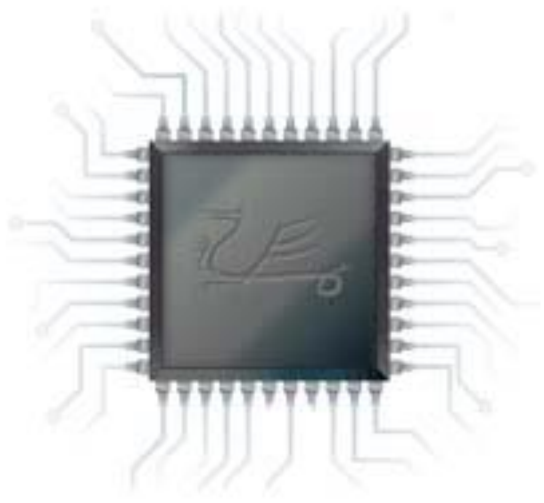
The Python/C Interface



Simon Southwell

5th September 2024

Preface

This document is a PDF version of an article written in September 2024 and uploaded to LinkedIn, on The Python/C interface and its application to co-simulation

Simon Southwell
Cambridge, UK
5th September 2024

Contents

INTRODUCTION **4**

PYTHON CALLING C **5**

Creating a Suitable C Program 5
Calling the C Code From Python 8
C types, Parameters, and Return Values 9

C CALLING PYTHON **11**

Python Library 12
The C code 12
Compiling the C code 14

CO-SIMULATION CASE STUDY **15**

The Vanilla VProc Component 16
Porting Python to VProc 17

CONCLUSIONS **20**

Introduction

In this document I want to talk about the Python C interface. Now that can be one of two directions, of course—either calling a C program from python code, or, if such a thing is possible, calling python code from C. The first of these, as we will see, is much simpler but the second is possible with a bit of know how. As well as some demonstration code in the document, we will also look at how the VProc's python interface works as a case study, including some current limitations.

I've said that it's a C interface (as this is what it is) but for those who code with C++ then you'll know about compiling a C++ function with the `export "C"` qualifier so that the C++ compiler gives it a C linkage, whilst its internal code can be C++, creating class objects etc., which C code can't do. I want to briefly talk about what this C linkage actually means, as this will be important later on (you can skip the next paragraph if you already know about this).

If you write a C function call, say, `func1`, and compile it with a C compiler like `gcc` it will give it a label called `func1` in the resultant output (whether an executable, object file, or whatever). If you compile the same code with a C++ compiler like `g++` then it will give the function a label called something like `_Z5func1v`. This is called 'mangling' and is to do with the fact that C++ can overload functions. That is, more than one function can be defined with the same name (in the source code) but with different numbers of parameters or with different types, or even different return types. Ultimately these all have to have unique labels in the output, and so the compiler 'mangles' these with additional characters depending on the parameter and return types. The Python C interface doesn't know anything about these, and just sticks to the name of the function, and so C linkage is needed when compiling with C++ so that it's not mangled. In theory, you could look up the mangled name for a particular function and use that, but this would be an awkward and manual process.

In the example code and discussion in the rest of this document I will assume this is all on Linux. I have tried these on Windows under the MSYS2/mingw-w64 Linux like environment, and there are a couple of minor differences and I'll point these out as we encounter them. I haven't tried it, but the Windows Subsystem for Linux (WSL) is likely to work exactly like a Linux environment. WSL isn't installed by default, but it is fairly straight forward and Microsoft provide instructions.

Python calling C

Creating a Suitable C Program

So, let's start with the easier of the two modes, with C being called from python. First let's have a very simple C code fragment to call from Python:

```
#include <stdio.h>

void hi (void)
{
    printf("Hello from C\n");
}

#ifdef TEST
int main(int argc, char** argv)
{
    hi();
}
#endif
```

This code, then, has a single function, `hi`, which we want to call from Python. I've added a `main` function purely for test purposes and is only compiled in if `TEST` is defined. Now we can compile this with `gcc` to be an executable that runs, assuming the source is in a file called `hello_func.c`:

```
gcc -DTEST hello_func.c -o hello_func
```

Running the executable will print the `Hello from C` message. We can look at the labels from the executable using a utility called `nm` which I believe stands for "name mangling" (as referenced above). Running `nm` on `hello_func` on my system resulted in the following output.

```
000000000000038c r __abi_tag
0000000000004010 B __bss_start
0000000000004010 b completed.0
                 w __cxa_finalize@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000001090 t deregister_tm_clones
0000000000001100 t __do_global_dtors_aux
0000000000003dc0 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003dc8 d _DYNAMIC
0000000000004010 D _edata
0000000000004018 B _end
0000000000001184 T _fini
0000000000001140 t frame_dummy
0000000000003db8 d __frame_dummy_init_array_entry
0000000000002118 r __FRAME_END__
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
```

```
0000000000002014 r __GNU_EH_FRAME_HDR
0000000000001149 T hi
0000000000001000 T _init
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
0000000000001163 T main
                 U puts@GLIBC_2.2.5
00000000000010c0 t register_tm_clones
0000000000001060 T _start
0000000000004010 D __TMC_END__
```

I've highlighted the entry for our function, and it shows a (virtual) address and is labelled 'T' indicating 'text'—i.e., code rather than data. It is this that the Python C interface can use to call the function—but not in an executable.

In the compilation for the executable the source code was both compiled and linked in one step. This hides the detail that the code would normally be compiled to an object file and then linked (potentially with other object files) to form the executable. We can, in fact, compile our code to an object file with `gcc -c hello_func.c`. This has turned the code into assembled machine code (in a file `hello_func.o`) but without committing where it will be offset within the executable, relative to other bits of code. Running `nm` on `hello_func.o` now gives:

```
0000000000000000 T hi
                 U puts
```

This is much simpler, and we have our function's label, and a `puts` label marked as 'U'. This means 'undefined' and is to do with the `printf` call made by the `hi` function. It's undefined as it is expected to be in another object file, and this will be resolved when they are linked. Multiple object files can be grouped together into a *static library* which can be linked with other object files (or other libraries) as a single unit. This is done with the `ar` (archive) utility. Say we have `hello_func.o` and `other_func.o`, we can create a library with:

```
ar cr libfuncs.a hellofunc.o other_funcs.o
```

We can link this library, containing the pre-compiled object files, with other code to make an executable:

```
gcc new_funcs.c -L. -lfuncs -o all_funcs
```

The two new options firstly define a search directory to find libraries (`-L .`), in this case the current directory, and then specify the library (`-lfuncs`). Notice that the option only needed '`funcs`' and automatically looked for a file starting '`lib`' and ending '`.a`'.

Many libraries in a Linux system are in this form. On my system, for example, is /usr/lib/x86_64-linux-gnu/libc.a, which is the standard C library.

Python isn't a compiled language, and so its source code isn't compiled and linked in the same way as for C or C++. Therefore, we need our C code to be in a different form than a static library—namely a dynamically linked library (DLL) using Windows terminology, or a shared object to use the Linux convention. Actually, the Windows naming is more descriptive of what this is, as this is a compiled library that can be linked at run-time by another program. It is nearer to a library than an object file. Whatever it should be called, it is compiled with some other gcc flags. Returning to our original hello_func.c example, we can compile this to be a shared object as shown below:

```
gcc -shared -fPIC hello_func.c -o hello_func.so
```

The -shared flag does what it says on the tin and tells gcc to generate a shared object. The -fPIC tells the compiler to generate 'position independent code'. This means it will use relative offsets for jumps and branches. Thus, the code can be located anywhere suitable in memory at run time without having to go into the code and change all the committed addresses that would otherwise have been compiled. It will also have something called a global offset table, that will have key label locations that code can use as relative offsets from these addresses, which allows the loader to easily position the code in memory by updating just this table.

If we look at the nm output for hello_func.so we get the following (on my system):

```
0000000000004028 b completed.0
                 w __cxa_finalize@GLIBC_2.2.5
0000000000001060 t deregister_tm_clones
00000000000010d0 t __do_global_dtors_aux
0000000000003e18 d __do_global_dtors_aux_fini_array_entry
0000000000004020 d __dso_handle
0000000000003e20 d _DYNAMIC
0000000000001134 t _fini
0000000000001110 t frame_dummy
0000000000003e10 d __frame_dummy_init_array_entry
00000000000020d0 r __FRAME_END__
0000000000004000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000002010 r __GNU_EH_FRAME_HDR
0000000000001119 T hi
0000000000001000 t _init
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U puts@GLIBC_2.2.5
0000000000001090 t register_tm_clones
0000000000004028 d __TMC_END__
```

This output is more like for the executable, and it is a more complete output. Notice that the `puts` function is still undefined but also has an `@GLIBC_2.2.5` tag. This is because this is going to be in another standard shared object. We can see what this is using the `ldd` (list dynamic dependencies) utility—`ldd hello_func.so`:

```
linux-vdso.so.1 (0x00007ffc1594a000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000799083600000)
/lib64/ld-linux-x86-64.so.2 (0x00007990838ef000)
```

As well as a couple of other system shared object libraries, `hello_func.so` is dependent on the standard C library, `libc`. These libraries will also be loaded (if not done so already) if our library is loaded at run time by a running program (such as Python).

Like for an executable, a shared object can be compiled from static libraries or object files, so code can be organised as required, the main difference being that a shared object doesn't have a `main` function and can't be executed by itself. Now we have a program that's suitable for use by the Python C interface.

Calling the C Code From Python

Now we have a shared object with our function compiled into it, how is it called from Python? There are a couple of steps to this, but these are straight forward enough. Firstly, we must load the shared object we created and get a 'handle' on this object using a `ctypes` python library. This will be just like a Python class object we could create, and the second step is to reference the function inside and call it. Below is shown some Python code to do just this:

```
# Load ctypes library to get access to C domain
import ctypes

cSharedObj = ctypes.CDLL('hello_func.so')

cSharedObj.hi()
```

Here we import the `ctypes` library and then use its `CDLL` method to load our shared object, saving the returned value into a variable, `cSharedObj`. How does the library method know where to look for the shared object? Well, in the example, since the single argument string has no path delimiters ('`/`') it will do a search. On Linux it will search a set of standard places but, for our purposes with our custom library, it will use the `LD_LIBRARY_PATH` environment variable. On my system this includes '`./`', so it will search the current directory. The argument can also use absolute or relative paths as well, and I could have used '`./hello_func.so`' just as well and it would not matter if `LD_LIBRARY_PATH` had a setting for the current directory, but this is hardwiring the location of the shared object making it harder to relocate if necessary. Unfortunately, on MSYS2/mingw-w64 it is more 'Windowy' and will not use the LD_LIBRARY_PATH (even though it is set in the shell). There is a separate DLL path thing which can, fortunately, be updated from inside the Python code. The os library (on Windows) has a method, add_dll_directory, to append to this search path. For Windows, then, we can update the code to the following:

```
# Load ctypes library to get access to C domain
import ctypes as ct
import os

# Add current directory to DLL search path in in Windows
if os.name == 'nt' :
  os.add_dll_directory(os.getcwd())

cSharedObj = ct.CDLL('hello_func.so')

cSharedObj.hi()
```

Note that I had to do a test that the script is running on Windows before calling the method because the os library on Linux does not have it, should it be run on that OS. Running this script now gives the following output.

```
$ python3 hello_func.py
Hello from C
```

And there we have it—we called and ran the C function from our python code. Note that any shared object can be loaded, including standard libraries, and their functions called from Python in this manner.

So that's all there is to it, right? Well, you know when you're taught something in class, like the logic for a full adder, and then the homework assignment is to design an ALU for a RISC processor, I don't want to leave things with just this. Calling a fairly passive function with no parameters that returns nothing is of limited use. There's no space here to go through them all, but if we get some basic types working, then the rest should follow on.

C types, Parameters, and Return Values

The way C and Python layout there variables can be wildly different, and we need to be able to exchange data between the two domains. The ctypes library has all the means to convert between the two and defines all sorts of types which are useful. A selection of these types are defined below:

- c_char: a C char type
- c_char_p: a pointer to a C char, zero terminated string
- c_byte: a C signed char type
- c_int: a C signed int type
- c_float: a C float type
- c_int8, c_int16, c_int32, c_int64: signed 8, 16, 32 ad 64 bit C signed integers
- c_uint8, c_uint16, c_uint32, c_uint64: signed 8, 16, 32 and 64 bit C unsigned integers
- c_void_p: A C *void type

There are other types defined, but you get the idea. There are even some specific to Windows, but I'll gloss over this. If you have an object of these type classes, then you can get the value of these using their value attribute. Any of them can also be made into arrays, with the simplest way by multiplying the type at construction, and even initializing it:

```
>>> import ctypes as ct
>>> array = (ct.c_int * 5)(1, 2, 3, 4, 5)
>>> print(ct.c_int(array[4]).value)
5
```

Accessing the array element in the print requires that it be cast to c_int so that the value attribute can be used to return a Python type integer (as print won't know anything about the C type). Alternatively, we could have cast it to a list first:

```
>>> list = array[:]
>>> print(list)
>>> [1, 2, 3, 4, 5]
```

So now a new example. We will assume that were are just extending the hello_func Python and C code. The new C function looks like the following:

```
int sum_and_clear (int *data, const unsigned len)
{
    int sum = 0;

    for (int idx = 0; idx < len; idx++)
    {
        sum += data[idx];
        data[idx] = 0;
    }

    return sum;
}
```

We can call this new function from Python, but there are a couple of things to do first. Here's the additional Python code.

```
import ctypes as ct

cSharedObj = ct.CDLL('hello_func.so') # as before

array = (ct.c_int * 6)(1, 2, 3, 4, 5, 6)

cSharedObj.sum_and_clear.argtypes = (ct.POINTER(ct.c_int), ct.c_int)
cSharedObj.sum_and_clear.restype  = ct.c_int

summation = cSharedObj.sum_and_clear(array, len(array))
print(summation, array[:])
```

Python needs to know what the argument types are, and what the return type is, as well. This is done by setting the `argtypes` and `restype` attributes of the C method of the DLL object as shown above. The first parameter is an integer pointer, so the basic C type of `c_int` is wrapped in a `POINTER` casting to return the pointer type for that base type. The second parameter is a straightforward `c_int` (which is the default type). The return, or result, type (`restype`) is also set to just `c_int` as well (also the default). We can now call the C function as before but passing in arguments and getting a returned result. For the first argument the array of `c_int` values can be used as is, and its length calculated and used as the second argument. The result is returned into a variable `summation`. The returned result, since we defined what type it was, is cast into a python integer type. So, when we run this we get:

```
>>> 21 [0, 0, 0, 0, 0, 0]
```

This confirms that the array was summed, and also cleared.

For this calling C from Python subject I've taken it to where I think useful code can be constructed but, of course, there is way more to this interface, and such things as unions and structures can be passed across the interface as well. A useful resource for the `ctypes` library can be found on the python.org website.

C Calling Python

When considering calling Python code from C a valid thought might be why? Well, we'll get to that in the case study, as one example, but there are good reasons and there is an interface to let us do just that. In fact, when we get to the case study, we'll set that we can uses the interface usefully in both directions.

Python Library

As has been mentioned before, Python is not a compiled language but an interpreted one (albeit via compiled bytecodes—but done when the script is run). Therefore, we can't 'compile' it into an object that C can then load and make reference to functions within it. What we actually need is a python interpreter built in with the C code that can run the script. Fortunately, Python provides a library for just this function. On my system the library (for Python v3.10) is at:

```
/usr/lib/python3.10/config-3.10-x86_64-linux-gnu/libpython3.10.so
```

There is also a static version of the library (`libpython3.10.a`) in the same directory. The header for the library is in `/usr/include/python3.10/Python.h` When we construct our source code we can use this header and link with the library to gain access to the C to Python features. Fortunately, these rather esoteric paths can be discerned using the `python3-config` utility. For example, with the `--includes` flag it will return the `gcc` command line options for the include paths. Similarly for the libraries and their paths with the `--ldflags` and `--libs` command line options.

The C code

There needs to be some Python to call before it can be called from C, so lets create a simple Python function to call, in a file called `PyFunc.py`, which takes two integer arguments, compares them and returns 1 if the first argument is greater than the second, 0 if they're equal, and -1 if argument 1 is less than argument 2.

```python
def py_func (arg1, arg2) :

  if arg1 > arg2 :
    return 1
  elif arg1 == arg2 :
    return 0
  else :
    return -1
```

There is nothing special about the Python function defined above, and nothing different needs to be done to make it accessible to C. All the action is done in the C domain.

Just like for the Python code calling C functions needed C type objects to handle these values, C needs something similar for Python types and a `PyObject` type is defined to handle these. Various functions are also provided by the Python library and before we can do anything we have to initialise the interface using a function called

Py_Initialize. Let's see some code (I've not put any checking in or any deleting of no longer needed objects, but this should be done at each stage):

```
#include <stdio.h>
#include <Python.h>

int main (int argc, char** arg1)
{

  Py_Initialize();   // only done once

  // Load Python script, and get python function object
  PyObject *pName     = PyUnicode_DecodeFSDefault("PyFunc");
  PyObject *pModule   = PyImport_Import(pName);
  PyObject *pFunc     = PyObject_GetAttrString(pModule, "py_func");

  // Create argument tuple
  PyObject *argsTuple = PyTuple_New(2);
  PyObject *pVal1     = PyLong_FromLong(123);
  PyObject *pVal2     = PyLong_FromLong(321);
  PyTuple_SetItem(argsTuple, 0, pIntVal1);
  PyTuple_SetItem(argsTuple, 1, pIntVal2);

  // Call the Python function
  PyObject *pValue    = PyObject_CallObject(pFunc, argsTuple);

  // Get the return value as an integer
  int rtnval          = PyLong_AsLong(pValue);

  printf("rtnval = %d\n", rtnval);

  return 0;
}
```

This program calls Py_Initialize before anything else, as required, and then we need to load the Python program. Everything is going to be a PyObject pointer, and so we convert a character string with the name of the Python program, without its .py suffix, to an object using a very cryptically named function which will look for it on the file system. Using this name object we load it with the PyImport_Import function and pModule is now pointing to the object for this whole loaded script. To access a particular function within that script we create a pointer to the function object with PyObject_GetAttrString, which searches the module object for the object inside as named by the string, which is the name of the function, py_func, we want to call. So we now have a handle on the function to call, but we can't just call it like a C function. If it has parameters, as ours does, then we must construct a 'tuple' with objects containing our argument values. A new tuple is created with PyTuple_New and takes an argument for how many entries we need for the parameters—two for the example. Each entry in the tuple needs to be an object, and these are created with

PyLong_FromLong with the argument values we want to pass to the Python function. These could have been constructed from other types, with PyLong_FromString as another example—handy if passing in arguments from the programs command line. Once there are objects for the arguments we set the entries in the tuple with PyTuple_SetItem, with an index to select the entry and the argument object. Now (finally) we can call the Python function by using PyObject_CallObject, passing in the function object and the tuple object. It returns another object with the result, and we can extract the integer value with PyLong_AsLong.

I mentioned before that code would normally check at each object's creation that no error occurred, with the functions returning NULL if something went wrong. If it did, a call to PyErr_Print will give a clue as to what went wrong. For example, the module object returned could equal NULL if there's a syntax error in the script, and the call to PyErr_Print will highlight the particular problem. Another check can be on the function object to make sure it's a callable object. Over the interface other objects can be fetched, such as a global variable, say, but they aren't callable. So PyCallable_Check can be used to make sure the handle is a callable object. In addition we created objects all over the place before finally calling the function, and even created another one to extract the returned value. It is important that these objects are destroyed when no longer needed. In the example many are intermediate, such as pName, pVal1 etc. Others may need a longer lifespan, such a pModule and pFunc but these should also be destroyed when finished with. The Py_DECREF function is used to do this. An object when created returns a single reference to the object, but other object pointers could point to this, and so it keeps a count, incrementing each time a new reference is created to the object. When code has finished with its particular reference to the object, Py_DECREF is called with the object pointer, and the internal object reference count is decremented. When the count reaches 0, the object is removed from memory. There is even a Py_INCREF function for cases where an object pointer is, say, copied without reference to the Python API functions. Not managing this properly can cause memory leaks in a large and complicated program.

Compiling the C code

Now we have our C code it needs to be compiled with flags to find the Python header and to find and link the python library. On my Linux system the command to compile the example program is shown below:

```
gcc -I/usr/include/python3.10 py_func.c -L/usr/lib -lpython3.10 \
    -o py_func
```

Your milage may vary with the paths, depending on the Python version and how it was installed. As mentioned before, you can use the `python3-config` utility to discern these paths for your system.

In order to run this compiled program, it needs to know where to find the Python script. Unless its in one of the standard Python installation locations (unlikely) it won't be able to. Since the example program doesn't check the API call return values (naughty) then it will crash without say what went wrong. The way to solve this is by setting the environment variable `PYTHONPATH` to include the directory where the script is located. The simplest thing to do, if the script is in the same directory, is to add a relative path with:

```
export PYTHONPATH=$PYTHONPATH:./
```

Of course, for real examples, the scripts would be located in proper organised directories, and `PYTHONPATH` set up correctly for these, usually using absolute paths. Now when we run the program we get:

```
$ py_func
rtnval = 1
```

At this point, then, this is complete for our example. There is more to this interface, of course. It is kind of object oriented, but in C, and the function names are not at all easy to remember and I don't find it particularly intuitive to use. I make constant reference to Python/C API reference Manual on docs.python.org.

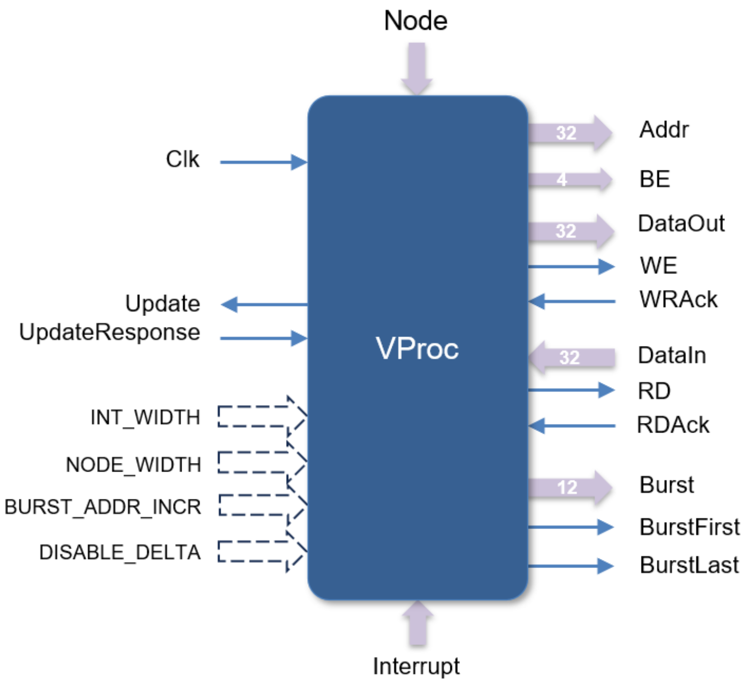
So we get back to the question of when might it be useful to call Python from C (or C++) code. One can execute Python scripts by simply calling Python from a system call and running the script in a separate process, perhaps with result sent to a file which can then be read by the C program when the system call to run Python has completed. For many purposes this would work well and is much simpler, but there are uses for this interface, and in the next section I want, as a case study, to summarise how the VProc virtual processor uses this, along with the C from Python interface, to add the Python language scripts to the programs that can be run on VProc, and thus co-simulate with logic on a simulator.

Co-simulation Case Study

I have written about the VProc virtual processor before, looking at how it works and how it can be used in various co-simulation application, from simple test programs, to running a RISC-V processor instruction set simulator, to drive a memory mapped bus in a logic simulation, enabling software and hardware co-design, and many other applications. So I don't want to go in to detail here, but will summarise the key points relevant to supporting Python on the device.

The Vanilla VProc Component

VProc has an HDL component (Verilog/SystemVerilog or VHDL) with a generic 32-bit memory mapped master interface, with address, write data with strobe and ack, and read data with strobe and ack. It also has other optional features for byte enables, bursts, interrupts and for delta-cycle updates, but we will skip these details for this discussion. The VProc component is shown below.



The VProc component might be used just as it is, but more usually it is wrapped in a bus functional model (BFM) behavioural wrapper to convert the generic signals to another bus protocol, such as AXI or Avalon, and this has, in fact, been done in the past. It can even be used, with delta cycle updates, to drive completely different kinds of interfaces, such as PCIe and USB, but this is beyond this document's scope.

On this component can be run C programs that are just normal programs compiled for the host computer as (usually) shared objects. The VProc HDL component is given a node number in the logic simulation (on its `Node` port) and, if multiple VProcs are instantiated, each must be given a unique node number. When the simulation is run the component will look for and run a C function called `VUserMain`*`<node>`*, which is the entry point for the user code running for that particular node (e.g., `VUserMain0`). A basic C API is provided to the user code, though this is also available wrapped up in a C++ class (but since the Python/C interface is C, we'll stick to this basic API). It is fairly lightweight, but we will mention only the three most essential functions that can do most of the work for most situations. Their prototypes are shown below.

```
int VWrite (const unsigned addr, const unsigned wdata, const int delta,
            const unsigned node);

int VRead  (const unsigned addr, unsigned *rdata, const int delta,
            const unsigned node);

int VTick  (const unsigned ticks, const unsigned  node);
```

Using these API functions C code can write and read over the generic bus or allow simulation time to pass. This latter is important as, as far as simulation time is concerned, the VProc code is running infinitely fast. The `VTick` function can be called to advance simulation time by a number of clock ticks to simulate processing time of the code, otherwise the read and write transactions will occur back to back. Often, when using VProc, I have wrapped the read and write API calls in new functions with a call to `VTick` with a random delay value between, say, 0 and 10 cycles to emulate processing and to give various delay cases to the simulation.

The read and write functions both have address parameters and either a data value (for writes) or a pointer for returning read data. The delta argument is for flagging whether it is a delta-cycle transaction or not, so let's assume its 0, and the node number defines which VProc node the code is running on, matching the called `VUserMain` function.

This is the basic VProc operation, but we want, on occasion, to write our VProc program in Python, not C/C++ and we can use the Python/C interfaces to do just this.

Porting Python to VProc

To mirror what the C code is doing, we want a Python function VUserMain<*node*> to be our entry point for the Python code. This code will have access to an API with the write, read and tick functionality (as well as all the rest) as methods within an API python class.

To gain access to the VProc C API we will need, firstly, to have a shared object that can be called from Python. VProc, coincidentally, compiles its user and VProc source code into a shared object, `VProc.so`. This is because most of the simulators require the programming logic interface code to be in this form: e.g., `vsim -pli VProc.so`. However, it is not desirable to have the Python interface library in the shared object the simulator will use, or to have all the VProc code in the shared object that the Python script will use. So `VProc.so` will be compiled as before, with `VUserMain` being a special (non-user) program that will make the connection with Python. This is in python/src/VUserMainPy.c. An additional PyVProc.so is compiled that contains the code for the Python interfacing, with source code located in python/src/PythonVProc.c (and accompanying header file PythonVProc.h). In the PythonVProc.c source are a load of wrapper functions to the VProc C API, including read, write and tick functions:

```
uint32_t PyWrite (const uint32_t addr, const uint32_t wdata, const int delta,
                  const uint32_t node);

uint32_t PyRead  (const unsigned addr, const int delta, const uint32_t node);

uint32_t PyTick  (const uint32_t ticks, const uint32_t  node);
```

These can't call the API functions directly. The Python script will load the PyVProc.so shared object, but not the VProc.so. It will, however, be in memory as the simulator will have loaded and run it, otherwise the Python script wouldn't be running. It's beyond the scope of this document to go into details, but like for the Python calling C code in the examples, C can also load a shared object using the dynamical loading libdl library to get handles on these functions and call them. The source code has a function called BindToApiFuncs that does this, using dlopen to access the VProc.so shared object and then look up the symbols for all the API functions using dlsym, which are saved off into function pointers. It is these that are called by the wrapper functions to access the API.

The API is now accessible from a Python program that loads the PyVproc.so shared object, but we must first run that Python program and the C to Python interface allows us to do just this. The program running on the VProc component, VUserMain0 for example, can now do what we did in the example above to access the script we want to run. In fact, the code simply calls another function RunPython(node) which does all the hard work, defined in python/src/PythonVProc.c. The content of this function looks very similar to the example, which was based on this, except it does all the proper checking and dereferencing. Depending on the node argument, it will try and load a VUserMain<node>.py script (e.g., VUserMain0.py) and then call a Python function called VUserMain<node> (e.g. VUserMain0).

In the test setup example in VProc, the user Python script is in the directory python/test/usercode. It makes use of the PyVProcClass defined in the script python/modules/pyvproc.py. The PyVProc.so shared object is loaded at construction of an object for this class and the provides API methods which wrap up calls to the C wrapper code, along with support for handling interrupts (not covered here). An abbreviated code fragment is show below:

```
class PyVProcClass :

  node = -1
  api  = None

  def __init__(self, nodeIn, cmodulename = "./PyVProc.so") :
    self.node = nodeIn
    self.api  = self.__loadPyModule(cmodulename)

  def __loadPyModule(self, name = "./PyVProc.so") :

    module = CDLL(name)
    return module

  def write (self, addr, data, delta = 0) :
    self.api.PyWrite(addr, data, delta, self.node)

  def read (self, addr,  delta = 0) :
    return self.api.PyRead(addr, delta, self.node)

  def tick (self, ticks) :
    for i in range(ticks) :
      self.api.PyTick(1, self.node) # tick once for interrupt granularity
```

The use of this class now exposes the `VUserMain0` Python function to all the VProc API functionality. The script containing this function is called using the C to Python interface, as described earlier, and the class makes calls to the C API using the Python to C interface, and the circle is complete.

There is one limitation that I have yet to solve in a sensible manner. VProc can support multiple VProc instantiations in an HDL simulation, each running their own programs. This is done using threads, though the details are not important. However, I have found that instantiating multiple Python running VProcs does not work as the Python/C interface doesn't appear to be re-entrant. That is, state is kept for only one interface, and a second use of the interface interferes with an already active use. This is disappointing as Python itself does support threads. There are things that could be done to ensure no clashes. The simplest solution is just to ensure only one `VUserMain` program has access to the interface at a time, but this means blocking it until a whole transaction has completed, and thus only one VProc component in the simulation would have an active transaction at a time, which is not a good verification limitation. Other solutions are to have non-blocking functions to pass requests over the interface (in zero simulation time), which get re-synchronised once the transaction is complete. Implementations I have attempted get very messy very quickly and this is still pending. For now, only one Python VProc can be instantiated at a time, and an error is generated if multiple instantiations are present. A single Python VProc can be used with multiple other C VProc instantiations (which do not have this issue) if that were useful.

Nonetheless, the current implementation serves as a working example and the code is readily available on github.

Conclusions

In this document we have looked at the Python/C interface, both from the perspective of calling C functions from a Python script, and for calling Python functions from C. The first of these is somewhat easier to use and understand, with the code compiled to a standard shared object which can be loaded and accessed with the `ctypes` library, but with care taken over the C types being somewhat different to the Python types.

The opposite direction is a little more constrained, with a C-to-Python library provided and a (somewhat esoteric_ C API, where everything is an object, and setting up arguments is a little convolved. Nonetheless, it does what's required, and details can be abstracted away from other code.

With the interface able to call in both directions, we looked at how this is exploited in the case study of running Python scripts on the VProc virtual processor to drive a memory mapped bus in a co-simulation system on a logic simulator.

This concludes the introduction to this interface, and it can be used in many more ways that are described here, I'm sure.