Ecole polytechnique fédérale de Lausanne (EPFL)

Cyberbotics

Master Thesis Project Report

Interfacing an Android-based phone-robot with Webots

Author:
Loïc Frund

Supervisor:
Dr. Olivier Michel
EPFL Supervisor:
Dr. Francesco Mondada

August 17, 2012

Abstract

The development of smartphones open new perspectives for low cost mobile robots. A smartphone has a lot of useful things for a robot: Wi-Fi, cameras, accelerometer, etc. It is relatively cheap and has a powerful CPU. In this project an Android powered device is used as a robot brain. It is connected to a robot platform with wheels and distance sensors. An Android application has been developed to interface the phone-robot with the Webots robots simulation software. Robot controllers can be programmed and simulated in Webots before transferring to the real robot or using the remote control library to control the robot from the PC. The remote control system of Webots has been redesigned to support the phoneBot. The phone has a good CPU power but is not a real time operating system and other processes execution slow down robot controllers and induce timing variability. Nevertheless the system still remains fast enough for its teaching purposes.

Contents

1 Introduction

Cyberbotics[1] is the company developing the Webots software, a development environment used to model, program and simulate mobile robots. They are also reseller of the e-puck[2], a mini mobile robot developed at EPFL for teaching purposes. This robot can be simulated, programmed and controlled remotely by the Webots software.

As a possible alternative to the e-puck, Cyberbotics is developing a phone robot in collaboration with GCtronic[3] (manufacturer of the e-puck). This phone-robot is made up of a basic robotics platform connected to an Android mobile phone. Smartphones are relatively cheap and very powerful devices with various sensors which make them very suitable to be used for mobile robots.

Users will be able to write and simulate robot controller programs with Webots before transferring them to the phoneBot. The robot can be controlled either remotely by Webots or robot controllers can be uploaded and run on the phone.

The phoneBot development had just started and nothing was already done. The main task was to create the necessary software in order to use this phone-robot with Webots while, in the meantime, GCtronic was working on the robot platform prototype.

1.1 Objectives

- To implement a programming interface compatible with the Webots API to access all the devices of the phone-robot.
- To implement a remote-control interface that can be operated from the Webots software.
- To implement a cross-compilation interface that can be operated from the Webots software.
- To implement a simulation model of the phone-robot in the Webots software.
- To implement a couple of demonstrations showing the capabilities of the system.
- To write the corresponding documentation.

[1] `http://www.cyberbotics.com/`
[2] `http://www.e-puck.org/`
[3] `http://www.gctronic.com`

2 Background

2.1 Robot

The robot is made of two parts, an Android phone and a mobile robot platform.

2.1.1 Mobile platform

GcTronic is currently developing the mobile robot part. It is has two motors, four infra red proximity sensors at the front to detect obstacles, and four other ones under the robot to detect the ground. The phone is connected with a micro USB cable. A micro usb port is used to charge the robot batteries and the phone.

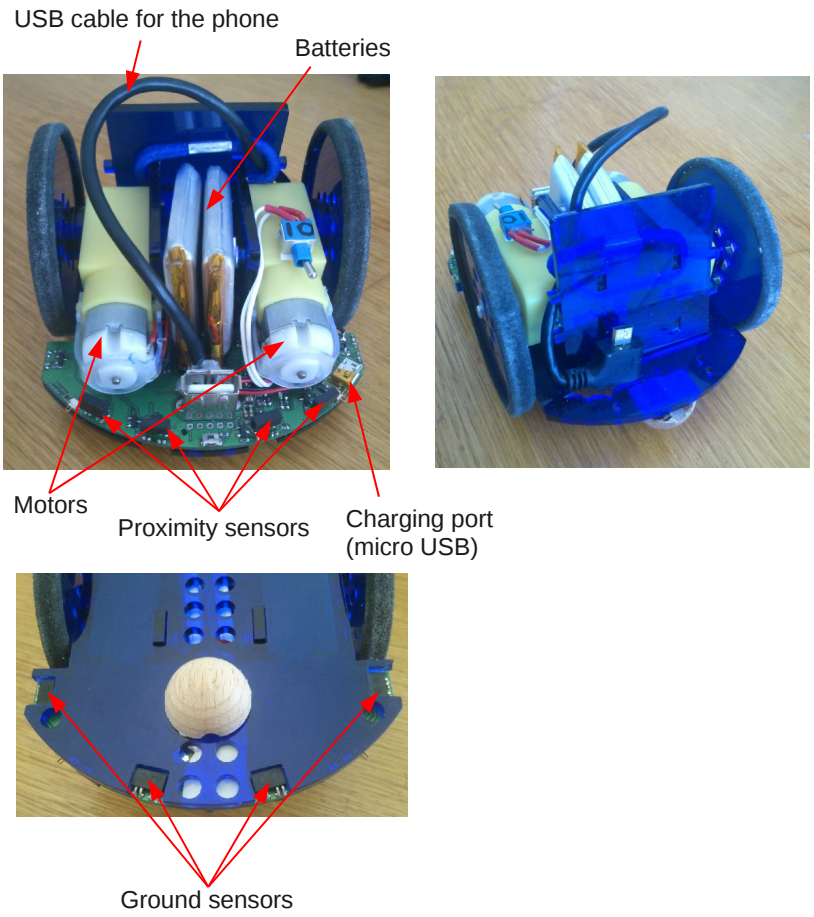


Figure 1: Current prototype (without phone)
Front, back, and bottom view

2.1.2 Phone

After having reviewed available Android phones, we have chosen the Sony Ericsson Xperia mini based on its low price, small size and good capabilities. It has a 1 GHz armv7 architecture processor. Armv7 processor can have the NEON extension, a single instruction multiple data (SIMD) instruction set. This phone has an accelerometer, a magnetic field sensor (compass), a gps, one camera, and of course a speaker and a microphone. There is also a proximity sensor used to detect when the phone is held near the ear to prevent unwanted interaction with the touch screen. But the sensor is not useful for the robot because of its position and since it only provides binary output (near or far). This phone is also supposed to have an ambient light sensor but, for unknown reasons, it is not visible to the programmer.

Operating system	Google Android 2.3 (Gingerbread)
Processor	1 GHz Qualcomm Snapdragon QSD8255
Size	88 x 52 x 16 mm
Weight	99 grams
Screen resolution	320x480 pixels
Internal phone storage	1 GB (up to 320 MB user free memory)
RAM	512 MB
Expansion slot	microSD, up to 32 GB
Camera resolution	5 megapixel with 8x digital zoom
Battery	1200 mAh

Table 1: Phone details

2.2 Webots

Webots is a development environment used to model, program and simulate mobile robots. It provides a 3D window that allows to see and edit the simulation world. Robots can be easily built by the use of basic building blocks. Webots can simulate various devices such as distance sensors, accelerometers, servo motors, etc.

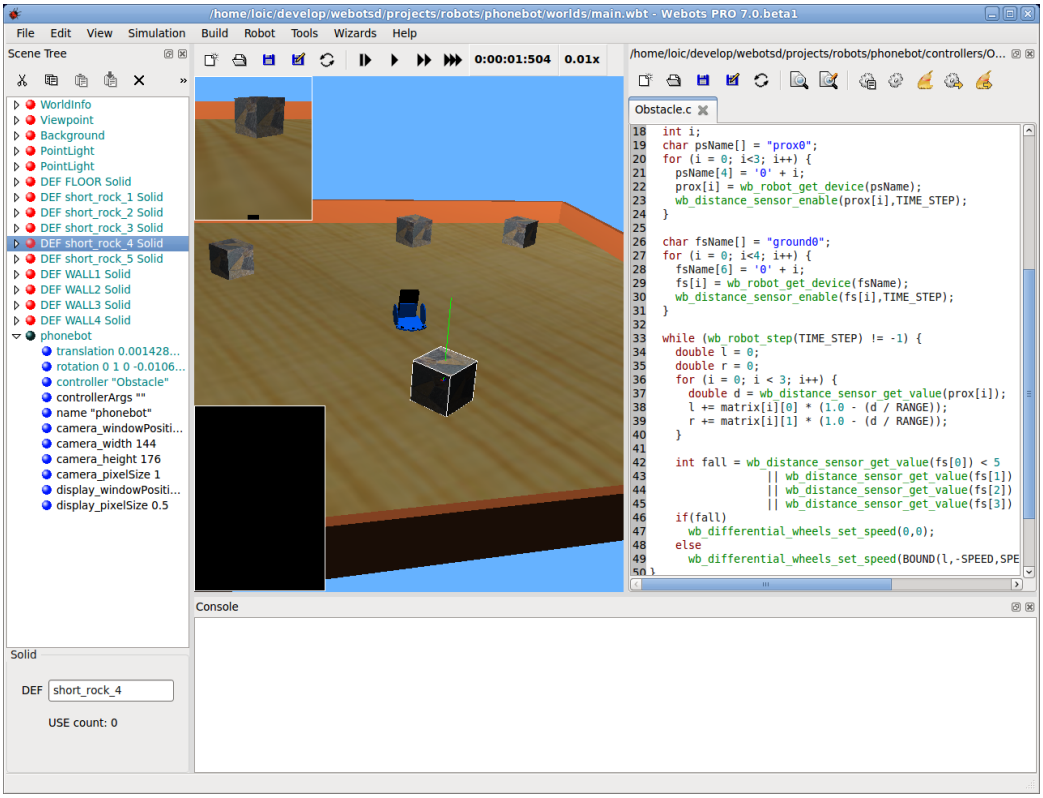


Figure 2: Webots window

Robots are controlled by a `controller` program. These Webots controllers are linked with a library called `libController` which provides the necessary API to control the robot. A controller runs in its own independent process which communicates with Webots via the `libController`. Webots controllers support different programming languages. The primary language is C in which the `libController` is written. Other languages are wrapper to the C functions. A typical C controller looks like this (as it can be seen in the Webots reference manual) :

```
//includes omitted
static WbDeviceTag my_sensor, my_led;
int main(void) {
  /* initialize the webots controller library */
  wb_robot_init();
  // get device tags
  my_sensor = wb_robot_get_device("my_distance_sensor");
  my_led = wb_robot_get_device("my_led");
  /* enable sensors to read data from them */
```

```
  wb_distance_sensor_enable(my_sensor, TIME_STEP);

  /* main control loop: perform simulation steps of
     TIME_STEP milliseconds and leave the loop when
     the simulation is over */
  while (wb_robot_step(TIME_STEP) != -1) {
    /* Read and process sensor data */
    double val = wb_distance_sensor_get_value(my_sensor);
    /* Ask to send device commands */
    wb_led_set(my_led, 1);
  }
  /* cleanup the webots controller library  */
  wb_robot_cleanup();
  return 0;
}
```

The key point is the step function that communicates with Webots to advance the simulation of TIME_STEP millisecond. At each step the controller library sends the devices commands to Webots and waits the answer that will contain the updated sensors values if available. Consequently, when the controller gets a sensor value, it obtains the value simulated during the last call to the robot step function. Devices commands are stored in an internal device state before being sent to Webots at the next robot step.

To save computation time and power consumption (on real robot) sensors must be enabled before being used. The enable function of a device takes a refresh rate parameter that tells how often the device need to be updated. Therefore sensors are not always updated at each step.

In addition to simulation, Webots can also control real robots. A remote control mode allows controllers running on the PC to communicate with a real robot instead of Webots. Controllers can also be uploaded to the robot and run directly on it.

2.3 Android

Android[4] is an open-source operating system for mobile devices. It is based on the Linux kernel. Applications are written in Java, they are compiled as any Java program and then packaged by the Android SDK tools along with any data and resource files.

Applications run in the `dalvik` virtual machine which is register based instead of stack based as usual Java VM. Each application lives in its own

[4] `http://www.android.com/`

security sandbox and everything is abstracted by the android application framework. It permits applications to be programmed without considering low level issues. This eases a lot the use of the phone devices but has the disadvantage of restring the capabilities to what the android API permits to do.

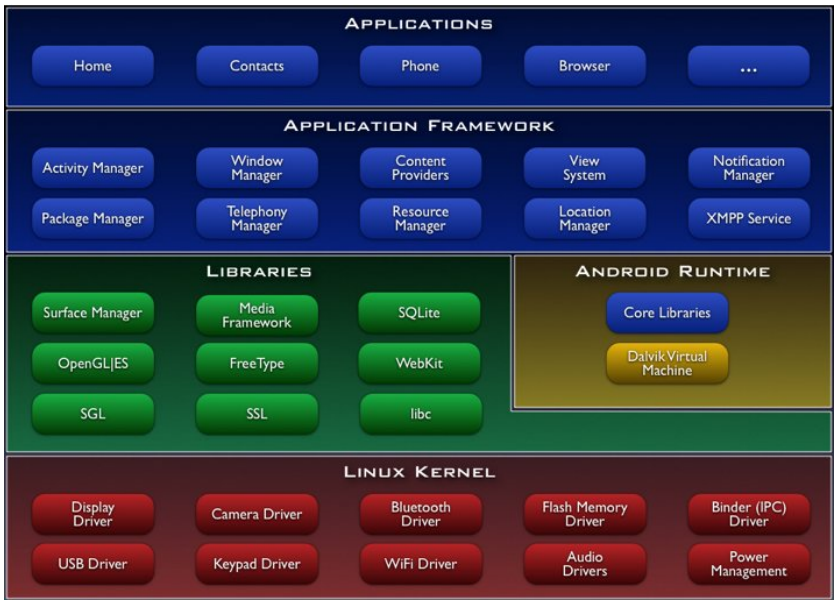


Figure 3: Android system-architecture

http://en.wikipedia.org/wiki/File:System-architecture.jpg

The main component of an Android application is an `Activity`. It represents a single screen with a user interface. A `Service` is a component for background task without user interface. Applications can have several activities and other components.

Android applications have a single thread design and are events oriented. When an application is launched, the main thread (sometimes called UI thread because it is the only way to interact with UI component) is created. Each time something happens, an event is generated and a corresponding callback is later called by the main thread. It is of course possible to create other threads but Andoid UI toolkit must only by used from the main thread. If another thread need to interact with the UI, it has to create and dispatch and event for the main thread.

2.3.1 The real time problem

Using Android has all the advantages of a full operating system and therefore eases a lot the programming of the Webots interface. However it lacks the possibility to run real time processes. Linux can have real time processes that are scheduled in a special manner but this cannot be used by android applications. Even increasing the priority level has not shown any changes. Therefore there is absolutely no guarantee of constant time execution, which means that the duration of a controller robot step is variable. If a robot step is shorter than requested, a sleep is used to wait the remaining time. A sleep can last longer than requested because it depends on when the process is given back the CPU. The process can also be preempted at any time to let another process be executed, increasing the duration of the step. If there is heavy computation during a step, the likelihood of preemption increases because the process never releases the CPU with a sleep.

Let's take a example with a 32 ms robot step and measure the actual robot step duration during 1000 steps. On the left graph there is no computation, thus the sleep is used to maintain a 32 ms duration. The step is most often a bit longer because of the sleep that lasts too long. The variance is low but there is some rare cases where a step last significantly longer because some other processes have used the CPU. On the right graph there is a heavy computation that need more than 32 ms to be achieved. In this case the controller is always requiring the CPU and the controller process is preempted to allow other processes to have the CPU. That's why the variance is larger.

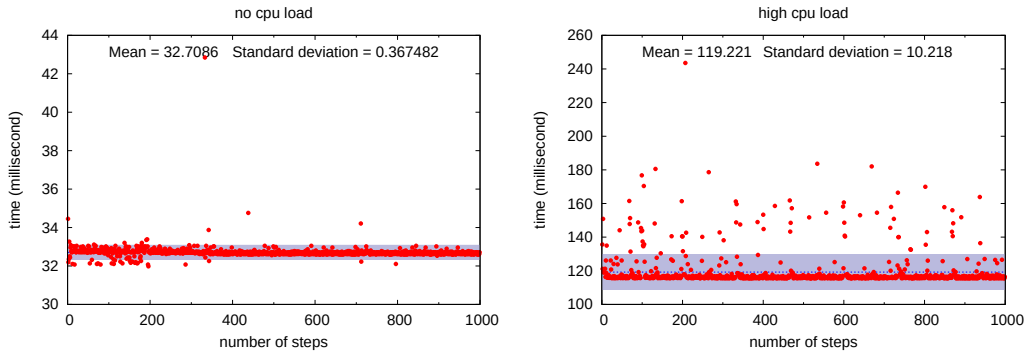


Figure 4: Timing variation

Another problem is that manufacturers install several useless applications that autostart at boot and cannot be uninstalled (e.g a facebook application). Some of them cannot be stopped and even if they can it would be required to stop them each time the phone is rebooted. The cost of that is probably

low but cumulated it can increase the variability of the robot step duration.

2.3.2 USB communication

The phone is connected to the mobile platform by USB[5]. It has the advantage to be available on most phone and is much faster than bluetooth. However the USB on the phone was primarily designed for programing and debugging, and accessing the data on the phone like a data storage device. But for the robot we need to control the USB to send specific data packets. The design architecture of USB is asymmetrical, consisting of a host connected to devices. It's always the host that initiates a transaction, then the device responds.

The first alternative is to used the phone as the USB Host. This is possible with this phone with the OTG[6] feature that can switch the phone to host mode. It requires a special cable which have its 4th and 5th(ground) pins short-circuited, in order to allows the phone to detect OTG mode and switch to host mode. It allows for example to plug a mouse on the phone. However the software support to plug a custom device is only from Android version 3 and therefore would have required an upgrade to a version than was not yet available. A solution would have been rooting the phone and compiling a kernel with the necessary options enabled. Like all smartphones, android phone are locked to prevent users for modifying the system. It is not possible to login as root user or install another operating system. However it is possible to overcome this limitation by a procedure call rooting. Support for rooting depends from each manufacturer : on some phone the only way is to exploit a security hole, on other, like Sony Ericson, it is supported by the manufacturer who provides the procedure. The drawback is that it generally voids the warranty, and require to apply a rather complicated procedure on each phone. Since Linux kernel is on GPL[7] the source of the kernel installed on the phone is available and can be recompiled with the required option. But doing that is risky since there no way to know in advance if it will work. And if the phone need to be changed (e.g if it is no longer sold), then everything should be redone since it will probably be different on another phone. Rooting the phone should therefore be avoided.

So the solution adopted conjointly with GCtronic was to put the host controller on the robot chip and use the phone as a UBS device. Starting from android 3.1 and backported to 2.3.4, there is a support for Android Open Accessory Protocol[8]. This protocol allows external USB device to

[5] `http://en.wikipedia.org/wiki/USB`

[6] `http://en.wikipedia.org/wiki/USB_On-The-Go`

[7] `http://en.wikipedia.org/wiki/GPL`

[8] `http://developer.android.com/tools/adk/aoa.html`

interact with an Android-powered device in a special accessory mode. The only problem is that it is not mandatory for a 2.3.4 phone to support this mode and it is not possible to find detailed enough phone specification to know before buying the phone if this feature is supported. Fortunately it was supported by this phone.

The communication protocol is very simple. The phone send an update packet with the speed of the left and right motors. Then the robot platform answers with the distance sensors value and battery level.

The communication is fast and take about 1.5 ms. As shows this graph it is relatively stable, but can sometime experience some lag.

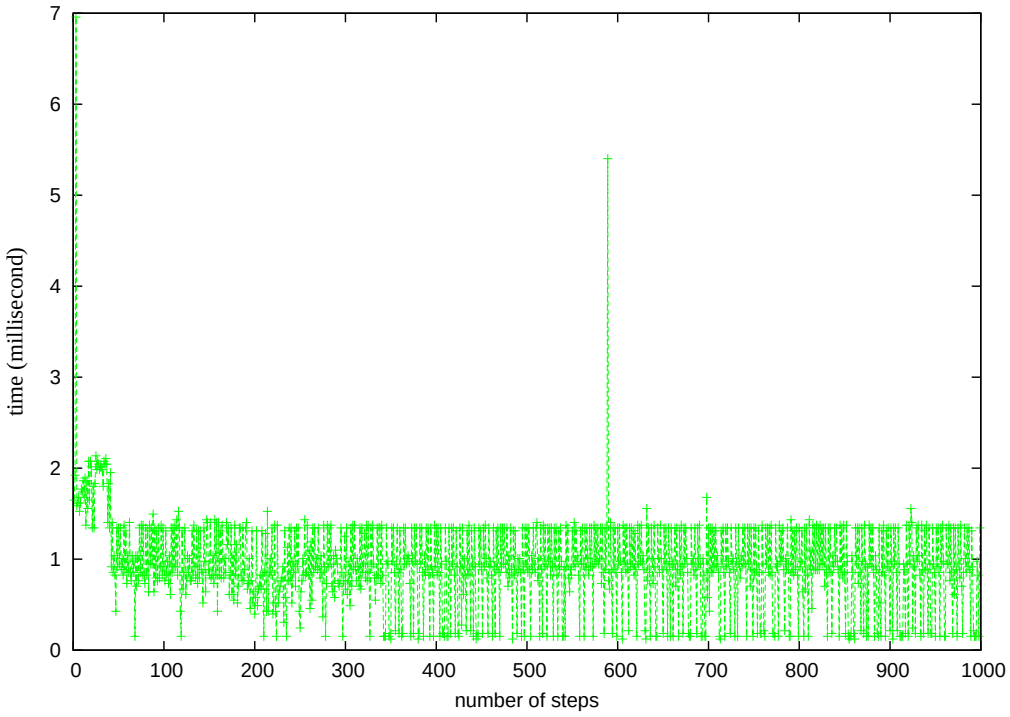


Figure 5: USB communication time

3 Implementation

3.1 Application overview

The Adroid application has the following main features :

- A user interface to choose and start robot controllers
- A preference user interface to set parameters like the camera resolution

- A Webots Java API implementation to give Webots controller access to all the robot devices
- C/C++ and Python Webots API wrapper
- A USB communication channel to control the mobile platform
- A TCP communication channel to allow the PC to upload Webots controllers and start a remote control session
- A remote controller that can receive command trough TCP.

The application is divided in two processes : a main one that is the starting point of the application, and a controller one in which the robot controller will run :

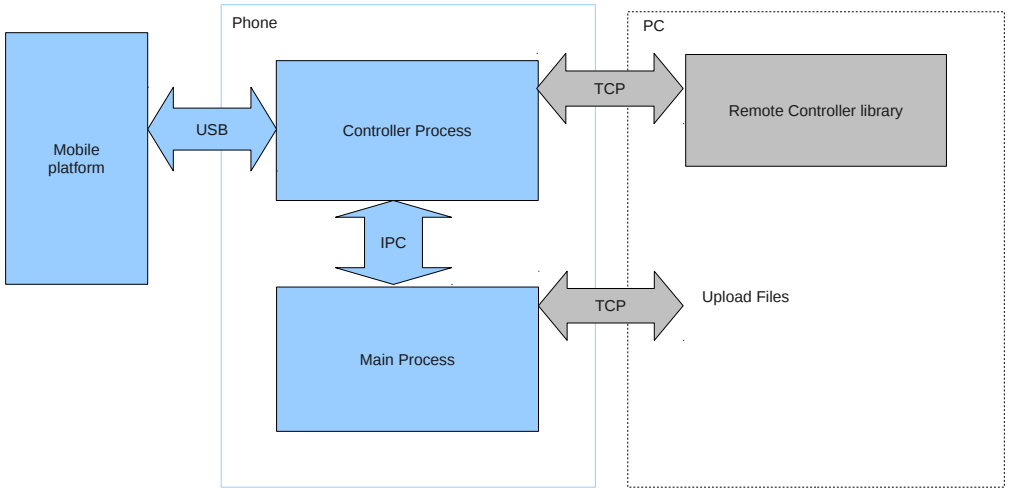


Figure 6: Application overview

The two processes use inter process communication (IPC). The PC can connect with the robot through TCP connections. One is on the main process and is used to upload files and start the remote control session. The remote control one is directly on the controller process to avoid passing through the main process and IPC. This scheme clearly separates the remote control protocol and the robot management protocol.

The reason for the two processes is because the application will run programs provided by the user which can crash or hang. With Java controllers crash can be prevented by exception handling but, since C/C++ controller are also possible, segfaults or memory corruptions and leaks could completely

crash the process. Having a dedicated process for the controller avoid crashing and corrupting the whole application in case of problem with a controller. It also allows to ensure complete reset when launching another controller. It is the same principle as in Webots where controllers are also run in their own processes.

In android it is not possible to directly start a process but an activity (or service) can be set to run in a different process. Activities cannot communicate with IPC except to return a result value on exit. To have better flexibility a Service is used. This component is bound by the main activity and can exchange messages with it.

The following graph shows the relation between the components of the application :

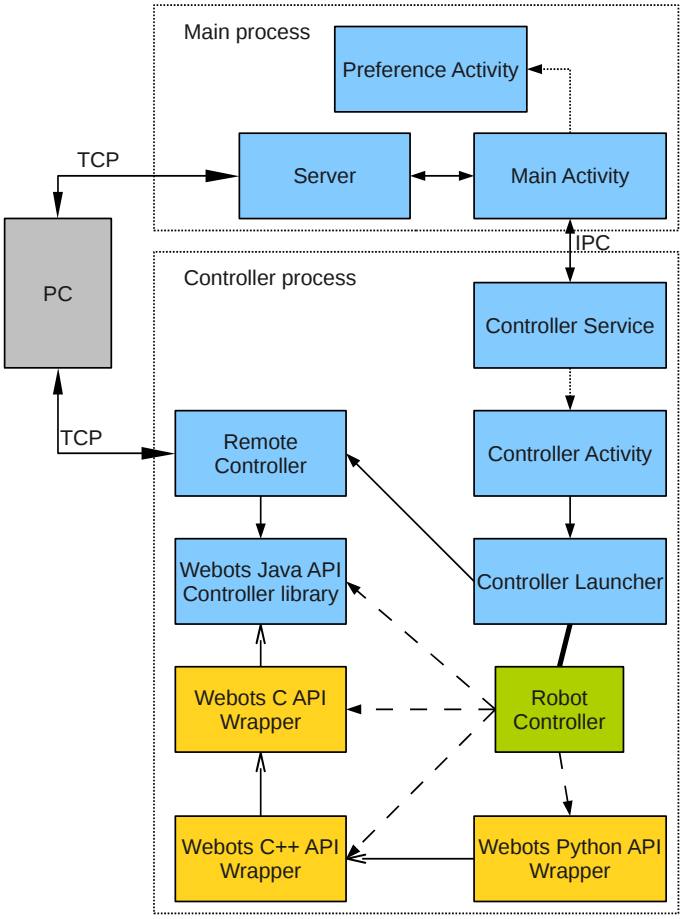


Figure 7: Application graph

On the main process the main activity is started first. It binds the service, which is started in another process by the android framework. The main activity also starts the TCP server. The Preference activity can be started by the user to show the settings screen.

When the user chooses a controller from the main activity, it sends the controller name to the service which in turn starts the controller activity. This activity is used to show another screen that will hold the Webots display device, touchscreen sensors, and also provide a menu to start and stop the controller.

When the robot controller is actually started by the menu of the controller activity, the launcher loads the necessary libraries and starts the controller in the appropriate manner depending of its programming language. Java Webots controller directly call the real API implementation while other ones are linked with their corresponding wrapper API that eventually map the the Java implementation.

If the remote mode is started then, instead of loading a robot controller, the launcher starts the remote controller that open a TCP socket to receive and send remote control commands.

3.2 Devices

In Android almost everything works with events and callbacks, therefore it is not possible to control exactly when a sensor gets a new value. To comply with the Webots model, each time a device callback gives a new value it is stored internally. The values visible to the robot controller are only updated during a robot step function and according to the device refresh rate.

3.2.1 Accelerometer

When registering the callback, the refresh delay can be chosen among four predefined values. However the behavior is not what is expected. Instead of having an update each requested millisecond, updates seem to be done only if a move is detected. If the phone is moved slowly the callback is never called and no new values are read. The problem doesn't occur with a high refresh rate. To circumvent the problem the accelerometer is internally activated with a high refresh rate, even if the user won't see updates faster than requested.

3.2.2 Compass

The compass has also the same update problem as the accelerometer but there is also another problem that prevents any update if the accelerometer is not

enabled. Usually the accelerometer values are required for the compass device to compute the phone orientation and convert the magnetic field measurement to a north vector. But obviously it only works if the phone does not accelerate. To avoid this problem the accelerometer values are not used, instead, the orientation is computed with a predefined angle. It is assumed that the robot is generally on a horizontal plane. Otherwise, if needed, it can be set in the preferences to use the accelerometer to compute the orientation but it will give less precise results.

But even if the accelerometer is not used, it still doesn't work if the accelerometer is not enable. To circumvent the problem, the accelerometer is transparently enabled when the user enables the compass. The activation of the accelerometer is not visible to the user, and he still needs to use the Webots API enable call to use the accelerometer.

3.2.3 Camera

The camera device is the most complicated. Since the camera API is only designed to take pictures and films by the user, it is required (at least on Android 2.3) to provide a `SurfaceView` to the camera that is used to draw the preview image and display it on the screen. Because we want to use the screen for the display Webots device, it is necessary to find some trick to get the camera working without having to display the preview on the screen. Contrary to the documentation, the camera can work without a `SurfaceView` but it creates hard to understand bugs. The camera didn't work properly whether it is enabled in the UI thread or in the controller thread, which is probably due to a deeper bug. To avoid relying on obscure bugs and to respect the documentation a dummy `SurfaceView` is given to the camera. The view is set to invisible and it is not actually drawn on the screen. It still produces a warning because the size of the view is zero since it has no screen space. This code will probably need to be rewritten for newer Android version which is supposed to provide better handling for this kind of use.

In the Webots API camera images are in RGB format. Android API doesn't provide easy access to such an image. It can easily save the images as a jpeg file but cannot access the RGB image buffer itself. The only way is to get the preview image which is only available in NV21 format with this phone. This format uses the YUV[9] color space. It has the advantage to use only 12 bits per pixel instead of a 24 bits RGB. The data are divided in two plans : Y and UV. In the Y plan one byte per pixels is used to represent luminance (brightness). The colors are represented by the UV plan. U and V

[9] `http://en.wikipedia.org/wiki/YUV`

values are interleaved and the same value is used for a square of 4 pixels.

Figure 8: NV21 file format

The following formula is used to convert from YUV to RGB color space. To avoid floating point arithmetic (which is slower), the coefficients have been multiplied by 1024, so the result is shifted by 10.

$$B = Y + (2066*U)>>10$$
$$G = Y - (833*V)>>10) - (400*U)>>10$$
$$R = Y + (1643*V)>>10$$

It exists different norms for YUV with different coefficients and there is no documentation about which one is used on Android. Usually Y is on the range [16-255] and need to be rescaled to [0-255]. On this case it seems that the values of Y are already on the range [0-255] since values lesser than 16 can be observed. In any case it does not really change the result. The obtained image looks like the same as the one shown by Android in normal camera usage with a visible `SurfaceView`.

The Android API doesn't provide any methods for converting YUV images to RGB. It could be a performance overhead to do the conversion in software. Hopefully the phone we use has a CPU that has the NEON extension. NEON is the `single instruction, multiple data` (SIMD) instructions set of arm architecture. It means that an operation can be performed on multiple data (pixel on that case) in parallel. NEON has 128 bits registers allowing up to 16 operations at the same time. Register can be treated as 8, 16, 32 or 64 bits elements vector. For example the instruction `vsub.s8 q1,q2` performs a subtraction on signed 8 bits elements which means that the 128 bits registers `q1` and `q2` each contains 16 elements.

These NEON instructions can be automatically generated by compilers by a procedure called vectorization but better code can be written by hand. There are `gcc` intrinsics that permit to use the NEON instructions in C as if they were functions. But the code generated by the `gcc` version 4.4.3 provided by android is very bad. It uses extensively the memory because of a buggy register allocation. So finally it was done directly in assembly by using the `gcc` features which allows to directly use assembly code inside a C function.

Another issue is the orientation of the camera which is rotated by 90°. Rotating an image is not complicated but it means that the pixels cannot be stored consecutively by chuck which therefore induces more memory latency.

To allow compatibility with non Neon architecture a C version of the code was also implemented, and also a Java version to do speed comparison.

Measures

The time taken by the conversion function is measured. Since a lot of unpredictable factors can change the timing, 1000 samples are measured to derive an average and standard deviation. In this figure the NEON, C and Java YUV conversion functions are compared using the maximum camera resolution : 1280x720.

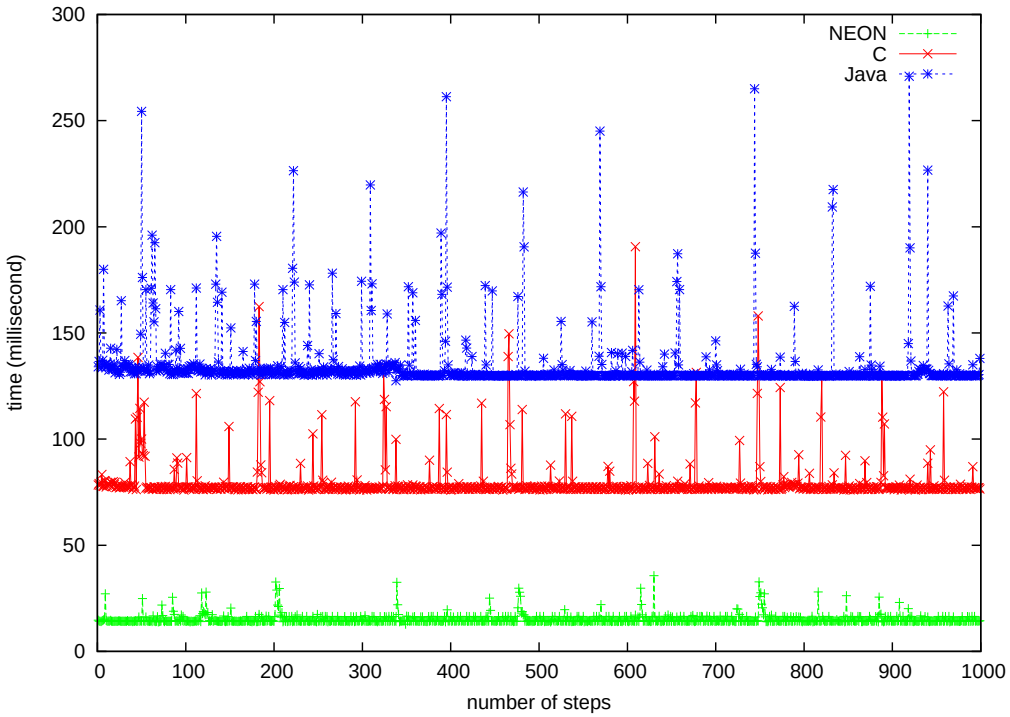


Figure 9: Comparison between convert implementation (1280x720)

As expected the Neon version is much faster than the others. It is also more stable because the faster it is the less likely it is to get another task executed during the conversion. The tables below show the time measurement for the smallest camera resolution (176x144) and the largest (1280x720) given in microsecond. There is also a comparison with the image rotation disabled to show the cost of it.

	mean	SD
Java	3'837	994
C	1'598	127
NEON	278	33
C (no rotate)	1'390	237
NEON (no rotate)	207	46

	mean	SD
Java	134'613	14'715
C	79'443	10'049
NEON	15'146	2'384
C (no rotate)	57'417	8'728
NEON (no rotate)	7'782	1'146

Table 2: Speed comparison left : 176x144, right : 1280x720

In most cases the smallest resolution will always be used because the capture and further image processing by the controller will be faster.

The camera itself needs also some time to capture the image. This time is unknown and cannot be measured directly because there is no precise control on when the camera captures an image. It depends on a lot of different factors like the luminosity and movement. It is probably due to the auto focus and other auto settings of the camera.

To comply with the Webots model and get an image when required, a lock is used to synchronized the controller thread with the camera callback thread. This will block the controller thread during the image capture. Because of this asynchronous model the image capture time is very sensitive to the scheduling. Depending of the time step used and the camera refresh rate different patterns can be observed :

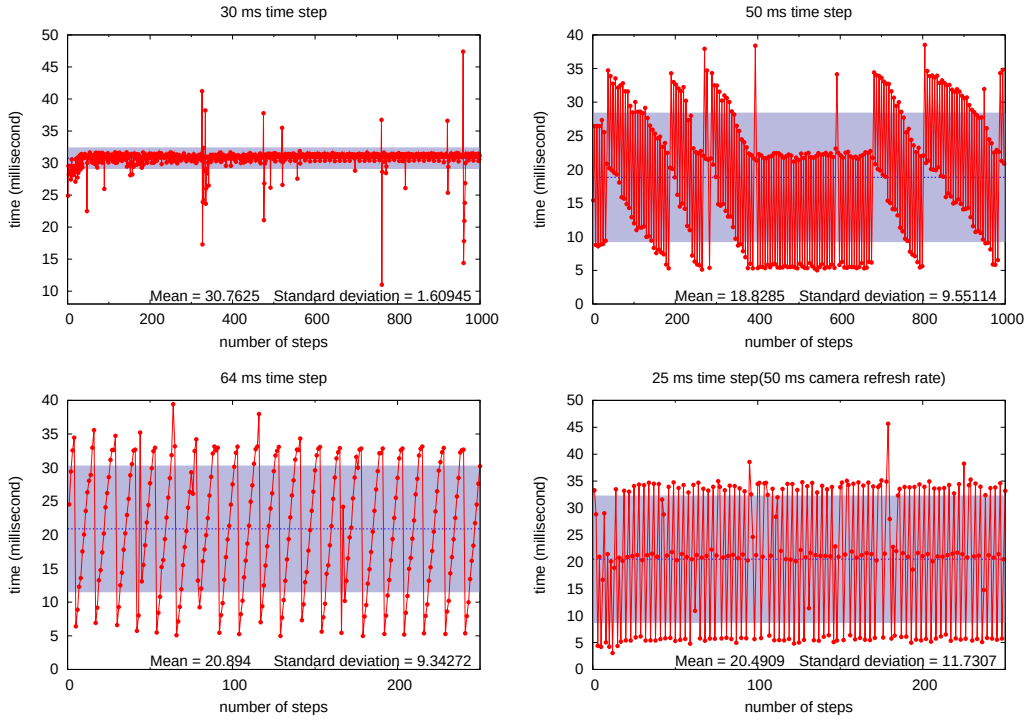


Figure 10: Camera Image capture time with different time step (176x144)

These patterns can be explained by scheduling competition between processes. Some specific time step leads to regularly having another (or several) process run before the camera thread or the controller thread are given the CPU resource. On the first graph this almost always occurs and the capture time is long. In the second graph it alternates between two states that shift over time. On the third graph the duration increases at each step until a maximum of about 35ms then fall to the minimum of about 5ms. And in the fourth graph it alternate between three values.

Since it is not possible to predict anything it is not possible to give an accurate lower boundary of the time required to capture an image. However by observing the previous measures we can find that the minimum required time to get an image is 5 ms. The usual maximum is about 35 ms but can sometime (but rarely) be higher. The average is generally between 18 and 20 ms and was never observed above 30 ms but there is no way to be sure it will never happen. Consequently it is recommended to never enable the camera with lower values than 32 ms even if the camera could be faster, and no assumption should be done about timing accuracy of the robot step when the camera is enabled.

openCV

Another alternative is to use openCV for android that has the necessary conversion utilities and camera API. This possibility was only discovered and explored afterward. The measures showed that the openCV native camera capture and conversion to RGB is about twice slower than my implementation. It is maybe because the library is not enough specialized for that use case or this Android version. But since openCV provides powerful feature for image processing, the possibility to use it from robot controller has been added.

3.2.4 Display

Webots has a display device that can be used to control the screen of the phone. Since the Android API provides all the required drawing functions, the display implementation wraps the Webots API to the underlaying Android canvas. The Webots API requires updates to be done only during the step function so the drawing is not directly done on the screen but on an internal bitmap. At each robot step the bitmap is drawn on the screen.

Webots display also allows to load and save images, as well as directly creating images with a buffer. Special care must be done with byte order with RGBA and ARGB mode. In the Java Webots API and Android API pixels are stored as integer and are therefore stored in the machine byte order. However the C Webots API consider pixels as byte array and they are therefore stored in the actual order. The C wrapper must therefore swap the byte before calling the corresponding Java method.

3.2.5 Distance Sensor

The distance sensors are infra-red emitter and receiver on the mobile platform. Each time the phone sends a request to the mobile platform, it answers with all its sensors values. The phone get a number between 0 and 1000 for the distance values. The phone directly store the value given by the mobile platform without any further processing.

3.2.6 Emitter and Receiver

The Webots emitter and receiver nodes are designed to simulate a radio device. Unfortunately there is no such (controllable) device on the phone. However the Wi-Fi can be used, either by connecting all robots to the same network, or by using direct Wi-Fi connection between the robots. There is an API for Wi-Fi direct in Android version 4. Another possibility is to use the bluetooth.

Since we currently have only one robot these possibilities have not been fully explored yet.

3.2.7 GPS

The Webots GPS model gives Cartesian coordinate in meters in the Webots world referential. The point of origin is the center of the world. But a real GPS device gives latitude and longitude. To comply with the Webots API, the coordinates are converted using a user-defined point of origin. The GPS can be switched in real mode in the phoneBot preferences to receive latitude and longitude as real GPS but it is not supported in Webots.

The GPS take some time to acquire the first location position and have a very variable accuracy. Location is also obtained with the network (cell tower and WiFi access points) if more accurate. Generally the GPS device will not give accurate enough measures to be really useful for the robot.

3.2.8 Microphone

The Webots sound API is currently under improvement in another project. For now when the microphone is enabled it use its refreshed rate to compute the necessary buffer size to record sound for this duration. At each step the sound data are transferred from the low level internal buffer to an application buffer that the user can access. If the call to the robot step function is delayed because of a too high CPU load, then the internal Android buffer will overrun and some sound data will be lost. The PCM-16 format is used with a sample rate of 44100 HZ and mono channel.

3.2.9 Speaker

Currently the Webots API only allow to play a raw sound buffer. Since Android supports a lot of high level sound capability, support for playing a file and voice synthesis have been added on the phoneBot. These features are not integrated in Webots and therefore works only on the phone. Android also supports two modes : static and stream. The first one corresponds to the Webots API and play a sound buffer once. The second one is explicit by its name, data are streamed to the sound device. Data must be pushed fast enough to prevent the sound buffer to become empty.

3.2.10 Touch Sensor (touch screen)

Touch screen on android works like always with callback. When the screen is touched a touch event is generated, and a callback is called with this event

as parameter. Touch events need therefore to be buffered by the phoneBot application. During a robot step function, if there is a pending touch event, the corresponding touch sensor state is set accordingly. Then the following call to robot step will clear this state and check if there is another pending touch event.

Webots cannot currently simulate a touch screen. There is however different types of touch sensor device that can be used to give controllers access to the phone touch screen. The `bumper` and `force-3d` types are used.

The `bumper` type is used to implements buttons. To allow flexibility the screen has been subdivided recursively : the whole screen is a touch sensor, then it is divided in two other touch sensors, and so on. In this way different resolution (button size) can be chosen. These sensors act as buttons and just return 1 if touched during the last robot step or 0 otherwise.

The `force-3d` type gives a double for the force applied in each of the three axes. To be able to access directly the coordinate of a touch on the screen, a `force-3d` touch sensor is used to return these values. So it is not a real `force-3d` sensor but instead return the x and y coordinate, and the pressure of a touch event. If several touch events are raised during the same robot step, only the last one is considered. Since the touch screen is multiple point there can be two touch points at the same time. The second point is store in the same array which is therefore of length 6 instead of 3. However accessing this second point is only supported on the robot because the size of this buffer in the Webots `libController` is only 3.

In the future Webots could integrate a real touch screen type and the phoneBot will have to be updated accordingly.

3.3 Remote Control

Webots provides a remote control mode that allows the controller to run on the PC with the Webots `libController` and sends instructions to a real robot. The robot specific remote control library is compiled as a dynamic library and linked with the `libController` at runtime.

In simulation mode, the `libController` and Webots communicate by the use of `WbRequest`. Each time a device command function of the `libController` is called by the controller, the internal state of the device is changed inside the `libController`. Then, on the next robot step, and if a device internal state has changed, this state is written in the `WbRequest` which will be sent to Webots. Finally, after the physics step, Webots sends back a `WbRequest` with the new sensors values. This `WbRequest` is parsed by the `libController` to update the internal values of each device accordingly.

In remote control mode the same system is used except that the `WbRequest` is not sent to Webots but is used by a remote control module of the `libController`. This module parses the request and call the corresponding device command functions of the remote control interface. The `WbRequest` is not directly exposed to the remote control library to keep this communication mechanism internal to Webots. It could sound cumbersome to write the data in a object and then read them in the same process to call functions. But it has the advantage to avoid adding special handling code in each function of the API and allows the remote control mode to work in the same way than the simulation mode.

However the `WbRequest` in not used in the other direction to set the sensor values. The `libRemoteControl` directly call functions in the `libController`. For each device get values function of the Webots API, a corresponding set values function is used to update the sensor values. These functions must only be called during the robot step.

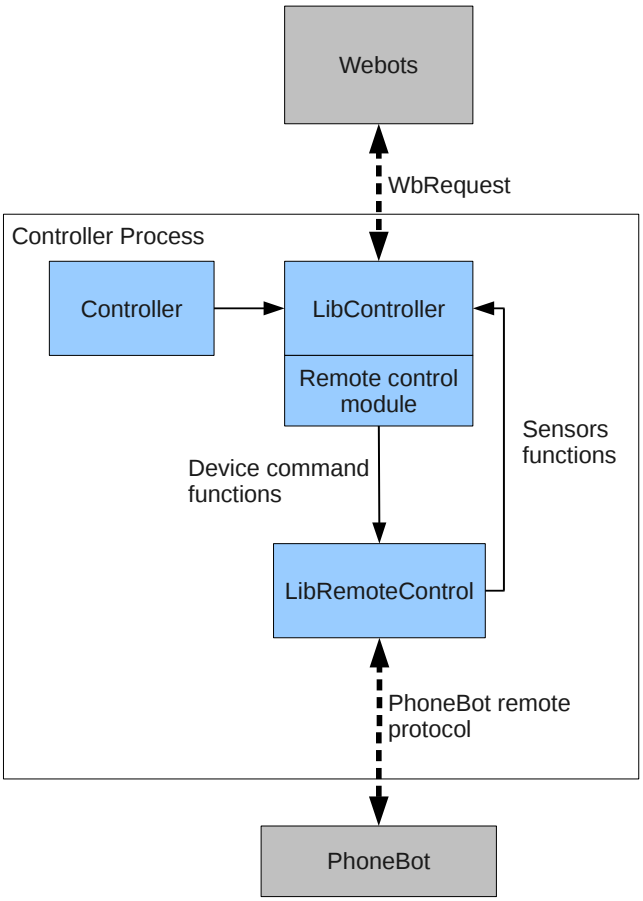


Figure 11: Remote control system

As stated before the remote control API is bidirectional : the `libController` need to call function in the `libRemoteControl` to actuate the devices and reciprocally to report the sensors values of the devices. The `libRemoteControl` is linked to the `libController`, call in this direction (for sensors functions) works as with any dynamic library.

For the other direction, one possibility is to compile the `libController` without checking at compilation time the existence of external function calls (which is the default with the GNU linker for dynamic library). Then, once the remote control library is loaded, these functions can be called and their actual addresses are dynamically resolved by the dynamic linker (as with any shared library). But there is a problem if a function of the `libRemoteControl` is called while the library is not loaded or if the function is not implemented, then it will crash with a undefined reference dynamic linker error. The behavior could also differ with a different dynamic linker on a different operating system. Another drawback is that it forces the `libRemoteControl` to write C functions with the expected name.

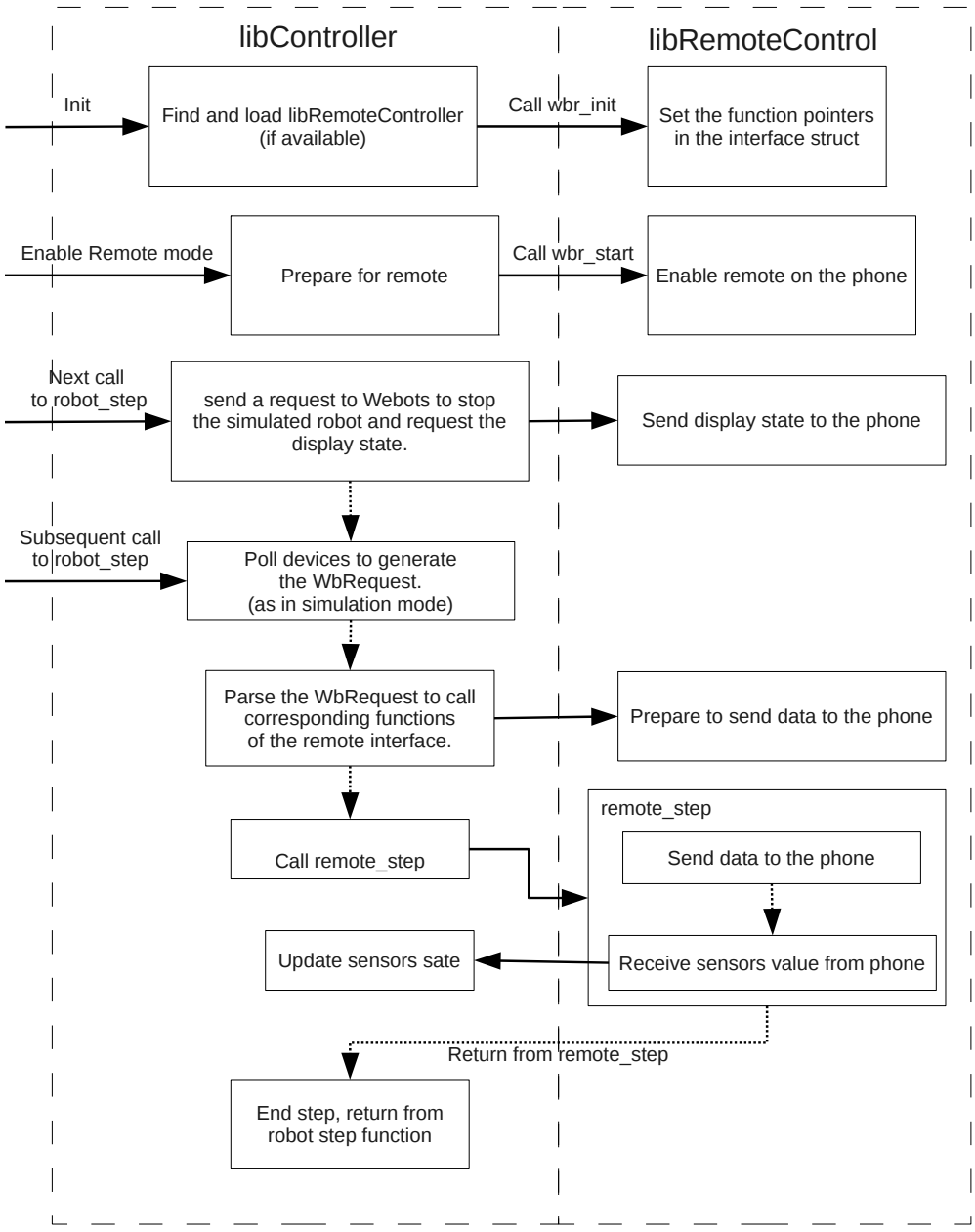
To fix these issues, the system works with function pointers defined in a C `struct` called remote control interface. A pointer to this `struct` is passed to the initialization function of `libRemoteControl`. The remote control library assigns these pointers to its own defined functions which can have any name and can be C++ function (except of course object non static method). Then the `libController` can call these functions through the function pointers. It can also check that a function is defined (not NULL) and can produce a warning instead of a crash. This allows remote controller libraries to no implement functions that does not exist on the real robot.

Devices retain their internal state like : is enabled, current actuator values, and so on. So when the mode switches to remote, a function is called in all devices to advertise them. In the next robot step, each device will then rewrite these values in the `WbRequest` as if the corresponding actuator functions have just been called.

The display device is much more complicated than the other ones for the remote controller. Indeed the internal state of the display is not stored on the `libController` but on Webots. Thus when the remote mode is started, it first asks Webots to send the current image of the screen, and all other state data. When the mode switches back to simulation, the same must be done in the other direction. It would require the `libRemoteControl` and the phoneBot to have special handling code and would really complicate the disabling of remote control. To avoid that, Webots keeps simulating the display which means that the part of the `WbRequest` concerning the display

is still sent to Webots.

The procedure is summarized in the following graph. Plain arrows represent function calls between the two libraries and dashed arrows represent logical transitions. The vertical order of the node represent their execution order.



3.3.1 Communication

The communication between the phone and the PC uses TCP/IP. The phone is expected to be connected by Wi-Fi but it could also use cellphone network. It is important to minimize the network delay.

On the PC side the TCP socket is created with the Qt framework to allow better portability. On the phone it obviously uses the Java API. Both the phone and the PC are little endian but the Java DataStream classes are only big endian because it is the default network byte order. Qt stream on the other hand can be configured in little or big byte order. To avoid rewriting the Java stream class, big endian byte order is used in the communication protocol. However buffer (for sound or image) are send as-is to avoid any useless swap overhead even if they contain integer. Buffers are therefore little endian.

Because of Wi-Fi and TCP characteristics it is better to send data in large packet and to avoid byte per byte transfer. On the Java side one must take care to use a buffered stream to ensure that. On the Qt side it is possible to set the `TCP_NODELAY` propriety to disable Nagle's algorithm. This algorithm is designed to limit the number of packets sent but it is not suitable in our case as the measures has shown.

The measures are done with a simple controller that sends 8 bytes and receives 16. It measures the time of the robot remote step, which mean the time to send the data, perform the step on the phoneBot (including communicate with the robot platform), and receiving the answer.

Without optimization	43 ms
With buffered stream	13.5 ms
With `TCP_NODELAY` and buffered stream	5 ms

Table 3: Communication Delay

Another performance issue is the Wi-Fi power save mode. If the channel is idle during some time, Wi-Fi switches to power save mode. In that mode the radio is not on all the time. Packets are buffered and sent periodically when the radio is up. It saves power but at the cost of increasing the network latency which is not suitable for the remote control mode. It causes the paradoxical effect of having robot step with a longer time step to perform worse than with shorter time step. It is because Wi-Fi will switch to power save mode above some time threshold. The android application must therefore acquire a `wake-lock` that prevents Wi-Fi from going to power save mode. To demonstrate the effect the following graph compares the no power save mode

forced by the `wake-lock` and the power save mode. The measures are done with a robot time step of 128 ms to trigger the power save mode.

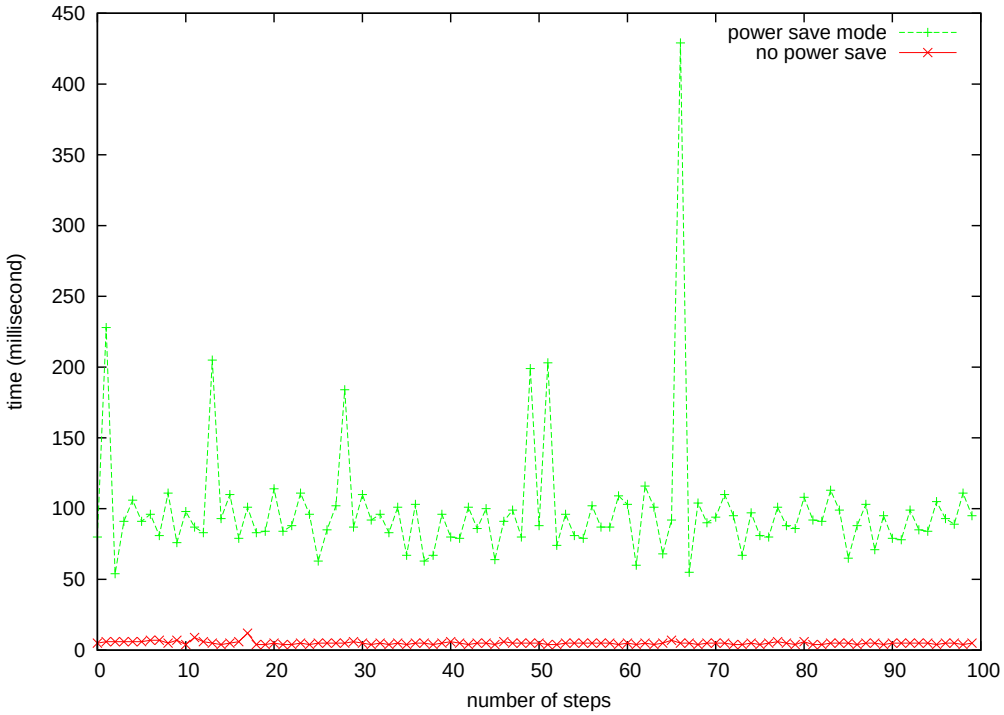


Figure 12: Impact of Wi-Fi power save mode

3.4 Programming Languages

The robot can be programmed in different languages. In remote mode, since the controller is run on the PC, all languages supported by Webots are available. The poneBot can run robot controllers in Java, C/C++, and Python.

3.4.1 Java

The phoneBot need to run external code in its own application which is a feature Java allows. With the reflexion ability of Java code can be easily loaded at runtime.

Since the phoneBot provides a compatible Java Webots API implementation, class files compiled for Webots can be directly run in Android. The only difference with standard Java is that Android compacts class files in one android `dex` format file which has a slightly different class format. The only

thing required to generate a phoneBot Java controller is to use the Android `dx` program to convert class files and compact them in a jar file. Then the phone can use the `DexClassLoader` to loads the controller class and call its main method. The main method must be in a class with the same name than the controller and therefore the same name than the generated jar file. This file is named with `.apk` extension to avoid confusion with normal jar files.

3.4.2 C/C++

Running native controller is more complicated. Android supports the possibility to use native libraries by the Java Native Interface (JNI). A native development kit (NDK) is provided to cross compile in all architecture supported by Android. The JNI provides the necessary tools for a native code to call Java methods by reflexivity. A Webots C API implementation compiled with the NDK has been written for the phoneBot. It is actually a wrapper to the corresponding Java API.

There is just a special case with the camera. The Java Webots API for the camera returns the image as a Java int array. Therefore it uses a 4 bytes format (RGBA). However for the C function the API requires to return an RGB image as a byte array (3 bytes per pixel). To avoid any copy overhead a special YUV convert function has been added to produce directly an RGB image. Since the conversion library is also native, it directly writes the result in a buffer of the native controller library avoiding any copy between Java and native arrays.

Webots C/C++ controllers are compiled with the cross compiler and header files of the NDK but use a custom Makefile. They are built as shared libraries because the native controller need to run in the same process as the phoneBot application to get access to the Java Webots API. So the controller process links dynamically the Webots controllers and calls their main function. The starting procedure is the following :

- Load the controller native library
- Copy the robot controller shared library from the sdcard to the local memory because android does not allow any files from the sdcard to be executed, even loading a shared library.
- Call the controller library initialization native method which :
  - Load the robot controller (dlopen POSIX function)
  - Get the main function symbol (dlsym POSIX function)
  - Call the main function of the robot controller

The Webots C++ API is only a wrapper to the C API. The same code as for Webots is used but devices that do not exist on the phoneBot have been removed because their corresponding function are not present in the phone C library.

3.4.3 Python

Python is an interpreted language, it's not directly supported on android. There is a project Python-for-android(P4A)[10] that proposes an application to run python script. It uses a feature of the Python library which allows to use it directly from C program to run python scripts. But P4A requires the script to be run in their application and doesn't provide enough control. However their slightly modified python library and header can be used directly. It is integrated in the project as a precompiled native library.

Webots Pyhton API is automatically generated with a tool called SWIG[11]. It produces a Python module made of a Python script file and a cpp file which wraps the C++ API. This file just need to be compiled for ARM as any native library. The only required trick is to rename the library to prefix it with 'lib' because the packaging tool only integrate libraries and it consider that library names are prefixed with 'lib'. So the SWIG generated files are patched to change all the references to the library name.

Since the Python script wrapper file is not a library it cannot be integrated in the same way. There is unfortunately not way to integrate custom files to be installed in an application package. The only ways is to use `assets` but these files are not actually created on the file system, they can only be read by Android API calls. Therefore an installer has been added to read these files from the `assets` and copy them as actual file on the required location.

There is also a documented bug with the the Python library :

> Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls Py_Initialize() and Py_Finalize() more than once.

The SWIG module seems to be in that case and it means than a Python controller cannot be restarted without relaunching the controller process.

4 Results

The main goals of the project have been achieved. As a result there is a fully functional Android application that allows Webots robot controllers to

[10] `http://code.google.com/p/python-for-android/`

[11] `http://www.swig.org/`

control the robot and all its devices. The application can run Java, C, C++ and Python controllers. Webots controllers have been written in all these languages to test and demonstrate the functionalities of the phoneBot.

The other part of the project was the integration with the Webots software. The remote control system of Webots has been redesigned and improved to add the support for all the Webots devices. A remote controller library has been written to communicate with the robot. A robot window for the PoneBot was also written based on the existing e-puck window. This window could be improved in the future to look better and to have more functionalities like a better interface to manage files on the phone, currently it only allows to upload controllers.

Th minimum time step duration has been tested with a robot controller that enables all the devices except the camera and uses some display draw functions. The TIME_STEP parameter is set to 0 as well as the devices refresh rates to be able to measure the maximum rate. The time between two steps is measured and an average is derived.

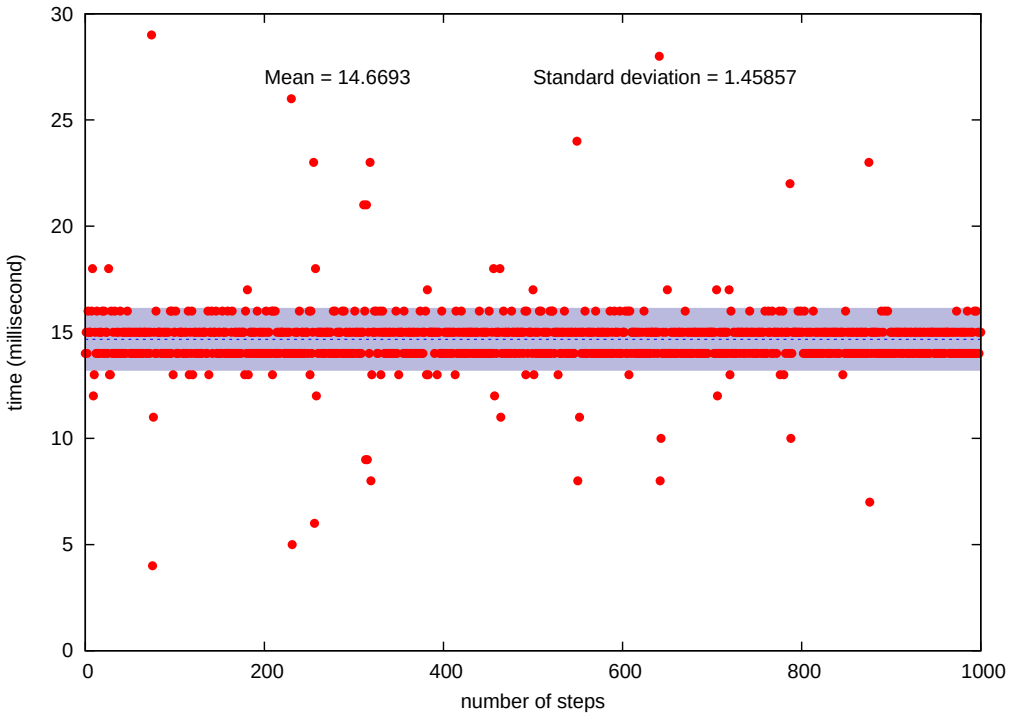


Figure 13: Performance

The average is 14.6 ms but the minimum is only 4 ms. If the system were clear of other processes better performance could be achieved. With the remote

control mode, the step on the phone (including the Wi-Fi communication) only lasts 7 ms on average. It is because Webots is slower and takes some time between two robot steps which allows the phone to schedule other processes during this idle interval instead of during the robot controller execution.

As discussed in the camera section, when enabling the camera the results are worse. Capturing an image only take 5 ms but the scheduling issue of the operating system make it much slower on average. Nevertheless 30 fps can be almost guaranteed and higher frame rate is possible on some conditions.

Sony Ericson has recently released the upgrade to Android 4.1 for Xperia mini. It was too late to take the risk to do the upgrade, but it could be done in the future. The new Android version could improve different things like fixing some issues with the camera and perhaps giving faster results.

The inter robots communication possibility has not been fully explored yet. Robots can of course use TCP/IP if they are connected to the network, but other options could be explored like bluetooth or Wi-Fi direct. But it required to have at least two phones to be able to test, and Wi-Fi direct requires the next android version.

The project was done with only one specific phone but, since the commercial availability cannot be controlled, it will be eventually necessary to change that phone. The difficulty is that it could change the model if there is not the same available sensors, for example not the same number of cameras. The 3D model in Webots would also have to be adapted. The major problem would be for the robot platform because of the change of size and weight of the phone, and the change of position of the USB port.

Only small and simple robot controllers have been written so far but with the capabilities of the phone more complex robot controllers could be written to demonstrate the capabilities of this robot. The support of openCV opens a wide range of possibility for interesting robot controller by using computer vision. Other libraries could be integrated for example to do sound processing. The possibilities of a full operating system like Android are limitless.

Appendix

A User Guide

This user guide is designed to give the necessary information to get started with the phoneBot. It only focuses on phoneBot related topic, consult the Webots documentation for more information about Webots and robot controllers programming.

A.1 PhoneBot application

To start using the robot, switch on the phone and connect it with the robot platform. Then turn on the robot platform. A pop up window will ask your permission to start the phoneBot application. You can also start the application by the app menu and switch on the robot platform afterward, it will also show a pop to ask permission to access the USB device.

Once started the application shows the list of available controllers. Simply click on a controller and choose launch. You can also choose to delete a controller (cannot be undone).

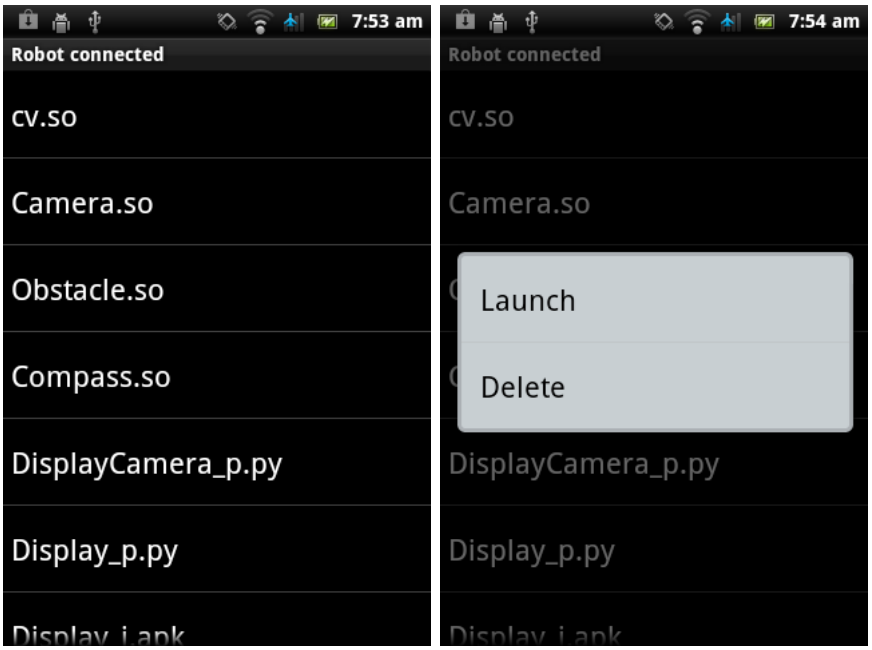


Figure 14: Main screen

Once the controller is launched the screen will become black and a menu is opened to actually start the controller.

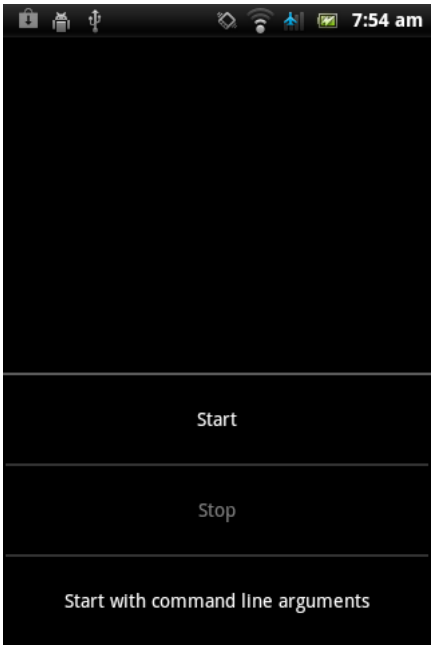


Figure 15: Controller Screen with the menu opened

While a controller is running you can stop it by pressing the phone menu button and choosing stop. It will tell the robot step function to return -1, and let one second for the main function to return(as in Webots). Otherwise the controller process is killed. If the controller was not kill you can restart it directly with the menu. To choose another controller press the phone back button.

You can also choose the `start with command line arguments` option to provide arguments to the controller in the same way as with the `controllerArgs` field of a robot node in Webots. One or more space separate two arguments, an argument cannot contain spaces.

A.1.1 File organization

The application uses the SD card to store all user files of the phoneBot application. If the application is started without the SD card it will use the internal phone memory but it is recommended to use the SD card. A `PhoneBot` directory is automatically created on the SD card. The same structure as in Webots is used : a `controllers` directory contains a sub directory for each controller that in turn contains the controller program.

```
PhoneBot/
  | ---- controllers/
            | ------ my-controller1/
            |              |--- my-controller1.so
            |
            | ------ my-controller2/
            |              |--- my-controller2.apk
            |
            | ------  ...
```

When using Webots API functions that access files (e.g camera save), relative path (ie path not starting with `/`) are computed from the `controllers/<controller-name>/` directory. Directories along the path are automatically created if necessary.

If you want to directly save a file with your own code use the `wb_robot_get_project_path` function that, as in Webots, returns the directory containing the `controllers` directory, namely the `PhoneBot` directory. The current directory of the application is defined by the android framework, do not directly save files with a relative path.

A.1.2 Configuration

The phoneBot can be configured in several way. First when the application is installed or updated a default configuration file is used to set the initial configuration. Then the configuration can be changed directly on the phone by pressing the phone menu button (while on the main screen).

Figure 16: Preference Screen

On the preferences screen you can use the phone menu to restore the configuration to the default one. You can also load a global configuration file that must be named `config.conf` and must be put in the phoneBot root directory on the sdcard.

An example of the configuration file with all the supported option can be found in the phoneBot directory of Webots.

It is also possible to provide a configuration file for a specific controller. Simply put a .conf file in a controller sub directory.

A.2 PhoneBot Model

The phoneBot can be used in Webots by importing the corresponding proto model from the `projects/robots/phonebot` in the Webots installation directory. The phoneBot is a differential wheel robot. Therefore OO controllers (Java,Python,C++) must inherit from `DifferentialWheel` instead of `Robot`. It has the following devices :

Device Name	Webots Type	Short description
prox0 - prox4	distance sensor	IR sensor looking forward
ground0 - ground4	distance sensor	IR sensor looking at ground
accelerometer	Accelerometer	
compass	Compass	
gps	GPS	
camera	Camera	
display	Display	Phone's screen
speaker	Speaker	
microphone	Microphone	
touch	touch sensor (bumper)	touch screen as a whole
touch0 - touch1	touch sensor (bumper)	half height of 'touch'
touch00 - touch11	touch sensor (bumper)	half width of 'touchx'
touch000 - touch111	touch sensor (bumper)	half height if 'touchxx'
touchScreen	touch sensor (force-3d)	direct touch coordinate

Table 4: Robot devices summary

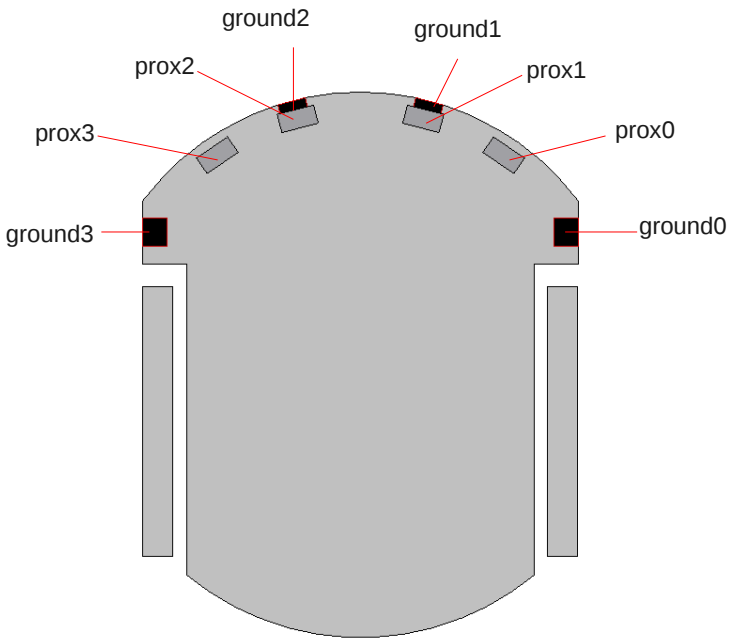


Figure 17: Position of distance sensors

Refer to the Webots reference manual for the devices API. You can also take a look at the sample controllers in the phoneBot directory. Some specificities are explained below.

Touch Screen

There is currently no touch screen device on Webots but the touch sensor device is used to implement it. You can use the touch screen in two ways. Either as a grid of buttons of different size which are implemented by touch sensor of bumper type. Or, if you want the coordinate of a touch point, you can access the device as a touch sensor of type force-3d.

The bumper touch sensors are organized in a hierarchical grid to allow you to easily program buttons of different sizes. They are numbered in a prefix manner which means that if, for example the top right corner is touched, then touch, touch0, touch1, and touch 010 are touched (if they are enabled). The `wb_touch_sensor_get_value` returns 1 is the sensors was touched during the last robot step and 0 otherwise.

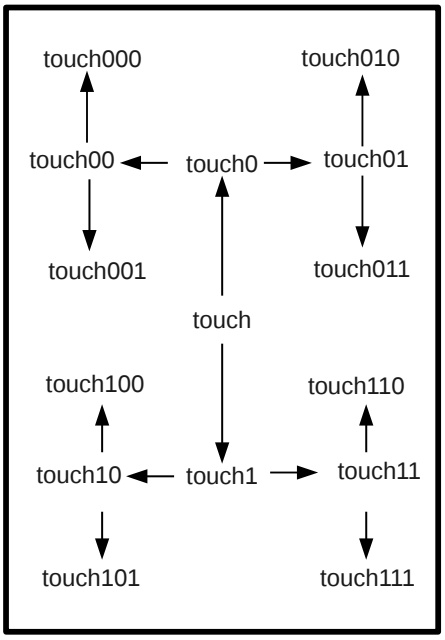


Figure 18: Hierarchy of touch sensor

If you prefer to directly get the coordinates of a touch event then use the `touchScreen` sensor. Instead of giving a 3d force vector as a real `force-3d` type, the `wb_touch_sensor_get_values` gives {x1,y1, pressure1, x2, y2,

pressure2}. The x and y coordinates are only valid if their corresponding pressure is not 0. The x2 and y2 represent a second simultaneous touch point since this touch screen supports multi touch. This is only supported on the real robot since the array returned by Webots is only of length three.

See the sample `TouchSensor` robot controller in the phoneBot directory to see how it works.

Camera

The camera supports multiple resolutions but it is recommended to use the smallest resolution (default) to fasten the capture time and also fasten your image processing. In Webots the camera resolution can be set in the proto parameters and is automatically used for remote control. You must only set supported camera resolution. On the phoneBot, for local controllers, the resolution can be set in the preferences.

144x176
240x320
288x352
320x480
480x640
480x864
720x1280

Table 5: Supported Camera resolutions (width x height)

The capture time is highly variable, the minimum is 5 ms but generally this time is between 20 and 35 ms. It is therefore not recommended to enable the camera at a faster rate than 32 ms and you should make no timing assumption on the camera speed. If your application is time sensitive you should measure the actual robot time step to experimentally determine the best parameters.

Compass

The compass works with a magnetic field sensor. To give accurate results it need to know the phone orientation. It is assumed that the robot is on a horizontal plane and the north vector is computed with that assumption. You can set in the preferences to use instead the accelerometer to compute the orientation. In this way the compass will give accurate results even if the robot doesn't lie on a horizontal plan. However if the robot is accelerating (change of motors speed, or rotating) the results will be wrong.

GPS

The GPS device uses the satellite and networks (cell tower and Wi-Fi access points) to perform localization. In the Webots models the position is given in Cartesian coordinate in meters from the center of the simulation world. On the real robot this center is specified in the preferences. This device cannot measure height, therefore the y coordinate is always 0.

You can also activate the real gps option in the preferences to transform the device to a real GPS. In that case the array returned contains : {latitude, longitude, accuracy}. Latitude and longitude are given in signed degree and the accuracy is in meters.

Note that the enable function of the GPS takes some time which produces a noticeable lag. If the GPS was never used by the phone, it can take a very long time to acquire a first position. Otherwise the last known position is used.

A.3 Sample controllers

You can find various robot controllers samples in the phoneBot controllers directory. They illustrate devices usage in the different programming languages supported by the phoneBot. It is a good reference to understand how a device works. The `TouchSensor` example shows how the touch button grid is organized and how to use the special touchScreen sensor. The `display` example shows all the function of the display API and there is a C, C++, Java and Python version that highlight the API differences of these languages. There are also examples to show openCV or NEON usage, and also a sound example.

Some small simple demonstrations are also included like a braitenberg collision avoidance algorithm, a robot following the compass, or an example of simple image processing with a robot that follows a colored ball.

A.4 Robot window

While the controller is running on Webots you can double click on a robot to show its robot window. It displays all the enabled sensors with their values. It also allows to upload controllers to the robot and to start the remote control mode.

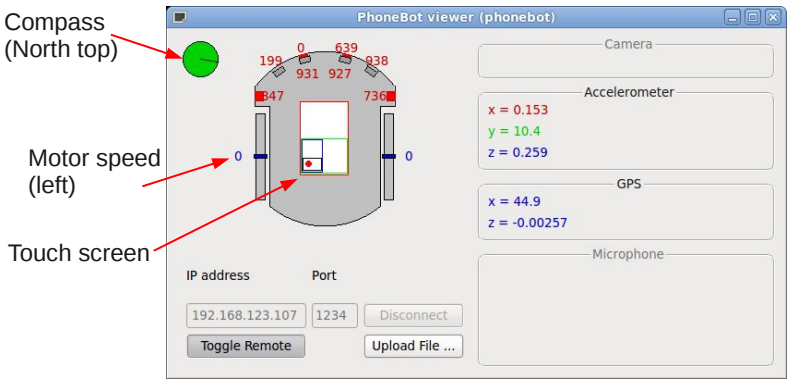


Figure 19: Robot window

A.5 Remote Control

To use the remote control, ensure the phone is connected to the network and that you know its IP address. On Webots double click on the robot to show the robot windows(the simulation must have started). Enter the ip address and port, then click connect. The default port is 1234 and can be changed in the preferences.

If the connection worked the `toggleRemote` button becomes clickable. It will switch to the remote control mode. The robot controller is not restarted and will continue running. You can therefore seamlessly switch between simulation and remote control.

When the remote mode is enabled the overlay of the camera in the webots 3D window shows the robot camera image (if enable). The image can also be seen in the robot window. The display is still simulated in Webots therefore you can still see the display overlay updating and showing the same image as the phone screen.

Note that in the remote mode the robot controller need to communicate with the real robot, it will create a delay depending of your network configuration. You can see the time and simulation speed in Webots as with simulation. Of course using the camera, sound, or display imageRef required to send more data over the network and could slow down the simulation.

A.6 Onboard Control

The phoneBot support robot controllers in Java, C/C++ and Python. To cross compile from Webots you must copy the `Makefile.phone-bot` from the phonebot project directory to your controller directory. Then you can

use the cross-compile button on the Webots controller editor. Note that if you add the Makefile while the controller file is already open, you will have to reopen or save the file in order to refresh the button.

Note that Java controller generate a .apk file and C controller a .so file. Since Python is not a compiled language you can directly upload the .py file. The phoneBot application will not recognized file with other extensions.

We recommend to use the robot window upload button to upload the file to the robot. Alternatively you can also directly put the file yourself on the correct folder in the SD card.

Due to a bug with the python library, python controllers cannot be restarted without restarting the process. This restart is done automatically.

A.7 Advanced

A.7.1 OpenCV

OpenCV is a library for computer vision which is also supported on Android. You can enable it on the phoneBot in the preferences. If not already installed on the phone, it will ask you to install it the next time you launch a controller. To cross-compile a controller with OpenCV, you first need to download the openCV android package on `http://opencv.org/downloads.html`. Extract it whenever you want (only the native folder is needed). Then edit your Makefile.phone-bot to add :

```
CFLAGS = -I<path-to-openCV>/native/jni/include
LIBRARIES = -L<path-to-openCV>/native/libs/armeabi-v7a \
            -lopencv_java
```

Don't be confused by the name of the library, it is not for Java. It is just used to link the controller, the real library will be used on the phone.

You must not use the camera from OpenCV, use the Webots API as usual. See the example in the sample.

A.7.2 NEON

NEON is the single instruction, multiple data (SIMD) instructions set of ARM. You can directly add instructions in assembly in your C/C++ controller with the GCC syntax for inline assembly. The required compilation option for NEON is already enabled.

We don't recommend to use the GCC NEON intrinsic because it seems to produce very unoptimized code (bad register allocation) with the GCC version provided by android.

You can also use the automatic vectorization option of GCC, simply add `-ftree-vectorize` or `-O3` to your `CFLAGS` variable in the Makefile.phone-bot. Consult the ARM documentation for more information about NEON.