



Robox notes

Work in progress 2018-06-27

Robox motion control



Copyright © 2018 Abed Ramadan

2018-06-27

ABED@ROBOX.COM.CN

ROBOX.IT

www.gnu.org/licenses/gpl.html

Contents

1	Introduction	9
I	AGV Manager	
2	RAT: Robox Agv Tool	15
2.1	RAT	15
2.2	Map	15
2.2.1	Vehicle	16
2.2.2	Lines	17
2.2.3	Generic point	18
2.2.4	User point	19
2.2.5	Battery point	20
2.2.6	Magnet point	20
2.2.7	Start point	21
2.2.8	Cross	22
2.3	Tips	23
3	AGV Manager	27
3.1	Overview	27

3.2	Installation	27
3.3	AGV configurator	29
3.4	AgvManager interface	29
3.4.1	AGV emulation	31
3.4.2	Point windows property	32
3.5	AGV script executing	33
3.5.1	Fundamental concepts	33
3.5.2	Main loop execution	34
3.5.3	Mission execution	35
3.6	Drag and drop example	35
3.7	Summary	45
4	More examples	53
4.1	Xscript Agv Data structures	53
4.1.1	XMapParams	54
4.1.2	XVehicleInfo	55
4.1.3	XSiteInfo	57
4.1.4	XListaSiti	59
4.2	Some useful functions	60
4.2.1	Movement functions	60
4.2.2	MICRO registration functions	60
4.2.3	Points	61
4.3	Creating functions and objects	61
4.4	Ex 01: Drag and drop example with loading and loading operations	62
4.5	Overview on some functions	70
4.5.1	Access level	70
4.5.2	onUpdateIO()	70
4.5.3	OnAgvStatusChange(): Agv status flag change	71
4.5.4	OnAgvModeChange(): Agv Operating mode change	71
4.5.5	Settings: XSettings	72
4.5.6	Interaction with user: XForm	72
4.5.7	Semaphores	72

4.6	Ex 02: Storage example	73
4.6.1	Missions	73
4.6.2	onNextMission()	74
4.6.3	Macros	77
4.6.4	Micros and Operations	80
4.6.5	Store handling	83

II

Motion controller RP-1

5	RTE - Real Time Extended	89
5.1	Multitasking	89
6	RDE - Robox Development Environment	93
6.1	First step	93
6.2	New RTE project	93
6.3	Tools	93
6.3.1	Command shell	93
6.3.2	Oscilloscope	97
6.3.3	Graphic panel	97
6.4	Hardware configuration	97
6.4.1	Important folders and files	97
6.4.2	Registers	98
6.4.3	Bus configuration	98
6.4.4	Axes configuration	98
6.5	Programming languages	98
6.5.1	R3	98
6.5.2	Predefined variables	98
6.5.3	Object block	99
6.6	Programs	102
6.6.1	Tasks	102
6.6.2	Rules	102
6.6.3	R3 example program	102
6.6.4	OB example program	102
6.7	Axis control example	102
6.7.1	Axis configuration	102

6.7.2	Power set and power handling	102
6.7.3	Velocity control	102
6.7.4	Position control	102

7 Motion control 107

7.1	Basics	107
7.2	Example with R3	107
7.3	Example with OB	107

8 AGV commissioning 109

8.1	MGV mechanical and electrical part	109
8.2	Dispan : Display panel	110
8.3	Commissioning	110
8.3.1	Potentiometer	111
8.3.2	Magnetic sensors and Canopen modules	115
8.3.3	Laser scanner	115
8.3.4	AGV parameters	116
8.3.5	Register backup	117
8.3.6	General definitions	118
8.4	AGV ecosystem	119
8.4.1	Software and tools	119
8.5	AgvManager-RP1 communication protocol	119
8.6	Hardware configuration	119
8.7	Program description	119

III

Appendices

9 External editors 123

9.1	Vim	123
9.1.1	Syntax highlight	123
9.1.2	Function list	123
9.2	Notepad++	123
9.2.1	Regular Expressions	123
9.2.2	Syntax highlight	124

9.2.3	Function list	124
9.3	UltraEdit	124
9.3.1	Regular Expressions	124
9.3.2	Function list and syntax highlight	124

IV

Bibliography



1. Introduction

TODO



AGV Manager

2	RAT: Robox Agv Tool	15
2.1	RAT	
2.2	Map	
2.3	Tips	
3	AGV Manager	27
3.1	Overview	
3.2	Installation	
3.3	AGV configurator	
3.4	AgvManager interface	
3.5	AGV script executing	
3.6	Drag and drop example	
3.7	Summary	
4	More examples	53
4.1	Xscript Agv Data structures	
4.2	Some useful functions	
4.3	Creating functions and objects	
4.4	Ex 01: Drag and drop example with loading and loading operations	
4.5	Overview on some functions	
4.6	Ex 02: Storage example	

To manage an AGV using Robox products, 3 components are needed: a map, motion control and plant specific logic.

Maps are drawn using RAT software then converted to an ASCII file with *.map* extension. This file is used by other two softwares: AGV manager and RDE.

AGV manger has two main parts: script and core. The specific logic of a plant is written in a script using XScript language, and given to the core that execute it. The core handle also the communication with the motion controller and eventually a plant PLC or database.

RDE is Robox IDE for motion control programming. The map is compiled by ICMaP and read by the RTE. This part will be explained in other chapters.

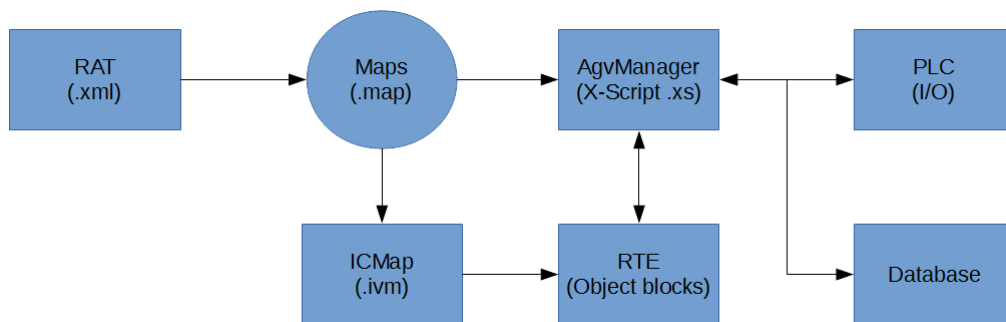


Figure 1.1: AGV block diagram. RAT give a *.map* file as output. The map is read by AGV manager. The map is comiled by ICMaP then read by RTE. AGV manager may communicate with a PLC or a database as an interface to the plant IO, and with RTE for motion control.

In the following chapters we will explain Robox software in order to draw a map and assign missions to Agv. Three softwares are needed : RAT, AgvConfigurator and AgvManager. Note that maps can also be edited by a text editor. AgvConfigurator is a part of AgvManager and are installed together.



2. RAT: Robox Agv Tool

2.1 RAT

RAT is a CAD software, fig.2.1, aimed to design maps and convert them into a formatted ASCII file with *.map* extension. RAT save the created files as *xml file*. A map can also be created using a text editor following some rules.

RAT can load *dxf* files as background, that can be used as a guide to design the desired map. Mainly RAT have lines, points, vehicles. These component will be explained later.

In the properties of the project some settings have to be changed in order to change the behavior of the AGV motion, for example *Trasversal navigation* is set by default to *disabled*. When it is enabled the AGV can move trasversally to a line, and options will be added to points. If some point's options are not visible, check if this property is not set to *enabled*.

2.2 Map

A map is composed by lines, points, crosses and vehicles. During the design of a map some constraint and configuration can be set. For example speed, direction of movement. The main property of a vehicle is the dimension. The length and width can be set here.

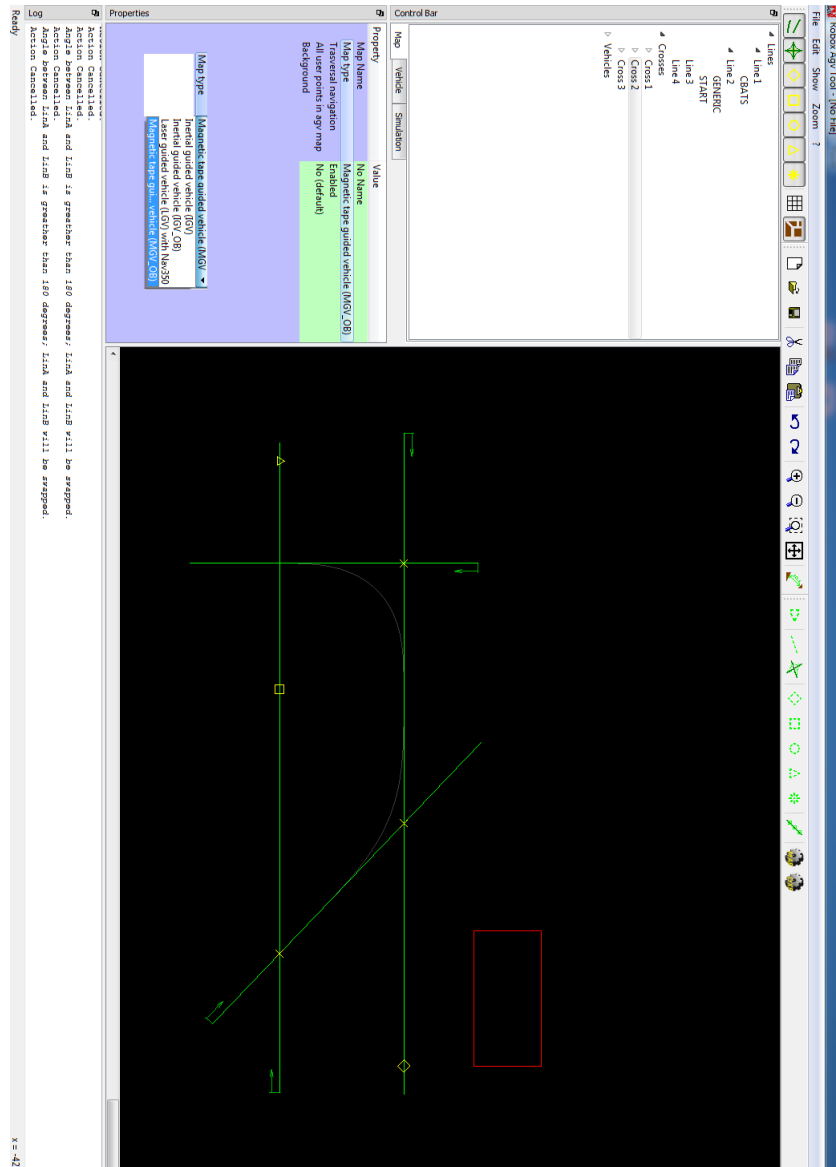


Figure 2.1: RAT main window

2.2.1 Vehicle

We can define the number of vehicles present in a plant, their shapes and dimensions. In our discussion we suppose a vehicle have an orientaion, a coordinate systems attached to it. We can imagine the vectors (arrow) \overrightarrow{BF} and \overrightarrow{RL} as coordinate system axis, fig.2.2, i.e. $\overrightarrow{x} = \overrightarrow{OF}$ and $\overrightarrow{y} = \overrightarrow{OL}$.

If we have more than one Agv, it is convenient to set different colors, we can do it changing the property *Enabled Colour*. The default value is white (255,255,255).

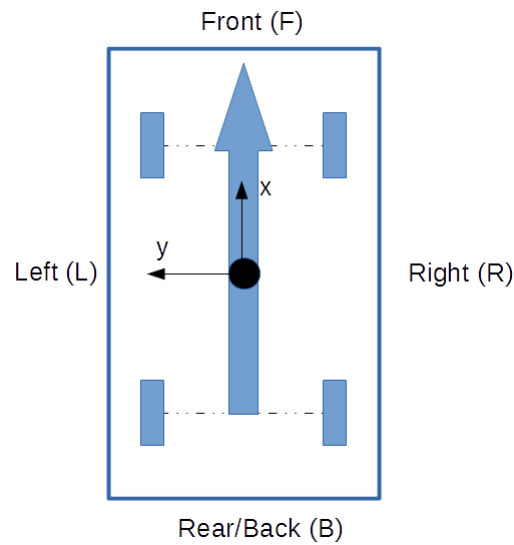


Figure 2.2: Vehicle orientation

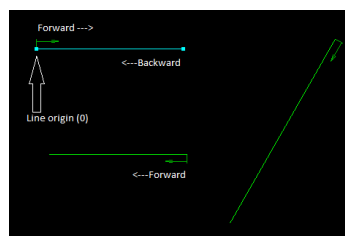
2.2.2 Lines

A line have mainly 2 properties, beside its location and origin fig.2.3b. Navigation direction and vehicle orientation. A line have to be seen as a vector, \vec{L} . For example a vector \vec{OX} has opposite direction of vector \vec{XO} , note that $\vec{OX} = -\vec{XO}$.

Two directions of motion are allowed: Forward and backward. Forward direction is shown by the arrow on the line, that is the positive movement, from O to X represented by \vec{OX} .

The vehicle can move longitudinally fig.2.4 to the line, i.e. \vec{BF} parallel to the line, or transversally (side navigation), i.e. \vec{BF} perpendicular to the line fig.2.5.

Precisely a line is a vector. The first point drawn P_1 (first mouse click) define the origin of the vector, the second point P_2 determine its direction. So the line is defined as $\vec{P_1P_2}$. The origin can be moved changing the parameter origin, when it is different from zero we can see the arrow on the line move, the position of the origin is calculated always from P_1 .



(a) Line or vector. Direction of motion

Property	Value
Name	LineName
Index	2
P1	-5.065, 1.871
P2	-3.876, 1.871
Origin	0.000
Side navigation	Forbidden
Longitudinal navigation	Enabled
	Forbidden

(b) Line property. Navigation type can be set : side, longitudinal or both

Figure 2.3: Map line

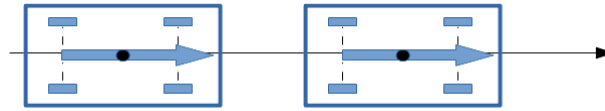


Figure 2.4: Longitudinal navigation. BF parallel to the line.

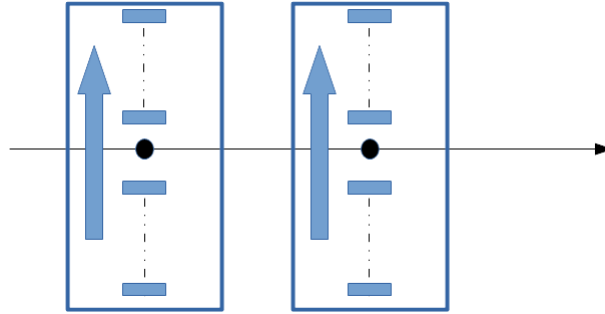


Figure 2.5: Side or traversal navigation. BF perpendicular to the line.

2.2.3 Generic point

There are 6 kinds of points as shown in fig.2.6. In term of object oriented approach we may say that all points derive from the base class Generic point. Those points share the following basic properties: Quote (position on the line), speed of the vehicle while crossing the point, direction (as a reference the line where the point is placed) and orientation (referred to the vehicle).

Genric points are used mainly to build the path of the vehicle. It is not necessary to assign a code to a generic point. AgvManager assign codes to Generic points that don't have one.

The following discussion can be applied to all kind of points, that have the properties direction and orientation (generic, user, cross, magnet, start, battery).

There are three allowed directions to approach and leave a point: Forward(F), Backward (R) and Anydirection (X). The allowed direction of point e.g. P_1 is meant as the direction of motion of the vehicle starting from this point toward another point in the positive direction of the line.

For example, if we set the allowed direction of point P_1 to Forward , and we want to move from P_1 to point P_2 placed at a coordinate greater than P_1 , the motion is allowed. But the motion from P_2 to P_1 is not allowed. The direction in point P_1 control the direction of motion starting from itself toward positive coordiantes, fig.2.8.

The allowed orientation is referred to the vehicle fig.2.2. A point have 7 allowed

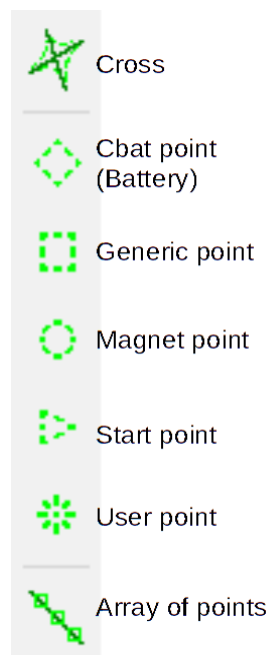


Figure 2.6: Kind of points

orientations. For example if the Front orientation is selected, the vehicle when is moving on the line, \overrightarrow{BF} have the same orientation of the line \overrightarrow{L} . A Front orientation on point P_1 , mean that the vehicle when moving from P_1 to positive coordinates the orientation of the vehicle is Front.

Semaphores can be created using any points except magnet point. When *semaphore index* is 0, there is no semaphore defined. When the index is positive the point define the semaphore start, when it is negative the point define semaphore stop. The semaphore is a rectangular area, with width define by the parameter *semaphore width*, and length defined by the position of the start and stop points.

Speed property????

Can also be created array of points of a selected kind on a line.

2.2.4 User point

User point are like generic point, but they are associated to operations. For example, loading and unloading operations can be associated to user points. Information about the operations done on user points can be written on a database.

A user point should have the code property not empty, but a generic point code could be empty. A point belong to a line, if we have for example a matrix of points and lines, let's suppose the points belong to the horizontal line, if we need to move vertically from a

Property	Value
Name	
Code	
Kind	GENERIC
Quote	109.130
Speed	1
Allowed Direction	AnyDirection
Allowed Orientation	AnyOrientation
Semaphore stop	Rear Front AnyOrientation

Property	Value
Name	
Code	
Kind	GENERIC
Quote	109.130
Speed	1
Allowed Direction	AnyDirection
Allowed Orientation	AnyOrientation
Semaphore stop	Left Right Trasversal Longitudinal Rear Front AnyOrientation

Property	Value
Name	
Code	
Kind	GENERIC
Quote	109.130
Speed	1
Allowed Direction	AnyDirection
Allowed Orientation	Backward Forward
Semaphore stop	AnyDirection

Figure 2.7: Generic point property

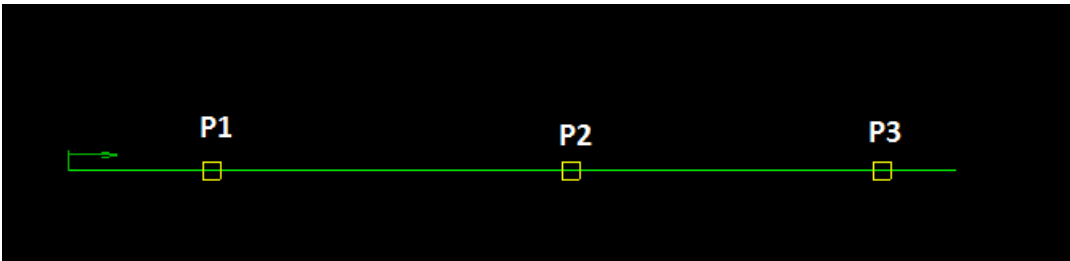


Figure 2.8: Line from left to right. P1 Forward direction, P2 Backward direction. Motion from P1 to P2 is allowed, from P2 to P1 is not allowed.

point to another, we can't do it, we need a cross point.

User kind ????????????

Side ????????????????

2.2.5 Battery point

CBats are battery points, i.e. charging station position. This point have the properties kind, index, side and the properties that derive from a generic point fig.2.10b.

CBats Kind ????????

Side ????????????????

Display ????????????????

2.2.6 Magnet point

A magnet point have the similar properties as a generic point, but is not used for path construction. A magnet point is used for position adjustment and reference. Every magnet point should have an Rfid code, this code must be unique.

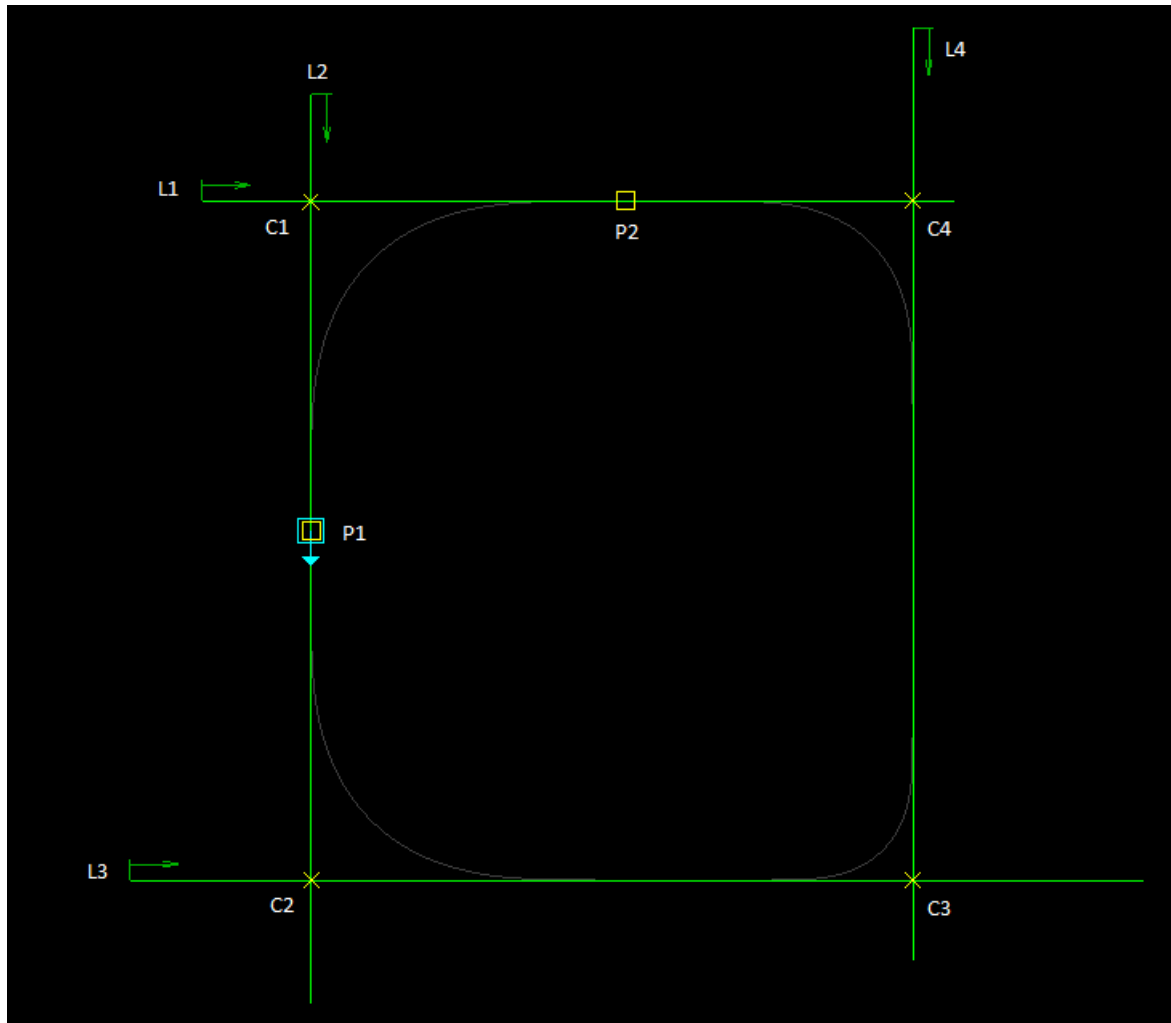


Figure 2.9: Point allowed direction. P_1 allowed direction is set to Forward. Motion from P_1 to P_2 crossing C_1 is allowed, because in C_1 the direction is not restricted, and because P_2 is not in the growing coordinate starting from P_1 .

Side offset ??????

Magnet type ??????

Forward mode ???????/

Backword mode ????????

A magnet point must be installed at 0.5 m from a curve. For example if we have a cross of type curve, and 1 meter of takeoff distance, 2 magnet points have to be installed at least at 1.5m from the cross 2.11.

2.2.7 Start point

A start point is used as a home reference for a vehicle. A vehicle, once turn on, doesn't know his absolute position. Start point, associated with magnet point can be used to

Property	Value
Name	
Code	
Kind	USER
Quote	32.550
Speed	1
Allowed Direction	Backward
Allowed Orientation	AnyOrientation
Semaphore index	0
User Kind	1
Side	Centre
Visualization	0, 0, 0

(a) User point properties

Property	Value
Name	
Code	
Kind	CBATS
Quote	24.932
Speed	1
Allowed Direction	Backward
Allowed Orientation	AnyOrientation
Semaphore index	0
CBats Kind	1
CBats Index	1
Side	Centre
Display	Side
Offset X (on line)	0
Offset Y (Side)	0
Offset (Z)	0

(b) Battery point properties

Property	Value
Name	
Code	
Kind	MAGNET
Quote	22.040
Speed	1
Allowed Direction	Backward
Allowed Orientation	AnyOrientation
Side Offset	0
Magnet Type	Single
Forward Mode	0
Backward Mode	0

(c) Magnet point properties

Property	Value
Name	
Code	
Kind	START
Quote	14.423
Speed	1
Allowed Direction	Backward
Allowed Orientation	AnyOrientation
Semaphore index	0
Start Index	1
Starting orientation	Front

(d) Start point properties

Figure 2.10: Map points

establish the position of a vehicle. In one map we may have more than one start point for one vehicle, pay attention to set the property Start index that should be unique number. If the index is not unique for start points RAT doesn't give any error (like for user points), but AgvManager will give an error when loading the map.

A reference position is composed from one start point and 2 magnet points. The position (quote) of the start point should be the same of one of the 2 magnet points.

Starting orientation ??????????

2.2.8 Cross

A cross is the intersection of 2 lines. An intersection have 4 quadrants. You can establish permission for vehicle in one or more quadrants. Three kinds of permission are available: Forbidden, curve and rotation fig.2.12.

A curve can be of 2 different types: **0- Odometric curve** and **1- Tape curve with 3 segments**.

Divieti is an 8 bit mask, the first 4 bits indicate the allowance of passing from line A to line B, and the second 4 bits indicate the passing from line B to A.

Occupable indicate if the quadrant is occupable, when the agv rotate around itself, if the value is yes, the agv can cross the quadrant while rotating. If all 4 value are no, for the 4 quadrante the agv can only rotate the wheels is the passage mode is rotation.

Under the fields, points on line A and B, we can set the allowed direction an orientations for each line.

Override angle?????/?

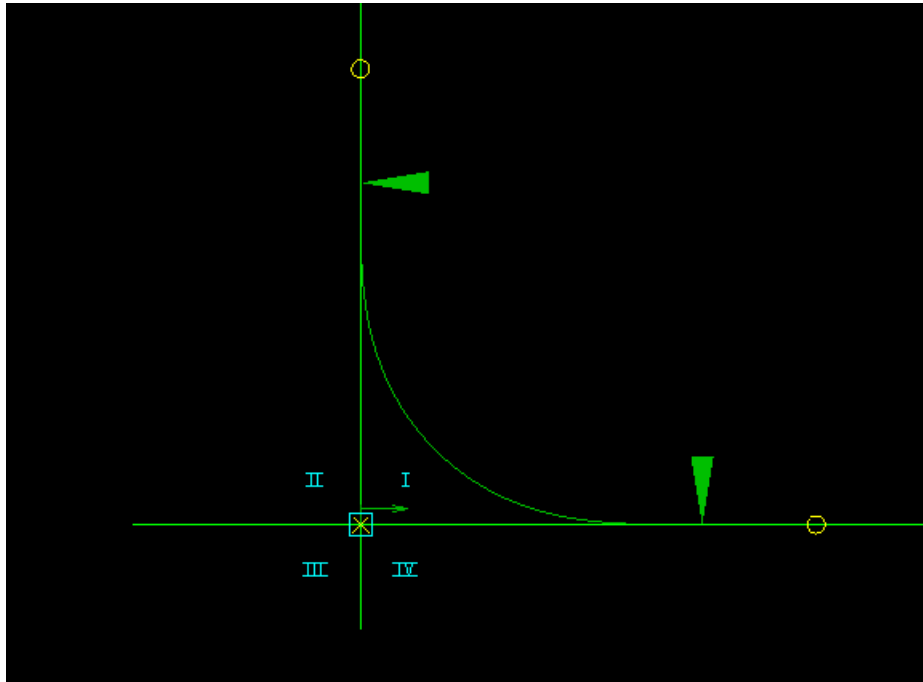


Figure 2.11: Two magnet points should be placed at least 0.5 meter from the end of a curve.

2.3 Tips

1. A reference point is composed from a start point and 2 magnets.
2. A curve should have 2 magnets placed at least at 0.5 meter from the end of the curve fig.2.11.
3. User points and generic points should be placed after the magnet points that form the curve fig.2.13.

Property	Value
Name	
Index	5
Code	Same as index
Divieti	00000000
Override angle	No
Points on line A	
Speed	0.35
Allowed Direction	AnyDirection
Allowed Orientation	AnyOrientation
Points on line B	
Speed	0.35
Allowed Direction	AnyDirection
Allowed Orientation	AnyOrientation
Quadrant 1	[Noc] Curve (1.5 m, 0.3 m/s)
Passage Mode	Curve
Occupable	No
Takeoff distance	1.5
Speed	0.3
Path length	0
Flags	0x00000001
Quadrant 2	[Noc] Forbidden
Quadrant 3	[Noc] Rotation
Passage Mode	Rotation
Occupable	No
Speed	1
Path length	0
Flags	0x00000001
Quadrant 4	[Noc] Forbidden
Passage Mode	Forbidden
Occupable	No
Path length	0
Flags	0x00000001

Figure 2.12: A cross is a point that joint two lines. It behave like a point on every line, orientation and direction can be set for every line independently.

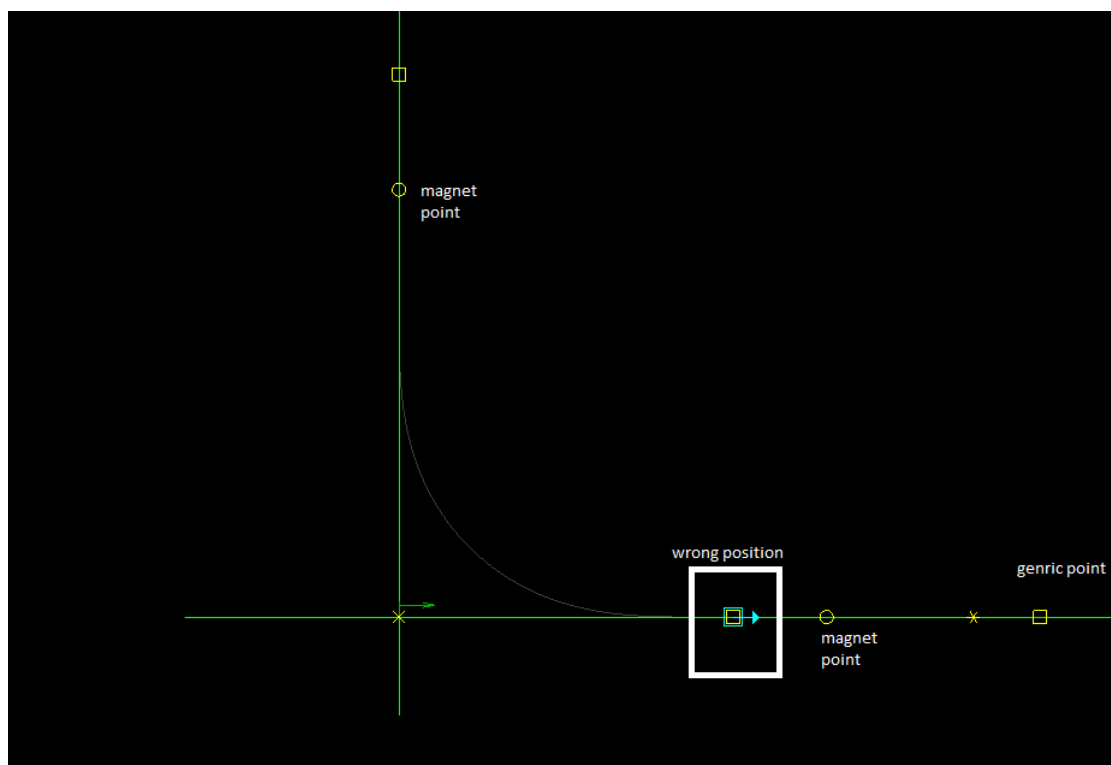


Figure 2.13: Generic points and user points should be placed outside a curve, i.e. after a magnet point.



3. AGV Manager

3.1 Overview

AGV Manager have 2 software components: AGV manager it self and AGV configurator. In AGV configurator are set some parameters like the map file directory, script directory, communication with the AGVs controllers, PLC communication and IO definition, database communication, emulator enabling, etc.

AGV manager will load the parameters set by Agv configurator, and main execute the script written for the specified plant. In AGV manager can be shown the map and motion simulation and modify the script. The script is written in XScript language (Robox scripting language) and executed by AGV manager.

3.2 Installation

The installation of AgvManger is straightforward, like any program in Microsoft Windows. AgvConfigurator is installed automatically with AgvManager.

In order to get the report from AgvManager a database should be installed. You can install MySql community version.

Once installed, a shortcut to AgvManager and AgvConfigurator have to be done. In the properties of the application, in [Start in](#) put the location of the folder where your projects are located, fig.3.2.

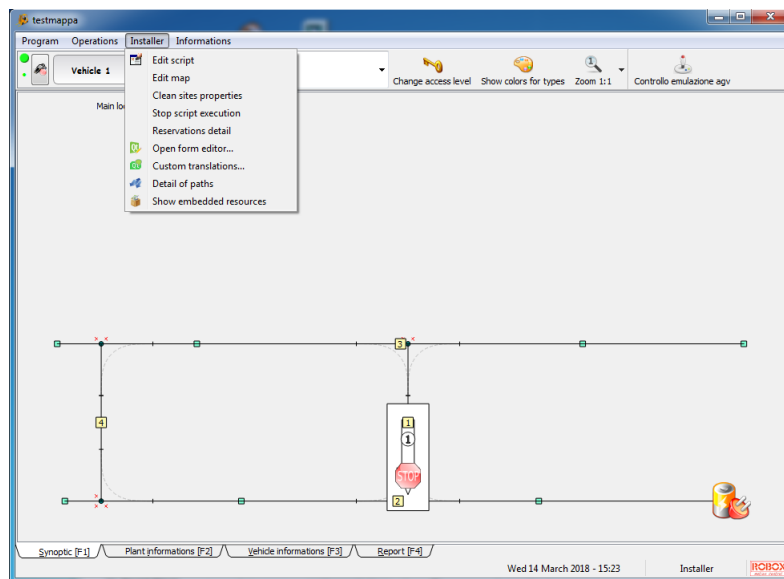


Figure 3.1: AGV Manager main window

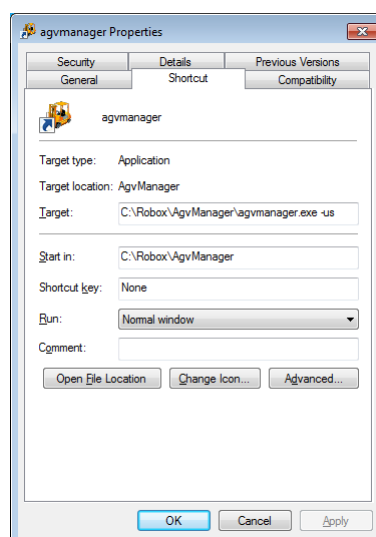


Figure 3.2: Application properties: Change the "start in" field to your projects directory

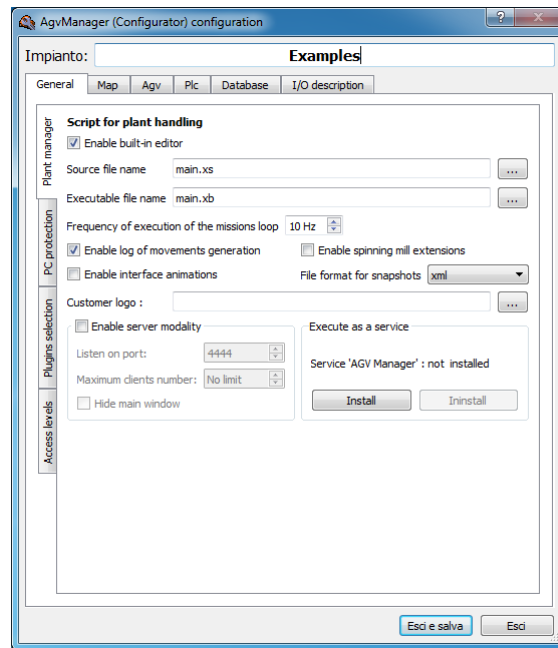


Figure 3.3: AGV configurator. General tab, where the .xs script file is selected. The script have to be created by an external editor. It is enough to write the name of the executable file with .xb extension, when AgvManager compile the xs file, the xb file will be created automatically.

3.3 AGV configurator

AGV configurator is a standalone program that create a configuration file *.fdoc* for AGV manager. From AgvConfigurator you can select the script to be executed by AgvManager fig.3.3, select the map fig.3.4 and agv 3.5 and plc communication.

Project folder should be placed in the directory specified in the field [Start in](#) of the application properties, otherwise it will not work. By default the start point of the application is its installation directory.

The script file should be in the first level of the project folder, it can't be placed in subdirectories, for example `appStartDir/Project01/scripts/main.xs` is not allowed, it should be `appStartDir/Project01/main.xs`.

3.4 AgvManager interface

AgvManager have one menu bar, one tool bar, one status bar, map visualization and different tabs [Fx].

In the tool bar, fig.3.6, we can find the button: Vehicle status, Commands insertion, Access level, color type, Zoom, Agv emulation, user define forms.

In the status bar we can see some message from the script, date and time, and current

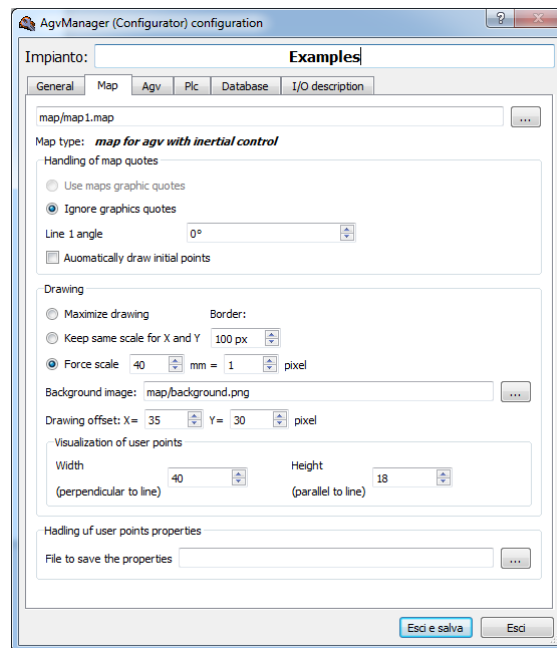


Figure 3.4: AGV configurator. Map tab, where the .map file is selected. In order to view the user point, you have to set the width and height of it, otherwise user points are not viewed. It is convenient to represent user points as rectangles, not squares.

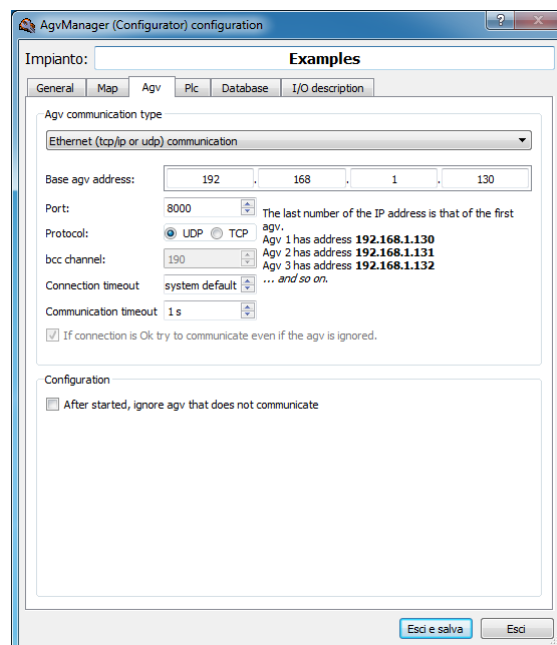


Figure 3.5: AGV configurator. AGV tab, where communication parameter with the AGV are set.



Figure 3.6: AgvManager tool bar

access level.

3.4.1 AGV emulation

If the flag *emuagv.dll emulazione agv* in AgvConfigurator, we can emulate the AGV in AgvManager. The windows of the emulation can be opened via the button *Controllo emulazione agv*.

The window in fig.3.7 shows the status of the Agv, groupbox "Stato", where are viewed the 32 Vehicle Status flags (XVehicleInfo.uStatus). The first 4 flags are written by the vehicle and the others can be defined by user. The first 4 bits (flags) are:

- Power enabled
- Execution command
- Charging battery in progress
- Load present, or Unit load present

These flags correspond to:

```

1 // Vehicle status flags , bit mask 2^n.
2 // 4 least significant bits
3
4 $define VST_POTENZA_ATTIVA 1 // Power active , mask bit 0
5 $define VST_EXEC_COMANDO 2 // executing command, mask bit 1
6 $define VST_CARICA_INCORSO 4 // charge in progress , mask bit 2
7 $define VST_CARICO_PRESENTE 8 // load present , mask bit 3

```

The Battery box, indicate the amount of power consumed, not the remaining one. for example if the status is 100%, this mean the battery is empty, if the progress bar indicate 20%, this mean the remaining power is 80%. The value of the remaining gpower is shown in the *Battery capacity* progress bar in the tab *Vehicle informations [F3]*.

To emulate the Agv, first the Agv emulation should be active, state shown in tab Vehicles (Veicoli). Then in the vehicle tab the operating mode can be selected, and the status can be emulated. The Agv should be in automatic and power is enabled in order to move the Agv. For example, if we set the flag *Load present* to one, the agv behave depending on the script logic. If the load is a loading unit and there is a load in some

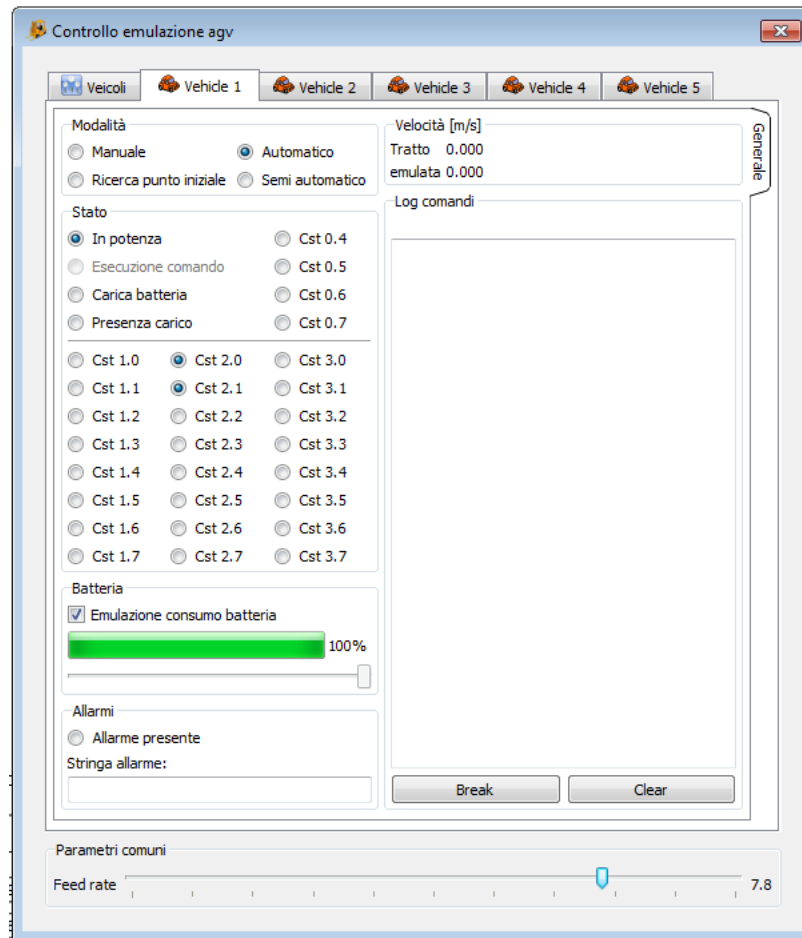


Figure 3.7: Emulation windows. We can see Agv status flags and other informations. The progress bar indicate the consumed power of the battery, not the remaining power, e.g. 100% is battery empty.

station to take out, the Agv will go to that station. Or for example if the load is properly the final product the agv may transport it to some unloading station.

3.4.2 Point windows property

A user point can be viewed as rectangle, where the dimensions are set in AgvConfigurator. A CBat (Battery point) is shown as a battery icon and . A double click on a user point or battery point a window is opened, see fig.3.8.

In the tab Storage informations, fig.3.8a, changing the type we can see the description associated to is, e.g. Type 1 is an empty trolley. The spin box (numeric updown control) is used to show only the type, the type is a combination of the checkboxes.

Properties assigned by the function *AddIntProperty()*, are shown in the tab Properties fig.3.8b.

3.5 AGV script executing

AGV manager can be compared to a plc (hardware and firmware) and the script to a plc program. The firmware is the same in all plant (beside updates and new functionality) and the script change from plant to another.

AGV scripts are written in Xscript language. Xscript have some OOP properties (creating classes and objects), some event handling (mouse move event) and callback functions.

3.5.1 Fundamental concepts

Callback functions are called automatically by AgvManager. A list of callback functions can be found in the documentation *x-script interface, Modules, Estensione x-script per AgvManager, Functions called by AgvManager (callbacks)*.

For example the callback function *OnApplicationStart() : bool* is called once, at the first execution of the script, and the function *OnApplicationStop() : bool* is called when the script execution is stopped.

An example of mouse event handling is the function *onAgvDroppedToPoint (uint uagv, uint upointid, uint orientation)*. When the agv is dragged and dropped to a point, agv manager call automatically the function *onAgvDroppedToPoint()* and the code implemented will be executed. As input parameters, the agv index (agv 1 have index 0), destination point and orientation are passed.

In the following section we will see when other callback function are called by AgvManager.

Some variable definitions (using *#define* keyword) can be found in the documentation *x-script interface, Modules, Estensione x-script per AgvManager, Funzioni per la gestione degli agv*. For example the AGV operative modes can be find with the prefix "MOD_ ", i.e. MOD_AUTOMATICO.

The following concepts have to be understood before proceeding: Mission, MACRO, MICRO and operations.

Let's say a vehicle have to go from P_1 to P_2 . This can be considered a mission. A Mission is started by calling *agvStartMission(agv id, missionCode, mission description)* and terminated by calling *agvStopMission(agv id)*.

A mission can be composed from different MACROs. Let's say a MACRO is a macro operation that subdivide the mission. For example our mission can have 3 different MACROs. If the AGV is charging the battery, we have to stop charging (if the energy is enough to execute the whole mission), move to destination, communicate the end of the

mission.

Using the 2 defined constants by AgvManager our mission is composed from : MAC_CHARGE_STOP, MAC_END and another macro that we can define using the `$define` keyword MAC_MOVE_TO_P. It is better to define our constants from 100 to avoid errors in the program logic. For example if the already defined constant MAC_END have value 10, and our constant MAC_MOVE_TO_WP have value 10, the compiler will not give errors and the agv will behave as is not expected.

AgvManager have in memory a list(array) of the MACROs to be executed. In the list are saved the agv number/id, MACRO code/id and other 4 parameters. When the function *agvaddmacro (uint uagv, uint ucode, int ipar1 = 0, int ipar2 = 0, int ipar3 = 0, int ipar4 = 0)* is called, the new MACRO is queued at the end of the list.

A MACRO is composed from MICROs. Let's say, low level micro instructions to be executed by the AGV. There are different types of MICRO, can be found in the constant definitions with prefix MIC_. For example MIC_MOVE is a MICRO that handle the motion instruction to the AGV.

A micro is registered (ask to be executed) by calling *AgvRegisterSystemBloccante*, *agvRegisterSystemPassante* or *AgvRegisterOperation*.

An operation is a type of MICRO, a typical kind of operations are loading and unloading, and can be performed on user points. More about MICRO and operations later and the commands sent to the agv in order to execute orders from AgvManager. Remember that not all MIC instruction send commands to the agv.

3.5.2 Main loop execution

The following is a simplified explanation of the main loop of and AGV script, which is in execution behind the scene. A more complex scenario is shown in fig.3.9. When the script is executed the first time, the function *OnApplicationStart()* is called. In this function you can initialize some variables and set some parameters. After that AgvManager wait for events e.g. mouse events, or operating mode change. And continue to execute some other functions.

In the case we have more than one Agv, the execution of the functions is sequential. This means that the set of functions of Agv 1 are called first, then those of Agv 2, then Agv 3, etc. In other words, the flowchart in fig.3.9 and fig.3.10 are executed starting from the first Agv until the last in sequence, this flow is repeated always.

Let's see a simple case. The AGV is in automatic mode (*MOD_AUTOMATIC*), and there is no mission in progress. If the AGV is enabled, AgvManager call automatically the callback function *onNextMission()*, where the programmer have implemented a logic to register the next mission to be executed. When a mission is in progress, AgvManager wait

(wait doesn't mean stop script execution) till the end of the mission in order to call again `onNextMission()`.

3.5.3 Mission execution

A mission is a set of MACROs. There is a list of MACROs, where the order of execution is assigned. A mission can be assigned typically inside the callback function `onNextMission()`, and started by calling `AgvStartMission()`. A mission can be also assigned in any other function. If the list of MACROs is not empty, the effective execution of the mission begin otherwise the old MICRO continue to execute until the end.

We take only one case, if the MACRO list is not empty, the function `onExpandMacro()` is called by AgvManager. Then `onExecuteMicro()` is called. At the last call of `onExecuteMicro()` the parameter `bLastCall` is assigned to true.

Fig.3.10 show a flow chart about the mission execution. That is a single step of the mission. We have to imagine that AgvManager continue to call the flowchart like a plc, cyclically.

3.6 Drag and drop example

Let's consider a simple example. We have to write a script that react to mouse events from user. The vehicle have to move from one point to another.

First let's make a simple map, fig.3.11, with one line L_1 and two generic points P_{10} and P_{20} , the script will work on any map. After the configuration with AgvConfigurator, we open AgvManager in order to write the script and simulate. Notice that, after any modification of the script, AgvManager must be closed then opened.

A simple program like this, should have at least the following callback functions:

1. `OnApplicationStart() : bool`
2. `OnAgvDroppedToPoint(uint uAgv, uint uUser)`
3. `OnExpandMacro(uint uAgv, uint uMission, uint iMacroCode, int iPar1, int iPar2, int iPar3, int) : bool`
4. `OnExecuteMicro(uint uAgv, bool bLastCall, int iMicroCode, int iPar0, int iPar1, int iPar2, int, int, int, int, int, int) : bool`
5. `OnAbortMission(uint uAgv)`

For simplicity we don't implement our own functions. Attached to this document will be provided the example we discuss here, and an equivalent example where other functions and files were added in order to keep the projects modular. The

modular example is organized as follow: 3 files, 5 callback functions, 2 user-defined functions and some variable and constant definitions. A file called `main.xs` contain the inclusion of the other 2 files. A file called `agvEventFunctions.xs` where callback functions are implemented. A file called `common.xs` where common functions and variables definitions are implemented. This structure is meant to be a template for future projects, as many functions can be reused. Of course other files and functions can be implemented. The structure of any project should be modular, portable and reusable.

The single file example have 5 callback functions and some constant definitions. By convention, constants are written using capital letters. We will discuss the functions in order of execution. The function `onAbortMission()` is called when a mission is aborted, it will be discussed at the end.

onApplicationStart()

The implementation of this function is shown in listing 3.1. As we say before, this is the first function called by `agvManager`. first we create a variable `mpar` of type `XMapParams`. This is a structure that will contain informations about the vehicle. By the function `agvGetMapParams(xmapparams&)` we read the existing data from `AgvManager` and we initialize the variable `mpar` with those data. We change some parameter using the dot operator of the structure, for example we set the dimension of the vehicle i.e. `mpar.setSymmetricalVehicleDimension(length, width)`. After we apply the changes to `AgvManager` using the function `AgvSetMapParams(@mpar)`.

When the execution of this function is done, `AgvManager` wait for some event. Let's suppose, the user drag the agv, in this case the event function `OnAgvDroppedToPoint` is called.

```

1 ;
   ~~~~~

; Function called at program startup
3 ; Operations to initialize the plant.
;
   ~~~~~

5 code OnApplicationStart() : bool
   SetVersionManager(nvmake(MAJOR_VERSION, MINOR_VERSION, BUILD_VERSION
   ))

7
   XMapParams mpar
9 ;
   ; Very important: before changing some parameters, load defaults!!!

```

```
11 ;
    AgvGetMapParams (@mpar)
13 ; Set vehicle dimensions: length , width
    mpar.setSymmetricalVehicleDimension(2300, 900)
15 ;
    ; Parameters to calculate length of paths
17 ;
    mpar.dHandicapForRotation = 8000      ; Distance added when using a
        cross for rotation
19 mpar.dHandicapForCurve = 4000          ; Distance added when using a
        curve
    mpar.dDistanzaDaIncrocioOkFermata = 0  ; Minimum distance from cross
        to allow agv stop executing a movement
21 ;
    ; Parameters to modify path assignment
23 ;
    mpar.bOkInversioneSuIncrocio = false   ; True to permit inversion on
        a cross point
25 mpar.bNoFermataSuIncrocio = true        ; True does forbid to stop on a
        cross point
    mpar.bNoMovSuTrattiPrenotati = false   ; True to forbid movement
        that ends on a point reserved for movement by another agv
27 mpar.bPalletBloccaPercorso = true       ; Load unit (trolley) on path
        blocks the agv
; mpar.bDontMoveOnDestMove = true          ; Do not
29 mpar.bNoPercorsoAgvDisab = false        ; Exclude paths occupied by agv
        that are not enabled
    mpar.bNoPercorsoAgvNoMis = false       ; Exclude paths occupied by agv
        that are not executing a mission
31 ;
    ; Set parameters
33 ;
    AgvSetMapParams (@mpar)
35
    AgvSetLunghezzaMove(12000)
37
    SetAccessLevelForOperation(DefQual_OpTrascinaAgvSuLinea , ACCESS_INST)
39
    return true
41 end
```

Listing 3.1: onApplicationStart implementation

OnAgvDroppedToPoint(uint uAgv, uint uPointId)

The call of this function is a response of mouse event. There are other mouse events like *onAgvDroppedToLine()*. In this function we set the behavior of the agv, what the agv have to do when it is dragged e.g. from P_{10} and dropped to point P_{20} . First let's put some requirements e.g. the agv should be in automatic mode, it should not be enabled, there is no mission in progress. If those conditions are met the agv can move from one point to another. The code to control such conditions is self-explanatory in listing.3.2.

This example will have only one mission, moving from one point to another. A mission should have at least one MACRO, every mission should have MAC_END. This MACRO inform the manager that it reach the end of the mission. There are some predefined constants for system used MACROs, they can be found in the documentation with the prefix MAC_. We can also define our own MACROs using the keyword Define. It is a good practice to use numbers from 100, every MACRO and mission should have a unique identifier.

Let's define our mission and a new MACRO:

```

1  ; Mission null , ther is no mission
   $define MIS_NULL          0
3  ; Mission move to point
   $define MIS_TO_POINT      14
5
   ; MACRO Movement to waypoint
7  $define MAC_MOVE_TO_WP    100

```

In this case the mission MIS_TO_POINT is composed from 2 MACROs. In order the *MACRO list* will have 2 elements:

1. MAC_MOVE_TO_WP
2. MAC_END

Before starting a new mission we check if there is a mission in progress. We call the function *AgvActualMissionCode(uint uAgvId)*, this function return the id of the mission in progress. If it return zero, it means there is no mission in progress. We have already define a constant MIS_NULL as zero. In the code we can write "*if(AgvActualMissionCode(uAgv)=0)*", but it is always more readable when using names instead of numbers, so instead of 0 we use MIS_NULL.

If there is no mission in progress, we can start the mission MIS_TO_POINT by calling the function *bool AgvStartMission(uint uAgvId, uint CodeMissione, string sMissionDescription)*, this function return true if the mission is in progress.

Now we have to fill the agv MACRO list with our 2 MACROs, by calling the funnction *bool agvAddMacro(uint uAgv,uint uCodeMACRO, int ipar1=0,int ipar2=0, int iapr3=0, int ipar4=0)*. The iparX have 0 as default value.

The first MACRO is MAC_MOVE_TO_WP, this is a motion MACRO. So we have to build the path of the agv by calling the function AgvAddWaypoint(), this function take as parameters the agv id, the point id and direction and return an id of the point.

Then the MAC_MOVE_TO_WP can be add to the list, by calling agvAddMacro(), giving it as ipar1 the return value of the function AgvAddWaypoint() and as ipar2 a flag to concatenate the execution of the next MACRO. Then the macro MAC_END that end our mission is add to the macro's list.

```

1  AgvStartMission(uAgv, MIS_TO_POINT, "Mission to point")
   ;
3  uint wpidx
   uchar destOrientation = 'X'
5  bool concatenateNext = true
   ;
7  wpidx = AgvAddWaypoint(uAgv, uUser, destOrientation)
   AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
9  ;
   AgvAddMacro(uAgv, MAC_END, MIS_TO_POINT)

```

```

;
~~~~~
2 ; Called when the user drags an agv (uAgv) to a point in map (uUser)
;
~~~~~

4 code OnAgvDroppedToPoint(uint uAgv, uint uUser)
   if (uAgv >= MAX_AGV)
6     MessageBox("Invalid AGV number: " + (uAgv + 1))
     return
8   end
   if (not AgvInAutomatico(uAgv))
10    MessageBox("AGV " + (uAgv + 1) + " is not in automatic mode.")
     return
12  end

```

```

14  if ( AgvAabilitato(uAgv) )
    MessageBox("AGV " + (uAgv + 1) + " is enabled." + chr(10) + "Please
        disable it to give commands.")
    return
16  end
    if ( AgvActualMissionCode(uAgv) != MIS_NULL )
18      MessageBox("AGV " + (uAgv + 1) + " is already executing a mission")
        return
20    end
    ;
22    AgvStartMission(uAgv, MIS_TO_POINT, "mission to point")
    ;
24    uint wpidx
    uchar destOrientation = 'X'
26    bool concatenateNext = true
    ;
28    wpidx = AgvAddWaypoint(uAgv, uUser, destOrientation)
    AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
30    ;
    AgvAddMacro(uAgv, MAC_END, MIS_TO_POINT)
32 end

```

Listing 3.2: OnAgvDroppedToPoint implementation. This function as input have the AGV id and the destination point id. This is an evznt function it is called when the user drag and drop the vehicle to the desired point

OnExpandMacro()

As we mentioned before, when a mission begin the function `OnExpandMacro()` is called automatically by AgvManager. We already started a mission in the function `OnAgvDroppedToPoint()` and filled the MACRO list with 2 MACROs. So now we have to implement the function `OnExpandMacro()`.

AgvManager executes the MACROs starting from the first one in the list. When it call the function `OnExpandMacro()`, give it the **Agv id**, **mission id**, **MACRO code/id** and the four parameters stored in the list. We can imagine every elements of the list, is composed from those fields. So in the implementation of this function we check the MACRO code to be executed. We can use the case statement or the if in order to select our logic.

The first MACRO is MAC_MOVE_TO_WP. Under the case MAC_MOVE_TO_WP we implement the instructions to AgvManager:

```

2  case MAC_MOVE_TO_WP
    ; iPar1 = Waypoint id
    ; iPar2 = (bool) do concatenate next macro

```



```

4      select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
| WpFl_EliminaCompletato))
        case EsitoMov_MovimentoCompletato ; Completed movement
6      case EsitoMov_RaggiuntoWaypoint    ; Waypoint reached
        if (iPar2)
8          AgvComputeNextMacro(uAgv)
        endif
10     return true
    default
12     return false
    endselect
14 return true

```

In this code the motion instruction is done by calling `AgvMoveToWayPoint()`, when this function return a value corresponding to `MoveResult_WaypointReached`, the next MACRO is expanded. The next MACRO in our example list is the `END_MACRO`.

*When the return value of `OnExpandMacro()` is **true**, it mean the current macro execution is terminate, and on the next call the following macro will be executed.*

As we say every MACRO consist of different MICROs. A MACRO that correspond to a motion have a MIC_MOVE. Here the MIC_MOVE is registered by the call of the movement function `AgvMoveToWayPoint()`.

The MAC_END register a MIC_SYSTEM micro type. When the MAC_END is expanded, it start or register a new micro. Simply this MACRO have only one MIC_SYSTEM micro type that is S_END.

This MICRO inform AgvManager that the mission is ended. In the case MAC_END the micro S_END is registered by calling `AgvRegisterSystemBloccante(uAgv, uMission, S_END)`, where the function `agvStopMission(uagv)` is called, as we will see in the function `onExecuteMicro()`.

As shown in fig.3.10, AgvManager continue to call `onExpandMacro()` and `onExecuteMicro()`.

When the `onExpandMacro()` terminate the function `onExecuteMicro()` is called.

In the tab **vehicle informations[F3]**, under Agv commands we can se a list of missions, macros expansion and micro instructions, as well as information about them, fig.3.12, fig.3.13 and fig.3.14.

```

;
~~~~~
2 ; Do the job assigned to the macro that has actually to be executed
;
4 ; Return TRUE when all work has been done, and the macro is finished.
;
6 ; Return FALSE when the work has not been finished: the function
; will be called again for this macro
8 ;
;
~~~~~
10 code OnExpandMacro(uint uAgv, uint uMission, uint iMacroCode, int iPar1
, int iPar2, int iPar3, int) : bool
;
12 ; Macro expansion, depending by the macro code
;
14 select (iMacroCode)
    case MAC_MOVE_TO_WP
16         ; iPar1 = Waypoint id
        ; iPar2 = (bool) do concatenate next macro
18         select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
| WpFl_EliminaCompletato))
            case EsitoMov_MovimentoCompletato ; Completed movement
20            case EsitoMov_RaggiuntoWaypoint ; Waypoint reached
                if (iPar2)
22                    AgvComputeNextMacro(uAgv)
                endif
24            return true
        default
26            return false
    endselect
28    return true

30 case MAC_CHARGE_STOP
    AgvRegisterSystemBloccante(uAgv, uMission, S_CHARGE_STOP)
32    AgvRegisterOperation(uAgv, uMission, O_CHARGE, O_CHARGE_STOP)
    AgvComputeNextMacro(uAgv)
34    break

36 case MAC_END
    SetAgvMessage(uAgv, "")
38    AgvRegisterSystemBloccante(uAgv, uMission, S_END)
    break

```

```

40
    default
42        qt_warning("Unknown macro: " + iMacroCode)
        break
44    end
    return TRUE
46 end

```

Listing 3.3: OnExpandMacro

OnExecuteMicro()

MICROs are instructions to the vehicle. MICROs are stored in a list, one MACRO can register more than one MICRO fig.3.12.

For example a MICRO can be registered by calling [agvRegisterSystemBloccante\(\)](#) or [agvRegisterSystemPassante\(\)](#) for MIC_SYSTEM type or [AgvRegisterOperation\(\)](#) for MIC_OPERATION type. See documentation for a more complete list of micro registration functions. These function have uAgv, uMission, MICROcode as input parameters.

There are different types of MICROs, that can be found in the documentation with prefix MIC_. Let's see MIC_SYSTEM to which the S_END belong, this type of MICRO doesn't send any instruction to the agv itself. For example S_END is need to end a mission, and is managed by AgvManager.

For example listing.3.4, register a 30 seconds waiting time.

```

1  $define S_START_WAIT      100
   $define S_EXEC_WAIT      101
3  AgvRegisterSystemBloccante(uAgv, uMission, S_START_WAIT, iPar1)
   AgvRegisterSystemBloccante(uAgv, uMission, S_EXEC_WAIT)

```

Listing 3.4: Wait time system micro

A MIC_MOVE type is related to instruction of motion sent to the agv. A MIC_OPERATION is an operation like loading and unloading.

There are 2 kinds of micros: blocking and non-blocking MICROs. The difference is that the blocking MICRO lock the execution of other micros till the end of the execution of itself or till the verification of a condition.

When expanding macros, the micro list is composed by calling the relative registration function. For example, in the function [OnExpandMacro\(\)](#) under the MAC_END, we register a blocking system micro, S_END. In the function [onExecuteMicro\(\)](#) under the case

MIC_SYSTEM and under the case S_END we call the function `AgvStopMission(uAgv)` in order to stop the mission. When a micro terminate the execution of the function `onExecuteMicro()` return true.

When the mission is stopped the macro list is eliminated, and the agv is ready to get another mission.

```

;
~~~~~
2 ; Execution of the actions related to vehicle operations ,
; and execution of the SYSTEM micro.
4 ;
~~~~~

code OnExecuteMicro(uint uAgv, bool bLastCall, int iMicroCode, int
    iPar0, int iPar1, int iPar2, int, int, int userId, int iMission,
    int) : bool
6 XVehicleInfo vInfo
  AgvGetVehicleInfo(uAgv, @vInfo)
8
  select (iMicroCode)
10
    case MIC_MOVE
12    case MIC_CURVE
    case MIC_ROTATION
14      return true

    case MIC_OPERATION
16
    case MIC_SYSTEM
18      select (iPar0)
20        case S_NULL
          ; Micro of that type are generated by AgvManager, I am not
          intereseted on it.
22          break

          case S_END
24            ; End of mission
            if (vInfo.uStatus & VST_EXEC_COMANDO)
26              MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": wait for
              agv commands finished")
28              return false
            endif
  endif

```

```

30      MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": finished
executing commands")
      AgvStopMission(uAgv)
32      SetAgvMessage(uAgv, "")
      break
34      default
          qt_warning("Unknown MIC_SYSTEM : " + iPar0 + " (mission = " +
iMission + ", par1 = " + iPar1 + ")")
          break
36      end
38      break

40      case MIC_PASSANTE
          select (iPar0)
42              default
                  qt_warning("Unknown MIC_PASSANTE : " + iPar0 + " (mission = "
+ iMission + ", par1 = " + iPar1 + ")")
                  break
44              end
46              break

48      case MIC_WAIT
          if (bLastCall)
50              MessageState("Agv " + (uAgv + 1) + ": passthrough operation
executed")
              return true
52          end
          return false

54      default
          qt_warning("Unknown micro: " + iMicroCode + " (mission = " +
iMission + ", par0 = " + iPar0 + ", par1 = " + iPar1 + ")")
          break
58      end
60      return true
end

```

Listing 3.5: OnExecuteMicro

3.7 Summary

In automatic mode, if the Agv is not executing a mission, if it is enabled the callback function `onNextMission()` is called. If there is a mission in progress, even if the Agv is

not enabled, continue to execute the mission till the end or till receiving an abort mission command, see fig.3.9.

For example, in the implementation of the function `onNextMission()` missions can be assigned to agv depending on the plant status, e.g. by calling a user defined function `RegisterMission()`. The signature of the the function `RegisterMission()` can be defined as we wish.

```

1  // onNextMission()
3  // register mission depending on the plant logic
   RegisterMission(uAgv, MIS\LOAD\FROM\STATION, iPar1 , iPar2 )

```

Listing 3.6: `onNextMission()` missions are assigned

In the function `RegisterMission()`, depending on the mission we compile the **macro list**. For example if our mission is to go to load a product the macro list will be composed in the following way:

```

1  // RegisterMission()
2
3  // Start mission with id uMissionCode
4  AgvStartMission(uAgv, uMissionCode , MissionDescription)
5
6  case MIS_LOAD_FROM_STATION
7  //
8  uint wpidx
9  uchar destOrientation='X'
10 bool concatenateNext=true
11 wpidx = AgvAddWaypoint(uAgv, ID_LOAD_STATION, destOrientation)
12 AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
13
14 // Wait for operator to load toilet
15 AgvAddMacro(uAgv , MAC_WAIT_END_LOADING)
16
17 // END of this mission
18 AgvAddMacro(uAgv , MAC_END, uCode)
   break

```

Listing 3.7: `RegisterMission()` macro list is composed

When a mission is in progress, and the macro list is not empty, the callback function `onExpandMacro()` is called. Depending on the macro we compile a list of micro instructions. For example one of the macros was `MAC_WAIT_END_LOADING`:

```

1  // onExpandMacro()
   case MAC_WAIT_END_LOADING

```

```

3   AgvRegisterSystemBloccante(uAgv, uMission, S_START_WAIT, 30)
   AgvRegisterSystemBloccante(uAgv, uMission, S_EXEC_WAIT)
5   AgvRegisterOperation(uAgv, uMission, O_WAIT_LOAD)
   break

```

Listing 3.8: onExpandMacro() micro list is composed

After the call of `onExpandMacro()` the callback function `onExecuteMicro()` is called, and depending on the micro we execute operation and instructions:

```

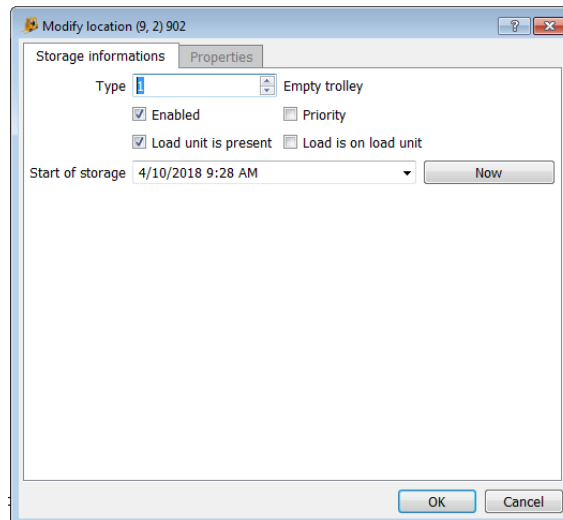
// onExecuteMicro()
2
case S_START_WAIT
4   timerWait[uAgv] = timeoutS(iPar1)
   break
6
case S_EXEC_WAIT
8   if (isTimeout(timerWait[uAgv]))
       return true
10  else
       MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : wait " + int(
secsToTimeout(timerWait[uAgv])) + "s")
12   return false
   endif
14  break

```

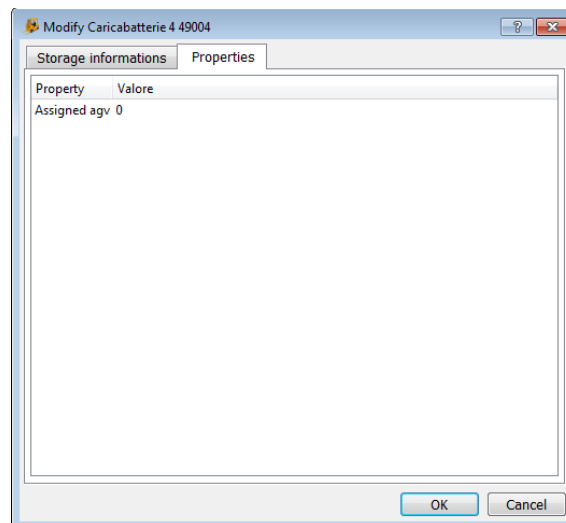
Listing 3.9: onExecuteMicro() micro are aexecuted

The state of mission execution is monitored cyclically. This mean that when a mission begin the functions `onExpandMacro()` and `onExecuteMicro()` are called repeatedly until the end of the mission, see fig. 3.10. The `onExpandMacro()` start executing the following macro in the list, when the previous returned true. The following micro is executed when `onExecuteMicro()` return true. the flag `bLastCall` is set to true by `agvManager`.

The main logic of AGVs is written in the function `onNextMission()`, where missions are assigned to AGV. When the logic is divided by missions, it is relatively easy to write the other 3 main functions : `registerMission()`, `onExpandMacro()` and `onExecuteMicro()`. Helper functions can be implemented to implement some useful logic.



(a) Storage information: Load information are shown, timestamp of loading, load type.



(b) Location properties

Figure 3.8: Location window: Storage information and properties

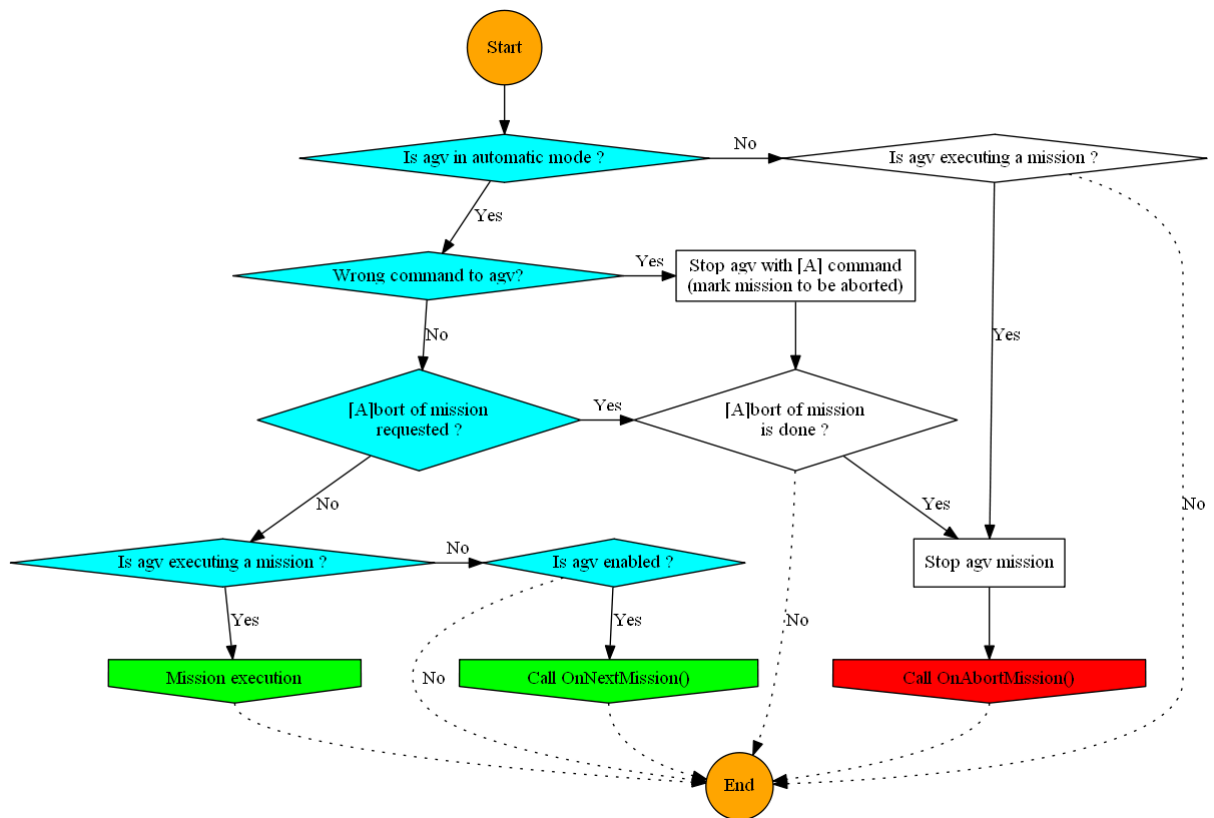


Figure 3.9: Main loop execution

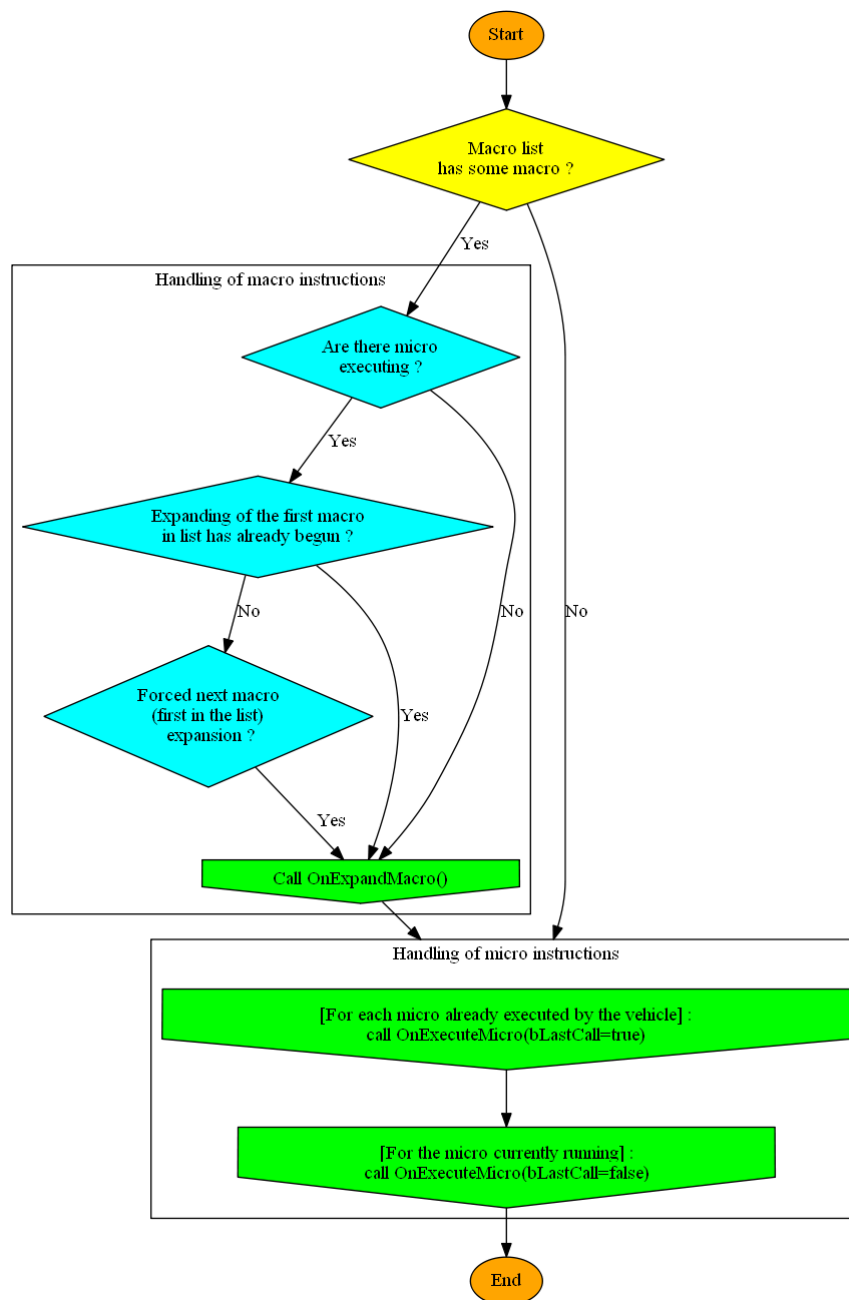


Figure 3.10: Main loop execution



Figure 3.11: Agv simple map, for drag and drop example. One line and two generic points.

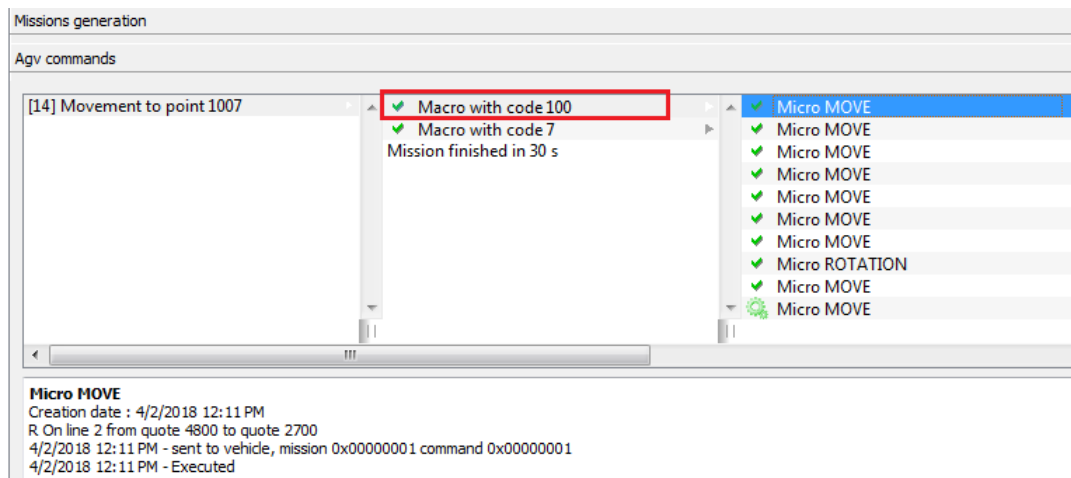


Figure 3.12: Movement to point 1007, Macro MAC_MOVE_TO_WP=100. As we can see, the macro consists of a list of MICROS. Selecting a mission or a macro or a micro we can see information about them.

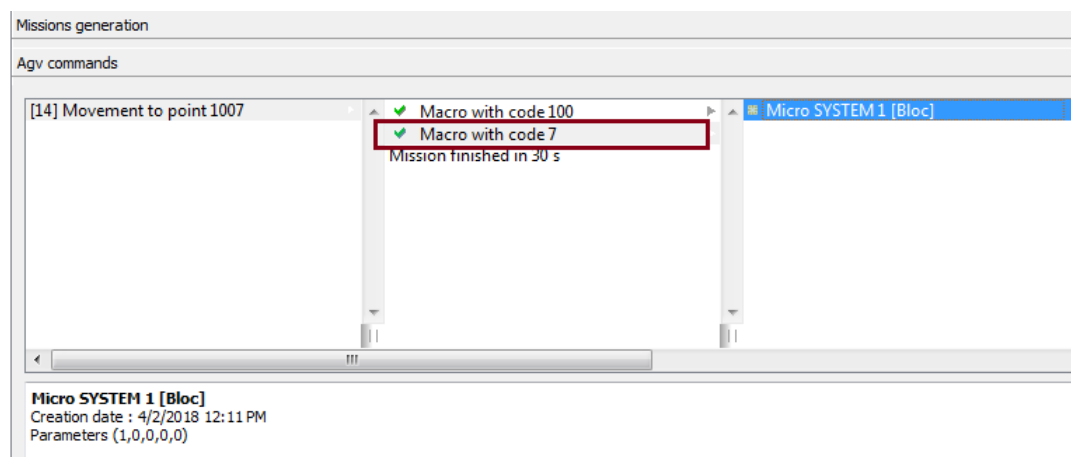


Figure 3.13: Movement to point 1007, Macro MAC_END=7. As we can see, the macro consists of one system micro that is S_END.

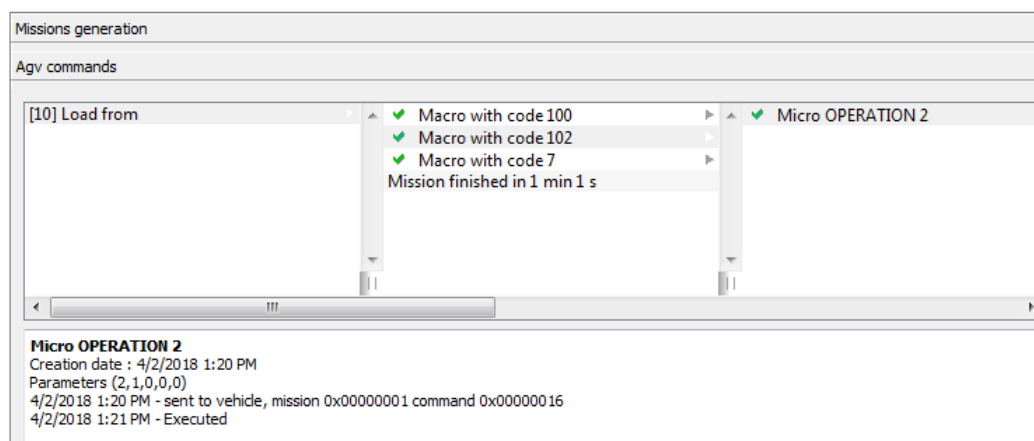


Figure 3.14: Mission load from a user point. The macro expansion shows 3 macros in the list. The MACRO 102, user defined, have only one micro of type operation that is O_LOAD, system defined. Later we will the commands sent to agv in order to execute operations.



4. More examples

In this chapter we will present some data structure of AgvManager and some examples on agv scripting. In this chapter we will show only piece of code necessary to explain the concepts. Complete examples are provided with this document. The examples package is divided by folders, every folder contain the script files. Mainly every example at least have 3 files: main.xs, common.xs and agvEventFunctions.xs. We will indicate in which file and function every piece of code can be found.

In AgvManager documentation we can find functions and data structures divided by argument, it means functions to manage vehicles, maps, databases, etc. Refer to the official documentation in order to get a complete list of functions and data structures.

An Agv usually transport a loading unit (UDC, LU) e.g. pallet, trolley, etc. , or a loading unit with a load on it e.g. a pallet with some mechanical parts on it.

For the presence Loading unit we find the variable bPresenza or bPres. For the presence of a load on board of the Loading Unit (UDC) we find the variable bVasiPieni.

Data structures and constants definition can be found in [Script editor - All functions](#).

4.1 Xscript Agv Data structures

In the documentation under the voice [Estensione x-script per AgvManager » Funzioni per la gestione degli agv](#), we can find some functions and data structures to manage AGVs. Here will present some data structures and functions that can operate on them.

Note that AgvManager have internal data structures where to save informations about vehicles, maps, points, etc. When we need, for example to get information about agv

number 4, we create a structure similar to the one AgvManager have and by calling a dedicated function we can get information on Agv 4.

4.1.1 XMapParams

This structure contains some fields to define the dimensions of an AGV and movement behavior. There are two functions that operate on this structure to get information from AgvManager and set information to it. For example, if we define a variable `mPar` as: `XMapParams mPar`, we can read parameter from AgvManager and store them into this structure by calling the function `AgvGetMapParams(@mPar)`. If we need to modify some parameter we can use the dot operator of the structure. To apply modification onto AgvManager we have to call the function `AgvSetMapParams(@mPar)`, that transfer the data from the structure `mPar` to AgvManager.

Note the use of `@` when passing the variable `mPar` to these 2 functions. The variable is passed by reference not by value.

Meaning of the structure fields??????????

```

;
;
// Parametri definizione comportamento movimentazione veicoli
;
object XMapParams
    internal 0x02000093 setSymmetricalVehicleDimension(int length, int
    width, int diagonal=0)
    internal 0x02000094 setVehicleDimension(int length_front, int
    length_rear, int width_left, int width_right, int radius = 0)
    int iLunghVeicolo           // Larghezza veicolo (per
    anticollisione)
    int iLarghVeicolo           // Lunghezza veicolo (per
    anticollisione)
    int iDiagVeicolo            // Diagonale veicolo (per
    anticollisione, autocalcolata se == 0)
    int iVehicleLengthFront     // Lunghezza veicolo dal centro
    in avanti (per anticollisione)
    int iVehicleLengthRear      // Lunghezza veicolo dal centro
    all'indietro (per anticollisione)
    int iVehicleWidthLeft       // Larghezza veicolo dal centro
    verso sinistra (per anticollisione)
    int iVehicleWidthRight      // Larghezza veicolo dal centro
    verso destra (per anticollisione)
    int iVehicleRadius           // Raggio di massimo ingombro
    durante rotazione (per anticollisione)
    real dHandicapIncrocio       // Distanza aggiunta per uso
    incrocio (DEPRECATO)

```

```

    real dHandicapForRotation          // Distanza aggiunta per ogni
    rotazione
18    real dHandicapForCurve           // Distanza aggiunta per ogni
    curva
    real dUseHandicap                  // Handicap per segmenti
    prenotati in senso contrario
20    real dFattoreCurva              // (Non usato , deprecato)
    Fattore moltiplicativo dell'ingombro in caso di curva
    real dAngMinCurvaOk              // Angolo per cui il cambio di
    corridoio e' possibile con una rotazione anche se c'e' divieto di
    ingombro dei quadranti (cambio verso)
22    real dHandicapIncontraDestMove   // Distanza aggiunta se si
    incontra un veicolo fermo. Se negativa non si passa proprio (
    ricerca di un percorso alternativo)
    real dHandicapMarciaIndietro       // Handicap moltiplicativo per
    tratti a marcia indietro
24    real dDistanzaDaIncrocioOkFermata // Distanza minima per fermata
    prima o dopo un incrocio in prenotazione movimento
    bool bOkInversioneSuIncrocio       // Ok inversione su incrocio
26    bool bNoFermataSuIncrocio        // Vietato fermare su incrocio
    bool bPalletBloccaPercorso         // Non si passa su user di tipo
    'C' occupato
28    bool bNoMovSuTrattiPrenotati     // Non muovere gli agv su tratti
    prenotati da altri agv
    bool bNoPercorsoAgvNoMis           // Escludi dal percorso tratti su
    cui si trovano agv non in missione
30    bool bNoPercorsoAgvDisab         // Escludi dal percorso tratti su
    cui si trovano agv disabilitati (e non in missione)
    bool bDontMoveOnDestMove           // Non muovere un agv se andrebbe
    a finire sulla destinazione di un altro agv
32    int iAgvPosThreshold              // Soglia di posizione agv
endobject

```

Listing 4.1: XMapParams

4.1.2 XVehicleInfo

This data structure contains information about the vehicle, for example alarm status, mission in progress, operating mode, capacity of battery, etc. To get information from AgvManager about the vehicle we can call the function `AgvGetVehicleInfo(uint agvId, xvehicleinfo& info)`, we pass to the function the index of the agv and an XVehicleInfo variable.

For example if we need information about Agv number 4, we create the structure `XVehicleInfo vInfo`, then we call `AgvGetVehicleInfo(4, @vInfo)`. In this way, we can for example read the battery status `vInfo.uBatteryCapacity`. Note that after a while the Agv is

working, this value will be different from the value AgvManager have, we have to call again the function `AgvGetVehicleInfo()` in order to update information.

The field `uint uStatus`, Vehicle Status Flag, is an 32 bit unsigned integer where information are saved in a binary way. There are defined some constants (flags) in order to decode information:

```

1  // Vehicle status flags , bit mask 2^n.
   // 4 least significant bits
3
   $define VST_POTENZA_ATTIVA    1 // Power active , mask bit 0
5   $define VST_EXEC_COMANDO     2 // executing command, mask bit 1
   $define VST_CARICO_PRESENTE   8 // load present , mask bit 3
7   $define VST_CARICA_INCORSO    4 // charge in progress , mask bit 2

```

For example we need to know if the vehicle have a load on board, we can write:
`bLoadOnBoard = vInfo.uStatus & VST_CARICO_PRESENTE.`

The first 4 bits (from 0 to 3), are reserved to system vehicle status (status communicated by the vehicle to AgvManager). The user can define its own flag status beginning from bit 4, depending on the state of the vehicle and the plant requirements.

```

1  //
   // Informazioni stato attuale veicolo
3  //
object XVehicleInfo
5   uint uLineID           // Id. linea attuale agv
   int iPosition           // Posizione [mm] dell'agv sulla linea
                           attuale
7   int iAngle             // Angolo in gradi attuale dell'agv
   uint uMode              // Modalita' attuale veicolo (vedi etichette
                           VM_*)
9   uint uStatus           // Flag di stato veicolo (vedi etichette VST_
                           ***)
   uint uAlarmStatus       // Flag di allarme veicolo
11  uint uBatteryCapacity   // Capacita' batteria:
   // 0    = Batteria completamente carica
13  // 1000 = Batteria completamente scarica
   float dBatteryPerc       // Percentuale carica batteria : 0.0 =
                           completamente scarica , 100.0 = completamente carica
15  uint uMission          // Id. missione attuale agv
   uint uCommand           // Id. comando attuale agv
17  uint uDirV             // Direzione sul corridoio: uno tra [FRSD]

```



```

    uint uExtendedPalletId // Identificativo informazioni estese pallet
    (se presenti)
19 XLastMoveInfo lastMove // Ultimo movimento registrato (.isValid
    indica validita', in piu' se non valido, .uLine == 0)
    XLastMoveInfo destMove // Destinazione finale (.isValid indica
    validita', in piu' se non valido, .uLine == 0)
21 endobject

```

Listing 4.2: XVehicleInfo

4.1.3 XSiteInfo

A data structure where information about site can be stored. A site can be a user point or a battery point. By calling `agvGetSiteInfo(int userPointId, XSiteInfo& sInfo)` we can get the user point informations from AgvManager. The function have as parameter the id or code of the user point and a reference of a `XSiteInfo` variable. The function return true if the user point exist. With the function `agvSetSiteInfo(int userPointId, XSiteInfo& sInfo)` we can set the parameter of a user point in AgvManager.

For example the field `bPresenza` is a boolean variable that indicate if the user point contain a loading unit or not.

When executing a loading operation into the vehicle (from station to vehicle), by calling the function `AgvExecLoad(agv,userPoint)` the value of `bPresenza` is set to false and the vehicle status flag, `uStatus`, corresponding to `VST_CARICO_PRESENTE` is set to true.

When executing an unload operation (from vehicle to the station), by calling `AgvExecUnload(agv, userPoint)` the value of `bPresenza` to true. Note that these functions execute a logical load and unload. No commands are sent to agv. To let the agv execute a load or unload operations, `agvRegisterOperation()` must be called.

Some fields can be read and write (`rw`) from the script others are read only (`ro`).

`bVasiPieni` is a variable that indicate the presence of a load on the UDC (Loading Unit).

```

1 //
  // Informazioni associate a punto USER
3 //
object XSiteInfo
5   bool bAttiva           // (rw)
   bool bPriorita         // (rw)
7   bool bPresenza         // (rw) // UDT on board
   bool bVasiPieni        // (rw) // product on borad of UDT
9   bool bInAllarme        // (rw)

```

```

11  bool bVisibile           // (rw)
    uint uTipo             // (rw)
    uint uFlags            // (ro) // Vehicle flags. see UF_***
13  real dStoreTime         // (ro) in giorni
    uint uLato             // (ro) [L (sinistra) | R (destra) | C (centro
    )]
15  uint uExtendedPalletId // (ro) Identificativo informazioni estese
    pallet (se presenti)
endobject

```

Listing 4.3: XSiteInfo

By calling the function `agvGetUserFlags(uint user)`, we get the user point flags (XSite-Info.uFlags).

```

2  //
    // Definizione codici User Flags
    //
4  // Flags riservati ad AgvManager :
    $define UF_MODIFIED           1
6  $define UF_NO_FREQ            2
    $define UF_INUSE             4
8  $define UF_PRE_INUSE         8
    $define UF_PALLET_SU_PERCORSO 32
10 $define UF_MASK_FLAGS_AGVM    4095

12 // Flags impostabili da script ed usati da AgvManager
    $define UF_ACCESSIBLE         0x00001000 // Passaggio attraverso
    user possibile (default vero)
14 $define UF_FORCE_STOP         0x00002000 // Obbligo di spezzare
    movimento
    $define UF_NO_STOP            0x00004000 // Punto di sosta vietata
16 $define UF_NO_STOP_CROSS      0x00008000 // E' vietato fermare l'agv
    su questo incrocio
    $define UF_FORCE_BREAK_CROSS  0x00010000 // Obbligo di spezzare
    movimento su incrocio
18 $define UF_BLINK_ICON         0x00020000 // Se sito in attesa di
    missione o riservato , icona blinka
    $define UF_NO_INVERSIONE      0x00040000 // Vietato fare inversione
    su questo punto
20
    // Flags impostabili da script e non usati da AgvManager
22 $define UF_RESERVED           0x00080000 // Prenotato da agv per
    missione (viene disegnato bollo rosso)
    $define UF_MISS_OK            0x00100000 // Sito in attesa di
    missione (viene disegnato bollo verde)

```

```

24  $define UF_FLAG_XSCRIPT          0x01000000 // Primo flag utilizzabile
    liberamente da script

26  // Esempio d'uso :
    // $define UF_MY_FLAG_1          shl(UF_FLAG_XSCRIPT,1)
28  // $define UF_MY_FLAG_2          shl(UF_FLAG_XSCRIPT,2)

```

Listing 4.4: User point flags

4.1.4 XListaSiti

The xListaSiti store a reference to a site (user o battery point). The operations done on elements of list are applied to sites in map. Imagine the list as a pointer to the sites in map.

```

// ~~~~~
2 //   Gestione lista siti
// ~~~~~

4
object XListaSiti
6   uint pObj                // INTERNAL POINTER – DO NOT TOUCH
   internal 0x02000600 Constructor()
8   internal 0x02000601 Destructor()
   internal 0x02000602 IsEmpty() : bool    // Test se lista vuota
10  internal 0x02000603 Count() : uint      // Ritorna numero siti in
    lista
   internal 0x02000604 Prepend(uint)       // Aggiunge sito in testa alla
    lista
12  internal 0x02000604 AddHead(uint)       // DEPRECATED
   internal 0x02000605 Append(uint)       // Aggiunge sito in coda alla
    lista
14  internal 0x02000605 AddTail(uint)      // DEPRECATED
   internal 0x02000606 Find(uint) : uint  // Torna posizione sito in
    lista (-1 se non trovato)
16  internal 0x02000607 RemoveFirst() : uint // Rimuove il primo sito
    dalla lista , e ne torna il valore
   internal 0x02000607 RemoveHead() : uint // DEPRECATED
18  internal 0x02000608 RemoveLast() : uint // Rimuove l'ultimo sito
    dalla lista , e ne torna il valore
   internal 0x02000608 RemoveTail() : uint // DEPRECATED
20  internal 0x02000609 RemoveAt(uint) : uint // Rimuove il sito alla
    posizione specificata (ritorna l'indice del sito rimosso)
   internal 0x0200060A RemoveAll()        // Svuota la lista
22  internal 0x0200060B At(uint) : uint    // Torna l'indice del sito
    alla posizione specificata
   internal 0x0200060C SetFlag(uint)     // Impostazione flag da settare
    per tutti i siti in lista

```

```

24  internal 0x0200060D AllFlags() : uint    // Torna l'or binario dei
    flags di tutti i siti in lista
    internal 0x0200060E contains(uint) : bool // Test user presente in
    lista
26  endobject

```

Listing 4.5: XListaSiti

4.2 Some useful functions

We already see some callback functions like [onApplicationStart\(\)](#), [onNextMission\(\)](#), [onExpandMacro\(\)](#), [onExecuteMicro\(\)](#) and some utility functions like [agvAddMacro\(\)](#), [agvAddWayPoint\(\)](#), [agvRegisterPassante\(\)](#), [agvRegisterBloccante\(\)](#), [agvRegisterOperation\(\)](#). There are a lot of functions provided by AgvManager. We will see some of them in the examples. We will see also how we can create our own functions and objects.

4.2.1 Movement functions

In the documentation we can find some functions, e.g. [agvMoveToWayPoint\(\)](#), [AgvRegisterMoveTo\(\)](#), etc. to define the movement destination as well as the path [agvAddWayPoint\(\)](#), and some constants.

Constants related to this category of functions begin with [MoveResult_](#) or [EsitoMov_](#), some of these constants are self-explanatory, e.g. [MoveResult_WaypointReached](#), [MoveResult_CompletedMovement](#).

For example if we want to give the final destination without caring about the path we can call [AgvRegisterMoveTo\(\)](#) in the callback function [onExpandMacro\(\)](#) giving to it a input the destination point and [agvManager](#) build the path automatically. The path may be recalculated every time [onExpandMacro\(\)](#) is called, depending on the state of the plant and other Agvs.

If we want to build the path we can use [agvAddWayPoint\(\)](#) and [agvMoveToWayPoint\(\)](#). We register different macros as the way point. We can build the path by waypoints when we compile the macro list, in [registerMission](#) and [registerMovement](#). Then the motion is executed in [onExpandMacro\(\)](#) by calling [agvMoveToWayPoint\(\)](#).

4.2.2 MICRO registration functions

The following functions register a micro operation or instruction:

1. [agvRegisterPassante\(,,,\) \[P\]](#) command, Pass-through operation.

2. `agvRegisterSystemPassante(,,,)` MIC_SYSTEM, system micro instruction.
3. `agvRegisterSystemBloccante(,,,)` MIC_SYSTEM, system micro instruction.
4. `agvReigsterOperation(,,,)` MIC_OPERATION [O] Operation to send to the vehicle.
The syntax of the command is: [Occccmmmm,type,p1,p2,p3,p4].
5. `agvRegisterWait(,,,)` [W] Wait condition operation.
6. `agvRegisterMovingOperation(,,,)` MIC_MOVE [Q] Operation with movement.

To get a list of all micro type search in the documentation the prefix "MIC_".

4.2.3 Points

- `agvUserExists(uint uCode)` : return true if a generic point, user point or cross exist.
- `siteExists(uint uCode)` : return true if the site (USER or CBat point) exists.
- `agvGetSiteInfo(uint userId, xSiteInfo &sInfo)`: get information about USER point with id userId.
- `SetSiteText(uint userId, string text)` : set a text to shown on the user point on the map. e.g. `SetSiteText(userId, "(" + row + ", " + col + ")")`.
- `SetSiteName(uint userId, string text)` : set the name of the site, visible in the tooltip

We can associate two different kind of properties to a point: int or string. Properties can be used by the script as we wish.

- `SetIntProperty(uint, string, int)`
- `IntProperty(uint, string)`
- `addInProperty(,,,)`

```
AddIntProperty(i, PROP_ASSIGNED_AGV, "Assigned agv", ACCESS_INST
, XSitePropertyFlg_volatile)
```

2

4.3 Creating functions and objects

Functions are useful to divided our logic and simplify the program. The keyword `code` is used to create functions. Use the keyword `Forward`, if you want to use the fuction in another function that you implemented before it. It is like the prototype of the function in C language.

Objects are like classes in object oriented programming. Objects can be created by using object and endobjects. Classes have a constructor function, that is called when the class is instantiated, when the the object is created from the class.

4.4 Ex 01: Drag and drop example with loading and loading operations

In this example we will see how we can perform a drag and drop to a user point. A user point represent a working station, that could be machine or simply a position in a store. For example in an automatic store, a user point may represent the position where materials can be stoked or picked. A user point have a property called `bPresenza` that indicate the presence of material in the position designated by the user point or its absence.

OnAgvDroppedToPoint()

In the function `OnAgvDroppedToPoint()`, after the verification of requirements, we will register 3 missions depending on the case if the point is a user point or generic point, if the agv have a load or the user point have a load. In listing 4.6 the code and explanations are shown.

The code that verify the conditions: vehicle exist, in automatic, not enabled, no mission in progress is not shown here. I can be find in the complete example.

Listing 4.6 can be found in the file `agvEventFucntions.xs` in the callback function `OnAgvDroppedToPoint()`.

```

1  // note comments in Xscript begin with ;
3  XSiteInfo sInfo // user point information strucutre
   XVehicleInfo vInfo // vehicle strucutre information
5
   // if user point and vehicle exist
7  if ( AgvGetSiteInfo(uUser , @sInfo) and AgvGetVehicleInfo(uAgv , @vInfo)
   )
9
   bool loadOnAgv , loadOnUser
   // read the bit corresponding to lpad present on agv
11  loadOnAgv = (vInfo.uStatus & VST_CARICO_PRESENTE)
13
   loadOnUser = sInfo.bPresenza
15
   //if both agv and user point have a load

```

4.4 Ex 01: Drag and drop example with loading and loading operations63

```
17     if (loadOnAgv && loadOnUser)
18         MessageBox("Cannot move agv " + (uAgv + 1) + " to " + GetSiteName
19         (uUser) + " : both have a trolley")
20         return
21     endif
22
23     // if only agv have a load , the mission unload to user is
24     registered
25     if (loadOnAgv && not loadOnUser)
26         // call to use defined function
27         RegisterMission(uAgv, MIS_UNLOAD_ONLY, uUser)
28         return
29     endif
30
31     //if only user point have load , the mission load to agv is
32     registerd.
33     if (loadOnUser && not loadOnAgv)
34         RegisterMission(uAgv, MIS_LOAD_ONLY, uUser)
35         return
36     endif
37
38     // register movement to point ,
39     //if there is no lad neither on agv neither on user point ,
40     //or if the point is a generic point
41     RegisterMission(uAgv, MIS_TO_POINT, uUser)
```

Listing 4.6: Drag and drop to user point and generic point

When the user drag and drop the vehicle onto a point, the callback function OnAgv-DroppedToPoint() is called, then the function [RegisterMission\(\)](#) is called inside it as we can in the listing 4.6.

RegisterMission()

The function [RegisterMission\(\)](#) is a user defined function, with the goal to assign missions, can be found in [common.xs](#).

The keyword [forward](#) is used to define a prototype function, it tell the program that somewhere the function is implemented. if forward is not used, and we implement for example a functionA before a fuctionB, and functionA call fuctionB, the program will give error, because he expect that fuctionB is implemented before functionA.

The function have 4 input parameters: [uAgv](#) (agv code), [uCode](#) (mission id), [iPar1](#) and [iPar2](#). Where in this case in [iPar1](#) is passed the point id.

Constants to identify missions and macros are defined as follow:

```

2 // Mission definition. Missions can begin from 0,
  //because there are no missions already defined in AgvManger

4 // No mission in progress
  $define MIS_NULL          0

6
  $define MIS_LOAD_ONLY      10
8  $define MIS_UNLOAD_ONLY   11
  $define MIS_TO_POINT       14

10
  // MACRO definition , begin always from 100
12 // Movement to waypoint
  $define MAC_MOVE_TO_WP     100
14 // Load from the point defined by par1
  $define MAC_LOAD_TROLLEY   102
16 // Unload on the point defined by par1
  $define MAC_UNLOAD_TROLLEY 103

```

In `RegisterMission()` we will start a new mission and fill the `macro list` with MACROs. We will use respectively `agvStartMission()` and `agvAddMacro()`.

As you can notice, a mission is started by calling `agvStartMission(uint agvId, uint missionId, string missionDescription)`. This function return true if a mission is in progress. We can define a new function that return a string value, to get the description of missions. After that we write a select case statement in order to fill the macro list depending on the mission code and to give movement instructions by calling the user defined function `RegisterMovement()`.

For example, if our mission is `MIS_LOAD_ONLY` we register a movement to the user point by calling `RegisterMovement(agvId,userPointId)`, where we will add the macro `MAC_MOVE_TO_WP`, then we add the 2 macros : `MAC_LOAD_TROLLEY` and `MAC_END`. So the macro list have 3 macros, table 4.1. This should be clear, the vehicle first move to the user point, once arrived, load the agv then finish executing the mission.

The same reasoning can be applied for other missions. Following the a part of the code:

```

1 // starting mission "uCode", with description "text"
  if (not AgvStartMission(uAgv, uCode, text))
3   return MIS_NULL
  end
5 // user point info strutcure

```


4.4 Ex 01: Drag and drop example with loading and loading operations65

uAgv	MAC code	iPar1	iPar2	iPar3	iPar4
1	MAC_MOVE_TO_WP	Waypoint id	concatenateNext		
1	MAC_LOAD_TROLLEY	User point code	bVasiPieni		
1	MAC_END	MIS_LOAD_ONLY			

Table 4.1: Macro list of the load mission, MIS_LOAD_ONLY. As you can see the paramters can assume different value types depending on the macro or micro

```

XSiteInfo sInfo
7
// Fill the macro list with the macro for the selected mission
9 // when we call registerMission(), we pass as iPar1 the user point
  index
select (uCode)
11 // Loading agv mission
  case MIS_LOAD_ONLY
13   if (not AgvGetSiteInfo(iPar1 , @sInfo))
      // Strange error. Should not happen!!!
15     AgvStopMission(uAgv)
      return MIS_NULL
17   endif
      // iPar1 = point in store where toilet must be taken
19   RegisterMovement(uAgv, iPar1)
      // Take the trolley with the toilet
21   // Trolley with toilet
      AgvAddMacro(uAgv, MAC_LOAD_TROLLEY, iPar1 , sInfo.bVasiPieni)
23   // END of this mission
      AgvAddMacro(uAgv, MAC_END, uCode)
25   break

27 // unloading agv mission
  case MIS_UNLOAD_ONLY
29   if (not AgvGetSiteInfo(iPar1 , @sInfo))
      // Strange error. Should not happen!!!
31     AgvStopMission(uAgv)
      return MIS_NULL
33   endif
      // iPar1 = point in store where toilet must be taken
35   RegisterMovement(uAgv, iPar1)

37   // Leave the trolley with the toilet
      // Trolley with toilet
39   AgvAddMacro(uAgv, MAC_UNLOAD_TROLLEY, iPar1 , sInfo.bVasiPieni)

```

```

41 // END of this mission
AgvAddMacro(uAgv, MAC_END, uCode)
break
43
45 // movement to a point mission
case MIS_TO_POINT
    //Move to selected point
    RegisterMovement(uAgv, iPar1)
    //END of this mission
47 AgvAddMacro(uAgv, MAC_END, uCode)
    break
49
51 // mission not defined
53 default
    MessageBox("Mission not implemented: " + uCode)
55 return MIS_NULL
end

```

Listing 4.7: RegisterMission() code fragment

The function [RegisterMovement\(\)](#) is self-explanatory.

```

1 code RegisterMovement(uint uAgv, uint userId, uchar destOrientation = '
    X',
    bool concatenateNext = true
3 )
    uint wpidx
5 //add waypoint, return an unique id of the added point.
    wpidx = AgvAddWaypoint(uAgv, userId, destOrientation)
7 // add movement macro related to the point we get previously
    AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
9 end

```

Listing 4.8: RegisterMovement() function

In this case mission are registered by calling the user defined function [RegisterMission\(\)](#). This function was called by the function [OnAgvDroppedToPoint\(\)](#). If we want to assign missions in another way, we can call the function [RegisterMission\(\)](#) inside the callback function [onNextMission\(\)](#) that is called when the agv is enabled.

Independently on how a mission is registered, when a mission is started the callback function [onExpandMacro\(\)](#) is called in order to begin the execution of macros and micors.

onExpandMacro()

[onExpandMacro\(\)](#) is called when there are MACROS in the macro list, check the flowchart in the official documentation and in the previous chapter, in the section mission execution.

4.4 Ex 01: Drag and drop example with loading and loading operations67

To this callback function are passed the agv index, mission index, MACRO index, and 4 parameters. The agv index and mission index are passed from AgvManager to the function, that are related the the list to be expanded. Every mission have its own macro list. The parameters are read from the macro list.

```
1 code OnExpandMacro(uint uAgv, uint uMission, uint iMacroCode,
2     int iPar1, int iPar2, int iPar3, int
3     ) : bool
4
5     select (iMacroCode)
6     case MAC_MOVE_TO_WP
7         // iPar1 = Waypoint id
8         // iPar2 = (bool) do concatenate next macro
9         select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
10 | WpFl_EliminaCompletato))
11         case MoveResult_CompletedMovement ; Completed movement
12         case MoveResult_WaypointReached ; Waypoint reached
13             if (iPar2)
14                 AgvComputeNextMacro(uAgv)
15             endif
16             return true
17         default
18             return false
19         endselect
20         return true
21
22     case MAC_LOAD_TROLLEY
23         // par1 is the point
24         // par2 is true if there is a toilet on the trolley
25         // par3 is true if the trolley is ready to be taken out of store
26         AgvRegisterOperation(uAgv, uMission, O_LOAD, iPar2, iPar3, 0, 0,
27 iPar1)
28         break
29
30     case MAC_UNLOAD_TROLLEY
31         // par1 is the point
32         AgvRegisterOperation(uAgv, uMission, O_UNLOAD, 0, 0, 0, 0, iPar1)
33         break
34
35     case MAC_END
36         SetAgvMessage(uAgv, "")
37         AgvRegisterSystemBloccante(uAgv, uMission, S_END)
38         break
39
40     default
```

```

39     qt_warning("Unknown macro: " + iMacroCode)
        break
41     end
        return TRUE
43 end

```

Listing 4.9: onExpandMacro()

Simply the macro load register on operation of type **O_LOAD**, the unload macro register the **O_UNLOAD** operation and the end macro register the system macro **S_END**. These three macros are already defined by AgvManager. Search in the official documentation the prefixes **O_** and **S_** to find a complete list Operations and System micro type.

The macro **MAC_MOVE_TO_WP** execute the movement command by calling **Agv-MoveToWayPoint(,)** and register a **MIC_MOVE** micro type. When the movement is completed or the waypoint is reached the function return true, that mean the expansion of the macro has finished.

After the expansion of macros, the micro are executed by calling the callback function **onExecuteMicro()**.

OnExecuteMicro()

We see how macros are registered when macros are expanded. Now we see how macros are executed. The **MAC_LOAD_TROLLEY** had registered an **O_LOAD** micro of type **MIC_OPERATION**. The **MAC_UNLOAD_TROLLEY** had registered an **O_UNLOAD** micro of type **MIC_OPERATION** and the macro **MAC_END** had registered an **S_END** of type **MIC_SYSTEM**.

```

1 case MIC_OPERATION
    select (iPar0)
3     case O_LOAD
        if (bLastCall)
5         MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : loaded from "
+ userId)
        SetAgvMessage(uAgv, "")
7         // Agv has finished the load:
        // AgvExecLoad() puts the logical content of the user point
        identified by userId
9         // on the agv, and removes from the user point.
        // NOTE: the operation was sent to the agv in OnExpandMacro()
11        // expanding the macro MAC_LOAD_TROLLEY
        AgvExecLoad(uAgv, userId)

```

4.4 Ex 01: Drag and drop example with loading and loading operations69

```
13     return true
14 else
15     MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : loading from
" + userId)
16     SetAgvMessage(uAgv, "Loading")
17     return false
18 endif
19 break

21 case O_UNLOAD
22     if (bLastCall)
23         MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : unloaded to "
+ userId)
24         SetAgvMessage(uAgv, "")
25         AgvExecUnload(uAgv, userId)
26         return true
27     else
28         MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : unloading to
" + userId)
29         SetAgvMessage(uAgv, "Unloading")
30         return false
31     endif
32     break
33 default
34     qt_warning("Unknown MIC_OPERATION : " + iPar0 + " (mission = " +
iMission + ", par1 = " + iPar1 + ")")
35     break
36 end

37 case MIC_SYSTEM
38     select (iPar0)
39     case S_NULL
40         // Micro of that type are generated by AgvManager, I am not
intereseted on it.
41         break
42     case S_END
43         // End of mission
44         if (vInfo.uStatus & VST_EXEC_COMANDO)
45             MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : wait for agv
commands finished")
46             return false
47         endif
48         MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : finished
executing commands")
49         AgvStopMission(uAgv)
50         SetAgvMessage(uAgv, "")
```

```

51      break
end

```

In the case of **O_LOAD** and **O_UNLOAD**, AgvManager send associated commands to the vehicle. When the vehicle terminate the execution of the command associated to the micro, AgvManger call the callback function **onExecuteMcicro()** with the parameter **bLastCall** is set to true. During the last call, when **bLastCall=true** we can perform also a logical load or unload by calling respectively **AgvExecLoad()** or **AgvExecUnload()**.

The case of **S_END**, the function **AgvStopMission(uAgv)** is called in order to stop terminate the mission execution.

The **MIC_MOVE** are handled by AgvManager not by the script. So is not necessary to write the case of micro movements.

4.5 Overview on some functions

4.5.1 Access level

Sometime in a plant to a worker is permitted to do some job, to maintainer other jobs and so. In AgvManager are defined 5 different levels of users, that correspond to 5 different constants:

```

1  $define ACCESS_USER1 0
2  $define ACCESS_USER2 1
3  $define ACCESS_USER3 2
4  $define ACCESS_INST 3 // Installer
5  $define ACCESS_NO_OP 4

```

The actual level can be read by calling the function **ActualAccessLevel()**.

SetAccessLevelForOperation(DefQual_OpTrascinaAgvSuLinea, ACCESS_INST).

4.5.2 onUpdateIO()

Drag and drop is useful to test vehicle. Normally a vehicle have to respond to some commands and react under some conditions that come from the plant. AgvManager can read input and output from a plc or a database. The callback function **onUpdateIO(.,.)** is used to read input from a plc and write outputs to a plc.

IO can be defined in AgvConfigurator, in the tab PLC we define the communication protocol and the number of DWord (uint 32bit) to be exchanged in input and output. In the tab I/O description we can assign names to digital inputs and outputs. In AgvManager,

in the tab Input/Output [F5] we can see the list of IO, read the value and force inputs and outputs.

The callback function `onUpdateIO` is called at the beginning of the main loop cycle. In this function we can read inputs by calling the function `agvGetInputXXXX` and write outputs by calling `agvSetOutputXXXX`.

There are different get and set functions to read inputs and write outputs, it depends on what and how we read or write. For example, `bool agvGetInput(uint offset)` reads the bit that has index "`offset`" and returns a boolean value depending on the value of that bit. `agvGetInputDWord(uint offset, uint& val)` reads the DWord at the index `offset` and writes the value in `val`, note that `val` is passed to the function by reference.

Note that the first bit has `offset = 1` not 0. It is convenient to define some constants that represent IO signals. For example, the signal `Unloading done`, e.g. a push button connected to input 7, i.e. byte 0 bit 7, we can define a constant like `$define INP_UNLOADING_DONE 7`, then call the function `bool iUnloadDone= AgvGetInput(INP_UNLOADING_DONE)`. The same can be done for Outputs.

4.5.3 OnAgvStatusChange(): Agv status flag change

When the flag status of the vehicle (`xVehicleInfo.uStatus`) changes value, the callback function `OnAgvStatusChange()` is called by AgvManager.

The function `SetAgvStatusDescription(uAgv, int stId, string desc)`. We can set the description of the bit with index `stId`. If we want e.g. to change the description of the status bit `VST_CARICO_PRESENTE` we have to write:

```
1  loadType = TYPE_EMPTY_TROLLEY
   SetAgvStatusDescription(uAgv, -3, "<font color=" + colorName(
       AgvGetTypeColor(TYPE_EMPTY_TROLLEY)) + ">Trolley on agv</font>")
3  AgvSetAgvLoadInfo(uAgv, trolleyOnAgv, loadType, toiletOnTrolley)
```

In this example we change the description into "Trolley on agv", the string is HTML formatted string. This information is shown in the windows "Vehicle information".

The function `AgvSetAgvLoadInfo()` sets information about the Loading Unit (UDC, Unita Di Carico) on the vehicle, e.g. `AgvSetAgvLoadInfo(uAgv, bpresenza, loadType, bVasiPieni)`, `bPresenza` will be `bPalletOnAgv`, `bVasiPieni` will be `bPalletFull`.

4.5.4 OnAgvModeChange(): Agv Operating mode change

When the vehicle operating mode (`xVehicleInfo.uMode`) changes, AgvManager calls the callback function `OnAgvModeChange(uint uAgv, uint oldMode, uint newMode)`.

By calling the function `bool AgvInAutomatico(uAgv)` we get true if the Agv is in automatic mode, i.e. `VM_AUTOMATICO`, or in manual emergency mode, i.e. `VM_MANU_EMERG`.

Look for the prefix `VM_` or `MOD_` to get a list of operating modes, depending on the Agv navigation type.

4.5.5 Settings: XSettings

settings.ini file structure

4.5.6 Interaction with user: XForm

Qt creator

4.5.7 Semaphores

A semaphore can be only set to green by the script. A semaphore is set to red when all Agvs leave the area of the semaphore.

per quanto riguarda i semafori, come si fa ad associare l'agv alla richiesta? in `AgvSetGreenSemaphore(uint, bool)` il primo parametro e' il codice di startSemaforo, non c'e' il numero dell'agv.

Marco, 8:40 PM Una volta che il semaforo è verde lo è per tutti gli agv

8:52 PM `AgvGetSemaphoreRequestMask(uint semaphoreStartId) : uint` // Ritorna stato richiesta semaforo. Torna la maschera degli agv che stanno chiedendo il semaforo come fai a settare i bit della maschera?

Marco, 9:06 PM Li setta `AgvManager` quando degli agv stanno aspettando di passare per l'area del semaforo. Serve che il percorso di un agv incroci l'area del semaforo, e che l'agv sia nelle vicinanze del semaforo stesso: in pratica la maschera viene impostata quando il semaforo sta bloccando il movimento dell'agv. Ovviamente serve anche che venga chiamata una funzione di esecuzione movimento. Viene anche impostata se ad esempio un operatore porta l'agv in manuale di emergenza dentro l'area di pertinenza del semaforo, o se all'avvio del software un agv si trova già dentro l'area.

```

1 ;~~~~~
; Gestione semafori
3 ;~~~~~

#define XSemaforo_nonesiste -1
5 #define XSemaforo_rosso 0
#define XSemaforo_verde 1
7 AgvGetSemaphoreRequestMask(uint semaphoreStartId) : uint //
    Ritorna stato richiesta semaforo. Torna la maschera degli agv che

```



```

    stanno chiedendo il semaforo
AgvSetGreenSemaphore(uint semaphoreStartId , bool)           // Imposta
    conferma via libera semaforo con id semaphoreStartId
9 AgvGetSemaphoreColour(uint semaphoreStartId) : int          // Torna
    colore semaforo con id semaphoreStartId

```

Listing 4.10: Semaphores functions and constants

4.6 Ex 02: Storage example

In this example we will see how to handle a store with one agv. Let's suppose that we have a store with a matrix layout, one loading station and two unloading stations. The loading unit is a pallet. The agv goes to the loading area with an empty pallet, a worker loads the product manually on the pallet, once finished the Agv takes the full pallet to the store. After a specific time the agv takes out the pallets from the store to the unloading area following the FIFO (First In First Out) logic. The Agv leaves the pallet in one unloading area, then goes to the store to take another pallet or to the loading station to take some products. Keep in mind that the Agv needs a pallet on board in order to load products, for this reason the Agv may take an empty pallet from the unloading area (if worker has already freed the pallet from products) or turn back to the store to take an empty pallet then go to the loading area.

4.6.1 Missions

First of all we have to identify the missions in order to accomplish the requirements. It is relatively simple to do it. It is enough to write what the agv has to do in sequence, then figure out some special cases. For example a normal working flow is:

- the Agv has to take a product from the loading station
- then take it into the store
- then take out the product from the store
- then take it to the unloading area

This normal flow is composed by 4 missions. We say that the agv needs a pallet on board in order to load products, so we need to take some pallet from somewhere. Let's say we can take empty pallets from the store or from the unloading area if the worker has already freed the pallet. So we have other 2 missions.

- take empty pallet from store
- take empty pallet from unloading area.

Let's say the worker frees the pallet in the unloading area, and we have to take out a full pallet from the store, before doing this we have to take out the empty pallet from the unloading area. Then take it to the store or to load products, notes that these 2 missions are already identified.

- take empty pallet out from the unloading area

Now suppose that we don't have anything to do and we have to wait to decide what to do, so we have a Null mission.

The agv use battery as power source, so a further mission could be: go to charging station. If the working time finish, at the end of the day we can decide to send the agv to some parking position: home position mission. We can figure out other mission, but for this example these missions are enough. Identified mission are summarized in the following listing:

```

1 // Missions
2 #define MIS_NULL 0 // Nothing to do
3 #define MIS_LOAD_TOILET_FROM_STATION 1 // Go to loading station
  to take a trolley
4 #define MIS_PUT_TOILET_TO_STORE 2 // Take the toilet to the
  store
5 #define MIS_LOAD_TOILET_FROM_STORE 3 // Take something out of
  store
6 #define MIS_LOAD_EMPTY_TROLLEY_FROM_STORE 4 // Take something out
  of store
7 #define MIS_TAKE_TOILET_TO_UNLOAD 5 // Take the toilet already
  on agv to unloading station
8 #define MIS_EMPTY_TROLLEY_FROM_UNLOAD 6 // Go take an empty
  trolley from unload area
9 #define MIS_PUT_EMPTY_TROLLEY_ON_STORE 7 // Agv has an empty
  trolley , put it on store

10 #define MIS_LOAD_ONLY 10
11 #define MIS_UNLOAD_ONLY 11
12 #define MIS_TO_BATTERY_CHARGE 12
13 #define MIS_TO_HOME 13
14 #define MIS_TO_POINT 14

```

Listing 4.11: Agv Missions

The main logic of assigning missions will be implemented in the function `onNextMission()`. After assigning a mission a macro list have to be compiled, and it will be done in another function, called from `onNextMission()`.

4.6.2 onNextMission()

Have a look on the main loop execution, fig.3.9 to get an idea about when the call-back function is called or read the documentation. Now let's assign some mission to the agv.

First let's read the state of the AGV by calling `AgvGetVehicleInfo(uAgv, @agvInfo)`

where **uAgv** is the Agv index and **agvInfo** is a **XVehicleInfo**. From the variable **agvInfo** we can get information about the battery capacity of the Agv, the status of the Agv, etc.

We have to check if the Agv have enough power to execute a mission, and if the Agv have a pallet (UDC) on board or not, if it have a pallet, it is empty or have some product? Let's define two flags in the agv flag status, **agvInfo.uStatus**. One to determine if the full pallet on the agv come from the store or from the loading area? For example **agvflg_ProductGoingOutOfStore = TRUE** then it is a pallet that come out from the store and should go to unloading area, otherwise if equal false, it come the loading area and have to go to the store. The other flag to determine if the pallet is empty or not, **agvflg_ProductOnPallet**.

```

1 // user defined vehicle status flags
  $define agvflg_ProductOnPallet      0x0100 // bit 8 in agvInfo.
    uStatus
3 $define agvflg_ProductGoingOutOfStore 0x0200 // bit 9 in agvInfo.
    uStatus

```

Then let's defined some boolean variables that will help in the decision and selection of missions:

```

1 bool mustGoToChargeBattery // no pwer, Agv must go the cahrg
  bool trolleyOnAgv // Agv have pallet or loading unit, don't consider
    the product
3 bool toiletOnTrolley // Agv have a full pallet = Loading unit +
  product
  bool takeOutToilet // Full pallet come out from store
5
  // Unloading unit have empty trolley to take away
7 bool mustRemoveTrolleyFromUnload

9 mustGoToChargeBattery = (agvInfo.dBatteryPerc <= MIN_BATTERY)

11 trolleyOnAgv = (agvInfo.uStatus & VST_CARICO_PRESENTI)
  toiletOnTrolley = trolleyOnAgv and (agvInfo.uStatus &
    agvflg_ProductOnPallet)
13 takeOutToilet = toiletOnTrolley and (agvInfo.uStatus &
  agvflg_ProductGoingOutOfStore)

```

Listing 4.12: Desision variables or plant status

Depending on the value of these variables we can assign mission to Agv. For example if the agv have a full pallet on board and that pallet come out from the store, it should go

to the unloading area.

```

1 if (takeOutToilet)
2   OnNextMissionDebugMessage(uAgv, "takeOutToilet=T : <font color=green>
   assign mission MIS_TAKE_TOILET_TO_UNLOAD</font>")
   return RegisterMission(uAgv, MIS_TAKE_TOILET_TO_UNLOAD)
4 endif

```

Listing 4.13: Mission to unloaing area

The function `OnNextMissionDebugMessage()` is used to show debugging messages in the vehicle information [F3] tab, in the box **Mission generation**.

If the Agv have a full pallet and the pallet doesn't come out from the store, it come from the loading are, the agv have to take the pallet to the store:

```

1 if (toiletOnTrolley)
2   int storePosition
   // Find a position on the store where to put the trolley
4   storePosition = store_hnd.positionForTakeInTrolley(uAgv, true)
   if (not SiteExists(storePosition))
6     OnNextMissionDebugMessage(uAgv, "toiletOnTrolley=T : <font color=
       red>position in store not found</font>")
       return MIS_NULL
8   endif
   OnNextMissionDebugMessage(uAgv, "toiletOnTrolley=T, storePosition=" +
       storePosition + " : <font color=green>assign mission
       MIS_PUT_TOILET_TO_STORE</font>")
10  return RegisterMission(uAgv, MIS_PUT_TOILET_TO_STORE, storePosition)
   endif

```

Listing 4.14: Mission take full pallet into store

If we look at the code again without the debugging info, it is simple. First we choose a position in the store where to go, then we assign the mission take trolley to the position chosen in the store.

```

1 if (toiletOnTrolley)
2   int storePosition
3   // Find a position on the store where to put the trolley
   storePosition = store_hnd.positionForTakeInTrolley(uAgv, true)
5   //check if postion exist or storePosition <> (-1), register mission.
   return RegisterMission(uAgv, MIS_PUT_TOILET_TO_STORE, storePosition)
7 endif

```

Listing 4.15: Mission take full trolley to a postion in the store

As we see the logic to assign Missions should not be complicated. Once missions are identified, it is enough to assign them to Agv without caring about the details of a mission. To do so, we call a user defined function `RegisterMission(uAgv, uMission, iPar1, iPar2)`, or any number of parameters we need. The detail about mission step (MACROs) are implemented in the function `registerMission`.

Until now we assign 2 missions `MIS_TAKE_TOILET_TO_UNLOAD` and `MIS_PUT_TOILET_TO_STORE`, depending on the conditions `takeOutToilet` and `toiletOnTrolley`.

Pay attention to the sequence of implementing the functions or to the conditions. It is better to write if (toiletOnTrolley and not takeOutToilet) then writing if (toiletOnTrolley), in this way we don't care about the sequence of writing the conditions.

4.6.3 Macros

Once missions are defined, we have to defined MACROs. First take a look at macros defined by AgvManager.

```

1 // MACRO code definition
  $define MAC_NULL          0
3 $define MAC_MOVE_TO_USER  1
  $define MAC_MOVE_TO_XY    2
5 $define MAC_CHARGE_BATT   3
  $define MAC_CHARGE_STOP   4
7 $define MAC_LOAD          5
  $define MAC_UNLOAD        6
9 $define MAC_END           7
  $define MAC_MOVE_AND_LOAD  8
11 $define MAC_MOVE_AND_UNLOAD 9

```

Listing 4.16: MACRO defined in AgvManager

If other macros are needed we can define also ours.

```

1 // Movement to waypoint
  $define MAC_MOVE_TO_WP      100
3 // Wait for the amount of seconds specified in par1
  $define MAC_WAIT_S          101
5 // Load a trolley from the point defined by par1
  // par2 is true if there is a toilet on the trolley
7 $define MAC_LOAD_TROLLEY    102
  // Unload a trolley on the point defined by par1
9 $define MAC_UNLOAD_TROLLEY  103

11 // Wait for the operator to load toilet on agv
  $define MAC_WAIT_TOILET     200

```

```

13 // Decide the unloading point where the toilet will be unloaded
    $define MAC_TAKE_TOILET_TO_UNLOAD 201
15 // Decide where to take an emty trolley: whether to load station or
    to store
    $define MAC_DECIDE_EMPTY_TROLLEY_DEST 202

```

Listing 4.17: MACRO defined by user

To define Macros, we go back to the list of missions, and write the detail of the missions. Once again without too much detail, because the instructions and operations sent to and received from agv are handled by MICROS. For our example we can assign the following macros to each mission. Usually this is done in register mission, or a function called from onNextMission().

- MIS_LOAD_TOILET_FROM_STATION 1
 - MAC_MOVE_TO_WP , move to station user point
 - MAC_WAIT_TOILET , wait loading toilet, wait for signal load ok
 - MAC_END
- MIS_PUT_TOILET_TO_STORE 2
 - MAC_MOVE_TO_WP , move to the store position
 - MAC_UNLOAD_TROLLEY , unload trolley
 - MAC_MOVE_TO_WP , go out from store
 - MAC_END , end macro
- MIS_LOAD_TOILET_FROM_STORE 3
 - MAC_MOVE_TO_WP
 - MAC_LOAD_TROLLEY
 - MAC_MOVE_TO_WP
 - MAC_END
- MIS_LOAD_EMPTY_TROLLEY_FROM_STORE 4
 - MAC_MOVE_TO_WP
 - MAC_LOAD_TROLLEY
 - MAC_MOVE_TO_WP
 - MAC_END
- MIS_TAKE_TOILET_TO_UNLOAD 5
 -
 - MAC_MOVE_TO_WP
 - MAC_TAKE_TOILET_TO_UNLOAD
 -
- MIS_EMPTY_TROLLEY_FROM_UNLOAD 6
 - MAC_MOVE_TO_WP

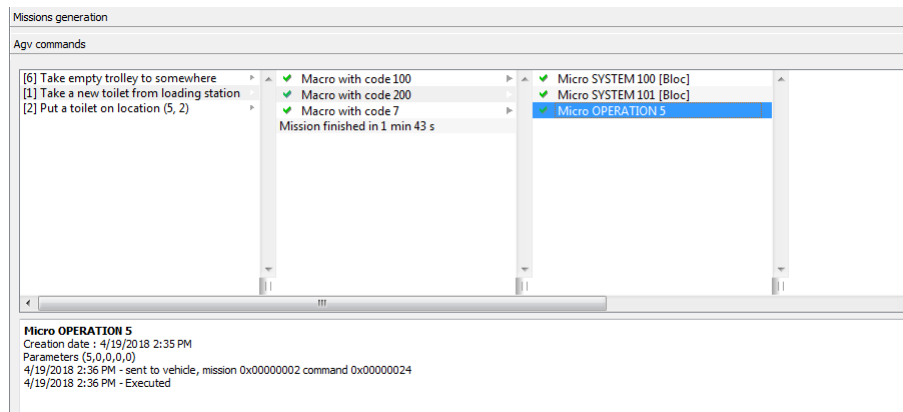


Figure 4.1: Macro expansion and micro details

- MAC_LOAD_TROLLEY
- MAC_MOVE_TO_WP
- MAC_DECIDE_EMPTY_TROLLEY_DEST
- MIS_PUT_EMPTY_TROLLEY_ON_STORE 7
 - MAC_MOVE_TO_WP , move to the store position
 - MAC_UNLOAD_TROLLEY , unload trolley
 - MAC_MOVE_TO_WP , go out from store
 - MAC_END , end macro

Once MACROS are assigned to missions, we have to assign MICROs to MACROS, this may be done in `onExpandMacro()`. Remember that when `onExpandMacro()` return true, mean the execution of the current macro is concluded, and the next macro in the list will be expanded (executed) on the next call of `onExpandMacro()`, of course if the current macro is not the last one in the list.

We can also end a mission and begin another one from `onExpandMacro()`. We don't have to do it always in `onNextMission()`. But keep in mind that the program should be linear and simple. Avoid **spaghetti code** when is possible.

When a macro is expanded we can see the result in vehicle information[F3], fig.4.1.

For example to make a loading operation, we have to call `AgvRegisterOperation()`, as in listing 4.18.

```

1 case MAC_LOAD_TROLLEY
   // par1 is the point
3 // par2 is true if there is a toilet on the trolley
   // par3 is true it the trolley is ready to be taken out of store
5 AgvRegisterOperation(uAgv, uMission, O_LOAD, iPar2, iPar3, 0, 0,
   iPar1)
   break

```

Listing 4.18: Loading MACRO

The call of `AgvRegisterOperation` in 4.18, register a MICRO of type `MIC_OPERATION`.

4.6.4 Micros and Operations

Micros are low level set of instructions. The following listing.4.19 show the different categories of micro instructions or operations defined by `AgvManager`.

```

1 // Definizione codici micro
#define MIC_NULL 0
3 #define MIC_MOVE 1 // M command, agvregisterMove****
#define MIC_CURVE 2 // M command
5 #define MIC_ROTATION 4 // M command
#define MIC_OPERATION 5 // O command, agvregisterOperation()
7 #define MIC_SYSTEM 6 // command are not sent to vehicle ,
    agvregistersystembloccante(), agvregistersystempassante()
#define MIC_PASSANTE 7 // P command, agvregisterPassante()
9 #define MIC_WAIT 8 // W command, agvregisterWait()
#define MIC_MOVING_OPERATION 9 // Q command,
    agvregisterMovingOperation()

```

Listing 4.19: Different category of MIC defined in `AgvManager`

The following are MICROs defined by `AgvManger`.

```

1 // Definizione codici operazioni
#define O_LOAD 2
3 #define O_UNLOAD 3
#define O_CHARGE 4
5 #define O_CHARGE_START 1
#define O_CHARGE_STOP 2
7
// Definizione codici micro System
9 #define S_NULL 0 // Serve (ad esempio) a spezzare le MIC_MOVE
#define S_END 1
11 #define S_CHARGE_WAIT 3
#define S_CHARGE_START 4
13 #define S_CHARGE_STOP 5
#define S_CONCAT_MACRO 8 // Concatena immediatamente la macro
    successiva

```

Listing 4.20: MICRO and OPERATIONS defined by `AgvManager`

We can define our own Micros and operations. Try to follow the naming style of `AgvManager`. Begin with the prefix `O_` for Operation category, with `S_` for System category.


```

1 // Micro SYSTEM
2 #define S_START_WAIT    100
3 #define S_EXEC_WAIT     101
4
5 // Micro OPERATION
6 // Wait toilet on agv
7 #define O_WAIT_TOILET   5

```

Listing 4.21: MICRO and OPERATIONS defined by user

Micros from category MIC_OPERATION, MIC_SYSTEM, MIC_PASSANTE, MIC_WAIT are assigned in onExpandMacro(), using one of the following functions:

```

1 AgvRegisterPassante ()
2 AgvRegisterWait ()
3 AgvRegisterSystemPassante ()
4 AgvRegisterSystemBloccante ()

```

Micros that belong to MIC_MOVE, MIC_CURVE, MIC_ROTATION are assigned by AgvMoveTo****() functions.

Micro execution is done in onExecuteMicro(), for example:

```

1 case MIC_SYSTEM
2   case S_END
3     ; End of mission
4     AgvStopMission(uAgv)
5     SetAgvMessage(uAgv, "")
6     break
7
8   case S_START_WAIT
9     // start timer. Not locking micro
10    timerWait[uAgv] = timeoutS(iPar1)
11    break
12
13   case S_EXEC_WAIT
14     // wait a timer to finish counting. locking micro
15     if (isTimeout(timerWait[uAgv]))
16       return true
17     else
18       MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : wait " + int(
19         secsToTimeout(timerWait[uAgv])) + "s")
20       return false
21     endif
22     break

```

```

22
case S_WAIT_INPUT
24 // wait a signal from plc. locking micro
    if ( AgvGetInput(INP_LOAD_TERMITAED) ==false )
26         return true
    else
28         return false
    endif

```

Keep in mind that when a micro terminate, onExecuteMicro() should return true. For example, if we are waiting for a signal to be false, and the signal is true, onExecuteMicro() return false, in this way the next micro will be the current one. When the signal become false, onExecuteMicro() return true and the execution of the current micro terminate.

When bLastCall will be set to true? at the next execution or immediately, and the effective termination will be at the next call?

```

case O_LOAD
2  if ( bLastCall )
    MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : loaded from " +
        userId )
4  SetAgvMessage(uAgv, "")
    ; Agv has finished the load:
6  ; AgvExecLoad() puts the logical content of the user point identified
    by userId
    ; on the agv, and removes from the user point.
8  ; NOTE: the operation was sent to the agv in OnExpandMacro()
    ; expanding the macro MAC_LOAD_TROLLEY
10 AgvExecLoad(uAgv, userId)
    ; Position is no more reserved to agv
12 store_hnd.unassignAgvToLocation(uAgv, userId)
    $ifdef EMULAZIONE_AGV_ATTIVA
14 if ( EMUAGV_IsAgvEmulated(uAgv) )
    if ( iPar1 )
16 vInfo.uStatus = vInfo.uStatus | agvflg_toiletOnTrolley
    if ( iPar2 )
18 vInfo.uStatus = vInfo.uStatus | agvflg_toiletGoingOutOfStore
    else
20 vInfo.uStatus = vInfo.uStatus & ~agvflg_toiletGoingOutOfStore
    endif
22 else
    vInfo.uStatus = vInfo.uStatus & ~(agvflg_toiletOnTrolley |
        agvflg_toiletGoingOutOfStore)
24 endif

```

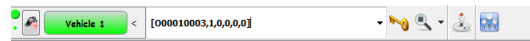


Figure 4.2: Commands insertion

```

26  EMUAGV_SetFlags(uAgv, vInfo.uStatus)
    endif
    $endif
28  return true
    else
30  MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : loading from " +
    userId)
    SetAgvMessage(uAgv, "Loading")
32  return false
    endif
34  break

```

Listing 4.22: Loading MIC OPERATION

For example if we register a micro from the category Operation, AgvManager will send some command to the vehicle. When the vehicle answer with operation concluded, AgvManager will set bLastCall to true, in this way we can terminate the micro execution.

From AgvManager we can send command to agv from the interpreter box. For example the operation command structure is: [Occccmmmm,type,p1,p2,p3,p4]. For example [O00010003,1,0,0,0,0], fig.4.2, we send to Agv an operation command [O], with operation number 1 and mission number 3.

Details about mission, macro and micro execution can be seen in the tab vehicle informations[F3].

4.6.5 Store handling

In a matrix store, or stack, products can be taken out following the logic of First in First out (FIFO) or Last in First out(LIFO), etc.

We can define some objects to handle the store, keep in memory the products in the store and track them, and also some algorithm to take out or take in products.



Motion controller RP-1

5	RTE - Real Time Extended	89
5.1	Multitasking	
6	RDE - Robox Development Environment	93
6.1	First step	
6.2	New RTE project	
6.3	Tools	
6.4	Hardware configuration	
6.5	Programming languages	
6.6	Programs	
6.7	Axis control example	
7	Motion control	107
7.1	Basics	
7.2	Example with R3	
7.3	Example with OB	
8	AGV commissioning	109
8.1	MGV mechanical and electrical part	
8.2	Dispan : Display panel	
8.3	Commissioning	
8.4	AGV ecosystem	
8.5	AgvManager-RP1 communication protocol	
8.6	Hardware configuration	
8.7	Program description	

Robox controller RP1, is a motion control parts, description,.....

RDE need [Microsoft c++ redistributable 2010 and 2015 x86](#)

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).



5. RTE - Real Time Extended

5.1 Multitasking

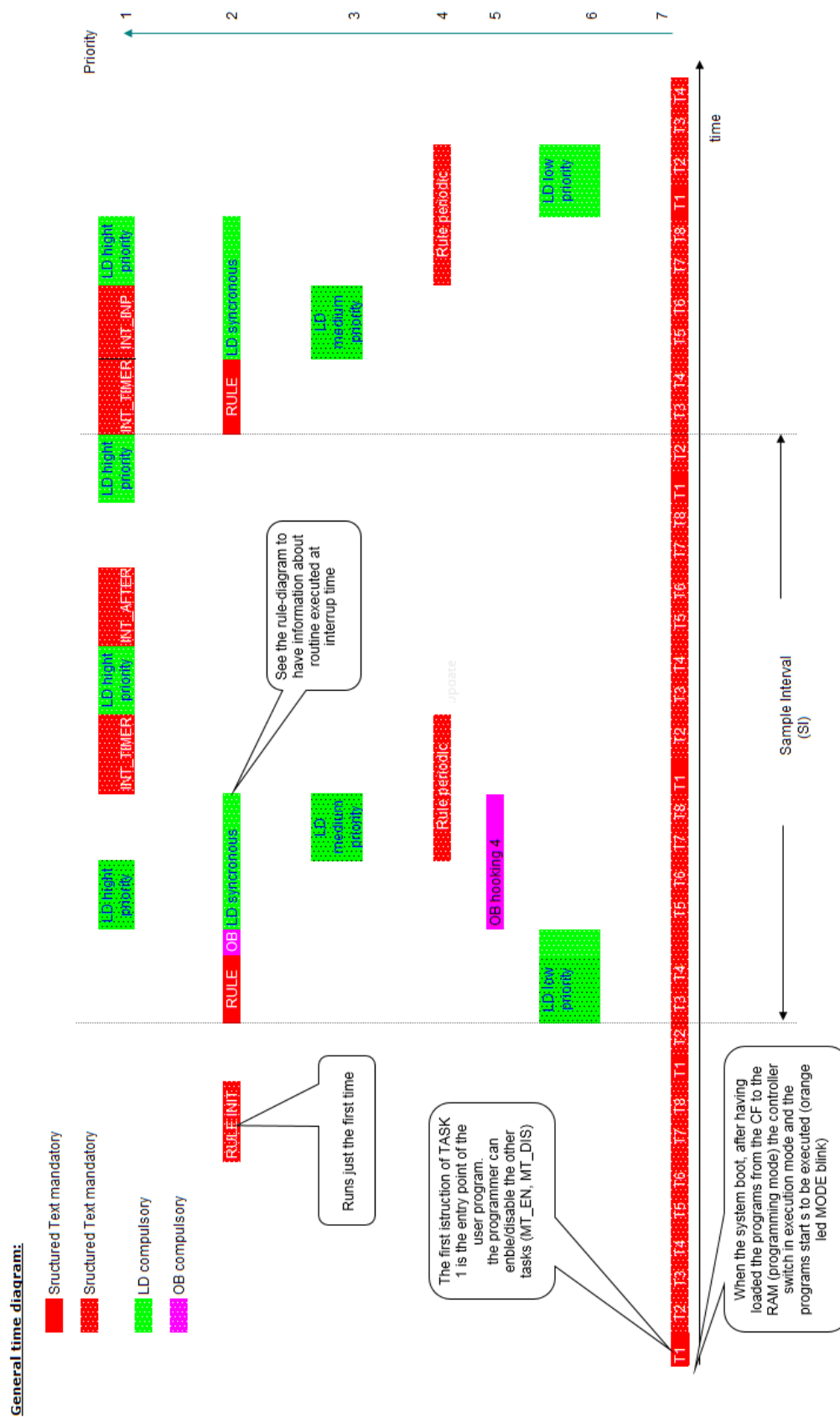
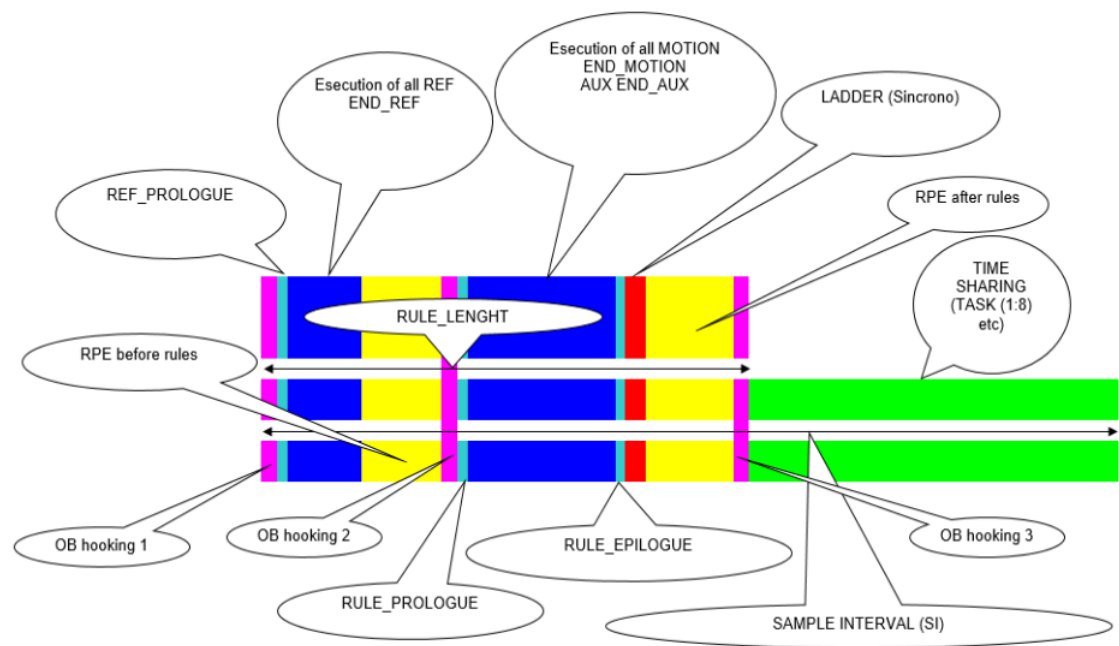


Figure 5.1: RTE multitasking

Priority	Task type
7	<p>TASK in BACKGROUND (time-sharing)</p> <p>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).</p> <p>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.</p> <p>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.</p> <p>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</p>
6	<p>LOW PRIORITY LADDER TASKS</p> <p>Ladder task whose execution frequency is programmed by the user (1Hz+2000Hz). Its length can be viewed through the predefined variable ltl_length</p>
5	OB service.
4	<p>RULE PERIODIC</p> <p>RULE executed at a frequency programmed by the user.</p> <p>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 + SYNCHRONOUS LADDER TASKS + HIGH PRIORITY LADDER TASKS + TASK ON EVENT, if present).</p>
3	<p>NORMAL PRIORITY LADDER TASKS</p> <p>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</p>
2	<p>RULES (fixed frequency functions -interrupt-)</p> <p>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)</p> <p>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</p> <p>RULE_INIT</p> <p>RULE executed just once, before the execution of the other RULES. It is executed the first time the instructions ORDER or GROUP is invoked.</p> <p>No RULE is active until the GROUP instruction has been executed or the predefined variable RC is set.</p> <p>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</p>
2	<p>SYNCHRONOUS LADDER TASK</p> <p>Task written in Ladder language, executed (if present) together with the RULES. Its length can be viewed through the predefined variable ltl_length</p>
1	<p>TASK ON EVENT</p> <p>They are particular tasks written in R3 language with max system priority and which are used to solve some particular requirements.</p> <p>The enabling events are:</p> <ul style="list-style-type: none"> *Variation edge of a digital input (INT_INP) *Set frequency (INT_TIMER) *Time delay (INT_AFTER) <p>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 performance can result in a degradation of the system's normal performance.</p>
1	<p>HIGH PRIORITY LADDER</p> <p>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</p> <p>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 performance can result in a degradation of the system's normal performance.</p>

Figure 5.2: 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 meticulous programmers
RULE_EPILOGUE	Optional function which, if defined, is executed immediately after the MOTION AUX fields
LADDER	SYNCHRONOUS LADDER TASK. Use the command LAD_STATUS to get information on the execution timing
SINCRONO	
RPE	Handling of a group of axes by RPE with selection "before the rules"
OB hooking 3	Object Blocks hooked hook3

Figure 5.3: Rule execution



6. RDE - Robox Development Environment

6.1 First step

In order to getting started with the controller, we need a memory card where we have to copy RTE and some configuration file. A new memory card will have the folder KEY that is generated by Robox for license.

After the installation of RDE, RCE and Icmapi we need to copy in the installation folder the license in order to compile programs. The license is provided by Robox.

Before creating a new project, a workspace have to be created. A workspace may contain more than one project. In the menu bar, the workspace menu, allow to open, create and manage workspaces, also to access the predefined examples .

6.2 New RTE project

6.3 Tools

From the workspace we can access the tools provided by RDE. Different kind of tools are provided to debug and monitor the software: panels, oscilloscope, variable monitors and command shell.

6.3.1 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

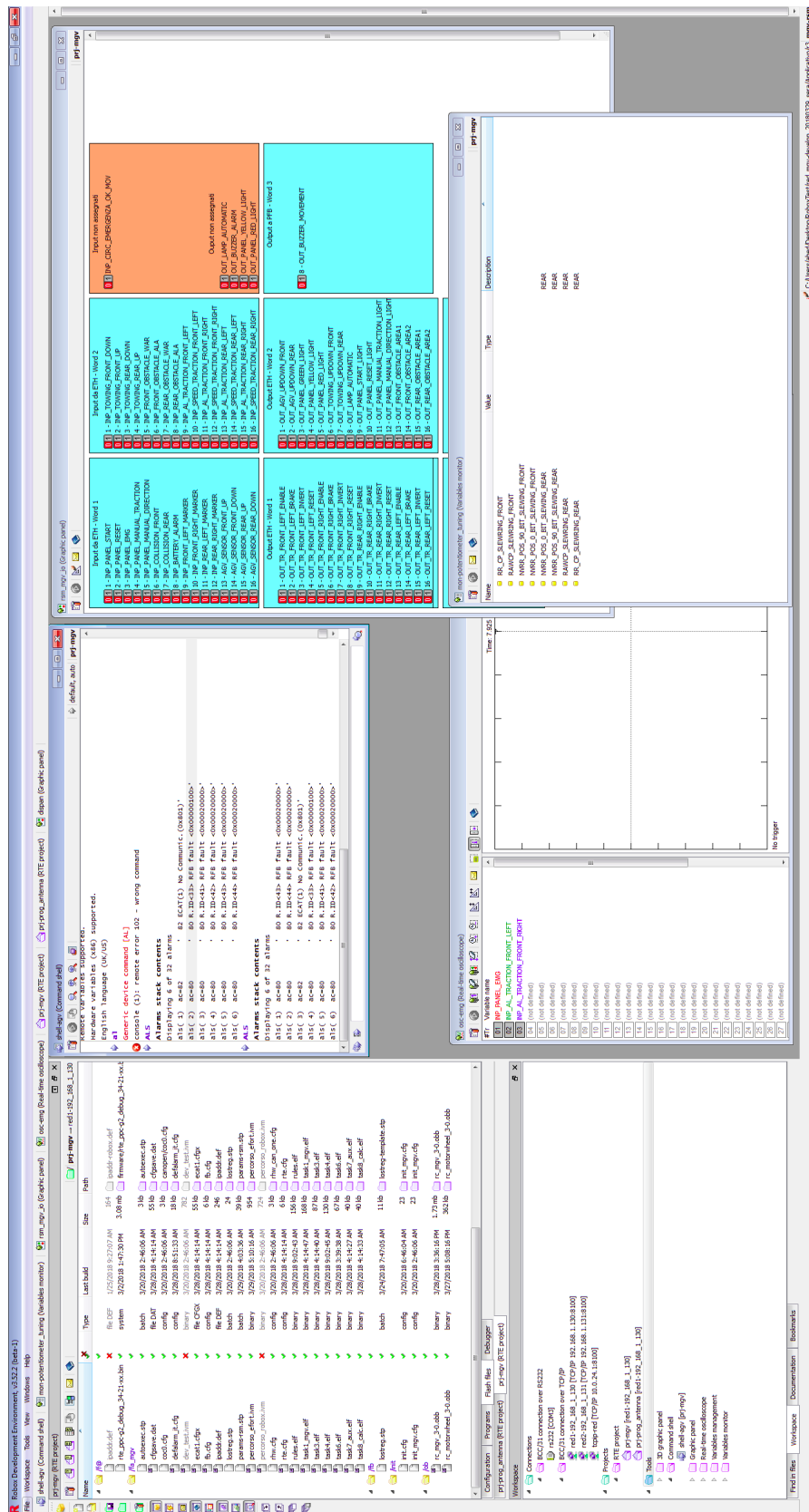


Figure 6.1: RDE main windows

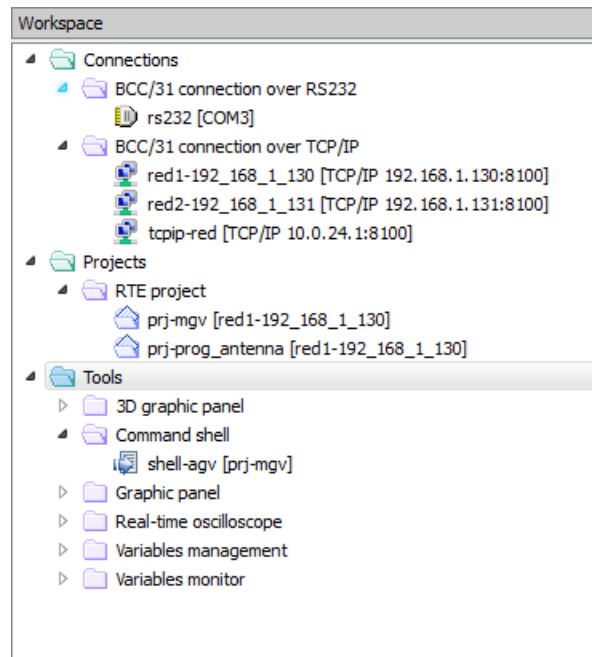


Figure 6.2: Workspace

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..

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.

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 fig.6.3 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

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

6.3.2 Oscilloscope

6.3.3 Graphic panel

6.4 Hardware configuration

6.4.1 Important folders and files

Add new folder fig.6.4

rte-xxxx.bin

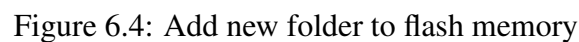
rte.cg

ipaddr.def

rhw.cfg

lostreg.stp

init.cfg



6.4.2 Registers

6.4.3 Bus configuration

6.4.4 Axes configuration

6.5 Programming languages

6.5.1 R3

6.5.2 Predefined variables

fr feed rate

kff feed forward factor

pro_gai position loop proportional gain

epos position error when the position loops are closed with a predefined formula

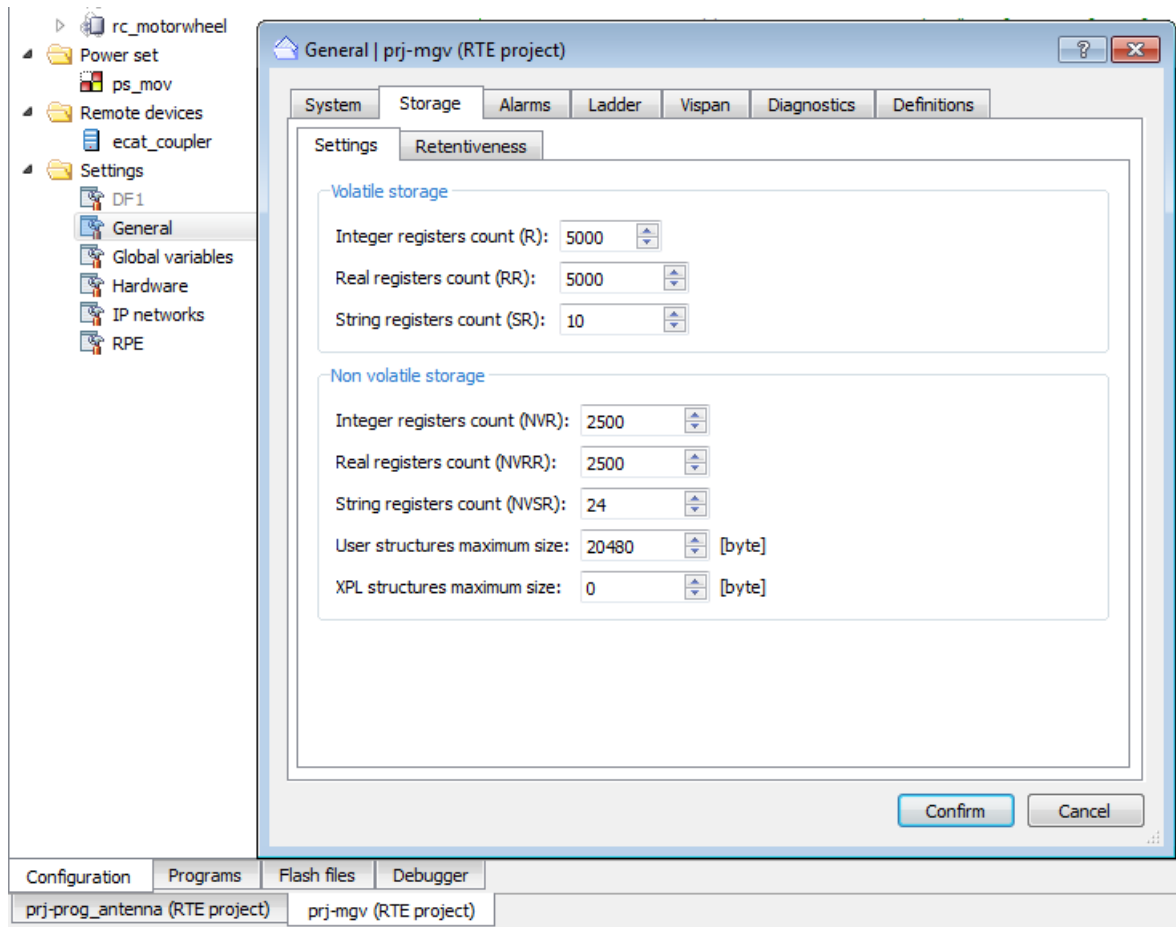


Figure 6.5: Register dimension

6.5.3 Object block

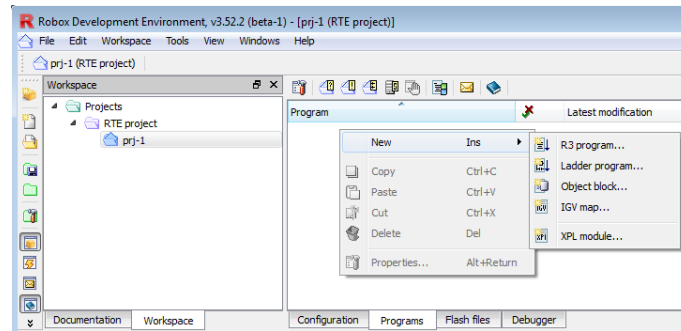
Object block is a C++ class, it is another option to write program in RDE. It is composed from a header file and a source file like any C++ class, in addition to these classic files, RDE use the obs file to describe the interface of the Object block.

In rte project, right click and add new Object block fig.6.6. A folder have to be selected for the compiled file. If the folder ob dosen't exist add it in the flash memory, see section files and folders.

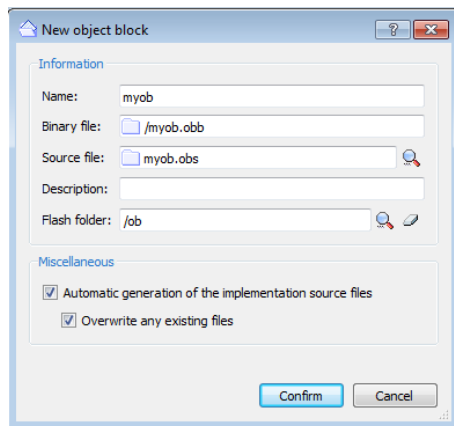
After the creation of a new object block we will obtain 4 files:

1. obs : object block interface file
2. h : C++ header
3. cpp : C++ source
4. obb : Object block binary file (compiled file)

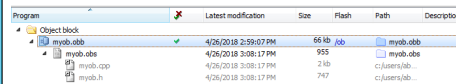
Fig.6.7, 6.8 and 6.9 show the auto generated files. As we can see the header and the source files have the structure of a classic C++ class with class name, class constructor and destructor.



(a) Create new Object block. right click in the tab program of an RTE project



(b) Write the OB name, select the folder of destination and check at least the first option.



(c) Object block structure files

Figure 6.6: New object block

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. The source file where the code is implemented is written in the block implementation. Fig.6.10 show an example of an obs file.

As any object oriented 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, fig.6.11. A class could have more than one instance. An OB is similar to an FB (Function block) in PLC programming.

Suppose we have the class `rc_mgv` and its instance `agv`, as in fig.6.11. We can call in R3 the method `get_status` of the class `rc_mgv` as we call it in C++: `agv.get_status(AgvStatus_t4)`. We can access properties using also the dot operator for



```

1  ;=====
2  ;
3  ; ROBOX SpA
4  ; Via Sempione 82, Castelletto Ticino, ITALY
5  ; +390331922086
6  ; http://www.robbox.it
7  ;
8  ;-----
9  ; Job number      :
10 ; Title           : Class MYOB project
11 ; Platform        : RTE
12 ; Generator       : Robox Development Environment, v3.52.2 (beta-1)
13 ;=====
14
15 define DEBUG_MYOB      ; Enable DEBUG for the class
16
17 object_block myob
18
19     ; General object block information
20     title
21     version 1.0.0
22     info
23
24     end_info
25
26     ; Class structures
27     structures
28     end_structures
29
30     ; Class properties
31     properties
32     ; Use 'ro' data modifier for read-only properties
33     ; 'ba' data modifier for bit access enabled properties
34     end_properties
35
36     ; Class methods
37     methods
38     end_methods
39
40     ; Implementations
41     implementation
42     source "myob.cpp"
43     end_implementation
44
45 end_block
46

```

Figure 6.7: Auto-generated OBS file

reading or writing: `agv.DRIVING_MODE_TAPE = 4` or `if(agv.TAPE_FOLLOW_LEFT)`.

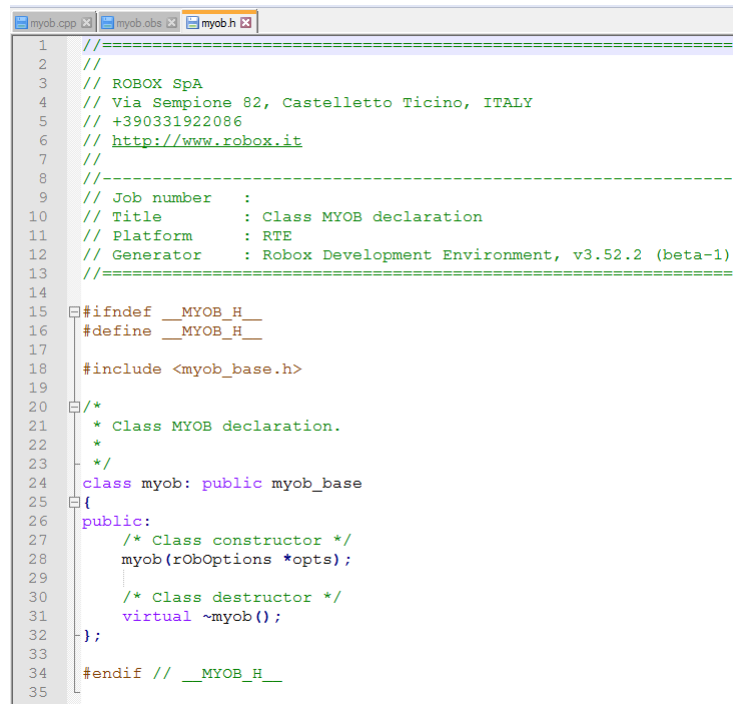
If we defined a structure in the obs file we can use it to define a variable of that type in R3 using the dot operator.

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 and a rules and task 1 implementation. The Class `rc_belt` is an OB that can be find in the Object Block library. 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`, we can see the interface of the Class, how to use another class by importing it, define inputs and outputs and some methods. Note that input and outputs deffer only with the keyword `ro`. When an property is declared as read only behave like an



```

1 //=====
2 //
3 // ROBOX SpA
4 // Via Sempione 82, Castelletto Ticino, ITALY
5 // +390331922086
6 // http://www.robox.it
7 //
8 //-----
9 // Job number      :
10 // Title           : Class MYOB declaration
11 // Platform        : RTE
12 // Generator        : Robox Development Environment, v3.52.2 (beta-1)
13 //=====
14
15 #ifndef __MYOB_H__
16 #define __MYOB_H__
17
18 #include <myob_base.h>
19
20 /*
21  * Class MYOB declaration.
22  */
23
24 class myob: public myob_base
25 {
26 public:
27     /* Class constructor */
28     myob(rObOptions *opts);
29
30     /* Class destructor */
31     virtual ~myob();
32 };
33
34 #endif // __MYOB_H__
35

```

Figure 6.8: Auto-generated C++ header

output, otherwise behave like an input.

In the implementation we can see the call to 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`.

A detailed example about OB implementation will be provided in the chapter related to motion control.

6.6 Programs

6.6.1 Tasks

6.6.2 Rules

6.6.3 R3 example program

6.6.4 OB example program

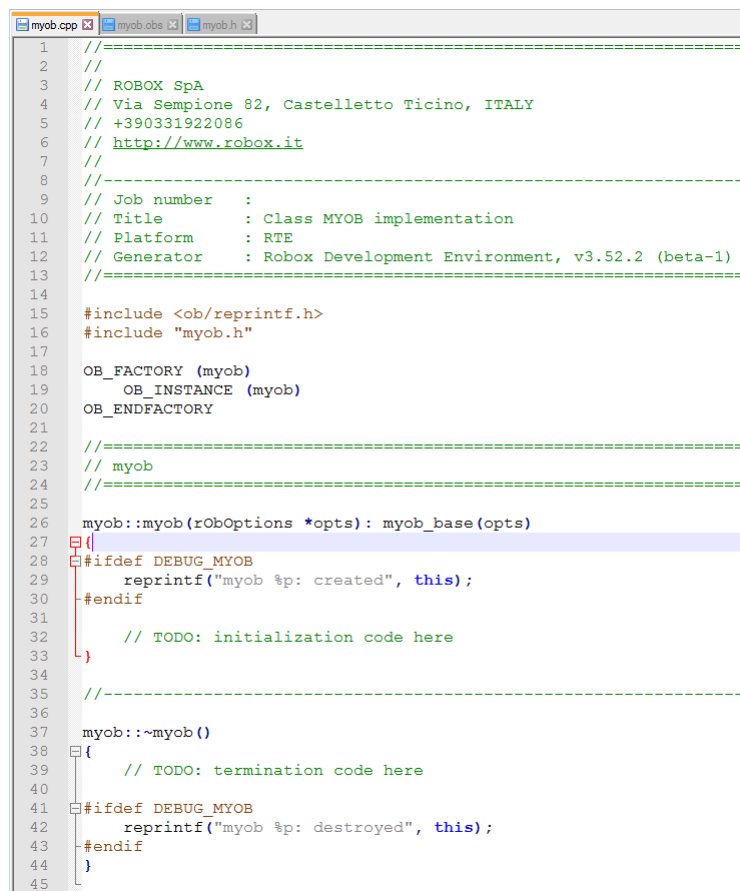
6.7 Axis control example

6.7.1 Axis configuration

6.7.2 Power set and power handling

6.7.3 Velocity control

6.7.4 Position control



```

1  //=====
2  //
3  // ROBOX SpA
4  // Via Sempione 82, Castelletto Ticino, ITALY
5  // +390331922086
6  // http://www.robox.it
7  //
8  //-----
9  // Job number      :
10 // Title           : Class MYOB implementation
11 // Platform        : RTE
12 // Generator       : Robox Development Environment, v3.52.2 (beta-1)
13 //=====
14
15 #include <ob/reprintf.h>
16 #include "myob.h"
17
18 OB_FACTORY (myob)
19   OB_INSTANCE (myob)
20 OB_ENDFACTORY
21
22 //=====
23 // myob
24 //=====
25
26 myob::myob(roboOptions *opts): myob_base(opts)
27 {
28 #ifdef DEBUG_MYOB
29   reprintf("myob %p: created", this);
30 #endif
31
32   // TODO: initialization code here
33 }
34
35 //-----
36
37 myob::~myob()
38 {
39   // TODO: termination code here
40
41 #ifdef DEBUG_MYOB
42   reprintf("myob %p: destroyed", this);
43 #endif
44 }
45

```

Figure 6.9: Auto-generated C++ source

```

define DEBUG_DYNCLASS      ; Enable DEBUG for the class
object_block dynclass

; 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
real torquedirection[6]
real reductionratio[6]
real ratedtorque[6]
real al[6],a[6],d[6],th[6]
real para1[17], para2[18], para3[12], para4[12], para5[12], para6[12]
real loadparas[7]
;output
real dyntorque[6]
end_properties

; Class methods
methods
int faldyntorque(double q[6], double qd[6], double qdd[6])
end_methods

; Implementations
implementation
source "dynclass.cpp"
end_implementation
end_block

```

Figure 6.10: Obs example file

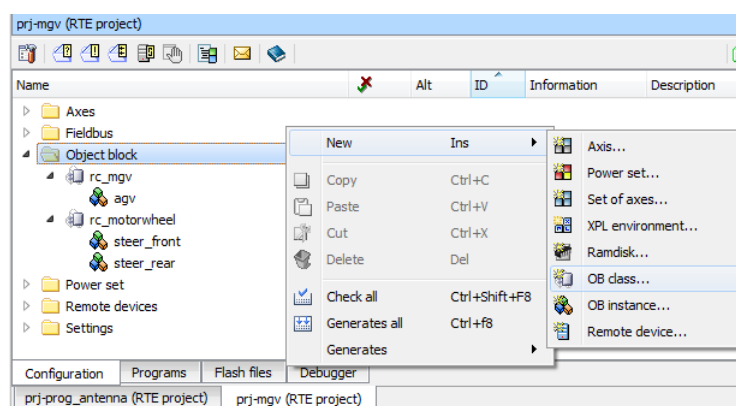


Figure 6.11: OB Class or instance. Add a class then add an instance. In the figure we can see 2 classes : rc_mgv and rc_motorwheel, and one instance of the first class and two instances of the second one.

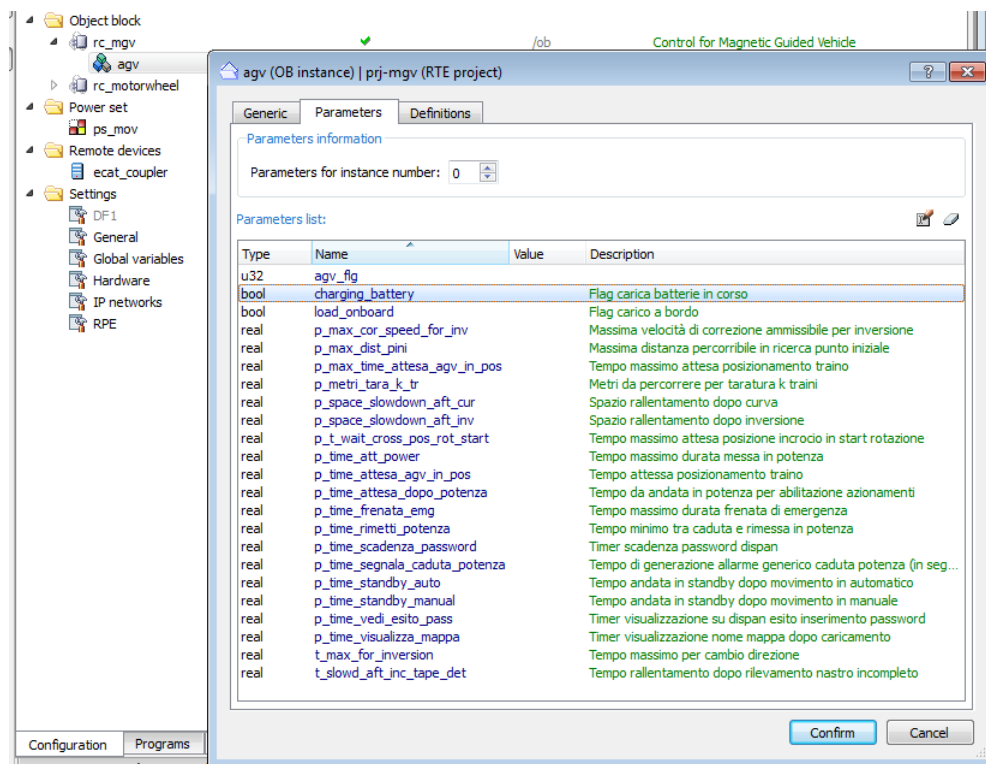


Figure 6.12: Object block instance parameters. In the column **Value** we can initialize the variables. To keep the program easy to read, it is better to initialize OB properties in R3. Note that properties declared as **ro** (read-only) are not shown here

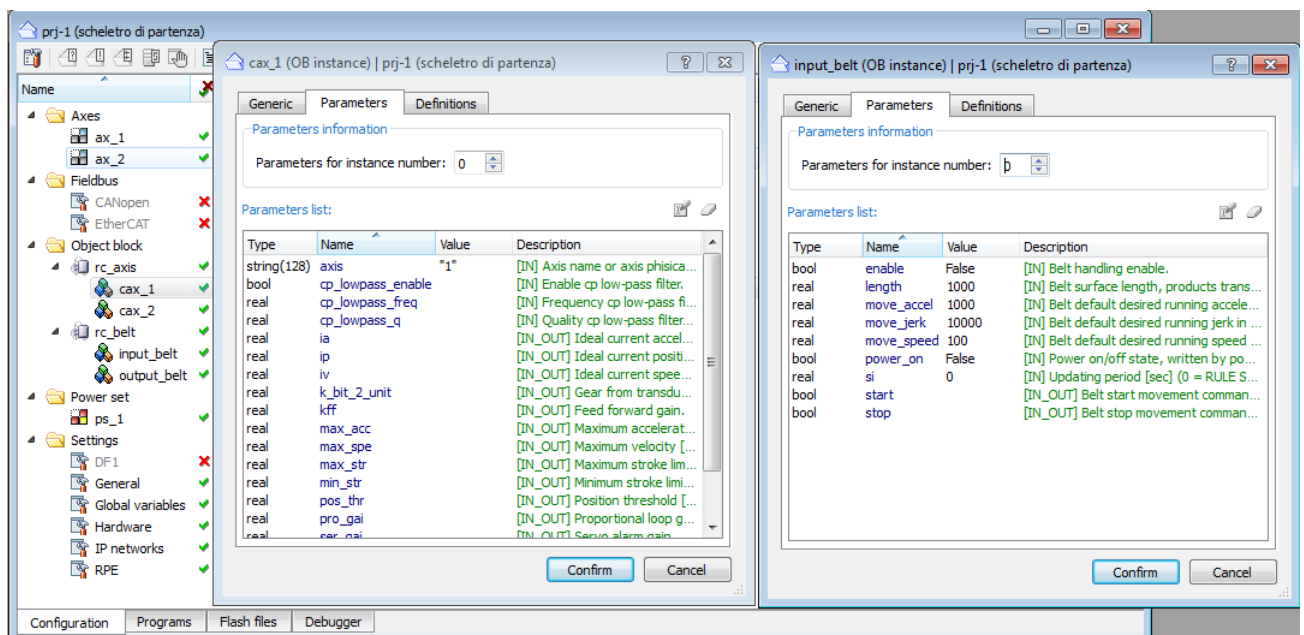


Figure 6.13: OB: Use and OB implementation predefined example.



7. Motion control

7.1 Basics

7.2 Example with R3

7.3 Example with OB



8. AGV commissioning

In this chapter we will see how to setup the parameters of the AGVs in RDE environment. Before doing it, we will have a look on mechanical and electrical part of the controller.

8.1 MGCV mechanical and electrical part

In this section we will describe one kind of AGV, that is MGCV a Magnetic guided vehicle. Some of the part of MGCV are used in other kind of AGVs. The difference is in the way of navigation and localization. The MGCV have to follow a magnetic tap, using magnetic sensors. In this example the MGCV use 4 magnetic sensors. Each magnetic sensor have 16 bits of output. The sensor is connected to a digital to canopen module in order to communicate with the controller.

In the middle of the MGCV an RFID reader is placed. RFID cards are placed along the path on the magnetic tape, and are used as position feedback. The feedback is not continuous.

Four motors and 4 drives are present. The drives take a velocity reference from the controller via analog signal, and send a velocity feedback from an encoder. The encoder signal is read in the controller via a high speed counter. Some digital IO are exchanged between controller and drive like drive reset, drive enable, drive ready or not in alarm, etc.

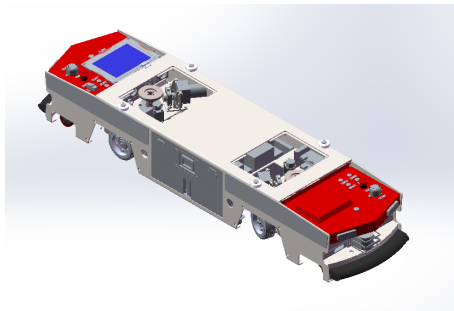
Steering is accomplished by modulating the speed of wheels. The angle of steering is relieved via a potentiometer, and read by the controller by an analog input module.

There are 2 laser scanner to monitor obstacles. Each laser scanner present 4 areas, that can be activated depending on the state of the AGV. Each area have 2 or 3 ranges of action.

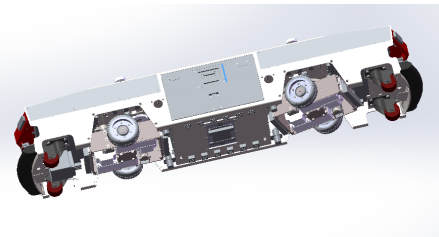
The first one, smallest one, is used as a safety stop of the AGV. The second range, middle, is used to slow down the AGV when the laser scanner detects some obstacle, and the third one is used as warning. The laser scanner has 4 digital inputs and 4 digital outputs.

The AGV is powered by batteries.

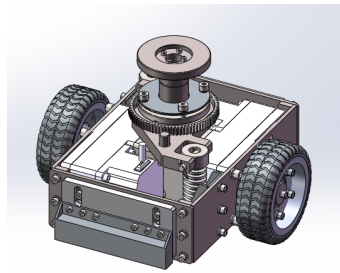
The communication between the AGV and AgvManager is done via an Ethernet to wireless device. The device is configured as access point. And the Ethernet to wireless device in AgvManager is configured as client.



(a) Top view



(b) Bottom view



(c) One set of wheels

Figure 8.1: Map line

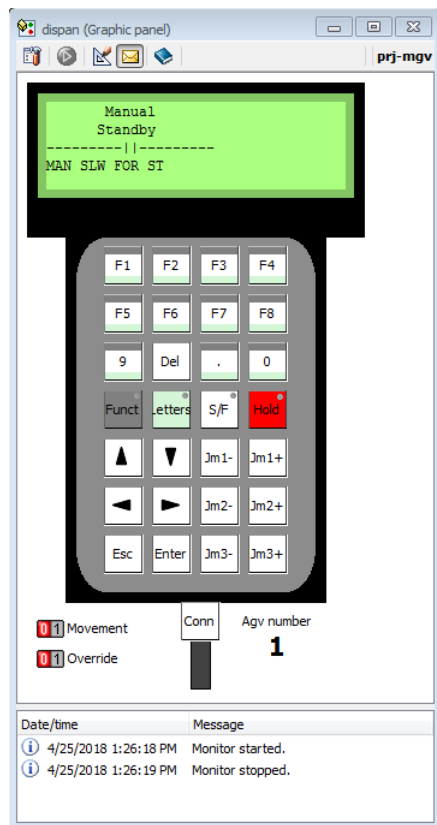
8.2 Dispan : Display panel

8.3 Commissioning

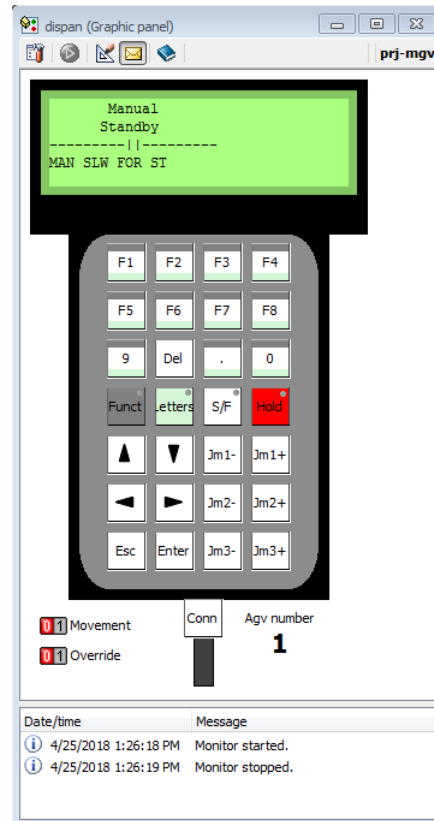
The first step in commissioning of any controller (plc or motion control) is to check the correct wiring of digital input and outputs. This step is relatively simple, using a variable monitor, online monitor or HMI we can debug the correct wiring of the signals following the electrical schematic.

Fig.8.3 show a template panel of IO signals for debugging purpose.

Once we are sure the electrical wiring is correct, we can proceed with tuning mechanical and electrical components and software parameters.



(a) Dispan in RDE Graphic panel



(b) Physical Dispan, communicate with the controller via RS485

8.3.1 Potentiometer

This AGV have 2 potentiometers, every one is connects to a set of two wheels. When a the 2 wheels connected to the same mechanical axes, rotate with different speeds, the set of wheels wil rotate around the center of the wheels coordinate system shown in fig.8.4a. The angle of rotation is measured via a potentiometer connected to the origin of the coordinate system via an elastic joint.

The potentiometer used in this AGV have a range between 0 and 270 degrees. The AGV can steer between -30° and 120° . We have to set the potentiometer in a way that the tension between pin 1 and 2 is around 3V when the wheels angle is at 0° .

After that we have to scale the analog input of the controller in order to read the correct angle. A linear relationship between the tension (or the raw value read by the analog input) an the angle in degree was supposed.

To adjust the parameters, we can use the dispan or a variable monitor. The procedure is to place the wheels at 0° then at 90° and read the corresponding value given by the analog input. Then assign the values to the variables `NVRR_POS_0_BIT_SLEWING_***` and `NVRR_POS_90_BIT_SLEWING_***`. Of course that same procedure have to be done for both potentiometers the front and the rear one fig.8.4a.

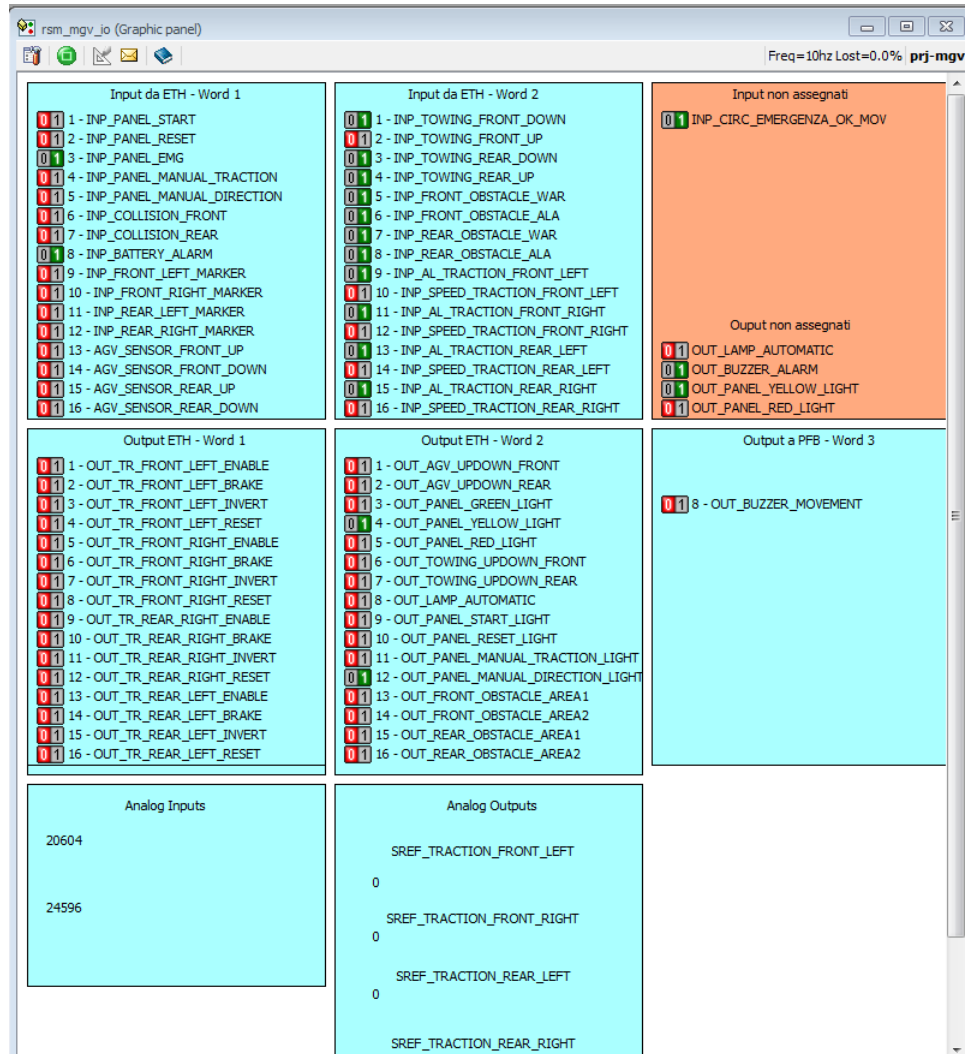
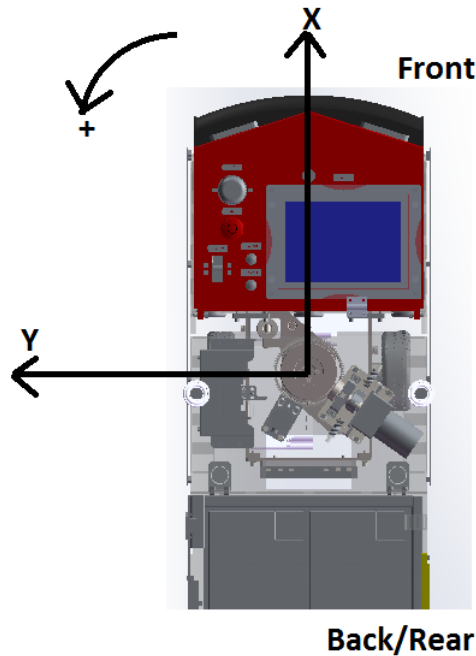
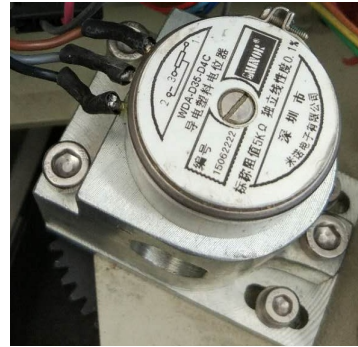


Figure 8.3: AGV IO signals. RDE graphic panel

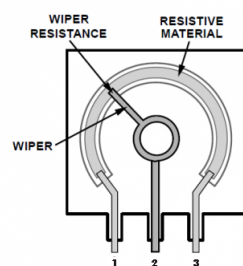
Using the dispan, we have to go to the voice F6 than choose 3 or 4 than save the parameters Figure.??.



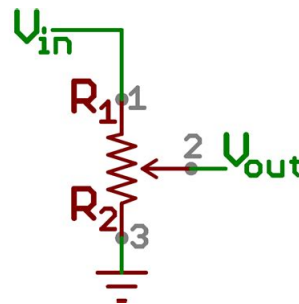
(a) AGV wheel coordinate system



(b) Potentiometer connected to wheel vis an elastic joint



(c) Potentiometer construction



(d) Potentiometer schematic

In listing.8.1 is shown the code R3 that represent the relationship between the analog input and the angle in degree.

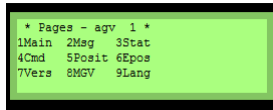
```

// Read actual position of slewrings (steers)
2 // NVRR_POS_0_BIT_SLEWING_FRONT = Position of front slewing axis at 0
  degrees in bits
// NVRR_POS_90_BIT_SLEWING_FRONT = Position of front slewing axis at
  90 degrees in bits
4 //
if (NVRR_POS_0_BIT_SLEWING_FRONT <> NVRR_POS_90_BIT_SLEWING_FRONT)
6   ScalaFront = 90.0 / (NVRR_POS_90_BIT_SLEWING_FRONT -
    NVRR_POS_0_BIT_SLEWING_FRONT)
  RR_CP_SLEWRING_FRONT = ScalaFront * (RAWCP_SLEWRING_FRONT -
    NVRR_POS_0_BIT_SLEWING_FRONT)

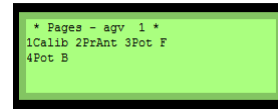
```

Name	Value	Type	Description
RR_CP_SLEWRING_FRONT	23.46	double	
RAWCP_SLEWRING_FRONT	20636	u16	
NVRR_POS_90_BIT_SLEWING_FRONT	8792	double	
NVRR_POS_0_BIT_SLEWING_FRONT	24812	double	
RR_CP_SLEWRING_REAR	1.62	double	REAR
RAWCP_SLEWRING_REAR	24596	u16	REAR
NVRR_POS_90_BIT_SLEWING_REAR	8408	double	REAR
NVRR_POS_0_BIT_SLEWING_REAR	24892	double	REAR

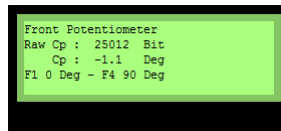
(a) RDE potentiometer variable monitor



(b) Dispan main menu, reached by F1



(c) Dispan 6Epos: Press 3 or 4 to tune the front or back potentiometer



(d) Front potenziometer tuning. Turn the Wheels to 0 degree then press F1. Turn the wheels to 90 degree than press F4.

Figure 8.5: Potentiometer tuning

```

8 else
    RR_CP_SLEWRING_FRONT = 999.9 // Error
10 endif

12 if ( not range ( CP ( AX_SLEWRING_FRONT ) , MIN_STR ( AX_SLEWRING_FRONT ) ,
    MAX_STR ( AX_SLEWRING_FRONT ) ) )
    alarm_set ( AL_WHEEL_RANGE , AX_SLEWRING_FRONT )
14 endif
    //
16 // NVRR_POS_0_BIT_SLEWING_REAR = Position of rear slewing axis at 0
    degrees in bits
    // NVRR_POS_90_BIT_SLEWING_REAR = Position of rear slewing axis at 90
    degrees in bits
18 ;
    if ( NVRR_POS_0_BIT_SLEWING_REAR <> NVRR_POS_90_BIT_SLEWING_REAR )
20 ScalaRear = 90.0 / ( NVRR_POS_90_BIT_SLEWING_REAR -
        NVRR_POS_0_BIT_SLEWING_REAR )
    RR_CP_SLEWRING_REAR = ScalaRear * ( RAWCP_SLEWRING_REAR -
        NVRR_POS_0_BIT_SLEWING_REAR )

```

```

22 else
    RR_CP_SLEWRING_REAR = 999.9    // Error
24 endif

26 if ( not range ( CP ( AX_SLEWRING_REAR ) , MIN_STR ( AX_SLEWRING_REAR ) , MAX_STR (
    AX_SLEWRING_REAR ) ) )
    alarm_set ( AL_WHEEL_RANGE , AX_SLEWRING_REAR )
28 endif

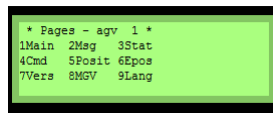
```

Listing 8.1: Scale potentiometer analog input

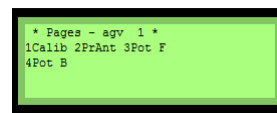
8.3.2 Magnetic sensors and Canopen modules

The 16 bits magnetic sensors are connected to canopen modules. Address have to be given to canopen modules, this can be done using the dispan and connecting every module to the **CAN2** port of the controller, and the address is given one by one to the modules.

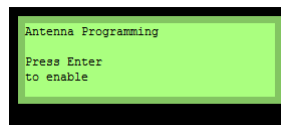
The figure fig.8.6 show the steps of the procedure.



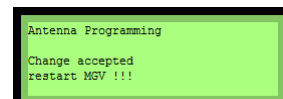
(a) Dispan main menu, F1



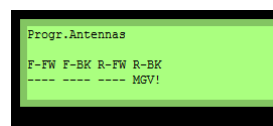
(b) Dispan F6 menu



(c) From F6 menu, press 2, to open the 2PrAnt page. Press enter to confirm



(d) Restart AGV



(e) After restarting AGV we can connect the canopen module to program them. When finish press MGV to terminate and restart the AGV.

Figure 8.6: Magnetic sensor canopen programming

8.3.3 Laser scanner

Every laser scanner have 4 inputs and 4 outputs. These digital signals are 4 bit code to identify the active area and the requested area. In our example we use only two input and two outputs.

Photo-coupler input(Anode common, Each input ON current 4mA)

Setting detecting area changeover

Set area No. by [Input 1], [Input 2], [Input 3] and [Input 4]

Stop emission by getting all [Input 1], [Input 2], [Input 3] and [Input 4] to ON

(OFF : H level input, ON : L level input)

[Input 1]	[Input 2]	[Input 3]	[Input 4]	Area patterns
ON	ON	ON	ON	Emission stop
OFF	ON	ON	ON	Area 1
ON	OFF	ON	ON	Area 2
OFF	OFF	ON	ON	Area 3
ON	ON	OFF	ON	Area 4
OFF	ON	OFF	ON	Area 5
ON	OFF	OFF	ON	Area 6
OFF	OFF	OFF	ON	Area 7
ON	ON	ON	OFF	Area 8
OFF	ON	ON	OFF	Area 9
ON	OFF	ON	OFF	Area 10
OFF	OFF	ON	OFF	Area 11
ON	ON	OFF	OFF	Area 12
OFF	ON	OFF	OFF	Area 13
ON	OFF	OFF	OFF	Area 14
OFF	OFF	OFF	OFF	Area 15

Figure 8.7: PBS-03JN laser scanner area code

The model of laser scanner we use, have 16 areas. The binary code is shown in fig.??.

Four different areas will be defined depending on the state of the agv. For example if the agv is not moving the alarm range of the 2 laser scanner should be extremely small. If the AGV is going forward the range of the front laser scanner should be bigger than the rear laser scanner and viceversa. We can define also another range for curving depending on the position of fixed obstacles.

Areas are activated using the digital inputs of the laser scanner (controller outputs). Following the codification of the laser scanner model e.g. fig.??.

8.3.4 AGV parameters

In task 1, AGV parameters file is loaded, the file name is written in register nsrv(1) NVSR_PLANT_CONFIG_FILE, by calling the function LoadAgvConfig(). We can create a parameter file called "params-rsm.stp" and assign the register nsrv(1) with file name.

In the parameter file we can find, the name of the map to be used, mechanical parameters listing.8.2, etc.

;

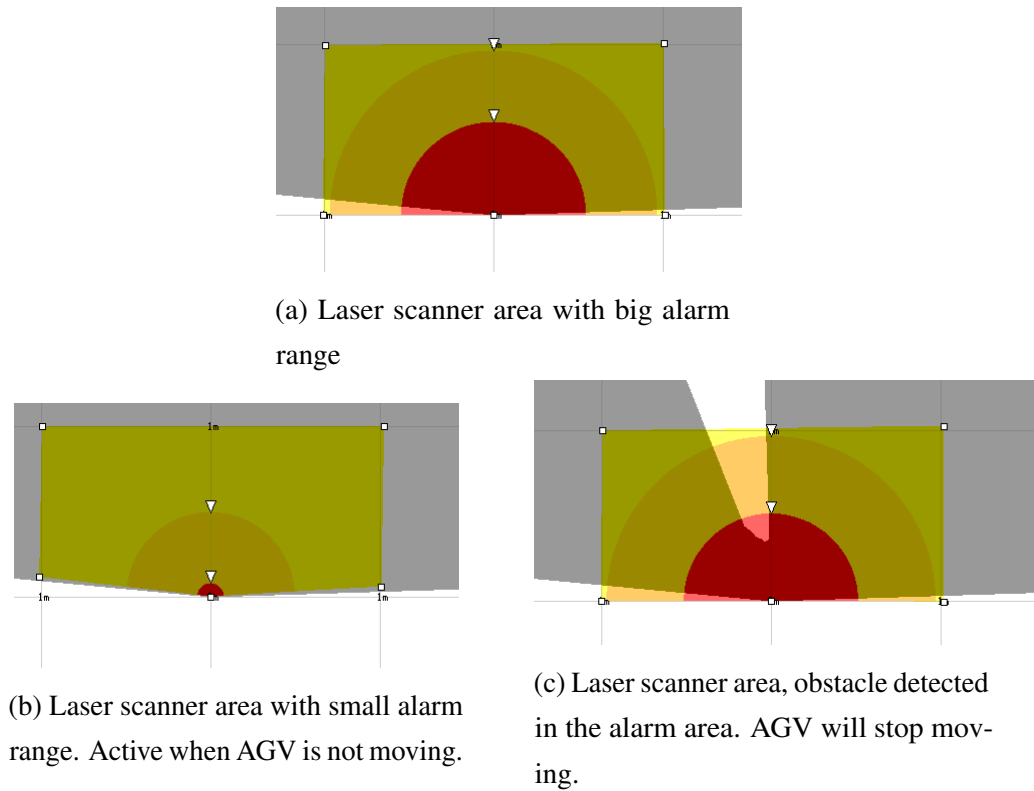


Figure 8.8: Laser scanner areas

```

2 ; Registers for mechanical configuration
;
4 rr(100) 0.4185 ; RR_POS_SLEWRING_STEER_FRONT_X [m] X position of
the center of the front slewing
rr(101) 0.0 ; RR_POS_SLEWRING_STEER_FRONT_Y [m] Y position of
the center of the front slewing
6 rr(102) 0.1525 ; RR_RADIUS_SLEWRING_FRONT [m] Distance from
the center of the wheel composing the electronic front steer
rr(103) -0.4185 ; RR_POS_SLEWRING_STEER_REAR_X [m] X position of
the center of the rear slewing
8 rr(104) 0.0 ; RR_POS_SLEWRING_STEER_REAR_Y [m] Y position of
the center of the rear slewing
rr(105) 0.1525 ; RR_RADIUS_SLEWRING_REAR [m] Distance from the
center of the wheel composing the electronic rear steer

```

Listing 8.2: params-rsm.stp Mechanical parameter of the AGV

8.3.5 Register backup

Non volatile registers are stored in memory, when RTE is loaded it retrieve the values from the memory. It is a good idea anyway to backup the values of non volatile registers to a file, and load them in case of loss of memory.

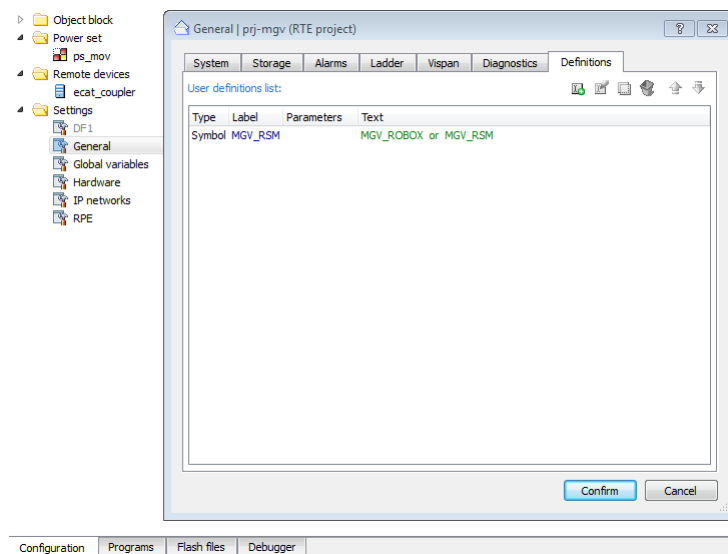
We can save the values of registers to a file by sending the command `uar /fb/lostreg.stp`. The file `lostreg.stp` is a file which contain desired register to be saved. For example if we want to save only the values of registers `nvrr(140)` and `nvrr(162)` we write in the file only these. When calling the command `uar`, only these registers are saved. There is a template file that can be used as a starting point.

If RTE lose memory, it load the file `/fa/lostreg.stp` when needed, the operator have to confirm it. In the file `/fa/lostreg.stp` there is a link to the file `/fb/lostreg.stp`.

For example if we have more than one agv, with different register values, we can call the command `uar` for each agv then copy the file `/fb/lostreg.stp` to our computer, e.g. `fsave -o /fb/lostreg.stp agv01/lostreg.stp`, this command copy the file from `/fb` to the folder `agv01` in the project directory.

8.3.6 General definitions

Variables defined globally, at project level fig.8.9a.



(a) Variable definitions, project level

```
; defined into general project definitions
$ifdef MGV_ROBOX
| $include "agv_io_robox\agv-io.i3"
$else
| $include "agv_io_cina\agv-io.i3"
$endif
```

(b) If the variable `MGV_ROBOX` is defined globally, a file is included otherwise another one is included.

Figure 8.9: Global variable definitions

8.4 AGV ecosystem

8.4.1 Software and tools

The software needed to program and setup an AGV are:

1. Map generator : RAT can be used
2. AgvManager : Script and path planning
3. Compilers: RCE and ICmap compilers
4. RDE
5. Database : optional

The electronic hardware needed:

1. RP1 : Robox controller to control AGV
2. personal computer: Where AgvManager will be executed
3. Plc or RP1: As an interface with signals from the plant
4. Wireless switch to communicate between RP1 and AgvManager

8.5 AgvManager-RP1 communication protocol

8.6 Hardware configuration

8.7 Program description

The Agv program consist of 2 RTE projects: AGV program and antenna program. The antenna program is used only to program the address of canopen devices. The AGV program is used to control the agv and communicate with AgvManager.

The agv program is written in R3 and in Object blocks. The R3 program consist of 7 tasks and one Rule. And the OB program consist of 2 object blocks: one for wheel control, with 2 instances, and one for AGV with one instance. The ob for Agv is changed depending on the type of the agv. In this example we will use the mgv object block for magnetic guided vehicles.

Some logic is written in R3 other logic is written in OB. Important non volatile parameters are saved in the file **lostreg.stp**, this file is used as backup in case the memory of the agv is lost.

We already see how to set some parameters and value in RDE or in the dispan. AGV operations changes from plant to plant. The main logic, a part from improvements, still the same. Plant specific logic changes. These kinds of operations, that the AGV get from AgvManager have to be implemented in R3.

For example, if we need to implement the logic of the LOAD command operation from AgvManager we have to do it in **rules→ function eseguiOperazioni()**. The constant

O_LOAD have to be defined in the file [operations.i3](#). The value of the constant should be equal to the value defined in AgvManager, that is 2.



Appendices

9	External editors	123
9.1	Vim	
9.2	Notepad++	
9.3	UltraEdit	



9. 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.

9.1 Vim

File needed and where to place them

9.1.1 Syntax highlight

9.1.2 Function list

9.2 Notepad++

File needed and where to place them

9.2.1 Regular Expressions

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

9.2.2 Syntax highlight

9.2.3 Function list

9.3 UltraEdit

File needed and where to place them

9.3.1 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++.

9.3.2 Function list and syntax highlight

