Creating a Graphical User Interface for Software and Logic Development

Simon Southwell

9th September 2025

Preface

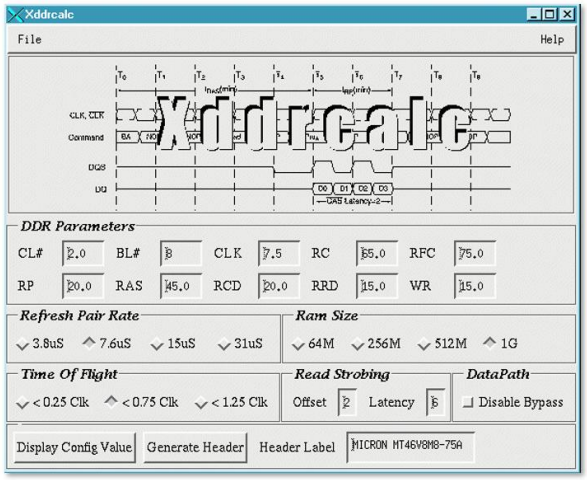
This document is a PDF version of an article written in September 2025 and uploaded to LinkedIn, on creating GUIs with Python and *TKInter* for software and logic development.

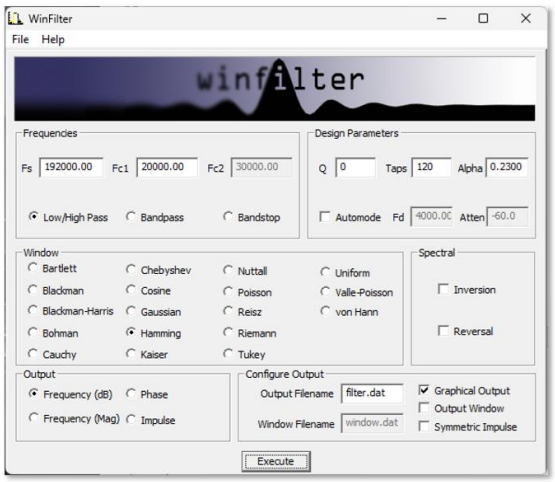
Simon Southwell
Cambridge, UK
September 2025

Contents

Introduction

In many of the projects I have done, whether application software, models of hardware in C++ and scripts for running synthesis and simulations, there is often a degree of configurability—selecting a test, setting a parameter or generic, or specifying a random number seed etc. Often these a presented as command line option to a program or script (via `getopt()`, say), or setting parameters of a make file on the command line. This can often involve a fair bit of typing with mistakes in input quite likely. If this could be captured within a simple graphical user interface (a GUI) then this makes the system much more usable and less prone to mistakes.





I have written a few GUIs in my career; initially in C or C++ using X11 (for Linux) or Microsoft Foundation Class (MFC, for Windows). These have included a DDR memory interface configuration GUI to select register settings of one of my DDR interface designs, for different memory specifications using X11, and for my *winfilter* finite impulse response filter design project using MFC.

There are challenges to using C++ libraries for writing GUIs, though it taught me a lot about how this kind of graphical system works. Fortunately, for the kind of GUIs I am talking about—front ends for configuring software or logic flows—we can create GUIs using scripting languages and provided GUI libraries, and this simplifies the process somewhat. The scripting language we are going to use is Python as it's popular and easy to use and has vast library support. Graphical functionality is not part of the native language and there are a number of libraries and packages available for GUI applications. The list below names just a few.

- PYQT
- PySide
- Kivy

- WxPython
- TKinter

This is not a comprehensive list by any means and each has their merits. A useful round up of these and some others is given here for those interested. For this document we are going to choose *Tkinter*.

Why TKinter?

We have been talking about configurability as a motivation for wanting to put together GUI front end applications but I have had input from a few very talented engineers that, wherever possible, things should work out-of-the-box for a user and configuration should be a feature for users that want more advanced features, and I have always tried to follow this principle. One of the things I want to avoid with my tools and applications is the user having to do a whole set of pre-requisite installs and setups before they can use the code. The *TKinter* library for Python is the de-facto graphics library for python and comes bundled with Python itself (certainly for Windows and MacOS), with very good portability between systems, allowing a user to fire up a script straight away (Okay, they need Python installed, but you get my meaning). *TKinter* is not the easiest of the packages to use, but it is not too complicated, as I hope to show you, and if you can master this library, you are set up for migrating to others in the future.

TKinter (Tk Interface), then, is a widget toolbox that was originally developed for the TCL scripting language but has been wrapped up in a Python package so that it can be used from this language in place of TCL. In this document I want to show you some fundamentals so that you can make some GUIs for yourselves with a lean towards software and logic design flow configurability and execution. There will be a couple of real-world application case studies that are open-source and so the source code is available for inspection.

Some Assumptions

For the rest of this document, so we can focus on the Tk and GUI aspects of Python, I have to assume you are familiar with the basic language and with its support of classes. If you are new to Python then this is not the place to start, but there are plenty of good resources out there. I find that I refer my mentees quite often to the w3school python resources.

TKinter Basics

Before we can do anything the relevant packages need to be imported. Fundamentally only the `tkinter` package is required, but a useful 'themed widget' package is also available which overrides the basic package methods and matches the theme of the windows manager it running on and also provides some extra functionality. So, in a Python script the packages are imported and a Tk object must be created:

```
from tkinter     import *
from tkinter.ttk import * # optional

root = Tk()
```

In order to display something in a window on the screen some sort of widget needs to be created and then the graphics window 'run' in a loop (more later). We can display a static text message using a 'label ' widget.

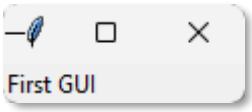
```
label0 = Label(root, text = "First GUI")
label0.grid()

root.mainloop()
```

Here a Label method is called to create the widget, and the return 'handle' is saved in the `label0` variable. It requires an argument specifying its parent object. In this case it is the top level Tk object that we saved in root but, as we will see, hierarchies of objects can be formed, and this argument specifies which object of which the widget is a direct child. Then, using a named argument, the text for the label is specified. The following line is required to specify the layout strategy—and, oddly, there's more than one. In fact, there are three built in layout managers:

- `pack()`: This uses horizontal and vertical boxes that can be placed in left, right, top and bottom positions and each box has a relative offset from the others. It is simple to use, but limited
- `place()`: Widgets are placed on a two dimensional plane using absolute x and y co-ordinates. Precise, but adding new widgets can be problematic.
- `grid()`: Widgets are placed on a two dimensional grid of rows and columns. A good compromise between control and complexity of use.

Note that these cannot be mixed, so pick one and stick to it. In this document, the grid methods will be used.

Finally, the top level's graphical 'main loop' is run. This will not return until the window is closed. It is at the point of running the main loop that the window is displayed.



This may not look too impressive, but we now have a framework with which more complex GUIs can be built. Before moving on, though, the code can be tidied into a more formal and useful class based structure and since the example we will be looking at also have this architecture it is useful to make this first simple example the same.

```
from tkinter     import *
from tkinter.ttk import *

class firstGui :

  def __init__(self) :
    self.root = Tk()

  def run(self) :

    label0 = Label(self.root, text="First GUI")
    label0.grid()

    self.root.mainloop()

gui = firstGui()
gui.run()
```

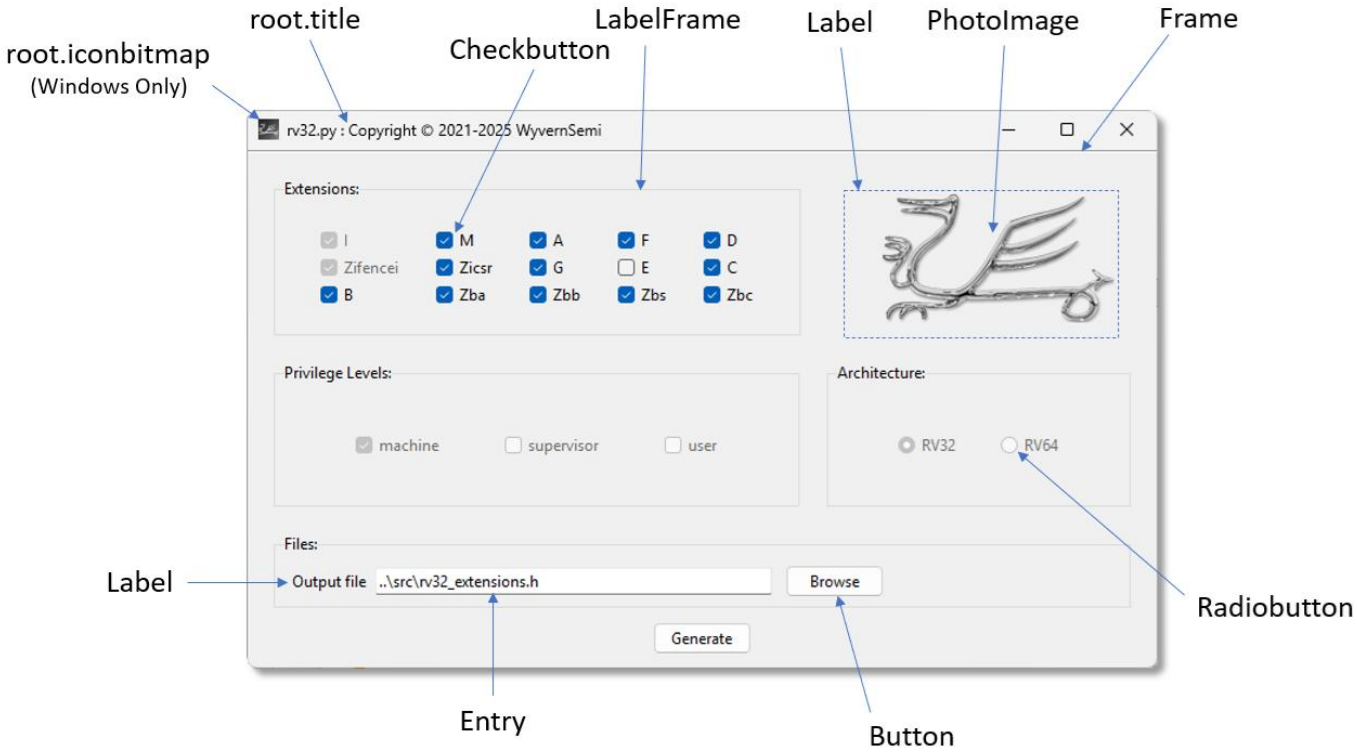
More on the main loop

In many ways it is not necessary to know about how the internals of the graphical package works but it is helpful to have a model in one's head when constructing more complex applications to understand how to go about things to avoid problems or understand why something is not working as expected.

Most, if not all, graphical systems are event based. This is just as true for the MFC or X11 C++ libraries as it is for *TKinter*. By event based, I mean it waits for some external input (an 'event') such as pushing a button or typing in a box, and then takes some action, such as updating what's displayed or passing information to the user's python program. When the main loop is run, it will wait for an 'event', process it, and then loop to wait for the next event. As we shall see later, it can call some user code receiving an event. If that code then blocks on something else, the main loop will also be blocked and will stop responding to new events. The manifests as the window 'freezing'—the familiar "*xxx* is not responding" message. So, it is important to keep this in mind when designing the code for a GUI application to avoid such lock-up scenarios.

Types of Widget

The example we have just looked at uses a 'Label' widget to display some text, which might be good for an alert or some such, but we are more likely to want to be able to input some information and get out some desired results. The diagram below shows one of my simpler GUIs, used to configure the *rv32* RISC-V Instruction Set Simulator. This GUI script uses some of the most common widgets and these will be discussed in this section. For now, we will skip how to organise these in the grid and how to connect our Python code to these widgets and concentrate on what these widgets do and how to create them in our GUI window.



Each of the widget types used are labelled on the diagram (just the once) but, before we even look at those, we can set some attributes on the 'root' `Tk` object. A title can be set for the whole window using the `title` method of the root `Tk` object:

```
root.title('rv32.py : Copyright \u24B8 2021-2025 WyvernSemi')
```

The strange `\u24B8` set of characters is just the unicode for a copyright symbol. Not only will this text appear in the window, but the taskbar object for the application will also be labelled the same way. In addition to a title, for windows, the icon can be changed from the default:

```
root.iconbitmap(os.path.join(os.getcwd(), 'favicon.ico'))
```

Here an `os` call has been made to get an absolute path to the current directory and the icon file name has been joined to this. This may seem a convoluted way to do this, but it makes this code portable, especially between Windows and Linux.

Frame and Label Frame

The first widget to look at is a frame as this is how we can add hierarchy to organise the widgets. It doesn't provide any graphical output itself by default, but is a vessel for containing other widgets, including other frames. In this particular GUI, a single frame is constructed as the child of the root object that all the other widgets will descend from (though may be further down the hierarchy). The code snippet below shows how to construct a frame:

```
frm = Frame(master = root)
frm.grid(row = 0, padx = 10, pady = 10)
```

The only argument that's required is the parent object but can have various other arguments to set things like background colour, height and width. Once the frame object is created then certain grid attributes can be set like row, column and padding (space around the widget). More details on the grid settings will be discussed later but suffice it to say here, a new frame forms a new grid with rows and columns numbered from 0 upwards, starting in the top left of the window. If a new frame is constructed as a child of a frame it will have a new grid associated with it, starting from 0 again for any widgets that are set as child objects of that frame.

A different type of frame is a `LabelFrame`. This type of frame very much like a `Frame` except some text can be set for it and a border will be set around the frame. From the diagram above, there are four `LabelFrame` objects for 'Extensions:', 'Privilege Levels:', 'Architecture:' and 'Files:'. Just as for Frames, these have their own internal grid.

```
lblfrm = LabelFrame(master = frm, text = 'Extensions:', padding = 20)
lblfrm.grid(row = 0, padx = 10, pady = 10)
```

Some things to note here are that the parent is now not the root object, but the top level frame created in the last section. Also, some text is provided to display and an optional padding argument is given. This padding, unlike the grid attributes set afterwards, is for internal padding from any widgets placed within the frame. By default, it will shrink to having the border very close to the widgets which may or may not be what you want. Experimentation with settings is needed to produce the required aesthetic.

In the example then, each of the `LabelFrame` objects are children of the top level frame, as is the object containing the 'wyvern' image and the 'Generate' Button (more on these last two shortly).

Label

We have seen the `Label` widget from the first GUI example, where some text can be set to display. In the *rv32* example, only the 'Output file' text is a `Label`. This was set in the same was as seen with the first GUI example.

In addition, though, the image of the wyvern symbol is also a Label. First the image must be loaded from a file. In this case the `PhotoImage` function is used.

```
img = PhotoImage(file = os.path.join(self.scriptdir.get(), 'icon.png'))
```

This function supports most common image file types, and this example uses `.png`. Once we have loaded the image we can call `Label` with slightly different arguments:

```
panel = Label(master = frm, image = img)
panel.grid(row = 0, column = 2, padx = 5, pady = 10)
```

Buttons

So far we have only looked at widgets that take no input from the outside world to allow a user to interact with the GUI. We'll start with the three different buttons widgets that the example GUI uses.

Button

This is just as the name implies, though push-button might be a better name as it is meant to be a push and release to instigate some action. In the *rv32* GUI example, the buttons labelled 'Browse' and 'Generate' are of this type. The snippet below shows an example to create a button:

```
goButt = Button(master = frm, text = 'Generate', command = self.__rv32Generate)
goButt.grid(row = 2, columnspan = 3)
```

One thing to note is that there is a `command` argument and the name of a method in a class. This defines a function (or class method) to be called when the button is pushed (i.e. clicked on) and this is how we can connect this GUI widget to our own code. This is a 'callback' function which if you're unfamiliar with this concept I have written an article on this subject for C and C++.

Also note that the call to the object's grid method has a `columnspan` argument with a value of three. Though it is not obvious, the top level frame has three columns (the

wyvern image was in column 2, counting from 0) but here we want the button to be central in a single column of the last row, and this is how this is done.

CheckButton

A Check button is a slightly different beast in that when clicked it change between two states—ticked or unticked.

```
hdl = Checkbutton(master = extFrm, text = 'Zicsr',
                  variable = self.Zcr, onvalue = 1)
hdl.grid(row = 1, column = 1, padx = 10)
```

The above example shows one of the check buttons for the check box that's labelled as *Zicsr* in the Extensions frame. As well as a text argument to give the label for the check box, a variable argument associates the check box with a certain type of variable which we will be able to query for the state of the check box within the Python code. These variables are covered in a later section. The `onvalue` argument gives the value to use as the 'on' or 'checked' state. This would normally be 1 but can be changed for another value if that would be convenient for processing in the code. An `offvalue` argument is also available, with the default off value being 0.

All the check buttons are independent of each other and changing the state of one doesn't affect the others. If that might be required then the user code must react to a check button being clicked to update state.

RadioButton

Setting up a radio button is almost identical to setting up a check button.

```
hdl = Radiobutton(master = archFrm, text = 'RV32',
                  variable = self.arch, value = 0)
hdl.grid(row = 0, column = 0, padx = 15)
```

One difference here is that instead of an `onvalue` argument we have a `value` argument. A radio button is invariably associated with a group of other radio buttons and only one in that group can be selected at anyone time—hence the name. (For younger readers, car radios in the 50's to the late 70's had a set of mechanical station tuning select buttons that when pushed released the currently selected one and latched the pushed one to select a new preset station frequency.)

To associate radio buttons together, all the members of the set will have the same variable selected in the variable argument and each needs to be set to a different value. In the *rv32* example there are two radio buttons in the 'Architecture' frame, 'RV32' with

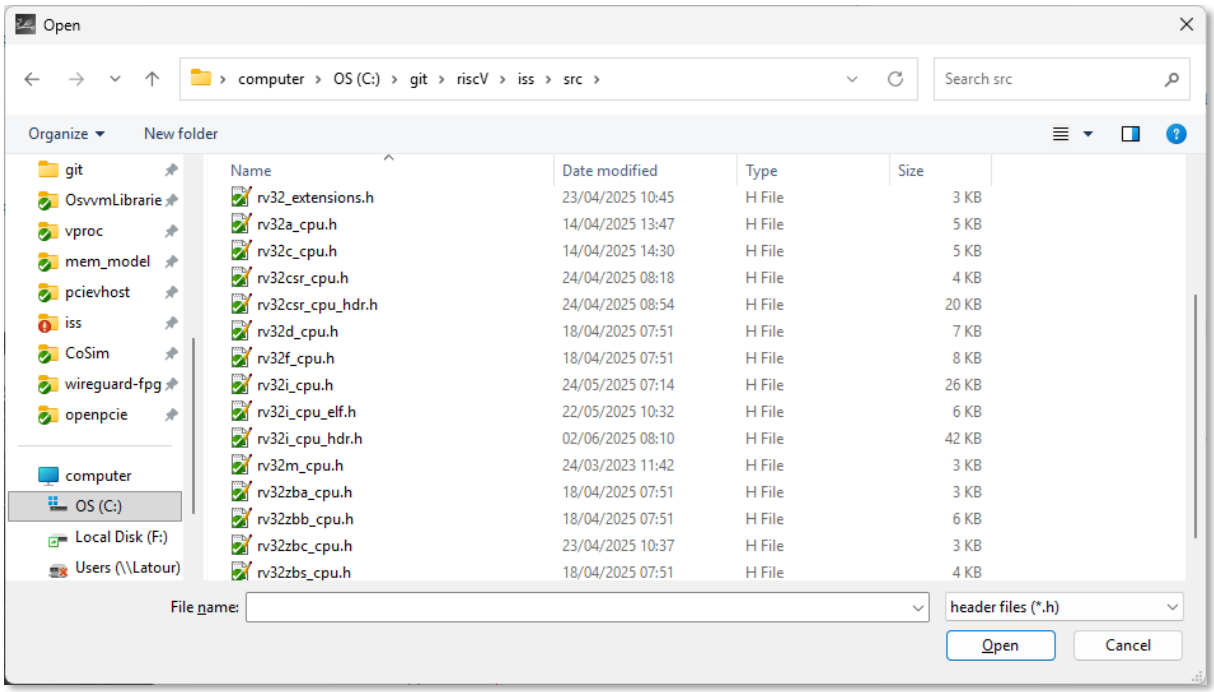
a value of 0 and 'RV64' a value of 1. When the variable is inspected for its current value, this will be one of these values to identify which button is currently active.

Entry

The last widget from the rv32 example is the Entry widget which allows textual user input.

```
hdl = Entry(master = filesFrm, textvariable = self.opfile, width = 50)
hdl.grid(row = 0, column = 1)
```

The equivalent of the `variable` argument for the buttons is the `textvariable` argument with the name of the special variable in which to place the input text. Jumping ahead momentarily, the value in an entry can be set from code (and we'll see how later) and in fact the 'Browse' button's associated callback runs a *TKinter* function `askopenfilename`, which pops up a whole new window.



This is the file navigator window for the OS and window manager that the script is running under, and the directory hierarchy can be navigated and a file selected. When the 'Open' button is clicked the function returns with a string with the selected filename and path. The callback can then use this to update the variable associated with the Entry object using a `set()` method available with the variable object, which will then also update what's displayed.

Disabling Widgets

Widgets can be disabled from code in which case they are displayed greyed out and don't respond to user input. In the *rv32* example the 'I' and 'Zifencei' check boxes are disabled since these extensions must always be present. In the 'Privilege Levels' and 'Architecture' frames, the buttons are disabled as only one of the modes in each are currently supported, so the others can't yet be selected, but the script is future-proofed. To disable a widget the widget's object has a `config()` method which can set certain states, one of which is `DISABLED`.

```
chkButtonHdl.config(state = DISABLED)
```

I've mentioned some permanently disabled widgets, but it might useful also to disable some widgets depending on the state of other settings. If a script has, say, a 'lite' mode and a 'full' mode, the 'lite' mode might not include some selections available in 'full' mode, and these might be set to the appropriate 'off' settings and then disabled.

What this amounts to is ensuring that only valid combinations of settings are permitted and this is visually feedback to a user. If a non-GUI script with command line options is mis-configured then it is likely that the script will print an error message and exit. With a GUI and the ability to disable widgets a 'gatekeeper' to valid settings can be put in place and make configuration easier and less error prone.

TKinter Class Variable

In the widgets that have been talked about so far, for the input methods (the three button types and the `Entry` widget) these have been associated with a variable (or text variable). The buttons need an integer, and the entry needs a string, but it is not possible to just use the basic Python types. Instead, these must be *TKinter* class variables. These have all sorts of hidden features that allow things to go on in the background whenever the value of an input widget is changed, such as update what's displayed. For the *rv32* GUI we need to use just two variable classes: `StringVar()` for text and `IntVar()` for integers, but there is also `DoubleVar()` for floating point and `BooleanVar()` for boolean.

To create a variable of this type it is just a matter of assigning a variable to the appropriate class:

```
self.extZcr             = IntVar()
self.opfile             = StringVar()
```

As we've seen before, these class objects can be set with a `set()` method but also the current value returned with a `get()` method.

```
self.opfile.set('..\\src\\rv32_extensions.h')
self.extZcr.set(1)
currOpfile              = self.opfile.get()
currZbc                 = self.extZbc.get()
```

When these variables are set from the code, the associated widget display will also be updated to reflect the new value.

I have spoken about setting and getting input widget values, but this can also be done to `Label` text. If it is useful to update the text of a label, if used to display some status for example, then a `StringVar()` can be associated with it at its creation using a `textvariable` argument just as for an `Entry`. Setting this string variable object will then update what's displayed for the label's text.

Setting up traces

With the Button widget, as we saw, a 'callback' function can be associated with it, called whenever the button is pressed. With the check- and radio buttons, we associated a *TKinter* class variable and so no callback is specified—actually internal callbacks are made behind the scenes. However, it is useful sometimes to call user code whenever a check- or radio button is changed. To allow other updates or actions to happen:

```
self.extG.trace       ('w', self.__rv32ExtGUpdated)
```

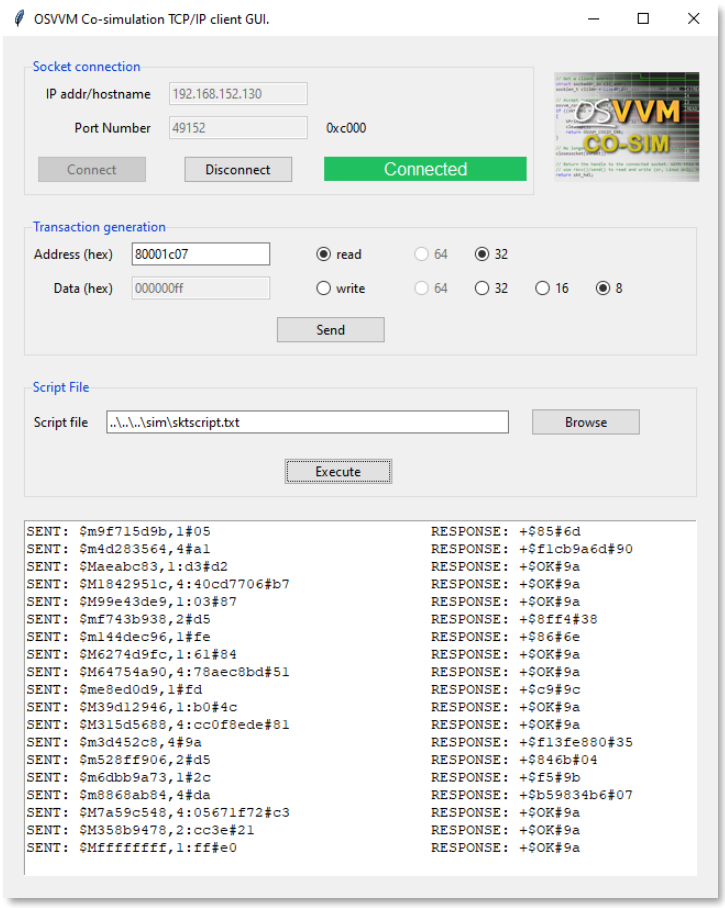
Here, using the variable's `trace()` method, the `extG` class variable has the method `__rv32ExtGUpdated` set as the callback whenever the associated widget is 'written'—i.e. updated, either from user input of via the code. There is also a 'r' setting for a callback on reading (from the code) and some others which we'll gloss over.

As an example of the use of this, with the rv32 GUI's 'Extension' check boxes, these select which standard extensions the RISC-V model should have includes in the build. Now, the 'G' extension is shorthand for including the 'I', 'M', 'A', 'F', 'D' and 'Zicsr' extensions. So if, for example' the 'A' check box is unchecked, then the set is no longer compliant for 'G' and this automatically gets unchecked. However, if various 'G' sub-extensions are unchecked and the 'G' check box is ticked, then it automatically turns on the 'G' sub-extensions. This is achieved by associating trace callback on the appropriate check box widgets, and this code validating and updating the settings as appropriate. An invalid setting is to have 'D' (double precision floating point extensions) selected without 'F' (single precision floating point), though the other way around is valid. The scripts, again, will force a correct setting using callback when these widgets are updated.

Strategies for Managing Grids

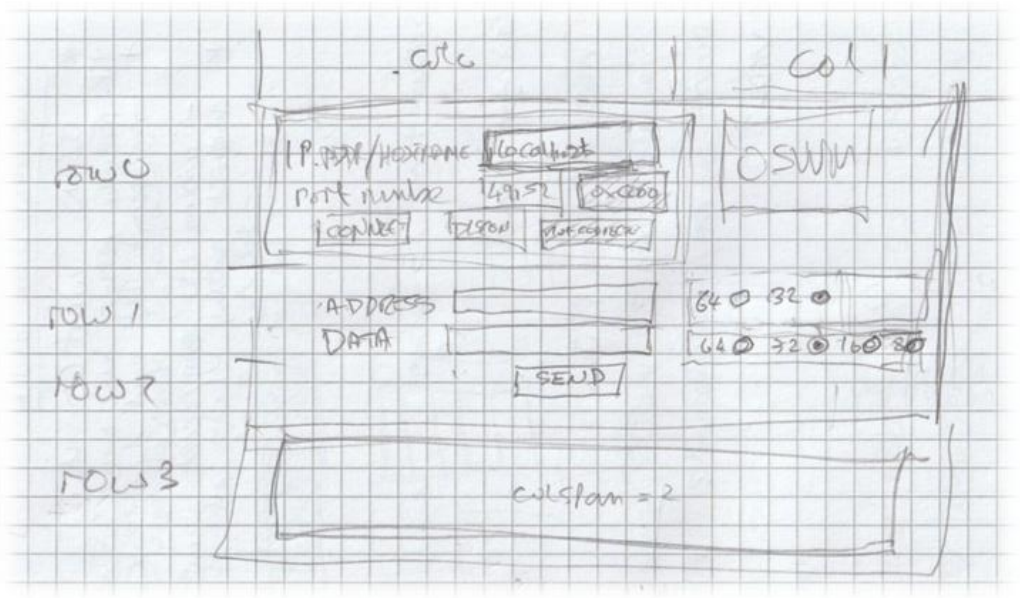
The examples given so far have all used the grid layout strategy but are fairly simple and the grid layout design has been glossed over somewhat. In this section I want to look at a slightly more complex GUI example with a more elaborate layout, though it uses only the same widget types that have already been shown except for a text output box.

The GUI was put together as part of demonstration and test code for the co-simulation features of the OSVVM VHDL Verification methodology and library. With OSVVM co-simulation a program can be run that is a TCP socket server for connection to an external program with a client that can connect. This could be anything, such as a QEMU process for example, with the appropriate client socket code. For the demonstration the GUI script was constructed to stand in for such a process, to be a client that can send commands to do writes and reads to memory in the logic simulation using a simple protocol. The important thing here is the layout of the GUI, which is shown below:

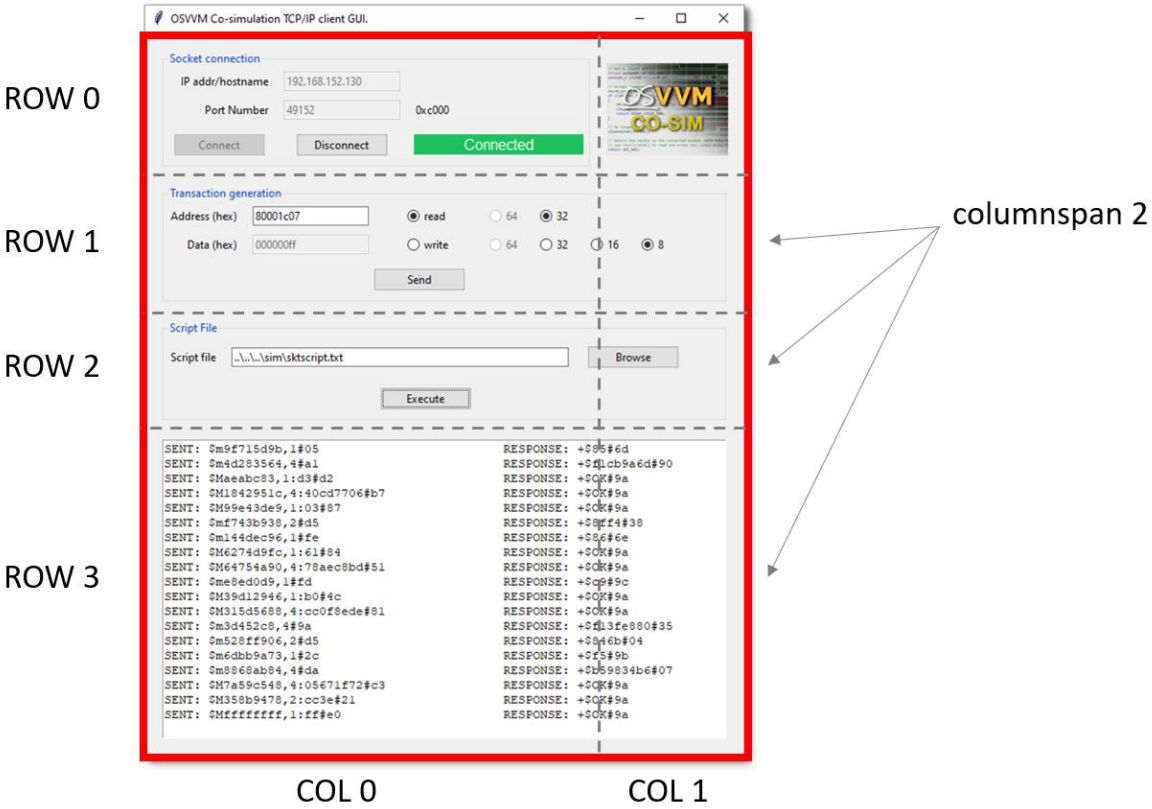


When designing a GUI like the above, I always try and sketch out a rough layout on paper first to answer certain fundamental questions, such as how many rows and

columns are need? What should be gathered into a sub-frame (repeating the rows and columns process for this)? What needs to span columns? The diagram below shows my original rough sketch for the above GUI.

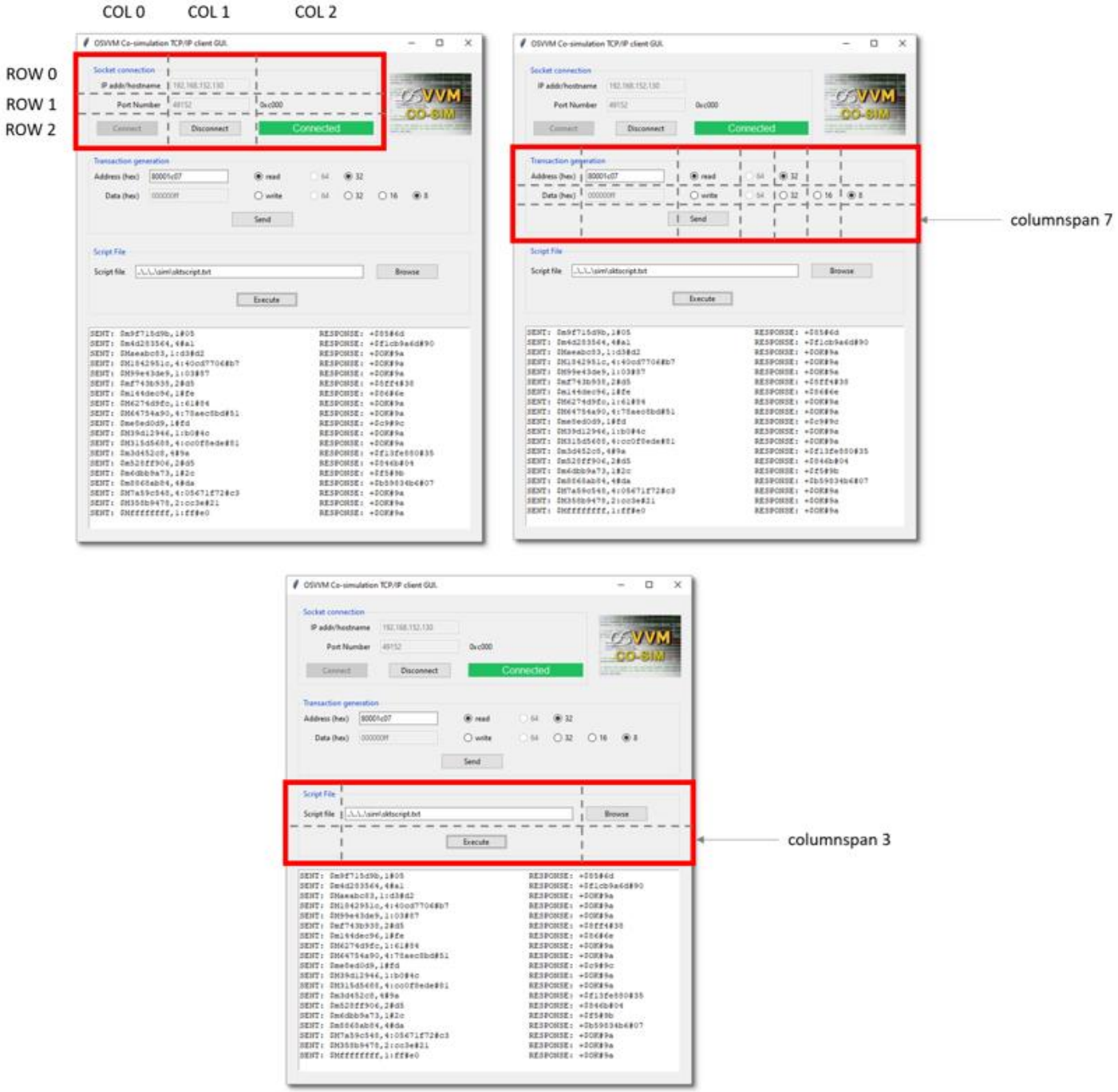


This is a bit messy to look out, but I worked out the basic layout of the top frame as four rows of two columns, identified some sub-frames and their contents, Identified the maximum number of columns required and worked out some column spanning requirements. Let's have a more formal look at the layout that was finally derived.



Here we see the four rows and two columns as in the sketch. Row 0 has a label frame in column 0 and an image label in column 1. All the other rows are single widgets (two frames and a text box) and so need to span the two columns of the top frame.

For reference, the rows, columns and column spans for the sub-frames are shown below:



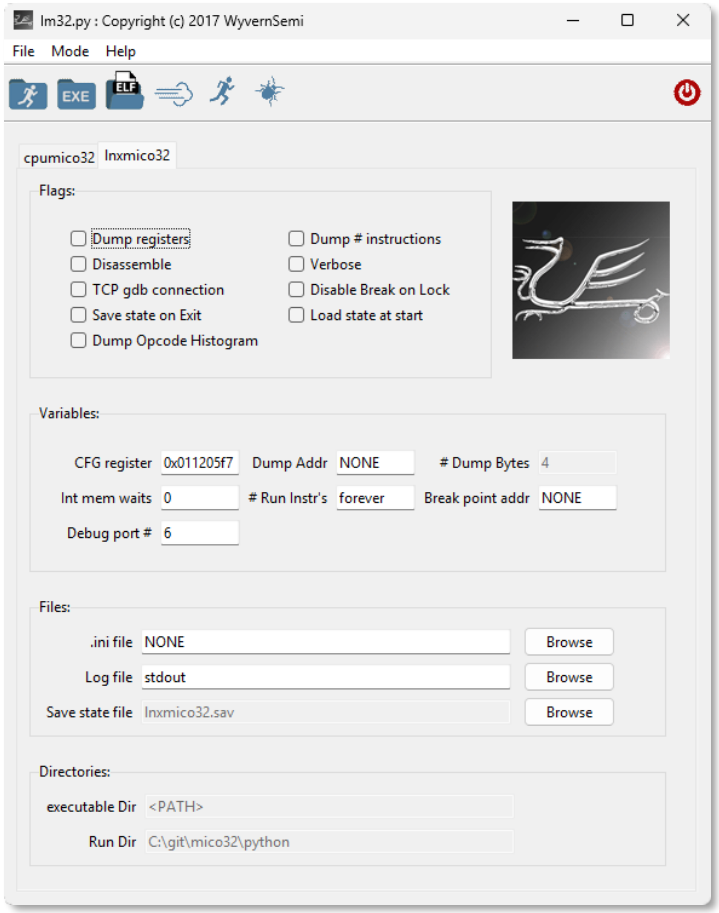
The first sub-frame is a 3×3 grid with no column spanning, whilst the second is a 3×7 grid with the last row spanning the whole frame. Lastly, the third sub-frame is 2×3 grid with row 1 spanning all the columns.

Having diagrams like this then helps with coding as it is obvious which frame a widget is destined for and which row and column within that frame.

With the GUI laid out, the use of *TKinter* class variables, trace and button callbacks, we now have a means to interface to some script functionality to actually implement a useful application. In the case of the rv32 GUI script, this generates a header to configure the instruction set simulator features, whilst for the OSVVM co-simulation script, this is to act as an external client to connect to a logic simulation and read and write transactions.

Conclusions

In this document we have looked at constructing GUIs using Python and the *Tkinter* package that are useful as user interfaces to software and logic processes that are superior to command line scripts and command line options. Necessarily only some basic widgets have been explored, but a couple of real-world examples have shown how these basic widgets can produce quite useful applications. There are other very useful widgets that can be employed for even more elaborate scripts, such as pull down menus, tabs and graphical buttons etc. The diagram below shows my most elaborate GUI (that's available as open source).



This GUI script is part of my LatticeMico32 softcore instruction set simulator project, *mico32*, and features menus, bitmap buttons with hover help boxes and tabs, as well as the widgets were discussed in the document. It is meant to configure and then run a program on the ISS, and a pop-up text window is produced to display the output of the simulation. And even this is not all that can be done.

We only looked at grid layout for *TKinter*, but feel free to try out the other strategies. In all cases I find that once the basic GUI layout is constructed enough to run and see what the output is, there is always a certain amount of tweaking of padding, or widths etc., to get just what you want. Other GUI packages are probably easier to use (I have no experience of these), and if using these meets your needs then these are just as valid *TKinte*r. The best way to get started is to envisage a small GUI and start writing the code. The *rv32* and OSVVM examples in this document, as well as the *mico32* GUI, are freely available as open-source code for inspection and experimentation—just follow the links in the text.