BDEDIT TECHNICAL REPORT

Contact Details

Developers
Samara
Email: samara.barwick@connect.qut.edu.au
Rory
Email: rory.higgins@connect.qut.edu.au
John
Email: john.wishart@connect.qut.edu.au
Daniel
Email: daniel.petkov@connect.qut.edu.au

Supervisor
Professor Peter Corke
GitHub: https://github.com/petercorke

Table of Contents

1. Context

In engineering, complex systems are often represented with block diagrams (refer to *Figure 1.1*), where blocks represent functions with inputs and outputs, and wires represent the flow of values between the ports of these functions.

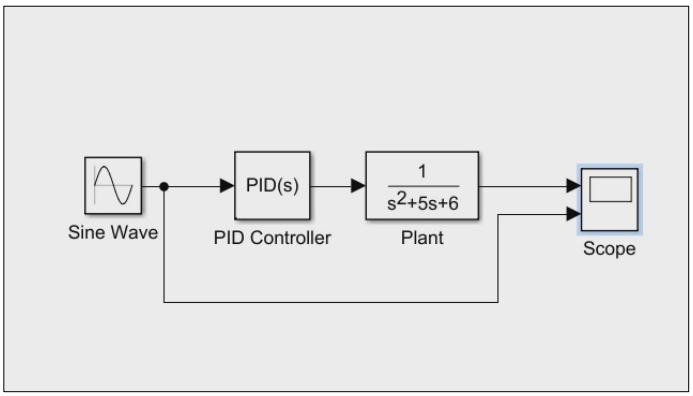


Figure 1.1 - Example of System as a Block Diagram

These block diagrams can be modelled and simulated as code through the bdsim[1] Python package developed by Professor Peter Corke, where the blocks and wires are represented in terms of Python class and method calls.

To aid with the conceptualization of the developed block diagram model and its modelling process, the bdedit package was developed as an addition to bdsim, allowing for block diagrams to be created graphically with items that visually represent the blocks, in/out ports and the wires (refer to *Figure 1.2*). Bdedit supports the saving and loading of these diagrams to and from a JSON file, which stores all the necessary data for the diagram to later be simulated through bdsim.

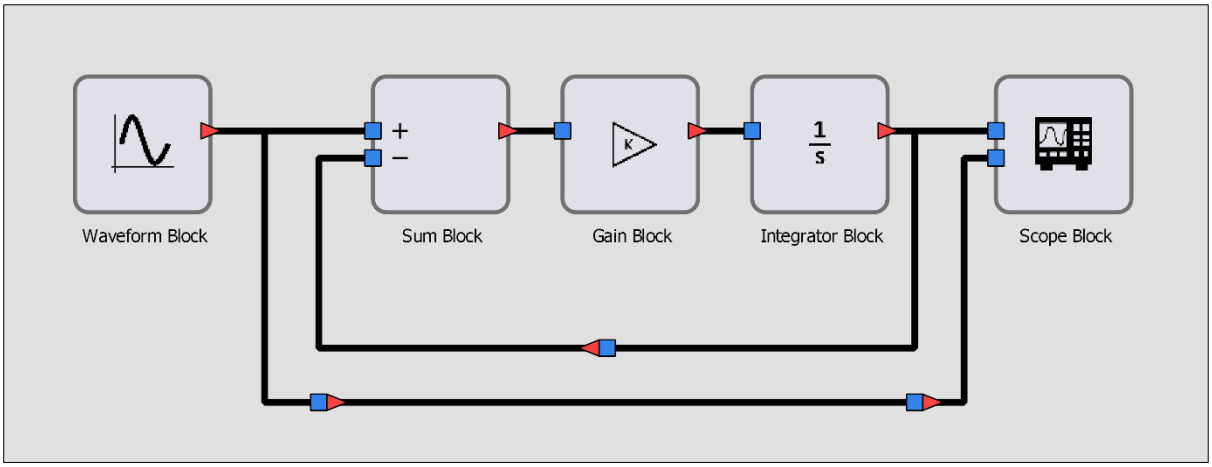


Figure 1.2 - Example of Block Diagram as represented in bdedit

[1] https://github.com/petercorke/bdsim

2. Feature Exploration of Bdedit

Interface

Installing the bdsim package and its necessary files, then running the bdedit.py[2] file, launches a new window containing a graphical user interface (refer to *Figure 2.1*). This interface contains three areas of focus, the canvas (grey grid space), the library browser panel, and the toolbar.

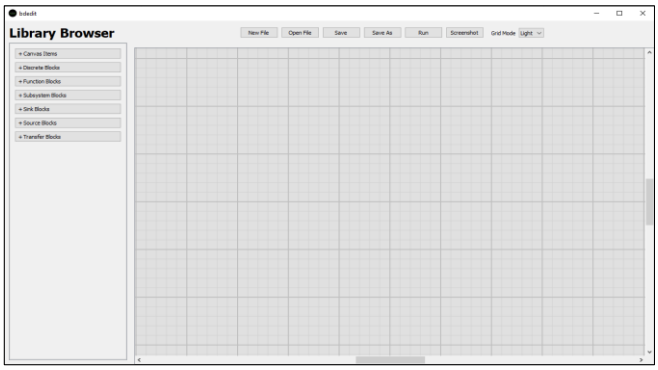


Figure 2.1 - Bdedit Graphical User Interface

Adding Blocks

Through this interface, the user can create a block diagram by choosing from a list of available blocks found within the Library Browser panel. These will call on the classes related to those blocks, to create a block which both stores its values internally within the program and graphically represents that block within the diagram. The graphical information of these blocks and wires is then stored within the canvas area (refer to *Figure 2.2*).

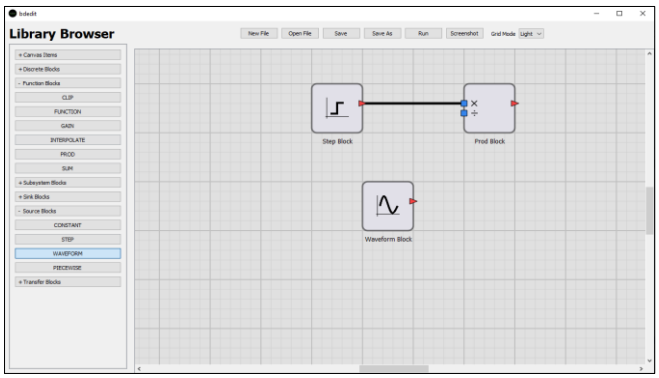


Figure 2.2 - Adding Blocks to Canvas and Connecting them

Sockets

These blocks have sockets, representing its inputs and outputs. These are determined through the block type, with some blocks only having input sockets (*sink blocks or INPORT blocks*), some only having output sockets (*source blocks or OUTPORT blocks*), and others having both input and output sockets (*function, transfer, discrete or SUBSYSTEM blocks*) (refer to *Figure 2.3*).

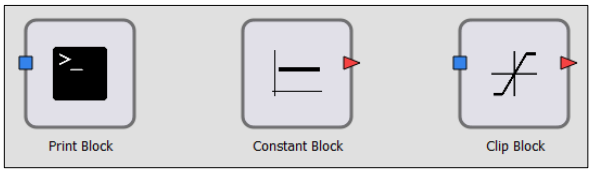


Figure 2.3 - Examples of Blocks;
Sink (left), Source (middle), Function (right)

These sockets can be used to connect the output of one block to one or more other blocks, creating a flow of data. The following logic is applied to these sockets based on their types:

- Sockets cannot connect to the same socket type (Output cannot connect to Output, Input cannot connect to Input)
- Input sockets can only have one wire connecting into them (any further wires that are connected will be disconnected until the existing wire is removed)
- Output sockets can have any number of wires connected to them

Socket Flipping

Blocks can also be flipped, reversing the sides on which the input and output sockets are located. This can be achieved through pressing the 'F' key (F for flip). These are only graphically updated and do not impact the flow of logic to those sockets (refer to *Figure 2.4*).

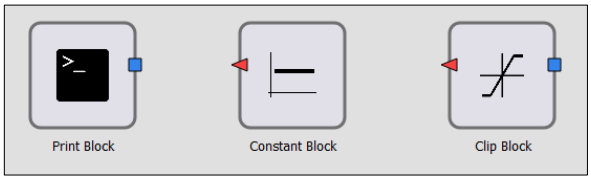


Figure 2.4 - Example of Flipping Socket Orientation

[2] File located at path: https://github.com/petercorke/bdsim/bdsim/bin/bdedit.py

Further Block Manipulation

Blocks can also be selected/ moved and deleted as desired. All items are restriced to being moved around within the borders of the canvas. Wires can also be selected and deleted, but not moved. Sockets cannot be selected, moved or deleted through mouse interaction. Selecting, or rather clicking on, a socket creates a draggable wire. As the block is moved, so to are its sockets. The only instance where sockets move relative to the block, is when the number of input or output sockets changes (which is controlled through the parameter window).

The selection of an item is indicated by a colour change. When a block, connector block or wire is selected, its outline changes from a thin black line, to a thicker bright orange outline (refer to *Figure 2.5*).

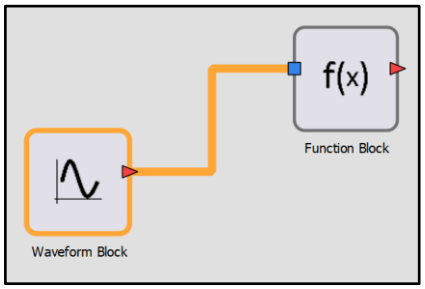


Figure 2.5 - Block and Wire Selected

As mentioned above, blocks and wires can be deleted when desired. This is done through first selecting the item, then pressing either the 'Backspace' or the 'Del' keys on the keyboard. If a wire is deleted, only the wire will be removed. If a block is deleted, any wires that were connected to it will also be deleted with the block.

Wires

Blocks can be connected to one another by either clicking on a socket of one block, then clicking on a socket of another block, or by click-and-dragging from one socket and releasing over the socket of another block (refer to *Figure 2.6*). When a wire is connected between two points, wire logic will be applied to it, in order to determine the path it should take to connect those two blocks. Moving the block around once the wire is connected, will update the position of the wire end points, and as such, will cycle through the wire logic to determine what path the wire should follow (refer to APPENDIX B for examples and a walkthrough of this logic).

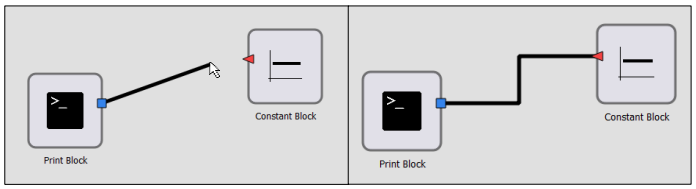


Figure 2.6 - Example of Dragging Wire
Pulling Wire from Input Socket to Output Socket (left),
Wire after it has been connected (right)

Connector Block

The sections of the wires cannot be pulled out and positioned to the user's liking, as their path is solely dependent on the position of the two points it connects. Hence, to assist with routing the wires when the diagrams become more complex with wires travelling in multiple directions, a connector block can be used as an intermediary point through which the wire must travel. These provide the user with more control over how the wire travels between any two points, by creating more points in between the start and end point, which the wire must first connect. The comparison between using a connector block and not, can be seen in *Figure 2.7*. The connect block appears as a single Input and Output socket joined together on one edge; and similarly to other blocks, it is also flip-able.

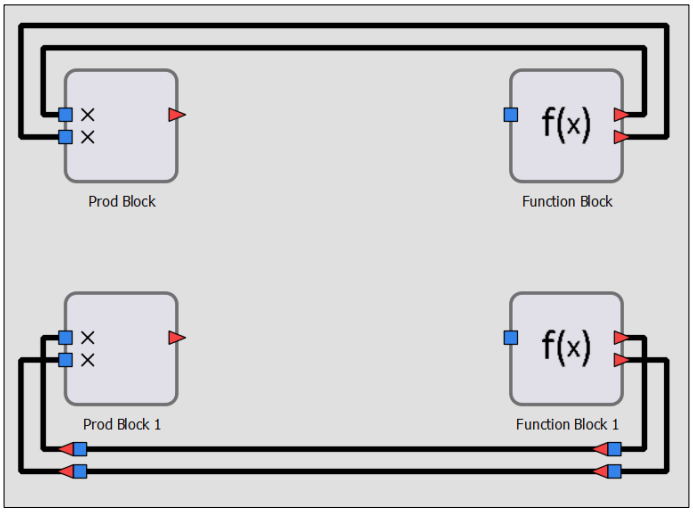


Figure 2.7 - Usage of Connector Blocks
Default wire wrapping logic (top),
User controlled wire wrapping; through using connector blocks (bottom)

Intersection Management

As seen in the bottom part of *Figure 2.7*, wires can overlap at times, and although they are fairly easy to follow in this figure, it can become difficult to follow the flow of logic when the diagram becomes more complex. To address this problem area, parts of wires that cross over each other can be separated to indicate they do not cross, but instead pass over each other (refer to *Figure 2.8*). This feature might not always be desired, so it has been disabled by default, however can be toggled on or off by the user through pressing the 'I' key (I for intersection). The wire logic at these intersection points, keeps all vertical segments of wires solid, and erases parts of any horizontal segments that they pass through.

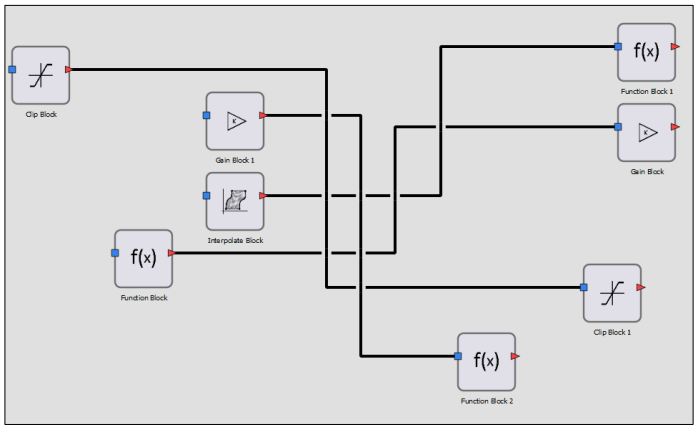


Figure 2.8 - Toggling Wire Intersection Detection ON

Editing Block Parameters

Integral to the creation and simulation of block diagrams, is the ability to edit the parameter values related to each block, as this dictates what output value a block produces and how blocks process any inputs that feed into it to produce an output. If supported for the given block, it is also possible to edit the number of input or output sockets a block has. All blocks (aside from connector blocks) can be named as desired by the user. As blocks are spawned they are given a default name which is auto incremented depending on the block type.

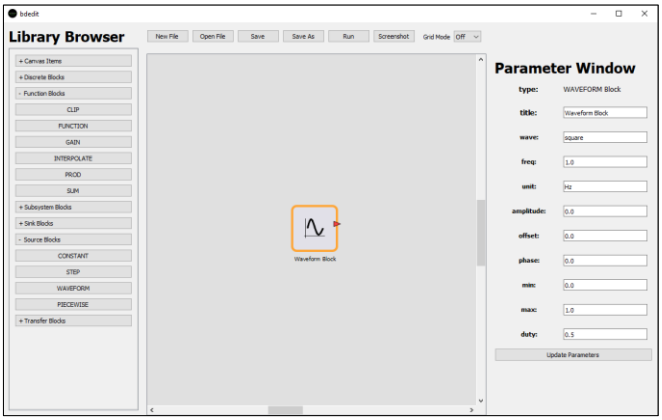


Figure 2.9 - Parameter Window

These user-editable block parameters are editable through a Parameter Window panel that appears on the right hand side of the screen when triggered to do so (refer to *Figure 2.9*). It can be toggled to open by first selecting a block, then right clicking the mouse; subsequently, it can be closed at any moment by either right clicking again or left clicking anywhere in the screen. Closing the Parameter Window will retain any values that have been edited, but will only update the block parameters once the 'Updated Parameters' button has been clicked.

When parameters are edited and the user selects update, all editable values within the Parameter Window will be checked to ensure they adhere to the conditions placed on them. If the block title is changed, this will be checked against other existing block titles, to ensure the name of the current block would not be a duplicate. If the block parameters are changed, they will be compared against their required types to ensure they match (e.g. float, int, bool, list, str), and if any further restrictions are placed on these parameters (e.g. matching to certain strings, or being within a certain range), they will be checked against those too. This information is defined internally within the block class, and applied to the block when it is created.

User feedback pop-up windows are also connected to this Parameter Window. If the user provides values that are incorrect - be they a duplicate block title, incorrect types or not adhering to further restrictions – the block parameters will not be updated, and an error message will be displayed notifying the user with useful information for where the issue occurred. If all values are correctly inputted, a success message will be displayed in the same area (refer to *Figure 2.10*).

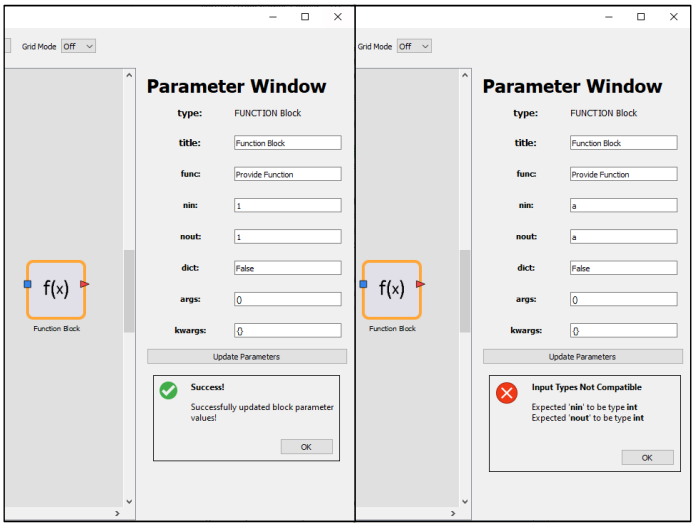


Figure 2.10 - User-Feedback Messages
Success Message (left), Error Message (right)

Screenshot

Upon creating a block diagram, the user can take a screenshot of all items within the canvas by simply pressing the 'Screenshot' button located within the toolbar. This will save a 4k resolution image of the entire canvas and everything in it. This resolution is chosen due to canvas size potentially being 5x the desktop screen size due to the zooming feature. Due to limited time to further develop this feature, at present time, this image will be saved with the '.png' file extension, under the name 'Scene Picture' in the same folder the interface is run from. **If taking multiple screenshots this way, be aware that this will override any previous screenshot you may have taken.**

Grid Mode

To improve the viewing quality of the screenshots taken, and reduce the amount of visual noise/messiness created by having the background be a grid, an option is available from the toolbar to disable the background by navigating to the 'Grid Mode" button in the toolbar, and selecting 'Off' from the drop down menu. Alternatively, the grid can be displayed in two other modes, Light (the default mode) and Dark (refer to *Figure 2.11*).

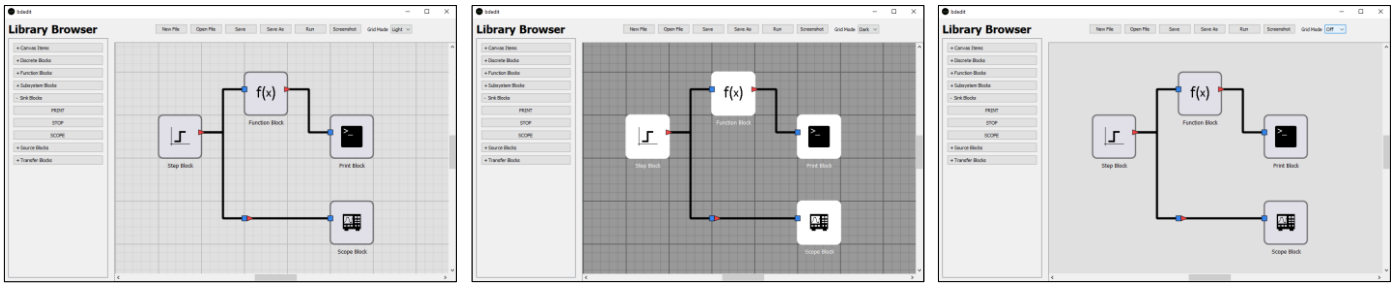


Figure 2.11 - Background Colour Modes
Light (left), Dark (middle), Off (right)

Grid Snapping

To improve the usability of the interface and the user experience when moving/aligning blocks within the canvas, a grid snapping feature has been implemented, where movement of the mouse will be restricted to moving the block in increments of 20 pixels (the width of the smaller grid squares). Additionally, all sockets are indexed in increments of the same value (20 pixels) in order to line up with these smaller grid lines. As such, since wires are automatically drawn between the socket positions, it is much easier to move blocks around in order to make them straight.

Saving and Loading

An integral component to this bdedit tool is the support for saving the current progress made on a block diagram, and the support to load a previously worked on block diagram (provided it is in a compatible format). Block diagrams are saved as JSON files, containing information about the canvas size, all the blocks (its name, on-screen position, parameters, and sockets), and finally all the wires and the sockets they connect to. This information is stored as a dictionary within the JSON file, which is parse-able as key-value pairs, where the key represents the name of the variable or parameter related to the scene, block, socket or wire, and the value representing the value that variable holds.

A file can be saved or loaded from through the associated 'Save' or 'Save As' and 'Load' buttons within the toolbar. Upon clicking on one of these buttons, the a file browser window will pop up (allowing the user to browse their devices' file structure), prompting the user to either select a file to load or to choose the location they wish for their file to be saved (and the name of the file if it's the first time saving or if saving the diagram as a new file).

Due to a limitation of time to further develop this feature, whether this file is in a compatible format is not checked before it is attempted to be parsed, so any errors that may occur due to an incorrect file being loaded, will result in the crashing of the bdedit tool.

3. Class Architecture (High Level)

The architecture of BDEdit can be summarize through the connectivity between 6 main classes as seen in Figure 3.1. Other classes are also essential, however it is through the interactions between these 6 main classes that the application is able to run. For a full size image of this architecture refer to APPENDIX A. These classes break down into the following:

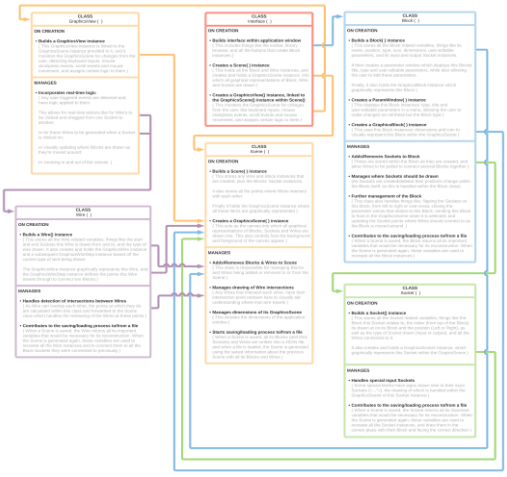


Figure 3.1 - BDEdit Architecture Diagram

1) The Interface Class – This class is responsible for the dimensions of the BDEdit window that appears when the application is run, as well as managing the layout of where all the interact-able areas of the application are, these being:

- The toolbar;
- The left side panel (otherwise named the Library Browser);
- The right side panel (otherwise named the Parameter Window); and
- The canvas or work-area (which is an instance of the GraphicsScene class, who itself is a child class of the Scene class, AND is connected to a GraphicsView class instance).

It is namely the creation of the GraphicsScene class that allows for the graphical representation of Blocks, Sockets and Wires, the culmination of which allows for the making of a Block Diagram.

2) The Scene Class – This class is responsible for three things.

- Storing all instances of any Wires and Blocks (and their Sockets) that are created, i.e. their structures, internal variables, lists, properties, etc. It also manages the adding/removing of these instances.
- Creating and storing a GraphicsScene class instance, in which all the GraphicsBlock, GraphicsSocket, and GraphicsWire class instances are added to graphically represent their class respectively. This allows for the all the relevant internal information of the Block, Socket and Wire classes to be represented graphically!
- Managing the intersection points at which any two (or more) wires overlap.

3) The GraphicsView Class – This class instance is connected to the GraphicsScene through the Interface class, and is responsible for monitoring the GraphicsScene and implementing logic to any user interactions within it. These interactions being: key presses, mouse press/release, mouse movement, scroll click and scroll movement events. These detected events are caught in real time, allowing for logic within the GraphicsScene to also be updated in real time. Some examples of this includes:

- the creation of a Wire (and subsequent GraphicsWire) when a GraphicsSocket is clicked on;
- the real-time updates to how the GraphicsWire is drawn while being pulled from one GraphicsSocket to another;
- the real-time updates to how the GraphicsBlock is drawn when the number of sockets on it changes, or when it is selected;
- the zooming in and out of, and panning of the canvas (GraphicsScene).

4) The Block Class – An instance of this class is created when a respective button is clicked from the Library Browser side panel in the Interface. This class is responsible for holding all Block related variables, these beings things like its name, position within the canvas, block type, icon, dimensions, user-editable parameters and a list of input and output Socket instances that are related to this Block. Additionally, this class relates and instance of the **ParamWindow** class and GraphicsBlock class to this Block.

The **ParamWindow** is a class which creates a Parameter Window in which are displayed this Blocks' type, title and user-editable parameters, and through which a user can edit the parameters of a given block. When this Parameter Window is opened, it appears in a right side panel within the Interface.

Similar to how the GraphicsScene represents a Scene class instance, the GraphicsBlock graphically represents a given Block, and sends that graphical information to the GraphicsScene to display.

5) The Socket Class – An instance of this class is created and connected to a Block, whenever one is made. This class is responsible for holding all Socket related variables, like its index (from the top of the Block, these are auto incremented as more Socket are made for a Block), the position to be drawn at (Left or Right of the Block), the type of Socket being drawn (Input or Output) and finally a list of all Wires that are connected to this Socket.

Additionally, some special blocks (PROD and SUM Blocks) have math operators (+,–,×,÷) drawn alongside their input Sockets depending on what string for one of those block's parameters. For example, that parameter may be the string "*/*" in the PROD block, and this will draw a '×', '÷', '×' alongside the first, second, third input sockets respectively.

The Socket class also holds an instance of GraphicsSocket which graphically represents the Socket, and is sent to the GraphicsScene to be drawn.

6) The Wire Class – As was mentioned in the GraphicsView, when it has detected that a GraphicsSocket has been clicked, this will create a Wire instance from that Socket, and a subsequent GraphicsWire from the GraphicsSocket to the mouse cursor, until either the GraphicsWire is clicked off of into an empty space within the GraphicsScene, or the wire is clicked off of onto another GraphicsSocket. If these socket types are different (i.e. both aren't input or output sockets), then the wire is connected and will remain so, even as the GraphicsBlock is moved around. The Wire class is responsible for holding all Wire related variables, these being what Sockets this Wire connects (start/end sockets) and the type of wire being drawn (Direct, Bezier or Step). The wire type dictates the style with which the wire is drawn. Direct draws the wire as a straight line between two points. Bezier draws the wire as a cubic between two points (think sinusoidal wave). Step draws a wire with 90 degree bends at each point the wire must turn to reach the end socket.

Although other methods and classes are involved in the process of making this application meet further functional requirements, these 6 classes are what tie everything together.

4. Making changes to code

Adding more blocks types to application

If the new block type falls under one of the following, already existing categories: Source, Sink, Function, Transfer, Discrete, INPORT, OUTPORT or SUBSYSTEM block (these last three are located within the hierarchy file), then that block simply needs to be added as a class of one of the Python files relating to those block types. These Python files are located in the "***bdsim/bdsim/bdedit/Block_Classes***" folder (refer to Figure 4.1), with the names seen in Figure 4.2.

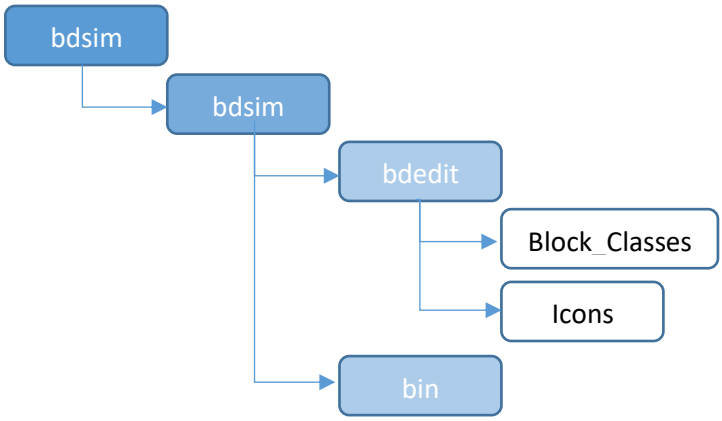


Figure 4.1 - Relevant bdsim File Structure

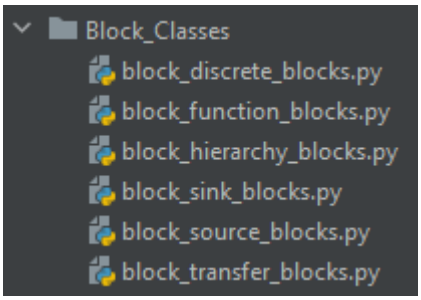


Figure 4.2 - Block Type Files

Within each of those files, the first few lines import the block type they inherit, for example, the "***block_function_blocks.py***" file imports the FunctionBlock class from the "***block.py***" file (located in "***bdsim/bdsim/bdedit***") inheriting its properties. An example of one of the blocks within these files (without the block comments) is given below:

```
class Function(FunctionBlock):
    def __init__(self, scene, window, func="Provide Function", nin=1,
nout=1, dictionary=False, args=(), kwargs={}, name="Function Block",
pos=(0, 0)):
        super().__init__(scene, window, name, pos)

        self.setDefaultTitle(name)

        self.block_type = blockname(self.__class__)

        self.parameters = [
            ["func", str, func, []],
            ["nin", int, nin, [["range", [0, 1000]]]],
            ["nout", int, nout, [["range", [0, 1000]]]],
            ["dict", bool, dictionary, []],
            ["args", tuple, args, []],
            ["kwargs", dict, kwargs, []]
        ]

        self.inputsNum = nin
        self.outputsNum = nout

        self.icon = ":/Icons_Reference/Icons/function.png"
        self.width = 100
        self.height = 100

        self._createBlock(self.inputsNum, self.outputsNum)
```

This block class type is constructed based on the definition of the corresponding bdsim block[3]. When adding in a new block, the following template can be copied and adjusted as needed.

```
# The name of this new block class should be unique, and
# it should inherit whichever block type it belongs to.
class My_New_Block(Class_the_new_block_relates_to):

    # Next, the scene and window are required to be passed to the
    # creation of this block (and will be done so automatically from
    # the interface). The following input parameters are the parameters
    # of the block and their respective default values.
    # Finally comes the block's default name (if the user doesn't provide
    # one), and the position is set to always spawn the block at the
    # centre of the work area.
    def __init__(self, scene, window, param1="parm1_default_value",
param2="parm2_default_value",
                 param3="parm3_default_value", name="My_New_Block Block",
pos=(0, 0)):
        super().__init__(scene, window, name, pos)

        # The chosen default name for this block is passed to
        # the setDefaultTitle function which will ensure no duplicate
        # names names of this block exist.
        self.setDefaultTitle(name)

        # The block type is set as the given name of this block class
        self.block_type = blockname(self.__class__)

        # The parameters of the block are wrapped into a list, where each
        # parameter sits inside its own list and defines
        # the name, type, default value, and any further restrictions
        # for each respective parameter.
        self.parameters = [
            ["parameter 1", str, param1, []],
            ["parameter 2", str, param2, []],
            ["parameter 3", str, param3, []]
        ]

        # The icon file path is matched to whatever name the icon for
        # this block was named within the Icons folder (this procedure
        # will require the icons resource file to be updated for a new
        # image to be findable within this folder).
        self.icon = ":/Icons_Reference/Icons/my_new_block.png"

        # The height and width are set for this block
        self.width = 100
        self.height = 100

        # Finally the block is created, with the number of input and
        # output sockets that have been assigned to this block. (This
        # will be inherited from the class this block inherits).
        self._createBlock(self.inputsNum, self.outputsNum)
```

This simply needs to be added to the end of the appropriate file (as chosen from Figure 4.2), and this will make the block automatically appear within the interface.

[3] https://petercorke.github.io/bdsim/bdsim.blocks.html?highlight=Function#bdsim.blocks.functions.Function

Block parameters explained

Each block has its own unique parameters, with their own unique names, types, default values and further restrictions (like being restricted to a certain range of allowable numbers). All of a given blocks' parameters are stored within the self.parameters variable list, with each individual parameter being stored as a list within the self.parameters list. Each individual parameter is defined with the following format:

- ***parameter = ["name", type, value, [restrictions]]***
  e.g. parameter = [["Gain", float, gain, []], ["Premul", bool, premul, []]]

The items which make up this list of the parameter, are as follows:

- ***name**: this is the name of the parameter as a string*
- ***type**: this is the type this parameter must be (e.g. int, str, float)*
- ***value**: this is the default value the parameter will be set to, if no other value is given. It must also adhere to the required type of the parameter.*
- ***restrictions**: this is a list (can be list of lists) containing further restrictions applied to the parameter.*

As multiple restrictions can be applied to a single parameter, each individual restriction is enclosed as a list. If no restrictions are applied to a parameter, the main restrictions list (inside the parameter list) will simply be an empty list, as seen in the example. These restrictions follow the following structure:

- ***restriction = ["restriction name", [condition(s)]]***

What these two items within the restriction list represent is explained below:

- ***restriction name**: can be only one of the following "keywords", "range", "type" or "signs".*
- ***condition(s):** differ based on the restriction name used, and will be of the following format:*

Currently, only the four restriction types mentioned below are recognized. These must be entered as a string in the first item within the restriction list ("restriction name"), to indicate what kind of restriction is being applied to this parameter. Following this first value, a list containing one or more conditions placed on the restriction is defined. These depend on the type of restriction chosen. Examples of these restrictions are given below:

- `[["keywords", ["sine", "square", "triangle"]]]`
  - This restriction compares the parameter value against the strings defined within the conditions list. This restriction should only be used on parameters whose required type is 'str' (string). If the parameter value doesn't match any of these strings, this will throw an error notifying the user that their input must match one of those strings.
  - Here the conditions list is just a list of all the variations the parameter value can be.
- `[["range", [-math.inf, math.inf]]]`
  - This restriction compares the parameter value against being within a range of given numbers, defined by a min and a max. This restriction should only be used on parameters whose required type is either 'float' or 'int'. If the parameter value is outside the allowable range, this will throw an error notifying the user that their input must be within the given range.
  - Here the conditions list simply is made up of a minimum and maximum value.

- `[["type", [type(None), int, float]]]`
    - This restriction compares the type of the parameter value against one of the additional allowable types for this parameter. If for instance, a parameter can either an integer or a float when defined, or a None type otherwise, this means the parameter can be one of 3 different types, and this restriction allows the parameter to pass as long as its type matches one of the defined types. In order to allow other types, the type set for this parameter (as the second item in the parameters list), must also be included in this conditions list, usually as the last item (for consistency sake). Note, when allowing a parameter to have None as a value, since None isn't a type, but rather a NoneType, type(None) is used to extract the type of None. If the type of the parameter value doesn't match one of the other allowable types, this will notify the user of the error, and the allowable types.
    - Here the conditions list is just a list of all the acceptable types for that parameter, with the required type of the parameter also being in that list (usually as the last value in the list; which is float in this case).
- `[["signs", ["*", "/"]]]` or `[["signs", ["+", "-"]]]`
    - This restriction compares the parameter value against the various characters that have been defined in the conditions list. This restriction should only be used for SUM and PROD type blocks, as these have input sockets which are labelled according to the signs defined in this condition list. The parameter value for these blocks (SUM and PROD) is entered as a string of characters, and each character for the respective block is checked against the ones that are allowed from the conditions list. If that string is made up of any number of characters that don't match the ones that are allowed, an error will be thrown, notifying the user of the allowable inputs.
    - Here the conditions list is just a list of all the allowable characters for that parameter

JSON file structure outline

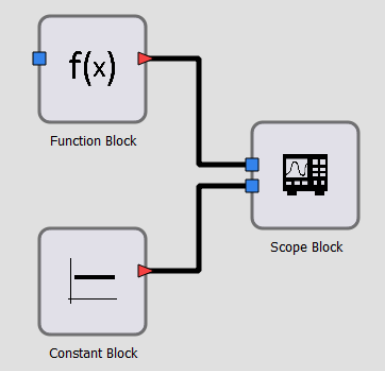
The JSON file structure contains all the necessary information of the reconstruction of blocks, sockets and wires that exist within a block diagram, and represents them as dictionaries, of key-value pairs which represent the name and value of parameters relevant to those items. All of these items (blocks, sockets, and wires) are contained within a Scene as explained in Section 3, hence they follow the following hierarchy:

- A Scene is represented as dictionary with:
    - Dimensions: these are two parameters of the width and height of the scene.
    - Blocks: this is a list of all the blocks within the diagram, with each block as a dictionary.
        - Sockets: each block has its own unique sockets, so these are stored as part of the block they belong to, also in a list, with each socket as a dictionary.
    - Wires: this is a dictionary list of all the wires within the diagram.

The following block diagram was made to aid with understanding this structure.

There are 3 blocks, created in the order:

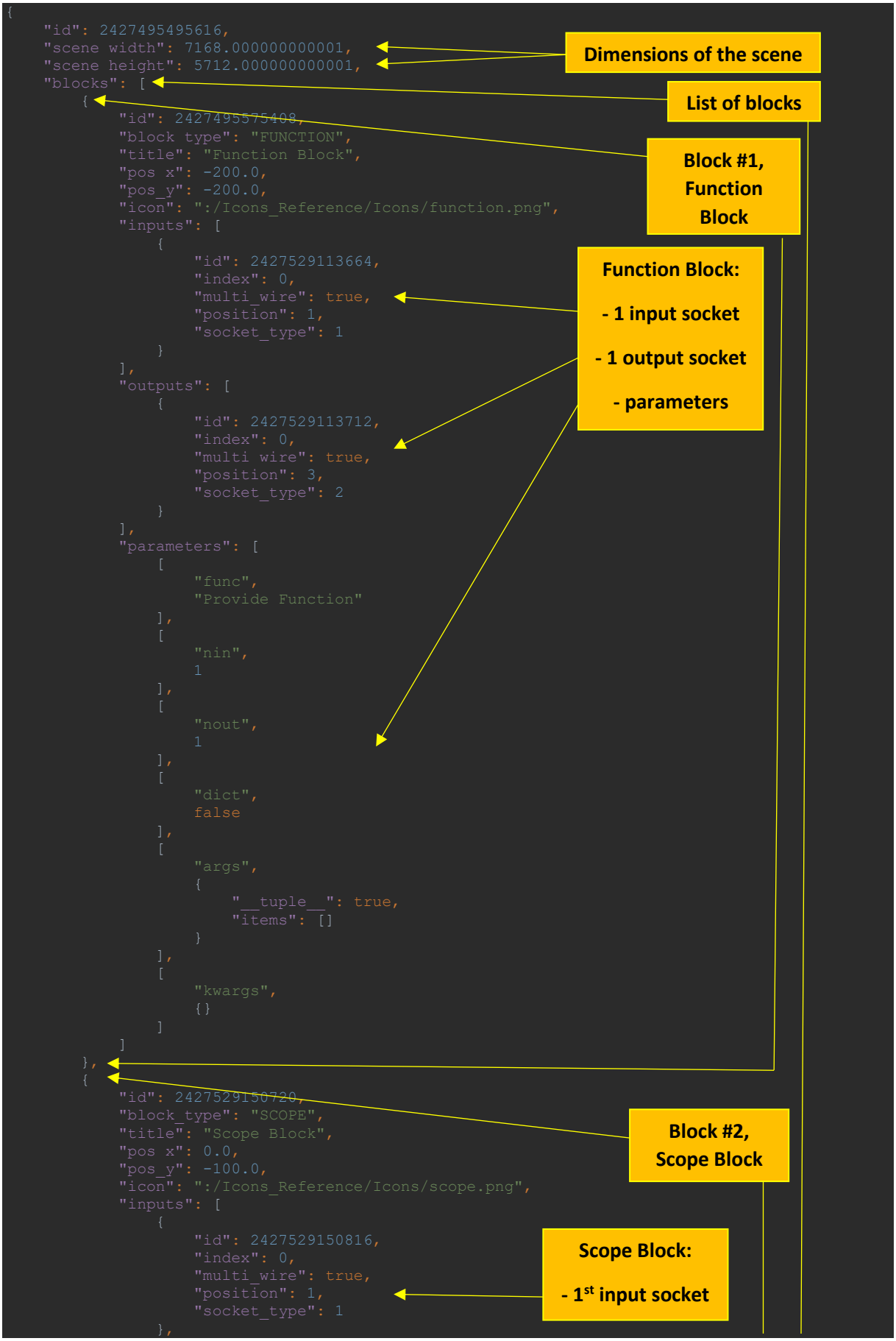
1. Function Block – has 1 input socket, 1 output socket
2. Scope Block – has 2 input sockets, no output sockets
3. Constant Block – has 2 input sockets, no output sockets

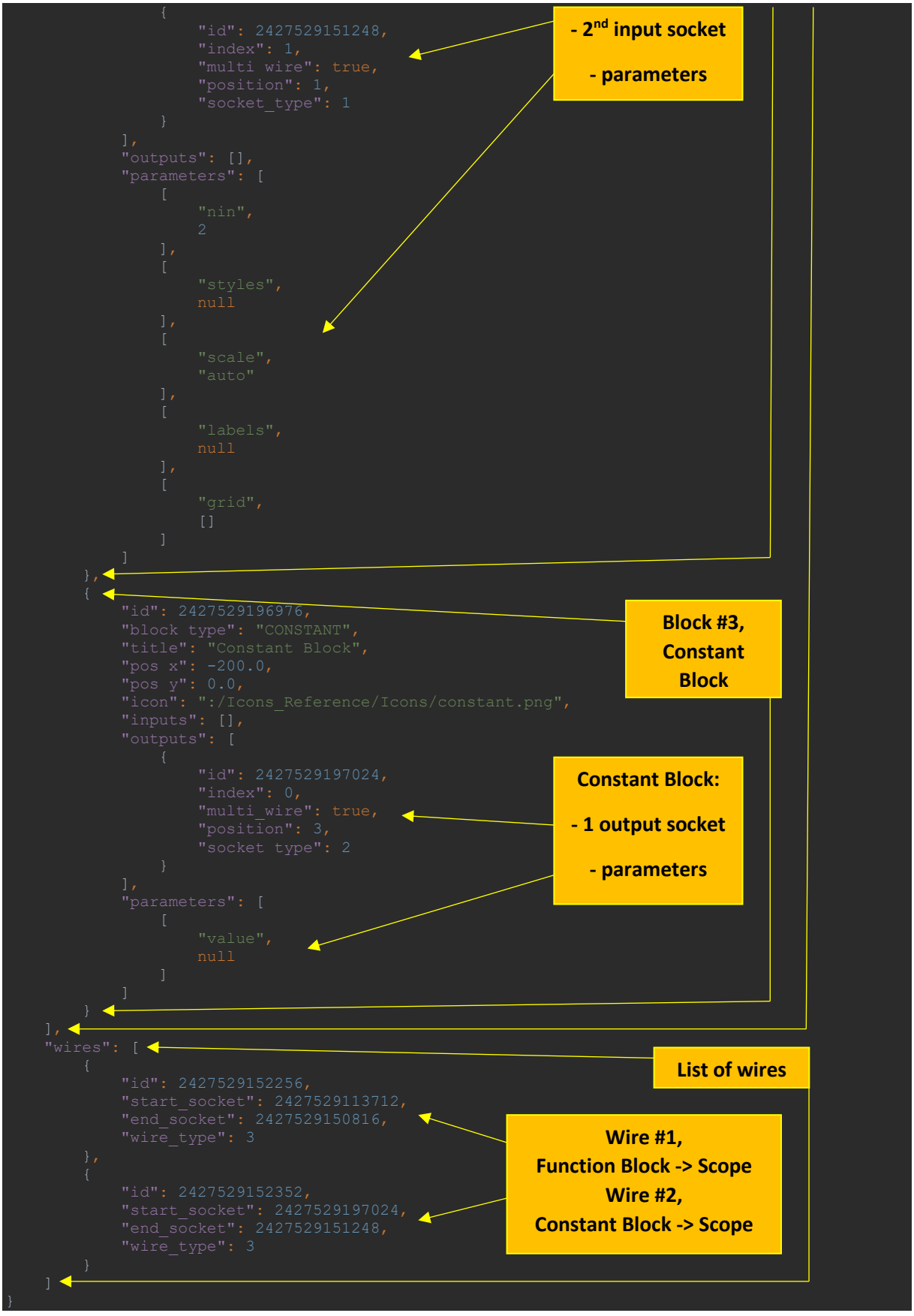


There are also 2 wires, created in the order:

1. Function block -> 1st input socket of Scope Block
2. Constant block -> 2nd input socket of Scope Block

Below, is the resulting JSON file that was generated.


{
"id": 2427495495616,
"scene width": 7168.000000000001,
"scene height": 5712.000000000001,
"blocks": [
{
"id": 2427495575408,
"block type": "FUNCTION",
"title": "Function Block",
"pos x": -200.0,
"pos_y": -200.0,
"icon": ":/Icons_Reference/Icons/function.png",
"inputs": [
{
"id": 2427529113664,
"index": 0,
"multi_wire": true,
"position": 1,
"socket_type": 1
}
],
"outputs": [
{
"id": 2427529113712,
"index": 0,
"multi wire": true,
"position": 3,
"socket_type": 2
}
],
"parameters": [
[
"func",
"Provide Function"
],
[
"nin",
1
],
[
"nout",
1
],
[
"dict",
false
],
[
"args",
{
"__tuple__": true,
"items": []
}
],
[
"kwargs",
{}
]
]
},
{
"id": 2427529150720,
"block_type": "SCOPE",
"title": "Scope Block",
"pos x": 0.0,
"pos_y": -100.0,
"icon": ":/Icons_Reference/Icons/scope.png",
"inputs": [
{
"id": 2427529150816,
"index": 0,
"multi_wire": true,
"position": 1,
"socket_type": 1
},
Dimensions of the scene
List of blocks
Block #1, Function Block
Function Block:
- 1 input socket
- 1 output socket
- parameters
Block #2, Scope Block
Scope Block:
- 1st input socket


{
"id": 2427529151248,
"index": 1,
"multi wire": true,
"position": 1,
"socket_type": 1
}
],
"outputs": [],
"parameters": [
[
"nin",
2
],
[
"styles",
null
],
[
"scale",
"auto"
],
[
"labels",
null
],
[
"grid",
[]
]
]
},
{
"id": 2427529196976,
"block type": "CONSTANT",
"title": "Constant Block",
"pos x": -200.0,
"pos y": 0.0,
"icon": ":/Icons_Reference/Icons/constant.png",
"inputs": [],
"outputs": [
{
"id": 2427529197024,
"index": 0,
"multi_wire": true,
"position": 3,
"socket type": 2
}
],
"parameters": [
[
"value",
null
]
]
}
],
"wires": [
{
"id": 2427529152256,
"start_socket": 2427529113712,
"end_socket": 2427529150816,
"wire_type": 3
},
{
"id": 2427529152352,
"start_socket": 2427529197024,
"end_socket": 2427529151248,
"wire_type": 3
}
]
}
- 2nd input socket
- parameters
Block #3,
Constant
Block
Constant Block:
- 1 output socket
- parameters
List of wires
Wire #1,
Function Block -> Scope
Wire #2,
Constant Block -> Scope

The important points to take away from this JSON structure, is that each block, socket and wire has unique ID's. Although the ID's of the blocks are less useful (can be helpful for differentiating between different blocks of the same time, although their name should be sufficient), the ID's of the sockets are very important. By definition each wire has a start socket and end socket that it is drawn between, and these are referred to by their unique ID. These unique ID's of the sockets, can then be traced back to the block they belong to, hence providing the crucial information of which block connects to which.

As an example, in this block diagram that was made, wire #1 connects the **Function block** from its **output**, to the **1st input** of the **Scope block**. In the wire, these start/end sockets are stored as "start_socket": 2427529113712 and "end_socket": 2427529150816, and if we check the socket id's of the **output socket** of the **Function block**, it is stored as "id": 2427529113712, which we expect to match our "start_socket" (which it does), and doing the same for the **1st input socket** of the **Scope block**, it is stored as "id": 2427529150816, which also matches our ID of the "end_socket".

How icons were created

Icons for bdedit were creating using the free image editing application, paint.net[4]. Paint.net is only compatible with windows operating systems, however there are other options available for image editing applications on Mac or Linux systems.

The sought after properties of image editing tools for creating/editing these icons were: for the ability to create layered images, moving layer position in the layer stack, hiding/showing the layer, selecting parts of images and separating them from the background (leaving only the outline/shape), and more importantly, support for transparent backgrounds.

Paint.net provides support for all these requirements, but other software options may also, so feel free to use whichever software, as long as the final images are saved in *'.png'* format and have transparent backgrounds. Paint.net also allows for files to be saved in their separate state as layers with the *'.pdn'* file type, however only Paint.net can open these files. For Windows users this shouldn't be a problem, but for users on other systems, the layers that make up each icon have been saved separately, which will allow them to be added as layers to which ever software is used on those operating systems.

Icons in general, were developed on a 250x250 pixel, transparent background. These dimensions were based off the blocks being 100x100 pixels when drawn within the interface, and the icon being drawn within these blocks having a space of 50x50 pixels to occupy. Having a 250x250 pixel raw icon image allows it to be scaled down to a 50x50 image through PyQt5's scaling tool, which retains the quality of the image better, resulting in it being less pixelated when zoomed in.

The items (text, dividing lines, function lines, etc.) within the icons were positioned with the help of a grid (named "*layout_grid.png*"), located within the Icons folder in which all the icons are stored. There wasn't a specific method for positioning the items within these icons, but depending on what the icon was (text, shape, axis with function) the lines of the grid were used to symmetrically place items and position them to allow for satisfactory visibility. These icons were developed in monochrome mode (black and white). The following values were used for lines and text:

- linewidth:
    - 6 for axis lines and outline of gain,
    - 8 for bolded thin lines (particularly as dividing lines),
    - 15 for outline of stop icon
    - 19 for function lines (clip, constant, piecewise, step, waveform)
- text font: Calibri
- text size: various amongst the text used, but will be one of the following:
    - 36, 48, 59, 72, 84, 108, 144
- text bolding: True for all text, apart from the stop icon

[4] Free paint.net download link - https://www.getpaint.net/download.html#download

Procedure for updating changes to existing icons, or adding new ones

As Python creates some difficulties in accessing absolute or relative file paths, and making this consistent across all users, an alternative available from PyQt5 was used, which is a QReferenceFile.

Similar to zipping files into a folder, this reference file packages image files into a Python importable file, which after importing, can then be accessed locally within the same directory. Hence all icons which were used, were placed into the Icons folder, located at "***bdsim/bdsim/bdedit/Icons***" folder (see Figure 4.1), and a reference file named "***Icons.qrc***" was made (following the structure below) in the bdedit package, located at "***bdsim/bdsim/bdedit***" folder (see Figure 4.1).

In order to make the "***.qrc***" (resource) file importable and usable in the Python scripts, it must be converted to a "***.py***" (Python) file (in our case named "***Icons.py***"). The steps for this procedure will be outlined below.

The following steps should be followed for updating an icon that already exists

1. The new icon (under the same name as the existing icon it's replacing) should be added as a "***.png***" to the "***bdsim/bdsim/bdedit/Icons***" folder, replacing the old version of the icon.
2. The "***Icons.qrc***" file located at "***bdsim/bdsim/bdedit***" should be converted to "***Icons.py***" with the following steps:
    a. Via the terminal, navigate to the bdedit directory
    b. Via the terminal, type "***pyrcc5 Icons.qrc -o Icons.py***" to convert and write the contents of the resource file (named "***Icons.qrc***") to a Python file (named "***Icons.py***"). The name of the resource file should match that of what was given to the resource file, and the name of the Python file will be what is called when importing into other Python files (as "***from bdsim.bdedit.Icons import ****"). DO NOT make any changes to this generated Python file, as this could result in unforeseen errors.
3. Assuming the icon that is being replaced, was previously set up and being used, re-running the program after updating the "***Icons.py***" file will update the changes made to this icon.

The following steps should be followed for adding in a new icon, that doesn't exist anywhere in the code:

1. The new icon (with a unique name) should be added as a "***.png***" to the "***bdsim/bdsim/bdedit/Icons***" folder.
2. The structure of the "***Icons.qrc***" resource file located at "***bdsim/bdsim/bdedit***" should be edited to include the file path to this newly added icon. The following steps should be taken:
    a. Open the resource file with any text viewer (should see a file similar to Figure 4.3)
    b. Add the file path to the new icon as "***Icons/filename.png***" enclosed in the <file> and </file> tags, indicating that this icon is located within the Icons folder. Note the qresource prefix name; this will be used for picking out specific icons from this file.
    c. Save the resource file and proceed to Step 3.
3. The "***Icons.qrc***" *resource* file located at "***bdsim/bdsim/bdedit***" should be converted to "***Icons.py***" with the following steps:
    a. Via the terminal, navigate to the bdedit directory

b. Via the terminal, type "***pyrcc5 Icons.qrc -o Icons.py***" to convert and write the contents of the resource file (named "***Icons.qrc***") to a Python file (named "***Icons.py***"). The name of the resource file should match that of what was given to the resource file, and the name of the Python file will be what is called when importing into other Python files (as "***import Icons***"). DO NOT make any changes to this generated Python file, as this could result in unforeseen errors.

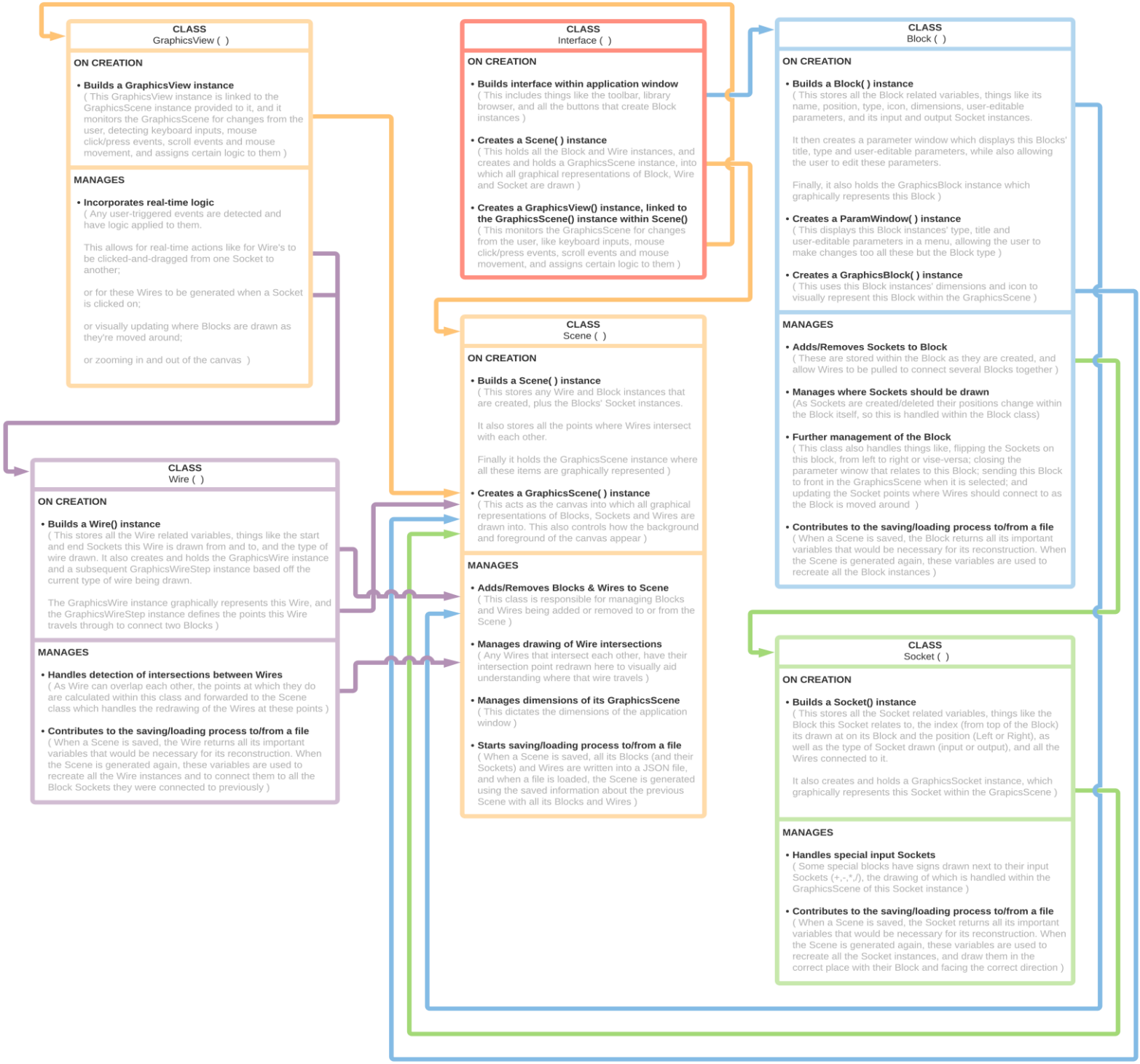
4. Next, you should open the file where you planned to use the icon (this should be one of the files in either the "***bdsim/bdsim/bdedit/Block_Classes***" folder or in the "***bdsim/bdsim/bdedit***" package).
    a. If in any file within the Block_Classes folder, continue to Step 5.
    b. If in any other within the bdedit package:
        i. Import the "***Icons.py***" file from the bdedit package as: "***from bdsim.bdedit.Icons import ****".
5. Still in the same file as Step 4, continue to the code where you want the Icon file path to be defined and insert the following string: "***:/Icons_Reference/Icons/filename.png***". The "***:/***" notation is important for navigating the "***Icons.py***" file. Also, remember from the note in Step 2, the qreference name is used here as "***Icons_Reference***" to refer to the "***Icons.py***" file. The "***/Icons/filename.png***" points to the path defined in the "***Icons.qrc***" file. Replace "***filename***" with whatever name you saved the icon under.
6. This should complete the process for adding in a new icon.

```
<!DOCTYPE RCC><RCC version="1.0">
 <qresource prefix="Icons_Reference">
     <file>Icons/clip.png</file>
     <file>Icons/function.png</file>
     <file>Icons/gain.png</file>
     <file>Icons/interpolate.png</file>

     <file>Icons/print.png</file>
     <file>Icons/stop.png</file>
     <file>Icons/scope.png</file>

     <file>Icons/constant.png</file>
     <file>Icons/step.png</file>
     <file>Icons/waveform.png</file>
     <file>Icons/piecewise.png</file>

     <file>Icons/integrator_B.png</file>
     <file>Icons/dintegrator_B.png</file>
     <file>Icons/lti_siso_B.png</file>
     <file>Icons/dlti_siso_B.png</file>
     <file>Icons/lti_ss.png</file>
     <file>Icons/dlti_ss.png</file>

     <file>Icons/integrator_L.png</file>
     <file>Icons/dintegrator_L.png</file>
     <file>Icons/lti_siso_L.png</file>
     <file>Icons/dlti_siso_L.png</file>

     <file>Icons/Success_Icon.png</file>
     <file>Icons/bdsim_icon.png</file>
 </qresource>
 </RCC>
```

Figure 4.3 - QResource File Structure

5. Appendices

APPENDIX A – High Level Class Architecture Diagram



APPENDIX B – Stepped Wire Drawing Logic

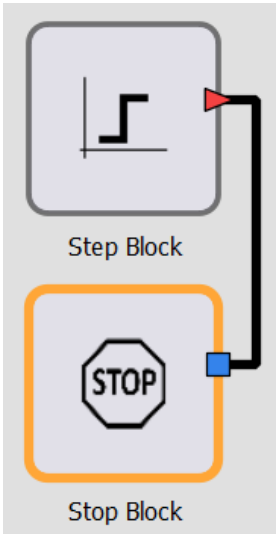
Drawing the step wire falls into three steps of logic.

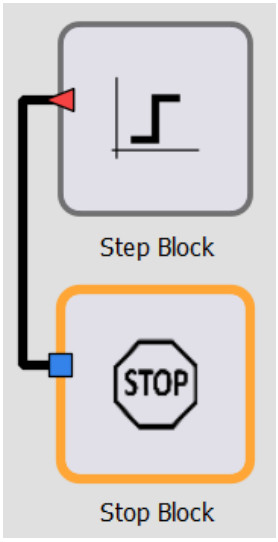
The first step: when a wire is being pulled from one socket to another, and has not yet been connected.

In this step, the wire is simply drawn as a straight light from the starting socket to the mouse cursor, up until the point the wire is connected to another socket. This is when the wire drawing logic falls into the following two steps of logic.

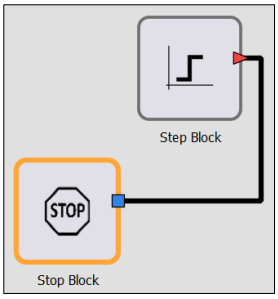
The second step: Wire routing logic between two blocks where the input and output sockets are on same sides

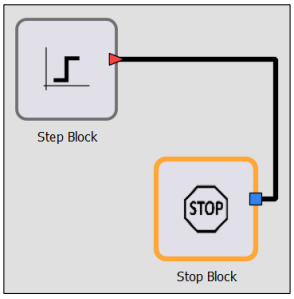
1) Starts off with blocks being above/below each other

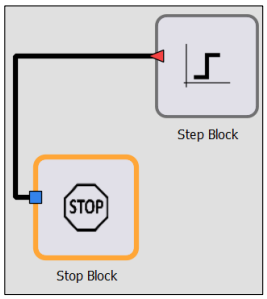


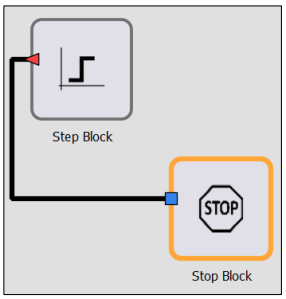


2) As one of the blocks is moved horizontally

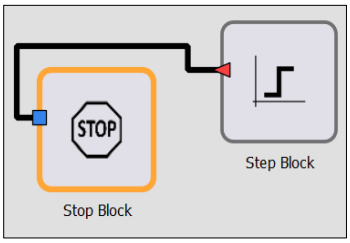


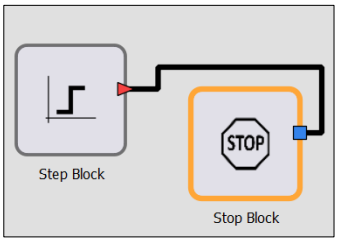


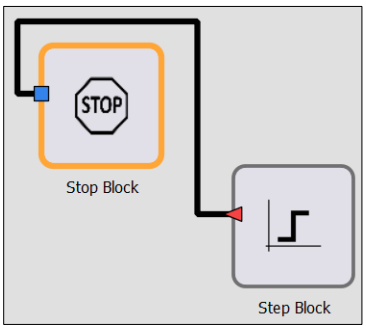


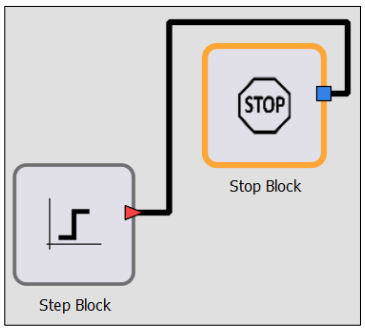


3) As one of the blocks is moved up/down such that it would overlap a horizontal wire segment the following wire is drawn. Moving the block further up/down will revert to one of the wires from 2).

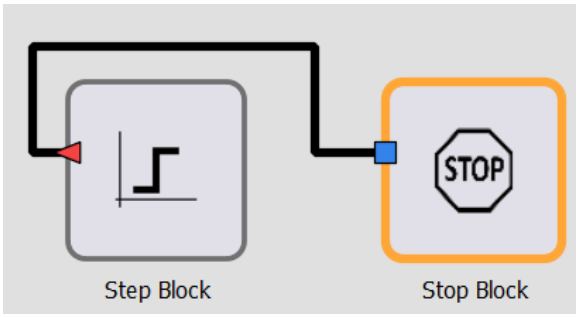


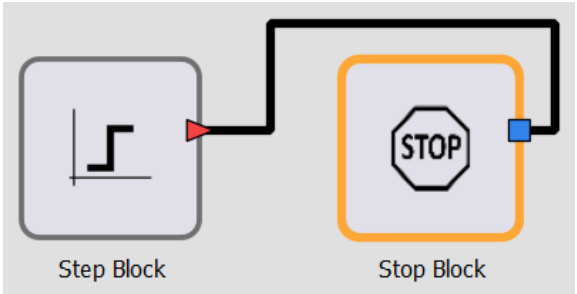






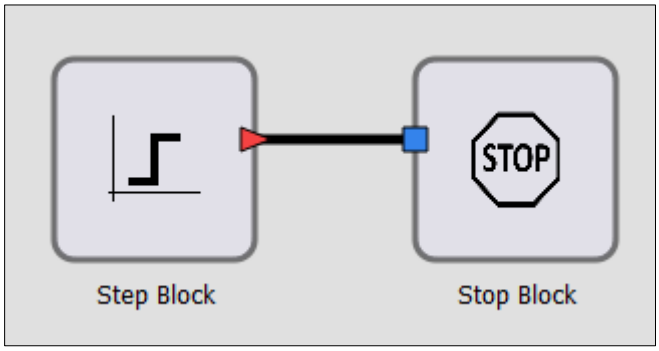
4) As one of the blocks is move to the opposite side of the block it's connected to, the following wire is drawn.



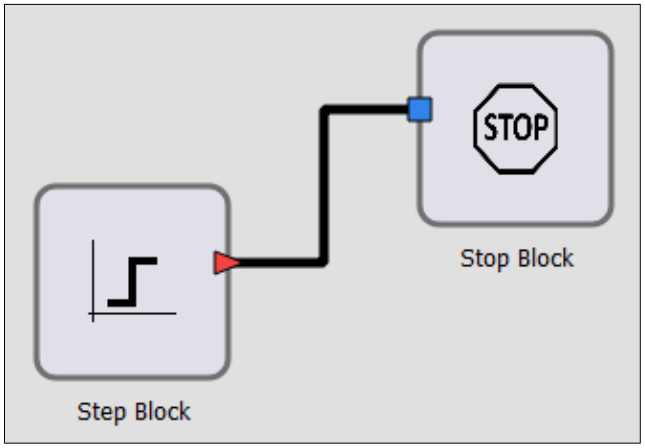


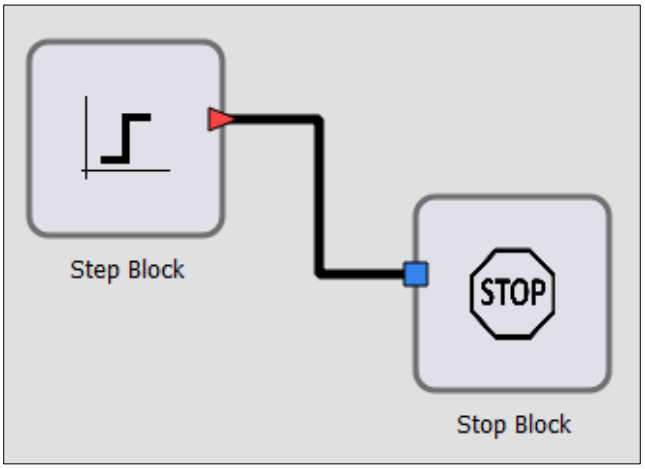
The third step: Wire routing logic between two blocks where the input and output sockets are on opposite sides

1) Starts off with straight line as blocks are side by side

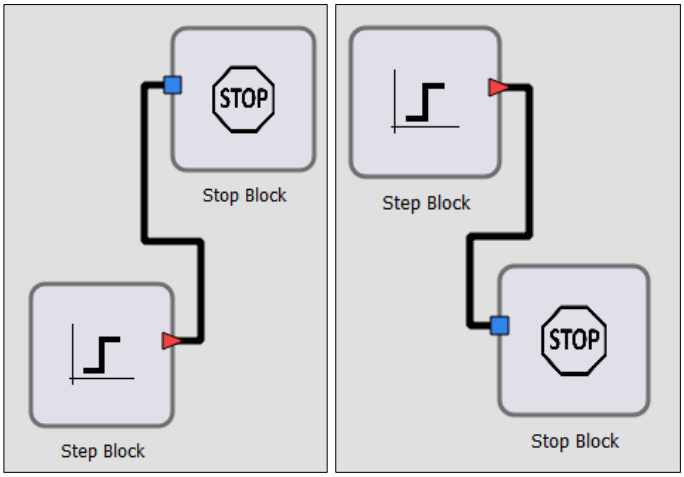


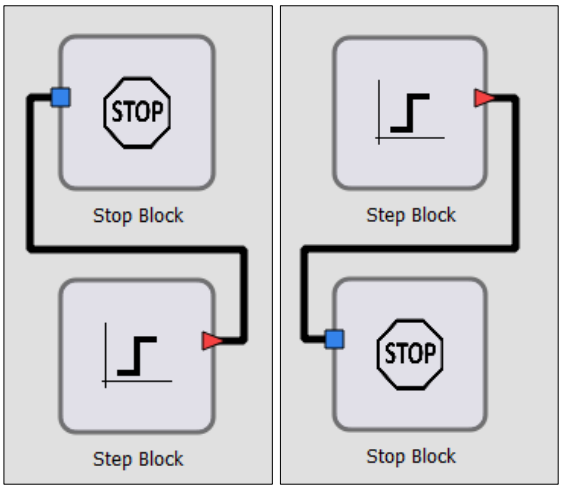
2) As one of the blocks is moved up or down a basic step wire is drawn

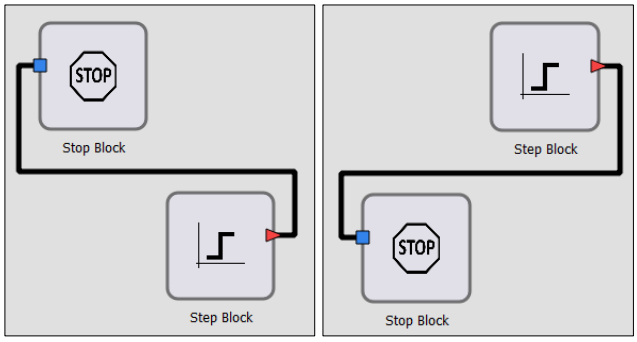




3) As the inside edges of the blocks start to overlap horizontally, a more complex (S or Z type) wire is drawn







4) As the selected block starts to overlap the middle horizontal segment between the two blocks, the wire is wrapped around the top of one of the blocks (whichever is higher)