

Project title:

Python GUI linux with PCI express configuration

Students Name:Adam Bance

Supervisor (TUD): Agert Berisha

Project Manager (AMD): Deepesh Shakya

Project Manager (TUD): Micheal Gill

Subject of project: Summer Internship

Date:1/09/2023 – 26/04/2024

Declaration

This project entitled “Python GUI Linux with PCI express configuration” is my original work and has not been submitted for any other purpose to any other institute.

Signed: Adam Bance

Full Name: *Adam Bance*

Student No: X00183340

Acknowledgements

I would like to thank My supervisor Agert, My project Manager Deepesh. My lecture Micheal and also my fellow class mates for their help, advice, suggestions during the whole process. Even during difficult times, they were able to provide me with some support. Because of this I'm getting real exposure to the real world of engineering.

Abstract

- The capability to detect and interact with PCI which stands for Peripheral Component Interconnect devices is crucial in Linux systems for a range of applications from hardware level programming to diagnostics.

- Python provides versatile solutions to access and control these devices. By leveraging Linux utilities like (lspci) command which will be further explained, Python enables efficient device listing, identification, and attribute reading.

Aims

The student/programmer aims is to:

- Learn python with python tutorials
- Understand Linux commands
- Look into subprocess
- Look into GUI design
- Create buttons and other features

Objectives

The primary objective of the PCI Device Viewer application is to provide a user-friendly interface for simplifying the management and understanding of Peripheral Component Interconnect (PCI) devices for a broad range of users, from novices to technical experts. It aims to make detailed PCI device information readily accessible, thereby enhancing the user's ability to quickly identify and comprehend the functionalities and settings of these devices. This application is particularly valuable in system diagnostics and troubleshooting, offering a structured and clear presentation of PCI data crucial for resolving hardware issues

Contents

Declaration.....	2
Acknowledgements	2
Abstract.....	2
Aims.....	3
Objectives	3
1. Introduction	5
2. Oracle VM virtual box	6
3. Ubuntu	11
4. Subprocess	13
5. Python on jupyter notebook.....	14
5.1 Code explanation	18
5.2 Sub GUI #2	20
6. Lspci	21
7. Early stages of GUI development.....	23

7.1 Widget	25
7.2 The Search bar	25
8. Lspci Buttons.....	27
9.Password Function.....	28
9.1 Updated GUI layout	30
10. Applying alternating rows	31
11. Xilinx filter button.....	32
12. GUI theme and background	35
13. PCI.....	36
13.1. PCI connection with LSPCI.....	37
13.2. Control and Status Fields.....	37
13.3. Values of Plus and Minus	38
14.PCI configuration.....	39
15.PCI Code fragments.....	41
16. Conclusion	43
17. Environmental impacts of software.....	45
18.Final Full code.....	47
19.Appendix.....	67
20.Reference	69

1. Introduction

Python is a great platform its extensive libraries and high-level data structures allow for rapid development and testing. This enables you to quickly prototype solutions for device detection and interaction, thereby reducing development time. By utilizing Python for PCI device management in Linux, the Student was able to start leveraging a powerful, flexible, and continually evolving language, which can make the graphical user interface more efficient, maintainable, and extensible.



Figure 1: Python Logo

The student was able to quickly prototype solutions for detecting and interacting with PCI devices(which again will talked about later in the report), thanks to Python's intuitive syntax and comprehensive standard libraries. Python enables swift iteration cycles, which are invaluable in reducing both development time and the time-to-market for new features or applications.

The language's flexibility also comes to the fore when applied to creating graphical user interfaces (GUIs). Libraries such as Tkinter and PyQt make it possible to develop user-friendly interfaces with relative ease. By utilizing Python for PCI device management, the student effectively began leveraging a language that is both powerful and continually evolving. So, the idea is to create graphical user interface

that can show case the user devices instead of using a terminal and get its information. In this report the student will discuss ubuntu, Linux commands Aswell as subprocess, GUI designs, VS code (visual studios code), PCI express configurations in details and more.

2. Oracle VM virtual box

There are ways to go about implementing this project as it can be done by having an Ubuntu machine (physically) or by using a virtual one which would user friendly . Using windows would not be an option in this case as Windows has different kernels, software packaging and licensing to the operating system the student will be using which in this case is (Linux)..

The VirtualBox is a powerful x86 and AMD64/Intel64 product for enterprise as well as home users use. The VirtualBox an extremely feature rich, high-performance product for enterprise customers, is free and open source to the public.



Figure 2 VM Virtual Box logo

Using this VirtualBox machine allows you run it on windows, mac OS and other systems making it cross platform. From this you can have multiple Virtual machine simulations and

configure with various setting. An example would be the number of CPUs, and the amount of RAM.

It is possible to connect A USB Device to the VM as if it was to connect to guest operating system. In other words, the VM can access and use the USB device just like a physical computer would, allowing for data transfer, device recognition, and other functionalities. This feature is commonly used in virtualization environments to provide flexibility and expand the capabilities of virtual machines.

In order to get started the user would have to download this application from the VM website for their operating System.

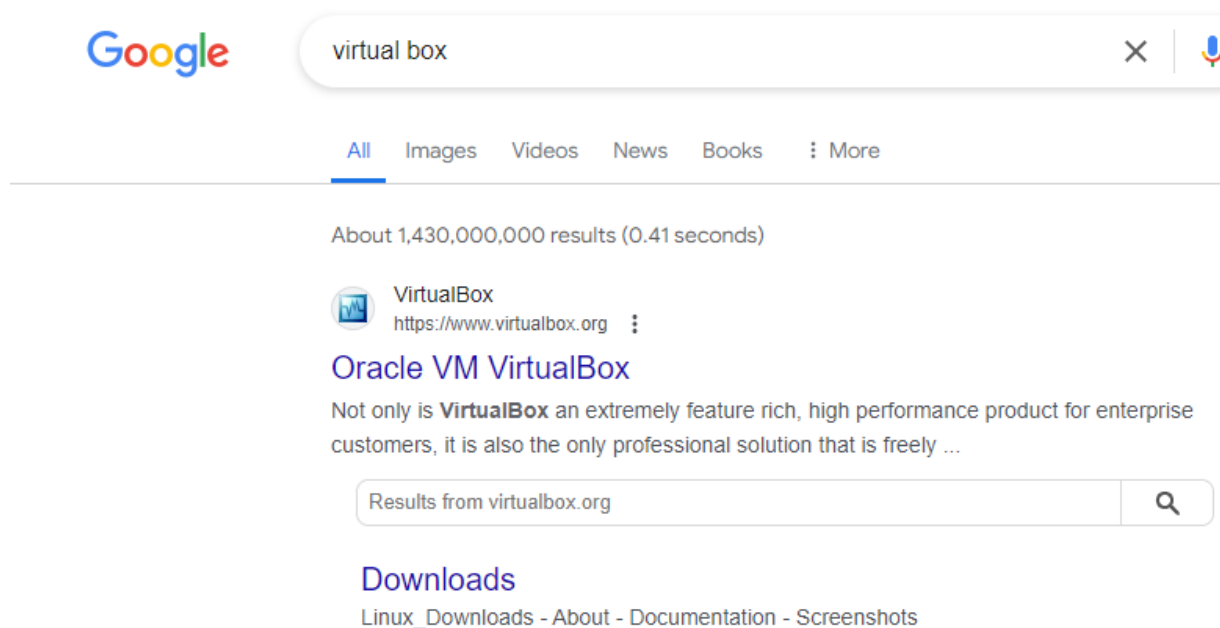


Figure 3 shows the VM website by search

The student simply searched for the word Virtual box and the official website popped up first after the search. Optionally you could actually click on the download link to take you straight to the downloads but for clarification the student clicked the link to the website.

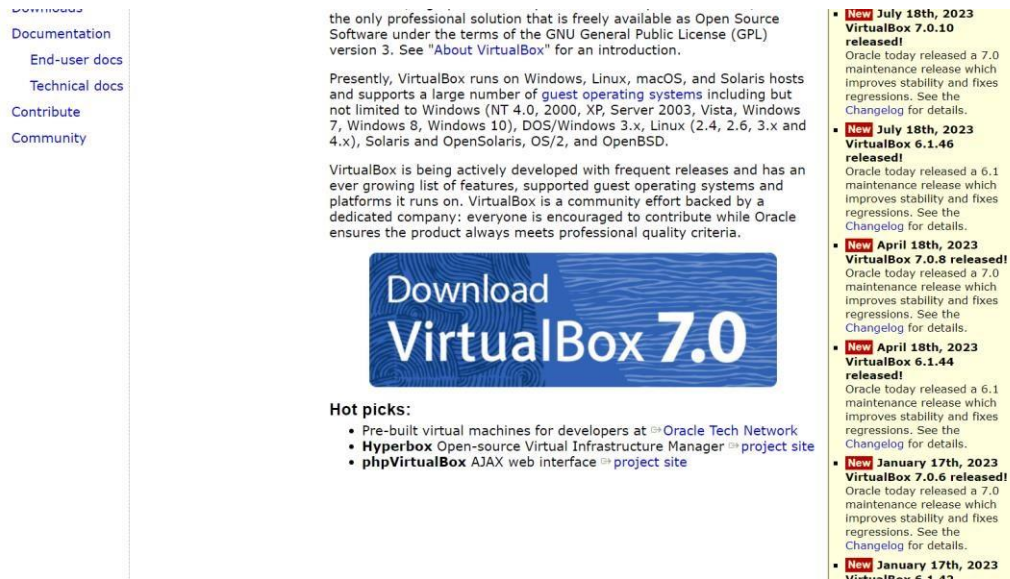


Figure 4 shows the download button inside the website

Upon opening the website there will be a big blue download button that will take you to the download page (shown above in fig 4) where you can download the application according to what operating system the User Has.

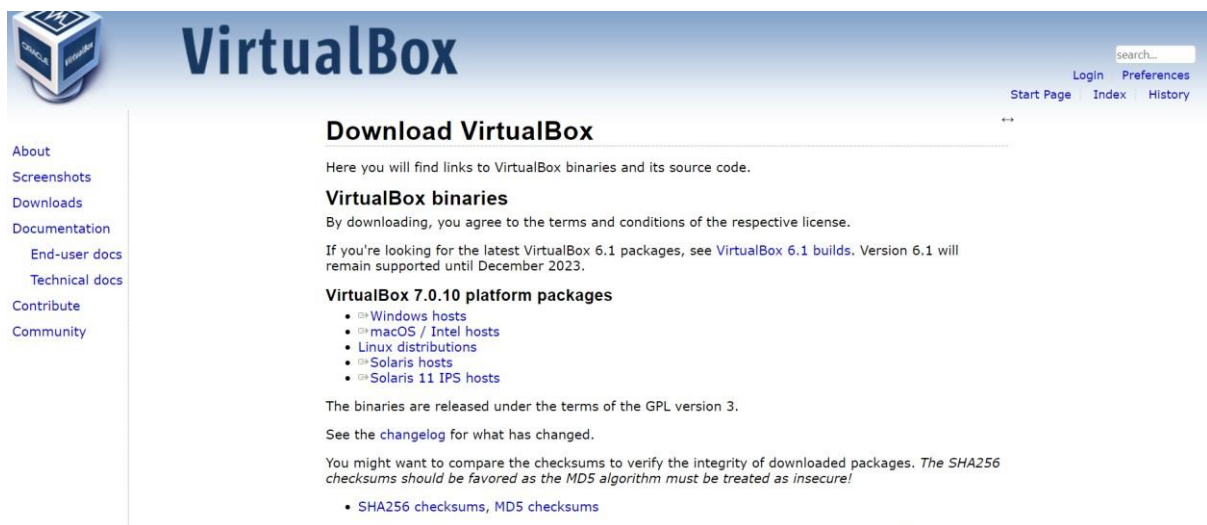


Figure 5 shows the download page for different operating Systems

From this page it shows the latest version of the application and depending on the Operating System, you can download a platform package that suits your PC. In this case the student is using Windows 11 so Windows hosts would be selected.

Once clicked it will download the application onto your PC and once its finished simply just click on it or if your unsure jest go to your files and go to download and click on it from there to set it up. When it is clicked, you should get this pop to start your setup.



Figure 6 shows the set up tab opening

From this point on you would then click next and go through the setup until a finish button appears down below replacing the next Button. After completing the setup, the student would then have to type the Virtual machine in the Windows Search bar and open the application. Upon opening it there shouldn't be an OS (operating System) inside. Since the Student had already had this installed there are two Systems listed which are Ubuntu and Windows 10.



Figure 7 shows main menu of virtual machine

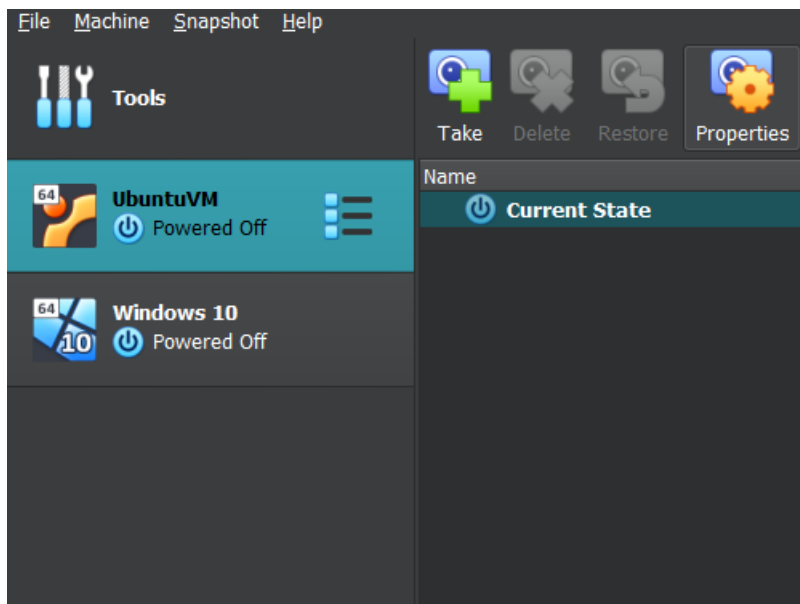


Figure 7 shows main menu of VM

To download Ubuntu the User would have to go into the Ubuntu website and download it from there. Once its downloaded, go back to the VM and go to the top right and select machine. Then the user would click on new and put in the Ubuntu details.

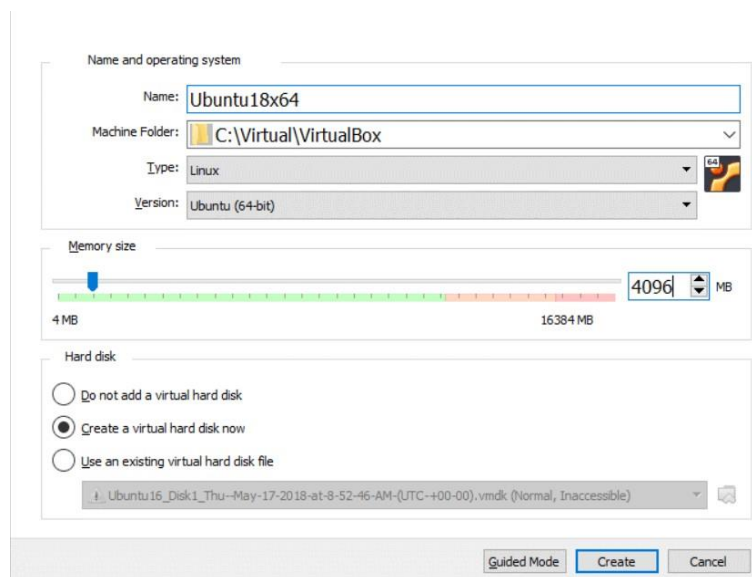


Figure 8 is from a website that shows Ubuntu set up on VM

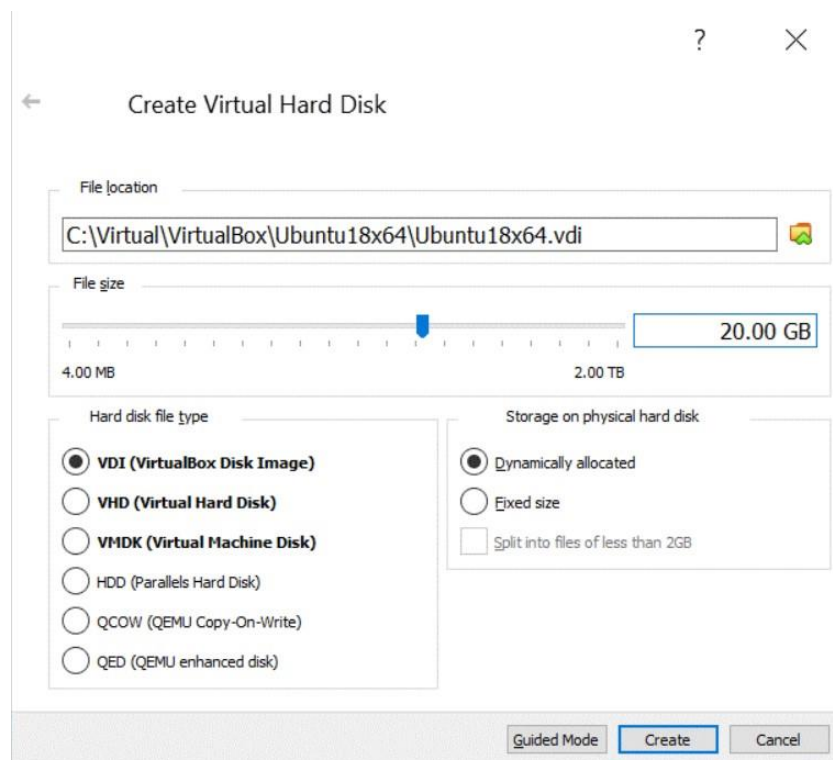


Figure 9 is from the same website that shows the final setup of ubuntu on the VM

The User would need to upload their file location, set the file size and determine the file type and storage. After everything is set, the user can now select the create button that will now display the Ubuntu System in the Virtual Machine.

To run the System, the Student would have to click on the green arrow at the top of the VM app to start.

3. Ubuntu

The student was given tasks within the first week of this the project. The first task was to

- 1) Open the terminal and download Jupiter notebook app
 - A nice easy platform to run Python code
- 2) Look into Linux commands such as, ls, man man and Lspci
 - man man: Access the manual pages for the man command itself.
 - Lspci: List all PCI devices connected to your system.

- 3) Observe and practice Subprocess with python
- 4) After completing the following tasks, the subsequent objective was to implement a basic GUI with buttons utilizing Linux command-line operations

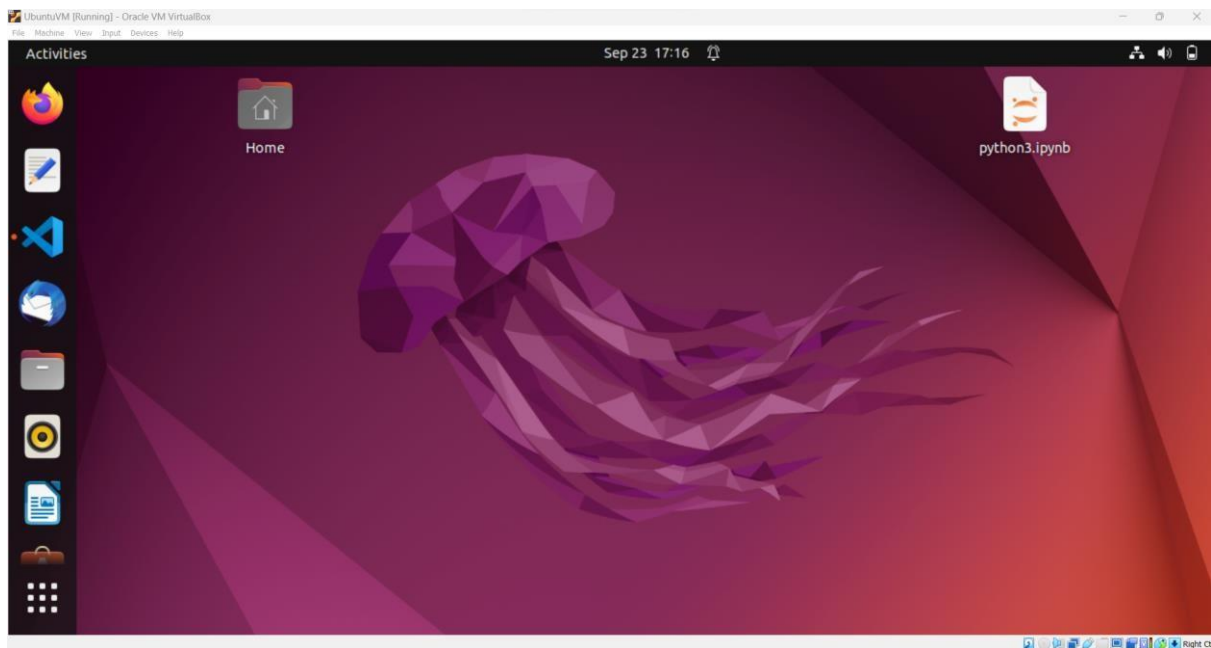


Figure 10 shows the home screen on ubuntu

Upon opening the terminal, the student entered this command step by step:

1. `sudo apt update` - Updates the package lists for available software packages in the system repositories.
2. `sudo apt upgrade -y` - Upgrades all installed packages to their latest versions, automatically confirming any prompts with the `-y` flag.
3. `sudo apt install python3 python3-pip` - Installs Python 3 and pip, the package installer for Python 3, facilitating Python package management.
4. `sudo apt install jupyter-notebook` - Installs the Jupyter Notebook application, a web-based interactive computing platform for Python.

For the early stages of creating button using sub process , Jupyter notebook was to create platform to perform tests with code and become familiar with sub process.

4. Subprocess

- To execute a command in a subprocess, the programmer first imports the subprocess module at the beginning of the script, indicating a need to interact with external commands. Following this import, the programmer intends to execute a specific command. For this purpose, they employ the subprocess. Run method. When they wish to list the files and folders in the current directory, they use the subprocess. Run('ls') command, where 'ls' is a Linux-specific command that serves this listing function. This approach is particularly effective in a Linux environment due to the native support of the 'ls' command.
- Additionally, to enhance the command with additional parameters or to execute it in a different shell environment, the programmer can pass arguments to the subprocess. Run method. For example, by adding shell=True, the command is executed within the shell, which can be crucial for certain types of commands or scripting scenarios. To include more arguments, such as listing files in a detailed format, the programmer simply inserts the argument -la before the shell argument. This extended format would then look something like subprocess. Run('ls -la', shell=True), allowing for a more comprehensive view of the files and directories.
- When a subprocess encounters an error during execution, it typically does not display an explicit error message directly to the user. Instead, it returns a non-zero error code, which is a common convention in many programming environments to indicate that an error has occurred. In the context of using the subprocess module in Python, this behavior is observed as well.
- To handle this situation and make the error more visible, the programmer can capture and display the returned error code. Assuming that the subprocess is invoked with a variable named 'A', as in A = subprocess.run(...), the programmer can then use print(A.returncode) to display the error code. This code is usually zero for a successful execution and non-zero for an error. For instance, if an error occurs, running this print statement might display '1', indicating that there was an error in the subprocess execution.
- However, just knowing that an error occurred may not be sufficient for debugging or understanding the problem. To obtain more detailed information about the nature of the error, the programmer can access the standard error output of the subprocess. This is done by printing A.stderr. Before this, it's important to ensure that the subprocess.run function is called with the argument stderr=subprocess.PIPE so that

the standard error output is captured. For example, the code might look like `A = subprocess.run(..., stderr=subprocess.PIPE)`. Then, `print(A.stderr)` will display the specific error message from the subprocess, offering more insight into what went wrong during its execution.

5. Python on jupyter notebook

- The objective here is to develop a basic graphical user interface (GUI) as a means of gaining familiarity with GUI programming concepts. To start with this task, the programmer chooses a fundamental approach: displaying the text "Hello World" on the GUI. This initial step is significant in GUI development as it represents the most basic form of interaction between the user interface and the user.
- Beginning with such a simple task allows the programmer to understand the foundational aspects of the GUI toolkit they are using, whether it be Tkinter, PyQt, or another framework. The process involves setting up a basic window or frame, which serves as the container for the GUI elements. Within this window, the programmer then creates a label or text element, which is programmed to display the text "Hello World". This is shown in Fig 12

```
from tkinter import *  
root = Tk()  
myLabel = Label(root, text="Hello world!")  
myLabel.pack()  
root.mainloop()
```

Figure 11 first code to implement GUI

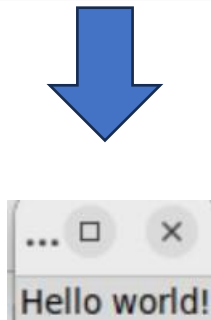
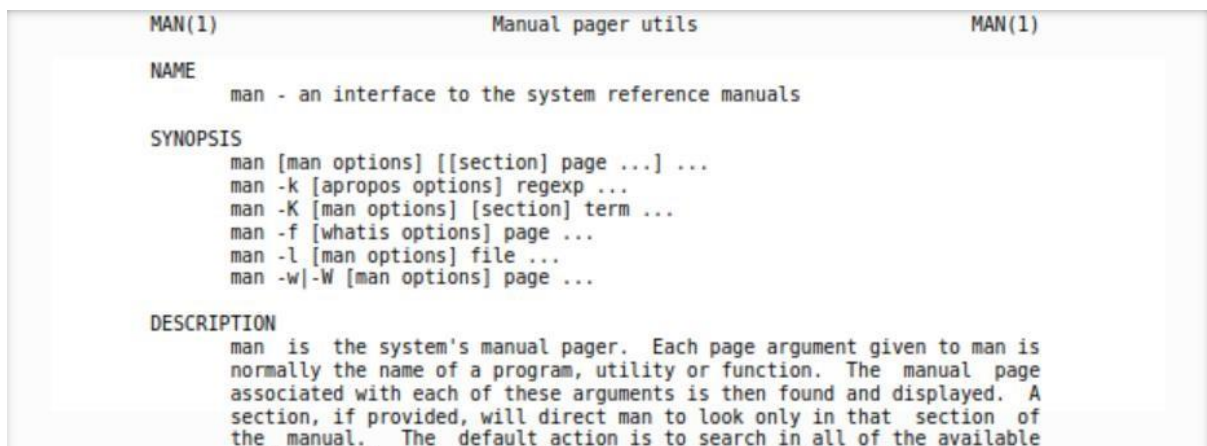


Figure 12 first GUI

- By focusing on this simple display, the programmer can learn about key concepts such as the main loop of the GUI, which keeps the window open and responsive to user actions, and the way components are added and managed within the window. This step, although basic, lays the groundwork for more complex GUI elements and interactions, such as buttons, input fields, and event handling. As the programmer becomes more comfortable with these initial concepts, they can gradually introduce more features and complexity into the GUI, building on the foundational knowledge gained from creating this simple "Hello World" display. The subsequent step in this GUI development project involves creating a more complex interface that features

buttons for executing Linux commands. The focal command for this endeavour is the 'man' command.

- The 'man' command in Linux is used to display manual pages, which are comprehensive documentation of various Linux commands, utilities, and programming interfaces. The name 'man' is an abbreviation for 'manual', highlighting its purpose as a reference tool. In the context of the GUI, the programmer aims to integrate this command so that users can easily access these manual pages through the interface.
- The GUI is designed to include buttons, each potentially corresponding to different Linux commands or aspects of the 'man' command itself. When a user clicks one of these buttons, the GUI is programmed to execute the 'man' command for the associated Linux command or option. The output from this command – which typically includes the name, synopsis, description, and other details of the command or utility – is then displayed within the GUI.



```
MAN(1)                                Manual pager utils                                MAN(1)

NAME
    man - an interface to the system reference manuals

SYNOPSIS
    man [man options] [[section] page ...] ...
    man -k [apropos options] regexp ...
    man -K [man options] [section] term ...
    man -f [whatis options] page ...
    man -l [man options] file ...
    man -w|-W [man options] page ...

DESCRIPTION
    man is the system's manual pager. Each page argument given to man is
    normally the name of a program, utility or function. The manual page
    associated with each of these arguments is then found and displayed. A
    section, if provided, will direct man to look only in that section of
    the manual. The default action is to search in all of the available
```

Figure 13 MAN output

- In figure 13, This output provides a detailed explanation of how the 'man' command works, including its options and usage straight from the Ubuntu terminal. By integrating this functionality into the GUI, the programmer not only enhances the usability of the application but also provides a valuable resource for users looking to learn more about Linux commands and their documentation.

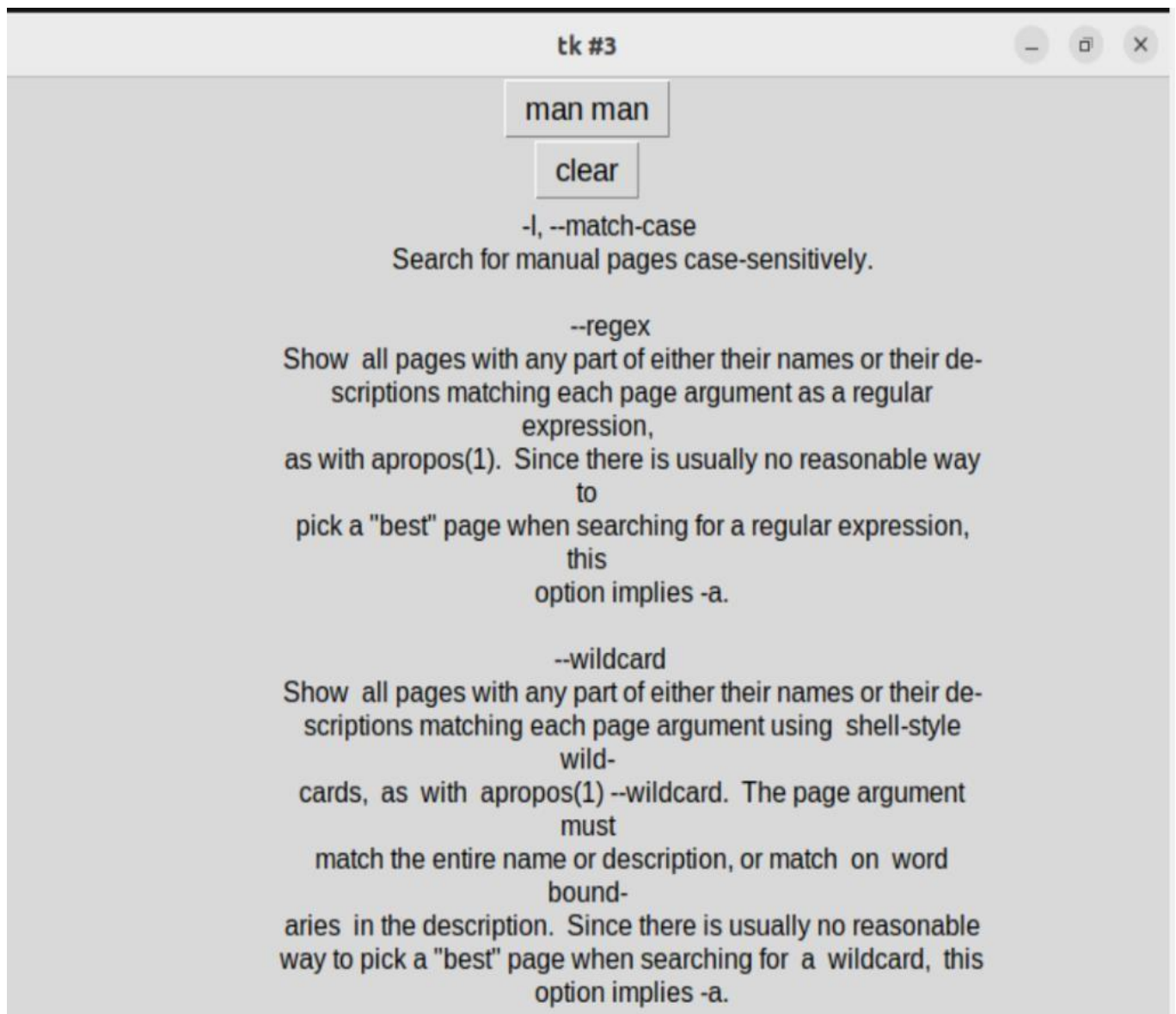


Figure 14 early GUI of Buttons and Output



```
import tkinter as tk
import subprocess

def click():
    subprocess.run(["man man"], shell=True)

def clear_terminal():
    subprocess.run(["clear"], shell=True)

root = tk.Tk()

manButton = tk.Button(root, text="man man", font=("Arial", 14), command=click)
manButton.pack()

clearButton = tk.Button(root, text="clear", font=("Arial", 14), command=clear_terminal)
clearButton.pack()
```

Figure 15 code for showing Output and clearing

5.1 Code explanation

1. Importing Libraries:

The script starts by importing two Python libraries:

tkinter (aliased as tk), which is used for building the GUI elements like windows and buttons.

2. subprocess, which is used for running shell commands.

Defining Functions:

click(): This function executes when the 'man' button in the GUI is clicked. It runs the Linux 'man -l' command using subprocess.run. The 'man -l' command is used to display manual pages in Linux. The shell=True parameter allows the command to be executed within the shell.

clear_terminal(): This function is called when the 'clear' button is clicked. It executes the 'clear' command using subprocess.run, which clears the terminal screen. The shell=True parameter is used here as well.

Setting Up the Main Application Window:

3. The script creates the main window (root) for the application using tk.Tk(). This window serves as the container for other GUI elements like Adding Buttons and search bar to the GUI.

A 'man' button is created using tk.Button. This button is configured to call the click() function when clicked. The button's text is set to 'man', and its font is set to Arial, size 14.

Similarly, a 'clear' button is created. This button is linked to the clear_terminal() function and is designed to clear the terminal screen when clicked. The button's text is set to 'clear', and it uses the same font styling as the 'man' button.

Arranging the Buttons in the Window:

Both buttons are added to the root window using the pack() method, which is a geometry manager in Tkinter. This method places the buttons one after the other in the window.

Starting the GUI Event Loop:

Finally, `root.mainloop()` starts the Tkinter event loop. This loop waits for events (like button clicks) and responds to them as programmed. It's essential for making the GUI interactive and responsive.

There were some issues and errors with this code as I was still using `from tkinter import` instead of `import tkinter`.

The student also had incorrect indentation which meant that I had lines of code that was outside the `clear terminal` function.

5.2 Sub GUI #2

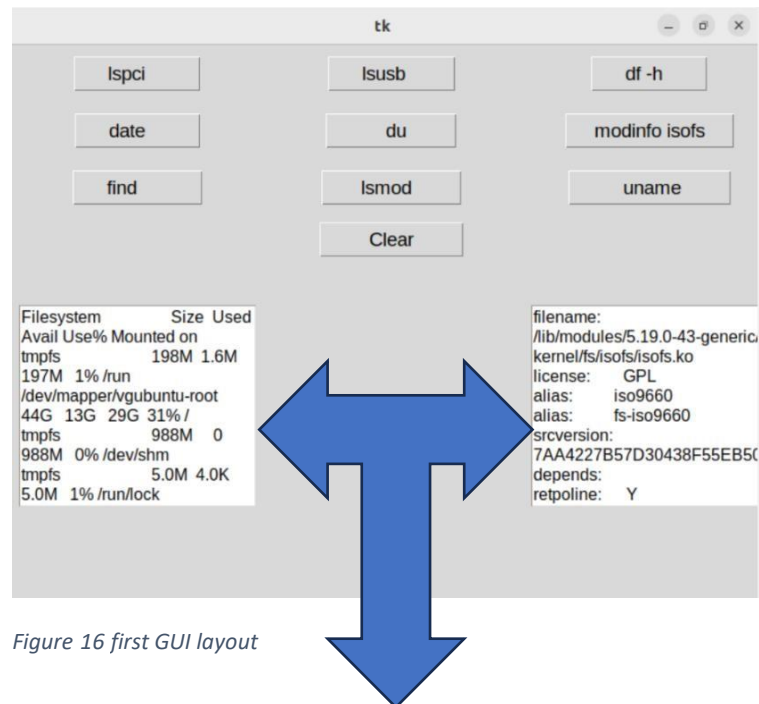


Figure 16 first GUI layout

More buttons with 2 text widgets

From the GUI above the programmer had created more command buttons using subprocess to execute the output into 2 text widgets. The text widgets on the left would be used mainly for output information and the one of the right would be used to check for detailed information about that specific command.

6. Lspci

- Lspci stands for “list PCI Devices”. It is a versatile tool in Linux systems, primarily used for displaying information about all PCI buses in the system and devices connected to them. Its functionality can be significantly expanded by using various options, each serving a specific purpose:
- **Basic lspci Command:**
 - Without any additional options, lspci generates a concise list of all PCI devices detected on the system. This output includes essential information such as the class of the device, its vendor, device ID, and specific location details like bus number, device slot, and function number. This basic output provides a quick overview of the PCI devices in the system.
- **lspci -k:**
 - The -k option extends the command's capability to show the kernel driver associated with each PCI device. This is particularly valuable for system administrators and developers to determine which specific kernel driver is managing each PCI device, aiding in system configuration and troubleshooting.
- **lspci -v:**
 - Employing the -v (verbose) option with lspci results in a more detailed display of device information. This includes additional technical specifics like IRQ (Interrupt Request) assignments, memory regions, I/O port addresses, and any kernel modules that are being used. This verbose output is useful for a deeper understanding of the hardware configuration and how the system interacts with each PCI device.
- **lspci -x:**
 - The -x option provides a hex dump of the PCI configuration space for each device. This advanced feature is particularly beneficial for low-level hardware debugging and analysis, offering an in-depth look at the configuration registers of PCI devices.
- **lspci -vvv:**
 - Using -vvv with lspci maximizes the verbosity level, revealing the most detailed information available about PCI devices. This includes in-depth details such as

device capabilities, extended register information, and more. Such detailed data can be crucial for comprehensive system diagnostics and advanced hardware troubleshooting.

- **lspci -D:**
 - The -D option modifies the output to include the numerical device IDs along with the usual device names. This feature is particularly useful for developers or for detailed hardware identification, especially when looking up specific devices in hardware databases or documentation.
- **lspci -n:**
 - With -n, lspci displays both the numerical IDs and the conventional names of the devices. This mode is often utilized for cross-referencing purposes, allowing users to match device names with their corresponding numerical identifiers.
- **lspci -tv:**
 - Finally, the -tv option generates a tree-like representation of the PCI devices. This hierarchical view reflects the bus topology, illustrating how devices are interconnected on the PCI bus. It provides a visual and structured overview of the device layout, useful in understanding the system's PCI bus structure.

7. Early stages of GUI development

```
def execute_lspci_command(command):
    result = subprocess.run(command, capture_output=True, text=True)
    lsmod_text.configure(state=tk.NORMAL)
    lsmod_text.delete("1.0", tk.END)
    lsmod_text.insert(tk.END, result.stdout)
    lsmod_text.configure(state=tk.DISABLED)

    # Clear the right text grid
    modinfo_text.configure(state=tk.NORMAL)
    modinfo_text.delete("1.0", tk.END)
    modinfo_text.insert(tk.END, "Select a device from the list on the left to view its information.")
    modinfo_text.configure(state=tk.DISABLED)

    # Make all devices clickable
    lsmod_text.tag_remove("selectable", "1.0", tk.END) # Remove existing tags
    for device in devices_list:
        index = lsmod_text.search(device, "1.0", tk.END)
        lsmod_text.tag_add("selectable", index, f"{index} + {len(device.split()[0])}c")
    lsmod_text.tag_configure("selectable", foreground="blue", underline=True)
```

Figure 17 executing commands with Lspci in GUI

- The `execute_lspci_command` function in this script is designed to interact with the system and update a graphical user interface accordingly. The student can through each snippet of code and explain its use for GUI development.

1The function takes a command as input, typically related to listing PCI (Peripheral Component Interconnect) devices on the computer. It uses `subprocess.run(command, capture_output=True, text=True)` to execute this command. The `capture_output=True` means that the output (information about the PCI devices) will be captured instead of being displayed directly on the screen. The `text=True` ensures that this captured output is in the form of readable text.

2The output from the Lspci command, which includes details about PCI devices, is stored in `result.stdout`. This information is crucial for understanding what PCI devices are present in the system and their details.

3 The function updates a text widget (a part of the graphical interface) named ``lsmod_text``. This widget is first set to a normal state using ``lsmod_text.configure(state=tk.NORMAL)``, allowing modifications to its content. The content of ``lsmod_text`` is cleared with ``lsmod_text.delete("1.0", tk.END)``, and then the `lspci` command output is inserted into it using ``lsmod_text.insert(tk.END, result.stdout)``. Finally, the widget is set to a disabled state using ``lsmod_text.configure(state=tk.DISABLED)``, making it read-only for the user.

4. Similarly, another text widget named ``modinfo_text`` is prepared. It's made editable, cleared, and a message prompting the user to select a device for more information is inserted.

5 The function also manages interactive elements in the ``lsmod_text`` widget. It removes any previous selections or special formatting tagged as "selectable". Then, it iterates through a list of devices (``devices_list``). For each device, it searches for its occurrence in ``lsmod_text`` and can potentially mark it or perform other actions (though this part of the functionality is not fully detailed in the given description).

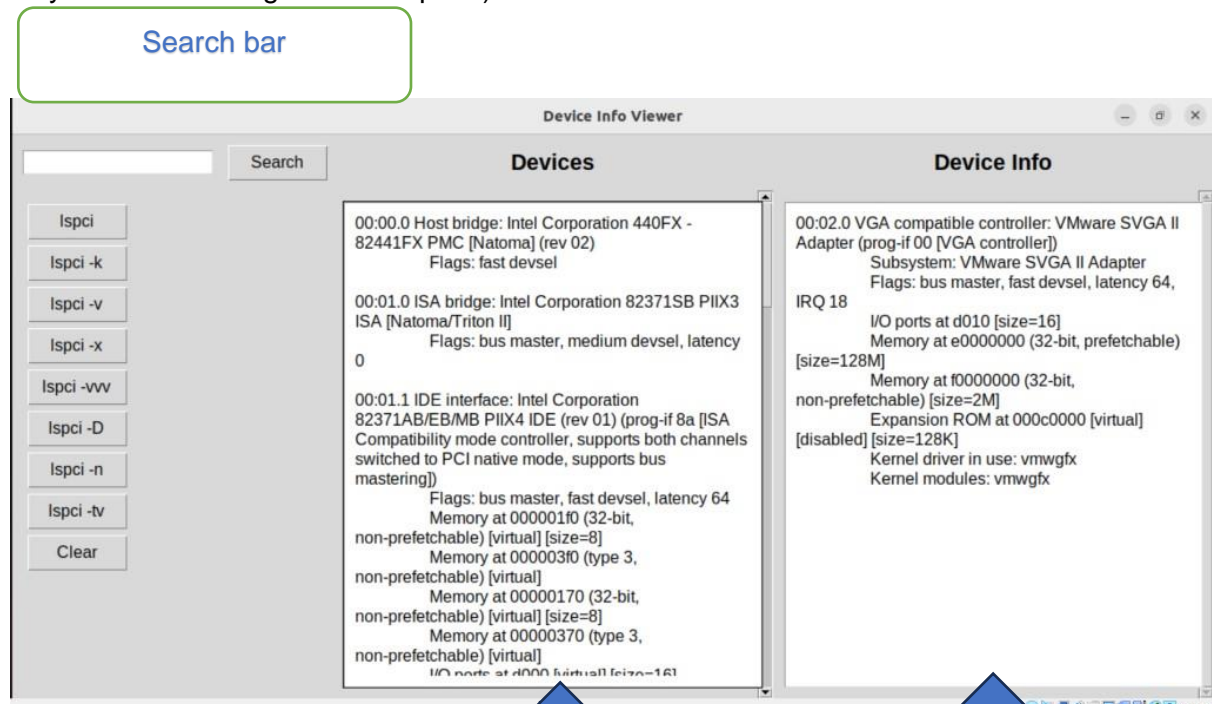


Figure 18 GUI layout

Devices listed
In text widget

Device detailed information

7.1 Widget

As seen above the purpose of the text widget on the left is list all devices detected in the PC and list them. And the right text widget is used for holding information from the selected device on the left widget with the selected buttons also.

7.2 The Search bar

- The search bar in this system serves as a convenient tool for quickly locating specific devices detected by the system. Users can enter a device number, bus, or name into the search bar, which then filters and displays all relevant devices that match the entered criteria. This functionality simplifies the process of identifying and accessing hardware components or connected devices, eliminating the need to manually sift through extensive lists of all detected devices, thereby enhancing the efficiency and user experience. The code is shown bellow in figure 19

```
self.search_entry = ttk.Entry(self.root, font=("Arial", 12))
self.search_entry.place(x=170, y=494, width=200, height=30)

self.search_button = ttk.Button(self.root, text="Search", command=self.search_devices)
self.search_button.place(x=380, y=494, width=70, height=30)
```

Figure 19 code for search bar and functionality

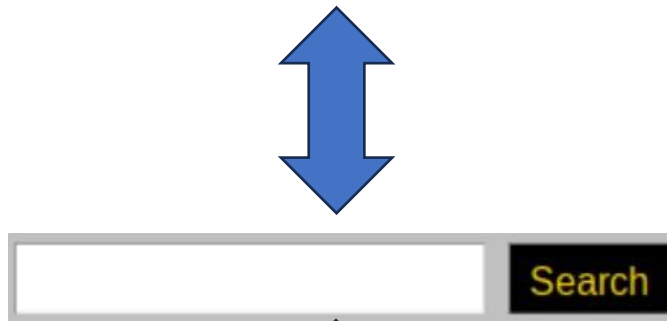


Figure 20 created search bar

```
def search_devices(self):
    query = self.search_entry.get().lower()
    result = self.run_command(f"echo '{self.sudo_password}' | sudo -S lspci")
    pattern = re.compile(r'^([0-9a-fA-F]{2}):[0-9a-fA-F]{2}\.([0-9a-fA-F]) (.+)')

    self.device_listbox.delete(0, tk.END)

    for line in result.splitlines():
        if query in line.lower():
            match = pattern.match(line)
            if match:
                device_code, device_name = match.groups()
                self.add_item_to_listbox(f"{device_code} {device_name}")
```

Figure 22 search functions for devices

- In fig 21 The search_devices function is designed to search for and display specific devices on a computer system based on a user query. It first retrieves the query from a search entry field and converts it to lowercase. Then, it executes a command (`lspci`) with superuser privileges to list all PCI devices on the system. A regular expression pattern is used to parse each line of the command output.
- The function clears any existing entries in a listbox widget, and for each line of the `lspci` output, it checks if the user's query is present. If a line contains the query and matches the regular expression pattern, the function extracts the device code and name from the line and adds this information to the listbox for display. This process effectively filters and displays device information based on the user's input.

8. The scrollbar features.

- This scrollbar is specifically designed for use with the device info text widget, enabling users to click and scroll through detailed information about devices. It

operates in tandem with the ``text.yview`` and ``scroll command`` functions to provide this interactive functionality.

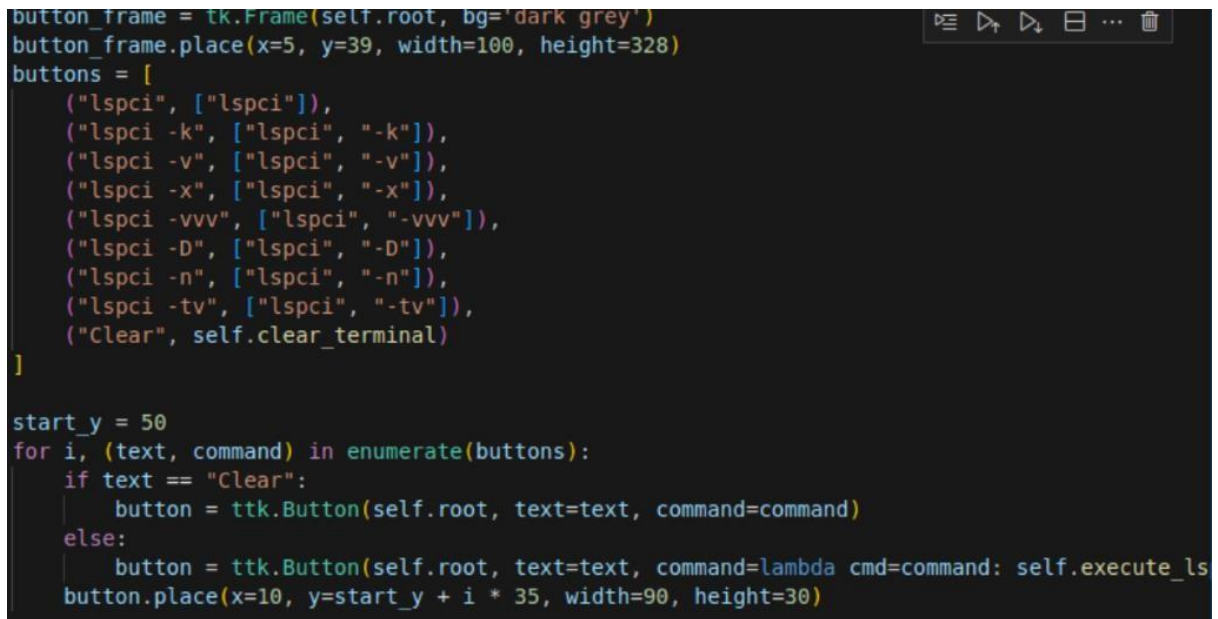
- Additionally, the device list box, which features selectable device entries, is equipped with its own scrolling mechanism, ensuring smooth navigation through the list of devices.

```
self.modinfo_scrollbar = tk.Scrollbar(self.root)
self.modinfo_scrollbar.place(x=1130, y=30, width=20, height=435)
self.modinfo_text.config(yscrollcommand=self.modinfo_scrollbar.set)
self.modinfo_scrollbar.config(command=self.modinfo_text.yview)
```

Figure 22 widgets scrollbar

8. Lspci Buttons

- The 'Button frame' in this context is configured to create a specific layout for buttons associated with LSPCI commands. It uses a grid system where the position of each button can be customized using row and column indices, as well as x and y coordinates. This allows for a flexible and adjustable button arrangement.
- A 'for loop', coupled with the 'enumerate' function, iterates through a list of buttons. The 'enumerate' function is particularly useful as it provides both the index and the value of each item in the list, aiding in the placement and identification of each button. Refer to fig 23 below.



```

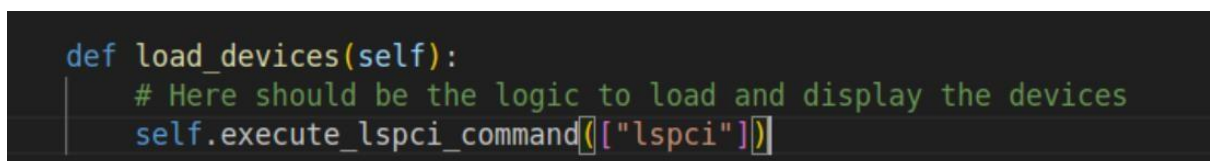
button_frame = tk.Frame(self.root, bg='dark grey')
button_frame.place(x=5, y=39, width=100, height=328)
buttons = [
    ("lspci", ["lspci"]),
    ("lspci -k", ["lspci", "-k"]),
    ("lspci -v", ["lspci", "-v"]),
    ("lspci -x", ["lspci", "-x"]),
    ("lspci -vvv", ["lspci", "-vvv"]),
    ("lspci -D", ["lspci", "-D"]),
    ("lspci -n", ["lspci", "-n"]),
    ("lspci -tv", ["lspci", "-tv"]),
    ("Clear", self.clear_terminal)
]

start_y = 50
for i, (text, command) in enumerate(buttons):
    if text == "Clear":
        button = ttk.Button(self.root, text=text, command=command)
    else:
        button = ttk.Button(self.root, text=text, command=lambda cmd=command: self.execute_ls)
    button.place(x=10, y=start_y + i * 35, width=90, height=30)

```

Figure 24 list of commands for buttons

- For the buttons themselves, the code includes a condition to check if the text associated with a button is 'clear'. If it is, a specific 'clear' button is created, equipped with a function to clear certain elements when clicked. For other buttons, the code generates them with labels based on the 'text' variable. These buttons are linked to specific actions through a 'lambda function', which is a brief, anonymous function used in Python. The lambda function is particularly handy because it allows passing an argument to a function without executing it immediately, thus enabling more dynamic interaction with the user interface.



```

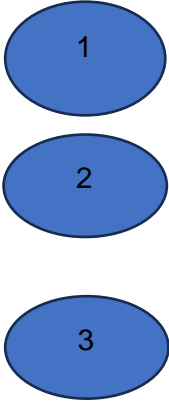
def load_devices(self):
    # Here should be the logic to load and display the devices
    self.execute_lspci_command(["lspci"])

```

Figure 25 code to load devices

9.Password Function

- To access sudo privileges for modifying device information, the system requires the entry of a sudo password. This security measure ensures that only authorized users can make changes at this elevated privilege level.



```
# GUI Components
self.setup_gui_components()

# Fetch sudo password and then start GUI
self.get_sudo_password()
if not self.verify_sudo_password():
    messagebox.showerror("Error", "Incorrect sudo password!")
    self.root.quit()
else:
    # Load devices upon correct password
    self.execute_lspci_command(["lspci"])
```

Figure 24 code to allow user access to capabilities in the output

1. The `self.get` function in this code is designed to prompt the user for their password, specifically for gaining sudo (superuser) privileges. This is achieved through the use of a graphical interface rather than the terminal. By importing `messagebox` from `tkinter`, the program requests the password through a pop-up message box, offering a more user-friendly experience.
2. The program then employs an `if` method to verify the correctness of the entered password. If the password is incorrect, it displays a notification to the user stating "incorrect password", and the subsequent line of code prevents the application from proceeding further.
3. In the event that the `if` condition is not met, meaning the password is correct, the `else` block is executed. This block calls the `execute_lspci_command` method within the class. This particular method is responsible for running the `lspci` command with the provided sudo password. This grants the necessary privileges to access detailed device information, thereby enabling the application to display this data to the user.

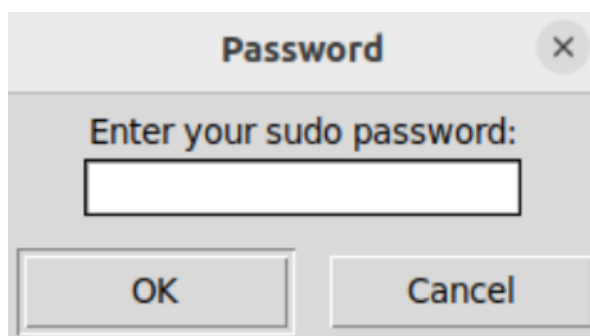


Figure 25 password tab

- The `def get_sudo_password` function in this code is designed to prompt the user for their sudo (superuser do) password. It displays a message box with the title "Password," instructing the user to enter their sudo password.
- Below this function, there is a `verify` function responsible for checking the correctness of the entered password. It uses a try-except block for error handling. If an error occurs during the verification process, the code moves to the except block.
- Inside the try block, the code executes a shell command using `subprocess.run`. This is where the `f'echo'` formatted string comes into play. It constructs a shell command that echoes the sudo password, stored in `self.sudo_password`. The `shell=True` parameter allows the command to be executed in the shell as a string.

```
def get_sudo_password(self):
    self.sudo_password = simpledialog.askstring("Password", "Enter your sudo password:", show='*')

def verify_sudo_password(self):
    try:
        output = subprocess.check_output(f"echo '{self.sudo_password}'|sudo -S echo 'Valid'", shell=True)
        if output == 'Valid':
            return True
    except:
        return False
    return False
def run_command(self, command):
    try:
        result = subprocess.check_output(command, shell=True, text=True)
    except subprocess.CalledProcessError:
        result = "Error executing the command."
    return result
```

Figure 26 code to ask sudo password

- The `text=True` parameter ensures that the output from this command is a text string, not bytes. The `.strip()` method is then used to remove any leading or trailing whitespace from this output string.
- An if statement checks if the output string is valid. If it is, the function returns `True`, indicating the sudo password was correct. If not, it proceeds to the next block, which returns `False`, signaling an incorrect password. A final `return False` acts as a safety net, used only if the `if output=valid` condition was not met, further ensuring the function's reliability in verifying the sudo password.

9.1 Updated GUI layout

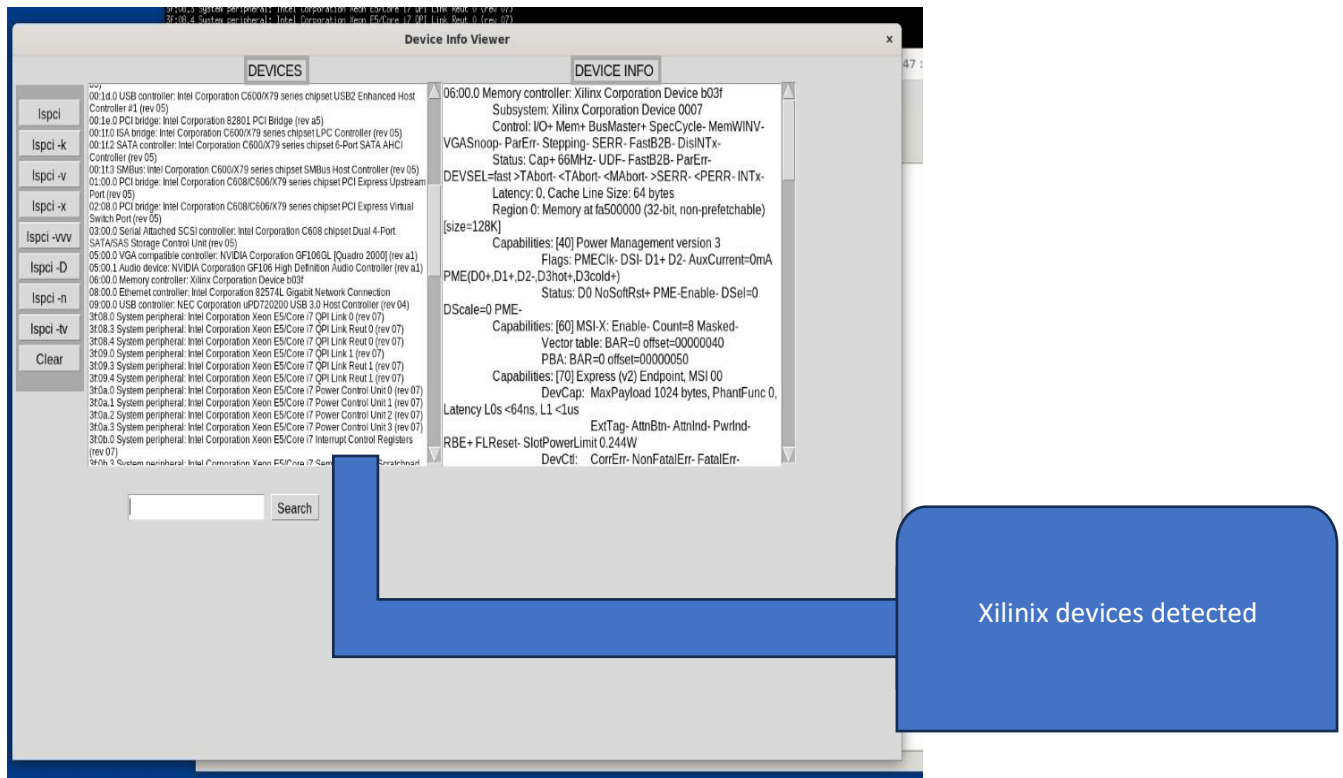


Figure 27 improved GUI design

10. Applying alternating rows

- The process of applying alternating rows in a list box involves defining a function specifically for adding items to it. This function systematically adds each new item to the list box, ensuring that alternate rows have distinct visual properties, such as different background colors, to enhance readability and visual appeal. This approach is particularly useful in distinguishing between successive rows, making the list box more user-friendly and easier to navigate.


```
def add_item_to_listbox(self, item):
    #print("adding:", item)
    index = self.device_listbox.size() # Get the next index
    self.device_listbox.insert(tk.END, item)

    # if ':' in item.split(" ")[0] and '.' in item.split(" ")[0]:
    if index % 2 == 0:
        self.device_listbox.itemconfig(index, {'bg': 'grey85'})
    else:
        self.device_listbox.itemconfig(index, {'bg': 'white'})
        self.device_listbox.pack(expand=1, fill="both")
```

Figure 28 alternating rows for device list

- The provided code snippet involves inserting elements into a graphical interface, specifically ending a loop that iterates through rows. The `tk.END` statement is used to signify the end of this loop. Additionally, an if-else construct is employed to alternate the background colors of the rows for better visibility and readability.
- This construct checks each row to determine if it's an odd or even number. Based on this determination, the row is highlighted either grey or white, creating a striped pattern that enhances the user interface's aesthetic and functional appeal.

11. Xilinx filter button

This function is meant to list Xilinx devices by parsing the output of the `lspci` command, but it contains a potential inconsistency in its search criteria (using "vga" instead of "Xilinx"). Also, it integrates with a graphical user interface, as indicated by the use of a listbox widget. Refer the following steps to figure 29.

1. **Function Definition** `def show_xilinx_devices(self):`

This line defines a function named `show_xilinx_devices` which is presumably a method of a class, given the `self` parameter.

2. Running a Command to List PCI Devices: `result =`

```
self.run_command(f"echo '{self.sudo_password}'|sudo -S lspci")
```

This line executes a command to list all PCI devices on the system. It uses ``sudo -S lspci`` to run the ``lspci`` command with superuser privileges. The ``sudo -S`` option allows the ``sudo`` command to read the password from the standard input, which is provided by the ``echo`` command.

3. Regular Expression Compilation:

```
pattern = re.compile(r'^([0-9a-fA-F]{2}:[0-9a-fA-F]{2}\.[0-9a-fA-F]) (.+)')
```

This line compiles a regular expression that is used to parse the output from the ``lspci`` command. The pattern is designed to match lines with a specific format typical of ``lspci`` outputs, capturing the device code and device name.

4. Search Key Initialization:

```
search_key = "vga"
```

Here, a variable ``search_key`` is set to the string ``"vga"``. This seems to be a remnant from a different version of the code or an error, as the comment suggests that it should be looking for lines containing "Xilinx".

5. Clearing Previous Listbox Entries:

```
self.device_listbox.delete(0, tk.END)
```

This line clears any existing entries in a listbox widget (presumably part of a graphical user interface), preparing it for new data.

6. Processing Each Line of Command Output:

The ``for`` loop (``for line in result.splitlines():``) iterates over each line of the ``lspci`` command output.

7. Searching for Xilinx Devices: `if search_key in line.lower():``

This conditional checks if the current line (converted to lowercase) contains the search_key` ("vga"). This part of the code seems inconsistent with the intended purpose (to search for Xilinx devices), and might require correction to check for "xilinx" instead of "vga".

8. Matching Line with Regular Expression:

```
match = pattern.match(line)`
```

If the line contains the search key, this code checks if it matches the compiled regular expression. If it does, `match` will contain the matching groups.

9. Extracting and Displaying Device Information: if

```
match:`    device_code, device_name = match.groups()

self.add_item_to_listbox(f"{device_code} {device_name}")`
```

If there's a match, the device code and device name are extracted and then added to the listbox for display.

```
def show_xilinx_devices(self):
    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S lspci")
    pattern = re.compile(r'^([0-9a-fA-F]{2}: [0-9a-fA-F]{2}\. [0-9a-fA-F]) (.
    search_key = "vga"

    self.device_listbox.delete(0, tk.END)

    for line in result.splitlines():
        if search_key in line.lower(): # Check if the line contains "Xilin
            match = pattern.match(line)
            if match:
                device_code, device_name = match.groups()
                self.add_item_to_listbox(f"{device_code} {device_name}")
```

Figure 29 code for xilinx devices only

12. GUI theme and background

The programmer was inspired by a youtuber who was doing a GUI design on how to go abouts applying different styling, font and background colour to the GUI design making it more appealing.

Here is an example from a heading in the GUI:

```
headinglabel_devices = tk.Label(self.root, text='Devices',  
headinglabel_devices.place(x=110, y=0)
```

Figure 30 label heading

```
font=('times new roman', 10, 'bold'), bg='grey20', fg='gold',
```

Figure 31 label font

```
bd=7, relief=tk.GROOVE, width=75, height=0)
```

Figure 32 label theme and size

This code is for creating and placing a label in a graphical user interface using Tkinter, a standard GUI toolkit in Python. The label, named `headinglabel_deviceinfo`, is created as an instance of `tk.Label` with several style and appearance parameters: it is attached to `self.root` (likely the main window of the application), displays the text 'Device info', and uses a 'Times New Roman' font of size 10 and bold style. The label's background (`bg`) is set to 'grey20', and its foreground (`fg`) to 'gold', creating a visually distinct appearance.

Additionally, it has a border (`bd`) of 7 pixels and a raised 'groove' relief, enhancing its 3D appearance. The label's width is set to 75, and its height is set to 0, which usually means the height is auto-adjusted to fit the content. Finally, the `place` method positions the label at coordinates (610, 0) in its parent window, `self.root`, specifying an absolute position within the GUI layout.

13. PCI

The Peripheral Component Interconnect, or PCI, is a standardized bus used in computers to connect hardware devices like network cards, sound cards, modems, and video cards to the motherboard. It was a parallel bus, synchronous to a single bus clock. Introduced by Intel in 1992, PCI became prevalent in computers from the late 1990s to the early 2000s, but it has since largely been superseded by PCI Express (PCIe), which offers improved speed and bandwidth over the parallel structure of traditional PCI.

PCI operates by being directly connected to the system's bus and allows attached hardware to communicate with the CPU without needing to match the CPU's native bus type. The initial version of PCI was a 32-bit bus, with the capability of 64-bit in later versions, and had speeds ranging from 133 MB/s to 533 MB/s depending on the bit width and clock speed. The development and subsequent revisions of PCI catered to the need for increased bandwidth and compatibility with newer hardware and operating systems.

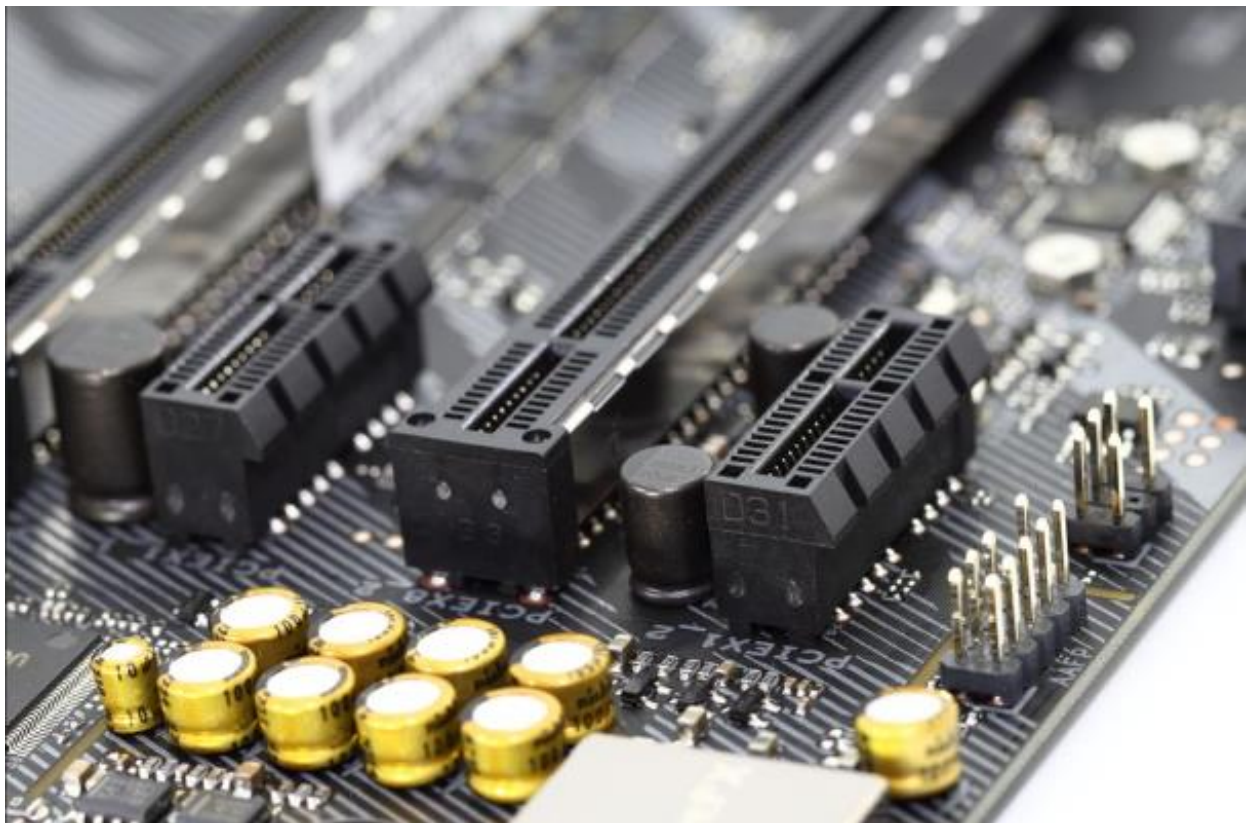


Figure 33 shows PCI hardware from a website

13.1. PCI connection with LSPCI

Linking PCI with lspci, the lspci command in Linux is used to list all the PCI buses and devices present in a system. This command helps users identify and configure hardware by displaying detailed information about PCI-connected devices, making it a valuable tool for system diagnostics and configuration in Linux environments.

The historical context of PCI's development and its technical specifications are an essential part of understanding how computers evolved to handle a variety of peripherals with increased efficiency and user-friendliness, particularly considering its compatibility with plug-and-play specifications. Despite its decline in favor of PCIe, PCI's legacy remains evident in various applications where backward compatibility or specific industrial requirements are a factor.

```
[00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01) (prog-if 8a [ISA
Compatibility mode controller, supports both channels switched to PCI native mode,
supports bus mastering]])
    Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
Stepping- SERR- FastB2B- DisINTx-
    Status: Cap- 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort-
<MAbort- >SERR- <PERR- INTx-
    Latency: 64
    Region 0: Memory at 000001f0 (32-bit, non-prefetchable) [virtual] [size=8]
    Region 1: Memory at 000003f0 (type 3, non-prefetchable) [virtual]
    Region 2: Memory at 00000170 (32-bit, non-prefetchable) [virtual] [size=8]
    Region 3: Memory at 00000370 (type 3, non-prefetchable) [virtual]
    Region 4: I/O ports at d000 [virtual] [size=16]
    Kernel driver in use: ata_piix
    Kernel modules: pata_acpi
```

Figure 34 shows detailed info from terminal by the command `LSPCI -V`

The output shown in the figure above is a result of the `lspci -v` command on a Linux system, which lists all PCI devices including detailed information about each device. The information provided is technical, and it corresponds to specifications and statuses of the PCI devices.

13.2. Control and Status Fields

The "Control" field generally lists the current settings for how the device is configured to interact with the rest of the system. This might include whether the device can respond to memory access and I/O space access, and whether it can act as a bus master.

The "Status" field reports the current status of the device, which may include whether it has detected any errors, the speed it is operating at, and other dynamic parameters.

13.3. Values of Plus and Minus

The plus Symbol (+) indicates that a particular feature or capability is enabled or set to '1'. It shows that the device has certain capabilities active. For example, in the context of an IDE interface, if 'BusMaster+' is shown as seen above , it means that Bus Mastering is enabled for the device, allowing it to initiate transactions on the PCI bus without CPU intervention.

The minus Symbol (-) indicates that a feature or capability is not enabled or set to '0'. It means that the specific function is not active or not supported by the device. For instance, 'VGA Snoop-' would indicate that the VGA Snoop feature is not enabled for this device.

14.PCI configuration

Every PCI device has a configuration space, which is a set of registers that describe the device's capabilities and control its operation. The configuration space contains important information such as:

- Device ID and Vendor ID: These are unique identifiers for the hardware.
- Status and Command(**control**) Registers: These control and report the status of the device.
- Class Code: This indicates what type of device it is (e.g., network card, graphics card).
- Interrupt Line and Pin: They show which interrupt line the device uses.
- Base Address Registers (BARs): They define the memory locations and I/O locations the device can respond to.

The lspci command in Linux reads the content of the PCI configuration registers and presents it in a human-readable format. It's a valuable tool for diagnosing hardware issues and ensuring that devices are correctly recognized and configured.

Device ID		Vendor ID		00h
Status		Command		04h
Class Code			RevID	08h
BIST	HdrType	LatTimer	\$LineSize	0Ch
BAR0				10h
BAR1				14h
SecLaTmr	SubBus#	SecBus#	PriBus#	18h
Secondary Status		IO Limit	IO Base	1Ch
Memory Limit		Memory Base		20h
Prefetch Mem Limit		Prefetch Mem Base		24h
Prefetchable Memory Base Upper 32 Bits				28h
Prefetchable Memory Limit Upper 32 Bits				2Ch
IO Limit Uppr 16Bits		IO Base Uppr 16Bits		30h
Reserved			CapPntr	34h
Expansion ROM Base Address				38h
Bridge Control		IntPin	IntLine	3Ch

Figure 35 Register fields from mind share Arbor application

The PCI configuration space is organized into a series of registers, which are typically 32 bits wide. These registers can be read using various tools or commands (like lspci in Linux).

1. Device and Vendor IDs (Offset 0x00): This 32-bit register contains the unique vendor ID in the lower 16 bits and the device ID in the upper 16 bits. These IDs are used by the operating system to identify the device and load the appropriate driver.
2. Status and Command Registers (Offset 0x04): This register includes the command register in the lower 16 bits, controlling the device's behaviour (e.g., memory space access, I/O space access, bus mastering). The status register in the upper 16 bits reports various device statuses such as interrupt status, whether the device has been configured, and if it has detected any errors.
3. Class Code, Subclass, Programming Interface, Revision ID (Offset 0x08): This register identifies the type of device, its specific function, and the programming interface it provides. The revision ID can be used to determine specific versions or features of the device.
4. BARs (Base Address Registers) (Offset starting at 0x10): These registers define the memory I/O addresses the device will use for its operation. The size of the memory region and whether it's memory or I/O mapped is also determined here.
5. Interrupt Line and Pin (Offset 0x3C): This shows which interrupt line the device uses and what pin it's connected to for interrupt requests.

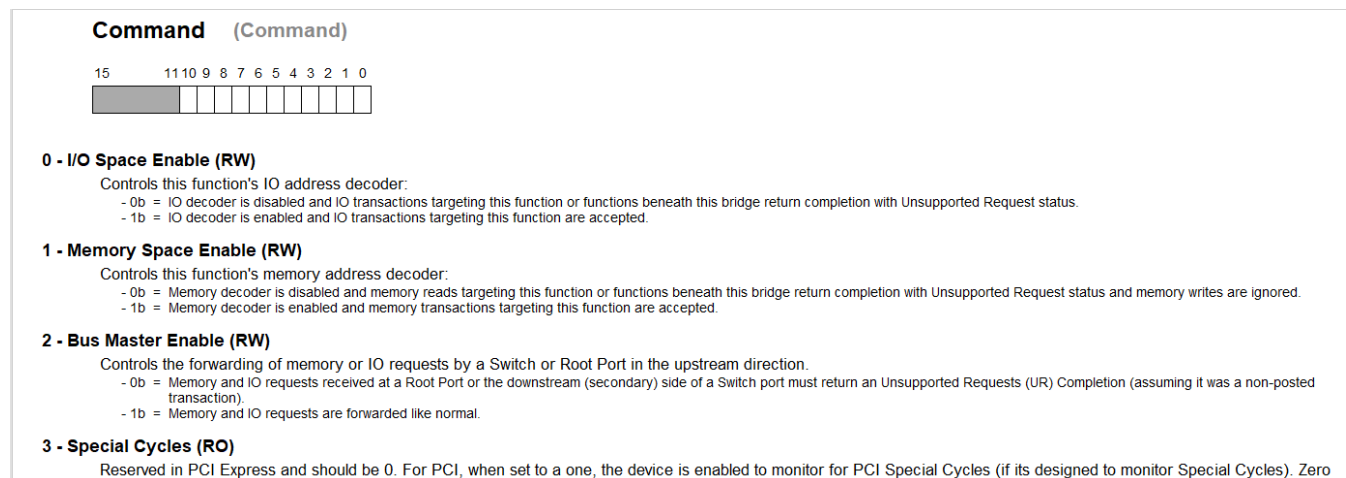


Figure 36 shows bit mapping and description of PCI from Mind share Arbor

15.PCI Code fragments

```
class DeviceViewer:
    def show_pci_capabilities(self):

        self.keywords_config_map = {
            "Vendor ID": [0, 2],
            "Device ID": [2, 2],
            "Command": [4, 2],
            "Status": [6, 2],
            "Revision ID": [8, 1],
            "Class code": [10, 3],
            "Cache Line": [12, 1],
            "M Latency T": [13, 1],
            "Header Type": [14, 1],
            "BIST": [15, 1],
            "Base Address": [16, 4],
            "reserved0": [41, 3],
            "SubsysVendor ID": [32, 2],
```

Figure 37 shows code fragment of PCI key words mapping

The function `show_pci_capabilities` is defining a dictionary called `self.keywords_config_map`. This dictionary appears to map PCI configuration space field names to their respective byte offsets and lengths within the PCI configuration space.

Each key in the dictionary is the name of a field within the PCI configuration space, and the associated value is a list with two numbers:

- The first number represents the offset (in bytes) from the beginning of the configuration space where the field is located.
 - The second number represents the length of the field (in bytes).
1. Vendor ID: [0,2] means the Vendor ID is at byte offset 0 and has a length of 2 bytes.
 2. Device ID: [2,2] means the Device ID follows immediately after at byte offset 2 and also has a length of 2 bytes.
 3. Command: [4,2] means the Command register is at byte offset 4 and has a length of 2 bytes.
 4. Status: [6,2] means the Status register is at byte offset 6 and has a length of 2 bytes.

5. Revision ID: [8,1] indicates the Revision ID is at byte offset 8 with a length of 1 byte.
6. ...and so on for the other fields.
7. The Base Address entry, [16,4], indicates that the base address registers start at byte offset 16 and span 4 bytes.

```
class DeviceViewer:
    def clicked(self, keyword, event):
        case "Vendor ID":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value, self.header_descriptions[keyword], self.bit_descriptions)
            insert_text_from_file_into_listbox('02h.txt')
        case "Status":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value, self.header_descriptions[keyword], self.bit_descriptions)
            insert_text_from_file_into_listbox('1.txt')
        case "Command":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value, self.header_descriptions[keyword], self.bit_descriptions)
            insert_text_from_file_into_listbox('2.txt')
```

Figure 37 code fragment of case statements for displaying info based on keyword clicked

This code fragment above is part of a Python class named `DeviceViewer`, which defines a method `clicked` that handles mouse click events on elements within the application's interface, presumably those representing different PCI configuration space fields

When a field in the GUI is clicked, the `clicked` method determines the action to take based on the keyword passed to it, which corresponds to the specific PCI field clicked by the user. For each case ('Vendor ID', 'Status', 'Command'), the method performs the following actions:

1. Prints a message to the console indicating that information about the clicked keyword is wanted.
2. Calls the method `self.draw_info_cells`, passing the keyword, some hexadecimal value presumably related to the keyword (though `hex_value` is not shown to be defined in this snippet), and a description fetched from `self.header_descriptions` using the keyword as a key. This method is responsible for displaying the information about the PCI field in the GUI.
3. Inserts text from a file into a list box widget, with the file being selected based on the keyword ('02h.txt' for 'Vendor ID', '1.txt' for both 'Status' and 'Command'). The text file likely contains a detailed description of the field and its values.

16. Conclusion

- The `DeviceViewer` application is a great tool that makes it easy for anyone to see and understand the settings of PCI (Peripheral Component Interconnect) devices on a computer. Built with Python and a graphical interface (GUI) using Tkinter, it allows users to interact with these devices in a simple way. The app's design is user-friendly, letting people find and learn about different PCI devices without needing deep technical knowledge.
- It fetches and shows detailed information about these devices, which is helpful for both regular users and tech-savvy individuals.
- While the app does a good job right now, there's room to make it even better in the future, like adding more features or making it work across different types of computers. Overall, `DeviceViewer` is a valuable tool that bridges the gap between complex computer hardware settings and everyday users, making it easier to manage and understand PCI devices.
- DeviceViewer stands out as a significant innovation, particularly for users who wish to navigate the complexities of PCI device configurations without delving into the intricacies of low-level system details. By leveraging a Python-based architecture complemented with a Tkinter graphical user interface, the application offers a level of accessibility and convenience previously unavailable to the average computer user.
- The strength of DeviceViewer lies in its ability to demystify the technicalities associated with PCI devices, presenting the information in a digestible format. This not only serves the curious and learning-oriented users but also provides tech enthusiasts with a detailed overview of device specifications, thereby catering to a broad user base.
- However, the journey of DeviceViewer doesn't end here. Recognizing that there is always room for enhancement is key to its evolution. Anticipating future updates that could include expanded compatibility and additional functionalities, DeviceViewer has the potential to grow into an even more comprehensive tool. The pursuit of these advancements could see it becoming an indispensable component of system management tools.

- In conclusion, DeviceViewer exemplifies a robust solution for interfacing with complex hardware settings. Its current capabilities facilitate a greater understanding of PCI devices, and its potential for future growth sets the stage for it to become an even more versatile and essential resource for users of all technical levels.

17. Environmental impacts of software

In this report, we delve into the surprisingly substantial environmental impact of the software development industry, an impact often overshadowed by more commonly discussed sectors like transportation and manufacturing. Despite its digital nature, software development is far from environmentally neutral, accounting for around 3% of global carbon emissions (Ref[10]), a figure astonishingly on par with the aviation industry. This footprint stems from various energy intensive phases in the software lifecycle. During the creation phase, substantial electricity is consumed in setting up and operating servers and databases, not to mention the energy used in coding and frequent software testing.

The deployment phase further escalates this consumption, especially in the operation of large data centers necessary for hosting software. These centres, crucial for load balancing and regular software updates, accounted for an estimated 240-340TWh of global electricity consumption in 2022 alone, approximately 1% of the global final electricity demand. Beyond these stages, the end-user's interaction with the software also contributes to energy use, factoring in the device's operational energy and the energy expended in data transmission.

The environmental cost is further compounded when considering the manufacturing of these devices. This report underscores the urgent need for the software industry to recognize and actively reduce its environmental footprint, mirroring efforts in other high impact industries. As the industry becomes increasingly aware of its role in environmental sustainability, the adoption of measures to mitigate this impact is not just beneficial but essential.

As the software development industry becomes increasingly cognizant of its significant role in environmental sustainability, the necessity for implementing measures to mitigate its ecological impact transitions from being merely advantageous to absolutely crucial. This realization is driving a paradigm shift, where companies are not only assessing their carbon footprints but are also actively seeking solutions that can integrate sustainability into their core operational strategies. This involves a multifaceted approach, encompassing everything from optimizing server efficiency to employing greener coding practices that reduce the energy demand during both development and execution phases.



Figure 39 laptop with a tree grown from it to tell us about environmental impacts from <https://res.cloudinary.com/>

Further exploration reveals that innovation in energy-efficient computing and the utilization of renewable energy sources in data centre's are becoming pivotal strategies. Companies are also focusing on enhancing the energy efficiency of their software products, which can have a significant downstream impact on the energy consumption of end-user devices. Moreover, there's an emerging trend towards cloud-based solutions, which, by virtue of shared resources, have the potential to be more energy-efficient than traditional on-premise solutions.

The industry is also witnessing a growing emphasis on 'green coding'—the practice of writing software in a way that minimizes energy consumption. This involves optimizing algorithms and reducing the computational intensity of tasks without compromising functionality. Additionally, more businesses are adopting lifecycle assessment (LCA) methods to evaluate the environmental impact of their software products from inception to end-of-life, thereby identifying key areas for improvement.

18.Final Full code

```

import tkinter as tk
from tkinter import ttk
from tkinter import simpledialog
from tkinter import Scrollbar
from tkinter import messagebox
from tkinter import Listbox, LEFT, BOTH, END, RIGHT, HORIZONTAL
from tkinter import font as tkFont

import re
import subprocess
import _json

class BitLevelMapping:
    def __init__(self):
        self.description = ""
        self.bitMap = {}

class DecriptionFieldMap:
    def __init__(self):
        self.description = ""
        self.fieldmap = {}

class DeviceViewer:
    def __init__(self):
        self.root = tk.Tk()

        self.padding = 2
        self.root.title("Device Info Viewer")
        self.root.configure(bg='silver')
        self.areas = {}
        self.device_identifier = ""
        #self.initialized_list_of_devices = False
        self.device_config_hex_buffer = []
        self.selected_rect_id = None
        self.header_text_id = None
        self.bit_descriptions = {}
        self.descriptionmap = {}
        # self.intializeddescriptionmap()
        # Fetch sudo password and then start GUI
        self.authenticate_and_initialize()
        #self.get_sudo_password()
    def authenticate_and_initialize(self):
        self.get_sudo_password()
        if self.sudo_password and self.verify_sudo_password():
            self.setup_gui_components()
            self.load_devices()
            self.execute_lspci_command(["lspci"])
        else:
            messagebox.showerror("Error", "Incorrect password or password
input canceled. Exiting...")
            self.root.destroy()
            return
        # GUI Components

        #self.canvas = tk.Canvas(self.root, bg='white', width=900,
height=600)

```



```

#self.show_pci_capabilities()
self.header_descriptions={
    "Vendor ID": "PCI Caps description",
    "Device ID": "PCI caps ID descr",
    "Status" : "Status Descr",
    "Command": "Command Descr",
    "Class code": "Device Status descr",
    "Revision ID": "Device Control desc",
    "BIST": "Link Caps Desc",
    "Header Type": "Link Status Descr",
    "M Latency T": "Link Control descr",
    "Cache Line": "slot Caps desc",
    "Base Address": "Slot status descr",
    "reserved0": "slot control descrip",
    "SubsysVendor ID": "Root Control descrip",
    "Subsystem ID": "Root Capabilites descrip",
    "Expansion ROM base Address": "Root Status descr",
    "Cap pointers": "Device Caps 2 desc",
    "reserved1": "Device Caps 22 desc",
    "reserved2": "Device Control2 desc",
    "int Line": "Device Control2 desc",
    "int pine": "Device Control2 desc",
    "min_Gnt": "Device Control2 desc",
    "Max_Lat": "Device Control2 desc",
    "Slot Status 2": "Device Control2 desc",
    "Slot Control 2": "Device Control2 desc",
    "Device Capabilities 2": "Description of Device Capabilities 2"
}

self.header_text_id = None

''' def intializedescriptionmap(self):
    description = "2:0 Max_Payload_Size Supported - This field indicates
the maximum payload size that the Function can support for TLPs."

    bitmap = {}
    bitmap["000b"] = "128 bytes max payload size "
    bitmap["001b"] = "256 bytes max payload size "
    bitmap["010b"] = "512 bytes max payload size "
    bitmap["011b"] = "1024 bytes max payload size "
    bitmap["100b"] = "2048 bytes max payload size "
    bitmap["101b"] = "4096 bytes max payload size "
    bitmap["110b"] = "Reserved"
    bitmap["111b"] = "Reserved"

    bit_level_mapping = BitLevelMapping(description, bitmap)
    description_field_map = DecriptionFieldMap("device capabilit",
{"2:0":bit_level_mapping})
    bitmap = {}
    bitmap["00b"] = 'phantom function is not avail'
    bitmap["01b"] = 'phantom function is not avail'
    bitmap["10b"] = 'phantom function is not avail'
    bitmap["11b"] = 'phantom function is not avail'

    bit_level_mapping = BitLevelMapping("4:3 Phantom Functions Supported
- This field indicates thesupport for use of unclaimed Function Numbers to
extend thenumber of outstanding transactions allowed by logicallycombining
unclaimed Function Numbers (called PhantomFunctions) with the Tag
identifier.",bit_level_mapping)
    description_field_map.fieldmap["4:3"] = BitLevelMapping

```

```

        self.descriptionmap['PCI caps ID'] = description_field_map'''
    # some JSON:
    x = '{"Vendor ID": { "description": "15:0 Vendor ID - PCI-SIG assigned. Analogous to the equivalent field in PCI-compatible Configuration Space. This field provides a means to associate an RCRB with a particular vendor.", "fields": {"Vendor ID":[{"colour": "black", "description": ""}] } } }'

    y = '{"Device ID": { "description": "31:16 Device ID - Vendor assigned. Analogous to the equivalent field in PCI-compatible Configuration Space. This field provides a means for a vendor to classify a particular RCRB.", "fields": {"Device ID":[{"colour": "black", "description": ""}] } } }'

    def setup_gui_components(self):
        # Create and style GUI components
        self.style = ttk.Style(self.root)
        self.style.configure("TButton", font=("Arial", 12), padding=5,
background='black', foreground='gold')
        self.style.configure("TLabel", font=("Arial", 14))

        #self.lsmid_text = tk.Text(self.root, font=("Arial", 9),
wrap="word")
        #self.lsmid_text.place(x=110, y=30, width=500, height=440)
        #self.device_listbox.pack(side=LEFT, fill=BOTH)

        self.xilinx_filter_frame = tk.Frame(self.root, bg='dark grey')
        self.xilinx_filter_frame.place(x=5, y=384, width=100, height=50)
        self.xilinx_filter_button = ttk.Button(self.root, text="Xilinx
filter", command=self.show_xilinx_devices)
        self.xilinx_filter_button.place(x=9, y=395, width=90, height=30)

        ''' self.vga_filter_button= ttk.Button(self.root, text="vga filter",
command=self.show_vga_devices)
        self.vga_filter_button.place(x=9, y=395, width=90, height=30)
        '''

        self.device_list_frame = tk.Frame(self.root, width=500, height=440)
        self.device_list_frame.place(x=110, y=30, width=500, height=440)
        #self.device_list_frame.pack(expand=True, fill=BOTH)
        self.device_list_frame.pack_propagate(0)

        self.device_listbox = Listbox(self.device_list_frame, font=("Arial",
12),selectbackground="Blue", highlightbackground="Blue")
        self.device_listbox.place(x=1, y=4, width=600, height=430)
        #
        self.device_list_scrollbar = Scrollbar(self.device_list_frame)
        self.device_list_scrollbar.pack(side=RIGHT, fill=BOTH)

        self.device_listbox.config(yscrollcommand=self.device_list_scrollbar.set)
        self.device_list_scrollbar.config(command=self.device_listbox.yview)

        self.device_list_hscrollbar = Scrollbar(self.device_list_frame,
orient=HORIZONTAL)
        self.device_list_hscrollbar.pack(side=tk.BOTTOM, fill=tk.X)

        self.device_listbox.config(xscrollcommand=self.device_list_hscrollbar.set)

        self.device_list_hscrollbar.config(command=self.device_listbox.xview)

        # self.device_listbox.tag_configure('odd' , background='grey85')
        #self.device_listbox.tag_configure('even' , background='white')
        '''self.lsmid_text.tag_configure('odd', background='grey85')

```

```

self.lsmode_text.tag_configure('even', background='white')
self.lsmode_text.bind("<1>", self.show_device_info)'''
self.device_listbox.bind("<<ListboxSelect>>", self.show_device_info)
self.modinfo_text = tk.Text(self.root, font=("Arial", 9, 'bold'),
wrap="word")
self.modinfo_text.place(x=610, y=30, width=520, height=440)
self.modinfo_text.insert('end', 'Please select a Device and a
command')
self.modinfo_text.tag_configure('odd', background='grey85')
self.modinfo_text.tag_configure('even', background='white')

#self.lsmode_scrollbar = tk.Scrollbar(self.root)
#self.lsmode_scrollbar.place(x=610, y=30, width=20, height=435)
#self.lsmode_text.config(yscrollcommand=self.lsmode_scrollbar.set)
#self.lsmode_scrollbar.config(command=self.lsmode_text.yview)

self.modinfo_scrollbar = tk.Scrollbar(self.root)
self.modinfo_scrollbar.place(x=1130, y=30, width=20, height=435)
self.modinfo_text.config(yscrollcommand=self.modinfo_scrollbar.set)
self.modinfo_scrollbar.config(command=self.modinfo_text.yview)
headinglabel_devices = tk.Label(self.root, text='Devices',
font=('times new roman', 10, 'bold'), bg='grey20', fg='gold', bd=8,
relief=tk.GROOVE, width=69, height=0)
headinglabel_devices.place(x=110, y=0)

headinglabel_deviceinfo = tk.Label(self.root, text='Device info',
font=('times new roman', 10, 'bold'), bg='grey20', fg='gold', bd=7,
relief=tk.GROOVE, width=75, height=0)
headinglabel_deviceinfo.place(x=610, y=0)

self.pci_express_button_frame = tk.Frame(self.root, bg='dark grey')
self.pci_express_button_frame.place(x=739, y=480, width=203,
height=44)
self.pci_express_button = ttk.Button(self.root, text="PCI
Capabilities", command=self.show_pci_capabilities)
self.pci_express_button.place(x=750, y=487, width=184, height=30)
# Search bar and button
self.search_entry = ttk.Entry(self.root, font=("Arial", 12))
self.search_entry.place(x=170, y=494, width=200, height=30)

self.search_button = ttk.Button(self.root, text="Search",
command=self.search_devices)
self.search_button.place(x=380, y=494, width=70, height=30)

# Command buttons
headinglabel_devices = tk.Label(self.root, text='Commands',
font=('times new roman', 10, 'bold'), bg='grey20', fg='gold', bd=8,
relief=tk.GROOVE, width=13, height=1)
headinglabel_devices.place(x=0, y=0)

button_frame = tk.Frame(self.root, bg='dark grey')
button_frame.place(x=5, y=39, width=100, height=328)
buttons = [
    ("lspci -m", ["lspci", "-m"]),
    ("lspci -k", ["lspci", "-k"]),
    ("lspci -v", ["lspci", "-v"]),
    ("lspci -x", ["lspci", "-x"]),
    ("lspci -vvv", ["lspci", "-vvv"]),
    ("lspci -D", ["lspci", "-D"]),
    ("lspci -n", ["lspci", "-n"]),
    ("lspci -nn", ["lspci", "-nn"]),

```

```

        ("Clear", self.clear_terminal)
    ]

    start_y = 50
    for i, (text, command) in enumerate(buttons):
        if text == "Clear":
            button = ttk.Button(self.root, text=text, command=command)
        else:
            button = ttk.Button(self.root, text=text, command=lambda
cmd=command: self.execute_lspci_command(cmd))
            button.place(x=10, y=start_y + i * 35, width=90, height=30)

    def get_sudo_password(self):
        self.sudo_password = simplifiedialog.askstring("Password", "Enter your
sudo password:", show='*')
        if self.sudo_password is None: # If user clicks cancel or closes
the dialog
            messagebox.showinfo("Info", "Password input canceled.
Exiting...")
            self.root.destroy()
            return # Exit the method
        if not self.verify_sudo_password():
            messagebox.showwarning("INCORRECT PASSWORD", "Incorrect
password, ")
            self.root.destroy() # Exit the application
            return

    def load_devices(self):
        # Here should be the logic to load and display the devices
        self.execute_lspci_command(["lspci"])

    def verify_sudo_password(self):
        try:
            subprocess.check_call(f"echo '{self.sudo_password}' | sudo -S -n
true", shell=True)
            return True
        except subprocess.CalledProcessError:
            return False
        except Exception as ex:
            print("Unexpected error:", ex)

            return False

        # return False
        #return False
    def run_command(self, command):
        try:
            result = subprocess.check_output(command, shell=True, text=True)
        except subprocess.CalledProcessError:
            result = "Error executing the command."
        return result

    def format_device_info(self, device_info):
        lines = device_info.split("\n")
        formatted_lines = []
        for idx, line in enumerate(lines):
            tag = 'odd' if idx % 2 == 0 else 'even'
            formatted_lines.append((line, tag))
        return formatted_lines

```

```

def execute_lspci_command(self, command):
    cmd = " ".join(command)
    if self.device_identifier:
        cmd = f"{cmd} -s {self.device_identifier}"
    print(f"...executing: {cmd}")
    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S
{cmd}")
    print(result)
    if self.device_identifier:
        detailed_info = result
        formatted_info = self.format_device_info(detailed_info)
        self.modinfo_text.configure(state=tk.NORMAL)
        self.modinfo_text.delete("1.0", tk.END)
        self.modinfo_text.insert(tk.END, formatted_info, 'please select a
Device and Command')
        #self.highlight_alternate_lines()
        self.modinfo_text.configure(state=tk.DISABLED)
    else:
        lines = result.split("\n")
        self.device_listbox.delete(0,END)

        #pattern = re.compile(r'^[0-9a-fA-F]{2}:[0-9a-fA-F]{2}\.[0-9a-fA-
F]$')

        for line in lines:
            if len(line) > 0:
                # self.modinfo_text.delete(1.0, tk.END) # Clear the
teDublinxt widget
                # self.modinfo_text.insert(tk.END, result) # Insert the
result

                self.add_item_to_listbox(line)

def add_item_to_listbox(self, item):
    #print("adding:", item)
    index = self.device_listbox.size() # Get the next index
    self.device_listbox.insert(tk.END, item)

    # if ':' in item.split(" ")[0] and '.' in item.split(" ")[0]:
    if index % 2 == 0:
        self.device_listbox.itemconfig(index, {'bg':'grey85'})
    else:
        self.device_listbox.itemconfig(index, {'bg':'white'})
        self.device_listbox.pack(expand=1, fill="both")

def highlight_alternate_lines(self):
    # Get the total number of lines
    total_lines = int(self.lsmode_text.index(tk.END).split('.')[0]) - 1
    for i in range(1, total_lines + 1):
        if i % 2 == 0:
            self.device_listbox.tag_add('even', f"{i}.0", f"{i}.end+1c")
        else:
            self.device_listbox.tag_add('odd', f"{i}.0", f"{i}.end+1c")

def show_device_info(self, event):
    index = self.device_listbox.selection_get()
    #self.lsmode_text.index(f"@{event.x},{event.y}")

    print(f"list selected: {index}")
    if index:
        #line_index = f"{index.split('.')[0]}.0"

```

```

        line_index = re.findall("[\S]+", index)[0];
        print(f"---selected device {line_index}")
        self.device_identifier = line_index
        #line = self.lsmode_text.get(line_index, f"{line_index} lineend")
        # selected_device = line_index #line.split()[0]
        '''
        command = f"echo '{self.sudo_password}'|sudo -S lspci -vvv -s
{self.device_identifier}"
        detailed_info = self.run_command(command)
        formatted_info = self.format_device_info(detailed_info)
        self.modinfo_text.configure(state=tk.NORMAL)
        self.modinfo_text.delete("1.0", tk.END)
        self.modinfo_text.insert(tk.END, formatted_info)
        #self.highlight_alternate_lines()
        self.modinfo_text.configure(state=tk.DISABLED)
        '''

        print(f"device_identifier: {self.device_identifier}")

def highlight_alternate_lines_modinfo(self):
    # Get the total number of lines
    total_lines = int(self.modinfo_text.index(tk.END).split('.')[0]) - 1
    for i in range(1, total_lines + 1):
        if i % 2 == 0:
            self.modinfo_text.tag_add('even', f"{i}.0", f"{i}.end+1c")
        else:
            self.modinfo_text.tag_add('odd', f"{i}.0", f"{i}.end+1c")

def format_device_info(self, detailed_info):
    formatted_info = ""
    sections = detailed_info.split("\n\n") # Split by blank lines

    for section in sections:
        lines = section.split("\n")
        for i, line in enumerate(lines):
            if i == 0: # If it's the first line of the section
                formatted_info += f"[{line}]\n" # Enclose in square
brackets
            else:
                formatted_info += f" {line}\n"
        formatted_info += "\n" # Separate sections

    return formatted_info

def search_devices(self):
    query = self.search_entry.get().lower()
    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S
lspci")
    pattern = re.compile(r'^([0-9a-fA-F]{2}: [0-9a-fA-F]{2})\.[0-9a-fA-F]
(.+)')

    self.device_listbox.delete(0, tk.END)

    for line in result.splitlines():
        if query in line.lower():
            match = pattern.match(line)
            if match:
                device_code, device_name = match.groups()
                self.add_item_to_listbox(f"{device_code} {device_name}")

def clear_terminal(self):
    self.modinfo_text.delete(0, tk.END)

```

```

self.modinfo_text.configure(state=tk.NORMAL)
self.modinfo_text.delete("1.0", tk.END)
self.modinfo_text.configure(state=tk.DISABLED)

def show_xilinx_devices(self):
    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S
lspci")
    pattern = re.compile(r'^([0-9a-fA-F]{2}):[0-9a-fA-F]{2}\.([0-9a-fA-F])
(.+)')
    search_key = "vga"

    self.device_listbox.delete(0, tk.END)

    for line in result.splitlines():
        if search_key in line.lower(): # Check if the line contains
            "Xilinx" (case-insensitive)
            match = pattern.match(line)
            if match:
                device_code, device_name = match.groups()
                self.add_item_to_listbox(f"{device_code} {device_name}")

def show_pci_capabilities(self):
    self.device_config_hex_buffer=[]
    if self.device_idenfifier == None:
        tk.Message(self,"Please select a device beforehand")
        return
    result=None

    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S
lspci -x -s {self.device_idenfifier}")
    print(f"output of command:\n{result}")
    lines = result.splitlines();
    for i in range(1, len(lines)):
        if len(lines[i]) > 0:
            line = lines[i].split(": ")[1]
            hexes = line.split()
            for hex_code in hexes:
                self.device_config_hex_buffer.append(hex_code)

    print(f"...hex codes: {self.device_config_hex_buffer}")
    capabilities_window = tk.Toplevel(self.root)
    capabilities_window.title("PCI caps")

    capabilities_window.geometry(self.root.winfo_geometry())

    headinglabel_capibilites = tk.Label(capabilities_window, text='PCI
Express Capability Structure', font=('times new roman', 10, 'bold'),
bg='grey20', fg='gold', bd=7, relief=tk.GROOVE, width=75, height=0)
    headinglabel_capibilites.place(x=20, y=1, width=1061, height=40)

    headinglabel_capibilites2 = tk.Label(capabilities_window, text='PCI
Terminal', font=('times new roman', 10, 'bold'), bg='grey20', fg='gold',
bd=7, relief=tk.GROOVE, width=75, height=0)
    headinglabel_capibilites2.place(x=484, y=163, width=600, height=23)

    self.info_label_text_id = None # Initialize to None

```



```

self.canvas = tk.Canvas(capabilities_window, bg='white')
self.canvas.place(x=20, y=40, width=450, height=400)
self.scrollbar = tk.Scrollbar(capabilities_window,
orient='vertical', command=self.canvas.yview)
self.scrollbar.place(x=470, y=40, width=15, height=400)
self.canvas.configure(yscrollcommand=self.scrollbar.set)

self.info_canvas = tk.Canvas(capabilities_window, bg='white')
self.info_canvas.place(x=484, y=40, width=600, height=123)

self.text_box2 = tk.Text(capabilities_window, bg='black', fg='green')
self.text_box2.place(x=484, y=185, width=600, height=440)
self.info_scrollbar = tk.Scrollbar(capabilities_window,
orient='vertical', command=self.text_box2.yview)
self.info_scrollbar.place(x=1070, y=187, width=15, height=441)
self.text_box2.configure(yscrollcommand=self.info_scrollbar.set)
self.info_scrollbar_x = tk.Scrollbar(capabilities_window,
orient='horizontal', command=self.text_box2.xview)
self.info_scrollbar_x.place(x=484, y=625, width=600)
self.text_box2.configure(xscrollcommand=self.info_scrollbar_x.set)
self.text_box2.insert('end', 'Please select a register on the left')

#self.canvas.configure(yscrollcommand=self.scrollbar.set)

self.frame = tk.Frame(self.canvas)
self.canvas.create_window((0, 0), window=self.frame, anchor='nw')

self.cell_sizes = [
    [(180, 40), (150, 40)],
    [(180, 40), (150, 40)],
    [(255, 40), (75, 40)],
    [(82.5, 40), (82.5, 40), (82.5, 40), (82.5, 40)],
    [(330, 240)],
    [(330, 40)],
    [(180, 40), (150, 40)],
    [(330, 40)],
    [(255, 40), (75, 40)],
    [(330, 40)],
    [(82.5, 40), (82.5, 40), (82.5, 40), (82.5, 40)]

]

self.info_cell_sizes = [
    [(550, 30)],
    [(550, 70)],
    [(550, 550)],

]

self.keywords = [
    "Device ID",
    "Vendor ID",
    "Status",
    "Command",
    "Class code",
    "Revision ID",
    "BIST",

```

```

        "Header Type",
        "M Latency T",
        "Cache Line",
        "Base Address",
        "reserved0",
        "Subsystem ID",
        "SubsysVendor ID",
        "Expansion ROM base Address",
        "reserved1",
        "Cap pointers",
        "reserved2",
        "Max_Lat",
        "min_Gnt",
        "int_pine",
        "int Line",

    ]

def read_info_text(file_name):
    with open(file_name, 'r') as f:
        return f.read()

self.keywords_description_map = {

    "Vendor ID":read_info_text('02h.txt'),
    "Device ID":read_info_text('info.txt'),
    "Status":read_info_text('02h.txt'),
    "Command":read_info_text('02h.txt'),
    "Revision ID":read_info_text('02h.txt'),
    "Class code":read_info_text('02h.txt'),
    "Cache Line":read_info_text('02h.txt'),
    "M Latency T":read_info_text('02h.txt'),
    "Header Type":read_info_text('02h.txt'),
    "BIST":read_info_text('02h.txt'),
    "Base Address":read_info_text('02h.txt'),
    "reserved0":read_info_text('02h.txt'),
    "SubsysVendor ID":read_info_text('02h.txt'),
    "Subsystem ID":read_info_text('02h.txt'),
    "Expansion ROM base Address":read_info_text('02h.txt'),
    "Cap pointers":read_info_text('02h.txt'),
    "reserved1":read_info_text('02h.txt'),
    "reserved2":read_info_text('02h.txt'),
    "int Line":read_info_text('02h.txt'),
    "int pine":read_info_text('02h.txt'),
    "min_Gnt":read_info_text('02h.txt'),
    "Max_Lat":read_info_text('02h.txt'),
    "Link control 2":read_info_text('02h.txt'),
    "Link Status 2":read_info_text('02h.txt'),
    "Slot Capabilities 2":read_info_text('02h.txt'),
    "Slot Control 2":read_info_text('02h.txt'),
    "Slot Status 2":read_info_text('02h.txt'),

}

self.keywords_config_map = {
    "Vendor ID":[0,2],
    "Device ID":[2,2],
    "Command":[4,2],

```

```

        "Status": [6, 2],
        "Revision ID": [8, 1],
        "Class code": [10, 3],
        "Cache Line": [12, 1],
        "M Latency T": [13, 1],
        "Header Type": [14, 1],
        "BIST": [15, 1],
        "Base Address": [16, 4],
        "reserved0": [41, 3],
        "SubsysVendor ID": [32, 2],
        "Subsystem ID": [34, 2],
        "Expansion ROM base Address": [36, 4],
        "Cap pointers": [40, 1],
        "reserved1": [41, 3],
        "reserved2": [41, 3],
        "int Line": [50, 1],
        "int pine": [51, 1],
        "min_Gnt": [52, 1],
        "Max_Lat": [53, 1],

    }

    self.info_keywords = [
        "Device Control",
        "",
        'description'
    ]

    self.draw_cells()
    self.canvas.config(scrollregion=self.canvas.bbox("all"))
    #self.update_button = tk.Button(capabilities_window, text="Update",
    command=self.update_hex_codes)
    #self.update_button.place(x=484, y=640, width=100, height=225) # 10
    pixels below the canvas at y=40+400+10

    # self.draw_info_cells()
    def update_info_label(self, keyword):
        #if self.info_label_text_id is not None:
        #    self.info_canvas.itemconfig(self.info_label_text_id,
        text=f"Configuration item: {keyword}")
        print("in method update_info_label...")

    def get_hex_value(self, keyword):
        start_pos, size_in_bytes = self.keywords_config_map[keyword]
        hex_value = self.device_config_hex_buffer[start_pos]
        if size_in_bytes > 1:
            for i in range(start_pos + 1, start_pos + size_in_bytes):
                hex_value = f"{self.device_config_hex_buffer[i]}{hex_value}"
        return hex_value

    def clicked(self, keyword, event):
        print("in 'onclicked' method ....")
        detailed_device_info = ""
        self.info_canvas.delete("info_text")

        rect_id = self.areas[keyword][0]
        if self.selected_rect_id is not None:
            self.canvas.itemconfig(self.selected_rect_id, fill='white')

```

```

self.canvas.itemconfig(rect_id, fill='green')
self.selected_rect_id = rect_id

'''if self.header_text_id:
    self.info_canvas.itemconfig(self.header_text_id, text=keyword)
else:
    print("Error: Header text ID not set.")

'''
def update_hex_codes(self):
    # Re-fetch the hex codes
    result = self.run_command(f"echo '{self.sudo_password}'|sudo -S
lspci -x -s {self.device_identifier}")
    print(f"output of lspci -x {self.device_identifier}\n{result}")
    lines = result.splitlines()
    new_hex_buffer = []
    for i in range(1, len(lines)):
        if len(lines[i]) > 0:
            line = lines[i].split(": ")[1]
            hexes = line.split()
            for hex_code in hexes:
                new_hex_buffer.append(hex_code)
    self.device_config_hex_buffer = new_hex_buffer
    self.hex_label.config(text=f"Hex Codes:
{self.device_config_hex_buffer}")

# Update the info_label with the new keyword
self.update_info_label(keyword)
self.info_canvas.delete("all")
'''start_pos, size_in_bytes = self.keywords_config_map[keyword]
hex_value = self.device_config_hex_buffer[start_pos]
if size_in_bytes > 1:
    for i in range(start_pos + 1, start_pos + size_in_bytes):
        hex_value = f"{self.device_config_hex_buffer[i]}{hex_value}"
'''
hex_value = self.get_hex_value(keyword)

print(f"...{self.device_identifier} header {keyword} hex code:
{hex_value}")
def insert_text_from_file_into_listbox(file_name):
    try:
        with open(file_name, 'r') as file:
            text = file.read()
            print(text)
            formatted_text = text.lower() # Example: Convert text
to uppercase
            self.text_box2.config(state='normal') # Allow
modifications

            self.text_box2.delete('0.0', 'end')
            self.text_box2.insert('end', formatted_text)
            self.text_box2.config(state='disabled') # Disable
modifications

            self.text_box2.config(wrap='word') # Enable word
wrapping
    except Exception as e:
        print(f"Error reading file: {e}")

def print_file_contents(file_name):

```

```

        try:
            with open(file_name, 'r') as file:
                for line in file:
                    print(line, end='') # Print each line without
adding extra newline
        except Exception as e:
            print(f"Error reading file: {e}")

print_file_contents('02h.txt')
match keyword:
    case "Device ID":
        print(f"information about {keyword} wanted")
        '''command = f"echo '{self.sudo_password}'|sudo -S lspci -
vnm -s {self.device_identifier}"
        results = self.run_command(command)
        results = results.splitlines() #re.findall("Device:\.+",
results)[1]

        for idx in range(1, len(results)):
            if (results[idx].find("Device:") != -1):
                results = results[idx][len("Device:") + 2:]
                break
        #detailed_device_info = f"Configuration item: {keyword}\n"
        detailed_device_info = f"{detailed_device_info}{results}"
        ...
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('04h2.txt')
    case "Vendor ID":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('02h.txt')
    case "Status":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('1.txt')
    case "Command":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('2.txt')
    case "Class code":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('3.txt')
    case "Revision ID":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('4.txt')
    case "BIST":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('5.txt')
    case "Header Type":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)

```

```

        insert_text_from_file_into_listbox('6.txt')
    case "M Latency T":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('7.txt')
    case "Cache Line":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('8.txt')
    case "Base Address":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('9.txt')
    case "reserved0":
        print(f" ...processing header {keyword}")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('10.txt')
    case "Subsystem ID":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('11.txt')
    case "SubsystemVendor ID":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('12.txt')
    case "Expansion ROM base Address":
        print(f"information about {keyword} wanted")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        command = f"echo '{self.sudo_password}'|sudo -S lspci -x -s
{self.device_identifier}"
        addressBars = self.run_command(command)
        print(addressBars)
        addressBars_lines = addressBars.splitlines();
        #detailed_device_info = f"Configuration item: {keyword}\n"
        for line_idx in range(1, len(addressBars_lines)):
            print(addressBars_lines[line_idx])
            if len(addressBars_lines[line_idx]) > 0:
                detailed_device_info = f"{detailed_device_info}
BAR{line_idx - 0}: {addressBars_lines[line_idx]}\n"
            insert_text_from_file_into_listbox('13.txt')
    case "Cap pointers":
        print(f" ...processing header {keyword}")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('14.txt')
    case "reserved1":
        print(f" ...processing header {keyword}")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('15.txt')
    case "reserved2":
        print(f" ...processing header {keyword}")
        self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        insert_text_from_file_into_listbox('15.txt')

```

```

        case "int Line":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
            insert_text_from_file_into_listbox('17.txt')
        case "int pine":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
            insert_text_from_file_into_listbox('18.txt')
        case "min_Gnt":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
            insert_text_from_file_into_listbox('19.txt')
        case "Max_Lat":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
            insert_text_from_file_into_listbox('20.txt')
        case "link Status 2":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
            insert_text_from_file_into_listbox('02h.txt')
        case "Link control 2":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        case "Slot Capabilities 2":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        case "Slot Status 2":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)
        case "Slot Control 2":
            print(f"information about {keyword} wanted")
            self.draw_info_cells(keyword, hex_value,
self.header_descriptions[keyword],self.bit_descriptions)

        #command = f"echo '{self.sudo_password}'|sudo -S lspci -vvv -s
{self.device_identfier}"
        #detailed_device_info = self.run_command(command)

        print("=====device info details
begining=====")
        #print(detailed_device_info)
        print("=====end of devie info
details=====")
        #self.info_canvas.create_text(50, 50, text=detailed_device_info,
anchor="nw", tags="info_text")

        print(f"...in clicked (device_identfier={self.device_identfier},
keyword={keyword}, x={event.x}, y={event.y})")
        """
        closest_item = self.canvas.find_closest(event.x, event.y)

        """

```



```

def update_cell_data(self, new_data):
    if self.device_identifier == "host_bridge":
        self.keyword_values = {"Device Control": "3E", "Device ID":
"1A",}
    elif self.device_identifier == "VGA":
        self.keyword_values = {"Device Control": "4A", "Device ID":
"2B",}

    # Update the text items in each cell to reflect the new numbers
    for keyword, (rect_id, text_id, value_id) in self.areas.items():
        self.canvas.itemconfig(value_id,
text=self.keyword_values.get(keyword, ""))

def draw_cells(self):
    self.frame.update_idletasks()
    vertical_start = 37
    gap_between_labels = 40
    right_offset = 350

    # Column labels
    column_labels = ["00h", "04h", "08h", "0Ch", "10h", "14h",
"18h", "1Ch", "20h", "24h", "28h", "2Ch", "30h", "34h", "38h", "3ch,"]

    x = right_offset
    for i, label in enumerate(column_labels):
        y = vertical_start + i * gap_between_labels
        self.canvas.create_text(
            x, y,
            text=label,
            anchor='w',
            font=('Arial', 10)
        )

    x = 0
    y = 0
    self.keyword_values = {}
    self.areas = {}
    for widget in self.frame.winfo_children():
        widget.destroy()

        y += cell_height # Increment y to a new line
        x = 19 # Reset x position

def bind_to_event(self, rect_id, text_id, keyword):
    # Binding for the rectangle (cell)

        self.canvas.tag_bind(rect_id, "<Button-1>", lambda event,
key=keyword: self.clicked(key, event))
        # Binding for the text (label)
        self.canvas.tag_bind(text_id, "<Button-1>", lambda event,
key=keyword: self.clicked(key, event))

    keyword_idx = 0
    y = 19
    for row in self.cell_sizes:

```

```

        x = 19
        for cell_width, cell_height in row:
            if keyword_idx < len(self.keywords):
                rect_id = self.canvas.create_rectangle(x, y, x +
cell_width, y + cell_height, outline='black')
                # text_id = self.canvas.create_text(x + cell_width/2, y +
cell_height/2, text=self.keywords[keyword_idx])
                text_id = self.canvas.create_text(x + cell_width/2, y +
cell_height/2 - 8, text=self.keywords[keyword_idx])
                #value_id = self.canvas.create_text(x + cell_width/2, y
+ cell_height/2, text="")
                #self.get_prefix()
                hex_code =
self.get_hex_value(self.keywords[keyword_idx])
                value_id = self.canvas.create_text(x + cell_width/2, y +
cell_height/2 + 9, text=hex_code, fill='blue')

                bind_to_event(self, rect_id, text_id,
self.keywords[keyword_idx])

                #self.areas[self.keywords[keyword_idx]] = (rect_id,
text_id)
                self.areas[self.keywords[keyword_idx]] = (rect_id,
text_id, value_id)

            keyword_idx += 1
            x += cell_width
            y += cell_height
    def get_prefix(self, hex_code):
        if len(hex_code) > 0:
            bit_size = 4*len(hex_code)
            return f"{bit_size-1}:0 - "

        return ""

    def draw_info_cells(self, header_string, hex_code, header_description,
bit_descriptions):
        print(f"... in draw_info_cells: header_string={header_string},
hex_code={hex_code}, header_description={header_description}")
        original_header_string = header_string
        x = 19
        y = 19
        cell_width = 550
        # we define header_rec here
        cell_height = 30
        header_rec = self.info_canvas.create_rectangle(x, y, x + cell_width,
y + cell_height, outline='black')
        # need to draw the header_string inside the above rectable
        text_x = x + cell_width / 2
        text_y = y + cell_height / 2
        header_string = f"{header_string} ({hex_code})"
        self.header_text_id = self.info_canvas.create_text(text_x, text_y,
text=header_string, anchor='center', font=("Arial", 14) )
        '''inner_x = x + 6 # Adjust the x coordinate for the inner
rectangle's top-left corner
        inner_y = y + 40 # Adjust the y coordinate for the inner
rectangle's top-left corner
        inner_width = 88
        inner_height = 20

```

```

        inner_rec = self.info_canvas.create_rectangle(inner_x, inner_y,
inner_x + inner_width, inner_y + inner_height, outline='red')
'''
        # we define the bitmap_rec here
        cell_height = 70
        bitmap_x = x
        bitmap_y = y + 30
        bitmap_rec = self.info_canvas.create_rectangle(bitmap_x, bitmap_y, x
+ cell_width, y + cell_height + 30, outline='black')
        up_offset = 10 # Move 10 pixels up
        left_offset = 10 # Move 10 pixels left

        # Width and between the two texts and all
        left_offset = 10
        width_between_texts = 88

        first_text_x = x + cell_width / 4 - left_offset
        first_text_y = y + 30 + cell_height / 2 - up_offset
        self.info_canvas.create_text(first_text_x, first_text_y, text="31",
font=("Arial", 9))

        # Draw second text ("0") beside the first text at the adjusted
position
        second_text_x = x + cell_width / 4 + width_between_texts -
left_offset
        second_text_y = y + 30 + cell_height / 2 - up_offset
        self.info_canvas.create_text(second_text_x, second_text_y,
text="23", font=("Arial", 9))
        third_text_x = second_text_x + width_between_texts
        third_text_y = second_text_y
        self.info_canvas.create_text(third_text_x, third_text_y, text="15",
font=("Arial", 9))

        #Draw fourth text ("31") beside the third text
        fourth_text_x = third_text_x + width_between_texts
        fourth_text_y = third_text_y
        self.info_canvas.create_text(fourth_text_x, fourth_text_y, text="7",
font=("Arial", 9))
        fifth_text_x = fourth_text_x + width_between_texts
        fifth_text_y = fourth_text_y
        self.info_canvas.create_text(fifth_text_x, fifth_text_y, text="0",
font=("Arial", 9))
        # we define the description_rec here
        cell_height = 2900
        description_rec = self.info_canvas.create_rectangle(x, y + 30 + 70,
x + cell_width, y + 30 + 70 + 1300, outline='black')
        description_rec_center_x = x + cell_width / 2
        description_rec_center_y = (y + 30 + 70) + cell_height / 2

        label = f"{self.get_prefix(hex_code)} {original_header_string}"

        up_offset = 1440# Adjust this to move the text up by the number of
pixels you desire
        left_offset = 174

        # description_text_id = self.text_box2.create_text(
        description_rec_center_x - left_offset,
        description_rec_center_y - up_offset,
        text=label,
        anchor='center',
        font=("Arial", 10, "bold"))

```

```

# )
customFont = tkFont.Font(family="Arial", size=10)
#customFont = tkFont.Font("Arial", 10)
font_width = customFont.measure("0")
font_height = customFont.metrics()["linespace"]
print(f"0 has width={font_width}, height={font_height}")
label = self.keywords_description_map[original_header_string]
bm_rec_width = font_width + (2*self.padding) # add 3 pixels to each
side of bit rect
bm_rec_width = bm_rec_width*(4*len(hex_code)) #total width of the
whole bitmap rect
bm_rec_height = font_height + (2*self.padding)

up_offset = 220 # Adjust this to move the text up by the number of
pixels you desire
left_offset = 20 # Adjust this to move the text left by the number
of pixels you desire

#creating bitmap flags rect
right_offset = -70 # Adjust this to move the rectangle to the right
up_offset = 1
coords = self.info_canvas.coords(bitmap_rec);
rightmost_x = coords[2]

bitmap_x = rightmost_x - bm_rec_width + right_offset
bitmap_y = coords[1] + 30 - up_offset
bitmap_flap_rec = self.info_canvas.create_rectangle(
    bitmap_x,
    bitmap_y,
    bitmap_x + bm_rec_width,
    bitmap_y + bm_rec_height,
    outline='red')
#convert hex_value to an array of bits
# then iterate left to right
# adjust the x, y, width
# draw the bit in its rect
#hex_code = hex_code.zfill(8)

bin_str = ''.join(format(int(digit, 16), '04b') for digit in
hex_code)
bit_array = [int(bit) for bit in bin_str]
bit_length = len(bit_array)

# Set bit_gap based on bit length
if bit_length == 8:
    bit_gap = -78 # set your desired gap for 16 bits
    offset_x = 3
elif bit_length == 16:
    bit_gap = -165 # set your desired gap for 24 bits
    offset_x = 4
elif bit_length == 24:
    bit_gap = -253 # set your desired gap for 32 bits
    offset_x = 2
elif bit_length == 32:
    bit_gap = -341 # set your desired gap for 24 bits
    offset_x = 3

else:
    bit_gap = 15 # default value

# bit_gap = -80

```

```

#offset_x = -100
#rightmost_x = bitmap_x + bm_rec_width - offset_x

offset_y = 10
for i, bit in enumerate(bit_array):
    #for i in range(0, len(bit_array)):
        x = bitmap_x + i * (bm_rec_width + bit_gap) + offset_x
        y = bitmap_y + offset_y
        print(f"bit = {bit_array[i]}")

    self.info_canvas.create_text(
        x,
        y,
        text=bit_array[i],
        anchor='w',
        font=('Arial', 9)
    )
def create_description_text(self, label, selected_item):
    # Create text inside description rectangle
    #if selected_item == 'Device ID':
        #left_offset = 10
        # up_offset = 140
        description_text_id = self.text_box2.create_text(
            description_rec_center_x - left_offset,
            description_rec_center_y - up_offset,
            text=label,
            anchor='n',
            font=("Arial", 9, "bold"))
        self.text_box2.config(scrollregion=self.text_box2.bbox("all"))
        self.text_box2.configure(state=tk.NORMAL)
        self.text_box2.delete("1.0", tk.END)
        self.text_box2.insert(tk.END, formatted_info)
        #self.highlight_alternate_lines()
        self.text_box2.configure(state=tk.DISABLED)

def mainloop(self):
    self.root.mainloop()

if __name__ == "__main__":
    viewer = DeviceViewer()
    if viewer.sudo_password is not None:
        if viewer.verify_sudo_password(): # Verify the provided password
            viewer.root.geometry("1235x510")
            viewer.root.mainloop() # Start the Tkinter main loop if the
password is valid
        else:
            messagebox.showerror("Error", "Incorrect password or password
input canceled. Exiting...")

```

19. Appendix

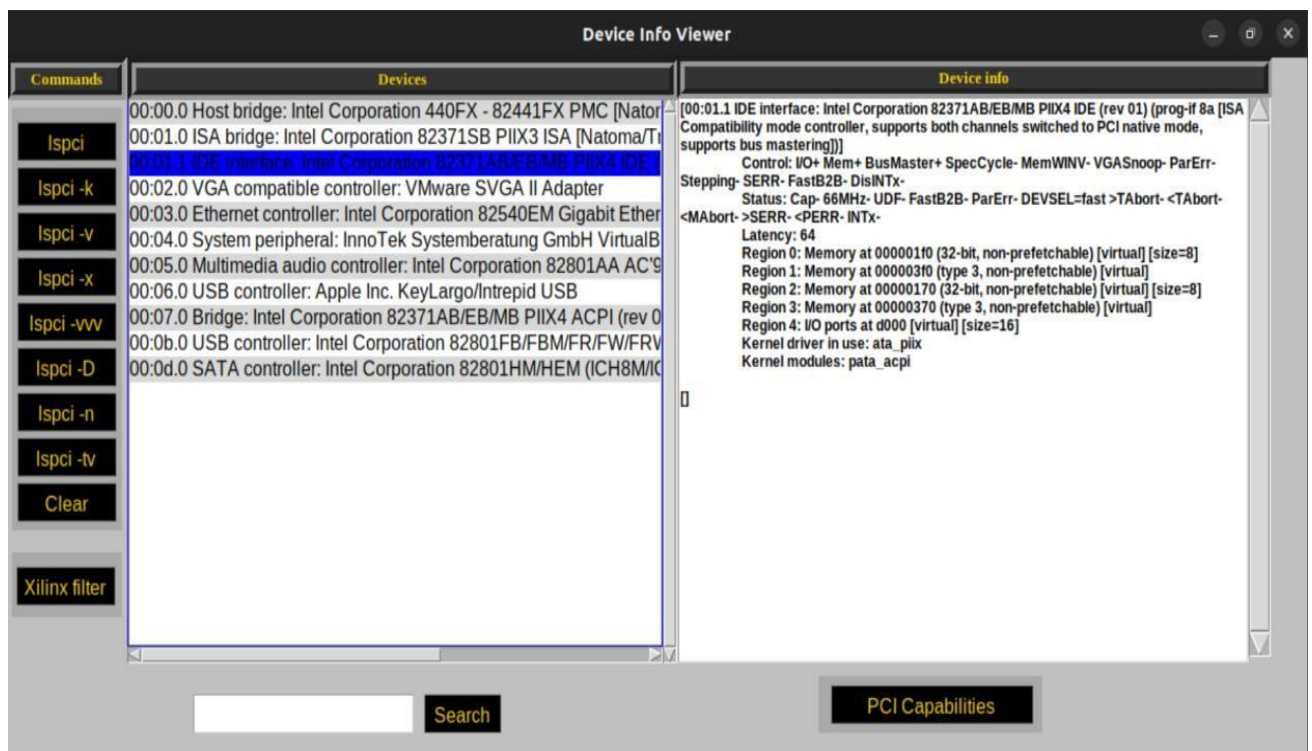


Figure 40 GUI#1 with final design

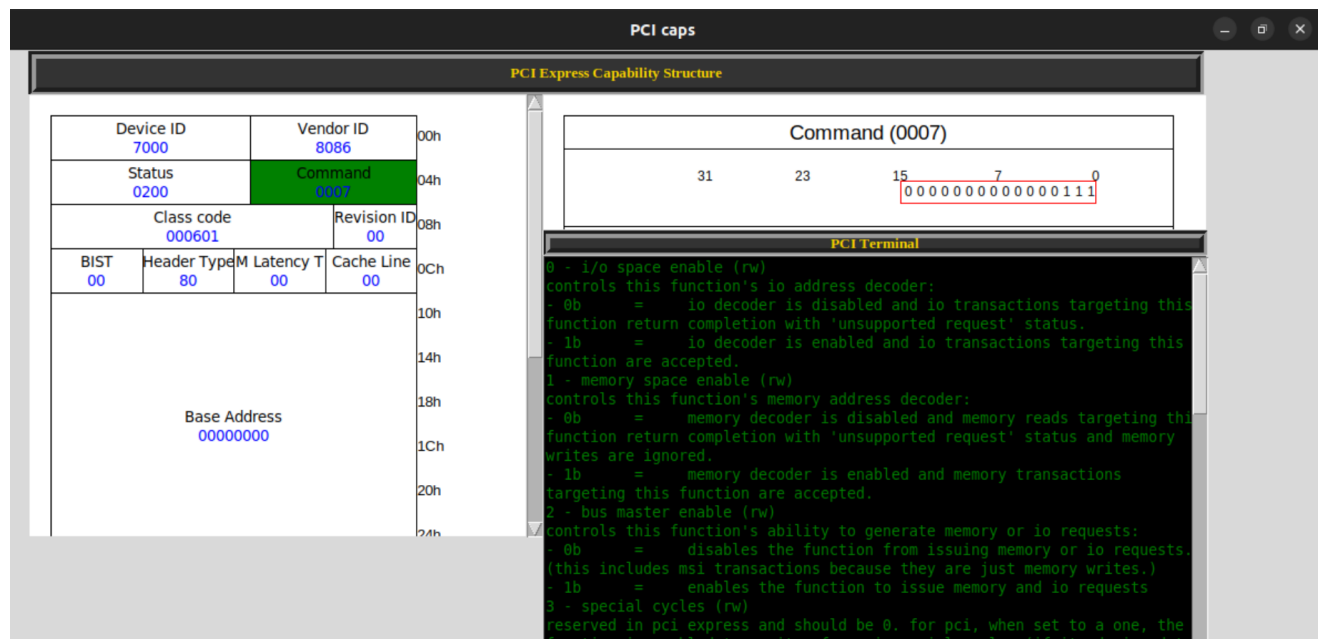


Figure 41 GUI#2 with final design

20.Reference

- [1] VirtualBox, "Oracle VM VirtualBox," Oracle Corporation. [Online]. Available: <https://www.virtualbox.org/>. [Accessed: Apr. 09, 2025].
- [2] Ubuntu, "Download Ubuntu Desktop," Canonical Ltd. [Online]. Available: <https://ubuntu.com/download>. [Accessed: Apr. 09, 2025].
- [3] YouTube, "Install Ubuntu Linux on VirtualBox in Windows 10," YouTube video. [Online]. Available: <https://www.youtube.com/watch?v=JUGEkFDeuwg>. [Accessed: Apr. 09, 2025].
- [4] Xilinx Support, "Xilinx Support Portal," Xilinx, Inc. [Online]. Available: https://support.xilinx.com/s/?language=en_US. [Accessed: Apr. 09, 2025].
- [5] Stack Overflow, "Where Developers Learn, Share, & Build Careers," [Online]. Available: <https://stackoverflow.com/>. [Accessed: Apr. 09, 2025].
- [6] Python Principles, "Python Tutorials and Exercises," [Online]. Available: <https://pythonprinciples.com/>. [Accessed: Apr. 09, 2025].
- [7] YouTube, "How to Parse JSON in Python," YouTube video. [Online]. Available: <https://www.youtube.com/watch?v=b093aqAZiPU>. [Accessed: Apr. 09, 2025].
- [8] YouTube, "How to Format JSON in Python," YouTube video. [Online]. Available: https://www.youtube.com/watch?v=VlfLqG_gjx0. [Accessed: Apr. 09, 2025].
- [9] AxiomQ, "Custom Web App Development & Digital Solutions," [Online]. Available: <https://axiomq.com/>. [Accessed: Apr. 09, 2025].
- [10] Cloudinary, "Image and Video Management Platform," [Online]. Available: <https://res.cloudinary.com/>. [Accessed: Apr. 09, 2025].
- [11] Tom's Hardware, "Tech News and Reviews," [Online]. Available: <https://www.tomshardware.com/>. [Accessed: Apr. 09, 2025].
- [12] MindShare, "Arbor Software Solutions," [Online]. Available: <https://www.mindshare.com/Software/Arbor>. [Accessed: Apr. 09, 2025].