, max_acc(1)) ; ramp down ideal speed to␣
↪0
    iv(axis_y) = ramp(iv(axis_y), 0, max_acc(1))
      end_motion
end_rule
```

Notice that we write only the ideal velocity or ideal position. Here we supposed that the drive close the velocity loop, and the deafult control loop of RTE is executed. More about this topic when we examine the `ref` block.

Now we write the code of the posiiotn rule. The rule have to move the axis to a defined poistion:

Listing 7: Rule move to a target position

```
rule R_POSITIONING
  axes(axis_x)
  motion
    ip(axis_x) = mv_to (MovResult, 1, lTarget, lSpeed, lAcc)
    if ( rise(MovResult = M_REACHED and similar(ip(1),lTarget,1) ) )
      ; do something
    end_if
  end_motion
end_rule
```

We already see how we can defined RULEs to do different things with axis, it is a simple concept. It is like writing functions to divide the tasks to do in a machine. We see also some motion instructions and how to used predefined variables related to motion control.

More motion control instructions can be found in the documention of R3 language.

Standard position loop algorithm

A rule contain the `ref` block that is optional, where control algorithms can be implemented. If the block is omitted, RTE will execute its standard, preimplemented control loop. In this case the controller will close the position loop, give a speed reference to the drive and the drive will close the speed loop and eventually other internal loops.

Listing 8: RTE standard closure of the proportional position loop with speed feed forward

```
; n is the axis index
epos(n) = p_ip(n,ipp_idx) - cp(n)
sref(n) = epos(n) * pro_gai(n) + kff(n) * iv(n)
```

RULEs execution

In our example we will enable one rule at one time. But in a multiaxis more complex machine we need to enable a group of axis, and maybe different rules at the same time. There are instruction to assign to RTE the execution order of the enabled order, in every sampling time.

The predefined variable `rule_length` give the execution time of what we call RULE. Remember that we don't mean a single rule, but the set of funtions, OB, etc. related to motion. So the time return by thi variable is the execution, let's say of everything exluding normal tasks. `rule_length` can't be bigger then `si`. Remember that in one period, `si`, RULES and some slices of tasks have to be executed. The rule frequency is set by the function `rule_freq (I32 freq)` or in RTE project configuration. The variable `si` is read-only. If the execution time is not enough, the rule frequency have to be increased.

Rules can be enabled when need, from a taks or from another rules. We can enable Rules, using different R3 functions. First we need to declare a variable of type `STRU_GROR` it is a `STRUC` which contain an array of `I32 idx[32]`. This strucure is used with the instructions `group(STRU_GROR rulegroup)` or `order(STRU_GROR rulegroup)`, and the meaning of each element depend on the instruction used.

RTE rule executer use a predefined variable `rc(n)`, which is an array of 32 elements. The value of `rc(n)` is a rule number. The standard order of rule execution is from index 1 until the last one. The order of execution can be changed using the instrction `order()`.

Let's make some example. First let's execute the rules in `rc` as are predefined in RTE, begining from the first one, so we don't use the instruction `order()`. In the follwing example, we define the strucure `rule_group`, then we

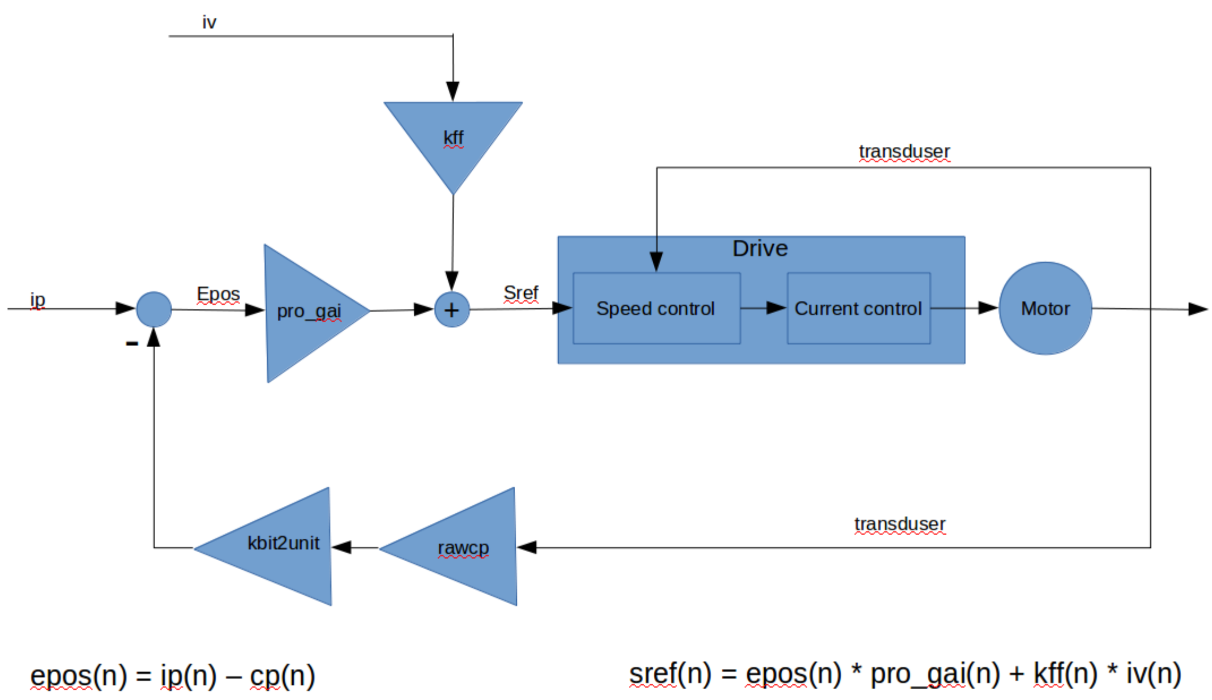


Fig. 67: RTE standard closure of the proportional position loop with speed feed forward

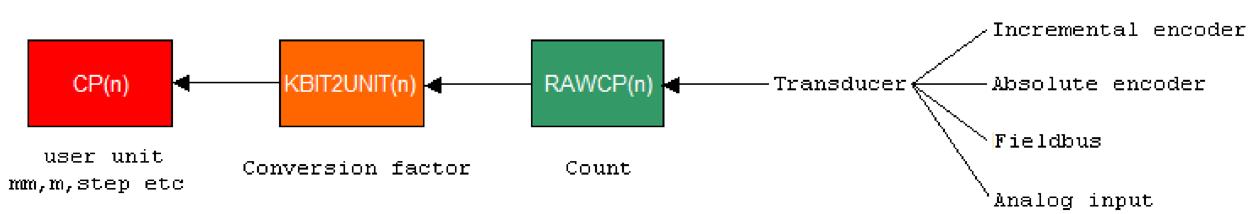


Fig. 68: Transduser count to physical unit convertion

assign the number of rule to be activated. The index of `idx` is the order of execution of the rules, that correspend to the index of `rc`.

Listing 9: Excution with normal order.

```
STRU_GROR rule_group

rule_group.idx[1] = 5 ; rule number , rc(1)=5
rule_group.idx[2] = 2 ; rule number
rule_group.idx[3] = 1 ; rule number
rule_group.idx[4] = 100 ; rule number

group(rule_group)
```

Now let' change the execution order of the rules. Let' execute in sequence `rc(4)`, `rc(1)`, `rc (3)` then `rc(2)`. The execution order will be `RULE 100`, `RULE 5`, `RULE 1` then `RULE 2`.

Listing 10: Execution user defined order

```
STRU_GROR rule_order
STRU_GROR rule_group

rule_order.idx[1] = 4 ; 4 is rc index
rule_order.idx[2] = 1 ; rc index
rule_order.idx[3] = 3 ; rc index
rule_order.idx[4] = 2 ; rc index

group(rule_group)

rule_group.idx[1] = 5 ; 5 is rule number , 1 is rc index. rc(1)=5
rule_group.idx[2] = 2 ; rule number
rule_group.idx[3] = 1 ; rule number
rule_group.idx[4] = 100 ; rule number

group(rule_group)
```

In summary, with the instruction `group` we assing rules to `rc`, e.g. `rule_group.idx[6] = 23` equivalent ot `rc(6) = 23`. And with the instruction `order` we change the order of exection of the `rc`, in other word we remap the indexes of `rc`.

We can assign `-1` to `rule_order.idx[n]`, in order to telle the executer that `n-1` is the last `rc` to be executed.

Listing 11: Execution user defined order

```
STRU_GROR rule_order
STRU_GROR rule_group

rule_order.idx[1] = 4 ; 4 is rc index
rule_order.idx[2] = 1 ; rc index
rule_order.idx[3] = 3 ; rc index
rule_order.idx[4] = -1 ;
```

Finally we assign rules to the ruel executer using directly the variable `rc`.

Listing 12: Execution user defined order

```
rc(1) = 2
rc(2) = 35
```

(continues on next page)

(continued from previous page)

```
rc(3) = 9
rc(4) = 1
rc(5) = 5
rc(6) = 7
```

Remember that task execution can interrupted, in order to execute another task or rule. This mean that, if the task is interrupted by the rules before the controller execute `rc(4)`, only rc(1), rc(2), rc(3) will be assinged to the rule executor. And the others will be assigned in the follwong sampling time. For this reason is better to use the instruction `group`, in this way all rules defined in `STRU_GROR` will executed.

Power

In another chapter we will examine a complete example, where we implement a state machine that enable rules depending on the machine requirment. In this section we will see some R3 instructions in order to deal with power handling.

In the initialization function, in `TASK1`, we need to enable the powerset and single axis:

```
ps_power_enable(ps,true)  ; enable powerset ps

; I32 ps_channel_enable (POWER_SET psname, I32 enableMask)
ps_channel_enable(ps,0x3) ; enable axis 1 and 2 in the powerset ps
```

We can invetigate the status of the powerset, using the instruction `I32 ps_status (POWER_SET psname)` that return:

```
0x00000001 (B0) at least 1 drive in fault
0x00000002 (B1) Powered
0x00000004 (B2) at least 1 drive enabled
0x00000008 (B3) all the powerSet drives are enabled
0x00000010 (B4) delayed (if required) state of the feedback
0x00000020 (B5) reserved
0x00000040 (B6) counting running in case of delay because of axis alarm causing the␣
↪power drop (power_off_delay_on_alarm)
0x00000080 (B7) counting running in case of delay because of feedback lack (power_off_
↪delay_on_no_feedback)
0x00000100 (B8) actual feedback state (not delayed)
0x00000200 (B9) the feedbacks for all the drives are present
0x00000400 (B10) the feedbacks of all the drives for which the enable command has␣
↪been activated, are present (instruction ps_channel_enable)
B11-B32 Reserved
```

For example we want to see

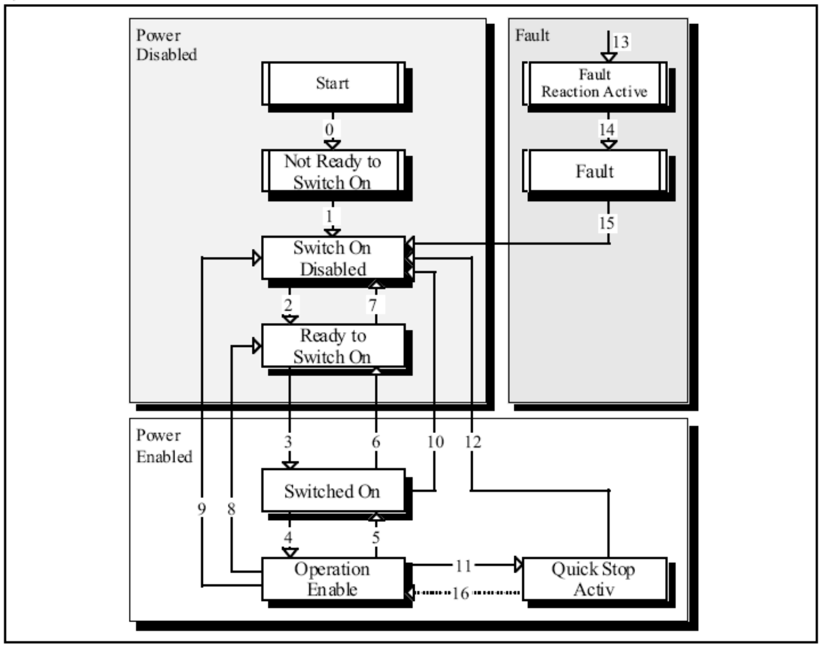
```
 if ( ps_status(ps) r_and 0x8)
  ; all the powerSet drives are enabled
 end_if

systemPowered = ((ps_status(ps) r_and 0x12) = 0x12) ; x012= 0x10 OR 0x02

powerGoingDown = (ps_status(ps) r_and 0xC0) ; C0 = 0x40 OR 0x80
```

Summary

TODO

1.1.13 Base project - State machine

The state machine in Figure describes the state machine of the device with respect to control of the power electronics as a result of user commands and internal drive faults.



Bit	Description	M /O
0	Ready to switch on	M
1	Switched on	M
2	Operation enabled	M
3	Fault	M
4	Voltage enabled	M
5	Quick stop	M
6	Switch on disabled	M
7	Warning	O
8	Manufacturer specific	O
9	Remote	M
10	Target reached	M
11	Internal limit active	M
12 - 13	Operation mode specific	O
14 - 15	Manufacturer specific	O

Syntax:	**I32 fb_stsw(nax)**
nax:	axes number
Note:	The update can be disabled through BIT 2 of the variable FB_FLAG

State machine

Homing

R3 Motion instructions

Note: Predefined examples on the use of R3 motion instructions are available in RDE

1.1.14 Control algorithm

Overview

todo

Algorithm

Ref

```
REF
  ; algorithm
END_REF
```

List of all the instructions/functions available with the R3 programming language to control the axes movement.

See the library **RPE (Robox path executor)** if a group of axes is used

Legend: R=allowed with Rules - T=allowed with Tasks - I=Instruction - F=Function (therefore returns a value)

Keyword	R	T	I/F	Description
mv_cam()	R	-	F	Executes a cubic cam described with a table of segments a,b,c
mv_crimper()	R	-	F	Crimper function for flow-pack
mv_follow()	R	-	F	Flying shear
mv_follow2()	R	-	F	Flying shear (advanced)
mv_mot_exec()	R	T	F	Interpolation function through the MOT table (multiple output table)
mv_phase_adj()	R	-	F	Phase correction between two axes
mv_phase_adj2()	R	-	F	Phase correction between two axes (advanced)
mv_ramp()	R	T	F	Ramps the controlled variable to a target value
mv_reach_target()	R	-	F	Docking between two axes
mv_sinecam()	R	-	F	Cam execution (y versus x)
mv_synchro()	R	-	F	Docking between two axes
mv_table()	R	-	F	Executes a cam described with a table of points
mv_tracking()	R	-	F	Tracks a target with specified position and speed
mv_to()	R	-	F	Movement towards a target
mv_to_vel()	R	-	F	Movement towards a target at a programmed final speed
mv_to_cj()	R	-	F	Movement towards a target with controlled jerk (parameters acquired only at the start)
mv_to_cjv()	R	-	F	Movement towards a target with controlled jerk (parameters run-time modifiable)
mv_to_cjv_info()	R	-	F	Auxiliary function to MV_TO_CJV
mva_jog()	R	-	F	JOG movement (manual)
mva_jog2()	R	-	F	JOG movement (manual) with controlled jerk
mva_open_loop()	R	-	F	Open loop (missing power RULE)
mva_to_n()	R	-	F	Movement of more axes towards a target
mva_to_n_v()	R	-	F	Movement of more axes towards a target with initial speed <>0
mva_to_n_cj()	R	-	F	Movement of more axes towards a target
mva_to_n_v()	R	-	F	Movement of more axes (even already in movement) towards a target
mva_zc()	R	-	F	Homing
ramp()	R	T	F	same as mv_ramp
kin_conv()	R	T	F	Calculates the acceleration and jerk values
tracking()	R	-	F	same as move_tracking

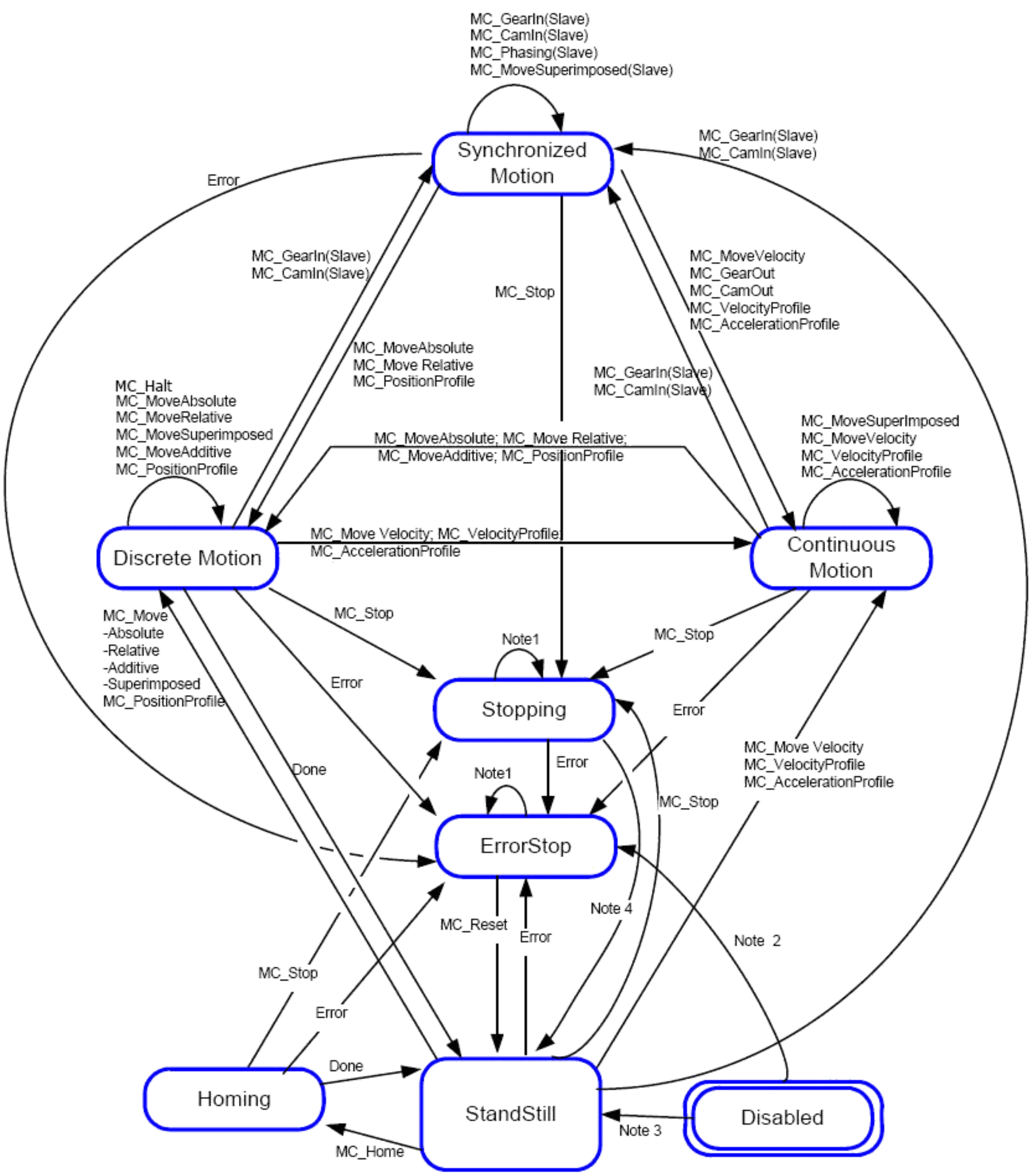
1.1.15 Object block

Overview

todo

1.1.16 PLCopen

IEC 61131



1.1.17 Drives

Overview

todo

1.1.18 RPE - Robox path executor

RPE is an extension of RTE. In order to use it, the binary file `rpe.bin` should be copied in `/f@`. The binary file of the platform you are using can be downloaded from Robox website.

RPE is used to develop robotic applications, using different models of robots: cartesian, anthropomorphus, parallel, etc.

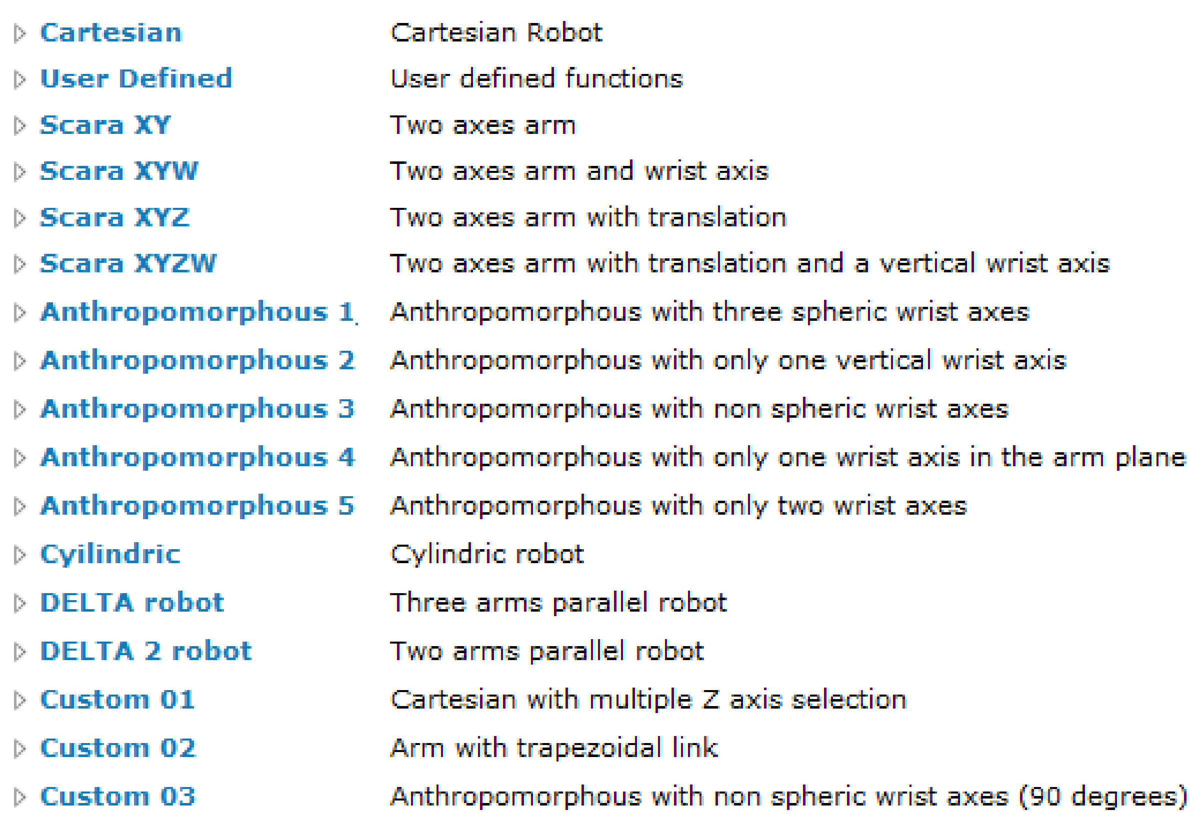
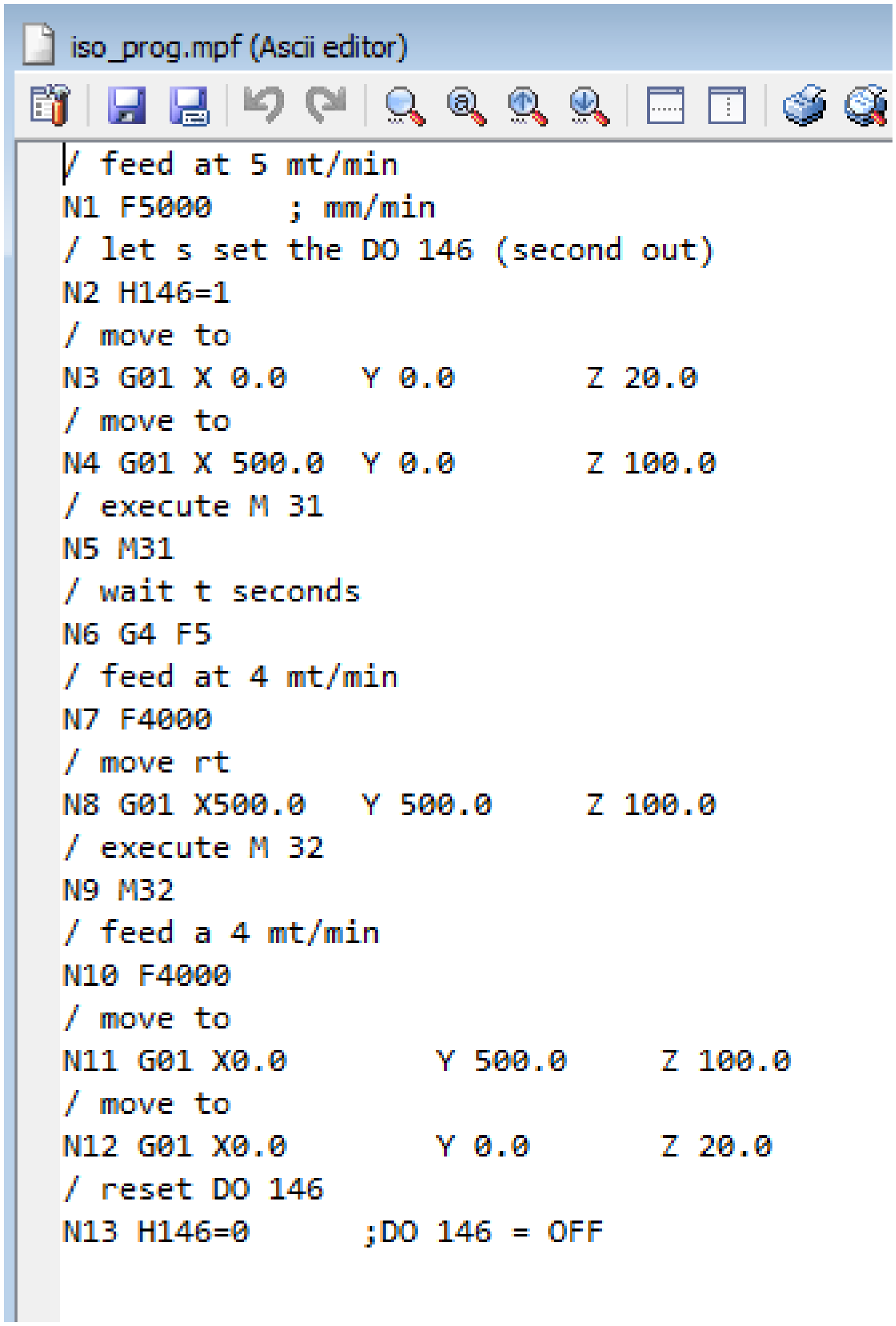


Fig. 69: Joint-cartesian models

The path to be executed can be defned through ISO language (G-code) with one of these extensions `.mpf`, `.iso`, `.nc`, or in a path class file with extension `.pth`.

Axis group

In the following animation we show how to create a set of axes. we don't show how to add all the parameters of the axes. Consult the predefined example **RPE: pick and place**.


iso_prog.mpf (Ascii editor)

```
/ feed at 5 mt/min
N1 F5000    ; mm/min
/ let s set the DO 146 (second out)
N2 H146=1
/ move to
N3 G01 X 0.0    Y 0.0       Z 20.0
/ move to
N4 G01 X 500.0  Y 0.0       Z 100.0
/ execute M 31
N5 M31
/ wait t seconds
N6 G4 F5
/ feed at 4 mt/min
N7 F4000
/ move rt
N8 G01 X500.0   Y 500.0     Z 100.0
/ execute M 32
N9 M32
/ feed a 4 mt/min
N10 F4000
/ move to
N11 G01 X0.0        Y 500.0     Z 100.0
/ move to
N12 G01 X0.0        Y 0.0       Z 20.0
/ reset DO 146
N13 H146=0      ;DO 146 = OFF
```

Fig. 70: Path defined in a file using G-code

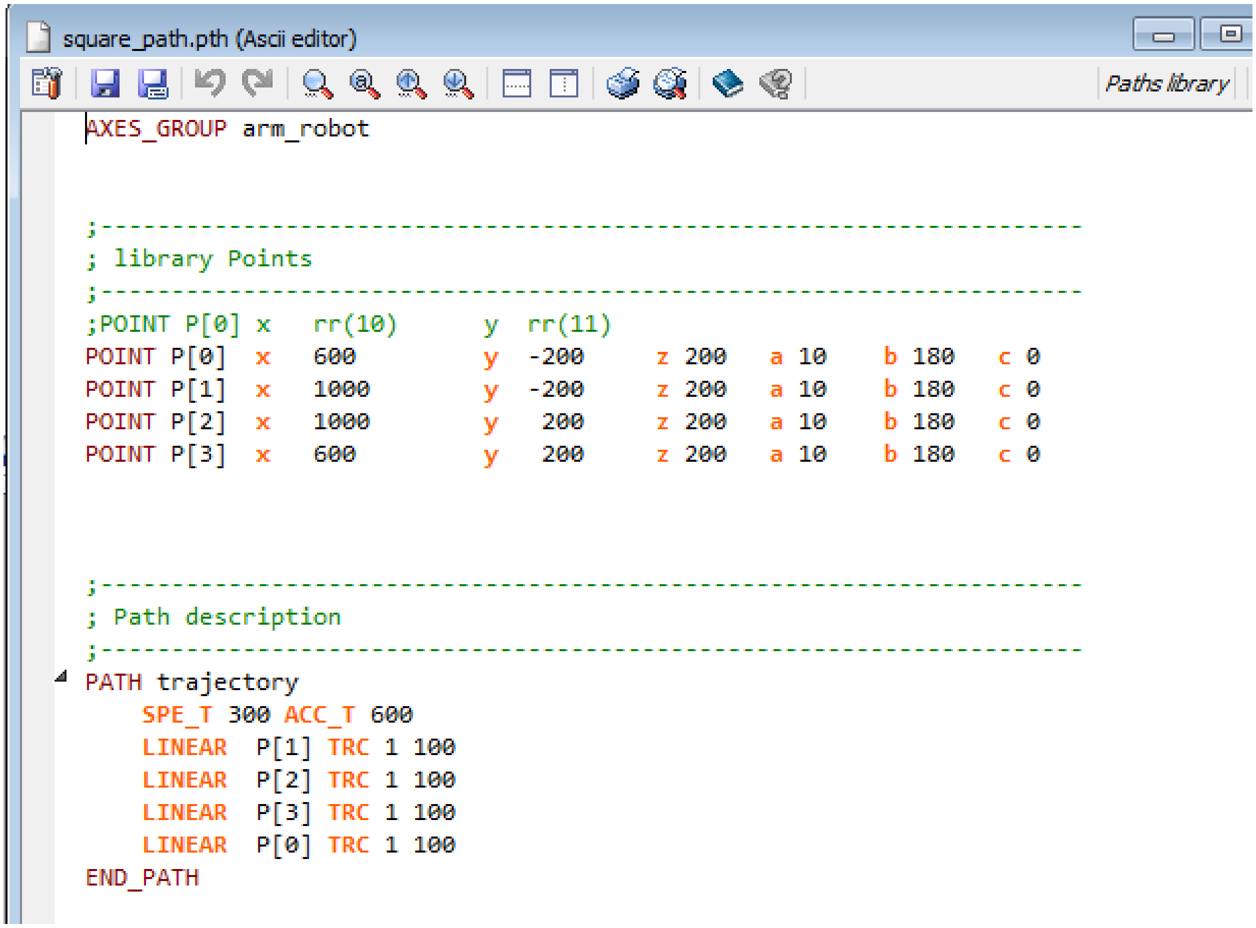


Fig. 71: Path defnied using Path Library

Fig. 72: Set of axes

Library points and paths

We will see how to define a square using library `POINT` and `PATH` defintion.

Listing 13: Library points and paths basic structure

```
AXIS_GROUP axis_group_name

POINT point_name
  ;
END_POINT

PATH path_name

END_PATH
```

We define the 4 vertices of the square in the space with coordinate (x,y,z,a,b,c). Using the path block we connect points via straight lines. These points and paths belong to the axis group defined in the file.

```
AXES_GROUP arm_robot


;---------------------------------------------------------------------
; library Points
;---------------------------------------------------------------------
;POINT P[0]        x         rr(10)         y  rr(11)
POINT P[0]        x              600                          y  -
↪200          z 200          a 10          b 180          c 0
POINT P[1]        x         1000                   y  -200               z 200         a␣
↪10        b 180         c 0
POINT P[2]        x         1000                    y   200                z␣
↪200        a 10         b 180         c 0
POINT P[3]        x         600                    y          200                z␣
↪200        a 10         b 180         c 0



;---------------------------------------------------------------------
; Path description
;---------------------------------------------------------------------
PATH trajecto300 ACC_T 600
        LINEAR         P[1] TRC 1 100
        LINEAR         P[2] TRC 1 100
        LINEAR         P[3] TRC 1 100
        LINEAR         P[0] TRC 1 100
END_PATH
```

The name of the points in the file `.pth` must be the same of the variable of type `POINT_L` declared in the task. Also the path name must coincide with the task varaible of type `PATH`.

Example files describing library points and paths :

- `Square`
- `Points`
- `Profile`

G-code

G-code is a standardized programming language used in CNC machines like lathe, mill, etc. G-code can written manually for simple manufacturing or generated automatically by CAD softwares for complex machining.

The most used code are G and M. G are preparatory commands usally for motion, and M are miscellaneous functions.

```
G00 Rapid positioning
G01 Linear interpolation
G02 Circular interpolation, clockwise
G03 Circular interpolation, counterclockwise
G21 programming in mm

M02 End of program
M03 Spindle on (clockwise rotation)
M04 Spindle on (counterclockwise rotation)
M09 Coolant off
```

In RPE documentation, you can find a list of code supported by RPE.

using in RPE . todo

G-code examples :

- `Path`
- `Logo`
- `cr028`
- `iso_prog`

Operative mode

RPE structures

RPE functions

1.1.19 RPE : Robot

Overview

todo

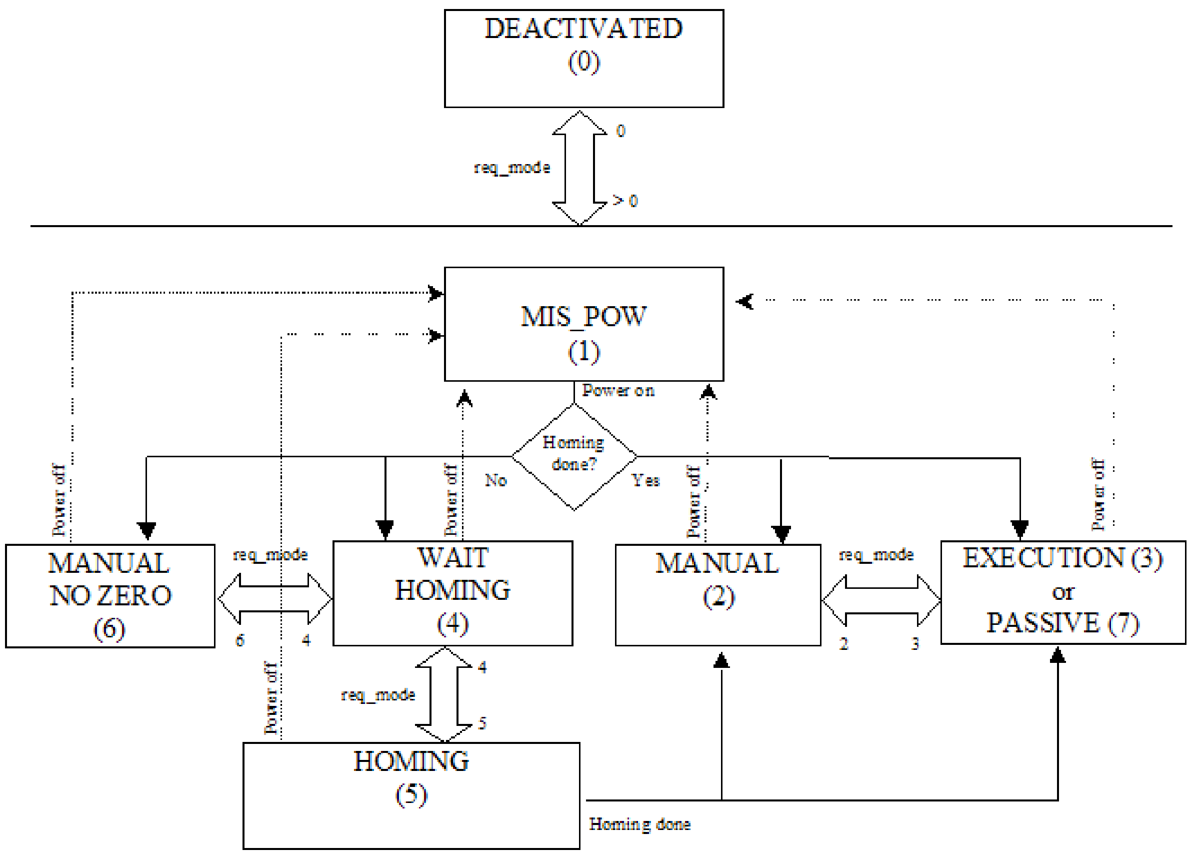
1.1.20 RPE : ISO

Overview

todo

Generate G-code from inkscape

inkscape


DEACTIVATED
(0)
req_mode
0
> 0
MIS_POW
(1)
Power on
Homing
done?
No
Yes
Power off
Power off
Power off
Power off
Power off
MANUAL
NO ZERO
(6)
req_mode
6
4
WAIT
HOMING
(4)
MANUAL
(2)
req_mode
2
3
EXECUTION (3)
or
PASSIVE (7)
4
req_mode
5
HOMING
(5)
Homing done

The predefined functions available in RPE are the following:

Axes group handling

▷ **pe_exec_path**	Launches path execution
▷ **pe_move_cart**	Launches a Cartesian linear movement execution
▷ **pe_move_circ**	Launches a Cartesian circular movement execution
▷ **pe_move_joint**	Launches a Joint movement execution
▷ **pe_fly_cart**	Cartesian movement with automatic calculation of acceleration and deceleration
▷ **pe_fly_joint**	Joint movement with automatic calculation of acceleration and deceleration
▷ **pe_fly2_cart**	Cartesian movement with automatic calculation of acceleration and deceleration
▷ **pe_fly2_joint**	Joint movement with automatic calculation of acceleration and deceleration
▷ **pe_get_move_info**	Reads informations about launched moves.
▷ **pe_abort_path**	Cancels paths execution
▷ **pe_reset_stop**	Resets first stop flag
▷ **pe_reset_stop_mode**	Resets first stop flag with restart mode
▷ **pe_set_tool**	Sets the transform from wrists Cartesian system to tool
▷ **pe_get_tool**	Gets the transform from wrists Cartesian system to tool
▷ **pe_set_coord**	Sets the transform from basic Cartesian to work system
▷ **pe_get_coord**	Gets the transform from basic Cartesian to work system
▷ **pe_distance**	Distance of the axes group from the specified point
▷ **pe_push_event**	Inserts the code in the events stack
▷ **pe_pop_event**	Takes the first code of the events stack
▷ **pe_set_transform**	Sets the transform to apply to Cartesian movements
▷ **pe_get_transform**	Gets the transform to apply to Cartesian movements
▷ **pe_set_position**	Sets the quotas of the axes group
▷ **pe_get_position**	Gets the quotas of the axes group
▷ **pe_get_path_exec_pos**	Reads the current positions of the path execution
▷ **pe_stop_moves**	Stops current move
▷ **pe_stop_path**	Stops current path execution
▷ **pe_save_path_exec_status**	Saves the current path execution status
▷ **pe_restore_path_exec_status**	Restores the current path execution status
▷ **pe_set_path_reverse_exec**	Sets the reverse path execution status
▷ **pe_push_event_par**	Inserts the code with an associated parameter in the events stack
▷ **pe_pop_event_par**	Takes the first code of the events stack with an associated parameter
▷ **pe_move_single_axis**	Single axis movement (auxiliary or joint)
▷ **pe_mj_convert**	Conversion from motor positions to joint positions and vice versa.
▷ **pe_jc_convert**	Conversion from joint positions to Cartesian positions and vice versa.
▷ **pe_set_joint_to_cart_function**	Sets the direct conversion function for User Defined Model.
▷ **pe_set_cart_to_joint_function**	Sets the inverse conversion function for User Defined Model.
▷ **pe_set_copy_storage_function**	Sets the retentive storage copy function for User Defined Model.
▷ **pe_start_tracking**	Request to start tracking handling.
▷ **pe_stop_tracking**	Request to stop tracking handling.
▷ **pe_set_linear_track_pos**	Sets linear user tracking position.
▷ **pe_set_user_track_tr**	Sets user tracking transform.

Libraries handling

▷ **pe_load_lib_path**	Loads points and paths from text files

Points handling

▷ **pe_get_point_c**	Converts library point to Cartesian point
▷ **pe_get_point_j**	Converts library point to joint point
▷ **pe_set_point_c**	Sets library point data from Cartesian point
▷ **pe_set_point_j**	Sets library point data from joint point
▷ **pe_set_model**	Transformation joint point to Cartesian and vice versa

Paths handling

▷ **pe_load_from_iso_file**	Converts an ISO program into a path
▷ **pe_clear_path**	Clears all the segments of the path
▷ **pe_add_linear**	Adds a linear segment to the path
▷ **pe_add_spline**	Adds a spline segment to the path
▷ **pe_add_circle_te**	Adds a circular segment to the path
▷ **pe_add_circle_3p**	Adds a circular segment to the path passing through an intermediate point
▷ **pe_add_circle_tu**	Adds a circular segment to the path tangent to the next segment
▷ **pe_add_joint**	Adds a joint segment to the path
▷ **pe_add_event**	Adds an event with the code *code* to *segm*.
▷ **pe_add_event_par**	Adds an event with the code *code* and parameter *par* to *segm*.
▷ **pe_set_stop_angle**	Angle between outgoing and incoming tangent
▷ **pe_set_stop**	Stop program at the end of segment
▷ **pe_set_trs**	Sets a link through SPLINE
▷ **pe_set_trc**	Sets alink through CIRC
▷ **pe_path_compile**	Compiles a path (without executing it)
▷ **pe_path_point_from_pos**	Calculates a Cartesian point on the path
▷ **pe_path_set_point_c**	Writes a path point
▷ **pe_path_get_point_c**	Reads a path point
▷ **pe_path_get_point_l**	Reads a path library point id
▷ **pe_path_set_model**	Associates an axes group to a path

Transform handling

▷ **pe_reset_tr**	Resets the transform
▷ **pe_invert_tr**	Inverts the transform
▷ **pe_translate**	Translation along three absolute Cartesian axes
▷ **pe_rotate_x**	Rotation of angle along the absolute X axis
▷ **pe_rotate_y**	Rotation of angle along the absolute Y axis
▷ **pe_rotate_z**	Rotation of angle along the absolute Z axis
▷ **pe_translate_rel**	Translation along three transformed Cartesian axes
▷ **pe_rotate_x_rel**	Rotation of angle along the transformed X axis
▷ **pe_rotate_y_rel**	Rotation of angle along the transformed Y axis
▷ **pe_rotate_z_rel**	Rotation of angle along the transformed Z axis
▷ **pe_transform_tr**	Transformation of the transform *tr2* according to *tr*
▷ **pe_transform_pc**	Transformation of a point *pt* according to *tr*
▷ **pe_transform_from_3p**	Compute transformation given 3 points
▷ **pe_transform_from_pc**	Compute transformation given 1 point
▷ **pe_tr_offs_from_4pc**	Compute transformation offsets given 4 points

G-code from CAD

Autocad

G-code libraries

C++, C#

1.1.21 Serial communication

RS232

RS485

1.1.22 UDP sockets

Sockets are used to communicate between different processes on the same machine or on different ones. There are different types of sockets. In this chapter we will deal with **Datagram sockets**. Datagram sockets use UDP **User Datagram Protocol**.

CLient-Server architecture is usually used between 2 applications in order exchange information. Typically a client make a request for information to a server. A client have to know about the exitence and the address of the server. The server typically answer to some request, and doesn't need to know about the existence the client prior to the connection being established.

For our purpose, in order to establish a communication we need a `hostname` and a `port`. A hostname could be a string or an ip address. A port is where a server listens for client's calls. It is recommended, when using Robox products, to use port nember outside the range [8000, 8999] to avoid conflicts with Robox's implementations.

R3: UDP Client/server

In **RDE predefined examples** you can find the demo **R3: UDP client/server**, it is a workspace with two RTE projects, one for server and one for client. In order to test this demo you need 2 Robox controllers.

Fig. 73: RDE predefined example: R3 UDP Client/server

`This is a modified version` of the sample, it is configured to work with two **RP1**. The first with address `192.168.1.130` act as server, the second RP1 act as client and have the address `192.168.1.131`.

Each UDP datagram is charecterized by a length, the length of a datagram is send along with the data.

UDP server

In the server code we will use the following R3 functions:

- `str_to_ipaddr( ipstring, ipnum)` : convert an ip address string to it's hexadecimal equivalent. e.g. `192.168.1.130` will be `0xC0A80182`
- `udp_open_server(server_port, server_address)` : open a server with assigned port and address, and return a handle of the socket.

- `udp_recv_from(sockethandle, buffer, buff_size, remote_port????, client_address)`
- `udp_send_to (sockethandle, buff, buff_size, remote_port, client_address)`

The `buffer` to be send and received can be a `STRING`, an `array` or a `struct`. If it is a `struct` the first member should be an `U32` that represents the message identifier, its value is fixed by the client which sent the request through the function udp_send_notify(). The remaining members are user defined, e.g. :

```
STRUCT_P buff_send
  U32 msgId ; message identifier
  I32 comando
  I32 reg_start
  I32 reg_num
END_STRUCT_P
```

The server in this example, have to send registers value to a client. It receive a command, `udp_recv_from`, that represent the register type to be sent. Once the command received, if the the comand is know, the server build a buffer with the data of the requested registers and send them to the client, `udp_send_to`.

First a command buffer is built:

```
STRUCT_P buff_send
  U32 msgId
  I32 comando ; command, register type
  I32 reg_start ; starting index of the register
  I32 reg_num   ; how many registers (number of repetitions)
END_STRUCT_P
```

Codification of register types:

```
LIT REQ_R                1
LIT REQ_RR             2
LIT REQ_NVR          3
LIT REQ_NVRR         4
```

Then a buffer to be sent to the client is constructed:

```
STRUCT_P buff_recv
      I32  msgId
      I32  comando
      I32  reg_start
      I32  reg_num
      REAL regs[30]
END_STRUCT_P
```

A server is opened using the function `udp_open_server()`. Data are received `udp_recv_from`, then depending on the command the buffer to be sent is filled with registers value, then sent to the client `udp_send_to`.

The complete code can be found in the attached project.

UDP client

In the client code we will use the following R3 functions:

- `str_to_ipaddr( ipstring, ipnum)` : convert an ip address string to it's hexadecimal equivalent. e.g. `192.168.1.131` will be `0xC0A80183`

- `udp_open_client(server_port, server_address)` : open a client communication, and return a handle of the socket.
- `UDP_SEND_NOTIFY(sockethandle, buff, buff_size)` : sends a UDP message with notification
- `UDP_RECV_NOTIFY(sockethandle, buff, buff_size, tmOutRx, tmOutTot)` : receives a UDP message (notified) from a previously selected station

The `buffer` can be a `STRING`, an `array` or a `struct`, see discussion in server section.

Appliaction Client

A client application will be developped that run on a personal computer. The *UDP server* code is that same we see before and will be running on Robox controller e.g. RP1 with address `192.168.1.130`.

We will develop a client similar to the one of RDE predefined example in *UDP client*.

Python

C++

1.1.23 BCC communication protocol

1.1.24 Fundamental of automation

Overview

Basics of electronics

Sensors

Actuators

Controllers

Some Examples

1.1.25 External editors

In order to write a program, you can use the internal text editor provided by AgvManager and RDE. You can use also external editors, the one you like. RDE support 3 external editors, this mean that in the configuration window, you can choose to open the source code in an external editor. Notepad++, UltraEdit and ConTEXT are supported by RDE.

In the following section we will see how we make configuration files in order to highlight the syntax of Xscript, R3 language and object blocks.

Vim

File needed??

Copy the files in vimfile in the instalation directory in windows or in /usr/share/vim/vimfiles in linux

Syntax highlight

Function list

Notepad++

File needed and where to place them

Regular Expressions

Notepad++ regular expressions use the standard PCRE (Perl) syntax.

Syntax highlight

Function list

UltraEdit

File needed and where to place them

Regular Expressions

UltraEdit doesn't use Unix style regex. There are some difference between the two styles. On the website of UltraEdit, we can find the difference between them. In the wordfile of UltraEdit regex of UltraEdit should be used, it is different from the one used in Notepad++.

Function list and syntax highlight

1.1.26 Version control

Note: Don't modify this file in RDE-Doc. Update the file in personal notes, then copy it here.

Git

Basics

installation

Create new repository

```
git init
git config --global user.email "abdo_sarter@hotmail.com"
git config --global user.name "Abed"
```

Clone existing repository

Clone repository

```
git clone https://github.com/abedGNU/QtSnap7.git
```

Commit and versioning

Push and pull

Branching

Github pages