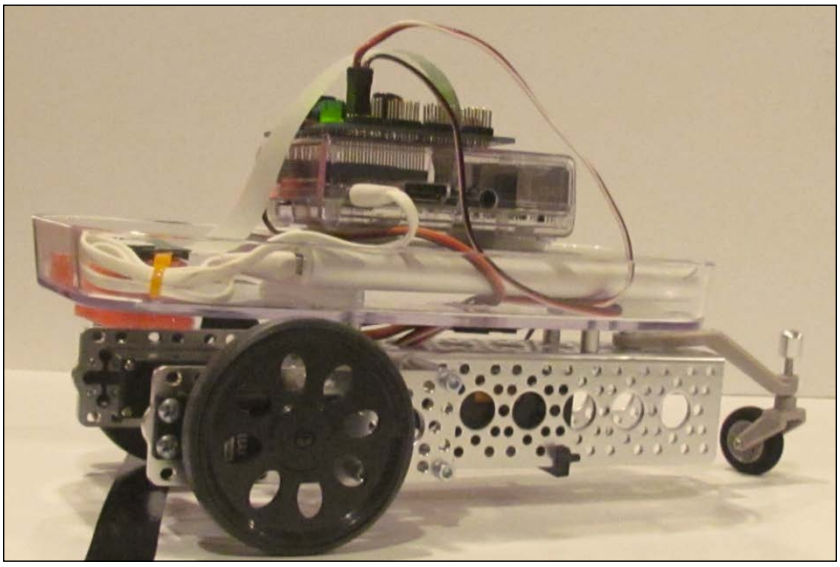
8

Interpreting Commands, Implementing Pipe-based Communications, and Testing Modules

When building any project that consists of multiple modules, it is beneficial to first incrementally build and test the individual programs in isolation. By building a small test application, it's possible to expose most likely errors before you have multiple modules talking to each other.

The materials in this appendix should help you to understand both the implementation methods and the testing routines that help us to gain confidence that the application is performing correctly.

The following image shows the line-following robot built and ready:



Interpreting commands

Let's start this part of the project by creating a small program that listens for user input (the keyboard is connected to the `stdin` port) for input commands and outputs results to the LXterminal text-mode display (`stdout` port). To create the program, perform the following steps:

1. Download the programs from the `Chapter 8` folder (which can be found at `http://1drv.ms/1ysAxkl`)into your `/home/pi/robot` directory.
2. Start IDLE 2 (we are using Python 2 instead of Python 3) and open `listener.py`.

   The listing is as follows:

```
#!/usr/bin/python

import sys

def userfunc1():
    print 'Data + 10 = ' + str(cmddata[1] +10) + '   ',

def userfunc2():
    print 'This is a test ',
```

```
def closeandexit():
    sys.exit('Bye....')


def main():
    while True:
        print 'ok:'
        global cmddata
        try:
            inputstr = raw_input()
            inputstr += ',0,0'          #need at least one
item
            cmddata = eval(inputstr,{'__builtins__':None},
                           {'userfunc2':userfunc2,
                            'userfunc1':userfunc1,
                            'close':closeandexit })
            cmddata[0]()                #execute the
            command
        except (KeyboardInterrupt, EOFError, NameError,
                SyntaxError, IndexError, TypeError) as err:
            if 'KeyboardInterrupt' in str(type(err)): break
            if 'EOFError' in str(type(err)): break
            print 'Error ',

if __name__ ==  '__main__':

    main()
```

When the program starts, it enters an infinite loop (while `True`) and waits for user input (`raw_input()`). The magic starts when the user enters a command that is recognized (the `eval()` function). The `eval()` function parses the input string into a tuple that contains a command address as the element `[0]`. This address can be executed (called) just like any other function in the program, and on the next line, the function in `cmddata[0]` is called (`cmddata[0]()`).

If there are any errors, they are handled in the `except:` function, and then the program prints results and loops back for more user input data.

The `eval()` function is a very powerful tool to parse directly validated function commands in the user input stream without having to do a lot of string comparisons and parsing checks.

The following is an example of what you can expect to see in the Python Shell window:

```
>>> ================================ RESTART ===========
>>>
ok:
userfunc2                    #command entered
This is a test  ok:          #the command executed
>>> type(cmddata)            #after a <Ctrl> + C
<type 'tuple'>
>>> type(cmddata[0])
<type 'function'>
>>> print cmddata            #print the tuple
(<function userfunc2 at 0xb521ee70>, 0, 0)
```

The `eval()` function is considered a serious security risk in Python since it can allow a hacker to execute arbitrary function code if not constrained. Here, we have specified (in curly braces) a whitelist of functions that can be accepted. For more details, see `http://lybniz2.sourceforge.net/safeeval.html`.

There is one small problem with the program in its current form, writing to the `stdout` port is buffered by the operating system. This means that you can never be sure when you get data sent to the `stdout` port that the program prints rendering it on the screen, or sent through a pipe. For our line-following robot project, we need to ensure that all data written to the `stdout` port is sent immediately, line by line without delay.

If you close the `listener.py` program and open the `listener1.py` program, you will see that the following code is identical except that all the print statements have been replaced by a function called `flushout('thestringtoprint')`:

```
#!/usr/bin/python

import sys

def flushout(mystring):
    sys.stdout.write(mystring)
    sys.stdout.flush()

def userfunc1():
    flushout ('Data + 10 = ' + str(cmddata[1] +10) + '    ' )

def userfunc2():
    flushout('This is a test ')
```

```
def closeandexit():
    sys.exit('Bye....')


def main():
    while True:
        flushout( 'ok:')
        global cmddata
        try:
            inputstr = raw_input()
            inputstr += ',0,0'          #need at least one item
            cmddata = eval(inputstr,{'__builtins__':None},
                           {'userfunc2':userfunc2,
                            'userfunc1':userfunc1,
                            'close':closeandexit })
            cmddata[0]()                #execute the command
        except (KeyboardInterrupt, EOFError, NameError,
                SyntaxError, IndexError, TypeError) as err:
            if 'KeyboardInterrupt' in str(type(err)): break
            if 'EOFError' in str(type(err)): break
            flushout('Error ')

if __name__ ==  '__main__':

    main()
```

The `flushout()` function uses `sys.stdout.flush` to immediately send data in the `stdout` buffer. The code shown in `listener1.py` will become the code used to handle input/output for the `rbuttons.py`, `rcam.py`, and `rwheel.py` modules of the robot.

If you have any problems with the Python IDLE environment not reflecting user input, try using the program from the command line, or close it and restart Python Shell.

Implementing pipe-based interprocess communication

Pipes are used commonly in the command line and in scripts to send the output data of one program to the input of another, for example, if you type `ls -l` in the command-line prompt, you get an alphabetically-sorted file listing. If you type `ls -l | sort`, you get the listing sorted with directories first. The "|" pipe symbol tells the shell to create a memory-resident connection between the two programs `ls` and `sort`.

Now that we have `listener1.py` working, we need to build a test program to instantiate the program with a pipe in place by performing the following steps:

1. Open `test-pipe.py` in the IDLE 2 IDE and examine the code.
2. Review the key functions of the module in the following table:

Function	Description
`initpipe()`	This uses a `with` statement to send a `stderr` output to a file and then starts a program using `Popen()` (in our case, it will be `listener1.py`) with pipe connections to `stdin` and `stdout`. It reads a line from the `stdout` pipe (this is the first data produced by the `listener1.py` module) and checks whether it contains `"ok:"` in the output string. If it prints a ready message (note that we are still using print functions to send the local output to the console in `test-pipe.py`).
`pingpipe()`	This is similar to `flushout()`; it sends the string we want to print to the remote `stdin` port but then waits to read the response from the remote `stdout` port.
`Main()`	This is the test logic, and it instantiates two copies of `listener1.py`, sends 200 or 2,000 commands (depending on whether or not you are printing) to the child processes, and calculates the number of messages sent per second.

The following is the code for test-pipe.py:

```
#!/usr/bin/python

import sys
from time import time
from subprocess import call, Popen, PIPE, STDOUT

piperef=[0,1]
campipe=0
```

```
i2cpipe=1
printon=False

def initpipe(pipenum, progstr):
    global piperef
    try:
        with open('stderr_' + str(pipenum) + '.txt', 'wb') as err:
            piperef[pipenum]=Popen(['python', progstr],
                                   stdout=PIPE, stdin=PIPE,
stderr=err)
            x = piperef[pipenum].stdout.readline()
            if 'ok:' in x:
                print progstr + ' ready'
                return False
            else:
                print 'Wrong response from child ' + progstr
                return True
    except IOError:
        print 'IOError on init'
        return True

def pingpipe(pipenum, sendstr):
    global piperef
    try:
        piperef[pipenum].stdin.write(sendstr)
        piperef[pipenum].stdin.flush()
        return piperef[pipenum].stdout.readline()
    except IOError:
        print 'IOError on pipe send ', pipenum
        return True

def main():
    if  initpipe(campipe, 'listener1.py'):
        #an Error occured
        print 'Error creating pipe ', campipe
        exit
    if  initpipe(i2cpipe, 'listener1.py'):
        #an Error occured
        print 'Error creating pipe ', i2cpipe
        exit
    start=time()
    if printon:
        loopcount=200
    else:
```

```
            loopcount=2000
        for i in range(1,loopcount):
            x = pingpipe(campipe, 'userfunc2\n' )
            if x == True:
                print 'Error on pipe send ', campipe
            else:
                if printon : print 'Pipe=', campipe, x,

            x = pingpipe(i2cpipe, 'userfunc1,2,3\n')
            if x == True:
                print 'Error on pipe send', i2cpipe
            else:
                if printon : print 'Pipe=', i2cpipe, x,
        stop=time()
        print int(2 * loopcount/(stop-start)), 'transactions per
second'

        piperef[campipe].kill()
        piperef[i2cpipe].kill()

if __name__ ==  '__main__':

        main()
```

Note that even though we are instantiating multiple copies of `listener1.py`, the descriptors are `campipe` and `i2cpipe` since these are what we will eventually use.

The code in the `main()` function sends 2,000 requests from a `userfuncx` command to each instance of `listener1.py` and calculates the number of transactions achieved. The number of message transactions that can be achieved depends on how many processes are running on the computer and the following conditions:

- Whether the transaction is being executed in the IDLE IDE or directly from the command line
- Whether the user is on a remote computer (SSH, PuTTy) or a local one
- Whether or not the controlling program is using print statements

Execution Location	**No print @ 700 MHz**	**No print @ 900 MHz**	**Print @ 700 MHz**	**Print @ 900 MHz**
Desktop IDLE IDE	690	760	7	7
Root Shell (no desktop)	790	870	225	260
LXterminal Shell	780	820	270	280

Execution Location	No print @ 700 MHz	No print @ 900 MHz	Print @ 700 MHz	Print @ 900 MHz
Remote PuTTy session (no desktop, eth0)	**800**	**830**	400	440
Remote PuTTy session (no desktop, wlan0)	**790**	**870**	430	500

Note that in the preceding table, the print statements in the Python IDE are particularly CPU-intensive, and the best performance (highlighted in bold) is via a remote session at 900 MHz with no printing to the `test-pipe.py` console. This means that it takes approximately 1.2 – 1.3 ms to send a command and receive a reply between the processes, providing there is no console printing required. Of course, the processing times for the remote application will extend this time in realtime use.

You can experiment with the `test-pipe.py` and `listener1.py` programs to evaluate their ability to parse commands and produce results.

Note that if the remote application does not emit the right number of lines (too many or too few), then you could get out of sync and the machine could hang. The critical element in using pipes in this manner is ensuring that for every input request, there is only a single-line response.

Test routines for robot modules

For each application used to implement the robot, there is a test routine available. You can use these test routines to help solve problems with each function. The programs are very similar to the `test-pipe.py` shown earlier. The test programs are as follows:

- `test-buttons.py`: This flashes the LED on and off to calculate the transaction rate and then waits for button activity. The button GPIO(21) will shut down the computer, and GPIO(20) will exit the test program. This instantiates `rbuttons.py` as the root.
- `test-camera.py`: This does 100 `line0` requests to `rcam.py` to test the transaction rate. You can turn on the camera preview window to show the image data if you have a local display.
- `Test-wheels.py`: This does 100 `servowr,0,450` commands to `rwheel.py` to calculate the transaction rate.

You can download all the test programs from the Chapter 8 folder at `http://1drv.ms/1ysAxkl`.

Testing rbuttons.py

Now that you've downloaded the files, complete a test cycle of `rbuttons.py` by performing the following steps:

1. Open IDLE again (non-privileged).
2. Open `test-buttons.py` in the Python IDE. This application is very similar to the `test-pipe.py` program and completes the following tasks:
    - It initializes the pipe (which instantiates the child program) and sends 200 or 2,000 commands to turn the LED on and off
    - It calculates the transactions per second
    - It monitors for switch events
    - It uses the switch on GPIO(21) to initiate a shutdown of the Raspberry Pi
3. Review the key functions in the following table:

Function	Description

`initpipe()`	One small but significant change in the definition is that the `Popen()` function now starts the child application using `sudo python rbuttons.py`, as shown in the following code fragment that gives it root privileges:
	`with open( progstr + '_stderr_' + str(pipenum) + '.txt', 'wb') as err:`
	`  piperef[pipenum]=Popen(['sudo', 'python', progstr],`
	`  stdout=PIPE, stdin=PIPE, stderr=err)`

`pingpipe()`	This is identical to that used in `test-pipe.py` and writes a string to the child `stdin` and reads from the child `stdout` port.

Function	Description

`Main()`	This is very similar to the one used in `test-pipe.py`. The transaction calculations are done with the `as` loop turning the LED on and off, and then using the following code to sense the switches and perform actions:
	`#Wait for switch action`
	`while True:`
	`x = pingpipe(digio, 'rdswevents\n' )`
	`if x == True:`
	`print 'Error on pipe send ', digio`
	`else:`
	`if 'ok:' in x and x[11] == ']':`
	`y=eval(x[0:12])`
	`if y[0] :`
	`z=Popen(['sudo', 'shutdown', '-h', 'now'])`
	`print 'Shutdown in process'`
	`break`
	`if y[1] : break`
	`else:`
	`print 'Return string error '`
	`sys.exit('Closing application')`

Notice that if the return string from the child contains "`ok:`" and has a "`]`" in the eleventh position of the string (we assume that the string is ok), it is then built into a list (`y=[]`) using `eval()`. We can now test each switch flag. `y[0] = gpio(21)` and `y[1] = gpio(20)`. We use `y[0]` to initiate a shutdown and `y[1]` to exit the program.

1. Familiarize yourself with the code in the IDE.
2. Close the IDE.
3. Execute the application from the command line using Python's `test-buttons.py`. If you set the executable flag on the program file, you can start it with `./test-buttons.py`.

Note that you no longer need to be concerned with the root privileges requirement. Providing that the user identity you use can use root privileges, the application will instantiate the `rbuttons.py` with the correct privileges to use the GPIO functions.

Testing rcam.py

The `test-camera.py` program is very similar to our other test programs. The program initializes the pipe (which instantiates the child program `rcam.py`) and sends 100 `line018` commands to capture frames and calculate the number of frames per second that can be processed. Perform the following steps to test `rcam.py`:

1. Open `test-camera.py` in the Python IDE.
2. Review the key functions in the following table:

Function	Description
`initpipe()`	This initializes the pipe and the in-memory buffer to capture frame data.
`pingpipe()`	This is identical to that used in `test-pipe.py`, writes string to the child `stdin`, and reads from the child `stdout`.

`Main()`	This is very similar to that used in `test-pipe.py`. The transaction calculations are done with the `as` loop capturing frames:
	`loopcount=100`
	`for i in range(1,loopcount):`
	`x = pingpipe(camref, 'line018\n' )`
	`if x == True:`
	`print 'Error on pipe send ', camref`
	`else:`
	`if printon : print 'Pipe=', camref,`
	`x,`
	`stop=time()`
	`print str(loopcount/(stop-start)), 'frames per second'`
	`sys.exit('Closing application')`

Testing the performance of rcam.py

Run the following tests to assess the performance of your `rcam.py` module:

- The cost of data:
    1. Set the command used for frame capture to either `line0` or `line018` in `test-camera.py`.
    2. Assess the cost of data sent back to the parent process.
- The cost of preview feature:
    1. Set `campreview=False` or `campreview=True` in `rcam.py`.
    2. Assess the cost of turning the PiCam preview feature and off.
- The benefits of higher clock speed:
    1. Set the CPU clock speed to 900 MHz using `set-perf +`.
    2. Assess whether higher clock speed is beneficial.

Potential test results

By understanding the implications of the various options, you can potentially achieve 7.5 frames per second in IDLE Shell at 700 MHz and close to 9 frames per second from the command line at 900 MHz.

Testing rwheel.py

As with our other modules, we will run a test program for `rwheel.py`; to do this, perform the following steps:

1. Open the Python 2 IDLE IDE (non-privileged).
2. Open `test-wheels.py`.

This application is very similar to the `test-pipe.py` program. The program initializes the pipe (which instantiates the child program), sends 1000 commands to `servowr,0,450`, and calculates the transactions per second.

3. Review the key functions in the following table:

Function	Description

`initpipe()`	One small but significant change in the definition is that the `Popen()` function now starts the child application using `sudo python rwheel.py`, as shown in the following code fragment, which gives it root privileges:
	`with open( progstr + '_stderr_' + str(pipenum) + '.txt', 'wb') as err:`
	`  piperef[pipenum]=Popen(['sudo', 'python', progstr], stdout=PIPE, stdin=PIPE, stderr=err)`

`pingpipe()`	This is identical to that used in `test-pipe.py`, writes strings to the child `stdin`, and reads from the child `stdout`.
`Main()`	This is very similar to that used in `test-pipe.py`. The transaction calculations are done with the `as` loop setting a servo value.

Notice that you no longer need to be concerned with the root privileges requirement for `rwheel.py`. Providing the user identity you use can use root privileges, the application will instantiate `rwheel.py` with the correct privileges to use the `smbus` functions.