MAKE Technical Documentation

Carston Wiebe

2024.10.22

MAKE is a Python-based embedded systems library designed for University of Nebraska-Lincoln's PROTO recognized student organization. It is mostly a wrapper around the CIRCUITPython library that provides methods and objects that simplify CIRCUITPython so that it can be easily taught to elementary and middle school students.

INTRODUCTION

MAKE is intended for use on the MakerPI RP2040 board and in conjunction with PROTO's robotics components. Its goal is to provide a 'pseudo programming language' that enables elementary and middle-school aged students to code a simple robot with no prior knowledge of Python nor any other coding language. It does this by implementing its own versions of functions and classes from both the standard library and from CIRCUITPython.

PHILOSOPHY & STANDARDS

A minimum amount of knowledge on coding should be assumed and expected, and the users may not have access to a proper IDE with features like intelligent syntax highlighting or auto-complete. As such, there are two standards to use depending on whether code will be outward facing– to be used by students– or inward facing– to be used by maintainers.

End users (students) should not have to import any library other than MAKE. If features from other libraries are needed, provide an implementation of that feature within MAKE (e.g. `sleep` from the `time` library is available within MAKE as `wait`).

For outward ('public') classes, functions, and variables:

- Prioritize short, easy to spell names– ideally one word
- Use `flatcase` as opposed to `snake_case`.
- Limit the number of function/constructor arguments
- Make lines of code read like English; e.g. `until(button.pressed)`
- Remove complexities whenever possible

For inward ('private') classes, functions, and variables you can follow standard Python form for the most part:

- `lower_snake_case`
- Each class gets its own file
- Limit lines to 80 characters when possible
- Use type hints for both function arguments and return types
- For functions and variables that are not meant to public, prefix them with __ to prevent them from being interfered with.

Lastly, there are some rules required for compatibility with CIRCUITPython:

- Not all Python libraries are available on the MakerPI; be careful when adding new imports. Custom implementations of some existing Python features may have to be done in-house.

USING MAKE

All MakerPi's should have MAKE installed on them by members of PROTO before being given to students. This can be done by plugging the MakerPi into a computer using a microUSB cable and viewing its contents like you would a USB drive. Copy the contents of the `src/lib` directory from the MAKE Github into the `lib` directory on the MakerPi. This is also the same method by which code will be uploaded to the MakerPi by students; their final program should be saved to a file named `main.py` or `code.py` and then copied into the root of the MakerPi (*not* inside `lib`) via microUSB.

PROGRAM STRUCTURE

All MAKE programs should follow the same basic three-part structure.

```
# first, all programs *must* start with `import make`
import make

# second, each component is given a name
name = make.compnent(port)
name = make.compnent(port)
name = make.compnent(port)

# lastly, actions are made using the named components or
#  using the prefix `make`
name.action()
name.action(option)
name.action(option, option)
make.action(option)

# this is also where 'control flow' statements are used
while name.action():
  if name.action():
    name.action(option)
  else:
    name.action(option)
```

GLOSSARY

Nearly all of MAKEs features are in the form of classes, but some orphaned functions also exist. The complete list of public classes, functions, and variables found in MAKE can be seen below.

Note that all function arguments (the values within the parenthesis after actions) do not have to be named; `make.button(port=1)` and `make.button(1)` are identical statements.

ORPHANED FUNCTIONS & VARIABLES

The following functions and variables do not belong to any class.

```
make.wait(seconds)
# pauses the program on the current line for the given number of
#  seconds

make.until(condition)
# pauses the program on the current line until the given
#  condition is true
# most likly to be used with buttons
```

```
#  e.g. `make.until(button.pressed)`
# note that the condition used eschews the usual parenthesis
#  attached to an action-- it's `until(button.pressed)` not
#  `until(button.pressed())`

make.reversed
# variable that equals -1
# used when reversing motors
```

BUTTONS

`buttons` can be made on ports eight and nine (the built in buttons) and ports one through five (the GROVE ports).

```
b = make.button(port)

b.pressed()
# returns whether or not the button is pressed
```

MOTORS

`smallmotors` can be made on ports one through five (the GROVE ports) and `largemotors` can be made on ports six and seven.

The following actions are the same for both `smallmotors` and `largemotors`.

```
m = make.smallmotor(port)
m = make.smallmotor(port, direction)
# direction can be 1 or -1 (defaults to 1), and `make.reversed`
#  equals -1

m.spin(power)
m.spin(power, seconds)
# spins the motor at the given power for the given time. if no
#  time is given, sets the motor spinning and immediately moves
#  to the next line without stopping it
# power is a number ranging from -100 to 100

m.stop()
# stops the motor
```

DRIVETRAINS

`drivetrains` are made from either two `smallmotors` or two `largemotors` and represent a two-wheeled 'cart'.

```
d = make.drivetrain(left_motor, right_motor)
d = make.drivetrain(left_motor, right_motor, direction)
# direction can be 1 or -1 (defaults to 1), and `make.reversed`
#  equals -1

d.drive(power)
d.drive(power, seconds)
# spins both motors at the given power for the given time. if no
#  time is given, sets the motors spinning and immediately moves
```

```
#  to the next line without stopping them
# power is a number ranging from -100 to 100

d.turn(power)
d.turn(power, seconds)
# spins both motors at opposite powers for the given time. if no
#  time is given, sets the motors spinning and immediately moves
#  to the next line without stopping them
# power is a number ranging from -100 to 100

d.curve(left_power, right_power)
d.curve(left_power, right_power, seconds)
# spins both motors at different powers for the given time. if no
#  time is given, sets the motors spinning and immediately moves
#  to the next line without stopping them
# power is a number ranging from -100 to 100

d.stop()
# stops both motors
```

EXAMPLE PROGRAMS

TEMPLATE WITH BLANKS

A header for a program for a robot that has one `button`, one `smallmotor` for an arm, and two `largemotors` for a drivetrain.

```
import make

_________ = make.button(port=_)
_________ = make.smallmotor(port=_)
left = make.largemotor(port=6)
right = make.largemotor(port=7)
_________ = make.drivetrain(left, right)

# make your actions...
```

COMMON LOOP MISTAKE

Demonstration of a common mistake when using MAKE.

```
import make

stopbutton = make.button(9)
motor = make.smallmotor(1)

while not stopbutton.pressed():
  make.wait(1)
  motor.spin(100, 1)
```

The code feels like it *should* alternate between spinning the motor for one second and waiting for one second, and should stop whenever the stop button is pressed. However, programs only run **one line at a time**.

This code will

1. Import make
2. Name the components
3. Check if the stop button is pressed
4. Pause
5. Spin the motor
6. And *then* check if the button is pressed again before looping

The button is not checked while the motor is running or while the program is waiting, so pressing it won't do anything unless you are holding the button down when the program reaches the end of the loop.

A better way of implementing this program that checks the button once every tenth of a seconds is seen below.

```
# this time, repeat the loop every 0.1 seconds instead of every
#  1 second-- checking the button x10 as often
# 10 loops = 1 second

count = -10  # increased each loop

# while count is negative, keep the motor stopped
# while count is positive, spin the motor
# when count reaches 10, reset it back to -10

while not stopbutton.pressed():
  make.wait(0.1)
  if count == 10:
    count = -10  # reset the loop
    motor.stop()  # stop the motor
  else:
    count = count + 1  # move forward in the loop
    if count > 0:
      motor.spin(100)
```

For most purposes however, the following code will work just fine.

```
motor.spin(100)
make.until(stopbutton.pressed)
motor.stop()
```