Robox notes

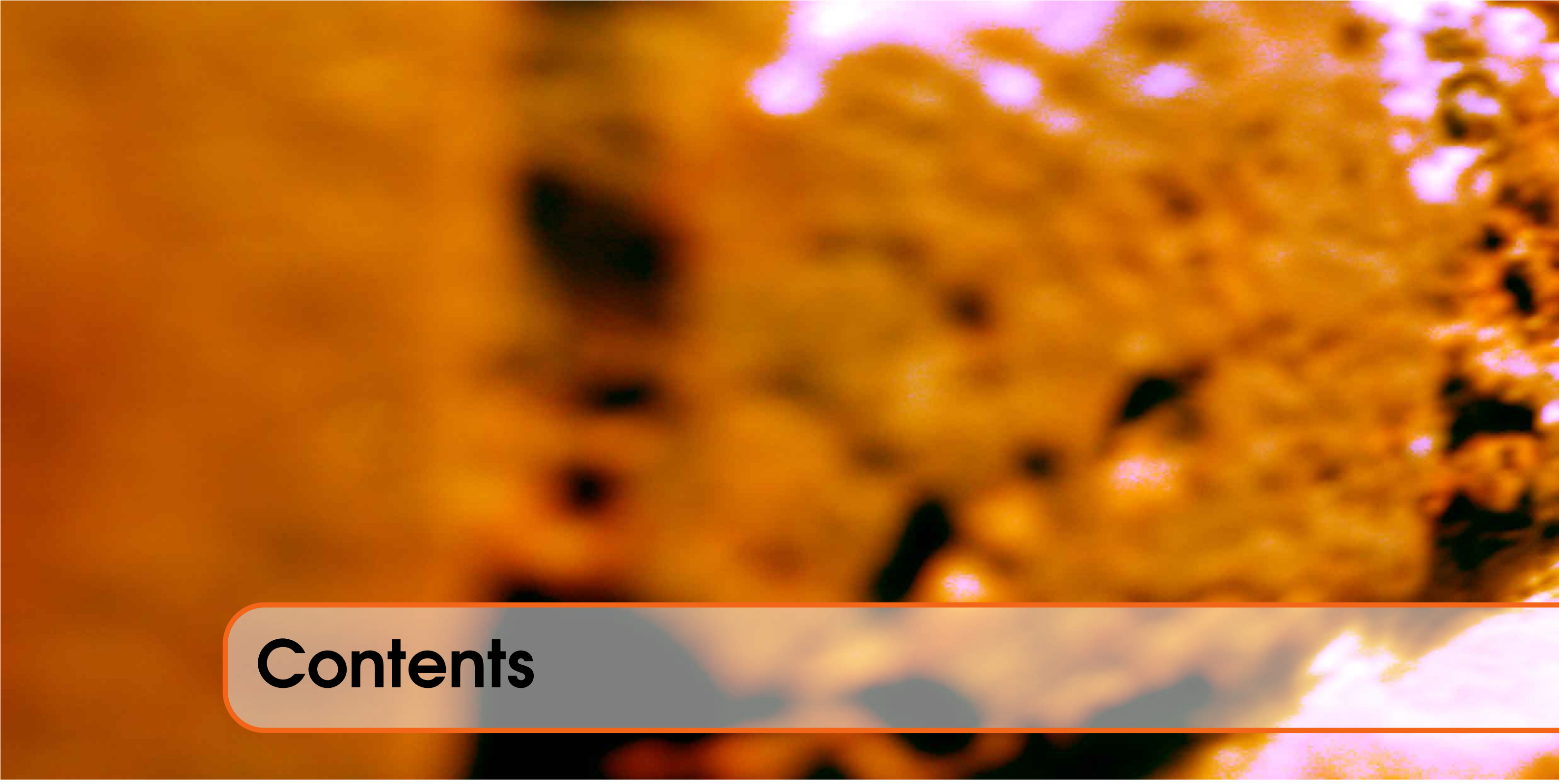
Work in progress 2018-04-12

Robox motion control

2018-04-12

ABED@ROBOX.COM.CN

ROBOX.IT



Contents

III Appendices

IV Bibliography

1. Introduction

TODO

AGV Manager

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.

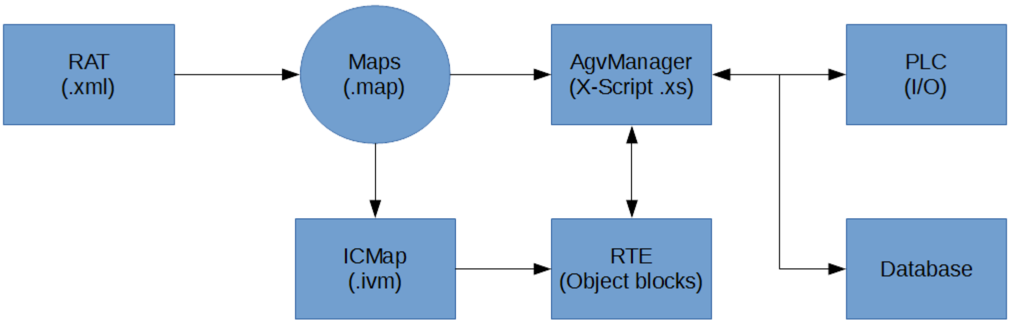


Figure 1.1: AGV block diagram. RAT give a .map file as output. The map is read by AGV manager. The map is comiled by ICMap then read by RTE. AGV manager may communicate with a PLC or a database as an interface to the plant IO, and with RTE for motion control.

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 togethers.

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.

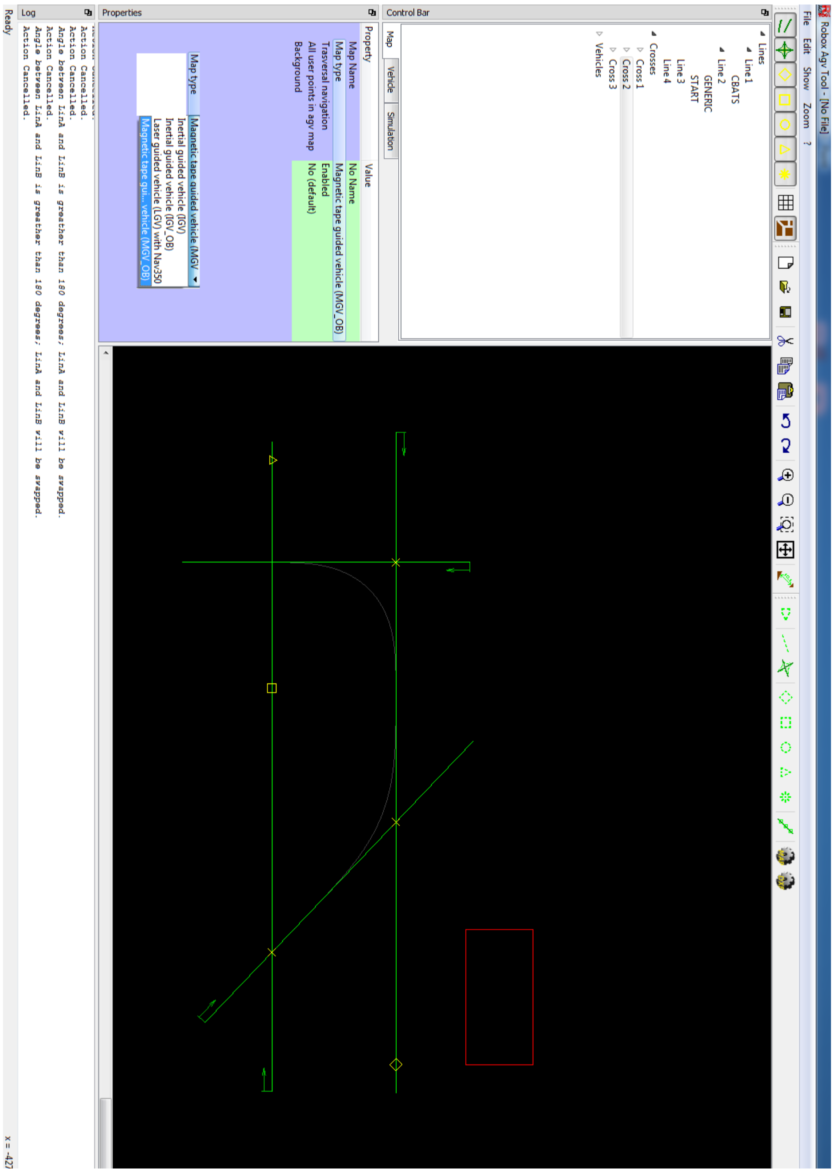


Figure 2.1: RAT main window

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 $(255,255,255)$.

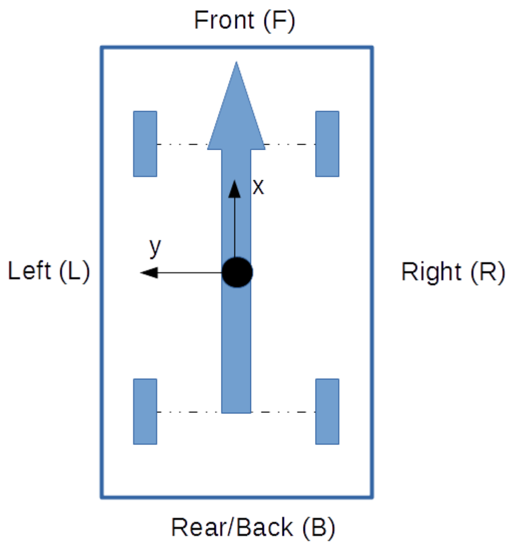
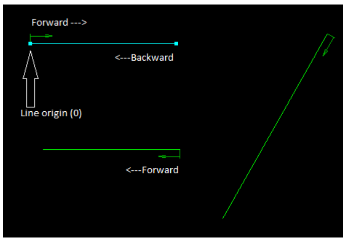
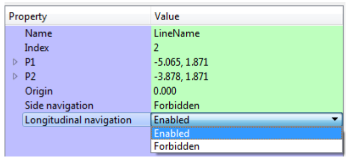


Figure 2.2: Vehicle orientation



(a) Line or vector. Direction of motion



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

Figure 2.3: Map line

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, $\overrightarrow{L}$. For example a vector $\overrightarrow{OX}$ has opposite direction of vector $\overrightarrow{XO}$, note that $\overrightarrow{OX} = -\overrightarrow{XO}$.

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 $\overrightarrow{OX}$.

The vehicle can move longitudinally fig.2.4 to the line, i.e. $\overrightarrow{BF}$ parallel to the line, or transversally (side navigation), i.e. $\overrightarrow{BF}$ 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 $\overrightarrow{P_1P_2}$. 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.

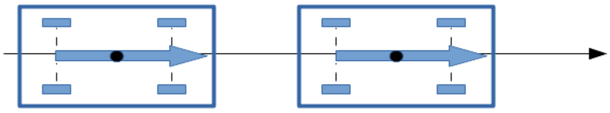


Figure 2.4: Longitudinal navigation. BF parallel to the line.

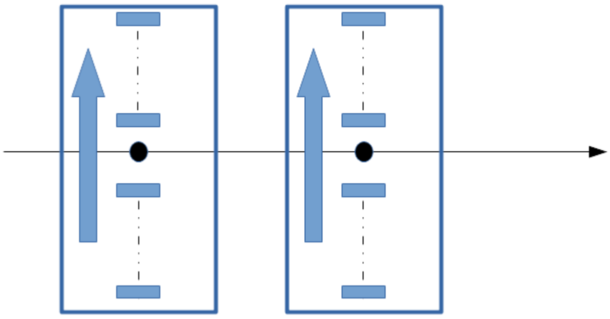


Figure 2.5: Side or traversal navigation. BF perpendicular to the line.

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, beside the cross. 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 (refered 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 excluded the cross.

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. 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 Front orientation is selected, the vehicle when is moving on the line, $\overrightarrow{BF}$ have the same orientation of the line $\overrightarrow{L}$

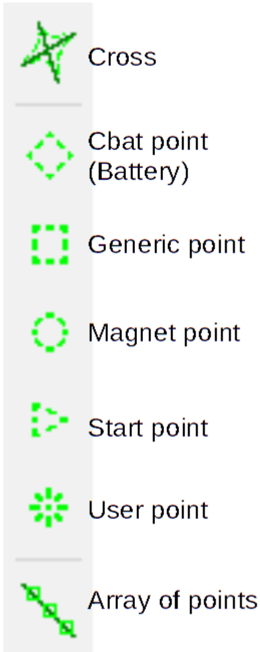


Figure 2.6: Kind of points

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 rectangluar 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 could be empty.

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.2.9b.

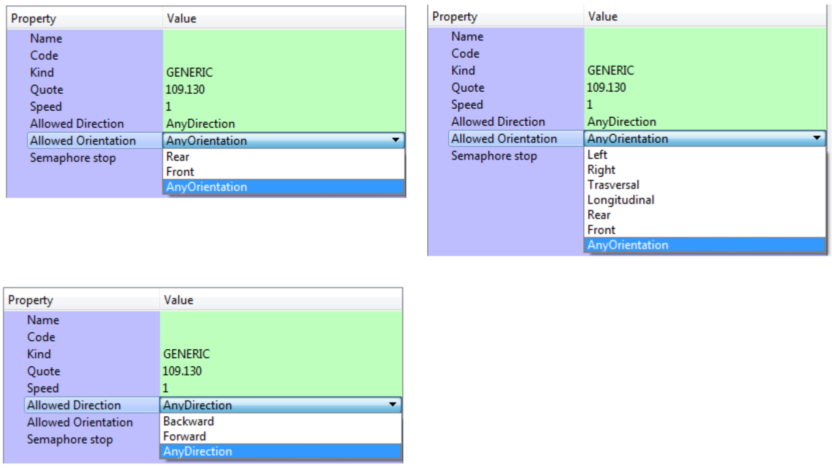


Figure 2.7: Generic point property

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.

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 2.10.

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.

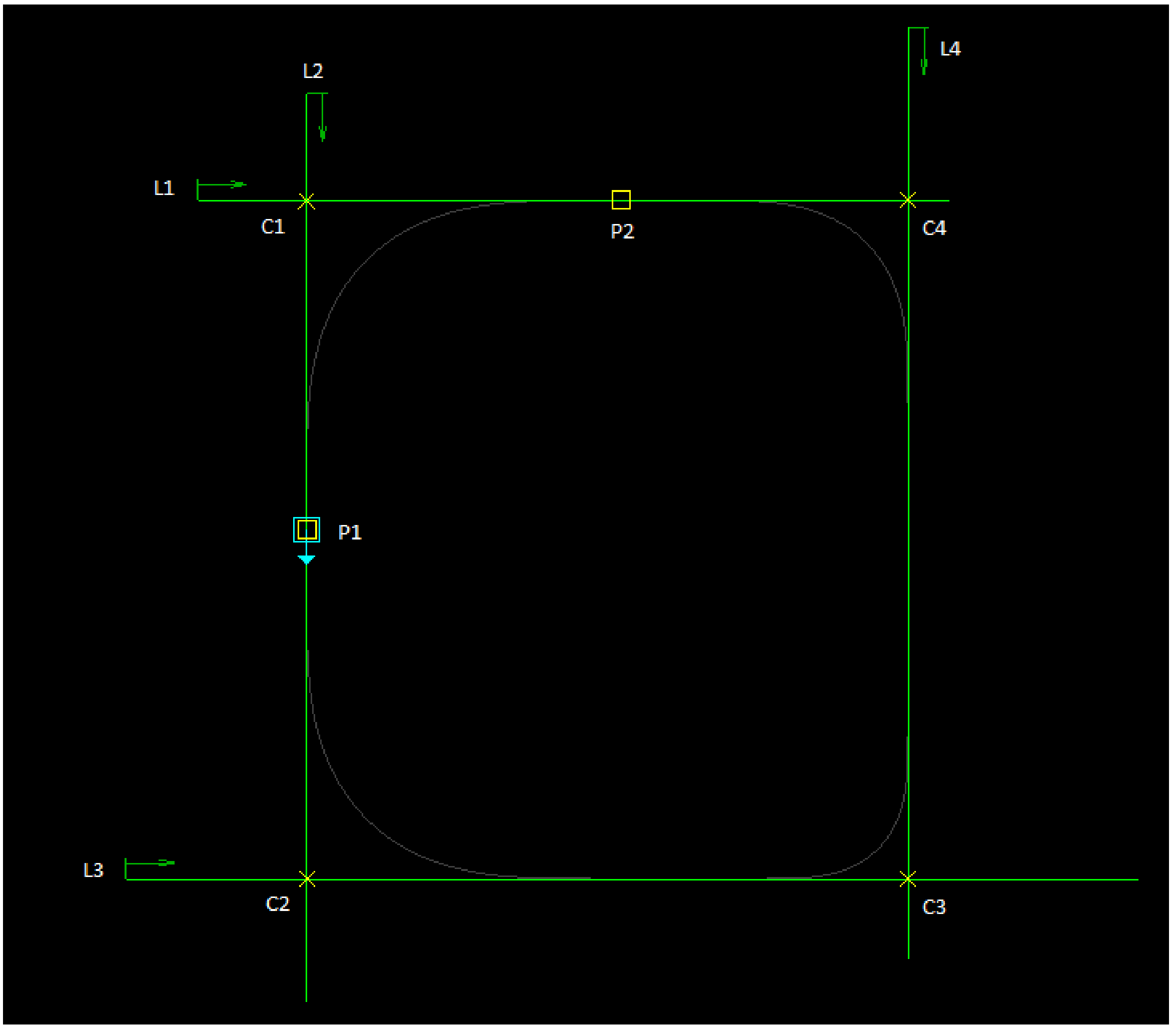


Figure 2.8: Point allowed direction. P_1 allowed direction is set to Forward. A vehicle moving from P_1 to P_2 will cross C_2, C_3, C_4. Instead a motion from P_2 to P_1 will cross only C_1

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.

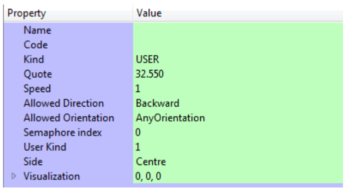
Starting orientation ??????????

2.2.8 Cross

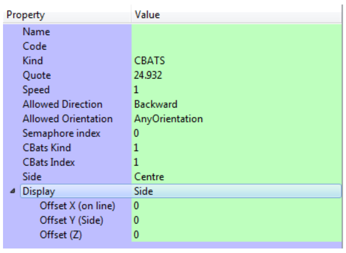
A cross is the intersection of 2 lines. An intersection have 4 quadrants. You can establish permission for vehicle in one or more quadrants. Three kinds of permission are available: Forbidden, curve and rotation fig.2.11.

Divieti is an 8 bit mask ?????????

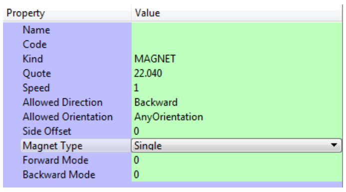
Override angle??????/



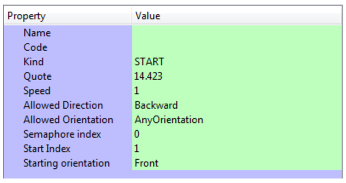
(a) User point properties



(b) Battery point properties



(c) Magnet point properties



(d) Start point properties

Figure 2.9: Map points

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.

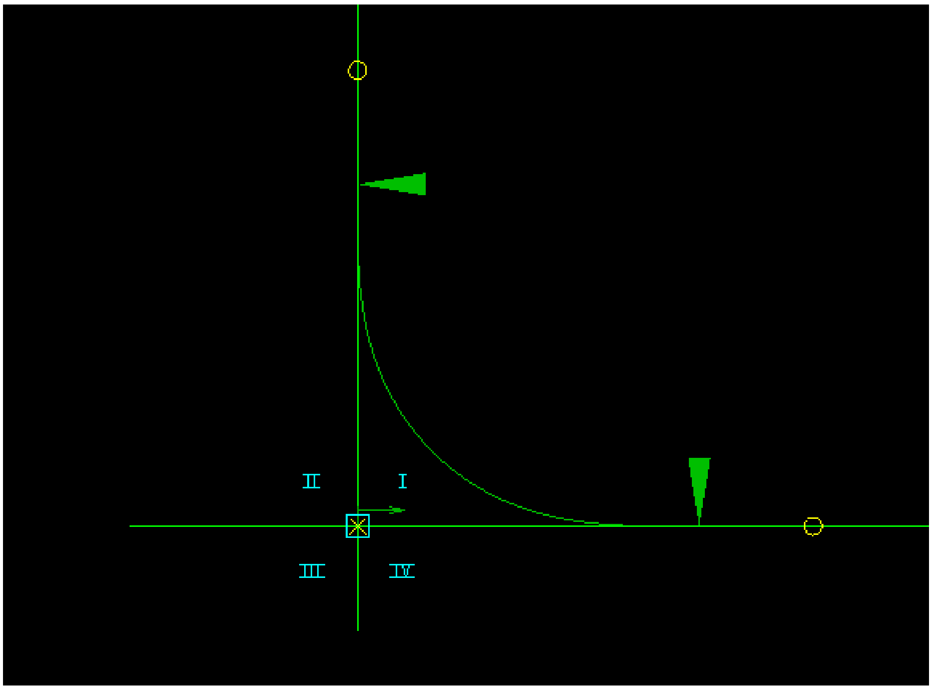
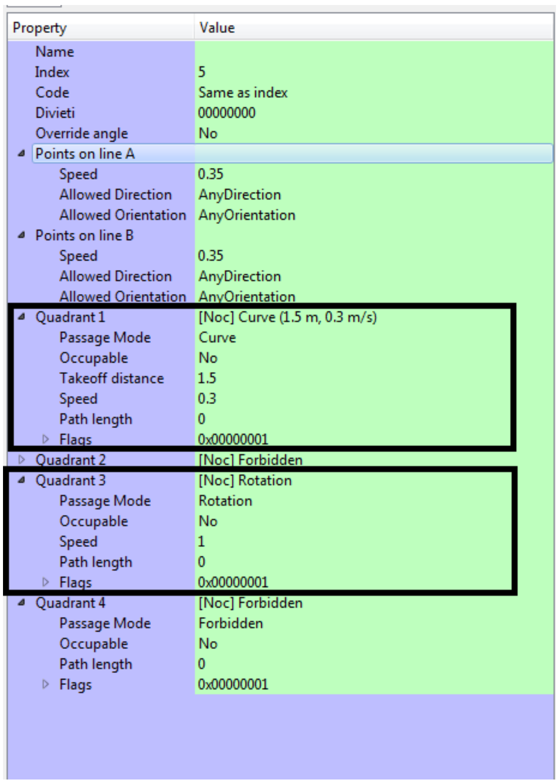


Figure 2.10: Two manget points should be place at least 0.5 meter from the end of a curve.



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.

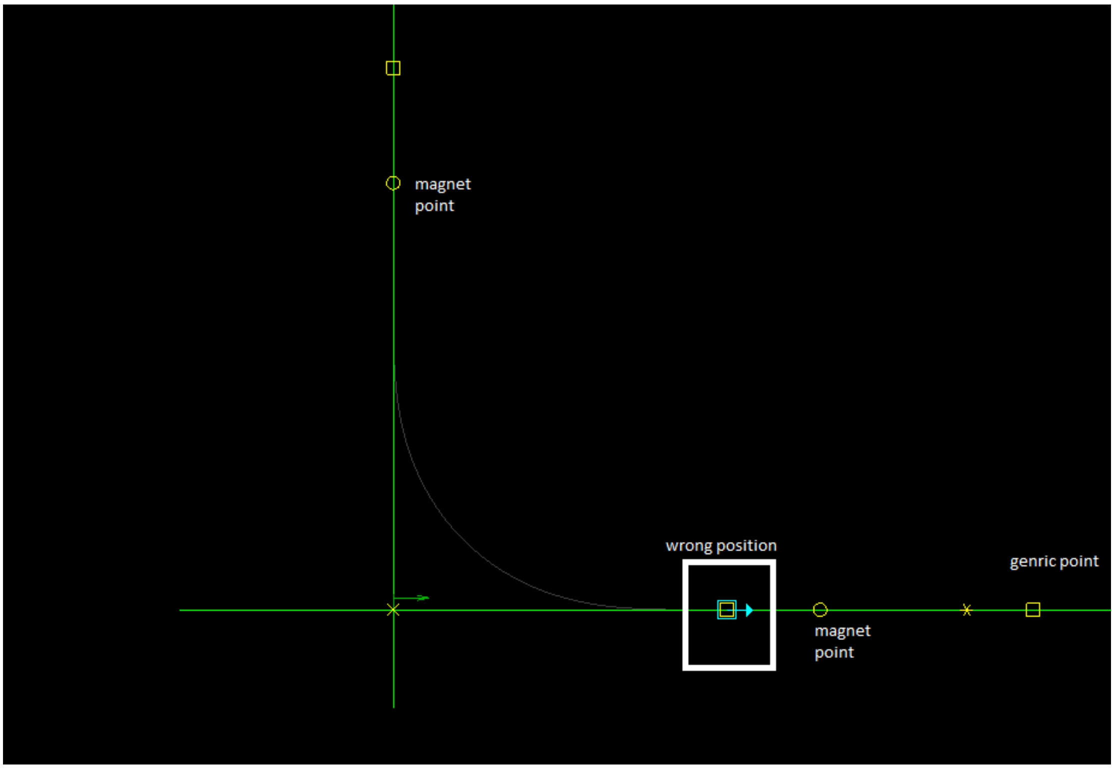


Figure 2.12: Generic points and user points should be placed outside a curve, i.e. after a magnet point.

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.

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

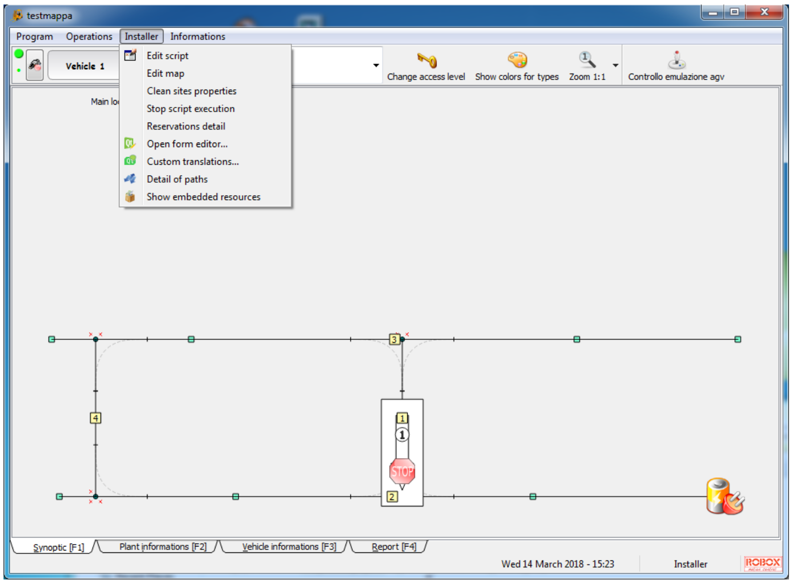


Figure 3.1: AGV Manager main window

fig.3.2, select the map fig.3.3 and agv 3.4 and plc communication. Project folder should be placed in the the AgvManager folder, otherwise it will not work. The script file should be in the first level of the project folder, it can't be placed in subdirectories, for example "*AgvManager/Project01/scripts/main.xs*" is not allowed.

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.3.5, 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 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.3.6 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

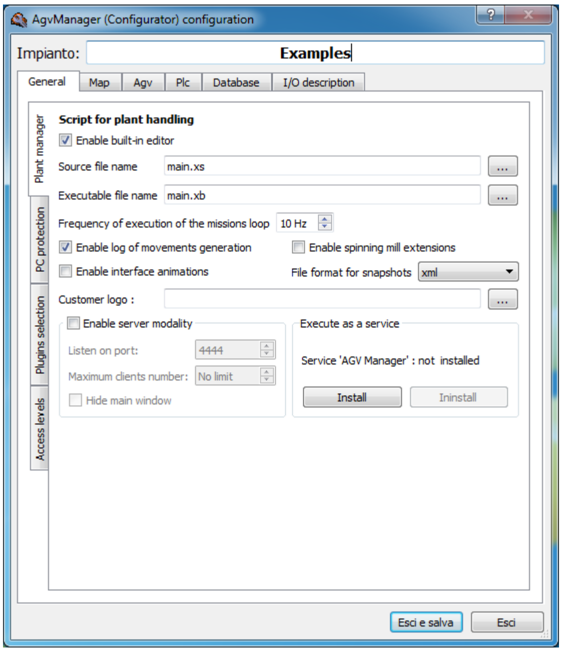


Figure 3.2: AGV configurator. General tab, where the *.xs* script file is selected. The script have to be created by an external editor. It is enough to write the name of the executable file with .xb extension, when AgvManager comple the xs file, the xb file will be created automatically.

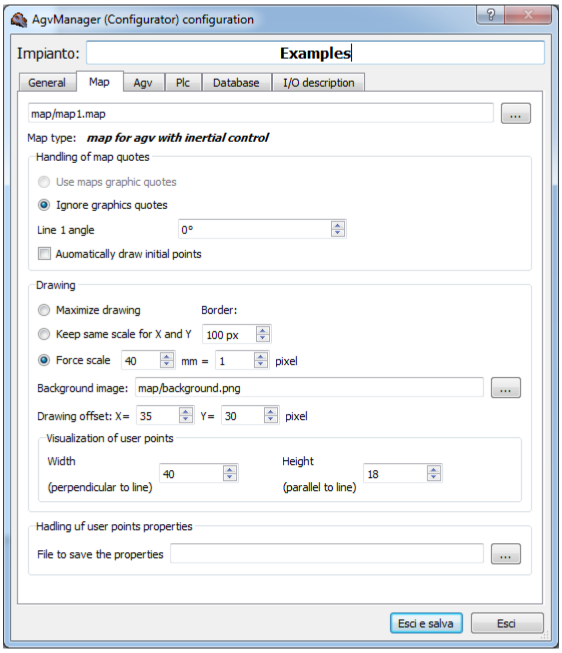


Figure 3.3: AGV configurator. Map tab, where the *.map* file is selected. In order to view the user point, you have to set the width and height of it, otherwise user points are not viewed.

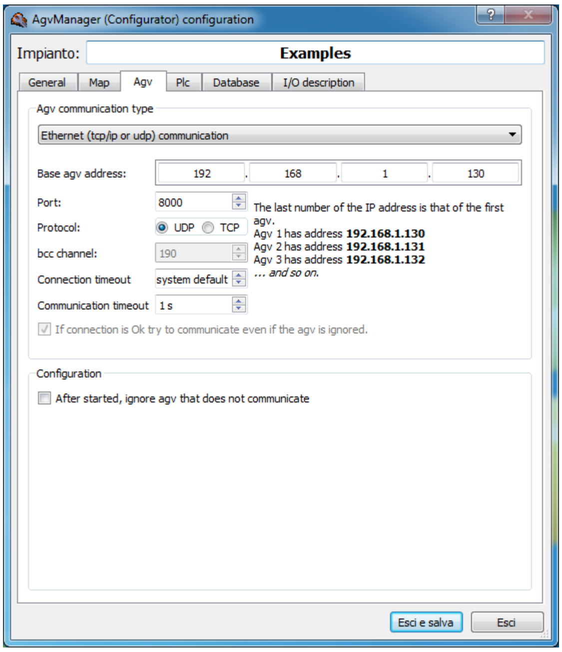


Figure 3.4: AGV configurator. AGV tab, where communication parameter with the AGV are set.

Figure 3.5: AgvManager tool bar

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

These flags correspond to:

```
// Vehicle status flags, bit mask 2^n.
// 4 least significant bits

$define VST_POTENZA_ATTIVA    1 // Power active, mask bit 0
$define VST_EXEC_COMANDO      2 // executing command, mask bit 1
$define VST_CARICA_INCORSO    4 // charge in progress, mask bit 2
$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 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.3.7.

In the tab Storage informations, fig.3.7a, 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.3.7b.

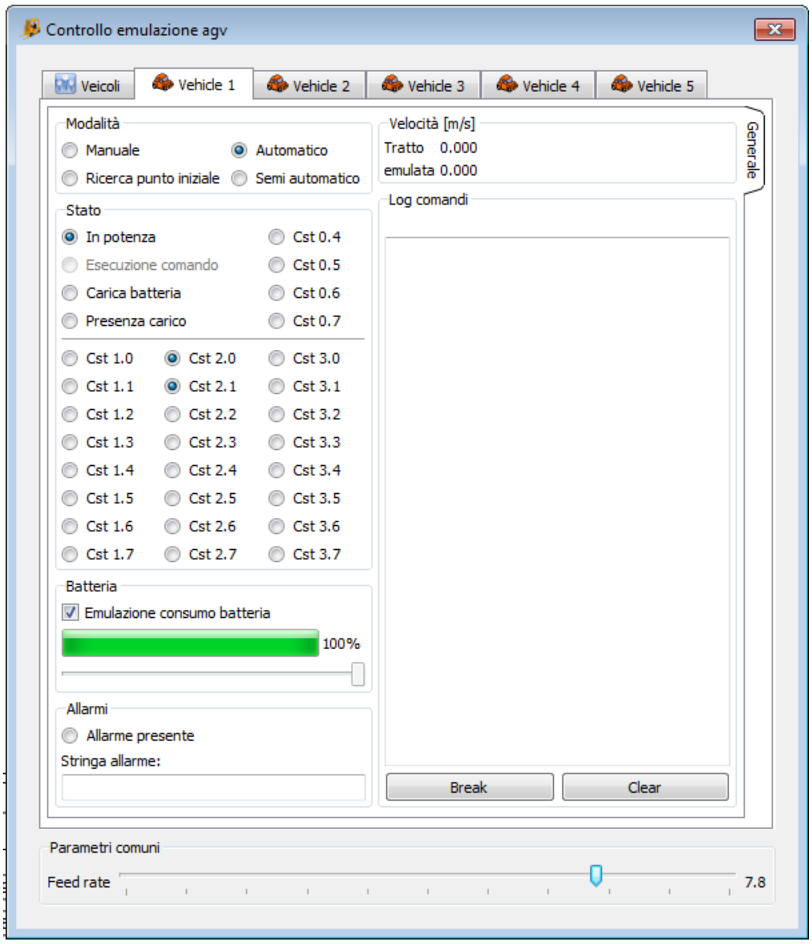


Figure 3.6: Emulation windows. We can see Agv status flags and other informations. The progress bar indicate the consumed power of the battery, not the remaining power, e.g. 100% is battery empty.

3.5 AGV script executing

AGV manager can be compared to a plc (hardware and firmaware) 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 se 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_P 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 defintions 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.3.8. 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.

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 by a call to *onNextMission()*, and started by calling *AgvStartMission()*. 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. At the last call of *onExecuteMicro()* the paramater *bLastCall* is assigned to true.

Fig.3.9 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.

3.6 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.3.10, 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 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 strucutre 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 *XMapParms*. 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 fucntion is done, AgvManager wait for some event. Let's suppose, the user drag the agv, in this case the event function *OnAgvDroppedToPoint* is called.

```
;
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

code OnApplicationStart() : bool
  SetVersioneManager(nvmake(MAJOR_VERSION, MINOR_VERSION, BUILD_VERSION
    ))

  XMapParams mpar
  ;
  ; Very important: before changing some parameters, load defaults!!!
  ;
  AgvGetMapParams(@mpar)
  ; Set vehicle dimensions: length, width
  mpar.setSymmetricalVehicleDimension(2300, 900)
  ;
```

```
    ; Parameters to calculate length of paths
    ;
    mpar.dHandicapForRotation = 8000        ; Distance added when using a
      cross for rotation
    mpar.dHandicapForCurve = 4000          ; Distance added when using a
      curve
    mpar.dDistanzaDaIncrocioOkFermata = 0   ; Minimum distance from cross
       to allow agv stop executing a movement
    ;
    ; Parameters to modify path assignment
    ;
    mpar.bOkInversioneSuIncrocio = false     ; True to permit inversion on
       a cross point
    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
    mpar.bPalletBloccaPercorso = true        ; Load unit (trolley) on path
      blocks the agv
;   mpar.bDontMoveOnDestMove = true          ; Do not
    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
    ;
    ; Set parameters
    ;
    AgvSetMapParams(@mpar)

    AgvSetLunghezzaMove(12000)

    SetAccessLevelForOperation(DefQual_OpTrascinaAgvSuLinea, ACCESS_INST)

    return true
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 meeted the agv can move from one point to another. The code to control such conditions is self-explainatory 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:

```
; Mission null, ther is no mission
$define MIS_NULL                0
; Mission move to point
$define MIS_TO_POINT            14

; 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 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 sMissioneDescription)*, 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 funntion *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.

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

```
;
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; Called when the user drags an agv (uAgv) to a point in map (uUser)
;
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
code OnAgvDroppedToPoint(uint uAgv, uint uUser)
  if (uAgv >= MAX_AGV)
    MessageBox("Invalid AGV number: " + (uAgv + 1))
    return
  end
  if (not AgvInAutomatico(uAgv))
    MessageBox("AGV " + (uAgv + 1) + " is not in automatic mode.")
    return
  end
  if (AgvAbilitato(uAgv))
    MessageBox("AGV " + (uAgv + 1) + " is enabled." + chr(10) + "Please
     disable it to give commands.")
    return
  end
  if (AgvActualMissionCode(uAgv) != MIS_NULL)
    MessageBox("AGV " + (uAgv + 1) + " is already executing a mission")
    return
```

```
  end
  ;
  AgvStartMission(uAgv, MIS_TO_POINT, "mission to point")
  ;
  uint wpidx
  uchar destOrientation = 'X'
  bool concatenateNext = true
  ;
  wpidx = AgvAddWaypoint(uAgv, uUser, destOrientation)
  AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
  ;
  AgvAddMacro(uAgv, MAC_END, MIS_TO_POINT)
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 OnAgv-DroppedToPoint() 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 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:

```
  case MAC_MOVE_TO_WP
    ; iPar1 = Waypoint id
    ; iPar2 = (bool) do concatenate next macro
      select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
   | WpFl_EliminaCompletato))
         case EsitoMov_MovimentoCompletato ; Completed movement
         case EsitoMov_RaggiuntoWaypoint   ; Waypoint reached
           if (iPar2)
             AgvComputeNextMacro(uAgv)
           endif
           return true
         default
```

```
            return false
        endselect
    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 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. 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 fucntion *agvStopMission(uagv)* is called, as we will see in the function *onExecuteMicro()*.

As shown in fig.3.9, AgvManager continue to call onExpandMacro() and onExcecuteMicro().

When the onExpandMacro() terminate the function onExcecuteMicro() 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 informations about them, fig.3.11, fig.3.12 and fig.3.13.

```
;
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; Do the job assigned to the macro that has actually to be executed
;
; Return TRUE when all work has been done, and the macro is finished.
;
; Return FALSE when the work has not been finished: the function
; will be called again for this macro
;
;
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

code OnExpandMacro(uint uAgv, uint uMission, uint iMacroCode, int iPar1
    , int iPar2, int iPar3, int) : bool
  ;
  ; Macro expansion, depending by the macro code
  ;
  select (iMacroCode)
    case MAC_MOVE_TO_WP
      ; iPar1 = Waypoint id
      ; iPar2 = (bool) do concatenate next macro
      select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
    | WpFl_EliminaCompletato))
        case EsitoMov_MovimentoCompletato ; Completed movement
        case EsitoMov_RaggiuntoWaypoint   ; Waypoint reached
          if (iPar2)
            AgvComputeNextMacro(uAgv)
          endif
          return true
        default
          return false
      endselect
      return true

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

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

    default
      qt_warning("Unknown macro: " + iMacroCode)
      break
    end
  return TRUE
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.3.11.

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.

```
$define S_START_WAIT         100
$define S_EXEC_WAIT          101
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.

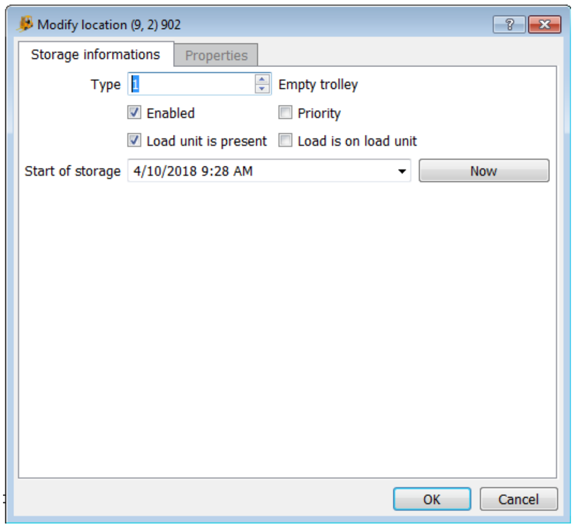
When expanding macros, the micro list is composed by calling the relative registration function. For example, in the function *OnExpandMacro()* under the MAC_END, we register a blocking system micro, S_END. In the function *onExecuteMicro()* under the case MIC_SYSTEM and under the case S_END we call the function *AgvStopMission(uAgv)* in order to stop the mission. When the mission is stopped, the micro terminate, and eventually other micros can start. When a micro terminate the execution of the function *onExecuteMicro()* return true.

```
;
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; Execution of the actions related to vehicle operations,
```

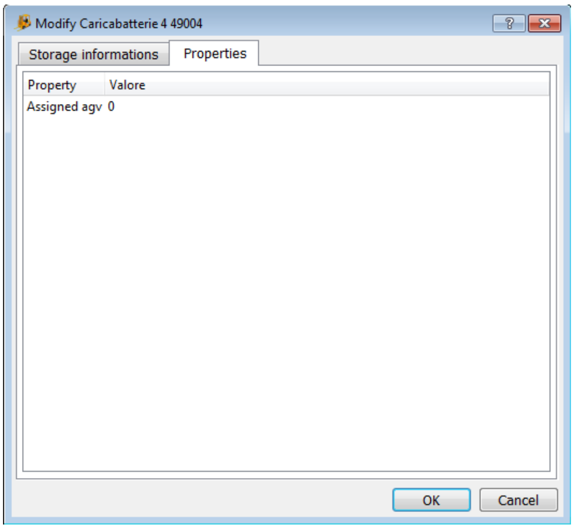
```
; and execution of the SYSTEM micro.
; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

  select (iMicroCode)

    case MIC_MOVE
    case MIC_CURVE
    case MIC_ROTATION
      return true

    case MIC_OPERATION

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

        case S_END
          ; End of mission
          if (vInfo.uStatus & VST_EXEC_COMANDO)
            MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": wait for agv commands finished")
            return false
          endif
          MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": finished executing commands")
          AgvStopMission(uAgv)
          SetAgvMessage(uAgv, "")
          break
        default
          qt_warning("Unknown MIC_SYSTEM : " + iPar0 + " (mission = " + iMission + ", par1 = " + iPar1 + ")")
          break
      end
      break
```

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

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

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

  end
  return true
end
```

Listing 3.5: OnExecuteMicro



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



(b) Location properties

Figure 3.7: Location window: Storage information and properties

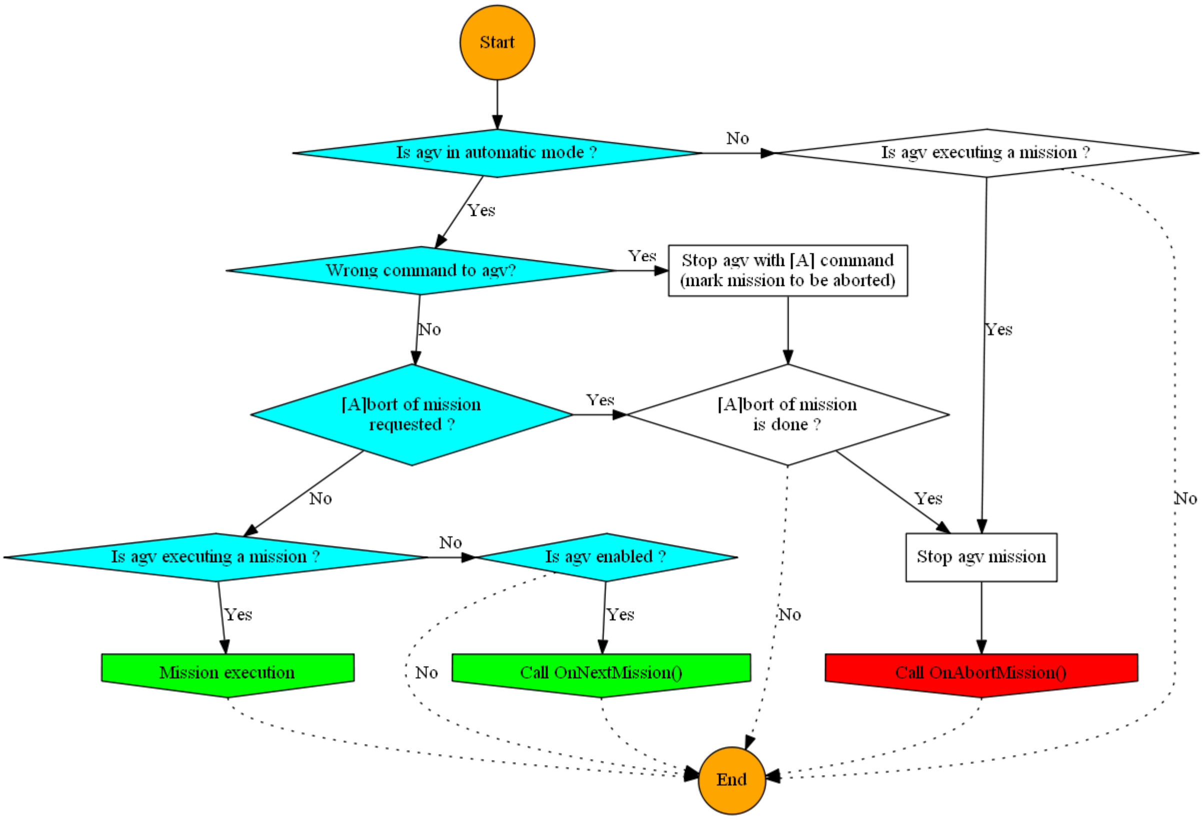


Figure 3.8: Main loop execution

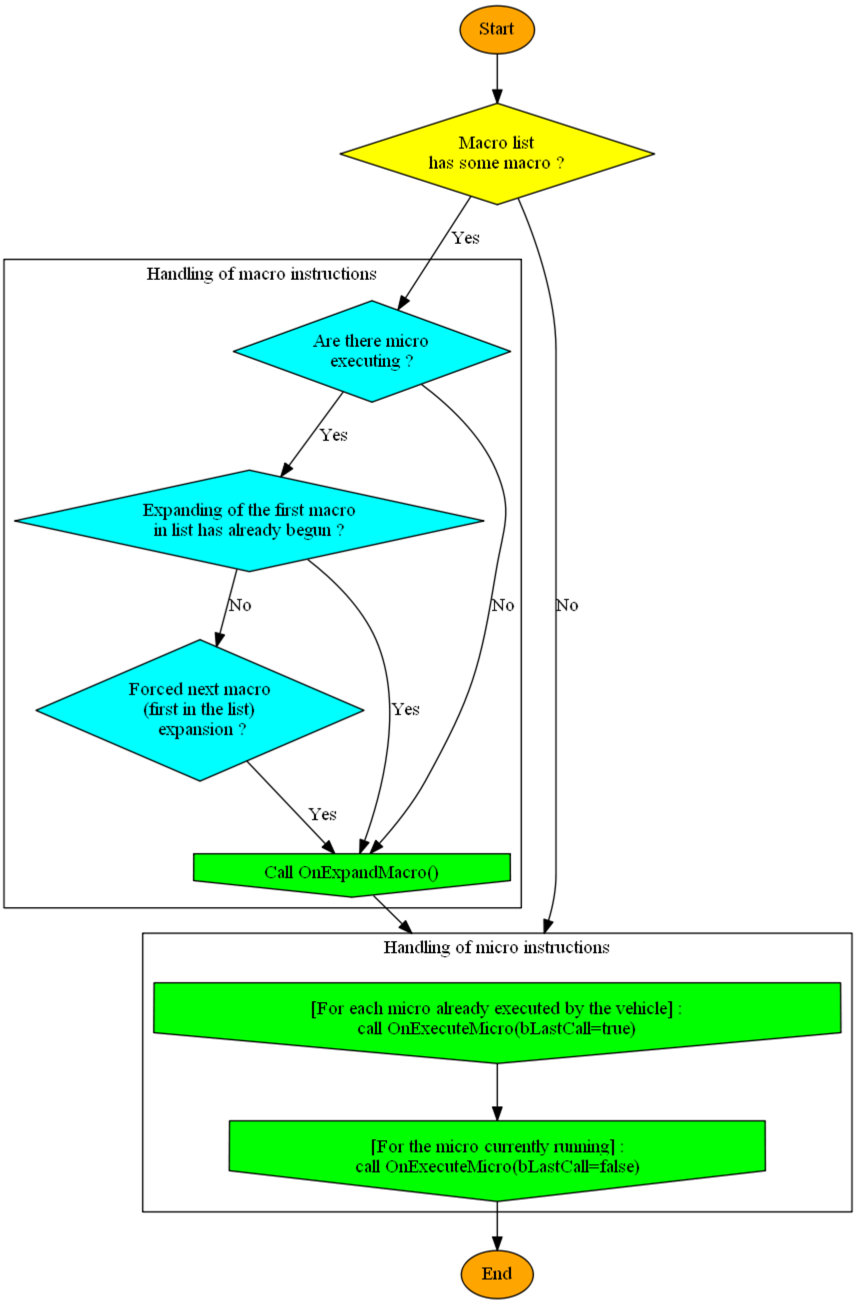


Figure 3.9: Main loop execution

Figure 3.10: Agv simple map, for drag and drop example. One line and two generic points.

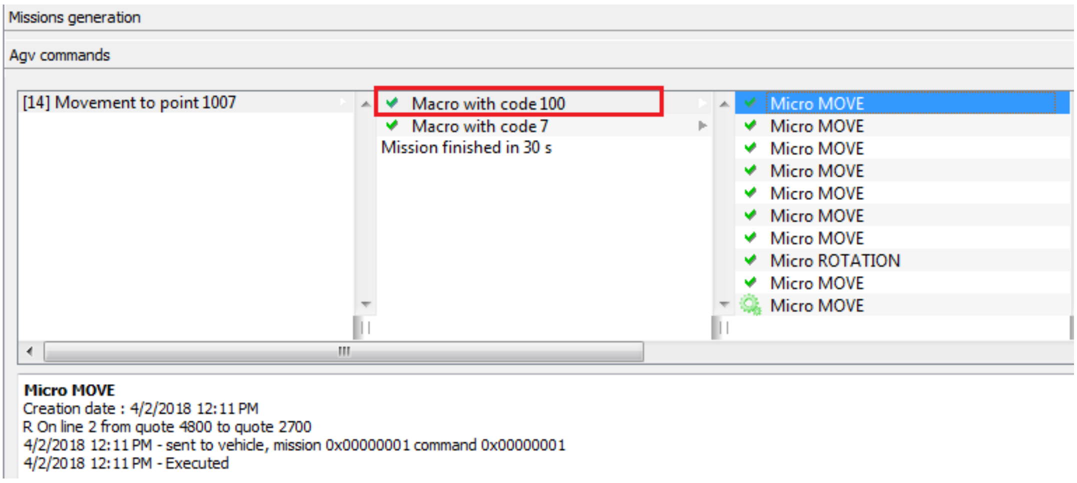


Figure 3.11: Movment to point 1007, Macro MAC_MOVE_TO_WP=100. As we can see, the macro consist of a list MICROs. Selecting a mission or a macro or a micro we can see informations about them.

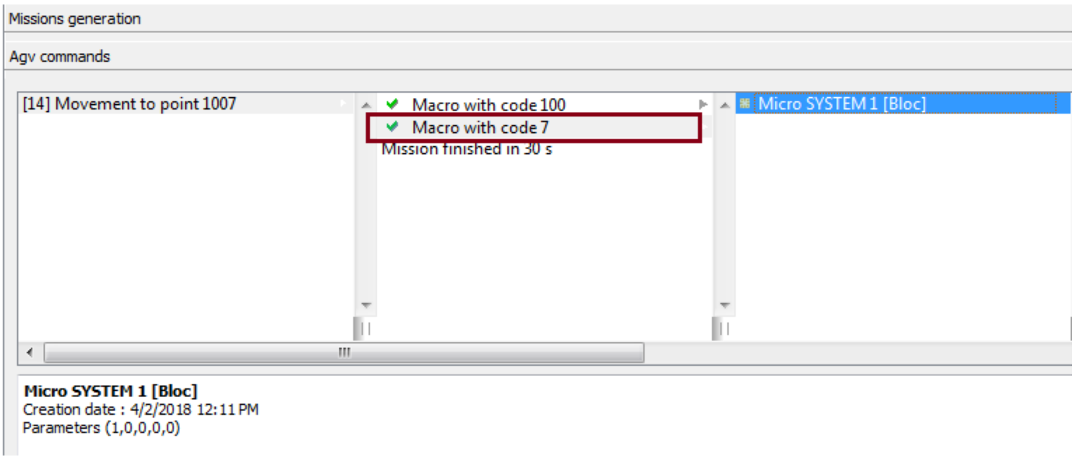


Figure 3.12: Movement to point 1007, Macro MAC_END=7. As we can see, the macro consist of on system micro that is S_END.

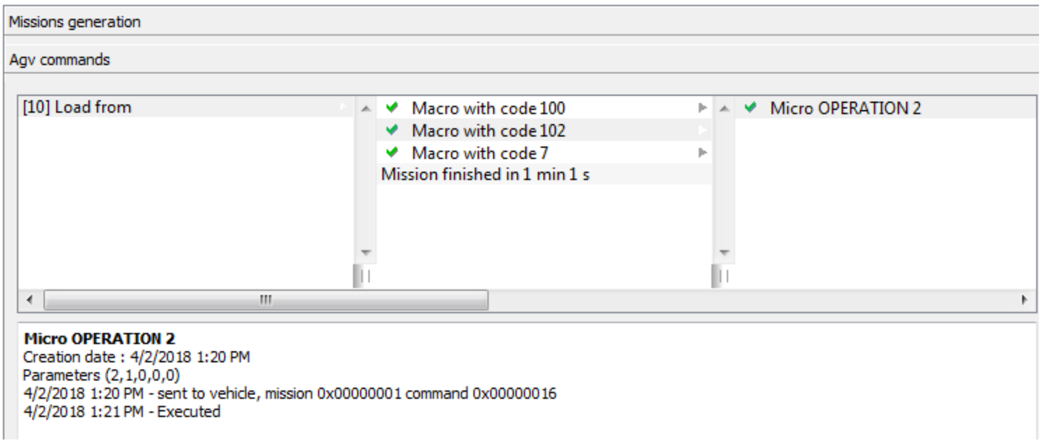


Figure 3.13: Mission load from a user point. The macro expansion shows 3 macros in the list. The MACRO 102, user defined, have only one micro of type operation that is O_LOAD, system defined. Later we will the commands sent to agv in order to execute operations.

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 expamples 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 presense of a load on board of the Loading Unit (UDC) we find the variable bVasiPieni.

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

4.1.1 XMapParams

This structures 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 AgvManger 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 we 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??????????

4.1.2 XVehicleInfo

This data structure contain 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:

```
// Vehicle status flags, bit mask 2^n.
// 4 least significant bits

$define VST_POTENZA_ATTIVA    1 // Power active, mask bit 0
$define VST_EXEC_COMANDO      2 // executing command, mask bit 1
$define VST_CARICO_PRESENTE   8 // load present, mask bit 3
$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.

4.1.3 XSiteInfo

A data structure where information about user points can be stored. 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 load 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.

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).

4.2 Some useful functions

We already see some callback funtions like onApplicationStart(), onNextMission(), onExpandMacro(), onExecuteMicro() and some utility functions like agvAddMacro(), agvAddWayPoint(), agvRegisterPassante(), agRegisterBloccante(), agvRegisterOperation(). There are a lot of functions provided by AgvManager. We will 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. agvAddWayPoint(), agvMoveToWayPoint(), AgvRegisterMoveTo(), to define the movement ways and methods as well as 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 .

more...........

- AgvRegisterMoveTo()

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 USER 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

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

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

4.3 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 4.1 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 4.1 can be found in the file agvEventFucntions.xs in the callback function OnAgvDroppedToPoint().

```
// note comments in Xscript begin with ;

XSiteInfo sInfo // user point information strucutre
XVehicleInfo vInfo // vehicle strucutre information

// if user point and vehicle exist
if (AgvGetSiteInfo(uUser, @sInfo) and AgvGetVehicleInfo(uAgv, @vInfo)
 )

  bool loadOnAgv, loadOnUser
  // read the bit corresponding to lpad present on agv
  loadOnAgv = (vInfo.uStatus & VST_CARICO_PRESENTE)

  loadOnUser = sInfo.bPresenza

  //if both agv and user point have a load
```

```
    if (loadOnAgv && loadOnUser)
      MessageBox("Cannot move agv " + (uAgv + 1) + " to " + GetSiteName(uUser) + " : both have a trolley")
      return
    endif

    // if only agv have a load, the mission unload to user is registered
    if (loadOnAgv && not loadOnUser)
      // call to use defined function
      RegisterMission(uAgv, MIS_UNLOAD_ONLY, uUser)
      return
    endif

    //if only user point have load, the mission load to agv is registerd.
    if (loadOnUser && not loadOnAgv)
      RegisterMission(uAgv, MIS_LOAD_ONLY, uUser)
      return
    endif

  endif

  // register movement to point,
  //if there is no lad neither on agv neither on user point,
  //or if the point is a generic point
  RegisterMission(uAgv, MIS_TO_POINT, uUser)
```

Listing 4.1: Drag and drop to user point and generic point

When the user drag and drop the vehicle onto a point, the callback function OnAgvDroppedToPoint() is called, then the function RegisterMission() is called inside it as we can in the listing 4.1.

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 fucntionB, and functionA call fucntionB, the program will give error, because he expect that fucntionB 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:

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

// No mission in progress
$define MIS_NULL                0

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

// MACRO definition, begin always from 100
// Movement to waypoint
$define MAC_MOVE_TO_WP              100
// Load from the point defined by par1
$define MAC_LOAD_TROLLEY              102
// 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:

```
// starting mission "uCode", with descrition "text"
if (not AgvStartMission(uAgv, uCode, text))
  return MIS_NULL
end
// user point info strutcture
```

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

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

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

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

```
      // END of this mission
      AgvAddMacro(uAgv, MAC_END, uCode)
      break

    // movement to a point mission
    case MIS_TO_POINT
      //Move to selected point
      RegisterMovement(uAgv, iPar1)
      //END of this mission
      AgvAddMacro(uAgv, MAC_END, uCode)
      break

    // mission not defined
    default
      MessageBox("Mission not implemented: " + uCode)
      return MIS_NULL
  end
```

Listing 4.2: RegisterMission() code fragment

The function RegisterMovement() is self-explanatory.

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

Listing 4.3: 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.

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.

```
code OnExpandMacro(uint uAgv, uint uMission, uint iMacroCode,
        int iPar1, int iPar2, int iPar3, int
        ) : bool

  select (iMacroCode)
    case MAC_MOVE_TO_WP
      // iPar1 = Waypoint id
      // iPar2 = (bool) do concatenate next macro
      select (AgvMoveToWayPoint(uAgv, uMission, WpFl_RicalcolaPercorsi
 | WpFl_EliminaCompletato))
        case MoveResult_CompletedMovement ; Completed movement
        case MoveResult_WaypointReached   ; Waypoint reached
          if (iPar2)
            AgvComputeNextMacro(uAgv)
          endif
          return true
        default
          return false
      endselect
      return true

    case MAC_LOAD_TROLLEY
      // par1 is the point
      // 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
      AgvRegisterOperation(uAgv, uMission, O_LOAD, iPar2, iPar3, 0, 0,
 iPar1)
      break

    case MAC_UNLOAD_TROLLEY
      // par1 is the point
      AgvRegisterOperation(uAgv, uMission, O_UNLOAD, 0, 0, 0, 0, iPar1)
      break

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

    default
```

```
        qt_warning("Unknown macro: " + iMacroCode)
        break
    end
    return TRUE
end
```

Listing 4.4: 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 micros are already defined by AgvManager. Search in the official documentation the prefixes O_ ad S_ to find a complete list Operations and System micro type.

The macro MAC_MOVE_TO_WP execute the movement command by calling AgvMoveToWayPoint(,,) 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 fucntion onExecuteMicro().

OnExecuteMicro()

We see how micros are registered when macros are expanded. Now we see how micros 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.

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

```
        MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : loading from " + userId)
        SetAgvMessage(uAgv, "Loading")
        return false
      endif
      break

    case O_UNLOAD
      if (bLastCall)
        MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : unloaded to " + userId)
        SetAgvMessage(uAgv, "")
        AgvExecUnload(uAgv, userId)
        return true
      else
        MultiMessageState(uAgv, "Agv " + (uAgv + 1) + " : unloading to " + userId)
        SetAgvMessage(uAgv, "Unloading")
        return false
      endif
      break
    default
      qt_warning("Unknown MIC_OPERATION : " + iPar0 + " (mission = " + iMission + ", par1 = " + iPar1 + ")")
      break
  end
case MIC_SYSTEM
  select (iPar0)
    case S_NULL
      // Micro of that type are generated by AgvManager, I am not intereseted on it.
      break
    case S_END
      // End of mission
      if (vInfo.uStatus & VST_EXEC_COMANDO)
        MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": wait for agv commands finished")
        return false
      endif
      MultiMessageState(uAgv, "Agv " + (uAgv + 1) + ": finished executing commands")
      AgvStopMission(uAgv)
      SetAgvMessage(uAgv, "")
      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 onExecuteMciro() 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.

4.4 Ex 02: Comple exaple

4.4.1 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:

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

The actual level can be read by calling the function ActualAccessLevel().
SetAccessLevelForOperation(DefQual_OpTrascinaAgvSuLinea, ACCESS_INST).

4.4.2 Settings: XSettings

settings.ini file strucuture

4.4.3 Interaction with user: XForm

Qt creator

4.4.4 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 fucntion agvGetInputXXXX and write outputs by calling agvSetOutputXXXX.

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

Note that the first bit have $offset = 1$ not 0. It is convenient to define some constants that represent IO signals. For example it 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.

4.4.5 Semaphores

AgvGetSemaphoreRequestMask AgvSetGreenSemaphore

4.4.6 Agv status flag change : OnAgvStatusChange()

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

The fucntion 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 descrition of the status bit VST_CARICO_PRESENTE we have to write:

```
loadType = TYPE_EMPTY_TROLLEY
SetAgvStatusDescription(uAgv, -3, "<font color=" + colorName(
 AgvGetTYpeColor(TYPE_EMPTY_TROLLEY)) + ">Trolley on agv</font>")}
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() set 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.

4.4.7 Agv Operating mode change: OnAgvModeChange()

When the vehicle operating mode (xVehicleInfo.uMode) changed, AgvManager call 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.

II Motion control

5. 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.

5.1 Notepad++

5.2 Ultra Edit

Appendices

IV

Bibliography