# Machine API

## Introduction

The Machine API is used to communicate with the machine via a serial cable.

The goal with the API is to provide a simple way for program developers to interact with the Pick and Place machine.

Not all functions that are available in the machine have been implemented in the API, only the functions that where requested by the customer.

The Machine API is designed to be used both by the GUI team but also to enable other people to build programs that use the machine.

## Result

The Machine API can do all of the things that is required of it. All goals were met. The outcome of the design and implementation is as desired, except for maybe a minor hassle when adding a command to the API since this requires changes in multiple files.

Because of the way it is implemented the API only works in a Microsoft Windows environment. The reason for this is the use of windows threads and serial port libraries. This should be fairly easy to port to other platforms.

## Design and architecture

When designing the API there were a few requirements to satisfy

* should not block when executing a command
* easy to extend and add new commands
* able to handle communication failures and unexpected errors
* fast and memory efficient
* keep as simple as possible, avoiding complex solutions

From these requirements a UML diagram was crafted.

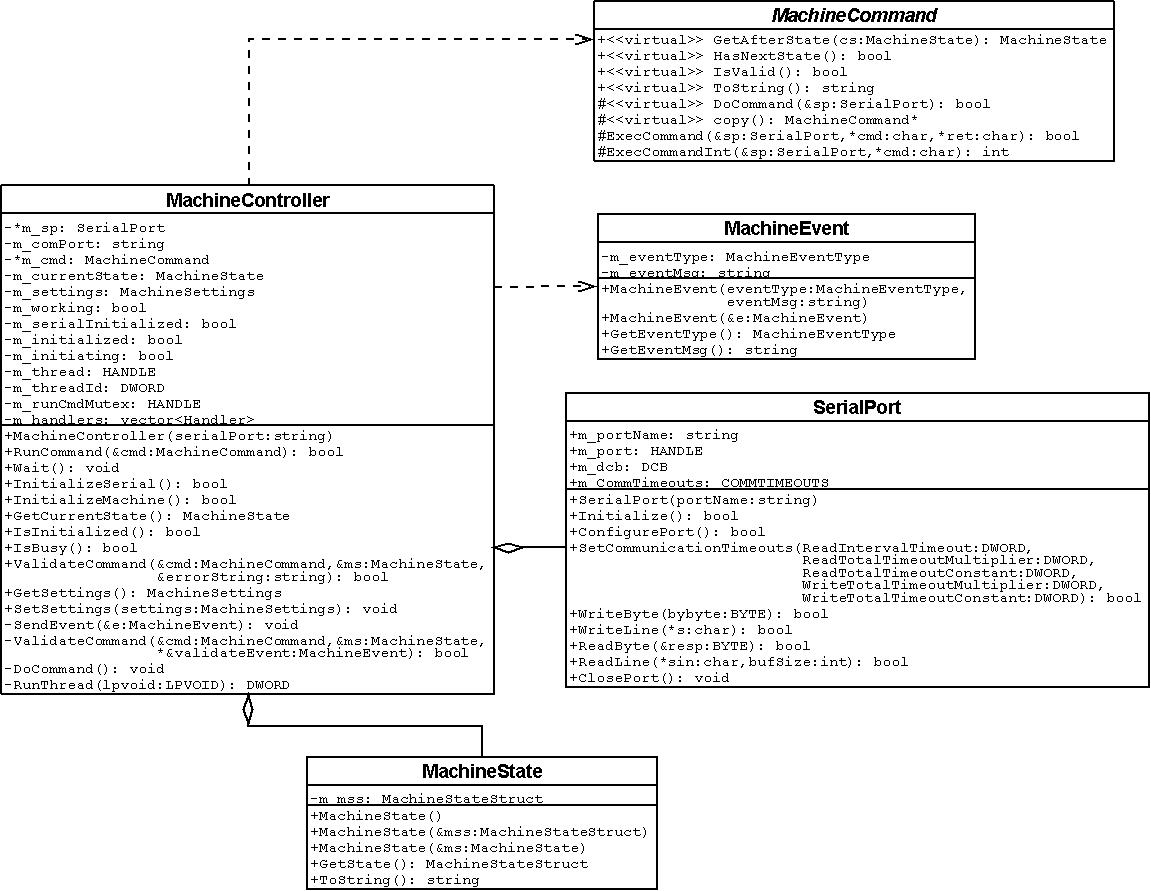


Figure UML diagram.

A main class, MachineController handles communication with the machine and is also the interface that applications use.

The MachineCommand interface was created to make creating new commands easy. This also has the benefit of making the design structured; with every new command being in its own class, implementing the MachineCommand interface.

To simplify the design, the API only handles one command at a time, although commands can be bundled with a MachineWrapperCommand object. Any command queuing is up to the caller to implement.

Execution of a command is done in a separate thread. Communication with the caller is done through an event system. An event is sent using a MachineEvent object, which contains the event type and an event message. These events are sent whenever a command has finished executing or an internal error has occurred. The caller registers a callback function in the MachineController that handles these events. Multiple callback functions can be used.

The MachineController contains the current state of the machine. This information is saved in the MachineState class. Information stored in this class is used to validate commands, store offsets for picking and dispensing, etc.

For more information about the commands and classes, see the doxygen documentation provided with the source code.

## Usage manual

### Commands

Creating a MachineController instance and initializing it:

#include "MachineController.h"

string comPort = “com1”; // Which com-port the machine is connected to

MachineController mc(comPort); // Create a new machine controller instance

/\*

\* Initialize the machine. This initializes the serial port and

\* issues a MachineInitCommand.

\* When this is done the machine is ready to accept commands.

\* This function returns true when the serial port is initialized,

\* but to know when the machine is ready a check for a MachineEvent

\* of type EVENT\_MACHINE\_INITIALIZED needs to be done.

\*/

if ( !mc.InitializeMachine() )

{

// Machine failed to initialize

return;

}

// Wait for the thread doing the initialization to complete.

// (This is never required, but no command can be issued until

// the current command is done)

mc.Wait();

Creating and running a command. Running any type of command is the same procedure.

// Move to x,y,z = ( 10000, 20000, 1000) µm from origo

MachineMoveAllCommand cmd(10000, 20000, 1000);

mc.RunCommand(cmd); // Validate and run the command.

mc.Wait(); // Wait for the machine to finish. (Optional)

Creating a polygon dispense command.

MachinePolygon mp; // Create a polygon to define the path

// Add points to the polygon:

mp.AddPoint(MachinePolygonPoint(10000, 10000)); // Index 0

mp.AddPoint(MachinePolygonPoint(10000, 50000));

mp.AddPoint(MachinePolygonPoint(20000, 50000));

mp.AddPoint(MachinePolygonPoint(20000, 10000));

mp.AddPoint(MachinePolygonPoint(30000, 10000));

mp.AddPoint(MachinePolygonPoint(30000, 50000));

mp.AddPoint(MachinePolygonPoint(40000, 50000));

mp.AddPoint(MachinePolygonPoint(40000, 10000)); // Index 7

mp.DelPoint(0); // Delete the point on the polygon with index 0

// (The point that used to be index 1 has now been moved to index 0)

mp.DelPoint(1); // Delete the point on the polygon with index 1

// Create a polygon dispense command from the polygon

MachinePolygonDispenceCommand cmd(mp);

mc.RunCommand(cmd);

mc.Wait();

Wrapping several commands into one

// Create a few commands

MachineMoveAbsoluteCommand moveX(AXIS\_X, 10000);

MachineMoveAbsoluteCommand moveY(AXIS\_Y, 20000);

MachineMoveAbsoluteCommand moveZ(AXIS\_Z, 1000);

// Create the wrapper and add the commands to it

// A wrapper takes several commands and bundles them into a command

// that executes them sequentially (and then optionally returns the machine to the coordinates

// it was at before the wrapper started.)

MachineWrapperCommand wrapper;

wrapper.Add(moveX);

wrapper.Add(moveY);

wrapper.Add(moveZ);

mc.RunCommand(wrapper);

mc.Wait();

For more information about each command, see the doxygen documentation.

### Adding a command

Create a new class that extends the MachineCommand class.

Example, adding “TestCommand “:

#### TestCommand.h

#ifndef \_\_TESTCOMMAND\_H\_\_

#define \_\_TESTCOMMAND\_H\_\_

#include "machinecommand.h"

class TestCommand : public MachineCommand

{

// Make the private methods of other commands

// availale to this command.

MACHINE\_COMMAND\_FRIENDS;

public:

TestCommand(Data inData);

~TestCommand(void);

string ToString();

MachineState GetAfterState(MachineState &oldms);

private:

TestCommand \*Copy();

bool DoCommand(SerialPort &sp);

Data m\_inData;

};

#endif // \_\_TESTCOMMAND\_H\_\_

Then add the command to the list in MachineCommands.hand to MACHINE\_COMMAND\_FRIENDS in MachineCommand.h.

If the command has several states, the *HasNextState()* methods needs to be overridden.

For more information, see the implementation of the commands.

### Events

Events are passed by calling a callback function. A callback functions should look something like this:

void handleEvent(MachineEvent \*e)

{

// Code for handling the event

delete e; // The event must be deleted when done

}

The callback function should then be registered to the API, like so:

mc.AddEventHandler(&handleEvent);

Now an event will be passed to the event handler when a command is finished, or has failed.

## Evaluation

We found a few memory leaks and situations where buffers could overflow. These were fixed.

## Building the API

The API has been successfully built using Microsoft Visual Studio 2008 and no other build systems have been tested.

Using Visual Studio 2008, open the solution file (MachineAPI.sln ) located in the MachineAPI folder in the source code.

When building, a library file (MachineAPI.lib) is produced, which can be linked from other projects.

## Known issues

The API is not in a specific namespace. Problems can occur with multiple classes of the same name.

## Discussion

The API works as desired.

## Future work

* Only use one thread for the commands, instead of spawning and new one every time and tearing it down after execution is completed.
* Implementing new commands if needed.
* Event on progress of wrapper command.