

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Artifact Information** | | | | | | | | |
| **Artifact ID** | | **Artifact Title** | | | | | | |
| KCS-001 | | Kivy Clock Summary | | | | | | |
| **Capstone Team** | | | | **Revision** | | **Artifact Date** | | |
| Capstone Team 27 - Granustem | | | | 1.0 | | Apr 13, 2019 | | |
| **Prepared by** | | | | **Checked by** | | | | |
| Ben Alexander | | | |  | | | | |
| **Revision History** | | | | | | | | |
| **Revision #** | **Date** | | **Prepared by** | | **Checked by** | | **Description** | **Approved by** |
| 1.0 | Apr 13, 2018 | | Ben Alexander | |  | | Initial Version | Reese Bastian |

1. Purpose

The purpose of this artifact is to describe the implementation of the Kivy Clock module used within our program in order to trigger interrupts in the code during which readings from the sensors could happen. The artifact will review some of the difficulties that we faced while implementing, as well as plans for how it could be improved.

2. Documentation

A comprehensive set of documentation for the Kivy Clock can be found with the following link: <https://kivy.org/doc/stable/api-kivy.clock.html>

3. Implementation, Woes, and Future Plans

The need for a clock that could trigger time-based interrupts was crucial to our project, because we needed a way to periodically gather data from the sensors. As one of our main goals was to increase the number of samples per second, it was important to find a clock that could handle this for us.

The Kivy clock has four different options – default, interruptible, free, and free only. The default clock can only create as many interrupts per seconds as there are frames per seconds (fps) being updated. Because the default fps for Kivy is only 60, we knew this wouldn’t be enough, because the best data rate we could get would be 60 samples per second. We set out to try using the interruptible clock instead, because it allows you to set an interrupt interval much higher than whatever the fps is. Note that you can read a lot more about the different clock types above. In the end, we hit many different struggles trying to get the interruptible clock to work (it always ran way faster than we wanted which caused major issues). So, instead, we just used the default clock but raised the fps for Kivy to a much higher number (250 fps). This comes with drawbacks, though, as this high fps brings with it a lot of unnecessary overhead, and we are suspicious that it could cause the program to crash from time to time. We definitely feel that the next team who works on this project should try to switch from the default clock to the interruptible clock and optimize. They could also try using an entirely different interrupt method as well that is not native to Kivy.

To understand how the clock is used, here are a few locations in the code to look at.

In GranuSoft/src/main.py, we include the following lines (they are not right next to each other, but should be easy to find):

from kivy.config import Config as KivyConfig

CLOCK\_TYPE = "default"

KivyConfig.set('kivy', 'kivy\_clock', CLOCK\_TYPE)

KivyConfig.set('graphics', 'maxfps', 250)

The first line of code imports the library necessary to modify the raspberry pi image’s Kivy configuration file. The second line simply creates a constant that indicates what clock we will use (default, interruptible etc.). The third line actually takes the constant value and sets the internal Kivy clock equal to the proposed setting. The fourth line is what is used to modify the fps in the Kivy configuration file. Note that you will need to run the script TWICE for any changes to take effect, as these changes will modify the Kivy configuration file, but the changes won’t exist in the program until the next time you run it. So, anytime you change these values, run the program twice.

We actually use the clock in several locations, but one of the easiest places to notice it is in GranuSoft/src/view/screens/main/LiveFeedScreen.py. In the on\_pre\_enter() function (which is called right before the screen is loaded) notice the following line:

self.event = Clock.schedule\_interval(self.update\_values, INTERVAL)

This line creates a class variable called event that represents a clock interrupt. The first parameter to the Clock.schedule\_interval function is the callback function, which we called update\_values(). The second parameter is the interval at which you’d like the interrupt to occur, in seconds (i.e. a value of .5 would call the callback function every half second).

Notice the following line that exists in the same file, in the on\_leave() function (which is called as we are leaving the screen):

self.event.cancel()

This line cancels the event that was created in on\_pre\_enter(), so that the interrupts won’t keep happening even after we’ve left the screen.

While used in other places as well (such as in the TestInProgress Screen), the techniques described above are the way the project used Kivy Clock throughout.