# Basic Training for Pioneer-3 Robot SDK

This documentation is intended for those who have access to MobileRobots/ActivMedia mobile robot platforms (e.g. Pioneer, PowerBot, AmigoBot) and wants to get to know the basics of using the SDK provided with them. The examples and configuration setup used here will be specifically for the Pioneer-3 robots, using C++ programming language on Win32 (WinXP) platform with testing done on a simulator.

## Requirements

Installation of these software...

- Aria - Pioneer SDK (C++ Toolkit)
- MobileSim - Robot Simulator for Pioneer Robot (Based on Player/Stage)
- C++ Compiler - Visual C++ 2008 Express Edition for Win32 OR GNU Compiler Collection for Linux

***Note**: All the above software are either open source or truly free software and can be downloaded from the respective websites.*

***Update**: Alternative download for VS2008 (with SP1)*

***Update**: GNU's `gcc4` compiler may not be supported (like, strict type-casting rules?) so, recommended version is 3.4*

***Hint**: Solution/Project/Source Files used in this document can be obtained from here*

***Hint**: The link given for MSVC++ is for an ISO image (suitable for redistribution on multiple PC) - to use it you need something like a Virtual CD Drive, unless you want to waste another CDR (who uses it anymore? 😮).*

## Environment Setup

The compiler setup for Win32 (Visual C++ 2008 Express Edition)... remember to do this for every new project you create. Alternatively, you can simply open an existing project and replace the source/header/resource files with the new ones.

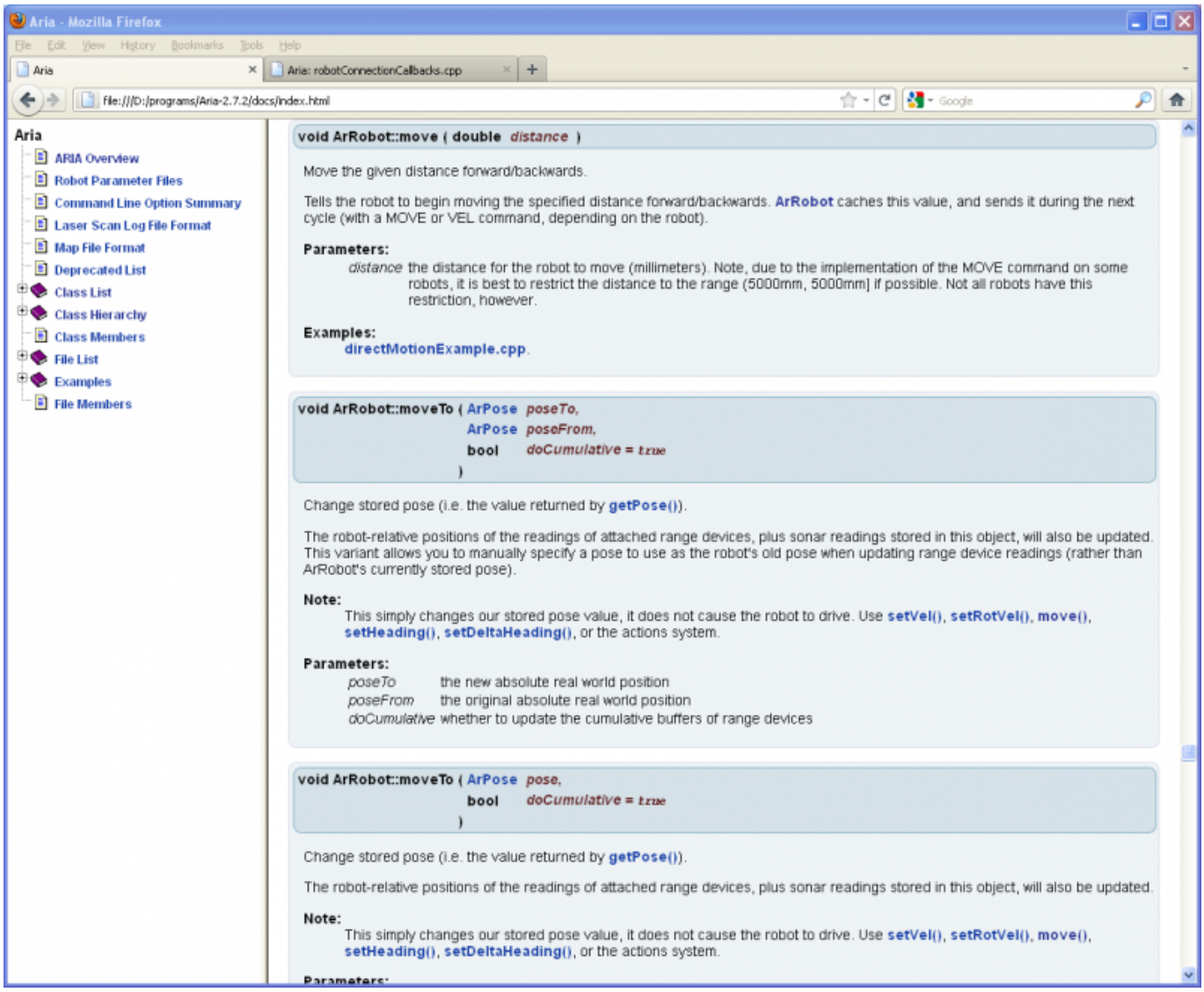
1. create an empty project (e.g. 'projectA')
2. right-click on projectA, click on Properties
    - you'll get 'projectA Property Pages' window
    - change Configuration drop-down list to All Configurations
    - look for Configuration Properties→C/C++ settings
    - add to Additional Include Directories: “<ARIA-install-path>\include”
    - look for Configuration Properties→Linker settings
    - add to Additional Library Directories: “<ARIA-install-path>\lib”

    - look for Configuration Properties→Linker→Input settings
    - add to Additional Dependencies: “ARIA.lib” (OR “ARIADebug.lib” if you want the debug version)
    - look for Configuration Properties→General settings (*this is optional*)
    - change Output Directory: “$(SolutionDir)\Bin”
3. right-click on projectA→Source Files, click on Add→New Item...
    - select C++ File, give a name (e.g. main.cpp), click Add

**Note**: You need to familiarized yourself with Visual C++ framework terminology like solutions and projects. It is possible to have multiple projects in a single solution. Each project can have different build environment/configurations.

# Introduction to ARIA C++ Library

It's just a library. Documentations for description of classes are available with the installation (check out the `docs` folder). We will be using the current (at the time this is written) latest version which is 2.7.2. A view of the provided documentation is shown below.



# A Simple Program

Given a simple C++ program (*note that it's not much different compared to a C program at the top level*) that basically initializes a robot and execute robot commands (in this case, nothing!). The code is shown below followed by some explanations on what each line does.

simple00.cpp

```
#include "Aria.h"

int main(int argc, char *argv[])
{
    int status = 0;
    ArRobot robot;
    ArSimpleConnector connector(&argc, argv);

    Aria::init();
    if(connector.connectRobot(&robot))
    {
        printf("Connected to robot... start to do stuffs!\n");
        robot.run(true);
    }
    else
    {
        printf("Could not connect to robot... exiting!\n");
        status = 1;
    }
    Aria::shutdown();

    return status;
}
```

Let's dissect the given code:

#include “Aria.h”
: Obviously the include directive for Aria header file(s)

ArRobot robot;
: Instantiation of Aria's Robot Object which is the primary class for the robot platform

ArSimpleConnector connector(&argc, argv);
: Instantiates the Connector Object that handles communications between the Robot Object (i.e. your program or commands) with the robot platform (i.e. firmware or simulator) and optionally sends any command-line parameter(s)

init();
: (Actually `Aria::init()`) Initializes Aria's Environment/Framework

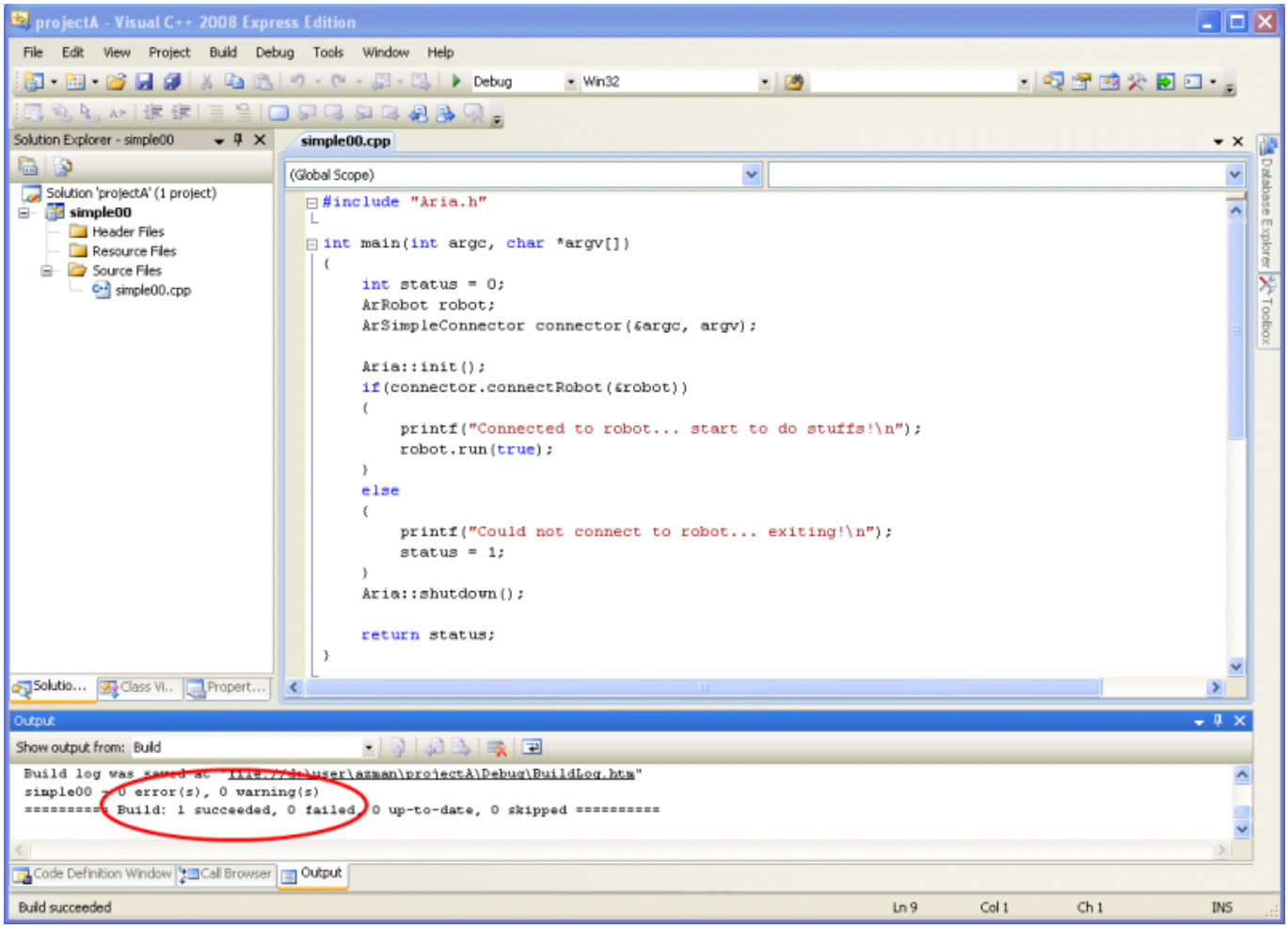
connector.connectRobot(&robot)
: Connector Object's API that tries to connect to a robot controller (either through ethernet or serial port) that can also be in form of a simulator

shutdown();

(Actually `Aria::shutdown()`) Gracefully terminates Aria's Environment/Framework
robot.run(true);

Starts the processing on connected robot platform (if any) and will return when it is no longer connected to the robot platform

The given program will compile cleanly in Visual C++ 2008 Express Edition as shown below.




However, it will not do anything much (nothing visible, at least). We'll discuss how to test the program in the next section. So, to make this program slightly more interesting, let's move the robot a bit.

***Note**: The `run` method is a blocking function - that means it'll keep executing until a return has been called (i.e. when there's no connection to the robot platform). This makes stopping the execution of the program a little bit tricky - there are two ways to do that and will be discussed later.*

# A Simple Program (The Robot Moves!)

If you go through ARIA's class API, you can find that there's a `move` method for ArRobot class that can move the robot straight for a given distance. This method requires the motors to be enabled with (duh!) `enableMotors` method. These are added to the previous code as shown in the next example.

***Note**: Methods are actually functions that are members of a C++ class.*

simple01.cpp

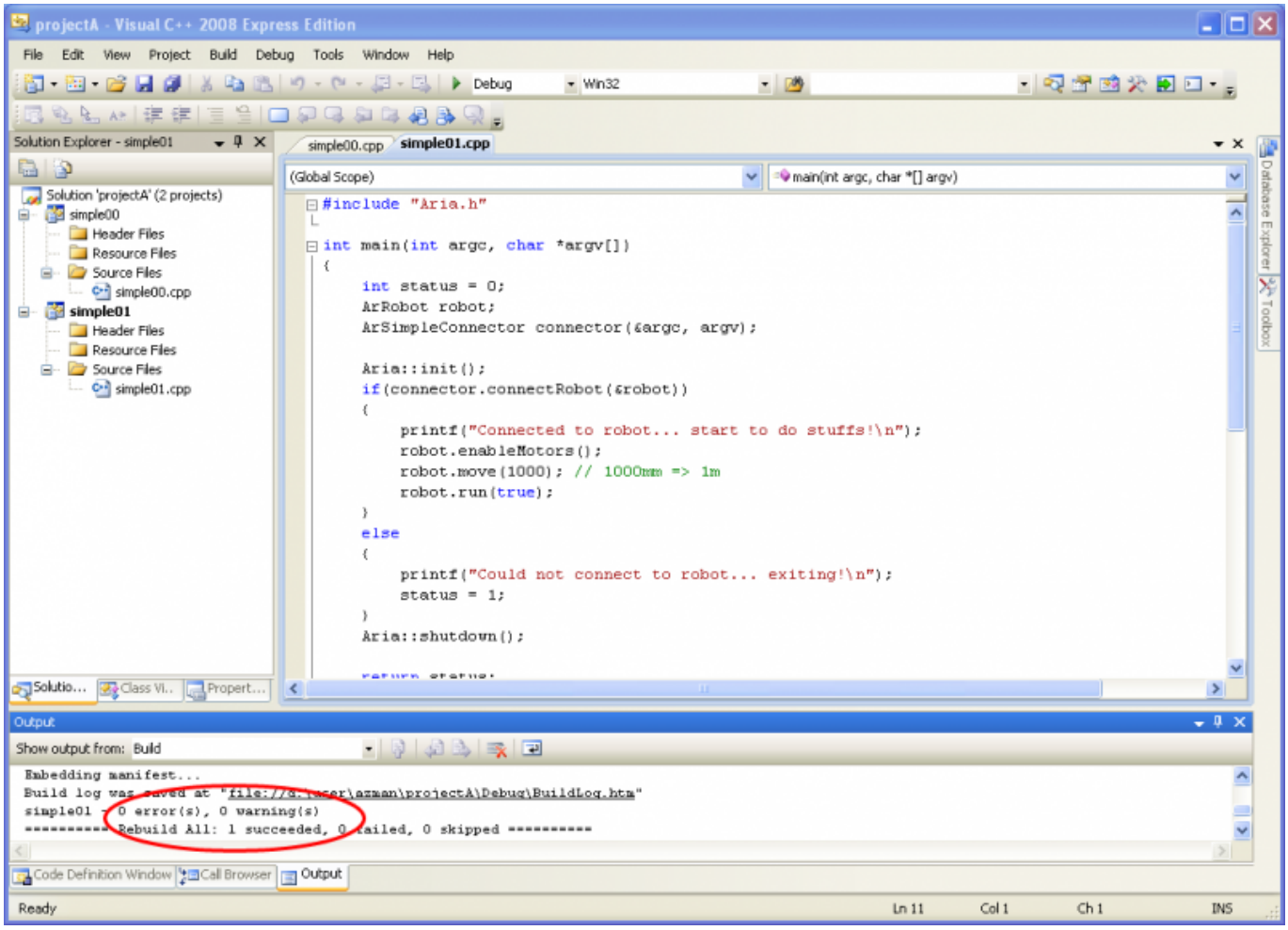
```
#include "Aria.h"

int main(int argc, char *argv[])
{
    int status = 0;
    ArRobot robot;
    ArSimpleConnector connector(&argc, argv);

    Aria::init();
    if(connector.connectRobot(&robot))
    {
        printf("Connected to robot... start to do stuffs!\n");
        robot.enableMotors();
        robot.move(1000); // 1000mm => 1m
        robot.run(true);
    }
    else
    {
        printf("Could not connect to robot... exiting!\n");
        status = 1;
    }
    Aria::shutdown();

    return status;
}
```

Again, the code will compile cleanly in Visual C++ 2008 Express Edition as shown below.



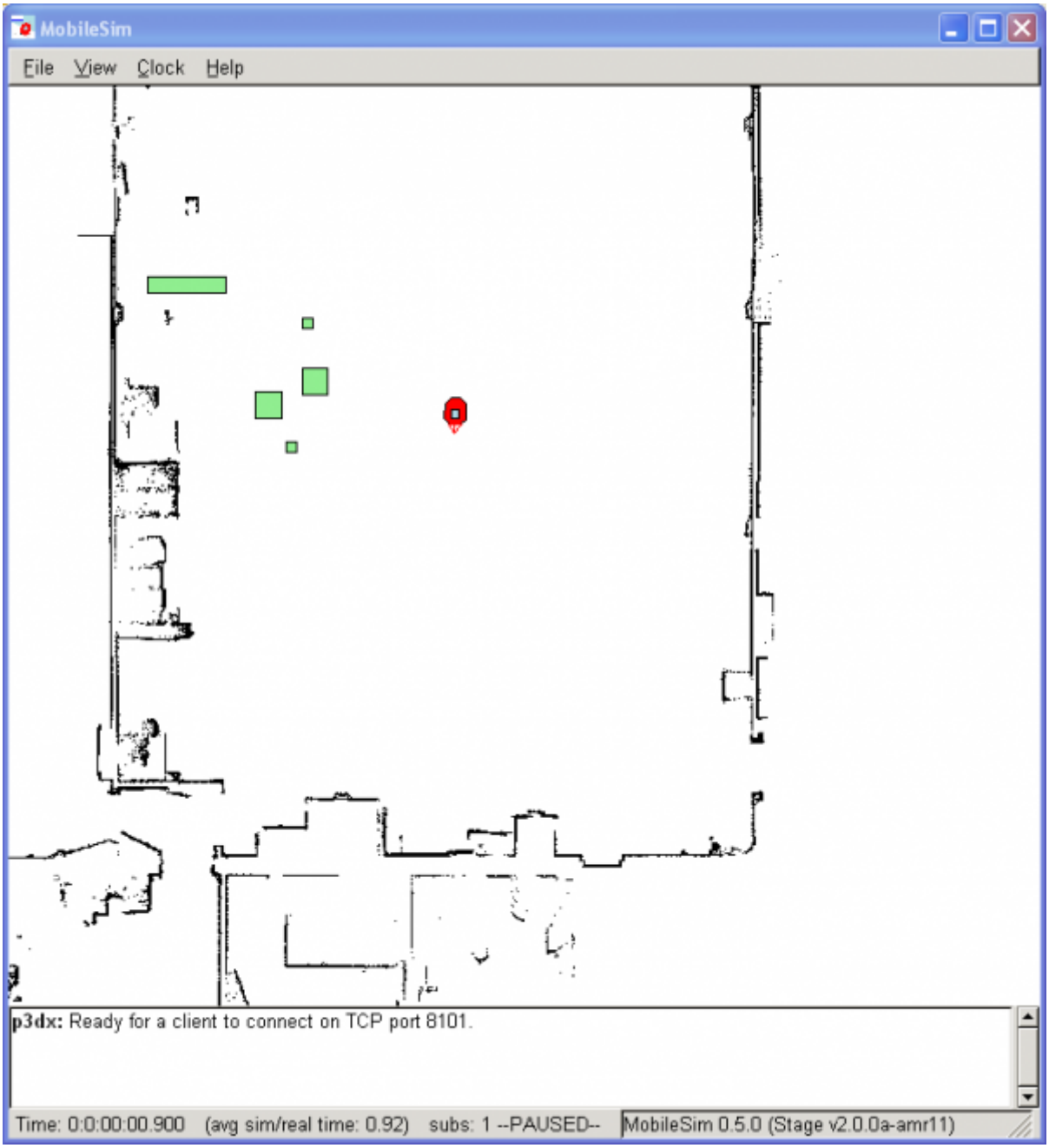

The `distance` argument for `move` method is in mm (millimeters). So, in the given code, we want to move the robot 1 meter forward. So how do we test this? Two options: (1) Run the program on a real robot, OR (2) Execute it in a simulator. Obviously, if we're testing miscellaneous things, going for the real robot is simply a hassle (that robot is quite heavy!). So, for this session we're going to go for the simulator approach.

# Introduction to MobileSim Simulator

You should have already seen this simulator in action (previous module?). We will be using the current (at the time this is written) latest version which is 0.5.0.

## Environment Setup

For this exercise, we're going to use the AMROffice.map file that is available with MobileSim installation (obviously, that file contains the map for the environment we want the robot to be in). The default starting position is in a small space. Drag the robot to the wide open space north of the default starting position. So our starting position would be something like as shown in the figure below.




You'll see that the simulator waits for a client robot to connect to it using TCP port 8101 (this is the default).

***Note**: In the simulator, notice later that the sonar is ALWAYS on when client is connected (even when the code doesn't request for it)? It could be the simulator default or a library thingy!*

## Executing Sample Program

There is a possibility that you'll have a run-time error when executing your program (i.e. double-clicking it). This is simply because ARIA library expects the run-time library files (DLLs) to be either in the system folder OR the current folder. Since cluttering the system folder (e.g. C:\WINDOWS\system32) is really NOT advisable, we can always copy the needed file(s) into the same folder as our executable. The files needed are (available in `bin` folder of ARIA installation path):

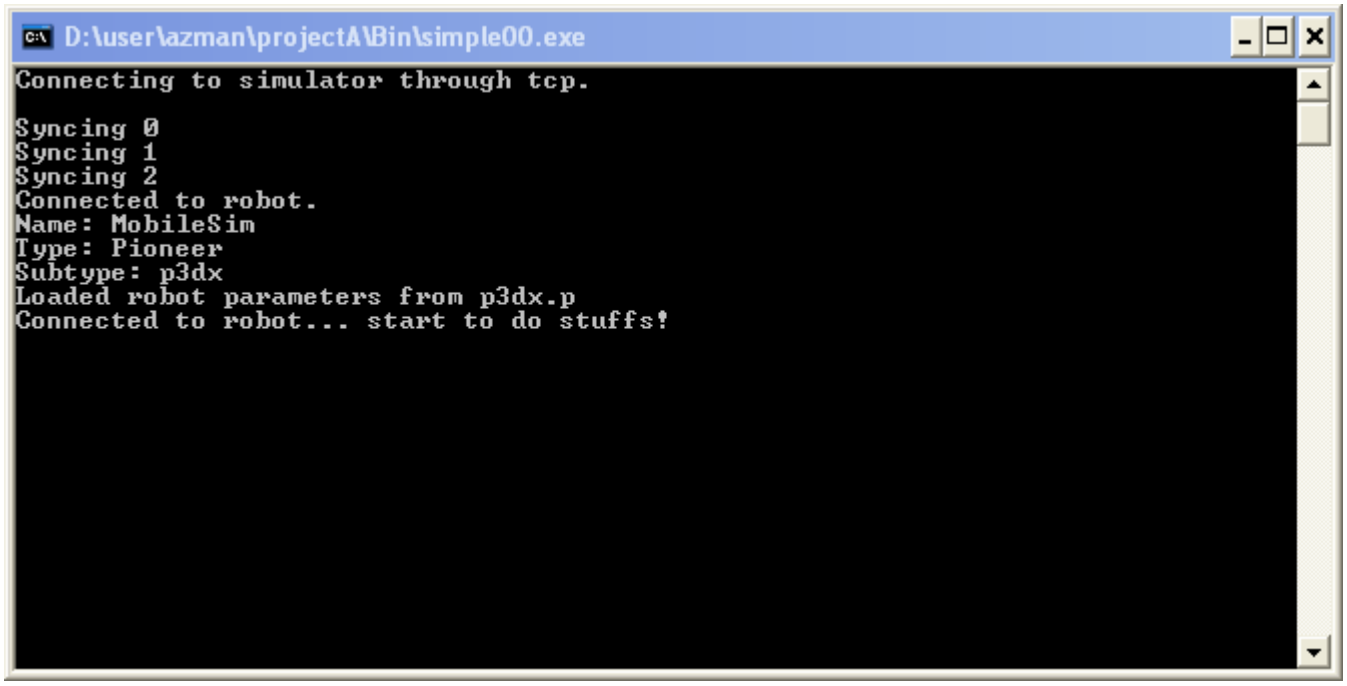
- **ARIA.dll (optionally: ARIADebug.dll)** - ARIA's run-time library

- **msvcp71.dll (optionally: msvcp71d.dll)** - Microsoft's C++ Run-time Library
- **msvcr71.dll (optionally: msvcr71d.dll)** - Microsoft's C Run-time Library

And yes, you NEED that C Run-time library!

***Note**: Using the given setup earlier, you executable should be named `simple00.exe` in the `Bin` folder of the solution. Don't forget to copy the run-time library files into this folder.*

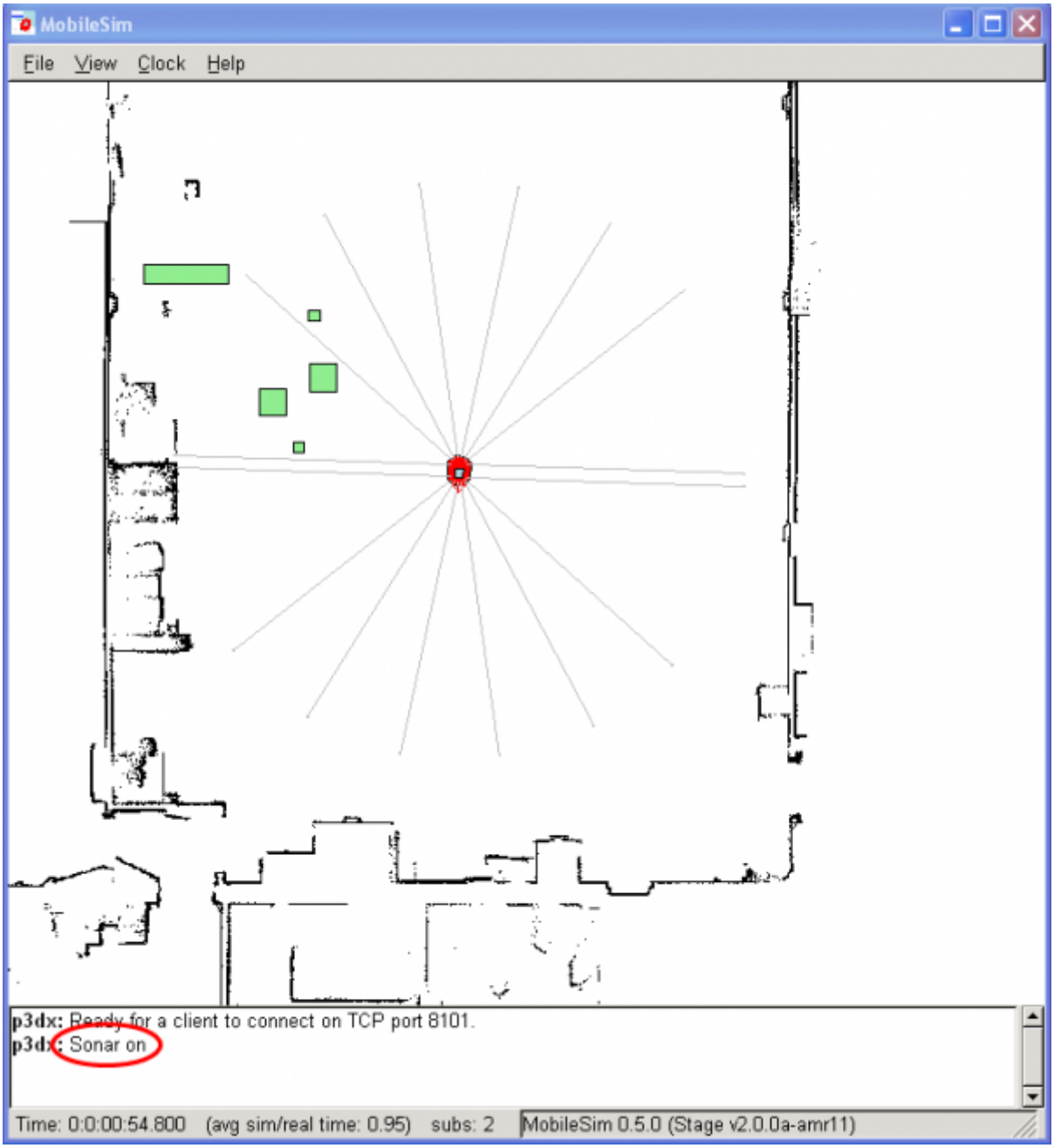
Now, if you run our program (`simple00.exe`) again you'll see something like this:




Please don't be alarmed if you're not familiar with this... this, is life before Windows95 😎 Our program is essentially a console program (i.e. one that does not have a GUI). We're supposed to look at the 'results' at the actual robot or simulator.

***Note**: If you see a warning about ArTime at the start of the program, you can safely ignore it and it is only visible when you use `ARIADebug.lib`. Use `ARIA.lib` instead if you're not comfortable with verbose information.*

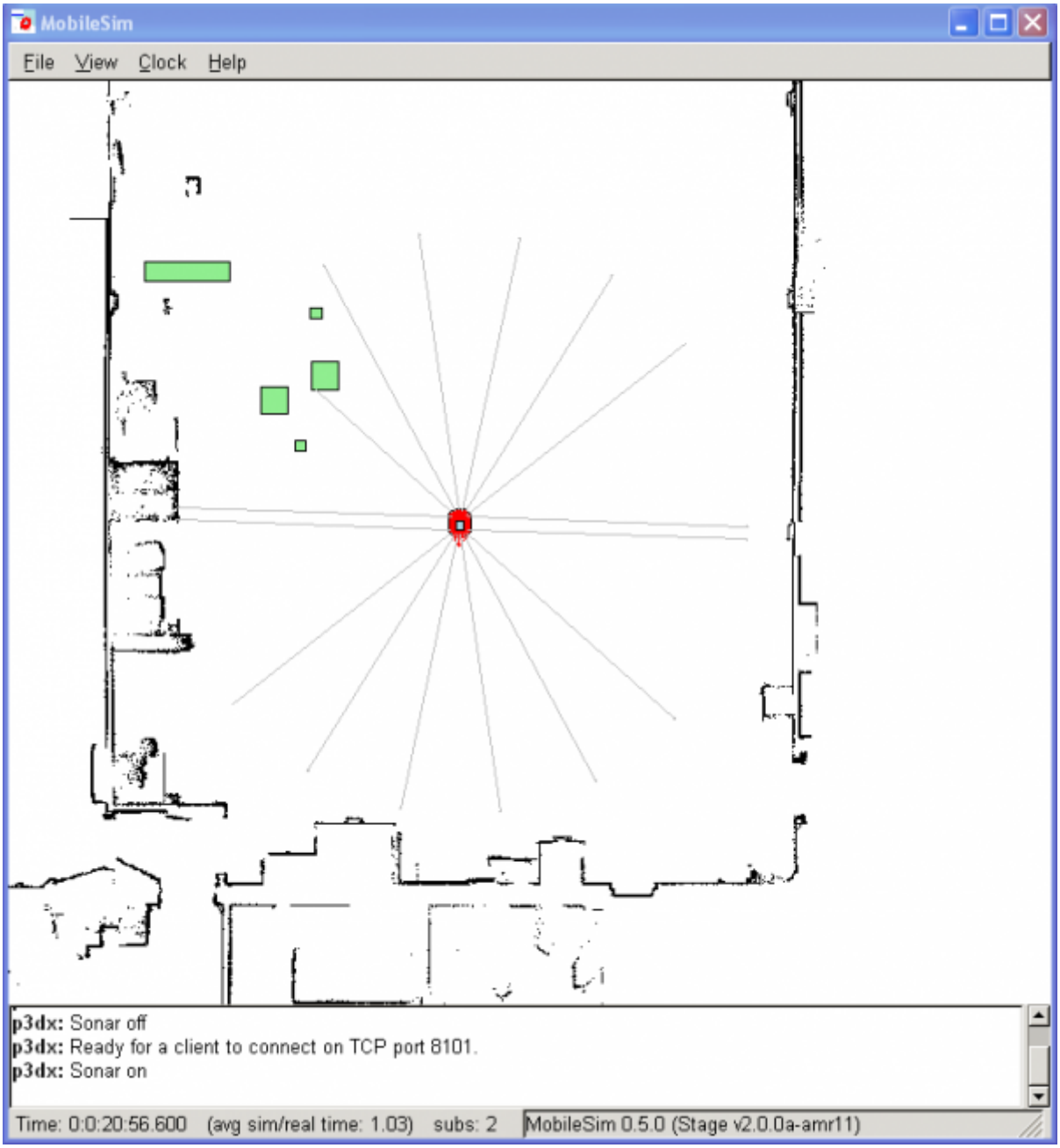
If you switch back to the simulator, you'll see that the robot is still where it was (doing nothing) and, as mentioned earlier, the sonar is ON (indicated by lines coming out of the robot). The console window at the bottom confirms that the sonar is ON. Refer to the figure below.




Let's try the other program that moves the robot. Before that, you'll notice that the program that we run is still 'executing'. To terminate it, either press <control+C> or close the console by clicking the usual cross (i.e. 'X' @ top-right). We will discuss how we can assign a key (like the ESC key) to exit from the program later.

If you execute the next program (`simple01.exe`) you'll get the exact same console window, but the robot will now move for about 1 meter before stopping and doing what it does best (at the moment)... nothing 😬.

**Hint**: If you cannot see the movement, pause the simulation (from menu or key 'P') before executing the program. After execution, then go back and resume simulation.




So, theoretically, you can start writing a program with any robot navigation algorithm that you know and try it here in this simulator. Obviously, your sensors/input are 'limited' (but Pioneer has a lot!) to what you have. You can, of course, create generic sensor devices but then it will not be 'practical'.

# Making It Better

This section adds other features that can make the simple program even better. We will only discuss the codes here, try it out to get a hang of what each code does.

## Using Key Handlers

As mentioned earlier, there is no exit method from program execution. One way to overcome that is

to use key handlers. Check out the code below.

keyhandler.cpp

```
#include "Aria.h"

int main(int argc, char *argv[])
{
    int status = 0;
    ArRobot robot;
    ArSimpleConnector connector(&argc, argv);
    ArKeyHandler keyHandler;

    Aria::init();
    Aria::setKeyHandler(&keyHandler);
    robot.attachKeyHandler(&keyHandler);
    if(connector.connectRobot(&robot))
    {
        printf("Connected to robot... start to do stuffs!\n");
        robot.enableMotors();
        robot.move(1000); // 1000mm => 1m
        robot.run(true);
    }
    else
    {
        printf("Could not connect to robot... exiting!\n");
        status = 1;
    }
    Aria::shutdown();

    return status;
}
```

Notice that we just need to add a key handler object (ArKeyHandler) and attach/set it to both the ARIA framework (`Aria::setKeyHandler`) and the robot (`robot.attachKeyHandler`). With this, you can now press the <ESC> key to exit gracefully from the program.

***Note**: One might ask what's the difference compared to using <ctrl-C> key. Well, the <ctrl-C> key is actually a process kill command - so, in the example, the shutdown procedure will not be executed. This is usually not critical in simulators and controlled environment, but in a robust and practical environment, it's always best to cleanly exit a system.*

## Using Pre-defined Actions

In this example, instead of giving explicit command(s) to the robot, we can actually assign specific 'action(s)' that, in turn, issue commands implicitly. The name of the action(s) represents what it needs to do. So, study them and try it out in the simulator. This example shows constant velocity action.

actionmove.cpp

```
#include "Aria.h"

int main(int argc, char *argv[])
{
    int status = 0;
    ArRobot robot;
    ArSimpleConnector connector(&argc, argv);
    ArKeyHandler keyHandler;
    ArActionConstantVelocity constantVelocity("Constant Velocity",
400); // default velocity

    Aria::init();
    Aria::setKeyHandler(&keyHandler);
    robot.attachKeyHandler(&keyHandler);
    if(connector.connectRobot(&robot))
    {
        printf("Connected to robot... start to do stuffs!\n");
        robot.enableMotors();
        robot.addAction(&constantVelocity, 25); // move at constant
velocity, priority 25
        robot.run(true);
    }
    else
    {
        printf("Could not connect to robot... exiting!\n");
        status = 1;
    }
    Aria::shutdown();

    return status;
}
```

***Hint**: You can also define custom actions! Multiple actions are managed based on priorities. Theoretically, you can define a new action handling mechanism in order to manage them differently.*

## The 'Wander' Program

This is actually the same code (similar... because slightly modified but with the same flow) as the one provided in the examples that come with ARIA SDK installation. This introduces more pre-defined actions. This code will essentially make the robot move around its environment, avoiding any obstacle that it comes into.

wander.cpp

```
#include "Aria.h"
```

```
int main(int argc, char *argv[])
{
    int status = 0;
    ArRobot robot;
    ArSimpleConnector connector(&argc, argv);
    ArKeyHandler keyHandler;
    ArSonarDevice sonar;
    ArActionStallRecover recover;
    ArActionBumpers bumpers;
    ArActionAvoidFront avoidFrontNear("Avoid Front Near", 225, 0);
    ArActionAvoidFront avoidFrontFar("Avoid Front Far"); //
450(distance), 200(velocity)
    ArActionConstantVelocity constantVelocity("Constant Velocity",
400); // default velocity

    Aria::init();
    Aria::setKeyHandler(&keyHandler);
    robot.attachKeyHandler(&keyHandler);
    robot.addRangeDevice(&sonar);
    if(connector.connectRobot(&robot))
    {
        printf("Connected to robot... start to do stuffs!\n");
        robot.enableMotors(); //robot.comInt(ArCommands::ENABLE, 1);
        robot.addAction(&recover, 100); // highest priority!
        robot.addAction(&bumpers, 75);
        robot.addAction(&avoidFrontNear, 50);
        robot.addAction(&avoidFrontFar, 49);
        robot.addAction(&constantVelocity, 25); // move if nothing
else...
        robot.run(true);
    }
    else
    {
        printf("Could not connect to robot... exiting!\n");
        status = 1;
    }
    Aria::shutdown();

    return status;
}
```

I 'wonder' if we need to do more...

*kaz20100808*