

Using Python on Wallaby

Frank Blackburn (Name Censored), Co-author Name Censored

Corresponding Author: Name Censored

School Censored

Abstract: This paper introduces the whole process of application development in the Wallaby controller by using Python language and discussing its shortcomings and future works.

Keywords: Python, Botball, Wallaby Controller.

Using Python on Wallaby

0. Introduction

The Wallaby controller is a Linux-based controller with ARM architecture. While mainly used for robot control in GCER conferences, wallaby can be more than a controller solely for robotics competitions, for its relatively high-performance processor and various built-in applications. In this paper, we introduce the main application of Python in the wallaby and discuss its shortcomings and future.

1. System configuration

The provided KIPR Software Suite, with GNU/GCC compiler; python interpreter; IDE for them and non-interactive command prompt, is indeed very powerful, but does not fully meet our satisfaction. We need an interactive shell to perform some of our specialized actions.

The first option is to use a computer with a program that can establish SSH shell connection to the wallaby controller, such as PuTTY on Windows or SSH on Linux.

After connecting to the controller via Wi-Fi, we establish SSH link to the Wallaby with address **192.168.125.1** and port **22** by default, and login with username **root** which has no password. You can see the shell as in the terminal on Wallaby afterwards, as shown in figure 1.1 and 1.2.

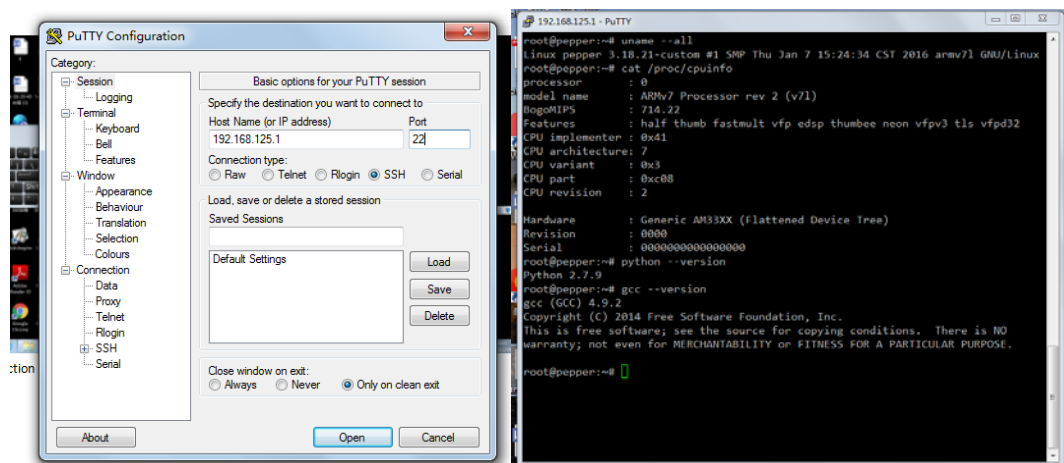


Figure 1.1, 1.2 Terminal on PuTTY

The second option is more direct - plug an USB keyboard into the USB slot on the controller and enter the terminal on the controller. The running effect is shown in figure 1.3.

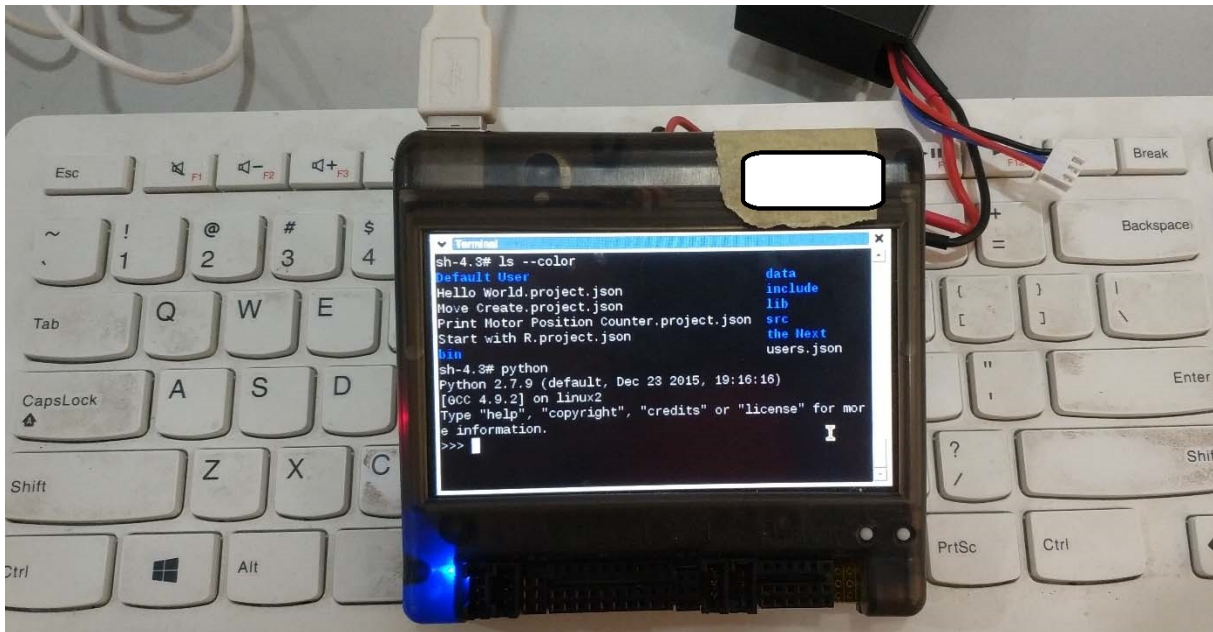


Figure 1.3 Running Effect on Wallaby Controller

2. Use of Python

2.1 Basic Information

Wallaby by default is equipped with Python 2.7.9 and GCC 4.9.2, which can be considered outdated[1, 2]. While basic wallaby control functions are implemented, some interactive-related functions, such as those for buttons, however, are not in the library for some Wallaby. See Appendix 0 for a full supported functions' list.

2.2 Basic usage

Although python is a scripting language, we can use python just like C. Referencing example given by KISS IDE, we can write our own python testing program for the controller.

```
#!/usr/bin/python
import os, sys
from wallaby import *

def main():
    print "Hello World"
    motor(0,100)
    msleep(5000)
```

```

    ao()
if __name__ == "__main__":
    sys.stdout = os.fdopen(sys.stdout.fileno(),"w",0)
    main();

```

Use `vim` to edit and save the file in `~/python.py`, and execute it by running `python python.py`. We can see the motors running for 5 seconds and then being shut down.

Do keep in mind that nothing will be outputted on PuTTY console, because `stdout` has been redirected to somewhere KISS IDE can receive them. Removing line 11 can be a good way to output normally as if you are using python on other platforms.

Here's the equivalent C file, as most of us are familiar with:

```

#include <kipr/botball.h>

int main(void){
    printf("Hello World!");
    motor(0, 100);
    msleep(5000);
    ao();
}

```

Using C, however, without KIPR IDE, things could get much harder. Executing `gcc main.c -o main` cannot compile the program due to lack of required libraries. Adding `libwallaby` to the compiling option will be a good solution - see section 2.4 to know further about this finding and solution.

2.3 Using Python in Real-time Debugging

Python is a scripting programming language, and this feature makes it very helpful in debugging. When using C, a compile must be done every time after the source code is changed to let the changings take effect, and sometimes it is very complicated to know where the program starts to give unwilling results. Plenty of debug-only print functions must be added and debuggers in the team will have to stare at the screen carefully to avoid missing any valuable information. We will introduce a much simpler python debugging workflow below.

To debug, execute `python` in interactive shell first to enter python, and you will be very likely to see something like this:

```

Python 2.7.9 (default, 2018-03-15, 19:16:16)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>

```

Execute `from wallaby import *` and `import os, sys` to import some of wallaby's essential feature. Afterwards, you can try step-by-step execution of code to find some constants, such as the time for motors to run, like this:

```

>>> def motormove(time):
>>>     motor(0, 100)
>>>     motor(1, 100)

```

```
>>> msleep(time)
>>> ao()
```

The advantage is that you can continuously execute `motormove(time)` without compiling it again. If the robot moves a distance less than expected, you can extend the time a little and perform another attempt easily.

Here's another possible usage:

```
>>> def findtime(power):
>>>     motor(0, power)
>>>     motor(1, power)
>>>     elapsedTime = seconds()
>>>     input()
>>>     return seconds() - elapsedTime
```

Execute it using commands like `findtime(87)` and press enter when the robot reaches designated location. Time required for such displacement will be displayed on the screen afterwards. It is also possible to do so with C, but the process will be much more complicated.

2.4 Extending python library with C

Sometimes existing libraries may be wanted by some teams that have been participating Botball tournaments for some time. It might be unwilling for them to use python, since everything needs to be re-constructed in another programming language; also, not all the features in C is fully implemented in Python. Nevertheless, CPython, the python interpreter that we are using, supports a feature that make python possible to execute dynamic link libraries compiled from C programming language.[3] This feature can significantly help and speed up the switching procedure.

Next, we will discuss a way to implement this feature.

2.4.0 Concepts

We use dynamic link library generated by gcc to complete the extension process mentioned in the previous section.

Dynamic link library is a manner to simplify the structure of source code and reduce the size of compiled program by creating a file that contains executable materials that can be executed by the main program only when needed[4].

2.4.1 Creating C files

First, create a new project on KISS IDE and reference existing libraries as functions. In this case, I am only compiling some random things:

```
//cDLLTest project
//Created by censored 2018/Mar/16
#include <kipr/botball.h>

int returnSomeAnalogValue(unsigned port){
```

```

    printf("So exciting to code with Miyamizu Mitsuha-chan!\n");//No,
there's no Mitsuha... Actually
    return analog(port);
}

```

This piece of code will print some random thing on the screen and return the analog sensor value of given port, as if you are calling it in `main()` function on C.

However, do keep in mind that this program will not, and also cannot be executed directly, and as a result, no `main()` function is needed. As a matter of fact, no main function should be in this program unless only one way to execute this program (that is, calling the `main()` function) is expected.

2.4.2 Compiling DLLs

Go to the position of this file in the interactive shell. In this specific case, I will need to execute:
`cd ~/Documents/KISS/Tachibana_Taki/cDLLTest/src`

Then generate the desired Dynamic Link Library using:

```
gcc main.c -fPIC -shared -o libtest.so
```

Use argument `-fPIC` to make the library location-independent (which means it can still be used when it's moved to other directories), and `-shared` to indicate gcc that we are generating a library.[5]

However, the compiler malfunctioned. Please read the following section to understand the symptom. This question remains for quite a few weeks, since we don't know the compiling options KIPR IDE uses.

Symptom

According to gcc manual, other program can use the compiled library by using argument `-l`. However, when testing our library using another c file, it did not work correctly. Some errors were given:

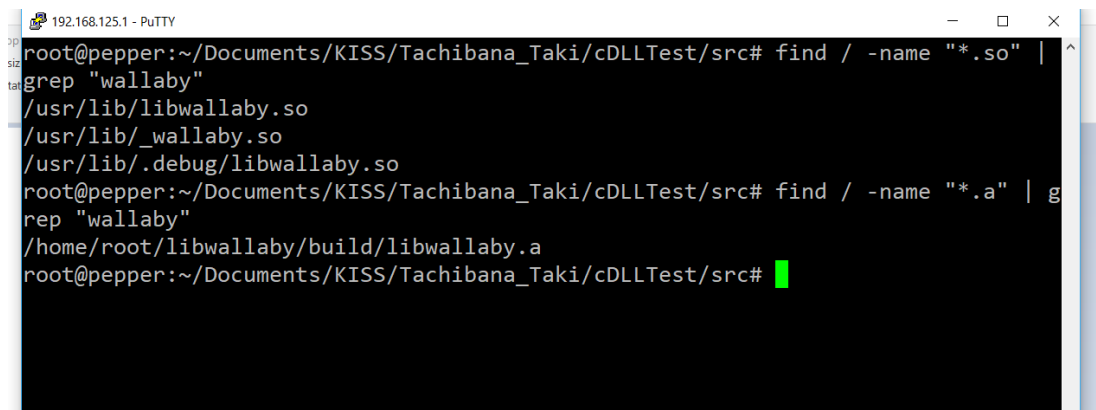
```

/usr/lib/gcc/arm-poky-linux-gnueabi/4.9.2/../../../../crt1.o:      In
function `__start':
/home/kipr/yocto/build/tmp/work/cortexa8hf-vfp-neon-poky-linux-
gnueabi/glibc/2.21-r0/git/csu/../sysdeps/arm/start.S:119:  undefined
reference to `main'
/tmp/cc0b00nf.o: In function `returnSomeAnalogValue':
main.c:(.text+0x24): undefined reference to `analog'
collect2: error: ld returned 1 exit status

```

However, since wallaby doesn't have `lld` program with it[6, 7], and how KIPR IDE uses gcc to compile source codes still remains unclear, the question remain unsolved for some time.

After a few weeks' investigation, it seems that some sort of library is missing, unclear whether static, dynamic or even both it is remains. We searched the entire wallaby with "wallaby", "*.so" and "*.a", and finally got our answer. Figure 2.1 is the search result:

A terminal window titled '192.168.125.1 - PuTTY' showing a search for files. The user runs 'find / -name "*.so" | grep "wallaby"' and gets three results: '/usr/lib/libwallaby.so', '/usr/lib/_wallaby.so', and '/usr/lib/.debug/libwallaby.so'. Then they run 'find / -name "*.a" | grep "wallaby"' and get one result: '/home/root/libwallaby/build/libwallaby.a'.

```
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src# find / -name "*.so" | ^
grep "wallaby"
/usr/lib/libwallaby.so
/usr/lib/_wallaby.so
/usr/lib/.debug/libwallaby.so
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src# find / -name "*.a" | g
rep "wallaby"
/home/root/libwallaby/build/libwallaby.a
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src#
```

Figure 2.1 Search Result

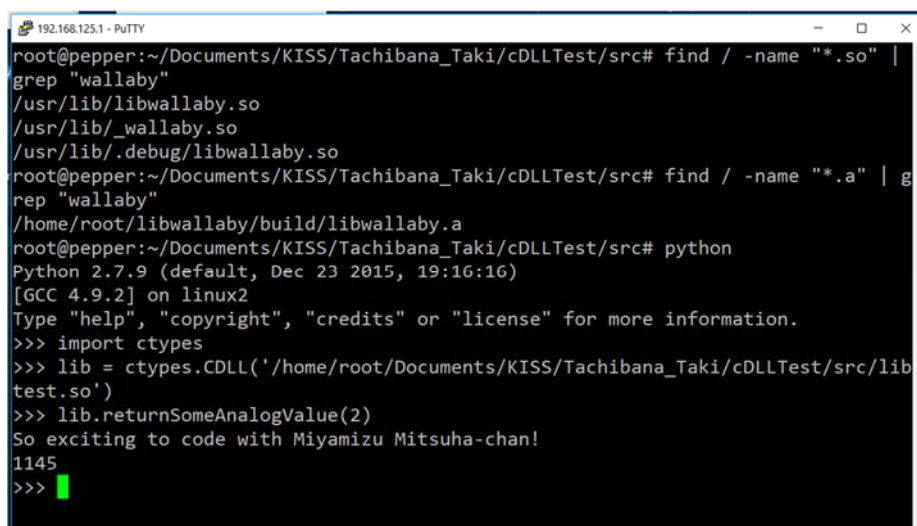
Yes, we missed libraries that we have no idea before: according to the naming rules, they should be referenced as **wallaby** together. And accordingly, the correct execution is:

```
gcc main.c -lwallaby -fPIC -shared -o libtest.so
```

2.4.3 Execution in Python

Import module **ctypes**, which is a core utility of python first. Use CDLL function in this class to let it generate an instance that links to the library - remember to use absolute directory (starting with /) to make sure that python finds the library in the right place.[8]

Call the function by executing the method that have the same name as the function's of the instance, and you can then successfully call the function correctly. Figure 2.2 is the result of executing the function we defined above.

A terminal window titled '192.168.125.1 - PuTTY' showing the same search results as Figure 2.1, followed by a Python session. The user runs 'python', then 'import ctypes', then 'lib = ctypes.CDLL('/home/root/Documents/KISS/Tachibana_Taki/cDLLTest/src/libtest.so')', and finally 'lib.returnSomeAnalogValue(2)'. The output is 'So exciting to code with Miyamizu Mitsuha-chan!' and '1145'.

```
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src# find / -name "*.so" | ^
grep "wallaby"
/usr/lib/libwallaby.so
/usr/lib/_wallaby.so
/usr/lib/.debug/libwallaby.so
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src# find / -name "*.a" | g
rep "wallaby"
/home/root/libwallaby/build/libwallaby.a
root@pepper:~/Documents/KISS/Tachibana_Taki/cDLLTest/src# python
Python 2.7.9 (default, Dec 23 2015, 19:16:16)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ctypes
>>> lib = ctypes.CDLL('/home/root/Documents/KISS/Tachibana_Taki/cDLLTest/src/lib
test.so')
>>> lib.returnSomeAnalogValue(2)
So exciting to code with Miyamizu Mitsuha-chan!
1145
>>>
```

Figure 2.2 Console Output

3. Experiences

An experiment was performed to demonstrate the utility of python language

Simple tasks were assigned to both of our high school Botball teams, given the context that all teammates were initially only used to C programming: two teams are told to program wallaby

controllers based on an existing framework, in C and Python respectively. Their goal is to drive a sensor-mounted robot from one starting area to the other end of the playground following the black lines to collect poms halfway.

Without surprise, although python team had some difficulties in getting familiar with different syntax, they soon become more efficient since they don't need to deal with annoying type conversions and intricate, indirect compiling errors. The time comparison of five experimental runs is shown in figure.

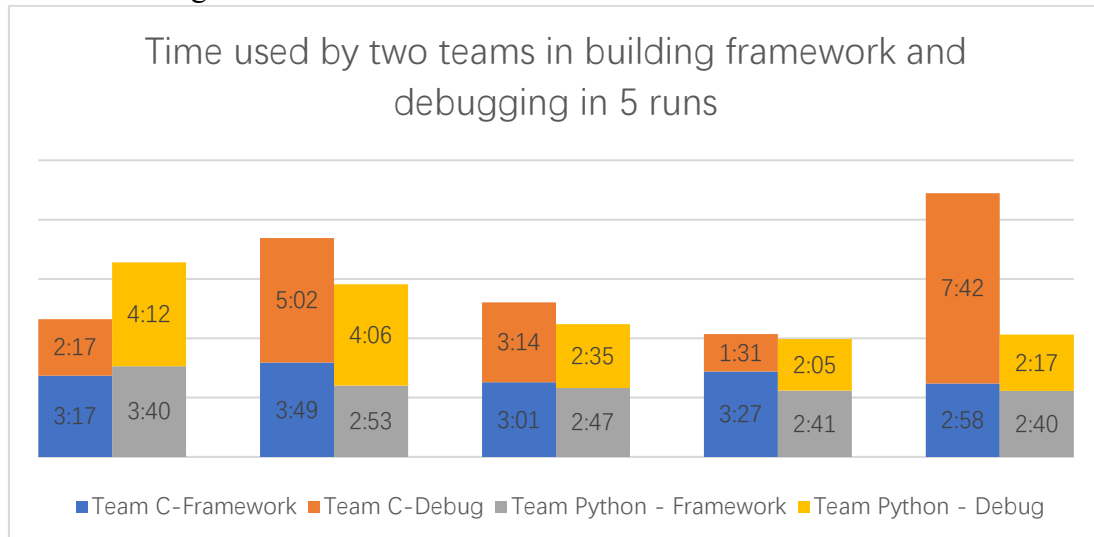


Figure 3.1 Time Comparison for Experiment 1

In another experiment, we set the frisbee to a random height, and two teams using identical mechanical arm are told to program the robot to catch the frisbee steadily for 3 seconds. Time they used to complete each run is shown in figure 3.2.

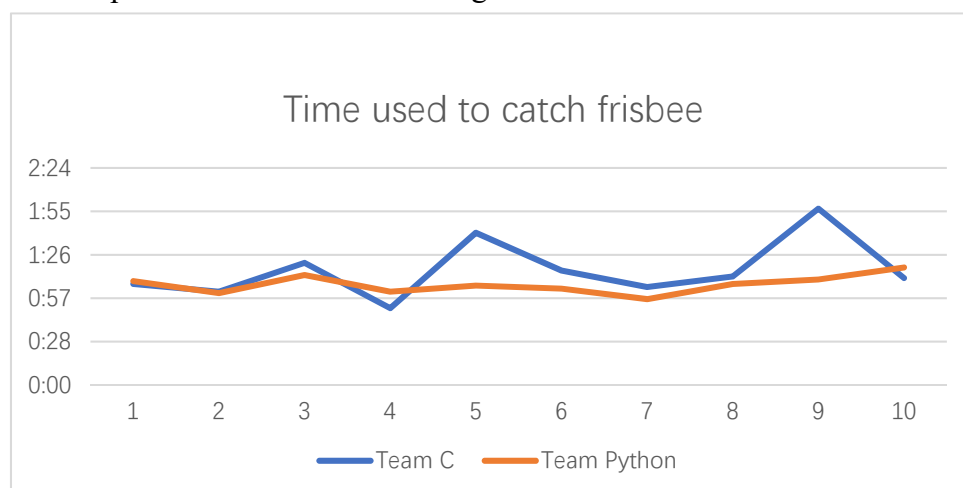


Figure 3.2 Time comparison for Experiment 2

With these data, we performed a 2-sample t-test to see if Python did use less time averagely than C.

Proof:

We represent the mean value of Python as \bar{x}_{py} , and the same rule applies for other values. All values are represented in seconds.

$$H_0: \mu_C = \mu_{py}$$

$$H_\alpha: \mu_C > \mu_{py}$$

$$\bar{x}_C = \sum_{i=1}^n x_{C_i} = 76.3, \bar{x}_{py} = \sum_{i=1}^n py_i = 66.7$$

$$s_{x_C} = 19.4196, s_{x_{py}} = 6.14727$$

Using our calculator, we get a result that shows $pVal = 0.08239$.

With significance level $\alpha = 0.1$, we have enough evidence to not accept that the average programming time of using C is shorter than that of Python's. See Appendix 1 for more information.

4. Advantages and Future

Python is sometimes criticized for its relatively poor running efficiency.[9] However, wallaby is hardly involved in scientific research and thus the slow running speed of python would create limited negative impact on the robot performance. Furthermore, python is more new-learner-friendly and coding-efficient compared to C programming language.

Nevertheless, the manuals, examples and library references (especially those translated materials given by ITCCC in China) are still restricted to C programming language, and some functions are not available in python, possibly due to compiling errors. We are expecting KIPR and other administrative bodies to offer greater support on python.

References

- [1] Free Software Foundation, GCC 4.9.2 Manual, I. Free Software Foundation, ed., 2014. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/>.
- [2] Python Software Foundation, "Python v2.7.9 Release Note," Available: <https://hg.python.org/cpython/raw-file/v2.7.9/Misc/NEWS>
- [3] G. v. Rossum, "Loading dynamic link libraries," in *Python Library Reference*, G. v. Rossum, Ed., ed. Amsterdam: Python Software Foundation, 2008.
- [4] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.
- [5] CSqingchen. (2016). *Generating and using dynamic and static library under Linux*. Available: <https://blog.csdn.net/CSqingchen/article/details/51546784>
- [6] fariver. (2017). *Commands useful for binary files*. Available: <http://www.cnblogs.com/fariver/p/6560885.html>
- [7] flyzteK. (2017). *Generating *.so file and call it under Linux using gcc*. Available: <https://blog.csdn.net/flyzteK/article/details/73612469>
- [8] zhuiy. (2017). *Calling C functions in Python*. Available: <http://www.cnblogs.com/zhuiy/p/4798642.html>
- [9] X. Cai, H. P. Langtangen, and H. Moe, "On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations," *Scientific Programming*, vol. 13, no. 1, 2005.