VectorOS User’s Guide

Table of Contents

Introduction

The Hackaday 2023 Supercon Badge has a sophisticated Raspberry Pi Pico RP2040 running MicroPython along with some sweet analog hardware. What will you do with it? To simplify your hacking, we've provided some example programs that use a simple framework called VectorOS.

VectorOS lets you cooperatively multitask (using MicroPython's *asyncio* library). It also allows you to read the keyboard, set the LEDs, and draw on the screen. There's a system to easily create menus as well. You can also set timers for one shot or repetitive timed tasks.

Creating a program using VectorOS involves a few steps:

- Configure VectorOS to know about your program
- Provide an entry point for the OS to run your code
- Use services and data provided by the OS
- Be sure your code yields to other tasks regularly

In addition, you can create "slots" that generate analog signals and then, with the help of some external wires, show them on the oscilloscope screen.

Important Note

Some of the files mentioned in this document are not on your badge or may be slightly out of date. Check the GitHub for the latest versions or if there are files mentioned herein that you don't see on your badge. (https://github.com/Hack-a-Day/Vectorscope)

Quick Start

If you want to jump right in, grab a copy of template.py from GitHub. Copy it to your Badge and edit it (we suggest mpremote (https://docs.micropython.org/en/latest/reference/mpremote.html) or Thonny (https://thonny.org/) to talk to the Badge). The comments in the example code (see GitHub; https://github.com/Hack-a-Day/Vectorscope) and some browsing should allow you to get something running quickly. If you didn't read the configuration section below, you should do that now. It's short, we promise. If you want to make oscilloscope traces, you might want to do a quick skim of Working with Waveforms, also below. Be sure to check out *template.py* for full programs and *template_slot.py* for waveforms, too.

If you want to configure what programs VectorOS can run, look in *vos_launch.py*. The menus are in *menudemo.py*.

Once you've had a taste, you can return here and get the details about how VectorOS can best help you work with the screen, keyboard, menus, and timers. It also manages starting your programs in the right environment.

Configuration (vos_launch.py)

You configure your VectorOS setup by changing the file vos_launch.py. Note that all the items here are strings or numbers. Don't use class names or import any packages in this file. Here are the things you must set:

Items in vos_launch.py

Item	Description
launch_list={ "key1": "command1", "key2": "command2",...}	Keys that refer to modules for launchable programs
auto_launch_list=["key1", "key2", …]	Keys from *launch_list* to launch on startup
auto_launch_repl=True	True to launch the onboard repl (to exit it, enter these command from the repl: Import sys sys.exit()
key_scan_rate=100	Milliseconds to scan keys. Zero if you don't want VectorOS to manage the keyboard scanning
gc_thread_rate=1200	Milliseconds between garbage collection. Set to zero for default garbage collection or to manage it yourself (see also, *vos_state.gc_suspend*)
debug_level=DEBUG_LEVEL_SEVERE	The *vos_debug.debug_print* function will only output messages marked at the indicated level or lower. Common values (defined in *vos_debug*) are: DEBUG_LEVEL_SILENT -1 DEBUG_LEVEL_SEVERE 0 DEBUG_LEVEL_ERROR 10 DEBUG_LEVEL_WARNING 20 DEBUG_LEVEL_INFO 30

You do not execute this file, but you should execute *vectoros.run()* (you can call it from *main.py*, if you like, but probably should not call it from *boot.py*). If you want to be able to "execute" this file, you can add this to the bottom:

```
if __name__=="__main__":
      import vectoros
      vectoros.run()
```

Here's an example file that loads several examples and runs menu and blinky on startup:

```
# list of things to launch from vectoros

# dictionary of tags to imports (will look for .vos_main() or .main())
launch_list={ "menu": "menudemo", "blinky": "blinker", 'sketch': 'sketch',
"screen": "screentest"}

# list what you want to start auto (maybe just one thing?) need tag
auto_launch_list=["menu", "blinky"]

auto_launch_repl=True

key_scan_rate = 100     # how often to scan the keyboard globally (ms; 0 to
do it yourself)

# how often to garbage collect
# if you set this to zero and do nothing else
# garbage collection will be automatic as usual and before new tasks launch
gc_thread_rate = 1200

# Base rate for the timer (ms)
timer_base_rate=100

# Debug level (messages must be < this level to print)
# That is, at level 0 only level 0 messages print
# at level 1 then level 1 and level 0 messages print
# Set level to -1 to stop all messages (assuming you only call debug_print
with positive values)

# if you want to use symbols for debug level, these are defined in
vos_debug:
DEBUG_LEVEL_SILENT=-1
```

```
DEBUG_LEVEL_SEVERE=0
DEBUG_LEVEL_ERROR=10
DEBUG_LEVEL_WARNING=20
DEBUG_LEVEL_INFO=30

debug_level=DEBUG_LEVEL_SEVERE


if __name__=="__main__":
      import vectoros
      vectoros.run()
```

Structure of a VectorOS Program

Typically, a VectorOS program will contain, at least, an *async vos_main()* function which will be called at startup. You can make the function synchronous (that is, leave off the *async*), but this can cause memory leaks and other problems. If there is no *vos_main()* function, the OS will attempt to load your *main()* function. Again, this can be synchronous or asynchronous, but if it is your actual entry point, you should prefer async.

By convention, the *vos_main()* function will be your asynchronous entry point, and then you can use *main* as the synchronous entry point for cases where you want to run without the OS (e.g., for testing). The main can start up any support you need (like the timer or keyboard loop) and then launch *vos_main()* using *asyncio.run*. It is possible for VectorOS to launch a synchronous *main* or *vos_main*, but it is generally unreliable. Try to use an async function if at all possible.

Programs can use at least two different techniques. It is possible to have *vos_main* (or something it calls) loop forever, but use an *asyncio.sleep()* (or *sleep_ms()*) call to allow other tasks time to run (see the template below). You can also set up keyboard and timer callbacks and then just loop and pause forever (see the program simple.py in the examples section). You can also, of course, do both.

It is a good idea to allow a way for other programs to pause and exit your main loop. When exiting, clean up your entry in the task table with *vectoros.remove_task()*. The basic template (template.py) shows all of these techniques:

1) Synchronous main
2) Async *vos_main*
3) Launching the keyboard loop when not using VectorOS
4) Pausing and exiting the main loop
5) Removal of the task on exit

```
# Use this template to start your VectorOS program:
```

```
import vectoros
import asyncio
import keyleds
import keyboardio

task_name="Template"  # use this name in vos_launch.py, too
_run_every_ms=500   # how often to loop

_freeze=False
_exit=False


# Outsiders can call this to exit you
def exit():
      global _exit,task_name
      _exit=True
      vectoros.remove_task(task_name)

# Outsiders can call this to pause you
def freeze(state=True):
      global _freeze
      _freeze=state

async def vos_main():
      global _freeze, _exit, _run_every_ms, task_name
      _freeze=False
      _exit=False
# if you want to control keyboard and LED without running the whole OS
      if vectoros.vectoros_active()==False:
      keyboardio.KeyboardIO.run(250)
      while _exit==False:
      if _freeze==False:
            print("Do something here")
      await asyncio.sleep_ms(_run_every_ms)
      _exit=False  # reset for next time
      vectoros.remove_task(task_name)   # make sure we are removed (also
does this in exit)

def main():
      asyncio.run(vos_main())
# code here will never, ever run under VectorOS
```

```
# if you want the possibility to run directly without VectorOS
#if __name__=="__main__":
#     main()

# if you want to configure to run in vos_launch but still run this file
# add the main function to to vos_launch.py and use this:
if __name__=="__main__":
      vectoros.run()
```

In addition, there is code at the bottom that, if uncommented, would allow you to run without VectorOS. However, the code that is uncommented has a different purpose. It allows you to run this file and start VectorOS. If this program isn't configured to automatically launch, that won't actually run the program.

In summary, each program VectorOS knows about is configured in *vos_launch*. Programs can be launched automatically or by programs using, for example, a menu or button callback. In addition to normal programs, there are also Vectorscope slots that generate analog signals and displays them. These can be launched from buttons A, B, C, D, or under program control. When slots are running VectorOS releases the screen, stops managing garbage collection, and the Vectorscope component controls the system.

Tutorial

Want to write Hello World for the badge? Here's the steps we'll take.

1. Create a program using template.py
2. Add our program to the list of programs in vos_launch.py
3. Add a menu item to launch our program.

We will assume you know how to use MicroPython using Thonny or mpremote. If not, maybe watch a YouTube before you get started. The tutorial program will show the word "Hello" on the screen and then count up periodically. You'll press the Menu button to exit. Code you need to change or modify will be marked with a sticky note.

Step 1. Create Program

Create a file called hello.py and copy the contents of template.py into it. Change the *task_name* variable near the top of the file to contain *hello* instead of *Template*. Be sure *hello* is all lower case. You can change the update rate in *_run_ever_ms* if you like (500 is a half second). You'll notice there are imports for some VectorOS services already present. The top portion of the file should look like this:

```
# Use this template to start your VectorOS program:
```

```
import vectoros
import asyncio
import keyleds
import keyboardcb
import timer
from vos_state import vos_state

task_name="hello"  # use this name in vos_launch.py, too  ADD THIS LINE
_run_every_ms=500   # how often to loop        OPTIONAL: CHANGE IF YOU LIKE
```

There are several global variables already defined:

```
_freeze=False
_exit=False
_menu_key=None
```

These variables are important:

- *_freeze* - if *True*, the main loop will continue but not do anything
- *_exit* - if *True*, the main loop will exit and the program will end, returning to the main menu
- *_menu_key* - This will hold the keyboard handler for the Menu key which needs to be shut down when the program exits

There are associated functions *exit()* and *freeze()* that set their respective variables and, in the case of *exit()* cleans up anything necessary.

Next is the *vos_main()* function, which is the normal entry point for the program. The line responsible for calling *exit* is there:

```
      _menu_key=keyboardcb.KeyboardCB({keyleds.KEY_MENU: exit})
```

Note that the function has a dictionary with one entry in it. If you wanted to handle more keys, you could simply add their names (from *keyless*) to the dictionary and then provide a callback name. You can have a single callback for each key, or point multiple keys at the same callback function.

In addition, you can add new keyboard handlers if you prefer. For example, you might want a different handler that allows keys to repeat or that knows how to combine multiple keys from the joystick into one synthetic key (e.g., JOY_N and JOY_E to JOY_NE). The keyboard handler here does not repeat at all. It is not uncommon to have multiple keyboard or joystick objects in a single program. However, be sure to clean up all the keyboard objects before you exit. The template does this is the *exit()* function, although you can also use Python context management to make this more automatic, if you like.

This will cause the *menu_key* function to run when you press Menu. The program loops every half second. Find the line that says:

```
      print("Do something here")
```

Replace that line with:

```
      hello()          # this replaces the print statement
```

Now we need two functions: *hello* and *menu_key*. Put these above *vos_main*:

```
count=0                 # Add this entire block of code above vos_main

def hello():
      global count
      screen=vectoros.get_screen()
      screen.clear(0)
      screen.text(40,120,f"Hello {count}!")
      count+=1
# End of block to add
```

Some of the code in the template allows you to run without VectorOS for testing purposes. If you want to make the file directly executable, you can uncomment the code at the bottom and comment out the existing code that runs VectorOS instead. We suggest not making any of the changes below. That way when you run your program, VectorOS will start (this is the default). However, during development, it can be useful to just run the program and this is how you do it.

```
# if you want the possibility to run directly without VectorOS
#if __name__=="__main__":      # uncomment these two lines to run
stand-alone
#     main()
# changes here are optional - recommend to leave it as is
# if you want to configure to run in vos_launch but still run this file
# add the main function to to vos_launch.py and use this:
if __name__=="__main__":       # If you uncomment above comment these two
      vectoros.run()
```

Note that the template has special code that allows the keyboard, joystick, and timers to work without VectorOS started and some VectorOS services are not available like this, so in many cases, you are just as well off leaving it as the default. This program, however, will work either way. Even if you didn't change the comments, you can always run your program from the MicroPython repl. Just type:

```
import hello
```

```
hello.main()
```

This assumes that your program doesn't require VectorOS running. If it does, you may be able to start it from the built-in repl (turn it on in the Advanced menu). Or, you may have to complete the next steps and test in the full environment. For this program, though, it should work fine.

If it doesn't work, you can use *print* statements to trace what's going on. Once it works, you are ready to go to step 2, telling VectorOS how to find your program.

Step 2. Configuring VectorOS

This is very simple. Open up vos_launch.py. Inside there is a dictionary called *launch_list.* Add an entry with your program's identifier (hello; the same one we used in *task_name*) and the module name (in this case, *hello*). The line should look something like this (you may have different programs; the key is to add the "hello" item there. :

```
launch_list={ "menu": "menudemo", "blinky": "blinker", 'sketch': 'sketch',
"screen": "screentest", "demo": "examples", "hello": "hello"}
```

Step 3. Launch!

Your other programs can run your new program, but it is handy to link it to a menu item. The menu is controlled by a file called menudemo.py. Open it up and you'll see several variables that hold the main menu and submenus.

Some menus dynamically change and that means the software needs to be able to find those items. That means inserting a menu item before a dynamic menu item may require further changes. For example, in the default main menu, the Blinker menu item changes to say Start Blinker or Stop Blinker. So as not to interfere with that, we'll put our Hello item just below that item. Here's what your main menu in menudemo.py should look like:

```
mainmenu=[["Demo",run_demo,None],
                        ["Sketch", runsketch, 0],
                        ["Badge", runbadge, 0],
                        ["??? Blinker",blinkrun,(0,amenu)], # Note this must
be in this position or you will have to change menu_custom above
                        ["Hello",hellorun,0],
                        ["GFX",gfxdemo,0 ],

                        ["Freeze",freeze,0,"async"],
                        ["Advanced>",SUBMENU,submenu]]
```

This tells the menu system to display the word Hello and when you select it to run the *hellorun* function which we need to write. The argument will receive a zero as an argument, but we won't use it anyway. Here's the function:

```
def hellorun(arg):                          # enter this OR use the next box
      vos_state.show_menu=False
      vectoros.launch_task('hello')
      return EXIT
```

However, you can also use the *launch* keyword instead. Then you don't need *hellorun* at all. If you want to do it that way, go back to the *mainmenu* definition line and change the Hello entry like this:

```
mainmenu=[["Demo",run_demo,None],
                        ["Sketch", runsketch, 0],
                        ["Badge", runbadge, 0],
                        ["??? Blinker",blinkrun,(0,amenu)], # Note this must
be in this position or you will have to change menu_custom above
                       # ["Hello",hellorun,0],        # remove
                         ["Hello",launch,'hello'],    # add
                        ["GFX",gfxdemo,0 ],

                        ["Freeze",freeze,0,"async"],
                        ["Advanced>",SUBMENU,submenu]]
```

Now run vectoros.py or any program (like menudemo.py) that is configured to start VectorOS. You can select your program from the menu and watch it run. Press Menu when you want to exit.

What's Next?

You did it! You wrote a program that runs on the Vectorscope badge! This kind of looping program is common although you can also set up a program to run with timers. VectorOS starts the timer system automatically, but if you want to run independently, you'd need to start the timers just like this program started the keyboard loop (with *run*) if it detected that VectorOS wasn't active.

If you want to get started writing Vectorscope waveforms, see Working with Waveforms, below.

VectorOS Services

There are several VectorOS services available to programs. The *sleep_for_ever* and *sleep* methods are special cases. If they are called from code that is already in asynchronous mode, they may hang. For asynchronous code, you can use *await asyncio.sleep()* or

asyncio.sleep_ms() in async methods. For more information about asynchronous programming, see the AsyncIO section below.

VectorOS Simple API (vectoros.py)

Call	Description
sleep_for_ever()	Sleep forever
sleep(n)	Sleep for n milliseconds
reset()	Hard reset the badge
soft_reset()	Soft reset the badge (keep connection to IDE; does not restart VectorOS)

These services are also available to programs:

VectorOS Normal API (vectoros.py)

Call/Variable	Description
get_screen()	Returns a *ScreenNorm* object for use to access the screen
vectoros_active()	Returns true if VectorOS is running and ready
launch_repl()	Launch the repl (assumes it isn't already running). You normally won't call this directly, but let vos_launch.py do it for you
launch(task_tag)	Launch a program by its tag defined in *vos_launch.py*
remove_task(task_tag)	Remove a task from the list by its tag. This does not stop or unload the task. It simply removes it from the list
launch_vecslot(slot)	Launch a *Vectorscope* routine by slot defined in *vos_launch.* Note that *SlotX* where *X* is A, B, C, or D will cause the slot to launch from the A, B, C, or D button.
vos_debug.debug_print(level,...)	Print (as a normal print) if *debug_level* in *vos_launch.py* is <= *level*
vectoros_startup(autolaunch=True)	Start up service and (optionally) autolaunch programs. Normally called automatically
vectoros_shutdown(deactivate	Attempt to shut down VectorOS services and

Call/Variable	Description
=True)	programs. If *deactivate* is *False*, don't signal other programs via *vos_state* that we are shutdown
ext_run(cmd)	Shutdown VectorOS and run an "external program." For example: *ext_run("import foo\nfoo.main")*

Working with the Debug Level

In *vos_launch.py*, you can set a debug level. Then, in your code, you can *import vos_debug* and use *vos_debug.debug_print()* with a level to output different messages. Each call to *debug_print()* has an associated level that must be less than or equal to the level set in *vos_launch*.

Here are a few examples:

```
import vos_debug
vos_debug.debug_print(vos_debug.DEBUG_LEVEL_ERROR,
                      "Error",v,"is greater than 10!")
```

The levels are simple numbers and should be positive. However, there are some predefined levels:

- DEBUG_LEVEL_SEVERE 0
- DEBUG_LEVEL_ERROR 10
- DEBUG_LEVEL_WARNING 20
- DEBUG_LEVEL_INFO 30

In addition, *DEBUG_LEVEL_SILENT* is equal to -1 which will suppress all debug messages if set in *vos_launch*. This assumes, of course, that you never tag messages with a negative level.

If you have the REPL set to run, you can manipulate the debug level at run time by importing *vos_launch* and setting or reading *vos_launch.debug_level* directly.

VectorOS State

The *vos_state* import provides a *vos_state* data structure that contains some variables about the state of VectorOS. The easiest way to use this structure is to include:

```
from vos_state import vos_state
print(vos_state.task_dict)   # for example
```

VectorOS State (vos_state.py)

Field	Description
task_dict={}	Dictionary of tags and task objects. If a task exits, this isn't updated automatically. In addition, $gc, $key, $timer, and $repl are system tasks, if so configured
gc_suspend=True	Suspend garbage collection
show_menu=True	Use this flag to activate the main menu (*menudemo*)
active	Returns *True* if VectorOS is running (also use *vectoros.vectoros_active()*
version	A unique version number

The *vos_state.py* file is a handy place (but not the only place) to put global variables you need. We suggest adding a single list or dictionary there to allow all programs to access variables with a separate name. For example, if your application is called SoundMachine, you might define:

```
sound_machine_data = { "option1": 0, "timeout": 30}
```

Then any file that wants to use this can write:

```
from vos_state import sound_machine_data
print(sound_machine_data["timeout"]
```

Working with the Keyboard, Joystick, and LEDs

The *keyboardio.KeyboardIO* object manages a single polling loop for the keys, including the joystick. You configure it to run as a task automatically in your VectorOS configuration, or you can do it manually. To use the keyboard, you normally will use *keyboardcb.KeyboardCB* or *joystick.Joystick* to set up callbacks to things you are interested in. You can also derive custom classes from *KeyboardIO* or any of the other keyboard classes. The class also manages the LEDs on the board. Key and LED definitions are in *keyleds.py*.

The classes are context managers, so you can use "*with*" to monitor their lifetimes. If you don't, it is important that they don't go out of scope, or that you call *detach()* when you are done so they stop getting keyboard events. See https://docs.python.org/3/reference/compound_stmts.html#the-with-statement for more details about context management.

Note that it is permissible to have multiple classes process keyboard events. For example, you might have a *Joystick* object with the direction keys set to repeat. You might also define a

KeyboardCB object that only handles the joystick press event in a non-repeat mode (and, perhaps, then, you don't handle it in the *Joystick* object).

KeyboardIO features

Initializing has an optional boolean flag. If *True*, the default, the instance will subscribe to keyboard events. If not, the object is made but will not be subscribed (use *attach()* later to subscribe). Note that all of these items are available in subclasses like *KeyboardCB* and *Joystick*. You will rarely use this class yourself unless you subclass it. You'll usually use one of the built-in subclasses.

KeyboardIO API (keyboardio.py)

Call/Variables	Description
KeyboardIO(attach=True)	Constructor. If *attach* is false, the instance won't process keyboard events
leds=0	Class variable. Set this to influence the LEDs (see *keyleds.py* for definitions)
task=None	Class variable. Task ID for the keyboard polling task
current_keys=[]	Class variable. Used only in keyboard callbacks. Has a list of keys pressed. This can be useful to check for multiple keys being pressed at once
run(timeout=100)	Class method. Start the main loop with the specified millisecond timeout. Set to zero to use the default (100ms) or omit the argument. You usually won't call this, but will set the timeout in *vos_launch.py*
attach()	Begin receiving keyboard events
detach()	Stop receiving keyboard events
capture(tf=True)	Prevent other keyboard objects from receiving keyboard events while the capture is active. Use False to release the capture. Returns false if the keyboard is already captured by someone else
key(buttons)	You don't call this directly, but you can override it in a subclass to receive all keyboard events

KeyboardCB Features

When you initialize a *KeyboardCB* object, you pass a callback. Callbacks can be in two forms: a function that gets all keys of interest (explained in a moment) or a dictionary of key IDs (from *keyleds.py*) and functions that take an argument telling you what key was pressed. The second parameter is a filter list of keys you are interested in. If this is present, you will only get these

keys in your callbacks, even if you have a dictionary with more keys. However, if you don't provide a filter list but you do provide a dictionary, the filter will be the dictionary's keys. If you don't provide either, your callback will get all keyboard events.

The last parameter to the constructor is a Boolean that is *True* by default. When *True*, this sets single key mode. In this mode, you will get one event for each key press. If this is false, then you will get multiple events as long as the key is pressed.

There is an *active* flag that stops all key processing in the object if it is True. There is also a *set_callback* method if you want to change the callback after construction.

The module has a non-class function called *replace_chord*. It takes a list of keys, a list of chord keys (that is, multiple key presses that should appear as one key), and a value to replace the chord with. This would be useful in a subclass that overrides key. For example, suppose you want to know if *keyleds.KEY_A* and *keyleds. KEY_B* are both set at the same time. The key vector might be *[KEY_A, KEY_SAVE, KEY_B]*. You could look for the key chord *[KEY_A, KEY_B]* and replace it with *KEY_AB* (which you would have to define). The resulting list would be *[KEY_SAVE,KEY_AB]*.

KeyboardCB API (keyboardcb.py)

Call/Variables	Description
KeyboardCB(callback={}, filter=[], single_key_mode=True, attach=True)	Constructor. The callback can be a single function or a dictionary that maps keynames to functions. In either case, the functions take an argument that will receive the key name from *keyleds.py*. If filter is provided (a list of keys from *keyleds.py*), only those keys will be passed to the constructor. If there is no filter, but the constructor is a dictionary, the keys of the dictionary will be used as a filter. If neither apply, then the callback will get all keyboard events (assuming you don't override the key function). The *single_key* parameter, if *True*, causes the callback to fire once per key press. If *False*, the callback is called at each scan period if the corresponding key is pressed. If *attach* is false, the instance won't process keyboard events
set_callback(func_or_dict)	If you don't specify a callback or you want to replace the existing one, you can use this call
keyboardcb.replace_chord(b, chord, value)	This is not a class method. It is useful in *key()* callbacks to search for multiple keys and replace them with a different key code. If *b* is the list of buttons pressed, chord is a list

Call/Variables	Description
	of keys to find (e.g., *[JOY_UP, JOY_RIGHT]* and *value* is the keycode to replace them with (e.g., *JOY_NE*). The search keys don't have to be adjacent or in order in the list
active	Set to *False* to temporarily stop processing

The KeyboardRepeat Class

The *KeyboardRepeat* class is exactly the same as the *KeyboardCB* class but allows keys to repeat slower than the scan rate. This is useful if you need a fast scan rate for some keys (e.g., the joystick) or to update LEDs quickly, but don't want a fast repeat on other keys. Note that if you don't mind keys repeating at the natural rate or you don't want to repeat at all, you can simply use *KeyboardCB*.

KeyboardRepeat API (keyboardrepeat.py)

Call	Description
KeyboardRepeat(repeat_count=3, callback={}, filter=[], attach=True)	Constructor. The callback is the same as *KeyboardCB*, other than you specify a repeat count and you can't ask for single key mode. The callbacks will fire when you first press the key and then repeat after the number of scans indicated

The Joystick Class

The Joystick class is exactly the same as the *KeyboardCB* class, but you only provide a callback and, optionally, if you want single keys or not (default is *False*, get repeat keys).

The result is the same as using the *KeyboardCB,* but it also synthesizes chords like *KEY_NE* for a northeast joystick press. These are defined in *keyleds.py*.

Joystick API (joystick.py)

Call	Description
Joystick(callback={}, single_key_mode=False, attach=True)	Constructor. The callback is the same as *KeyboardCB*, and the filter is automatically set to *JOY_ALL*. The *single_key_mode* and *attach* parameters are the same as in *KeyboardCB*

A Demo of the Keyboard Classes

The following program is not a VectorOS program, although it does use the *sleep_forever* call from VectorOS. Pressing certain keys will print messages on the terminal to illustrate various features of the keyboard system.

The callbacks here are all local to the *main()* function, but this does not have to be the case. Note that after defining callbacks, the demo spins up a keyboard polling task using *keyboardio.KeyboardIO.run(50)* so the scan rate is approximately 50 milliseconds.

There are 6 different keyboard handlers in this code, each showing off different aspects of the system:

- debugkey - Print messages for key A, B, C, and D. This uses a filter and a single callback
- cb1 - Watch for the SAVE key. The callback can distinguish between SAVE and SAVE+A being pressed.
- cb2 - Also print messages for key A, B, C, and D using a dictionary callback and filter. If you press D, the *debugkey* handler will be detached. Pressing A will reattach it.
- joy - This handles the joystick. It also shows how a lambda can be used as a callback.
- rpt - Monitor the RANGE key with a slow repeat rate (50x20 is about 1 second).
- fast - Monitor the LEVEL key with a natural repeat rate (50 milliseconds)

```
import keyleds
import keyboardio
import keyboardcb
import keyboardrepeat
import joystick
from vectoros import sleep_forever



# This demo does not handle all keys (but it could)
# it does handle
# 1 - A, B, C, D with a generic catch all callback
# 2 - A, B, C, D with separate callbacks
# 3 - Save and A+Save
# 4 - The Joystick including synthetic corner directions
# 5 - Range button on a 1 second repeat
# 6 - Level button on a 50ms repeat

if __name__ == "__main__":

      def main():
```

```
      global debugkey
# stupid callback
      def generic_cb(key):
            print(f"Got {key}")
# example callback that is not in a class. It also detects the A modifier
key
# you can also build classes to recognize chords like joystick does
      def global_cb(key):
            if keyleds.KEY_A in keyboardcb.KeyboardCB.current_keys:
      print("A+Save pressed")
            else:
                  print("Global Save CB")
      debugkey = None

      def cba(key):
            keyboardio.KeyboardIO.leds=keyleds.LED_X
            print("A!")
            debugkey.attach()   # if we press A after D, reattach!
      def cbb(key):
            print("B!")
            keyboardio.KeyboardIO.leds=keyleds.LED_Y
      def cbc(key):
            print("C!")
      def cbd(key):
            print("D!")
            debugkey.detach()   # shut the other one up after pressing D
      def ranger(key):
            if key==keyleds.KEY_RANGE:
                  print("Range repeats slowly!")
            else:
                  print("Normal repeat")


# spin up the keyboard "server" which runs once
      keyboardio.KeyboardIO.run(50)

      debugkey=keyboardcb.KeyboardCB(generic_cb,keyleds.KEY_ABCD)  # watch
ABCD with default callback
      cb1=keyboardcb.KeyboardCB(global_cb,keyleds.KEY_SAVE)  # watch
KEY_SAVE
# probably wouldn't do two different ones for same keys
# but wanted to show how to do individual callbacks -- you could build this
dictionary on a different line and pass it
```

```
# callbacks={...}
# xx=KeyboardCB(callbacks,...)
      cb2=keyboardcb.KeyboardCB({keyleds.KEY_A: cba, keyleds.KEY_B: cbb,
keyleds.KEY_C: cbc, keyleds.KEY_D: cbd })
# just to show off... a lambda callback -- could have been a function or a
dictionary
      joy=joystick.Joystick(lambda k: print(f"joystick {k}"))

      rpt=keyboardrepeat.KeyboardRepeat(20,{ keyleds.KEY_RANGE: ranger})
      fast=keyboardcb.KeyboardCB({keyleds.KEY_LEVEL: ranger},[],False)
      sleep_forever()


      main()
```

Working with Multiple Key Presses

There are several ways to detect that multiple key presses have occurred at the same time (e.g., press A and move the joystick up). First, during a callback, *KeyboardIO.current_keys* contains a list of all keys currently pressed (regardless of your filter). So you can check to see if a key is down or not by testing for membership in the list. For example:

```
      def global_cb(key):
            if keyleds.KEY_A in keyboardcb.KeyboardCB.current_keys:
      print("A+Save pressed")
            else:
                  print("Global Save CB")
```

In this case, *KeyboardCB* accesses the list, which is fine since it is a subclass of *KeyboardIO* and the *current_keys* belongs to the class, not a particular instance.

The second method is to create a subclass of another keyboard object. The *Joystick* class (see below) does this. The *key()* method receives a filtered list of keys, and you can find key combinations and replace them with equivalent key codes. Because this is so common, the *keyboardcb* module has a global-level function to do this.

The *replace_chord* method takes a list of keys, a list of chord keys (that is, multiple key presses that should appear as one key), and a value to replace the chord with. Here's a portion of the code in *joystick.py*:

```
      def key(self, b):
      # generate synthetic keys
      b1 = keyboardcb.replace_chord(b, [keyleds.JOY_N, keyleds.JOY_E],
```

```
keyleds.JOY_NE)
      b1 = keyboardcb.replace_chord(b1, [keyleds.JOY_N, keyleds.JOY_W],
keyleds.JOY_NW)
      b1 = keyboardcb.replace_chord(b1, [keyleds.JOY_S, keyleds.JOY_E],
keyleds.JOY_SE)
      b1 = keyboardcb.replace_chord(b1, [keyleds.JOY_S, keyleds.JOY_W],
keyleds.JOY_SW)

      super().key(b1)  # dispatch callbacks as usual
```

Working with LEDs

The LEDs are not directly connected to the CPU I/O ports. They are connected to a shift register that is also used to scan the keys. Therefore, to use the LEDs, you must have a running keyboard task, which, if you are using VectorOS, you do. However, you can also use your own code to launch a keyboard loop (*keyboardio.KeyboardIO.run*).

KeyboardIO has a class variable called *leds* that reflects the state of the LEDs, and they are updated at the keyboard scan frequency. If you don't mind writing to that byte value using the constants in *keyless.py*, then that's all you need to do. However, if you would prefer a higher-level interface, you can use the *LED* objects defined in *led.py*. There are eight predefined objects that you can use to control each LED easily. You do not need to create instances or subclass *LED*. In fact, accessing the LEDs through derived classes will not work unless the derived class is the one running the event loop.

LED API (led.py)

Call/Property	Description
set()	Set (turn on) the LED
reset()	Reset (turn off) the LED
toggle()	Flip the LED from on to off or vice versa
value	A property. Set or read the LED state as a True/False or 1/0 (always reads True or False)

In addition to the above calls, the objects themselves are callable, which is the same as setting the value property. For example:

```
      LED.Square(True)    # Turn on square wave LED
```

The predefined LED objects are:

- X

- Y
- Triangle
- Square
- Sine
- Sig
- Scope
- Saw

Here is a sample VectorOS program that uses this style of interaction with the LEDs:

```
import vectoros
from timer import Timer
import led

flip=False
# run from timer
def tick():
      global flip
         if flip:
         led.Y.set()
         led.X.reset()
         flip=False
         led.Scope.value=1
         led.Sig(False)
      else:
         led.Y.reset()
         led.X.set()
         flip=True
         led.Scope.value=0
         led.Sig(True)
         led.Sine.toggle()

def main():
      Timer.add_timer(10,tick)
      vectoros.sleep_forever()


if __name__=="__main__":
      import vectoros
      vectoros.run()
```

Working with Timers

In addition to creating a loop that sleeps, you can also install a callback on a timer. The timer's base rate is set in *vos_launch* (*timer_base_rate*) in milliseconds. All timeouts are computed against that base rate. The base rate should not be too fast (e.g., 10 milliseconds is probably too fast, and you will get recursion errors as the timer's loop overruns itself). The default is 100 milliseconds.

You don't have to instantiate the timer object. You only need to refer to *Timer*. VectorOS runs the timer loop as part of its startup (unless you set the base rate to zero). The only methods you normally deal with are *add_timer* and *remove_timer*. The *add_timer* call takes a number of ticks and a callback. There is also an argument to specify that the timer should fire only once or if it should repeat.

Because of the nature of *asyncio*, you can't depend on these timers to be extremely accurate, and any time you spend processing synchronously may affect the accuracy of the timer. A synchronous callback needs to exit as soon as possible. Still, for many tasks, they work well. If you need more robust timing services, check out https://docs.micropython.org/en/latest/rp2/quickref.html#timers.

Note that the callback can be the name of a function (the default) or an "awaitable." For example:

```
def cb1():    # sync callback
     print("Tick")

async def cb2():   # async callback
     print("Tock")

tick_timer=timer.Timer.add_timer(100,cb1)
tock_timer=timer.Timer.add_timer(100,cb2)
```

If you use an awaitable (e.g., an *async* function), the timer will call it with *await*. However, if needed, it can then create a new async task and return. In other words, either way, the timer will stall until the callback completes.

You can also create subclasses of the *Timer* object. The timer object is a context manager, so you can use "*with*" to ensure it doesn't stay active when you are done with it, or you can manage that yourself by ensuring timers are paused when you are done with them. Here's a simple example of a custom Timer class:

```
      class Test(Timer):
      def __init__(self):
            super().__init__(20)
            self.ct=0
```

```
    def action(self):
          self.ct=self.ct+1
          print("**",self.ct)
          if (self.ct>=10):
                self.pause()
```

At a base rate of 100ms, this timer will print “**1”, “**2”, etc. every two seconds until it reaches 10. Then, it will stop. To use it, you only have to write:

```
twosec=Test()
```

Or use a context manager:

```
with Test() as twosec:
    ...
```

Timer API (timer.py)

Call	Description
run()	Class method. Start the timer loop (usually not required with VectorOS)
baserate	Class variable. The base rate in milliseconds (usually not required to set when using VectorOS)
add_timer(ticks,callback, oneshot=False)	Class method. Add a timer callback which may be synchronous or asynchronous and occur once or repeat. Returns a timer ID
remove_timer(id)	Class method. Remove the specified timer ID
gc_delay	Class variable. Usually not needed with VectorOS. If this is set, then garbage collection will occur after the specified number of ticks inside the main timer loop
Timer(ticks, paused=False,oneshot=False)	Constructor. Create a timer that will call *self.action* as indicated
pause()	Pause this *Timer* instance
resume()	Start a paused timer or oneshot instance

Call	Description
action()	Override to add your code to a timer instance (do not call the base class)

Working with the Screen

The screen isn't really a VectorOS resource, but you can use the *ScreenNorm* object to control it. By default, the screen is awake, but you can put it to sleep with *idle()* or wake it up again with *wake()*. Since there only needs to be one *ScreenNorm* object, VectorOS provides *get_screen()* to return the screen instance. The screen has nominal dimensions of 240x240.

Screen API (screennorm.py)

Call/Variable	Description
ScreenNorm()	Constructor. No arguments. The screen starts in the awake state
tft	The underlying screen object (see https://github.com/russhughes/gc9a01_mpyhttps://github.com/russhughes/gc9a01_mpy)
idle()	Turn off the display and release the SPI bus
wake()	Turn on the display
jpg(filename)	Show a 240x240 JPG file (convenience wrapper for *tft*)
text(x,y,text,fg_color, bg_color)	Display text at location x,y in the given foreground and background colors (omit for default colors)
text_font(font, x, y, text, fg_color, scale=1.0)	Display text using vector font (as text). Always transparent. Use None for the default font or one of the fonts at https://github.com/russhughes/gc9a01_mpy/tree/main/fonts/vector
clear(color)	Clear the screen with the given RGB565 color (default black)
pixel(x,y,color)	Set pixel at x,y with given RGB565 color
get_font()	Get the default bitmap font
get_vfont()	Get the default vector font

Screen Colors

The *colors.py* file has several colors you can use with the display, including those that mimic a green CRT.

Colors (colors.py)

Color/Call	Notes
BLACK, BLUE, CYAN, GRAY, GREEN, MAGENTA, RED, WHITE, YELLOW	
PHOSPHOR[n]	Green CRT-style colors n=0 to 31
PHOSPHOR_BRIGHT	Highlight color
PHOSPHOR_DARK	Normal color
PHOSPHOR_BG	Background color
rgb(red, green, blue)	Convert a normal 8-bit per color RGB triple to a screen color

Working with Menus

It is simple to create menu-driven user interfaces with the *Menu* class. You configure the menu with a nested list containing a sublist for each menu item. The sublist has the format: *[text, callback, argument]*. For example:

```
[[ "Item #1", item_process, 1],["Item #2", item_process, 2],
 ["Item #3", item_3,0]]
```

In this case, the *item_process(arg)* function would handle the first two menu items and *item_3(arg)* would handle the last one. You must call the menu from an *async* function using *await*:

```
      with Menu(clear_after=True) as amenu:
                  await amenu.do_menu(mainmenu)
```

The with statement allows the menu, which is a context manager, to release its internal keyboard handler when it goes out of scope. You can also manage this without using context management. You also have the ability to set a callback (or subclass menu) to dynamically set the menu based on current conditions.

To specify a submenu, use an action of *[]* (or *Menu.SUBMENU*) and provide the variable containing the submenu definition as the argument. If you use *None* as the action, the menu item will work as a back button.

Menu callbacks receive an argument from the configuration list. They are expected to return a value. Returning *Menu.CONT* allows the menu system to continue. *Menu.BACK* will do a back action. *Menu.EXIT* will exit the menu system.

Typically, callbacks are synchronous (non-async) functions. If you need an asynchronous call, just add an additional argument (we like "async" but anything will work) as the fourth item of the menu list.

```
[["My async Item",asynccb,0,"async"]....
```

A common thing to do is to launch a predefined VectorOS task (defined in *vos_launch*). You can do this by specifying *menu.launch* as the action of a menu item and use the tag string as an argument.

Menu System API (menu.py)

Call	Description
Menu(fg_color, bg_color, cursor_fg, cursor_bg, clear_after, joy_controller, scanrate)	Constructor. All arguments are optional. The *fg* and *bg* parameter sets colors. Set *clear_after* to True to have the menu system clear the screen on exit. The *joy_controller* argument sets a custom keyboard handler (you will rarely use this). Set *scanrate* to a non-zero to have the menu start a main keyboard task (never set this while using VectorOS). If *cursor_fg* or *cursor_bg* are *None*, they assume the value of the background and foreground, respectively
detach()	Stop responding to keyboard events
menu_custom()	Override this to have a chance to change the menu each time it is displayed. By default, it will call a callback for this purpose if you set one
set_callback(func)	The default menu_custom function calls the callback you set here before the menu draws or redraws
set_font(font,scale=1.0)	Set a vector font or *None* for the default font or "*" for the default vector font. The scale is a float and defaults to 1.0
menu_update()	Redraw menu
do_menu(menulist)	Execute menu
menu.launch(tag)	Global: turn off menus and launch a tag defined in

Call	Description
	vos_launch.py. Set vos_state.show_menu to True to turn menus back on later.

You often won't need to customize the menu. However, if you need to dynamically change entries (for example, change an entry from On to Off depending on some external state) you can do it. To customize the menu you can override *menu_custom* or use *set_callback* to cause the default *menu_custom* to call you each time is about to draw the display. Here's an example of a callback function:

```
# change menu[0][2] depending on state of blinky
def menu_custom(the_menu):
      if the_menu.level==1:
      if 'blinky' in vos_state.task_dict:
            the_menu.current[2]=["Stop Blinker",blinkrun,(0,the_menu)]
      else:
            the_menu.current[2]=["Start Blinker",blinkrun,(1,the_menu)]
```

The numbers here would change, of course, if you changed the structure of the menu. Setting the callback is simple:

```
      with Menu(clear_after=True) as amenu:
            amenu.set_callback(menu_custom)
            await amenu.do_menu(mainmenu)
```

Note that, in this case, only the top-level menu changes (because of the test in *menu_custom* for *the_menu.level*). The callback function sets the altered menu item to have a callback that takes an argument composed of a number and a pointer to the menu because this item might want to read or alter things in the menu, too. You can write that function (in part) like this:

```
def blinkrun(arg):
      flag, amenu = arg
       . . .
```

Menu List Reference

Menu items use a list to configure each item, and the menu itself is a list of lists. You can use the following items in a menu:

Menu List Reference

Action	Argument	Description
<function>	<argument>	Call function. Return MENU.BACK, MENU.EXIT, or (normally) MENU.CONT.
[]	Submenu list	Open submenu
Menu.SUBMENU	Submenu list	Open submenu
None	Ignored	Back (or exit from top level)
Menu.m_back	Ignored	Back (or exit from top level)
Menu.m_exit	Ignored	Exit menu
menu.launch	tag	Launch a tag after turning off menu

For example, you might have:

```
[["Program 1", run,1],["Program 2", run, 2],["Submenu"],
 [Menu.SUBMENU,menu_3],["Back",Menu.m_back,0]]
```

About AsyncIO

Python allows cooperative multitasking via coroutines using the *asyncio* library. MicroPython provides a subset of *asyncio* with some extensions. VectorOS uses *asyncio* to allow you to run different parts of your code in a way so that it appears that it is all running at the same time.

By appears, we mean that the code doesn't actually run at the same time. Rather, each piece of code (an *asyncio* task) runs until it has to stop for some reason. Then, another piece of code can run. This is simple and avoids many multitasking problems with critical sections. However, it means that a piece of code that refuses to stop will hang the entire system.

VectorOS tries to hide much of the complexity of *asyncio*. If you want, VectorOS can launch your code using a normal Python function. In some simple cases, that may be all you need. However, if you intend to do much of anything with other programs running, you'll want to make your main function be async. Your main function should be named *vos_main*, although *main* is acceptable. By convention, *main* is a non-async entry point used when running code without the whole OS running. The template (*template.py*) sets this up for you:

```
async def vos_main():
      global _freeze, _exit, _run_every_ms, task_name
```

```
      _freeze=False
      _exit=False
      while _exit==False:
      if _freeze==False:
            print("Do something here")
      await asyncio.sleep_ms(_run_every_ms)
      _exit=False  # reset for next time
      vectoros.remove_task(task_name)   # make sure we are removed (also
does this in exit)

def main():
      asyncio.run(vos_main())
# code here will never, ever run under VectorOS

# if you want to configure to run in vos_launch but still run this file
# add the main function to to vos_launch.py and use this:
if __name__=="__main__":
      vectoros.run()
```

Inside an async function, you can call normal functions, although they will, generally, prevent you from releasing time to other functions. You can also create tasks (*asyncio.create_task*) or call other async functions and wait for them using *await*.

Note that *vos_main* loops until it sees the *_exit* flag set. It regularly calls another async function: *asyncio.sleep_ms* to pause to let other things run.

For more about asyncio, see the References section below.

Working with Waveforms

What good is a vectorscope if you can't produce signals and watch them on the screen? Of course, you can.

There are several parts to creating a waveform display:

1) Write a file that provides a function *slot_main* that takes a single argument. That argument is a *Vectorscope* object. Use that object to produce your analog voltages for display on the screen. If you think you'll want to use VectorOS services, you'll wnat to make the function *async*.
2) Register the function with VectorOS by editing *vos_launch.py*. The entry must be in the *vectorscope_slots* dictionary. Slots named *slotA*, *slotB*, *slotC*, and *slotD* will run when you press the A, B, C, or D button from the menu screen.
3) If you want to launch some other way, you can call *vectoros.launch_vecslot* with the name of the slot.

4) When the routine finishes, the badge will reset.

You can think of *Vectorscope* as a driver that manages the oscilloscope display. The way to control this is through the *wave* member that provides a *Waveform* object. This uses a 256-sample buffer to drive X and Y voltage outputs on the SIG GEN pins. If you then feed the X and Y outputs to the badge's SCOPE X and Y pins, you can draw anything you want. The members appear in the table above.

Waveform API (waveform.py)

Call/Variable	Description
outBuffer_ready	True when the out buffer is ready for new data
num_samples	Number of samples per frame
outBufferX	The X output buffer
outBufferY	The Y output buffer
packX(wavelist)	Put list of integers into X buffer
packY(wavelist)	Put list of integers into Y buffer
constantX(value)	Fill X buffer with integer
constantY(value)	Fill Y buffer with integer
point(x,y,value)	Fill buffer to produce a point

For example, here's a simple "slot" example:

```
import math
import time

from vectorscope import Vectorscope
from random_walk import RW

import vectoros
import keyboardcb
import keyleds
import asyncio


_abort=False
```

```
_xscale=1
_yscale=1

async def kminimal_example(v):
      ## Minimal example
      global _abort, _xscale, _yscale
      while _abort==False:
      for t in range(200):
            if _abort:
                  print("Get out!")
                  break
            v.wave.constantX(int(math.cos(t * math.pi / 180 * 5 *_xscale) *
10000))
            v.wave.constantY(int(math.sin(t * math.pi / 180 * 5 *_yscale)*
10000))
            await asyncio.sleep_ms(10)


def do_abort(key):
      global _abort
      _abort=True

def do_xscale(key):
      global _xscale
      _xscale+=1
      if _xscale>6:
      _xscale=1


def do_yscale(key):
      global _yscale
      _yscale+=1
      if _yscale>6:
      _yscale=1



from vos_state import vos_state

async def slot_main(v):
      global _abort,_continue
      # watch the keys (LEVEL=cycle X, RANGE=cycle Y, D=Exit)
      mykeys=keyboardcb.KeyboardCB({ keyleds.KEY_LEVEL: do_xscale,
```

```
keyleds.KEY_RANGE: do_yscale, keyleds.KEY_D: do_abort})
      await kminimal_example(v)
      print("OK done")
```

The main loop uses the fact that you can describe a sine wave as:

A*sin(2πFt+Φ)

Where A is the amplitude, t is the time, F is the frequency in Hertz, and Φ is the phase angle in radians. The cos wave is the same, but phase shifted. Read more: https://en.wikipedia.org/wiki/Sine_wave.

The *slot_main* function sets up the keyboard using VectorOS and then calls the *kminimal_example* function. You can press the Level and Range buttons to change the X and Y frequencies between 1 and 6. Press D to exit.

Note that the scope display routine needs time to process, but if you are using VectorOS service (which you don't have to), then the cooperative multitasking requires some pauses in your code. However, the bulk of the scope code runs on the Pico's other core to minimize this issue. Still, you may have to plan your timing in your code correctly to allow VectorOS services to run and still have data for the scope.

When you are not running a slot, VectorOS provides garbage collection for you. When you are running a slot, the *Vectorscope* code does it for you. In addition, while running a slot, the VectorOS screen is unavailable.

You may find the file template_slot.py useful as a starting point for writing your own slots. You can also find a more complete example in the example code below.

Inside generate_wavetables.py

You may find functions in *generate_wavetables.py* useful to create waveforms using degree measurements.

Function	Description
phaseSteps(maxphase, length=256)	Generate phase table
sine(maxPhase=360,length=256)	Sine wave
sawtooth(maxPhase=360, length=256)	Sawtooth
square(maxPhase=360, length=256)	Square
triangle(maxPhase=360, length=256)	Triangle
bandlimitedSawtooth(numberPartials,	Sawtooth with bandwidth limiting

maxPhase=360, length=256)	
bandlimitedSquare(numberPartials, maxPhase=360, length=256)	Square with bandwidth limiting
bandlimiteTriangle(numberPartials, maxPhase=360, length=256)	Triangle with bandwidth limiting
scaleAndRound(data, scale=2**16-1, signedInt=True)	Scale samples

Example Code

Example code is available on GitHub (https://github.com/Hack-a-Day/Vectorscope).. You can use the joystick to navigate the menus. Pushing the joystick or moving it to the right selects an item. Moving left goes back, as do some menu items marked “Previous.”

Some of the example applications include:

- Lissajous - Use the joystick keys to adjust frequency (left/right for X, up/down for Y). If you hold down RANGE while moving the joystick to control the amplitude. Hold down LEVEL to adjust phase with the joystick, although the code also periodically adjusts the phase. To change the waveform used, select X or Y with the bottom left buttons and then press the waveform button to cycle through sine, square, sawtooth, and triangle. Menu button exits.
- Sketch - Use the joystick to move the cursor. If the cursor is reddish, you’ll leave a trail in the current color. Push the joystick to toggle to move only mode (greenish cursor). The A, B, C, and D buttons select colors (black, red, green, blue). The Level button will change the color to white (erasing). The User button clears the screen. Menu will return to the menu. The most recent version allows you to press SAVE+A,B,C,D to save up to four screens for recall. Press User+A,B,C,D to recall.
- Planets - Show a graphic slide show of the planets (and the moon). The joystick can advance (right) or pause (left toggles). Use up and down to change the speed of the slide show. Menu exits.

You can also find a template for writing a simple VectorOS program in the *template.py* file (see below). It shows how you can set up the program to run virtually standalone for debugging or how you can set it to run and start up VectorOS.

```
# Use this template to start your VectorOS program:

import vectoros
import asyncio
import keyleds
import keyboardio
```

```
task_name="Template"  # use this name in vos_launch.py, too
_run_every_ms=500   # how often to loop


_freeze=False
_exit=False


# Outsiders can call this to exit you
def exit():
      global _exit,task_name
      _exit=True
      vectoros.remove_task(task_name)


# Outsiders can call this to pause you
def freeze(state=True):
      global _freeze
      _freeze=state


async def vos_main():
      global _freeze, _exit, _run_every_ms, task_name
      _freeze=False
      _exit=False
# if you want to control keyboard and LED without running the whole OS
      if vectoros.vectoros_active()==False:
      keyboardio.KeyboardIO.run(250)
      while _exit==False:
      if _freeze==False:
            print("Do something here")
      await asyncio.sleep_ms(_run_every_ms)
      _exit=False  # reset for next time
      vectoros.remove_task(task_name)   # make sure we are removed (also
does this in exit)

def main():
      asyncio.run(vos_main())
# code here will never, ever run under VectorOS

# if you want the possibility to run directly without VectorOS
```

```
#if __name__=="__main__":
#     main()

# if you want to configure to run in vos_launch but still run this file
# add the main function to to vos_launch.py and use this:
if __name__=="__main__":
      vectoros.run()
```

Inside simple.py

By default, simple.py isn't included in the demo. It is perhaps the least amount of code required to blink the LEDs in a VectorOS program. It starts with the template, but does not use any async calls at all. In truth, it could be a little less complex, but it does expose functions to allow you to pause and stop it.

Here's the code:

```
# Almost the simplest VectorOS program:

import vectoros
import keyleds
import keyboardio
from timer import Timer



task_name="Simple"  # use this name in vos_launch.py, too
_run_every_ms=500   # how often to loop


_freeze=False
_exit=False
_timerid=None


# Outsiders can call this to exit you
def exit():
      global _exit,task_name
      _exit=True
      vectoros.remove_task(task_name)
```

```
# Outsiders can call this to pause you
def freeze(state=True):
      global _freeze
      _freeze=state


def callback():
      global _exit, _freeze, _timerid
      if _exit:
      Timer.remove_timer(_timerid)
      return
      if _freeze==False:
      keyboardio.KeyboardIO.leds^=0xFF


def main():
      Timer.add_timer(5, callback)   # 5 * 100 = 500 ms
      vectoros.sleep_forever()
# code here will never, ever run under VectorOS

# if you want the possibility to run directly without VectorOS
#if __name__=="__main__":
#     main()

# if you want to configure to run in vos_launch but still run this file
# add the main function to to vos_launch.py and use this:
if __name__=="__main__":
      vectoros.run()
```

The main program is just one line and the work happens in a timer callback. The line at the bottom means that when you run the program, it will launch VectorOS. However, unless you've configured *simple* to run, it won't start this program. To do that, you need to edit *vos_launch.py* like this (only part of the file is shown):

```
# list of things to launch from vectoros

# dictionary of tags to functions
launch_list={ "menu": "menudemo", "blinky": "simple", 'sketch': 'sketch',
"screen": "screentest"}

# list what you want to start auto (maybe just one thing?) need tag
auto_launch_list=["menu", "blinky"]
```

```
auto_launch_repl=True

key_scan_rate = 100     # how often to scan the keyboard globally (ms; 0 to
do it yourself)

# how often to garbage collect
# if you set this to zero and do nothing else
# garbage collection will be automatic as usual and before new tasks launch
gc_thread_rate = 1200

# Base rate for the timer (ms)
timer_base_rate=100
DEBUG_LEVEL_SILENT=-1
DEBUG_LEVEL_SEVERE=0
DEBUG_LEVEL_ERROR=10
DEBUG_LEVEL_WARNING=20
DEBUG_LEVEL_INFO=30
debug_level=DEBUG_LEVEL_INFO

if __name__=="__main__":
      import vectoros
      vectoros.run()
```

Note that while this replaces simple for the original blinky task, the menu doesn't know about it, so the Start/Stop Blinker and Freeze menu items won't do what you think. However, it would be easy to modify the *menudemo.py* file to fix that. As an exercise, you could modify *vos_state.py* to have some pseudo-global variables to control the blinking task and then rewrite both *simple.py* and *blinker.py* to use that variable. There are other ways you could this, too, if you want to experiment.

Inside blinker.py

The *blinker.py* example is one of the simplest examples provided, yet it shows two different ways to handle repeating tasks. The lower 8 bits of the LED register (inside *KeyboardIO* or any of its subclasses) are managed by a loop that sleeps for one second total using two sleeps of 500 milliseconds each.

The top 8 bits are managed by a *Timer* callback that fires every 1.5 seconds. Here is the main function (you can find the entire code on GitHub):

```
async def vos_main():
      global _freeze, _exit
```

```
      _freeze=False
      _exit=False
      if vectoros.vectoros_active()==False:
      keyboardio.KeyboardIO.run(250)
      timertask=timer.Timer.add_timer(15,callback1500ms)
      while _exit==False:
      if _freeze==False:

keyboardio.KeyboardIO.leds=(keyboardio.KeyboardIO.leds&0xF0)|0xA

      await asyncio.sleep_ms(500)
      if _freeze==False:
            keyboardio.KeyboardIO.leds=(keyboardio.KeyboardIO.leds&0xF0)|5
      await asyncio.sleep_ms(500)
      _exit=False
      vectoros.remove_task('blinky')
```

Insider timer.py

The *timer.py* file has a test program at the bottom that does not directly use VectorOS. However, it does show the different kinds of timers you can set, namely:

- Synchronous timers
- Asynchronous timers
- Synchronous one shots
- Asynchronous one shots

It also uses the API to stop a timer. Here's the main code:

```
      async def amain():
      global onesecid
      Timer.run()   # this is not needed under VectorOS

      # save the timer ID to kill later
      onesecid=Timer.add_timer(10,callback1sec)   # every one second
      Timer.add_timer(50,callback5sec)           # 5 second synchronous
repeating
      Timer.add_timer(100,once,True,True)       # one shot and remove the 1
second tick
      Timer.add_timer(55,acallback(),False,True)  # one shot
```

Inside csvdemo.py

If you would like to use a spreadsheet to calculate a waveform, have a look at csvdemo.py (below).

```
# simple Vectorscope "Slot"
import math
import time

from vectorscope import Vectorscope
from random_walk import RW

import vectoros
import keyboardcb
import keyleds
import asyncio


_abort=False


      # put your CSV file below
      # It should have 256 samples
      # While the theoretical limit is +/-32K,
      # the center of the round display is around +/-10000
      # Just paste the data between the csv=[ line and the ] line

csv=[



]


async def kernel(v):
      ## Minimal example
      global _abort
      csvlen=len(csv)
      if csvlen!=256:
      print(f"Warning wrong csv size {csvlen}")
      v.wave.packX(range(-2**15,2**15,2**8))
      v.wave.packY(csv)
      while _abort==False:
      await asyncio.sleep_ms(20)


def do_abort(key):
```

```
      global _abort
      _abort=True


async def slot_main(v):
      global _abort,_continue
      # watch the keys (D to exit)
      mykeys=keyboardcb.KeyboardCB({ keyleds.KEY_D: do_abort})
      await kernel(v)
```

The *slot_main* sets up the keyboard and goes to the waveform kernel. The X axis gets a “sweep” ramp and the Y axis comes from the csv list. To create this, use a spreadsheet program to compute 256 values and export them as csv. Paste them into the file and it is ready to go.

Useful Code Snippets

Generate Sine Wave

```
int(amplitude * math.sin(time * math.pi/180 * freq)
```

Call a function with no return that may be async (or not)

```
try:
   async func(1,2,3)  # must not return value!
except TypeError:
   pass
```

Connect buttons to callbacks:

```
def cbA(key):
   print("A")

def cbB(key):
   print("B")
keyab=keyboardcb.KeyboardCB({ keyleds.KEY_A: cbA, keyleds:KEY_B: cbB})
```

Read multiple key presses in a keyboard callback

```
      def global_cb(key):
            if keyleds.KEY_A in keyboardcb.KeyboardCB.current_keys:
      print("A+Save pressed")
```

```
            else:
                  print("Global Save CB")
```

Launching VectorOS when your program runs

When developing, it is handy to be able to run your program and actually launch the OS. To do this, add the following to your program:

```
if __name__=="__main__":
      import vectoros
      vectoros.run()
```

Dual-mode programs

It is possible to write programs that run under VectorOS or run by themselves. You have to detect that VectorOS is not running and start things like the keyboard loop or the timer loop, if you want to use them.

To do this, you can replace the code you run when *__name__* is __main__. You have several options:

1) Set up services you need and call *vos_main* yourself (or do whatever you want there)
2) Call a function (e.g., *main()*) to do the setup and either launch *vos_main* or do whatever you want to do in that function
3) If you run *vos_main*, you can detect if VectorOS is active using the *vos_state*.active flag, the *vectoros.vectoros_active()* call, or you can check if the objects you are interested in have a *task* field of *None*. If VectorOS is inactive, you will be responsible for polling the keyboard, for example or, at least, running the *KeyboardIO* loop with *run*.

Your exact code will depend on what you want, but you might use something like this:

```
if __name__=="__main__":
      keyboardio.KeyboardIO.run(50)
      main()
```

Generate an ascending ramp on the X axis of the scope

```
      ramp = range(-2**15, 2**15, 2**8)
      v.wave.packX(ramp)
```

References

- MicroPython - https://docs.micropython.org/en/latest/
- MicroPython for Raspberry Pi Pico - https://docs.micropython.org/en/latest/library/rp2.html and https://docs.micropython.org/en/latest/library/machine.html and

https://docs.micropython.org/en/latest/rp2/quickref.html and https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-python-sdk.pdf

- Raspberry Pi Pico - https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf
- LCD Library - https://github.com/russhughes/gc9a01_mpy
- AsyncIO Tutorial - https://github.com/peterhinch/micropython-async/blob/master/v3/docs/TUTORIAL.md
- Using Context Managers - https://docs.python.org/3/reference/compound_stmts.html#the-with-statement
- Hardware Documentation - https://github.com/Hack-a-Day/Vectorscope
- Generating Sine Waves - https://en.wikipedia.org/wiki/Sine_wave
- GitHub - https://github.com/Hack-a-Day/Vectorscope

Key and LED Definitions

Key Definitions

ID	Key Number	Description
JOY_ALL	[JOY_PRESS, JOY_UP, JOY_DN, JOY_RT, JOY_LF, JOY_NE, JOY_NW, JOY_SE, JOY_SW]	All Joystick keys (useful for filters)
JOY_DN	11	Joystick Down
JOY_E	JOY_RT	Joystick Right
JOY_LF	14	Joystick Left
JOY_LT	JOY_LF	Joystick Left
JOY_N	JOY_UP	Joystick Up
JOY_NE	100	Joystick Up and Right
JOY_NW	101	Joystick Up and Left
JOY_PRESS	17	Joystick Center Button Pushed
JOY_RT	23	Joystick Right
JOY_S	JOY_DN	Joystick Down
JOY_SE	102	Joystick Down and Right
JOY_SW	103	Joystick Down and Left
JOY_UP	20	Joystick Up
JOY_W	JOY_LF	Joystick West
KEY_A	8	A
KEY_ABCD	[KEY_A, KEY_B, KEY_C, KEY_D]	A, B, C, and D buttons (useful for filters)
KEY_ALL	[KEY_MENU, KEY_USER, KEY_SAVE, KEY_A,	All keys (Possible filter although an empty filter should be the same)

ID	Key Number	Description
	KEY_B, KEY_C, KEY_D, KEY_LEVEL, KEY_RANGE, KEY_XY, KEY_WAVE, KEY_SCOPE]	
KEY_B	7	B
KEY_C	1	C
KEY_D	4	D
KEY_EVERYTHING	JOY_ALL + KEY_ALL	All keys and joystick
KEY_LEVEL	5	Level
KEY_MENU	19	Menu
KEY_RANGE	2	Range
KEY_SAVE	22	Save
KEY_SCOPE	13	Scope/Sig Gen
KEY_USER	24	User
KEY_WAVE	16	Wave
KEY_XY	10	XY

LED Definitions

ID	Key Number	LED object Name	Description
LED_SAW	4	Saw	Sawtooth LED
LED_SCOPE	2	Scope	SCOPE LED
LED_SIG	1	Sig	SIG GEN LED
LED_SINE	32	Sine	Sine LED
LED_SQ	16	Square	Square LED
LED_TRI	8	Triangle	Triangle LED
LED_X	128	X	X LED
LED_Y	64	Y	Y LED

Files on the Badge

While subject to last minute changes, here's a list of some of the files you'll find on your badge and what they are for.

File Name	Description
Startup & Configuration	
main.py	Runs on badge startup
menudemo.py	Demo menu (change the menu here)
vos_launch.py	VectorOS configuration file
VectorOS Files	
aiorepl.py	REPL for AsyncIO loop
colors.py	Definition of colors
joystick.py	Joystick class
keyboardcb.py	Keyboard to callback functions
keyboardio.py	Runs the single keyboard reading loop

keyboardrepeat.py*	Like keyboardcb, but slower repeat rate
keyleds.py	Define keys, key combinations, and LEDs symbolically
led.py*	Manage LEDs individually
menu.py*	Core of the menu system
romans.py	Vector font
screennorm.py*	Screen management
timer.py	Timers
vectoros.py*	Main VectorOS file
vga1_16x32.py	Base bitmap font
vos_debug.py	Debug logging support
vos_state.py	Information about VectorOS

* File updated or not present on badge