Robox notes

Work in progress 2018-03-15 rev 0.1

Robox motion control

Contents

II	Motion control

III	Bibliography

1. Introduction

TODO

AGV Manager

To manage and AGV using Robox software, 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 x-script language, and given to the core that execute that script. The core handle also the communication with the motion controller and eventually a plant PLC.
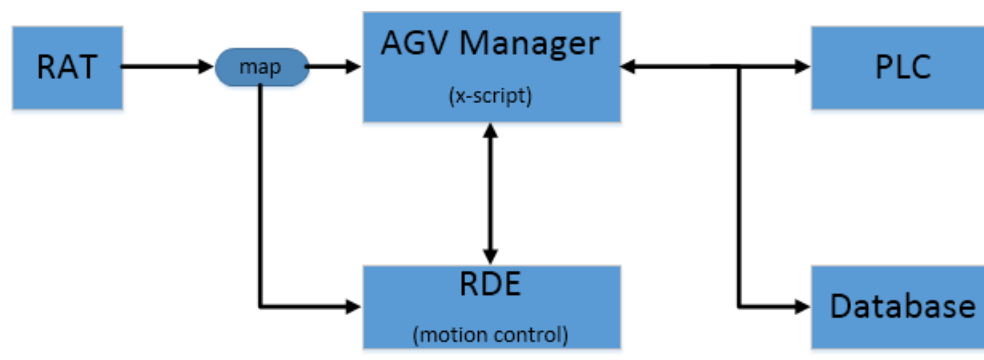

Figure 1.1: AGV block diagram. RAT give a .map file as output. The map is read by AGV manager and RDE. AGV manager may communicate with a PLC or a database as an interface to the plant IO, and with RDE for motion control.

2. RAT: Robox Agv Tool

2.1 RAT

RAT is a CAD software, fig.3.4, 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 and vehicles. During the design of the 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 Lines

A line have mainly 2 properties, beside its location fig.2.3. Navigation direction and vehicle orientation. Two directions are allowed: Forward and backward. Forward direction is shown by the arrow on the line, that is the positive movement. The vehicle can move longitudinally fig.2.4 to the line or transversally (side navigation) fig.2.5.
Precisely this line is a vector, so it have a direction. 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 $\overrightarrow{P_1P_2}$.

2.2.2 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 point 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). 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 other 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 refered to the vehicle fig.2.9.

The agv have strange behavior, I din't understand how it work.

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

2.2.3 Battery point

2.2.4 Magnet point

2.2.5 Start point

2.2.6 User point

2.3 Cross
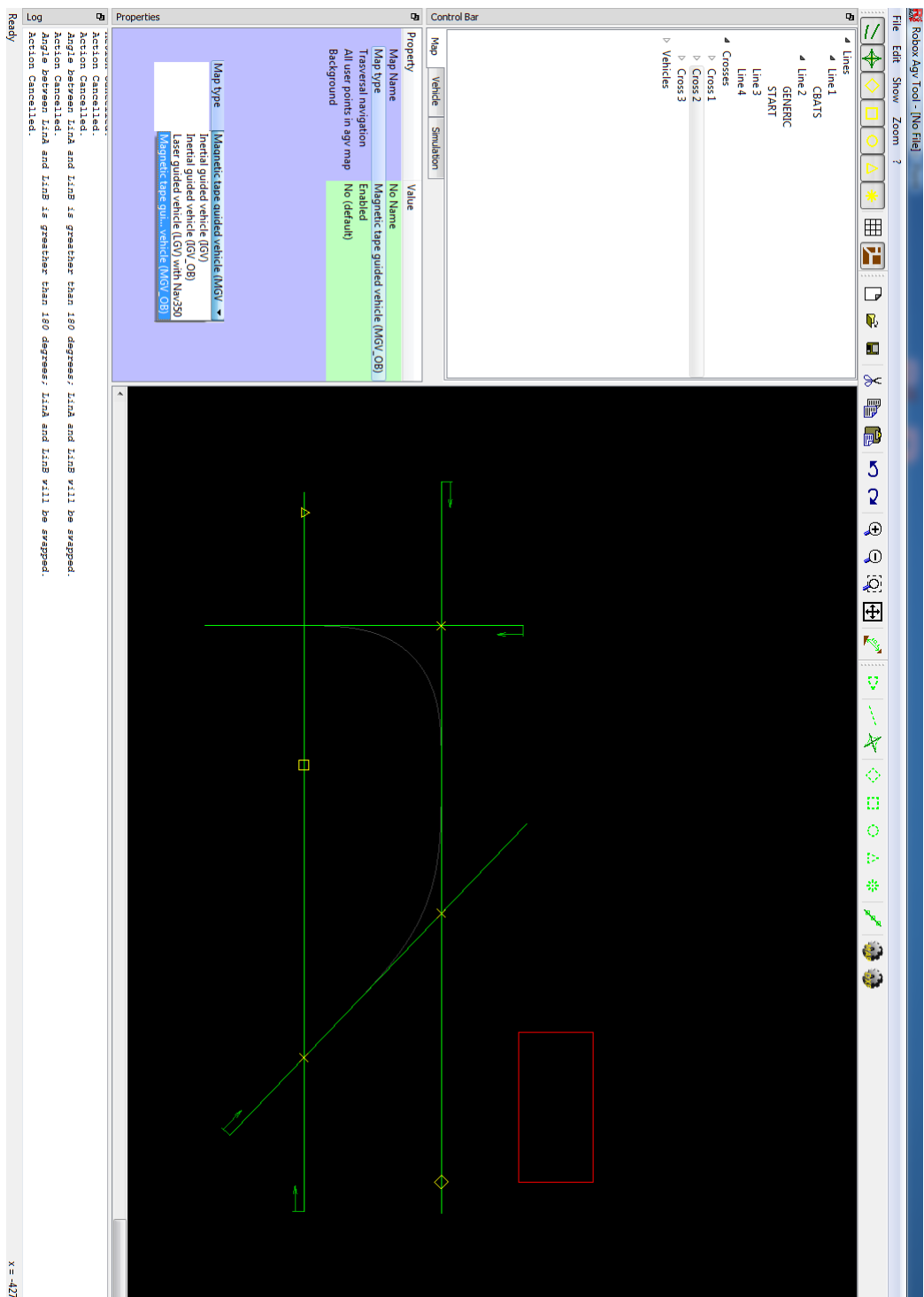
Figure 2.1: RAT main window
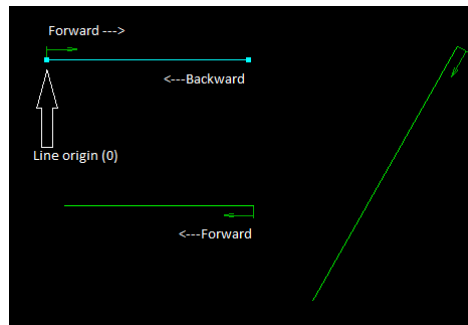
Figure 2.2: Line or vector. Direction of motion

Figure 2.3: Line property. Navigation type can be set : side, longitudinal or both
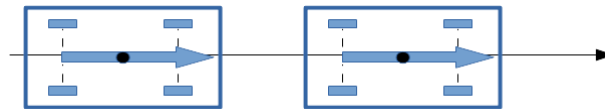
Figure 2.4: Longitudinal navigation
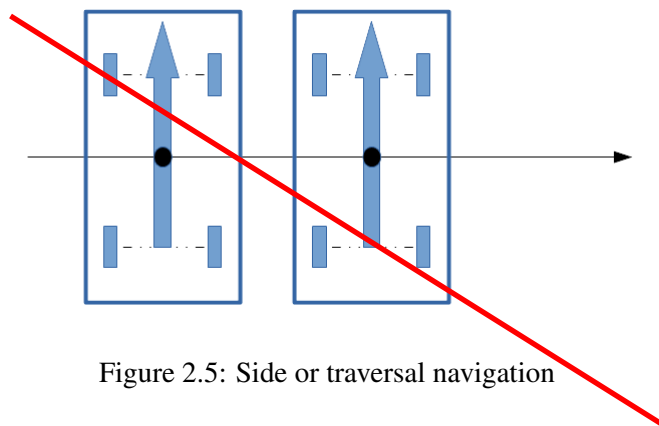
Figure 2.5: Side or traversal navigation
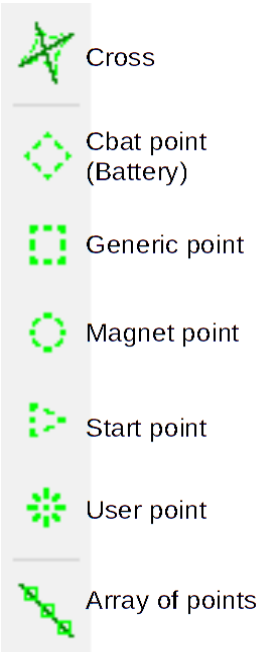
Figure 2.6: Kind of points

Figure 2.7: Generic point property
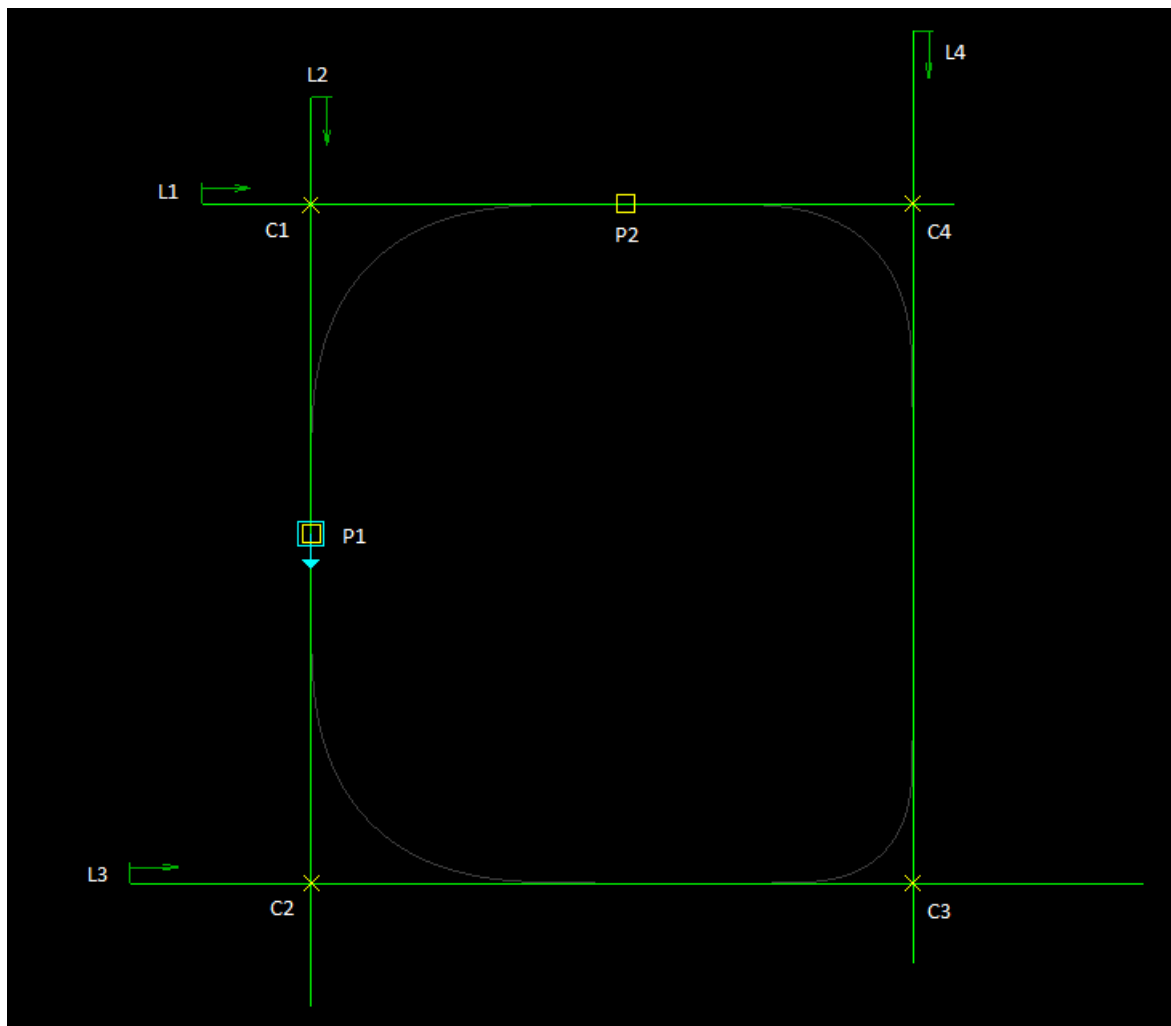
Figure 2.8: Point allowed direction. P_1 allowed direction is set to Forward. A vehicle moving from P_1 to P_2 will cross C_2, C_3, C_4. Instead a motion from P_2 to P_1 will cross only C_1
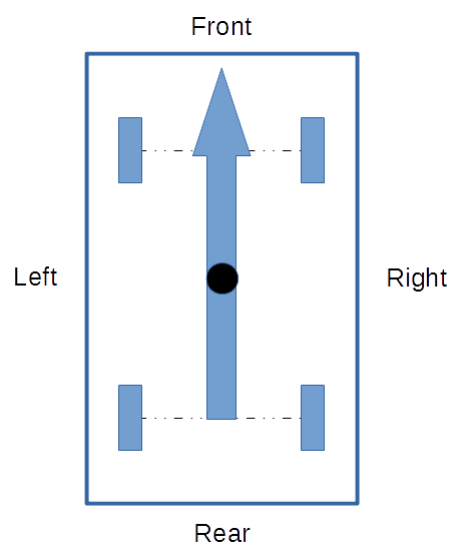
Figure 2.9: Vehicle orientation

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 language) and executed by AGV manager.

3.2 Installation and creation of new project

TODO

3.3 AGV configurator

AGV configurator is a standalone program that create a configuration file *.fdoc* for AGV manager.

3.4 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.

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

Figure 3.1: AGV Manager main window

Figure 3.2: AGV configurator. General tab, where the *.xs* script file is selected

Figure 3.3: AGV configurator. Map tab, where the *.map* file is selected

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

3.4.1 Fundamental concepts

Callback functions are called automatically by AgvManager. A list of callback functions (not all) can be found in the documentaion *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 automatically by AGV manager.
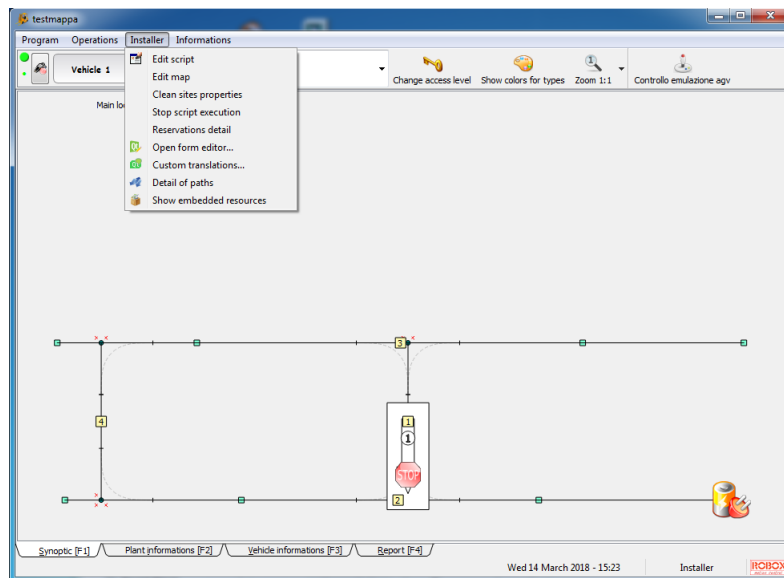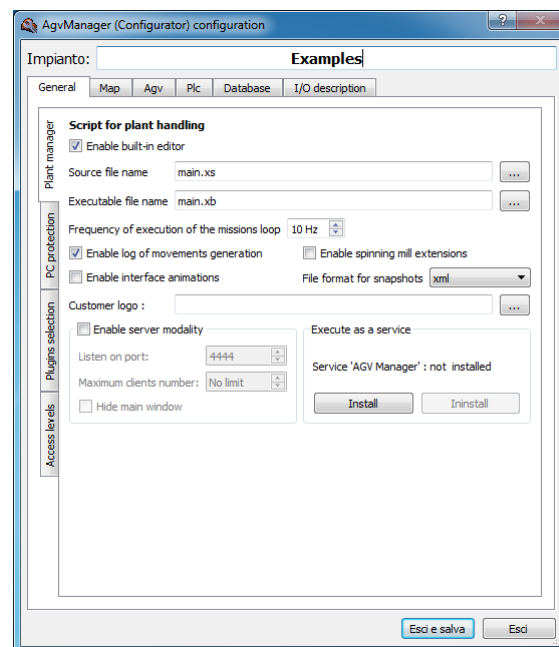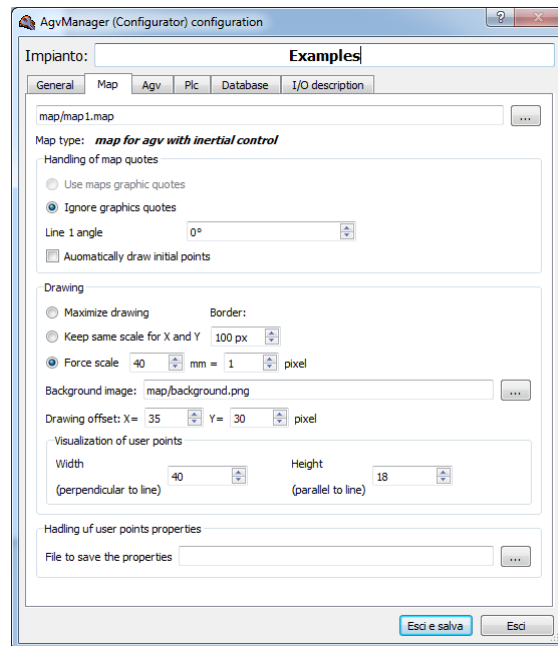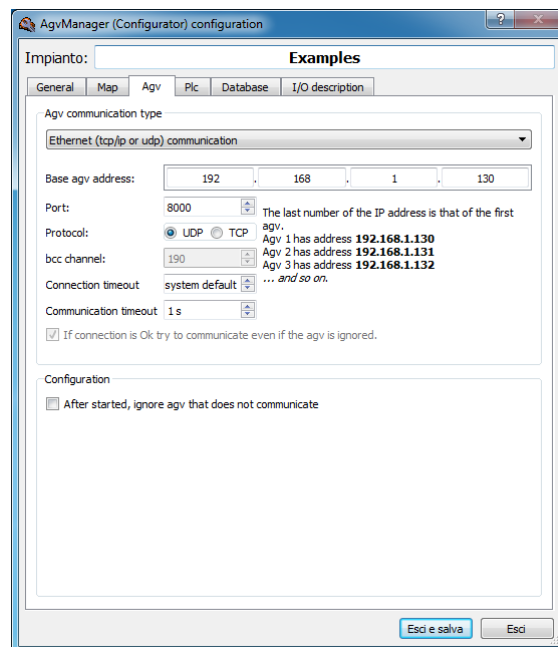
Some variable definitions (using #define keyword) can be found in the documentation *x-script interface, Modules, Estensione x-script per AgvManager, Funzioni per la gestione degli agv*. For example the AGV operative modes can be find there with the prefix MOD_, i.e. MOD_AUTOMATICO.

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 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.

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 MARCROs, 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.
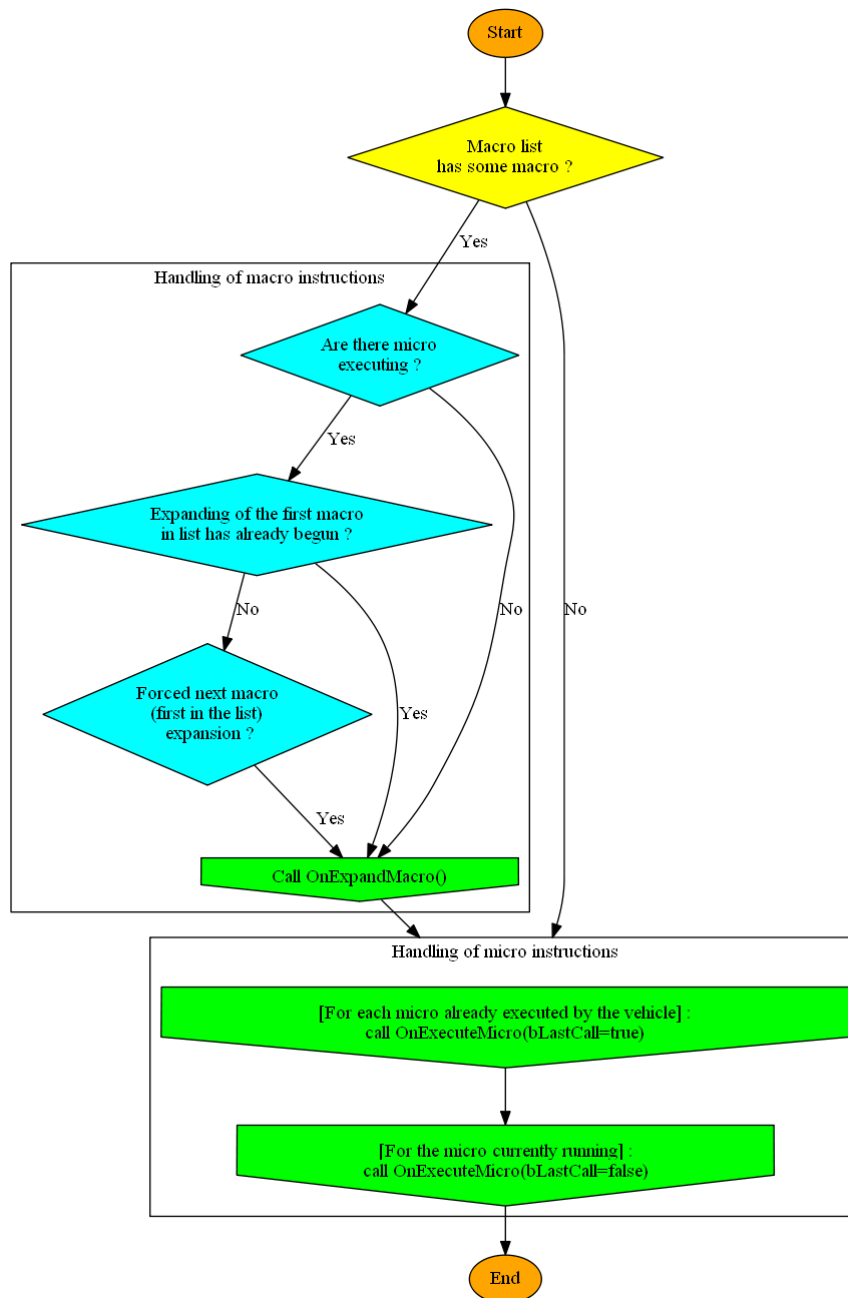
Figure 3.6: Main loop execution

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 strucutre 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 funtions 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.

```
1  ;
       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

5  code OnApplicationStart() : bool
   SetVersioneManager(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 ;
```

```
17   ; Parameters to calculate length of paths
     ;
     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-droppedtoline()*. 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-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. 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
```

```
      $define  MIS_TO_POINT                    14
5
      ;  MACRO  Movement  to  waypoint
7     $define  MAC_MOVE_TO_WP                   100
```

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

1. MAC_MOVE_TO_WP
2. MAC_END

Before starting a new mission we check if the is mission in progress. We call the function *AgvActualMissionCode(uint uAgvId)*, this function return the id of the mission in progress. If it reurn zero, it means there is no mission in progress. We have already define 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 return true if the mission is in progress.

Now we have to fill the agv <u>MACRO list</u> with our 2 MACROs, by calling the fucntion *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 fucntion take a paramter 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 list.

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

```
     ;
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2    ;  Called  when  the  user  drags  an  agv  (uAgv)  to  a  point  in  map  (uUser)
     ;
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
4    code  OnAgvDroppedToPoint(uint  uAgv,  uint  uUser)
       if  (uAgv  >=  MAX_AGV)
6        MessageBox("Invalid  AGV  number:  "  +  (uAgv  +  1))
         return
8      end
       if  (not  AgvInAutomatico(uAgv))
```

```
10      MessageBox("AGV " + (uAgv + 1) + " is not in automatic mode.")
        return
12    end
    if (AgvAbilitato(uAgv))
14      MessageBox("AGV " + (uAgv + 1) + " is enabled." + chr(10) + "Please disable
         it to give commands.")
        return
16    end
    if (AgvActualMissionCode(uAgv) != MIS_NULL)
18      MessageBox("AGV " + (uAgv + 1) + " is already executing a mission")
        return
20    end
    ;
22    AgvStartMission(uAgv, MIS_TO_POINT, "mission to point")
    ;
24    uint wpidx
    uchar destOrientation = 'X'
26    bool concatenateNext = true
    ;
28    wpidx = AgvAddWaypoint(uAgv, uUser, destOrientation)
    AgvAddMacro(uAgv, MAC_MOVE_TO_WP, wpidx, concatenateNext)
30    ;
    AgvAddMacro(uAgv, MAC_END, MIS_TO_POINT)
32 end
```

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

OnExpandMacro()

As we 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:

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

In this code the motion instruction is done by calling AgvMoveToWayPoint(), when this function return a value corresponding to 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 registred by calling *AgvRegisterSystemBloccante(uAgv, uMission, S_END)*.

As shown in fig.3.6, AgvManager continue to call onExpandMAcro and onExcecuteMicro.

When the onExpandMAcro() terminate the fucntion onExcecuteMicro() 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, "")
38     AgvRegisterSystemBloccante(uAgv, uMission, S_END)
       break
```

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

Listing 3.3: OnExpandMacro

OnExecuteMicro()

MICROs are instructions to the vehicle. MICROs are stored in a list, one MACRO can register more
than one MICRO. A MICRO is registered by calling agvRegisterSystemBloccante() or agvRegister-
SystemPassante(). 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
execusion of itself or till the verification of a condition.

There are different types of MICROs, that can be found int he documention 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
      case MIC_CURVE
13    case MIC_ROTATION
        return true

15    case MIC_OPERATION

17
      case MIC_SYSTEM
19      select (iPar0)
          case S_NULL
21          ; Micro of that type are generated by AgvManager, I am not
    intereseted on it.
            break
23
          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

Motion control

Bibliography