The Jetson GPIO library provides all public APIs provided by the RPi.GPIO library. The following discusses the use of each API.

1. Importing the libary

To import the Jetson.GPIO module use:

```
import Jetson.GPIO as GPIO
```

This way, you can refer to the module as GPIO throughout the rest of the application. The module can also be imported using the name RPi.GPIO instead of Jetson.GPIO for existing code using the RPi library.

2. Pin numbering

The Jetson GPIO library provides four ways of numbering the I/O pins. The first two correspond to the modes provided by the RPi.GPIO library, i.e BOARD and BCM which refer to the pin number of the 40 pin GPIO header and the Broadcom SoC GPIO numbers respectively. The remaining two modes, CVM and TEGRA_SOC use strings instead of numbers which correspond to signal names on the CVM/CVB connector and the Tegra SoC respectively.

To specify which mode you are using (mandatory), use the following function call:

```
GPIO.setmode(GPIO.BOARD)
# or
GPIO.setmode(GPIO.BCM)
# or
GPIO.setmode(GPIO.CVM)
# or
GPIO.setmode(GPIO.TEGRA_SOC)
```

To check which mode has be set, you can call:

```
mode = GPIO.getmode()
```

The mode must be one of GPIO.BOARD, GPIO.BCM, GPIO.CVM, GPIO.TEGRA_SOC or None.

3. Warnings

It is possible that the GPIO you are trying to use is already being used external to the current application. In such a condition, the Jetson GPIO library will warn you if the GPIO being used is configured to anything but the default direction (input). It will also warn you if you try cleaning up before setting up the mode and channels. To disable warnings, call:

```
GPIO.setwarnings(False)
```

4. Set up a channel

The GPIO channel must be set up before use as input or output. To configure the channel as input, call:

```
# (where channel is based on the pin numbering mode discussed above)
GPIO.setup(channel, GPIO.IN)
```

To set up a channel as output, call:
GPIO.setup(channel, GPIO.OUT)
It is also possible to specify an initial value for the output channel:
GPIO.setup(channel, GPIO.OUT, initial=GPIO.HIGH)
When setting up a channel as output, it is also possible to set up more than one channel at once:
# add as many as channels as needed. You can also use tuples: (18,12,13)
channels = [18, 12, 13]
GPIO.setup(channels, GPIO.OUT)

5. Input
To read the value of a channel, use:
GPIO.input(channel)
This will return either GPIO.LOW or GPIO.HIGH.

6. Output
To set the value of a pin configured as output, use:
GPIO.output(channel, state)
where state can be GPIO.LOW or GPIO.HIGH.
You can also output to a list or tuple of channels:
channels = [18, 12, 13] # or use tuples
GPIO.output(channels, GPIO.HIGH)
# or GPIO.LOW# set first channel to HIGH and rest to LOW
GPIO.output(channel, (GPIO.LOW, GPIO.HIGH, GPIO.HIGH))

7. Clean up
At the end of the program, it is good to clean up the channels so that all pins are set in their default state. To clean up all channels used, call:
GPIO.cleanup()
If you don't want to clean all channels, it is also possible to clean up individual channels or a list or tuple of channels:
GPIO.cleanup(chan1) # cleanup only chan1
GPIO.cleanup([chan1, chan2]) # cleanup only chan1 and chan2
GPIO.cleanup((chan1, chan2)) # does the same operation as previous statement

8. Jetson Board Information and library version
To get information about the Jetson module, use/read:
GPIO.JETSON_INFO
This provides a Python dictionary with the following keys: P1_REVISION, RAM, REVISION, TYPE, MANUFACTURER and PROCESSOR. All values in the dictionary are strings with the exception of P1_REVISION which is an integer.
To get information about the library version, use/read:
GPIO.VERSION

This provides a string with the X.Y.Z version format.

9. Interrupts

Aside from busy-polling, the library provides three additional ways of monitoring an input event:

The wait_for_edge() function

This function blocks the calling thread until the provided edge(s) is detected. The function can be called as follows:

```
GPIO.wait_for_edge(channel, GPIO.RISING)
```

The second parameter specifies the edge to be detected and can be GPIO.RISING, GPIO.FALLING or GPIO.BOTH. If you only want to limit the wait to a specified amount of time, a timeout can be optionally set:

```
# timeout is in milliseconds
GPIO.wait_for_edge(channel, GPIO.RISING, timeout=500)
```

The function returns the channel for which the edge was detected or None if a timeout occurred.

The event_detected() function

This function can be used to periodically check if an event occurred since the last call. The function can be set up and called as follows:

```
# set rising edge detection on the channel
GPIO.add_event_detect(channel, GPIO.RISING)
run_other_code()
if GPIO.event_detected(channel):
    do_something()
```

As before, you can detect events for GPIO.RISING, GPIO.FALLING or GPIO.BOTH.

A callback function run when an edge is detected

This feature can be used to run a second thread for callback functions. Hence, the callback function can be run concurrent to your main program in response to an edge. This feature can be used as follows:

```
# define callback function
def callback_fn(channel):
    print("Callback called from channel %s" % channel)
# add rising edge detection
GPIO.add_event_detect(channel, GPIO.RISING, callback=callback_fn)
```

More than one callback can also be added if required as follows:

```
def callback_one(channel):
    print("First Callback")
def callback_two(channel):
    print("Second Callback")
GPIO.add_event_detect(channel, GPIO.RISING)
GPIO.add_event_callback(channel, callback_one)
GPIO.add_event_callback(channel, callback_two)
```

The two callbacks in this case are run sequentially, not concurrently since there is only thread running all callback functions.
In order to prevent multiple calls to the callback functions by collapsing multiple events in to a single one, a debounce time can be optionally set:
# bouncetime set in milliseconds
GPIO.add_event_detect(channel, GPIO.RISING, callback=callback_fn,bouncetime=200)
If the edge detection is not longer required it can be removed as follows:
GPIO.remove_event_detect(channel)

10. Check function of GPIO channels
This feature allows you to check the function of the provided GPIO channel:
GPIO.gpio_function(channel)
The function returns either GPIO.IN or GPIO.OUT.

11. PWM
See samples/simple_pwm.py for details on how to use PWM channels.
The Jetson.GPIO library supports PWM only on pins with attached hardware PWM controllers. Unlike the RPi.GPIO library, the Jetson.GPIO library does not implement Software emulated PWM. Jetson Nano supports 2 PWM channels, and Jetson AGX Xavier supports 3 PWM channels. Jetson TX1 and TX2 do not support any PWM channels.
The system pinmux must be configured to connect the hardware PWM controlller(s) to the relevant pins. If the pinmux is not configured, PWM signals will not reach the pins! The Jetson.GPIO library does not dynamically modify the pinmux configuration to achieve this. Read the L4T documentation for details on how to configure the pinmux.