Student Robotics Programming Reference

October 13, 2008

Abstract

This document details the programming interface for Student Robotics robots. For more information on programming in Python and about how to bring all this together see the tutorial.

1 Directory structure

The robot source code must be in a zip file named `robot.zip`. This zip file must act as a Python module, and so it must contain a file named `__init__.py`. The zip file must contain a second file named `robot.py`. Other `.py` files can be included to improve code organisation. The checkout function of the RoboIDE system creates zip files in this format.

1.1 __init__.py

This file is used so Python can identify the zip file as a compressed module. It should be empty.

1.2 robot.py

This file must at least contain the main function for the robot control algorithm.

1.3 File structure

1.3.1 Imports

All files containing parts of the robot control algorithm must import the Student Robotics interface components with: `from sr import *`. Other standard Python modules can be imported as required.

2 Main function

In `robot.py` there must be a function with the form:

```
def main(corner, colour, game):
```

When the robot is started, this function will be called.

corner This is a number from 0 to 3 that identifies the corner the robot starts in. The number is the colour number of that corner.

colour The colour of the tokens that the robot is looking for.

game The game being played. 0 for collect any token, 1 for collecting only tokens of a specific colour.

3 Yielding control and interrupts

Once the main function is called, the algorithm has control of the robot. The algorithm must `yield` control of the robot once it has processed any incoming data, and configured any outputs accordingly. The `yield` command passes control of the robot over to the Student Robotics environment. When an event occurs, the environment will pass control back to the algorithm at the instruction after the `yield` statement. When control is passed back to the algorithm the `event` variable has been set to contain information about the event that just occurred.

3.1 Using `yield`

3.1.1 Timeouts

The `yield` statement can be used to wait for a set amount of time. For example:

```
yield 4
```

Causes the program to be delayed for 4 seconds.

3.1.2 Waiting on events

The `yield` statement can be used to wait for an event to occur. For example:

```
yield io
```

Causes the algorithm to wait until a dio (digital IO) event is received.

3.1.3 Combinations of events and timeout

The `yield` statement can be used to wait on a combination of events and a timeout, whichever occurs first:

```
yield io, 5
```

Causes the algorithm to wait until a dio event is received or 5 seconds, whichever comes first.

3.1.4 Calling subroutines

Control of the robot can be passed to subroutines, allowing the program to be split up into easier to understand parts. The `yield` statement can be used to pass control of the robot to a subroutine. When the subroutine returns, the program continues from the yield statement. If more information is passed to yield after the name of a subroutine, then that information is passed as argument(s) to that subroutine. For example:

```
yield findTokens, 3
```

Causes control of the robot to be passed to the findTokens subroutine with the argument 3.

3.2 The `event` variable

The `event` variable is set when control is returned after a yield. It contains information regarding the event that caused control to be passed back.

3.2.1 Finding the event source

The `event` variable will return true when compared to the event source that led to its creation:

```
yield io, 5 #Wait until a dio event or 5 seconds passes

if event == io:
    pass #A io event happened first (pass means do-nothing)
else:
    pass #A timeout is the only other option
```

3.3 IO Events

IO events are caused by a change in the digital inputs of the Joint IO board. After a io event causes a `yield` statement to return control of the robot to the control algorithm, the `event` variable will be set. The io inputs that have changed are contained in a dictionary that can be accessed from `event.pins`:

```
yield io, 5

if event == io:                 #Is it an io event?
    if 0 in event.pins:         #Has input 0 been changed?
        if event.pins[0] == 1: #Is input 0 now a 1?
            pass
```

3.3.1 Monitoring specific pins

Sometimes you will want to only wait for events on specific pins. This can be done by using `setsensitive` and `removesensitive` functions:

```
removesensitive(2) #Ignore pin 2 for now
yield io, 5

#Do stuff!

setsensitive(2) #Resume monitoring pin 2
```

3.4 Visual Events

When the `vision` event source is passed to the `yield` command, the robot will grab an image with its webcam and start to process that image. When this processing has finished, the vision system will raise an event (whether a token is seen or not). Any tokens found are placed in the list `event.blobs`. Each blob in this list has a set of properties:

colour The colour of the blob

mass The number of pixels that make up the blob

centrex The number of pixels between the left of the image and the centre of the blob.

centrey The number of pixels down from the top of the image to the centre of the blob.

```
yield vision    #This will always return as soon as
                #a frame has been processed

for blob in event.blobs:    #Go through each blob in
                            #turn. This does nothing if
                            #there are no blobs
    if blob.colour == 0:    #For each blob, check its colour
        pass
```

4 Controlling the robot

4.1 Controlling the motors

The motor controller has two outputs. The `setspeed` command sets the direction of each motor and the power delivered to each one.

setspeed(n) This sets both of the motors to n% of maximum power. $-100 < n < 100$. Negative values of n drive the motors backwards.

setspeed(n, m) This sets the power delivered to motor 1 to n%, and the power delivered to motor 2 to m%. $-100 < n, m < 100$.

4.2 Controlling the PWM board

The Pulse Width Modulation board is used to control servo motors. It is controlled with the `setpos` function that takes two arguments:

```
setpos(n, m)
```

This will set PWM board output $0 < n < 5$ to the position $0 < m < 100$.

4.3 Controlling the JointIO Board Outputs

The outputs of the JointIO board can be controlled using the `setoutput` function. This takes two arguments:

```
setoutput(n, val)
```

This will set output $0 < n < 4$ to the value val, which is either 0 or 1. If value is 1 then the output will be 3.3V, otherwise it will be 0V.