Chapter 4

Interface for User Control Functions

4.1 Introduction

The previous chapter introduced the user interface to the circuit simulator. In particular, the simulation parameter spreadsheet circuit_inputs.csv was described and it was shown how a row was devoted to specifying the control functions in the simulation. In the previous chapter, the data structure of each type of circuit component was described in the form of tables. The components that could be controlled were described to have ControlTags with control values that can be modified by user defined control functions. This chapter is completely devoted to user-defined control functions. The word controller and control code will be used for any code that the user wants to implement in the circuit simulation. As already stated before, it is not necessary that this control code be performing a control action and for that matter there need not be a controllable component in the circuit. This control code could be merely a processing tool used by the user in real-time as the simulation progresses such as calculating efficiency, harmonic content, root mean square values etc.

The prime target of the circuit simulator is to simulate circuits with multiple power electronic converters. Therefore, including control functions in a simulation must be convenient to the user as the user may need to design control functions for a number of a converters. This chapter will describe how each control function has a descriptor which defines not only the input-output map of the control function but also defines special variables for the control function. The chapter will describe the need to define special variables and how the user can use them particularly when cascaded or embedded control functions need to be designed.

In addition, the chapter will also discuss the concept of time scheduling of control events and also how the circuit simulator can ensure that every control code will run at exactly the intended time instant even when the time step of the controller can be very different from the simulation time step.

This chapter primarily intends to show how the control interface with this circuit simulator is about as effective and easy to use as the control interfaces available with most commercial software. This chapter describes how multiple control functions can be incorporated in a single simulation and how the simulator enables connectivity between control functions. The objective of designing the control interface in this manner is to replicate the implementation of control algorithms on micro-controllers in hardware. In the future, the option might be provided to design control with a particular hardware architecture to decrease the time for implementing the final hardware.

4.2 Inclusion of control in the simulator

Chapter 3 described the user interface for designing the circuit and specifying simulation parameters. As was mentioned, the spreadsheet circuit_inputs.csv specified the simulation parameters - circuit schematic spreadsheets, simulation time step and time duration of simulation. There is also another row for control files. If this row is left blank, the simulator will not process any control files. If the user wishes to specify control files, the user can specify any number of control files in separate cells in that same row. Table 4.1 is an example of how

Table 4.1: Control files in circuit_inputs.csv

Name of control files	control1.py	control2.py

two control files control1.py and control2.py can be specified by the user in circuit_inputs.csv. There is no limit to the number of control files that can be included in a simulation. At the time of writing this book, control can be implemented as Python 2 code and the control files have to listed as .py files. Specifying a control file that does not exist is an error and the simulator will abort with an error message.

The simulator deals with user control files in a manner that is described by the following block diagram. The user writes code as a .py file. The control code written by the user are

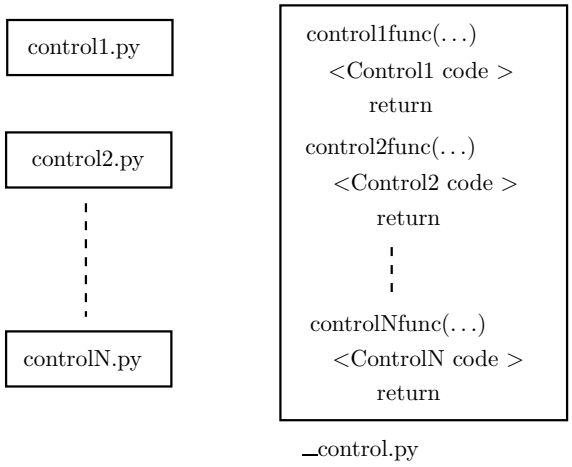

Figure 4.1: Inclusion of control functions in the simulator

shown as files Control1.py, Control2.py to ControlN.py on the left. Each such control file is inserted into a function by the simulator. This is shown on the right. Control1.py is inserted into the function Control1_func, Control2.py is inserted into the function Control2_func and so on. Essentially the function which contains a user control file has "_func" appended to the name of the control file. This enables the simulator to execute the control files by evaluating the functions. Since there can be multiple control files in a simulation, the code in each control file will be inserted into a separate function by the simulator. All these functions will in turn be inserted into the file __control.py. Therefore, the user cannot name a control file by this name. The simulator will import the file __control.py thereby gaining access to all the member functions within it which are the functions containing user control code.

In most commercial software, a controller is usually a block with input and output ports. In a similar manner it becomes necessary to describe the port connections with this circuit simulator. This is done by a descriptor spreadsheet. When the user wishes to implement a controller, all the user needs to do at the beginning is name the file - for example, let us call a control file by user_control.py. When the circuit simulator is notified of the control file in circuit_inputs.csv, the simulator will look if a descriptor exists for such a file. Any control file will have a corresponding descriptor spreadsheet with _desc.csv appended to the name - for example, for a control file user_control.py, the descriptor file will be user_control_desc.csv. When the user specifies a new control file, a blank descriptor spreadsheet with default input and output will be created by the simulator. The flowchart in Fig. 4.2 describes the process.

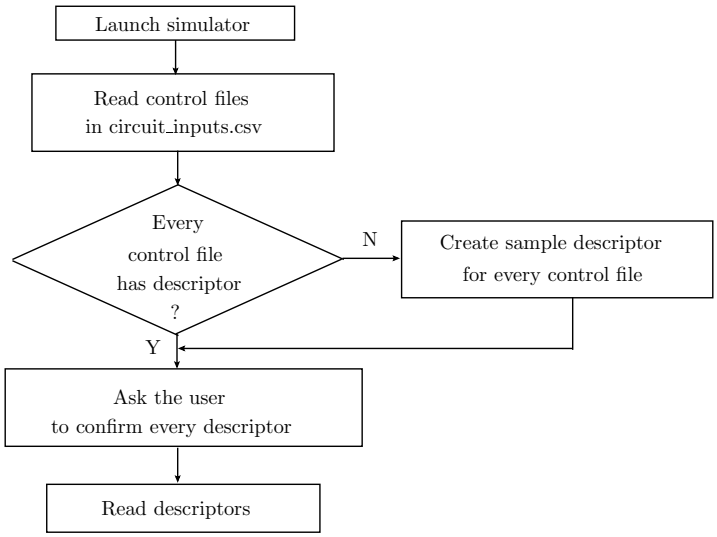

Figure 4.2: Descriptor for control functions

When the simulator finds a descriptor file, it will use that file to read the latest parameters of each control file. We will now look at the concept of how a control file is handled by the simulator with the following basic example with a single input and single output.

The inputs to the controller are usually meter outputs - ammeter and/or voltmeter. The later section will describe how a controller could be interfaced with another one. The circuit simulator will create a single entry in any new control descriptor file with the structure shown in Table 4.2. These three fields of an input port will be present in a row of any descriptor

Table 4.2: Descriptor entry for control input

Input	Element name in circuit spreadsheet = Ammeter_A1	Desired variable name in control code = curr_input

spreadsheet the simulator creates by default. The first column "Input" is to let the user know that the row corresponds to an input port to the controller. The second column is the meter which serves as the input to the control code. This has to be equal to the component which is the meter. Notice that the entire component as it appears in the spreadsheet should be specified. The third column is for the user to specify how the user wants to access this input in the control code. For the above example, the user can access the measured current output of Ammeter_A1 by the variable curr_input in the control code user_control.py. The simulator

will copy the current output of Ammeter_A1 to the variable curr_input and then execute the control code. This variable name is up to the user and can be anything as long as it is a legal variable name and also should be unique within the control code - i.e it should not be used for another input or output port or any other variable to avoid corruption of data. The circuit simulator will create only one such row above and will extract a meter at random from the circuit. The user can have multiple inputs by adding rows similar to the one above and adding more meters. There is no limit to the number of inputs there are in a control descriptor spreadsheet. The simulator determines a row to be an input port when the first column is "Input".

The next port to be described is the Output port. A row in the descriptor spreadsheet corresponding to an Output port will be as shown in Table 4.3. The first two columns are

Table 4.3: Descriptor entry for control output

Output	Element name in circuit spreadsheet = ControlledVoltageSource_Vin	Control tag defined in parameters spreadsheet = Vsource	Desired variable name in control code = Vsource	Initial output value = 0

similar to the Input port. The word "Output" in the first column tells the simulator that the row corresponds to an output port. The second column is the name of the controllable component. In this example, a ControllableVoltageSource_Vin has been used. The simulator will look for a controllable component in the circuit, and if one is found will insert that into the descriptor spreadsheet as an example for the user to add others. The simulator will insert only one controllable component and this is chosen randomly. The third column is the control tag of the controllable component. At this point, the reader should refer to Chapter 3 where the ControllableVoltageSource component type is described. Any controllable component type has a parameter called ControlTags. This is a parameter of the component type that can be changed by the user in the parameters descriptor spreadsheet. To begin with this parameter is a list because a controllable component could accept multiple control signals. In the case of the ControllableVoltageSource, the list has only one parameter and that is the desired output voltage of the source. Therefore, ControlTags by default would be ["Voltage"].

This is the default name given by the simulator to the control input of the component of type ControlledVoltageSource in the parameter spreadsheet. As stated in Chapter 3, it is always recommended to change the name of the inputs in ControlTags to something unique and easily identifiable. Let us then, rename the control input to Vsource. This renaming is not done in the control descriptor spreadsheet. It is done in the parameter spreadsheet corresponding to the circuit as the name of the control input is a parameter of the component ControllableVoltageSource_Vin. In the control descriptor spreadsheet, the simulator automatically extracts one of the items in the parameter ControlTags from the component object ControllableVoltageSource_Vin and inserts it into the third column as shown in Table 4.3. It should be noted that in case a component does have multiple control inputs and ControlTags is a list with multiple entries, the entries would have to be made in separate rows of the descriptor spreadsheet. A single row of a descriptor spreadsheet will only connect a single control input of a controllable component to a variable inside the control code. The simulator will only extract one control tag of a controllable component to serve as an example to the user. The fourth column will ask for the name of the variable that the user wishes to refer to the control input by. In the above example, it has been used as Vsource. Therefore, any value assigned to the variable Vsource in the control code will automatically be transferred to the Vsource ControlTag in ControllableVoltageSource_Vin. The last column is an initial value. This will be the value of the output when the simulation starts.

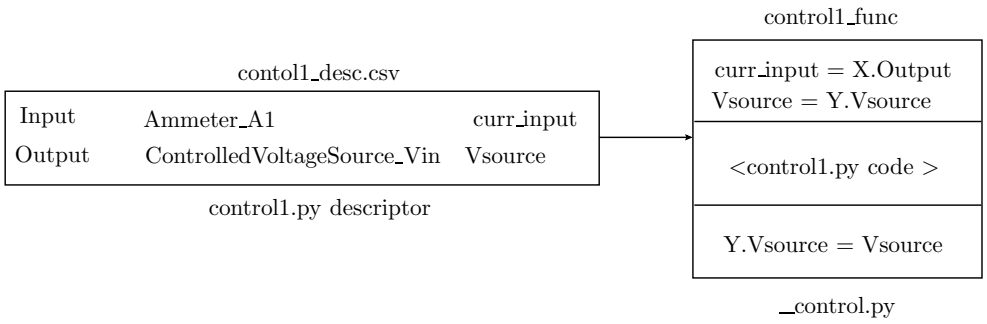

Figure 4.3: Input and output ports in a control function

The block diagram of Fig. 4.3 describes how the simulator uses the control descriptor spreadsheet. Let us consider a user file Control1.py. This control file will have the descriptor spreadsheet Control1_desc.csv. If it exists, the simulator will read the parameters of the control file and if it doesn't exist, the simulator will create a blank spreadsheet for the user

to change. On the right is the function Control1_func into which the control code within Control1.py is inserted by the simulator. Let us consider the basic example above of a single Input port and a single Output port. On the right, it is shown how the Input and Output ports are made available to the user. The example above considers an Input port which is fed by an Ammeter_A1 and the variable by which the user can access the Ammeter measured current is through the variable curr_input. The simulator assigns the variable curr_input to the current output of the Ammeter_A1 inside the function Control1_func before the user code with the statement:

```
curr_input = X.Output
```

Here X is the component object corresponding to the Ammeter_A1. This concept has been explained in Chapter 3. Each component in the circuit is an object created by instantiating a class of that component type. Therefore, in the above statement, X is the object created by instantiating a class of the Ammeter type for the component Ammeter_A1. Chapter 3 has the details of how this is done. What needs to be emphasised here is that the simulator automatically extracts the measured output of Ammeter_A1 by accessing the object corresponding to Ammeter_A1 and assigning it to curr_input. Therefore, the subsequent user control code which uses curr_input now contains the measured current of Ammeter_A1.

In a similar manner, the output is also interfaced. The only difference being that Output ports are interfaced twice - before the user control code and after the user control code. The variable Vsource is made available to the user to access the ControlTag Vsource of ControllableVoltageSource_Vin. The user control code will therefore change the variable Vsource within control code using any set of statements. This variable Vsource will directly alter the ControlTag of the controllable component specified in the descriptor spreadsheet - in this case, the ControlTag is Vsource. This is done using the last statement:

```
Y.Vsource = Vsource
```

In this case, Y is the object produced by instantiating the class ControllableVoltageSource for the component ControllableVoltageSource_Vin in the circuit. Therefore, any change in the user variable Vsource will automatically be transferred to the controllable component. The reverse statement is included before the control code in the function:

```
Vsource = Y.Vsource
```

This is to provide the user with the updated value of the ControlTag of the component in case another control function also changes the value of the ControlTag.

4.3 Special variables in control code

As described in the previous section, any control code written by the user and specified in simulation parameters spreadsheet circuit_inputs.csv will be inserted into a function. So control code in Control1.py will be inserted into the function Control1_func. All the control functions will be written in the file __control.py. By doing so, the file __control.py can be imported by the simulator and each control function can be executed using the "eval" function.

The previous section described how a basic controller can be designed with one input and one output port. However, with just input and output ports and no other special functionalities, only extremely simple controllers can be designed. As an example, the user can use a number of variables for various mathematical operations. In Python, it is not necessary to declare a variable. A variable comes into existence the first time it is used. However, in the context of a function, a variable exists only within the function. The variable is created when the function is called and is destroyed when the function is terminated. Any variable used by the control code will not store its value between iterations and will always start with default values. In some cases, this may be a problem. Consider the case of an integrator. The user wishes to integrate the measured current curr_input in the previous example. The expression for that would be:

```
curr_integ = curr_integ + curr_input*dt
```

As is fairly obvious from the above equation, the integrator is based on storage. It adds the new value curr_input*dt to the stored value of the integrator - curr_integ. If curr_integ is initialized to a default value (say zero) in the beginning of the control code, when the function is executed at every iteration, it will always start at this default value and add only the latest integrator input. This is not the desired operation of the integrator. The integrator value should be initialized to a default value once in the beginning of the simulation and after that should accumulate the latest integrator inputs. In order to do this, we need to store the integrator output between iterations of the simulation. This can be done with StaticVariables.

A StaticVariable is a parameter of a controller. This implies it has to be defined in the

Table 4.4: Descriptor entry for StaticVariables

StaticVariable	Desired variable name in control code = curr_integ	Initial value of variable = 0.0

control descriptor spreadsheet. Table 4.4 is a typical entry for a StaticVariable in a control descriptor spreadsheet. As an example, let us consider the variable curr_integ above. The first column StaticVariable tells the simulator that the row describes the parameters of a StaticVariable. The second column is the name of the variable the user wishes to use for this static variable. In a manner similar to the Input and Output ports, the user can use the defined StaticVariable in control code and the simulator will ensure that the latest value of the StaticVariable is copied into it before the user code begins. The third column is the initial value of the StaticVariable. The default is zero but it can be any finite number. Similar to ports, a control code can have any number of StaticVariables. StaticVariables need to be unique within a control code but they can be repeated in other control codes. The simulator will keep StaticVariables in a control code separate from the StaticVariables in other control code. Therefore, StaticVariables are local to a control code and cannot be used to share data between controllers. It is recommended that a user declare the variables in user control code as StaticVariables unless a variable is a constant or some other basic parameter which is initialized at the beginning of the control code. In that case, the variable will be assigned a value every time the user control function is evaluated.

StaticVariables within a user control function are dictionary items. In the above example of the integrator, the StaticVariable curr_integ is in a dictionary with another StaticVariable pi_output:

```
{''curr_integ'': 0, ''pi_output'': 0}
```

The keys of the dictionary are the names of the StaticVariable. This ensures that a StaticVariable within a file is unique as two keys in a dictionary cannot be identical. StaticVariables need to be unique within a control function to prevent data corruption. However, identical control code can be present in different user control functions and this implies identical StaticVariables in different user control functions. This functionality could be needed when a circuit could have modular but identical blocks where each block is controlled by identi-

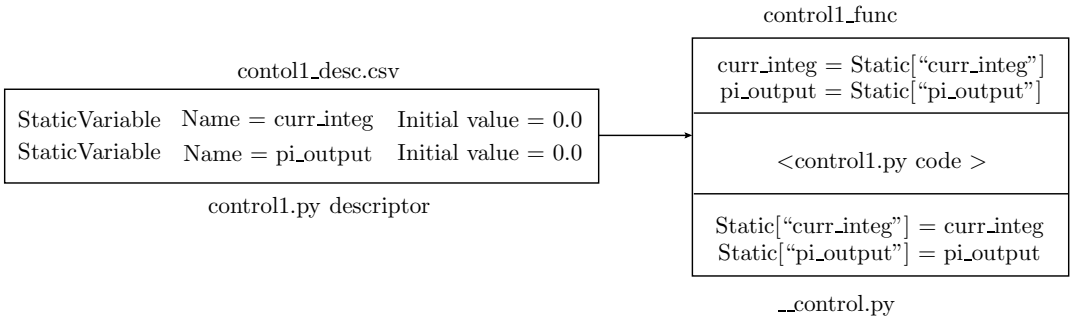

Figure 4.4: StaticVariables in a control function

cal user control functions where the only difference may be the inputs and the outputs. To achieve this, each user control function has its own StaticVariable dictionary. The dictionary structure makes it convenient to insert the variables into the user control functions and also to extract the updated values at the end of the control function as shown in the block diagram of Fig. 4.4. Since the keys of the StaticVariable dictionary are the desired names of the StaticVariables to be used in the control code, the assignments before and after the user control code ensure that the latest values of the StaticVariables are passed back and forth between the user control functions. These assignments are done automatically by the simulator in a manner similar to the previous cases.

As stated before StaticVariables are defined for every user control function and the StaticVariables defined for one are not accessible in another. In the previous section, Input and Output ports for a control function were described. However, the input ports are typically the measured values of meters like Ammeters and Voltmeters while the Output ports are control signals to ControlTags of controllable component objects. However, at times, control files need to be connected together - the output of one control file is the input to another. Complex control schemes may have several stages of cascaded control or embedded control. Another aspect that is not dealt with using StaticVariables or Output ports is that in order to debug a controller, it may be necessary to plot a control variable. In order to do so, the control variable needs to be written to the output data file and therefore access outside the user control function needs to be provided. Another type of variable is provided for both these tasks and these are called VariableStorage. VariableStorage objects have two properties - they are made available to all control functions that the user defines and they have a field specifying whether they should be plotted in the output data file. The parameters of a

Table 4.5: Descriptor entry for VariableStorage elements

VariableStorage	Desired variable name in control code = plot_variable1	Initial value of variable = 0.0	Plot variable in output file = yes

VariableStorage type are listed in Table 4.5. The VariableStorage in the first column shows the circuit simulator that the variable is of the type of VariableStorage. The second column is the name of the variable that the user wishes to use in the control code. Since, this is a stored variable, an initial value is needed to ensure that a random garbage value is not generated. The last column is the column that asks the user whether the variable should be written in the output data file so that it can be plotted by the user. If the user says "Yes" as in the above case, the variable will be written to the output data file while if the user days "no", the variable will not be written in the output data file. In either case, a variable of type VariableStorage, will be made available across all control functions, irrespective of which control file descriptor contains the definition. On the contrary, once a variable has been defined in a control file descriptor spreadsheet, a repeat definition in another control file descriptor spreadsheet is an error and the simulator will abort with an error message.

The method of implementation of VariableStorage is different from other variable types. Instead of having different dictionaries for each user control function, there is a single dictionary for all the user control files in a simulation. By defining a single dictionary and making it available to all user control functions, the sharing of data contained by the VariableStorage elements is possible across all user control functions.

```
{''plot_variable1'': [0.0, ''yes'']}
```

The above dictionary shows how the VariableStorage element in the example above is defined. The value of the key is equal to the name of the VariableStorage object while the value is a list with two elements - the first being the value of the VariableStorage element and the second being the flag whether the element should be written to the output data file. Fig. 4.5 will show how VariableStorage elements are used by the simulator. The VariableStorage element "plot_variable1" above could have been defined in the control descriptor spreadsheets of any one of the control files. However, as shown, it is made available in each user control function

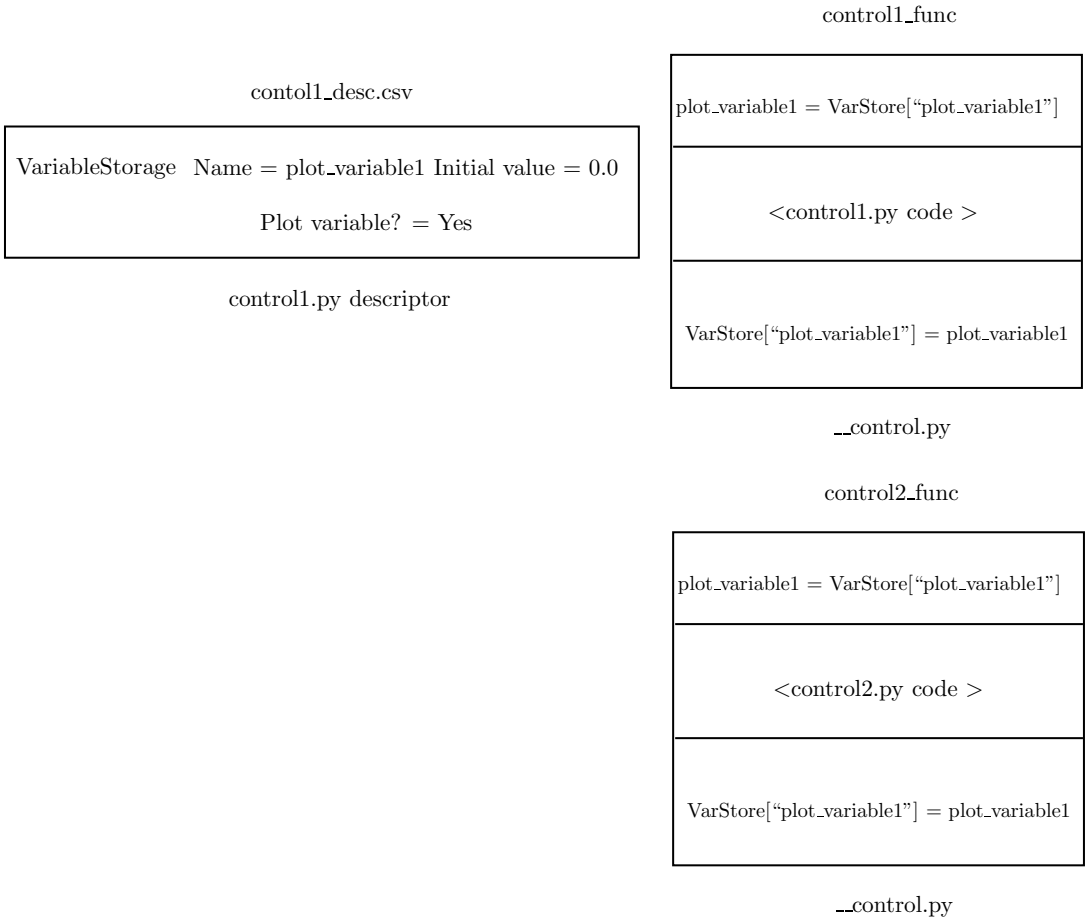

Figure 4.5: Variable storage types in control functions

before the user control code and also extracted from each user control function at the end of the control code. As can be seen from the block diagram of Fig. 4.5, VariableStorage is as extremely powerful variable type which is similar to a global variable in other programming languages. It could provide a great deal of flexibility in control design by allowing exchange of control variables. However, there is a great potential of data corruption as variables can be changed in every control function. Therefore, the use of this VariableStorage element should be limited to writing control variables to the output data file and for connection between control files. The use of VariableStorage elements as replacements for regular StaticVariables is not recommended as the risk of data corruption is high. The example provided in Chapter 5 will describe the recommended use.

With respect to these two variable types described above, there is a special property related to the Python programming language that the user can use. When a variable is

declared as StaticVariable or VariableStorage, by default it is a floating point number whose default initial value is 0.0. The user can change this default initial value to any other floating point number. However, in Python, every variable is an object of a particular type. The object corresponding to a variable can be changed by initializing it to that type. To elaborate, let us examine a variable "x":

```
x = 0.0
```

In Python, a variable does not have to be declared. Therefore, the above statement will create an object of floating point type and assign that to x. However, if another statement is written:

```
x = [0.0, 0.0, 0.0]
```

This will destroy the object of floating point type and create another object of type list and assign that to x. This facility in Python can be used to create other objects for StaticVariables and VariableStorage. For example, the StaticVariable "curr_integ" defined above and initialized to 0.0 in the control descriptor can be changed by the user within the control code to:

```
curr_integ = [0.0, 0.0, 0.0]
```

By writing the above statement in the control code, the StaticVariable dictionary for the control function will have the key "curr_integ" with the above list as its value rather than the floating point number. This is a particularly useful concept if control code needs to be written in matrix form for polyphase systems. The above variable curr_integ could now contain the integrated values of the currents in phase a, phase b, and phase c of a three-phase power system.

4.4 Time scheduling control code

The previous sections described the input/output structure of control in the simulator and also the special types of variables that can be used - StaticVariables and VariableStorage. In this section, time scheduling of control functions will be described. A control function is very rarely required to execute at the same time step as the simulation. In most cases, the time step of the control function is much larger than the simulation time step as this is how the

control will be implemented in a microcontroller or microprocessor and for real-time control, practical values of control frequency need to be chosen. In some cases, the control function needs to be run at a fixed time step due to the nature of control - for example, in a resonant converter, the control has to be timed with respect to the resonance of the circuit.

Every control function defined by the user will be provided with the current time instant of simulation through the construct t_clock. This variable can be accessed in every control algorithm by the user with the simulator updating this variable with the latest time instant of simulation before the user control code. In order to schedule the control code, the simulator has a special construct called a TimeEvent. The parameters of a TimeEvent are listed in Table 4.6. As before, the TimeEvent in the first column tells the simulator these are the

Table 4.6: Descriptor entry for TimeEvent variables

TimeEvent	Desired variable name in control code = t1	First time event = 0.0

parameters of a time event variable. The second column is the desired name in the control code. The user can use this variable to assign control events at future time instants and this variable will be stored and used by the simulator to ensure that the control code executes at that time instant. The third column is the first time event that needs to be scheduled. The default is the start time 0.0 but can be changed to anything. This time event is typically used in conjunction with t_clock in the manner described below:

```
if t_clock>t1:
        ________________
        Control code
        ________________
        t1 = t1 + t1_period
```

By inserting the control code within a conditional statement that checks if t_clock is greater than the time event t1, it is possible to adjust t1 using t1_period to ensure that the control code will run only every t1_period. Moreover, t1_period need not be a constant and can be a variable based on the control code.

A control function can have any number of TimeEvent variables. Every control function

has its own dictionary that stores the TimeEvent variables defined in the descriptor as follows:

```
{''t1'': 0.0, ''t2'': 0.0}
```

It is completely up to the user to generate values for the TimeEvent variables in any way desired. Since, there is a separate dictionary containing TimeEvent, it is possible to define identical TimeEvent variables for multiple control functions. However, it is very important for the user to update every TimeEvent in the control code. Failure to do so will result in the TimeEvent not changing. As will be described soon, this will cause the control code to execute at the rate of the fastest among the remaining control functions or the simulation time step, whichever is faster. Usually, neither of these options are what the user wishes for the time step of executing control functions. Therefore, it is recommended that the user make sure that any TimeEvent defined for a control function is updated correctly. The advantage of defining TimeEvents is that control functions can be executed at any arbitrary time instant in the simulation. As said before, not only can the time period of control functions be greater than the simulation time step but they can also be smaller. Before explaining this concept, the link between the control functions and the simulator will be described.

The question is how are the simulator circuit analysis blocks - loop analysis and nodal analysis - linked to the control functions? In the absence of control functions, the circuit analysis blocks would run at the simulation time step. When the simulation contains control functions, the circuit analysis blocks will run either at the simulation time step or when a control function generates an "event". A control function generates an event when any of its output ports changes. There is no threshold for change, if the value of the output port in the current execution of the control function differs from the value before the execution, an event is generated. When an event is generated by one of more control functions, the circuit analysis blocks are executed. To elaborate on how this is co-ordinated, let us consider the following example in Fig. 4.6. In this example, the simulation has three control files. In this section, the details of the control files are not important. The emphasis is on the time scheduling of these control functions. As shown, the first control function is called Reference Voltage Loop. This is the outermost control loop and is a slow control loop that has a time period of 1 millisecond. The second loop is the Voltage Control Loop which is a faster inner control loop with a time period of 100 microseconds. The third and the innermost control loop is the Pulse Width Modulator which has a time period of 100 nanoseconds. The simulation

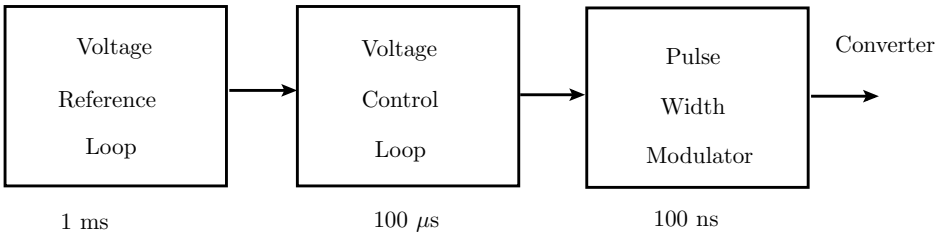

Figure 4.6: Nested control to describe timing

time step has been chosen to be 5 microseconds. This is a typical control layout for many power electronics simulations - 1 to 5 microseconds is a reasonably small time step for a power converter switched between 1 to 10 kHz. The voltage control can be at a time period much higher than the simulation time step. However, the Pulse Width Modulator is a hardware implementation detail. This is achieved using specialized hardware on microcontrollers or can be achieved using analog comparators. In either case, the resolution of the Pulse Width Modulator is very high and therefore the time period of this control block may have to reflect a hardware detail. For a 10 kHz converter, the switching time period is 100 microseconds, for which the Pulse Width Modulator time period of 100 nanoseconds results in 1000 samples in a switching period. In most cases, this results in sufficient accuracy for generating switching signals. The time period can be reduced further if the converter is a resonant converter or if the switching frequency can be higher.

The significance of the above example is that there are now four different time periods in the simulation - three from control functions and the fourth from the simulation time step. The time periods mentioned are just examples and can change. However, it is important to note that there are control time periods that are greater than the simulation time step and control time periods that are smaller than the simulation time step. To begin with, each of the control functions can be called Volref.py, Volcon.py and Pwm.py. Therefore, there are descriptors for each control function - Volref_desc.csv, Volcon_desc.csv and Pwm_desc.csv. The TimeEvent parameters in each control descriptor spreadsheet are provided separately in Tables 4.7, 4.8 and 4.9. The TimeEvent variables could have been declared to be the same as TimeEvents are local to a control function and are not global like the VariableStorage variables. However, for clarity in examining the time scheduling of these control functions, distinct variables are chosen. Each control function will have a separate TimeEvents dictionary:

Table 4.7: TimeEvent in Volref_desc.csv

TimeEvent	Desired variable name in control code = tvolref	First time event = 0.0

Table 4.8: TimeEvent in Volcon_desc.csv

TimeEvent	Desired variable name in control code = tvolcon	First time event = 0.0

Table 4.9: TimeEvent in pwm_desc.csv

TimeEvent	Desired variable name in control code = tpwm	First time event = 0.0

```
{''tvolref'': 0.0}
{''tvolcon'': 0.0}
{''tpwm'': 0.0}
```

Let us assume that the control files will update their TimeEvent variables according to the time periods discussed before. Therefore, the following statements will be present in the respective control files,

```
Volref.py :     tvolref = tvolref + 0.001
Volcon.py :     tvolcon = tvolcon + 100.0e-6
Pwm.py :        tpwm = tpwm + 100.0e-9
```

Refer to the conditional statement shown above where the time event was compared with the time instant of simulation t_clock, to know how the TimeEvent update takes place and control code can be executed once within the time period.

With the above background, let us examine the process of scheduling. The simulation creates a list called TimeVector. Let us start from a random time instant when the simulation execution runs completely. This implies circuit analysis - loop and nodal analysis as well as execution of all control functions. At this point of time how does the simulation proceed? When a control function is evaluated, it may so happen that the control code is not executed

due to time instant of simulation t_clock being less than the TimeEvent for that control function. For example, let us say that the current time instant of simulation t_clock is 80 microseconds while Volref.py has a TimeEvent of 1 millisecond. Since, t_clock<tvolref, the control code in Volref.py will not execute and execution will only take place when t_clock is 1 millisecond. Evaluation of all control functions, will result in values of their TimeEvents remaining unchanged or updated to new values according to their time periods. These values of TimeEvents are added to the TimeVector list. Therefore, after evaluating all the control functions, TimeVector will be:

```
TimeVector = [tvolref, tvolcon, tpwm]
```

The list contains the values of the TimeEvents. To this list is now added the simulation time instant which we called tode - meaning time instant of the Ordinary Differential Equation solver. So, TimeVector is:

```
TimeVector = [tvolref, tvolcon, tpwm, tode]
```

The above list is arranged in ascending order such that the smallest time value is the first element. This smallest value will now be the next instant of evaluation.

The word "evaluation" has been used instead of "execution" This is because the simulation now checks for another status. Did any of the control functions generate an event? As described before, a control function is said to generate an event when one or more of its outputs changes. There are no minimum or maximum requirements for a change in the output. Any change in the output, however small or large will trigger an event. If one or more control functions generate an event, the simulator will now go through the entire simulation cycle - loop and nodal analysis as well as control functions. This is because when a control function generates an event, this means an output has changed which in turn causes a change in the circuit and therefore the circuit needs to be solved again completely. To compare this with the other possibility, what happens when no control function generates an event because no control function experiences any change in any of its outputs? In this case, the simulator will only evaluate the control functions at the next time instant which is the smallest time instant in TimeVector. The following example will describe the difference between simulation evaluation and simulation execution. Let us consider the time instant of simulation marked as t in Fig. 4.7. As is evident, this time instant was chosen because tpwm

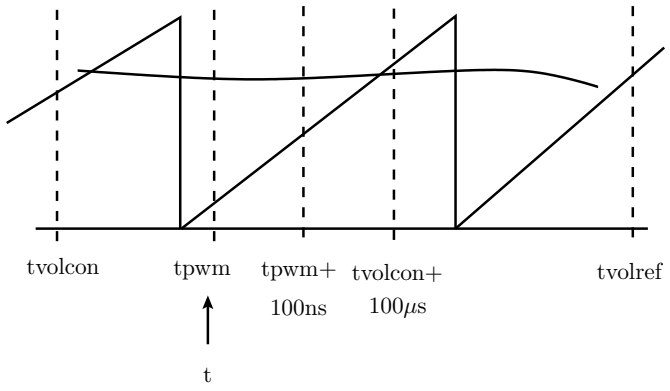

Figure 4.7: Timing diagram of control functions

was the smallest in TimeVector. TimeVector can be:

$$\text{TimeVector} = [\text{tpwm},\ \text{tvolcon}+100.0\text{e}{-6},\ \text{tvolref}]$$

Let us neglect the time instant from the simulation time step to focus on the effect of control functions. Let us suppose that at the instant t shown, none of the control functions generated an event. Only Pwm.py updated its TimeEvent from tpwm to tpwm+100.0e-9. The remaining TimeEvents in the other two control functions are the same and are shown in the figure. So TimeVector is:

$$\text{TimeVector} = [\text{tpwm}+100.0\text{e}{-9},\ \text{tvolcon}+100.0\text{e}{-6},\ \text{tvolref}]$$

Since, none of the control functions has generated an event, there is no point in running the circuit analysis. There has been no change in the circuit and the time difference between the tpwm and tpwm+100.0e-9 is 100 nanoseconds. This is much less than 5 microseconds which is the simulation time step. The simulation time step is chosen with respect to stability of the simulation. As will be described in Chapter 7, if the simulation time step is not small enough as compared to the time constant of the branches in the circuit, the simulation can become unstable. However, when the simulation updates the time instant from tpwm to tpwm+100.0e-9, the simulation is checking for control events that change the state of the circuit. Simulating at a potential time step of 100 nanoseconds when 5 microseconds has been found to be sufficient will increase the computational burden of the simulator significantly and slow it down. Therefore, if no event has been generated at t=tpwm, at the next time instant t=tpwm+100.0e-9, it is only necessary to evaluate the control functions. The simulator will evaluate all control functions, however, by using the conditional time check shown above, only

Pwm.py will be evaluated. Any control function may generate an event in any time instant of simulation or it may so happen that no control function may generate an event at all for a number of these time instant updates.

Let us examine two possibilities. The first, suppose Pwm.py generates an event. This could be a change in one of its outputs which may be the switching signal to the converter - turning on or off a switch. When an event occurs, the state of the circuit has changed. In this case, the resistance of a branch has changed and in this particular case, the resistance of a branch can change by a huge order - a few milliohms to several megaohms. For accurate simulation that resembles hardware, it is essential that the simulator now execute all the circuit analysis functions and update the currents and voltages and all other variables in component objects in the circuit. As before, all control functions are also executed and TimeVector is updated. The next time instant of simulation will be the smallest element of TimeVector. The second possibility is that the smallest time instant may be generated by Volcon.py i.e tvolcon shown in the figure above. As an example, let us assume in this case that Volcon.py does not produce an Output but only generates the duty cycle or modulation signal for the pulse width modulator. Therefore, Volcon.py does not affect a circuit component directly in the way Pwm.py does though it affects another control function which is Pwm.py. The duty cycle generated by Volcon.py is of the type VariableStorage as the duty cycle has to be accessed in Pwm.py. Details of the control will be provided in the next section. A change in one or more variables of the type VariableStorage will not generate an event in the simulator. This is because VariableStorage is used for interfacing between control functions and for writing control variables to the output data file. VariableStorage types will not directly impact the state of the circuit the way Output types do. Therefore, if tvolcon is the smallest in TimeVector, it is ensured that the simulator will evaluate the control function at that particular time instant. However, circuit analysis will not be performed without an event being generated.

In the above description, the time step of the simulator was left out to focus on the effect of TimeEvents generated by the control functions. However, even in the presence of control functions, the time step of the simulator can decide the next time instant of simulation. As stated before, the time instant of simulation is always added to TimeVector.

```
TimeVector = [tpwm, tvolcon+100.0e-6, tvolref, tode]
```

The smallest time instant in TimeVector will be the next simulation time instant. In case tode is the smallest, the simulator will execute the entire cycle even if no event is generated by a control function. This is for two reasons. In case there are no control functions, TimeVector will only contain tode and therefore, the time step of simulation alone will decide the next instant of simulation. Therefore, the simulator must execute all circuit analysis functions as this will be a case of a fixed time step circuit simulation which is quite often the case when simulating passive circuits. The second reason is that when tode is the smallest time instant in TimeVector even when control functions are present, it may be possible that the control functions have significantly larger time periods than the simulation time step. In this case, the simulator must execute all circuit analysis functions when tode is found to be the smallest as failure to do will result in the simulation being unstable. As stated below a simulation at every simulation time step is a necessity for stable simulation - that is how the simulation time step is chosen. However, when tode is found to be the smallest time instant in TimeVector, it is necessary to update tode with the simulation time step. So,

```
tode = tode + dtode
```

Where dtode is the simulation time step specified by the user in circuit_inputs.csv.

4.5 Interfacing control code

The previous section described how the control functions are scheduled and how the circuit simulator interfaces the control functions with the circuit analysis functions. However, the control functions shown in the previous example were treated as black boxes with an emphasis only on how the TimeEvents are generated. In this section, it will be described how the control functions are interfaced. Detailed control algorithms will not be presented. The focus will be on showing cascaded control can be designed with the circuit simulator.

Let us start with the design from the inner most control function to the outermost. Therefore, first Pwm.py. The parameters in Pwm_desc.csv will be as listed in Table 4.10. Let us assume there is only one Switch called S1 in the converter - for example, a simple buck converter. This Switch_S1 has a control tag named as S1_gate in the circuit parameters spreadsheet. This control tag is accessed by S1_gate in Pwm.py. Since, a pulse width modulator is being programmed, a carrier wave will need to be generated. A StaticVariable called

carr_signal is defined for this purpose. The duty cycle will be the output of the controller which regulates the voltage of the buck converter and this code is found in Volcon.py. The duty cycle is defined as a VariableStorage type as it is an input from another control function. This could have been defined in the descriptor spreadsheet of Volcon.py but let us define all the variables of Pwm.py first. Lastly, the TimeEvent tpwm as already described in the previous section.

The code in Pwm.py will be similar to this without going into the details:

```
if t_clock>tpwm:
        carr_signal += (1/5000)*100.0e-9
        if (carr_signal>1):
                carr_signal = 0
        if (duty_cycle>carr_signal):
                S1_gate = 1
        else:
                S1_gate = 0
        tpwm += 100.0e-9
```

A brief description of the code is as follows. The block of code executes only when the time instant of simulation t_clock is greater than TimeEvent tpwm. The carrier waveform is a saw tooth waveform of unity magnitude and of frequency 5 kHz as shown in the previous section. Therefore, (1/5000) is the slope of the waveform. This needs to be multiplied by the time period of the modulator i.e. 100 nanoseconds and finally needs to be limited to unity. The next step is the comparison between the duty_cyle and the carr_signal and this generates the output S1_gate which is connected by the simulator to the control tag S1_gate of Switch_S1. Finally, the TimeEvent tpwm is updated by 100 nanoseconds. A few things to note. The input is duty_cycle. Since this variable is of type VariableStorage, it can be accessed in any control function. The next control function will describe how this duty_cycle is produced. The carr_signal variable is a StaticVariable and therefore it can be directly accessed within Pwm.py and is updated by a += since the stored value is made available in the function by the simulator. Similarly, the updated value is extracted by the simulator and stored in a dictionary for the next iteration.

Now, the next level of control - Volcon.py. This function contains the controller which we

Table 4.10: Pwm_desc.csv

Output	Element name in circuit spreadsheet = Switch_S1	Control tag defined in parameters spreadsheet = S1_gate	Desired variable name in control code = S1_gate	Initial output value = 0
StaticVariable	Desired variable name in control code = carr_signal	Initial value of variable = 0.0		
TimeEvent	Desired variable name in control code = tpwm	First time event = 0.0		
VariableStorage	Desired variable name in control code = duty_cycle	Initial value of variable = 0.0	Plot variable in output file = yes	

have chosen to be a Proportional Integral (PI) controller. This control generates the duty_cyle variable used in Pwm.py. The parameters in Volcon_desc.csv are listed in Table. 4.11.

```
if t_clock>tvolcon:
        volt_error = volt_ref - volt_output
        volt_integral += volt_error*100.0e-6
        duty_cycle = 0.001*volt_error + 0.01*volt_integral
        if duty_cycle > 0.98:
                duty_cycle = 0.98
        tvolcon += 100.0e-6
```

The purpose of the control is to regulate the output voltage to the reference voltage volt_ref. This reference voltage is generated by the outer control Volref.py and described next. 0.001 is the proportional gain while 0.01 is the integral gain. These values are purely arbitrary and just examples to show how to write control code. This control function contains the PI controller which regulates the output voltage and therefore takes the measured voltage as an Input. It

Table 4.11: Volcon_desc.csv

Input	Element name in circuit spreadsheet = Voltmeter_Voutput	Desired variable name in control code = volt_output	
StaticVariable	Desired variable name in control code = volt_error	Initial value of variable = 0.0	
StaticVariable	Desired variable name in control code = volt_integral	Initial value of variable = 0.0	
TimeEvent	Desired variable name in control code = tvolcon	First time event = 0.0	
VariableStorage	Desired variable name in control code = volt_ref	Initial value of variable = 0.0	Plot variable in output file = yes

is assumed that the circuit schematic contains a Voltmeter called Voltmeter_Voutput. The measured voltage is made available in the control function by the simulator as the variable volt_output. The error in the voltage and the integral of the error are defined as StaticVariables. The variable volt_error does not have to be defined as StaticVariable as it is calculated within the control code and used immediately. However, it is a safe practice to define as many variables as StaticVariables rather than using the default Python objects to ensure control can be tracked. To elaborate on this, if volt_error was not a StaticVariable, it would not exist if the code block was not executed. Therefore, if for control debugging, the variable is accessed outside this block, the simulator would exit with an error as it is accessing a variable which does not exist. However, when a variable is defined as a StaticVariable, it always has a value even if that value is the initial value specified in the descriptor spreadsheet. The duty_cycle is generated according to PI control. As mentioned before, the duty_cycle is made available in all control functions once it has been defined in one of them - in this case Pwm_desc.csv. Also, duty_cycle does not have to be defined as it is already defined in Pwm_desc.csv and to

redefine it is a violation for which the simulator will abort with an error.

Finally, the outermost control function Volref.py. This function will generate the voltage reference for the previous control function Volcon.py. Volcon_desc.py contained the definition of volt_ref as VariableStorage. The parameters in Volref_desc.csv are listed in Table. 4.12.

```
if t_clock>tvolref:
        volt_ref += 0.01
        if volt_ref>volt_setpoint:
                volt_ref = volt_setpoint
        tvolref += 0.001
```

Table 4.12: Volref_desc.csv

StaticVariable	Desired variable name in control code = volt_setpoint	Initial value of variable = 100.0
TimeEvent	Desired variable name in control code = tvolref	First time event = 0.0

This control function generates the reference voltage to be used by Volcon.py as a gradual ramp which is clamped at volt_setpoint. volt_setpoint is defined a StaticVariable with the initial value as 100. Therefore, Volref.py ensures that buck converter starts up gradually with a steadily increasing reference voltage.

This section has described how three control functions in a simulation can be interfaced with VariableStorage and how these variables can be accessed in each control function. It should be noted that VariableStorage can be defined in any control function descriptor. This example was fairly simple and the purpose was to provide an introduction to writing control functions. A far more elaborate example will be provided in the next chapter.

4.6 Conclusions

Since this circuit simulator is targeted towards power electronics applications particularly with multiple converters, user-defined control functions are an extremely critical component.

It is extremely important that a user be able to integrate control into a simulation with the same ease as in a commercial simulator. This circuit simulator allows a user to define multiple control functions with no limits on the number of files. By providing the facility of Static-Variables, the user can implement higher order control functions with complex mathematical calculations. In order to develop control in a modular manner, the VariableStorage type has been provided for the user to be able to break up their control algorithms into modules and interface those modules. The only visible disadvantage is that all the control functions have to be developed with code unlike a commercial software where control functions can be developed by connecting blocks from an in-built library. However, for most complex circuits that are intended for hardware implementation, control functions are developed as code to be programmed in microcontrollers or other forms of embedded controllers.

In a complex circuit with multiple converters, control algorithms can be fairly complex. One of the major challenges with control functions is the time of execution. As an example, in hardware, a control function may be an interrupt service routine connected to a timer. The timer may be configured to generate an interrupt at a constant time period or the time period may be calculated and loaded into the timer after every iteration. The interrupt service routine ensures that the control function will be executed at the desired time instant. For the simulation to match the hardware implementation, it is essential that the simulator provides a guarantee that every control function will execute at the exact time instant that is desired. Moreover, a user should not have to adjust the simulation time step in order to ensure that it matches the control function time step. In this circuit simulator, a time scheduler ensures that all control functions are evaluated at the desired time instant by providing the variable TimeEvent. The user can specify a TimeEvent for every control function and update this TimeEvent with any time period that are not necessarily multiples of each other or even multiples of the simulation time step. In this manner, the simulator provides accurate time resolution comparable to hardware implementation in a form that is convenient to the user.

This chapter and the previous chapter have described two aspects of the user interface. The first being how the circuit is represented and how the parameters of the circuit components are updated. The second being how control functions are implemented. Chapter 5 will describe in detail how a circuit can be simulated. Moreover, Chapter 5 will provide the user with an example on how a circuit can be developed in stages and how control can be

developed and tested in a modular manner. The circuit chosen in Chapter 5 is such that all the functions available with the simulator are used in simulating it. Chapter 5 will show how the interface provided in this circuit simulator is sufficient to design fairly complicated power electronic circuits. Moreover, writing control functions with the user interface described in this chapter provides a powerful platform which is also simple to use at the same time. The reader is encouraged to switch between Chapter 3, Chapter 4 and Chapter 5 while reading Chapter 5.