



Robox notes

Work in progress 2018-03-30 rev 0.3

Robox motion control



Copyright © 2018 Abed Ramadan

ABED@ROBOX.COM.CN

ROBOX.IT

www.gnu.org/licenses/gpl.html

Contents

1	Introduction	5
I	AGV Manager	
2	RAT: Robox Agv Tool	11
2.1	RAT	11
2.2	Map	11
2.2.1	Vehicle	11
2.2.2	Lines	12
2.2.3	Generic point	12
2.2.4	User point	14
2.2.5	Battery point	15
2.2.6	Magnet point	15
2.2.7	Start point	15
2.2.8	Cross	15
2.3	Tips	16
3	AGV Manager	21
3.1	Overview	21
3.2	Installation and creation of new project	21
3.3	AGV configurator	21
3.4	AGV script executing	21
3.4.1	Fundamental concepts	25
3.4.2	Main loop execution	25
3.4.3	Mission execution	26

3.4.4	Drag and drop example	26
3.5	More about MICRO	34
4	More complete example	35

II

Motion control

III

Bibliography



1. Introduction

TODO



AGV Manager

2	RAT: Robox Agv Tool	11
2.1	RAT	
2.2	Map	
2.3	Tips	
3	AGV Manager	21
3.1	Overview	
3.2	Installation and creation of new project	
3.3	AGV configurator	
3.4	AGV script executing	
3.5	More about MICRO	
4	More complete example	35

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.

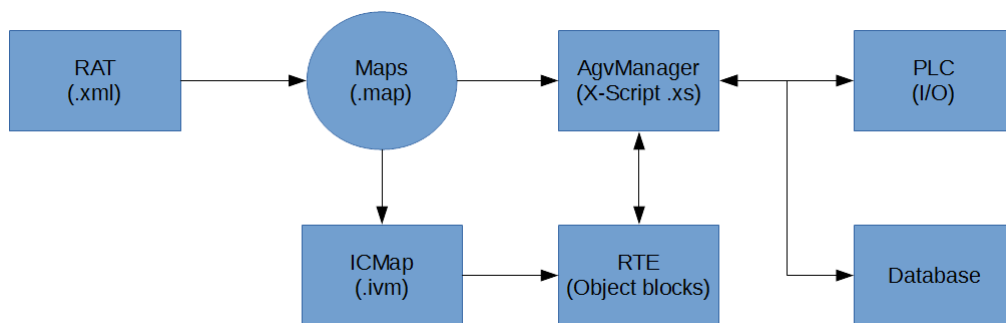


Figure 1.1: AGV block diagram. RAT give a .map file as output. The map is read by AGV manager. The map is comiled by ICMaP then read by RTE. AGV manager may communicate with a PLC or a database as an interface to the plant IO, and with RTE for motion control.

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 to 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 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.

2.2.1 Vehicle

We can define the number of vehicles present in a plant, their shapes and dimensions. In our dicussion 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. $\vec{x} = \overrightarrow{OF}$ and $\vec{y} = \overrightarrow{OL}$.

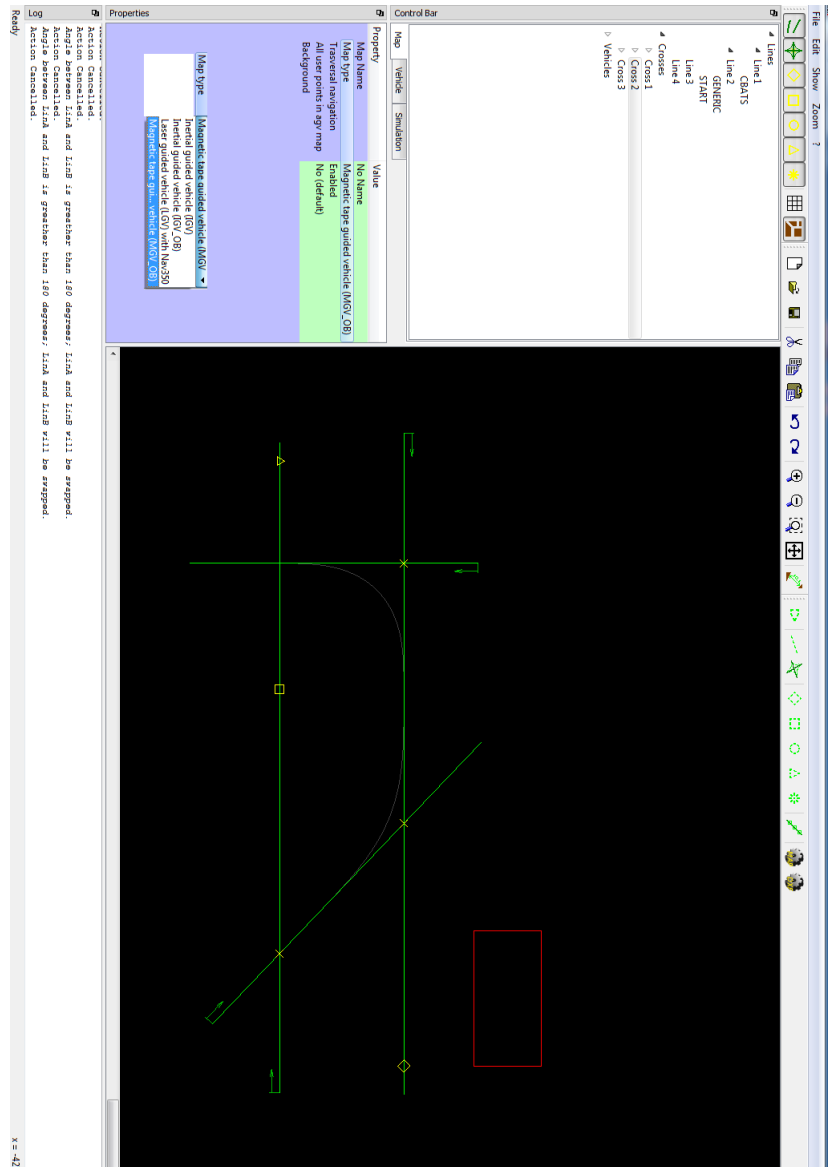


Figure 2.1: RAT main window

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} . Two directions are allowed: Forward and backward. Forward direction is shown by the arrow on the line, that is the positive movement, i.e. vector orientation. 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 this 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 the direction. So the line is defined as $\vec{P_1P_2}$. The origin can me moved changing the parameter origin, when it is different from zero we can see the arrow on the line move.

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, beside the cross. Those points share the

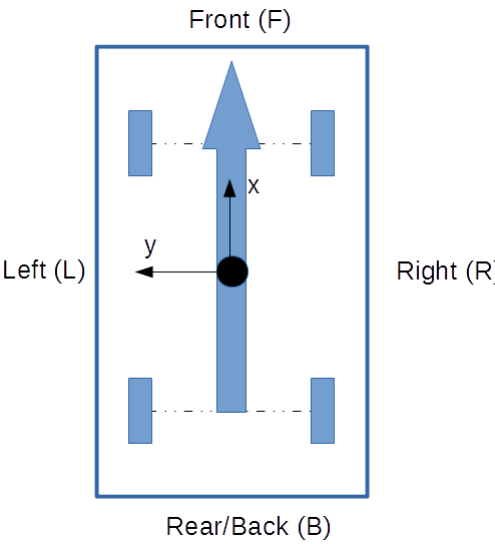
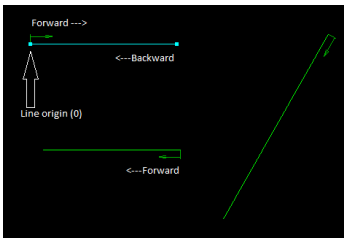


Figure 2.2: Vehicle orientation



(a) Line or vector. Direction of motion

Property	Value
Name	LineName
Index	2
P1	-5.065, 1.871
P2	-3.878, 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

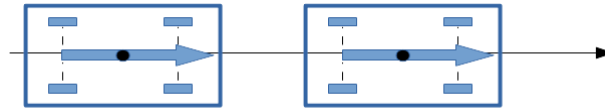


Figure 2.4: Longitudinal navigation. BF parallel to the line.

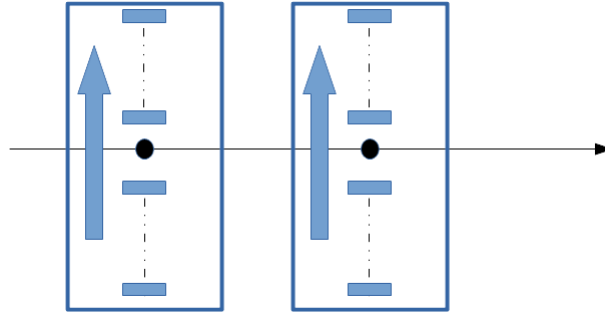


Figure 2.5: Side or traversal navigation. BF perpendicular to the line.

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). Generic 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 excluded the cross.

Three allowed directions can be assigned to 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. If we set the allowed direction to Forward, and we want to move from P_1 to point P_2 , the motion direction will be in positive direction (Forward). But if we want to go from P_2 to point P_1 , the vehicle will move in any direction, maybe taking the shortest way fig.2.8.

The allowed orientation is referred to the vehicle fig.2.2. A point have 7 allowed orientations. For example if the Font orientation is selected, the vehicle when is moving on the line, \overrightarrow{BF} have the same orientation of the line \overrightarrow{L}

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 rectangular area, with width define by the parameter *semaphore width*, and length defined by the position of the start and stop points.

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.

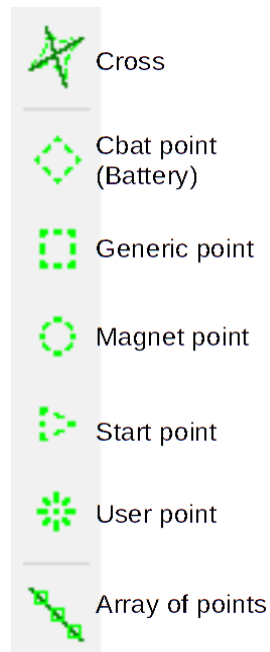


Figure 2.6: Kind of points

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.2.9b.

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.

A mangnet 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 2.10.

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 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.

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. Thre kind of permission are available: Forbidden, curve and

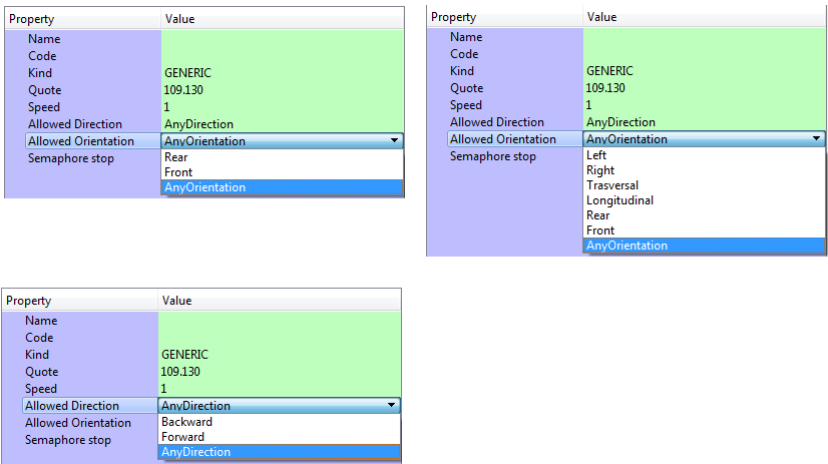


Figure 2.7: Generic point property

rotation fig.2.11.

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.2.10.
- 3. User points and generic points should be placed after the magnet points that form the curve fig.2.12.

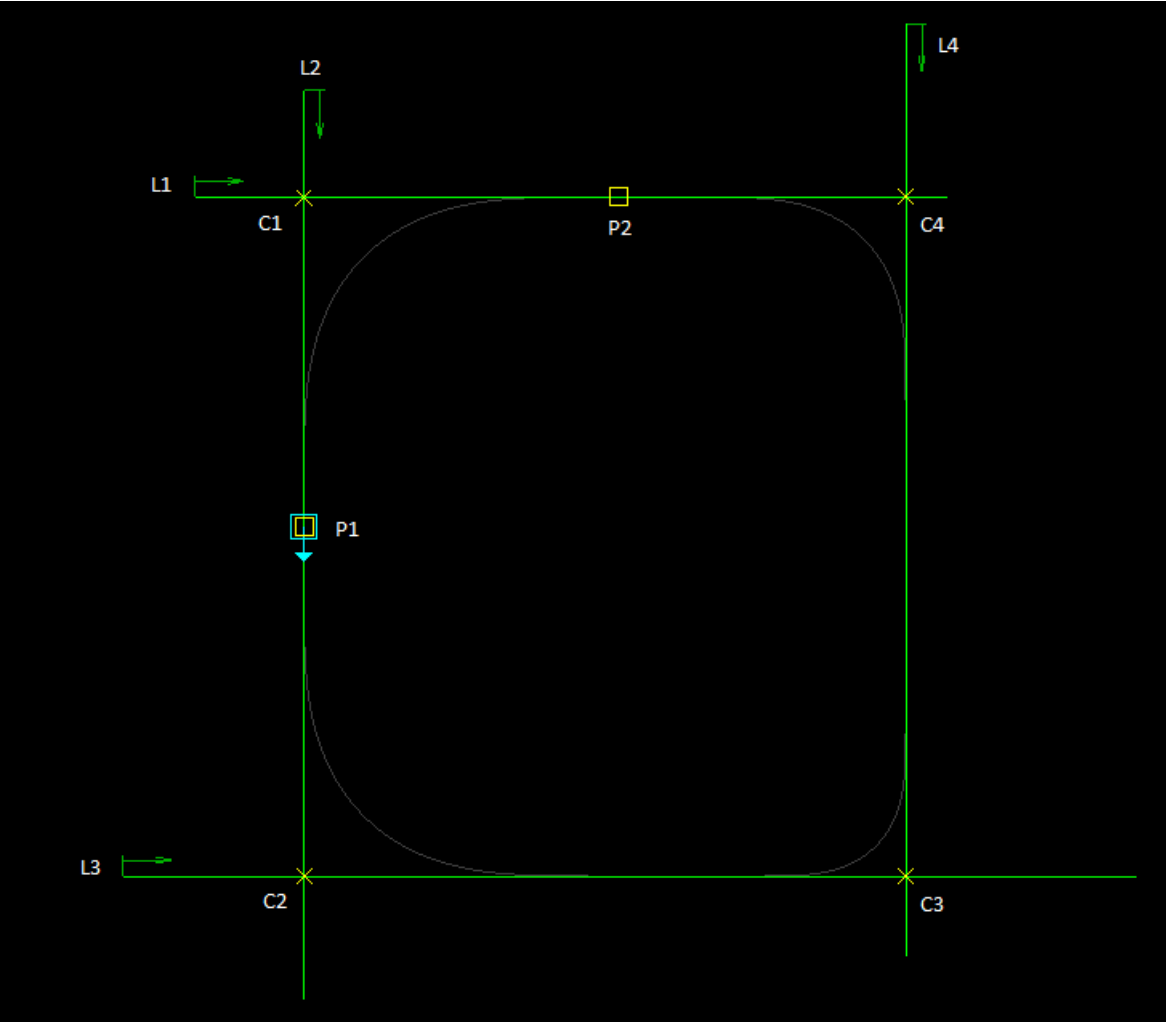


Figure 2.8: Point allowed direction. P_1 allowed direction is set to Forward. A vehicle moving from P_1 to P_2 will cross C_2 , C_3 , C_4 . Instead a motion from P_2 to P_1 will cross only C_1

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 2.9: Map points

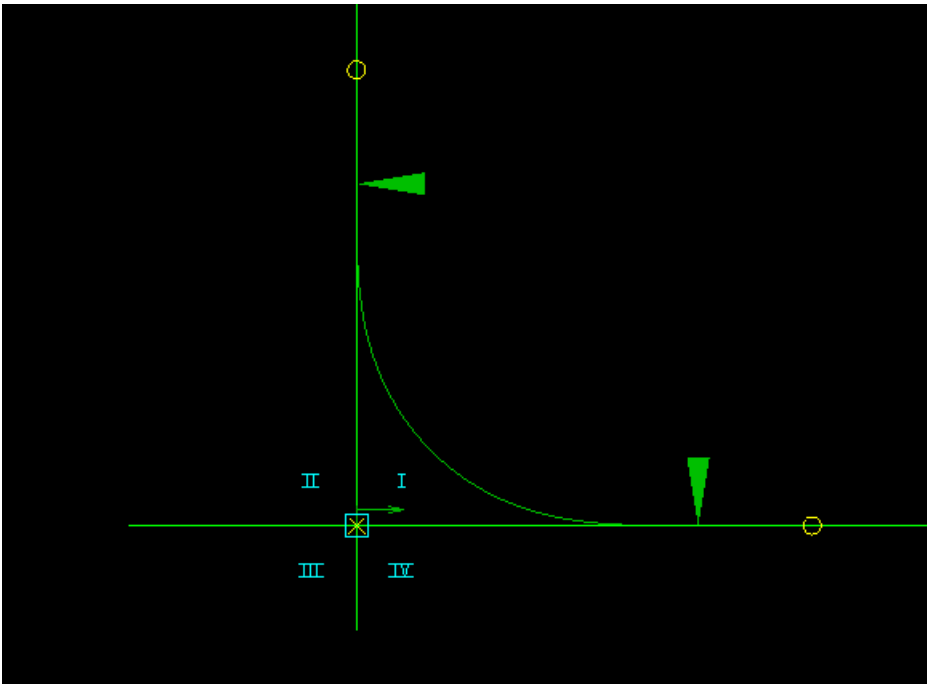


Figure 2.10: Two manget points should be place at least 0.5 meter from the end of a curve.

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 2.11: Two manget points should be place at least 0.5 meter from the end of a curve.



3. AGV Manager

3.1 Overview

AGV Manager have 2 software components: AGV manger 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 and creation of new project

The installation of AgvManger is straightforward, like any program in Microsoft windows. Agv-Configurator is installed automatically with AgvManager.

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

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.3.2, select the map fig.3.3 and agv 3.4 and plc communication.

3.4 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.

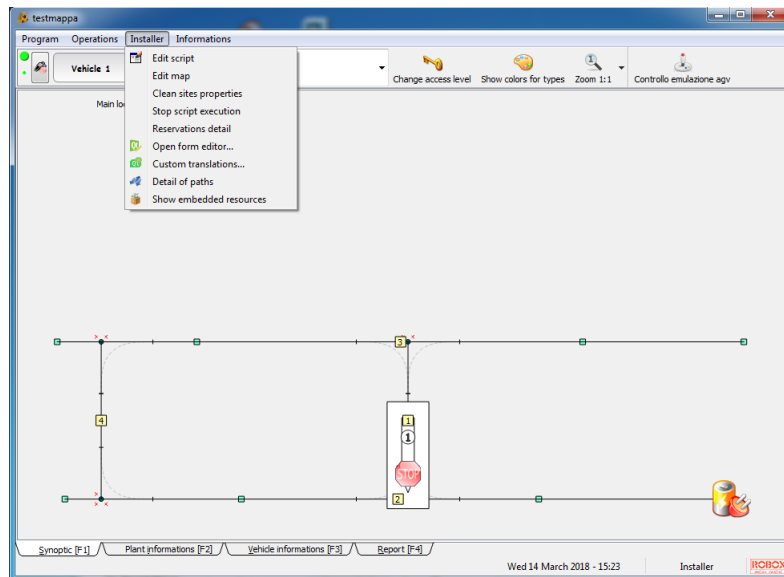


Figure 3.1: AGV Manager main window

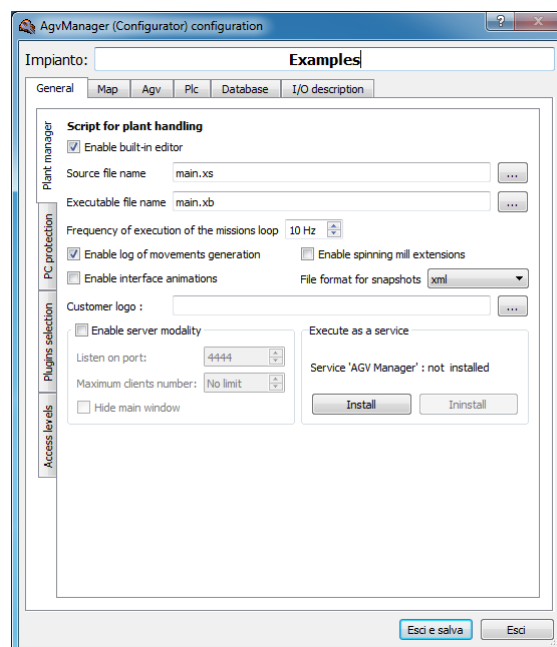


Figure 3.2: AGV configurator. General tab, where the .xs script file is selected. The script have to be created by an external editor. It is enough to write the name of the executable file with .xb extension, when AgvManager compile the xs file, the xb file will be created automatically.

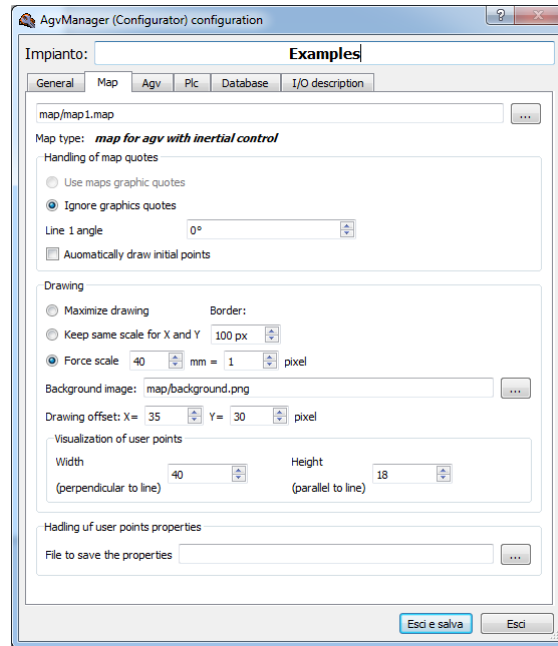


Figure 3.3: AGV configurator. Map tab, where the .map file is selected. In order to view the user point, you have to set the width and height of it, otherwise user points are not viewed.

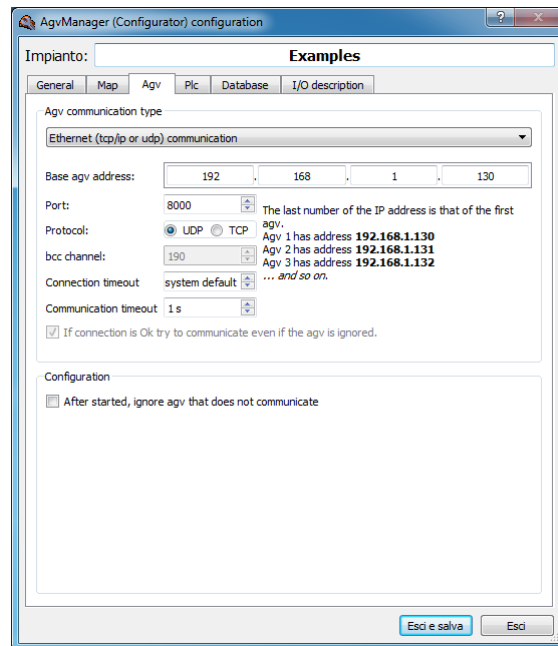


Figure 3.4: AGV configurator. AGV tab, where communication parameter with the AGV are set.

Xscript have some OOP properties (creating classes and objects), some event handling (mouse move event) and callback functions.

3.4.1 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 automatically by AGV manager.

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 there with the prefix *MOD_*, i.e. *MOD_AUTOMATICO*.

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

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_P* have value 10, the compiler will not give errors and the agv will behave as is not expected.

AGV manager 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*.

More about MICRO later.

3.4.2 Main loop execution

The following is a simplified explanation of the main loop of the script, which is in execution behind the scene. A more complex scenario is shown in fig.3.5. When the script is executed the first time, the function *OnApplicationStart()* is called. In this function one 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.

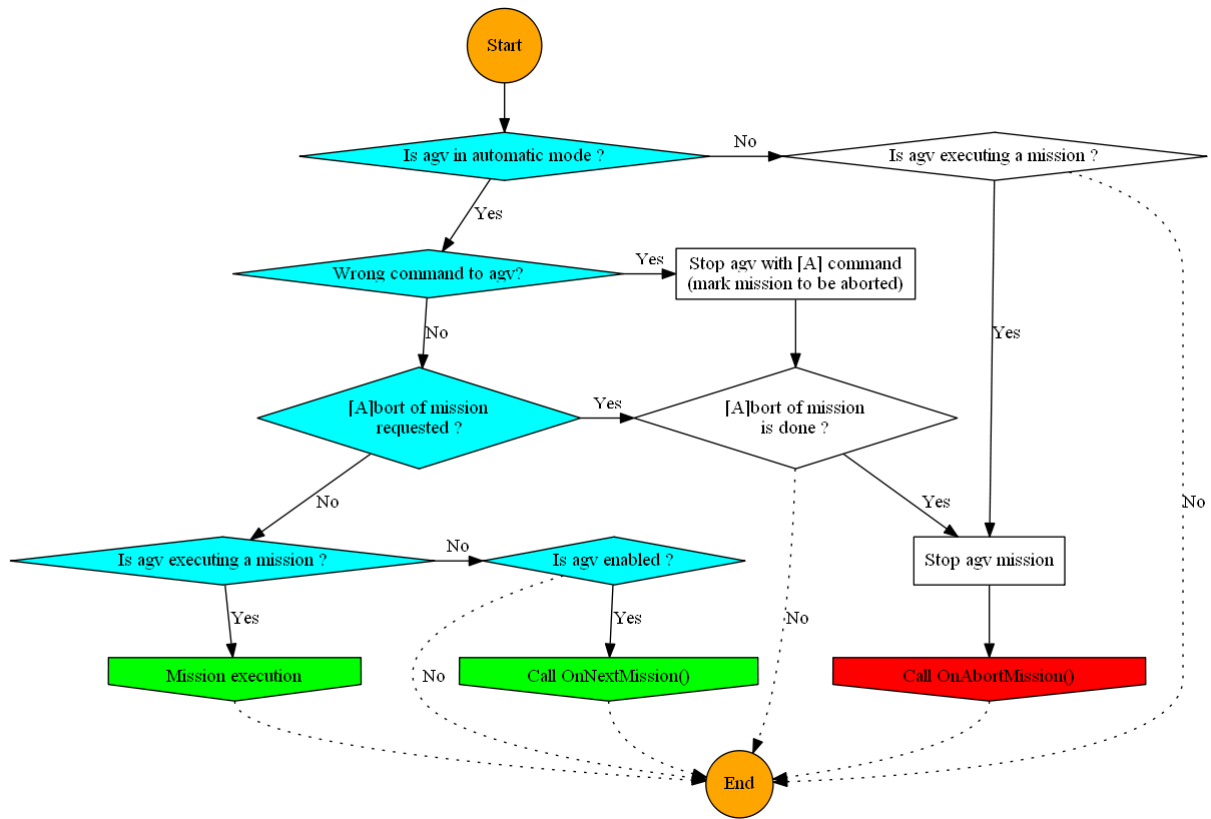


Figure 3.5: Main loop execution

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

3.4.3 Mission execution

A mission is a set of MACROS. There is a list of MACROS, where the order of execution is assigned. When a mission is started by the call of onNextMission(), 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 until the end of the MICRO execution.

Fig.3.6 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 every function like a plc, cyclically.

3.4.4 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.3.7, 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.

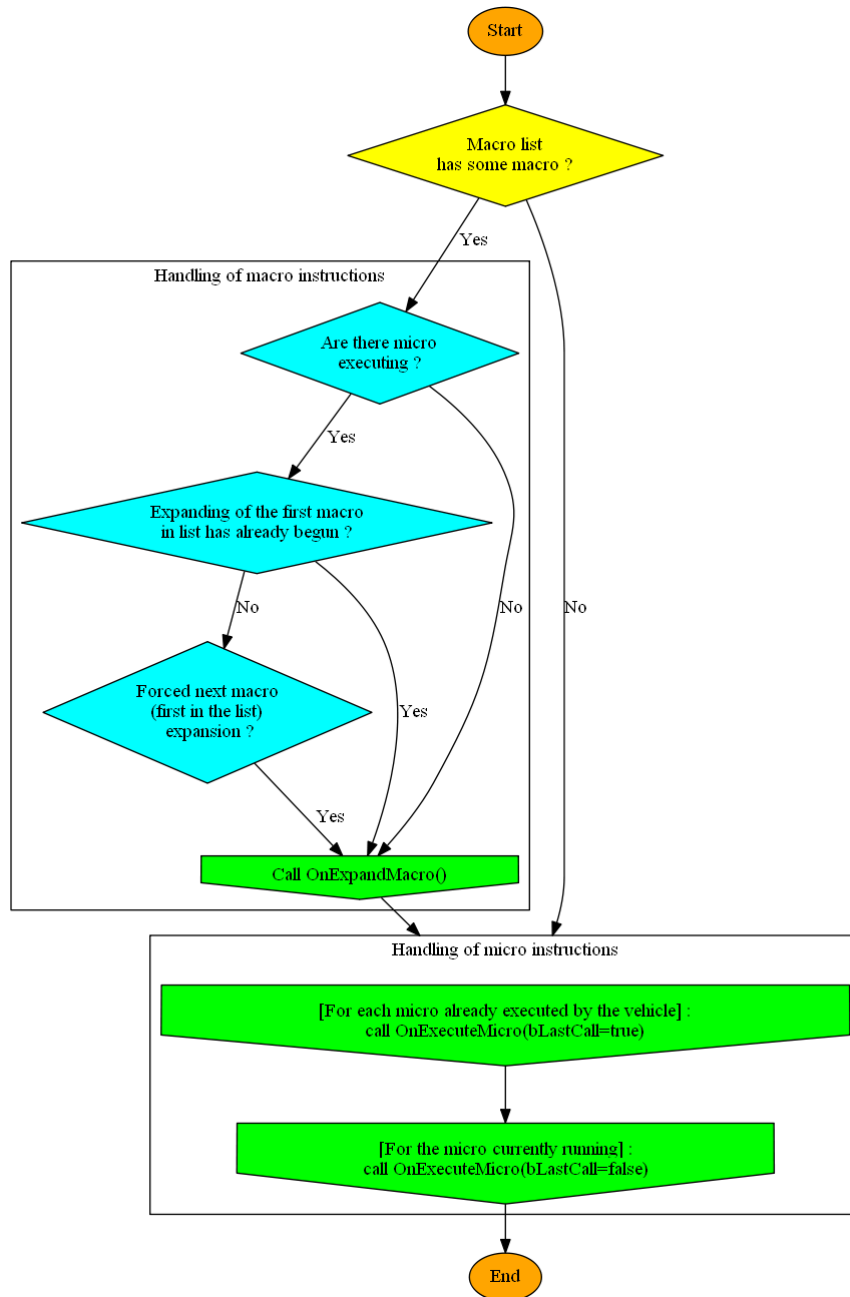


Figure 3.6: Main loop execution



Figure 3.7: AGV configurator. AGV tab, where communication parameter with the AGV are set.

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 userId, int iMission, 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 function 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 *onagv-droptedtoline()*. 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 set some constraint e.g. the agv should be in automatic mode, it should not be enabled, there is no mission in progress. If those condition are meeted the agv can move from one pint 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. Ther are some predifenied 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

```

```

5  $define MIS_TO_POINT          14
7  ; MACRO Movement to waypoint
   $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 returns the id of the mission in progress. If it returns zero, it means there is no mission in progress. We have already defined a constant MIS_NULL as zero. In the code we can use 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 sMissioneDescription)*, this function returns true if the mission is in progress.

Now we have to fill the agv MACRO list with our 2 MACROs, by calling the function *bool agvAddMacro(uint uAgv, uint uCodeMACRO, int ipar1=0, int ipar2=0, int ipar3=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 takes a parameter the agv id, the point id and direction and returns an id of the point.

Then the MAC_MOVE_TO_WP can be added 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 ends our mission is added to the 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.")
11     return
12 end
13 if (AgvAbilitato(uAgv))
14     MessageBox("AGV " + (uAgv + 1) + " is enabled." + chr(10) + "Please disable
15         it to give commands.")
16     return
17 end
18 if (AgvActualMissionCode(uAgv) != MIS_NULL)
19     MessageBox("AGV " + (uAgv + 1) + " is already executing a mission")
20     return
21 end
22 ;
23 AgvStartMission(uAgv, MIS_TO_POINT, "mission to point")
24 ;
25 uint wpidx
26 uchar destOrientation = 'X'
27 bool concatenateNext = true
28 ;
29 wpidx = AgvAddWaypoint(uAgv, uUser, destOrientation)
30 AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
31 ;
32 AgvAddMacro(uAgv, MAC_END, MIS_TO_POINT)
33 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 mention 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, mision id, MARCO 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 statment or the if in order to select out logic. The first MACRO is MAC_MOVE_TO_WP. Under the case MAC_MOVE_TO_WP we implement the instructions to AgvManager:

```

1     case MAC_MOVE_TO_WP
2         ; iPar1 = Waypoint id
3         ; iPar2 = (bool) do concatenate next macro
4         select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi |
5             WpFl_EliminaCompletato))
6             case EsitoMov_MovimentoCompletato ; Completed movement
7             case EsitoMov_RaggiuntoWaypoint ; Waypoint reached
8                 if (iPar2)
9                     AgvComputeNextMacro(uAgv)
10                endif
11                return true
12            default
13                return false
14        endselect
15    return true

```

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

As we say every MACRO consist of different MICROs. A MACRO that correspond to a motion have a `MIC_MOVE`, the `MAC_END` register a `MIC_SYSTEM` micro type. We will speak more about MICROs in the following section.

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)`.

As shown in fig.3.6, `AgvManager` continue to call `onExpandMacro` and `onExecuteMicro`.

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

```

;
; ~~~~~
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, "")
    AgvRegisterSystemBloccante(uAgv, uMission, S_END)
38    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. A MICRO is registered by calling `agvRegisterSystemBloccante()` or `agvRegisterSystemPassante()`. These function have `uAgv`, `uMission`, `MICROcode` as input parameters. The difference is that `agvregistersystembloccante` lock the execution of other micros till the end of the execution of itself or till the verification of a condition.

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`. A `MIC_MOVE` type is related to instruction of motion sent to the agv.

In the function `OnExpandMacro()` we register a block system micro, `S_END`. In the case under `MIC_SYSTEM` and under the case `S_END` we call the function `AgvStopMission(uAgv)` in order to stop the mission. When the mission is stopped, the micro terminate, and eventually other micros can start. When a micro terminate the execution the function `OnExecuteMicro()` return true.

```

1 ;
    ~~~~~
; Execution of the actions related to vehicle operations ,
3 ; and execution of the SYSTEM micro .
;
    ~~~~~

5 code OnExecuteMicro(uint uAgv, bool bLastCall, int iMicroCode, int iPar0, int
    iPar1, int iPar2, int, int, int userId, int iMission, int) : bool
    XVehicleInfo vInfo
7    AgvGetVehicleInfo(uAgv, @vInfo)

9    select (iMicroCode)

11    case MIC_MOVE
12    case MIC_CURVE
13    case MIC_ROTATION
        return true

15
16    case MIC_OPERATION

17
18    case MIC_SYSTEM
        select (iPar0)
19        case S_NULL
21            ; Micro of that type are generated by AgvManager, I am not
            intereseted on it.
            break

23
24        case S_END
25            ; End of mission
            if (vInfo.uStatus & VST_EXEC_COMANDO)

```

```

27         MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": wait for agv
commands finished")
        return false
29     endif
        MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": finished executing
commands")
31         AgvStopMission(uAgv)
        SetAgvMessage(uAgv, "")
33         break
        default
35         qt_warning("Unknown MIC_SYSTEM : " + iPar0 + " (mission = " +
iMission + ", par1 = " + iPar1 + ")")
        break
37     end
        break
39
        case MIC_PASSANTE
41         select (iPar0)
            default
43             qt_warning("Unknown MIC_PASSANTE : " + iPar0 + " (mission = " +
iMission + ", par1 = " + iPar1 + ")")
            break
45         end
            break
47
        case MIC_WAIT
49         if (bLastCall)
            MessageState("Agv " + (uAgv + 1) + ": passthrough operation executed")
51             return true
            end
53             return false

55         default
            qt_warning("Unknown micro: " + iMicroCode + " (mission = " + iMission + "
, par0 = " + iPar0 + ", par1 = " + iPar1 + ")")
57             break

59     end
        return true
61 end

```

Listing 3.4: OnExecuteMicro

3.5 More about MICRO



4. More complete example

