Programmable Buck Converter

Documentation, Schematics, Simulation Results

By

Dai Ahmed Tag
Mohamed Gamal Mohamadeen
Luai Waleed Abdelkarim

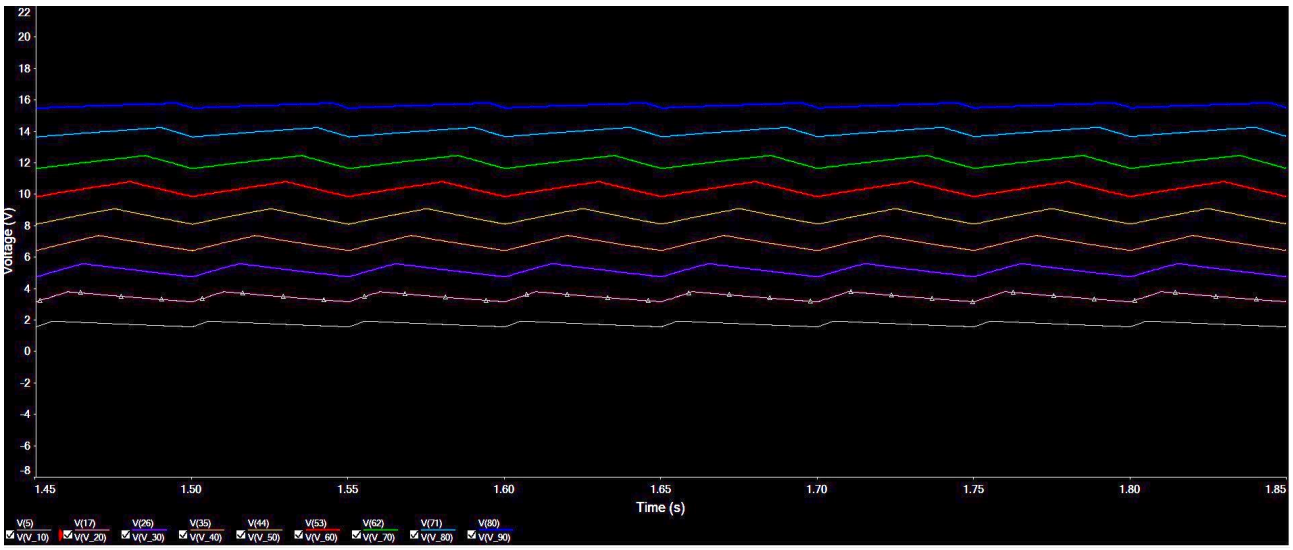
A buck converter is a type of DC-to-DC converter that reduces the voltage of an input DC signal to a lower DC output voltage. It is a type of switching converter, which means that it uses a transistor to switch the input voltage on and off rapidly. This creates a high-frequency square wave that is then used to regulate the output voltage.

The buck converter circuit consists of the following components:

- A switching transistor (IRF7403): This is the key component of the buck converter. It is used to switch the input voltage on and off rapidly.
- An inductor (20mH with 120Ω internal resistance): The inductor stores energy during the on-time of the switching transistor and releases it during the off-time. This helps to smooth out the output voltage.
- A capacitor (2420μF): The capacitor stores charge during the on-time of the switching transistor and discharges it during the off-time. This helps to regulate and filter the output voltage.
- A schottcky diode (1N5822) : The diode is used to prevent the current from flowing back through the inductor when the switching transistor is off, it also has a very low forward voltage, which means it provides higher efficiency than other diode models

The buck converter circuit works as follows:

1. The switching transistor is turned on. This allows current to flow through the inductor and the load resistor.
2. The inductor stores energy during this time.
3. The switching transistor is turned off. This stops the current from flowing through the inductor.
4. The inductor releases its stored energy through the load resistor.
5. The diode prevents the current from flowing back through the inductor.
6. The process repeats itself at a high frequency.

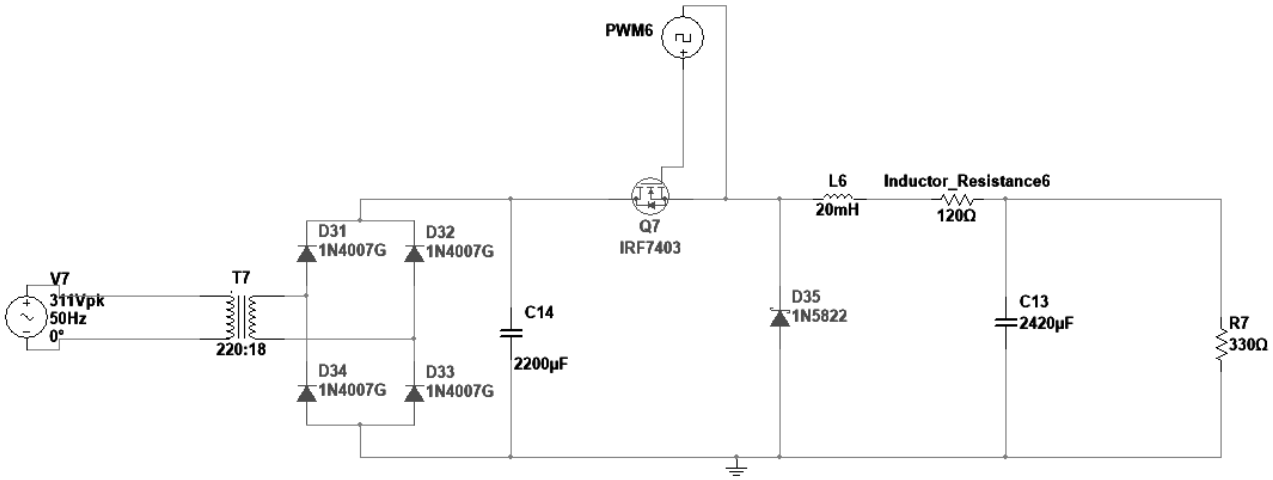


Output Voltage across a 330Ω with duty cycle ranges from 10%-90%

Rectifier circuit

The circuit consists of the following components:

- Two diodes: The diodes are used to change the direction of the current flow in each half cycle of the AC signal.
- A smoothing capacitor: The capacitor stores charge during the positive half cycles of the AC input signal and discharges it during the negative half cycles.



Circuit Schematics

And here is some actual results of the circuit

100Ω

V_{out} (V)	Frequency (Hz)	Cycle (%)
0.8	20	5
1.6	20	10
2.3	20	15
2.8	20	20
3.3	20	25
3.9	20	30
4.3	20	35
4.7	20	40
5.1	20	45
5.5	20	50
5.8	20	55
6.1	20	60
6.4	20	65
6.7	20	70
7	20	75
7.2	20	80
7.5	20	85
7.8	20	90
8	20	95
8.3	20	100

330Ω

V_{out} (V)	Frequency (Hz)	Cycle (%)
2.5	20	5
3	20	6
3.5	20	7
4	20	8
4.5	20	9
5	20	10.2
5.5	20	11.9
6	20	13.3
6.5	20	14.7
7	20	17
7.5	20	19
8	20	20.5
8.5	20	23
9	20	26
9.5	20	29
10	20	31
10.5	20	35
11	20	39
11.5	20	42
12	20	45

Programmable Buck Converter
Desktop App Documentation

By
Tasneem Eid Mostafa
Shahd Mahmoud Ali
Zyad Mohamed Elsayed

We used the technology of **Glade Interface Designer,** it is a graphical user interface builder for GTK, to build the application . The GTK+ and it is library provides an extensive collection of user interface building blocks such as text boxes, dialog labels, numeric entries, check boxes, and menus. the building blocks are called *widgets*. So we used Glade to place widgets in a GUI. Glade allows us to modify the layout and properties of these widgets.

```
def __init__(self):
        self.builder = Gtk.Builder()
        self.builder.add_from_file(UI_FILE)
        self.builder.connect_signals(self)

        self.window = self.builder.get_object("window")
        self.window_label = self.builder.get_object("Controller")
        self.window.show_all()
```

The
constructor of the class and The self parameter refers to the newly created object and it does the following :

1. Creates a Gtk.Builder object is used to load and manage the UI of the application.
2. Loads the UI file from the UI_FILE variable, The signals of the UI file are connected to the methods of the class so that the class can respond to user actions.
3. Connects the signals of the UI file to the methods of the class.
4. Gets the window and Controller objects from the UI file.
5. Shows the window object.

```
def ibutton_clicked(self, increase):
        self.step = self.builder.get_object("step")
        self.CycleIncrease = float(self.step.get_text())

    def dbutton_clicked(self, decrease):
        self.step = self.builder.get_object("step")
        self.CycleIncrease = -1 * float(self.step.get_text())

    def button_clicked(self, Apply):
        self.Entry = self.builder.get_object("Entry")
        self.Vout = float(self.Entry.get_text())
        self.Resistance = self.builder.get_object("resistance")
        self.resistance = float(self.Resistance.get_text())
        if (self.Vout / self.resistance) > 0.1:
            self.warninglabel = self.builder.get_object("warning")
            self.warninglabel.set_label("Couldn't apply voltage, current will exceed 100mA")
        else:
            self.desiredVoltage = self.Vout
            self.warninglabel.set_label("")

    def stop_clicked(self, Stop):
        self.desiredVoltage = 0

    def updatePWMlabel(self, cycle):
        self.label = self.builder.get_object("PWM")
        self.label.set_label(str(cycle) + "%")

    def get_voltageDesired_button_value(self, v_desired):
        self.Entry = self.builder.get_object("Entry")
        self.desiredVoltage = float(self.Entry.get_text())
```

Here there are some methods that handles all the events of the GUI.

```
def windows_destroy(self, window):
        Gtk.main_quit()
```

It is a method that is called when the user closes the window. The self parameter refers to the object that the method is attached to.

```
def main(self):
        Gtk.main_iteration()
```

It is the main method of the application. It only runs for one iteration, it is made to be run in another loop, It is responsible for starting the GTK+ main loop and running the application.

Programmable Buck Converter
Microcontroller Documentation

By
Luai Waleed Abdelkraim
Dai Ahmed Tag

We used the Raspberry Pi 3 Model B V1.2

- CPU: Quad core 1.2GHz Broadcom BCM2837 64bit
- RAM: 1GB LPDDR2 SDRAM
- Storage: microSD card (up to 2TB)
- Networking: 100Mbps Ethernet, 802.11n wireless LAN, Bluetooth 4.1
- I/O: 40-pin GPIO header, 4 USB 2.0 ports, HDMI port, composite video port, audio jack
- Power: Micro USB power supply (5V/2A)

The Raspberry Pi 3 is a small, powerful computer and a great option to go with.

The Code:

At first we have defined our timestep to be 0.01s

Also we have 3 predefined dictionaries for 3 different resistors to improve the precision of the output voltage across this range, this is mainly because the closed loop control is almost impossible with a controller without ADC.

We have also set the mode of the board to be Broadcom Channel system, we should only worry about this when defining our pins since this is a numbering system nothing more.

```
def setCycle(self, ui):
       if self.prevDesiredVoltage != ui.desiredVoltage:
           if ui.desiredVoltage != 0:
               if ui.resistance != 330 and ui.resistance != 560 and ui.resistance != 100:
                   if ui.resistance > 100 and ui.resistance < 330:
                       self.dutyCycle = self._100ohm[ui.desiredVoltage] if (ui.resistance < 215) else self._330ohm[ui.desiredVoltage]
                   elif ui.resistance > 330 and ui.resistance < 560:
                       self.dutyCycle = self._330ohm[ui.desiredVoltage] if (ui.resistance < 445) else self._560ohm[ui.desiredVoltage]
               elif ui.resistance == 100:
                   self.dutyCycle = self._100ohm[ui.desiredVoltage]
               elif ui.resistance == 330:
                   self.dutyCycle = self._330ohm[ui.desiredVoltage]
               elif ui.resistance == 560:
                   self.dutyCycle = self._560ohm[ui.desiredVoltage]
               else:
                   self.dutyCycle = 0
           else:
               self.dutyCycle = 0
       self.prevDesiredVoltage = ui.desiredVoltage
```

We have this method which utilises the 3 predefined dictionaries to provide a fairly accurate duty cycle for resistors between 100 and 560 ohms, it also sets the cycle to be zero for higher resistances, allowing user to choose his own output voltage using the manual control feature.

```
def manualControl(self, ui):
        self.dutyCycle = round((self.dutyCycle + ui.CycleIncrease + ui.CycleDecrease),3)
        ui.CycleIncrease = 0
        ui.CycleDecrease = 0
```

Here we have this manual control methods which reads an input from the user in the gui and changes the cycles gradually according it.

```
def pwmSignal(self, duty_cycle, frequency):
        if self.currentFrequency != frequency or self.currentCycle != duty_cycle:
            self.disablePWM()
            self.enablePWM(duty_cycle, frequency)
            print("PWM changed")

    def disablePWM(self):
        try:
            subprocess.call(["pkill", "-f", "pwm.py"])
        except:
            pass

    def enablePWM(self, duty_cycle, frequency):
        pwmScript = subprocess.Popen(
            ["python", "/home/proj/Documents/embproj/Pi3BScripts/pwm.py", "13", str(frequency), str(duty_cycle)])
        self.currentCycle = duty_cycle
        self.currentFrequency = frequency
        print("pwm enabled")
```

Here we have a very simple yet very versatile and important algorithm which manages the pwm signal and changes the duty cycle smoothly, it also ensures that the pwm will keep working concurrently with the main application loop, It waits for a change on its parameters before it disables and renables the pwm signal with the new parameters.