malcolmjs Documentation

Release 1.7.12-2-g16292d4-dirty

Tom Cobb, Ray Millward, Bryan Tester

Feb 10, 2020

Overview

CHAPTER 1

Introduction

1.1 Overview

MalcolmJS provides a web-based graphical interface to the underlying functionality provided by the Malcolm middleware library, which itself provides a common interface to a range of control system and data management technologies.

The web-based environment does not provide direct access to hardware components. Instead, information about the hardware components themselves, and the data they generate, is shared with our user environment via Malcolm. This allows you to generate a *virtual representation* of your control system before providing design and configuration information about it to Malcolm via a dedicated service. Malcolm takes this information and uses it to interface with the underlying physical hardware. We can consider this schematically as:

The Malcolm Service provides three key capabilities:

- Details of the building blocks available to our user environment, allowing a System Implementor to set up their control system components realistically. During this process design decisions can be validated against pre-defined parameters to ensure validity of the design.
- Once deployed into an operational environment the service monitors attribute status and data as it moves through the system, providing near real-time monitoring of key components and indicating when pre-defined operating parameters are exceeded.
- Ongoing persistance of design information and results during operation, providing assurances that designs are documented (and can be interrogated in the future), and can be re-used in the future.

1.2 How is the Documentation Structured

This documentation is divided into two sections:

- The *User Guide* provides information about how to work with your new environment. The *Quick Start* guide within this contains everything you need to get up-and-running. Subsequent sections provide more reference-focussed material describing functionality and capabilities in more detail.

- The *System Maintenance Guide* describes how the system has been designed and implemented. It also contains information on the routine maintenance activity you will need to perform to keep your system performing in an optimal way.

CHAPTER 2

Getting Started

This user guide introduces the web-based environment, toolkit and processes available to you as you develop and manage your control system through its integrated user interface. This interface allows you to:

- Design and configure the building blocks and linkages between them as a *virtual representation* of your control system.
- Optimise your `design` through interaction with the physical hardware of your control system implementation.
- Utilise and manage your `design` in real-world operational scenarios through monitoring and basic analysis tools.
- Save snapshots of your `design` for re-use later, or for traceability purposes in support of good scientific practice.
- Load saved designs to quickly revert to a 'known good state' or to reconfigure your underlying hardware.

CHAPTER 3

Quick Start

This quick start walkthrough aims to show you how a simple workflow might be carried out using MalcolmJS, it will be brief but the user interface should feel intuitive enough that this guide will allow you to get started quickly on your own instance.

1. Start by navigating to `http://{malcolm host}/gui/` where `malcolm host` is the malcolm instance you're trying to connect to, e.g. `localhost:3000`

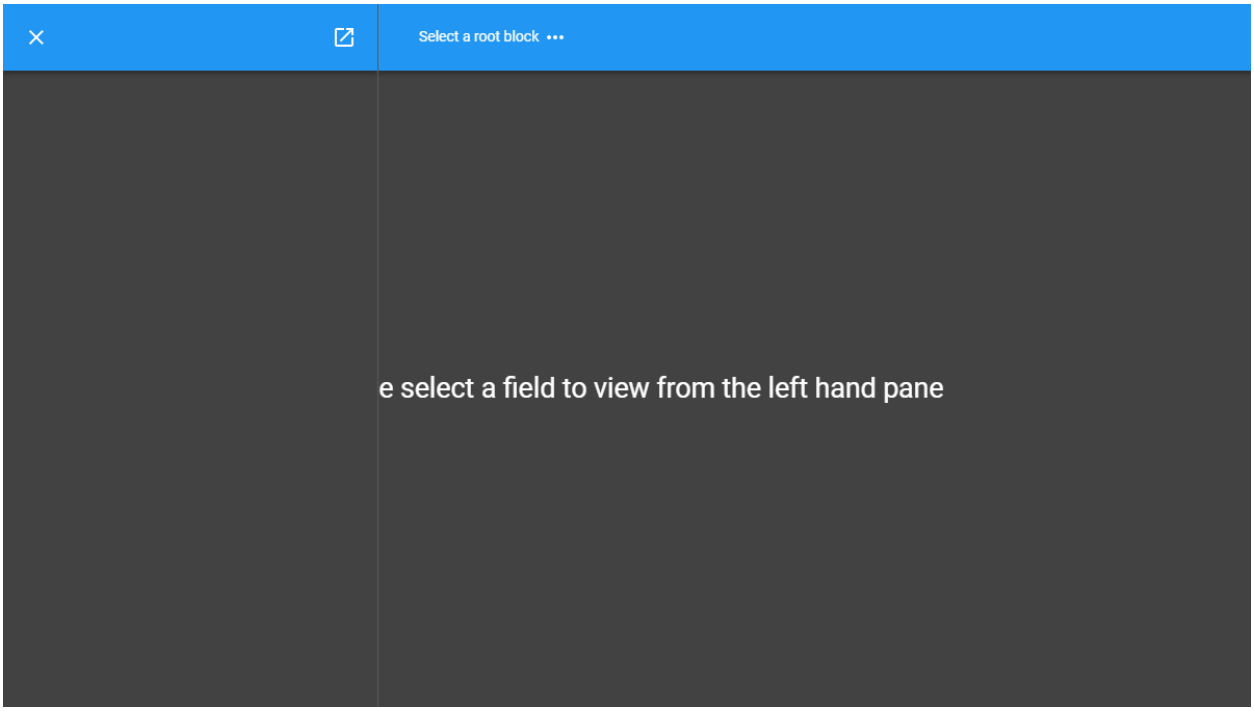

Fig. 1: The intial screen

2. When the page has loaded there will be a drop down at the top of the page indicating that you need to pick a root node. Click on the drop down and you will get a menu with the list of root blocks.

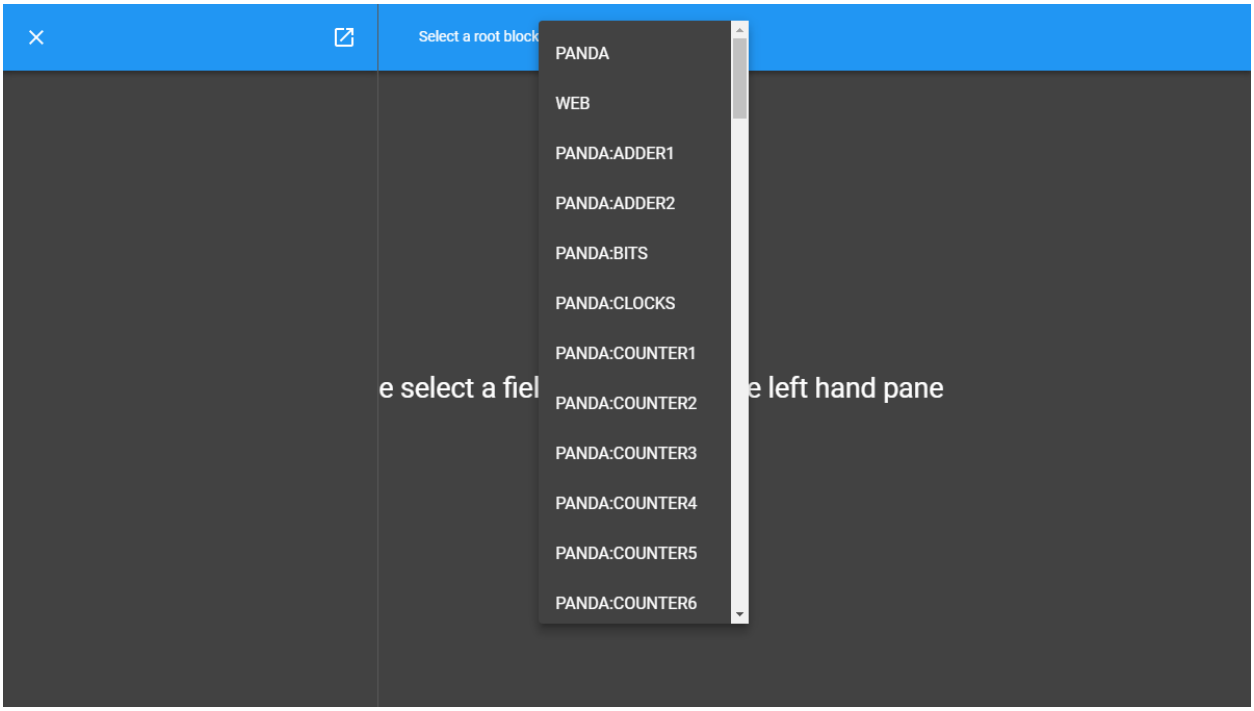

Fig. 2: Selecting a top level block

3. In this case click on `PANDA` and this will start loading the details about the `PANDA` block in the left hand panel, this is known as the parent block.
4. Once the block details have loaded you will see various attributes, these are either top level attributes, attributes in a group or methods. There will be an attribute called `Layout` that will look like the screenshot below.
5. Click on the `Edit` button to load the block layout for the PANDA.
6. You can then drag blocks around to update the layout, be aware that it may take a short amount of time for the position to update on the server. This is indicated by the spinner icon.

   Note as well that clicking on a block will load the details for that block in the right hand panel, this is known as the child block.
7. You can then click on a port and drag to another port to create a new link.

   In this way blocks can be wired together to perform more complex operations.

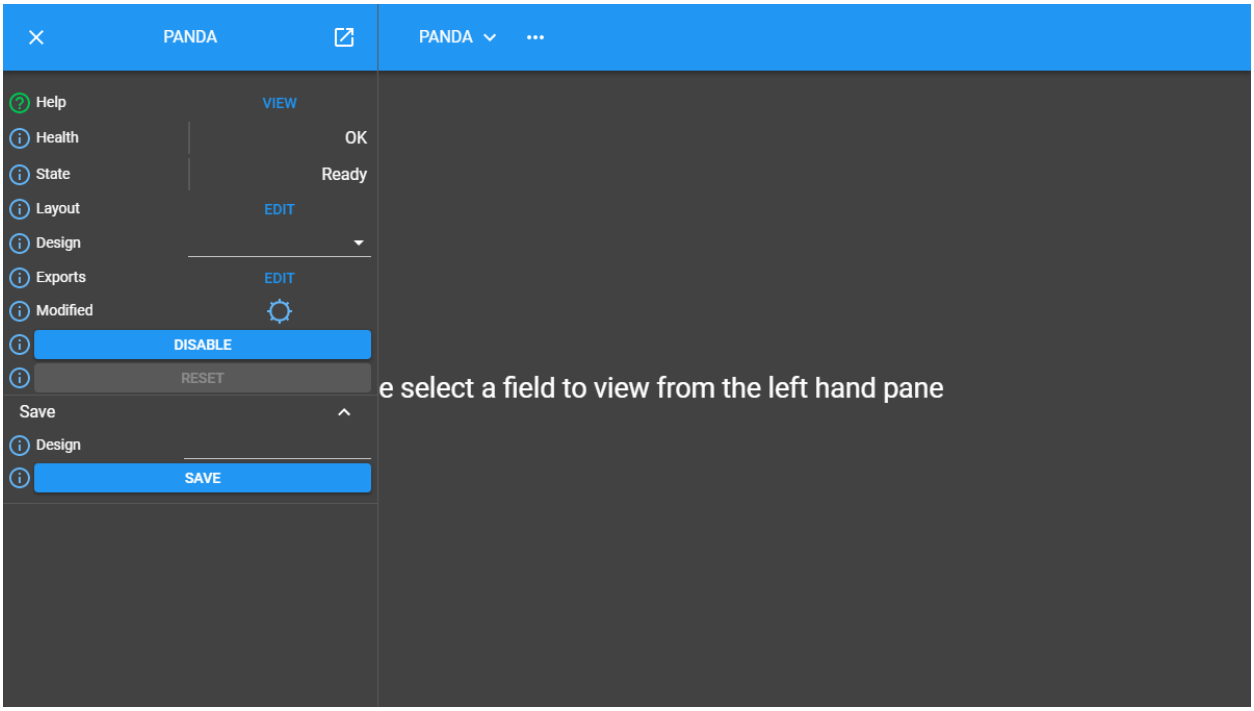

Fig. 3: The details for a PANDA block

Fig. 4: Click on the `Edit` button in the layout attribute to see the layout in the middle panel

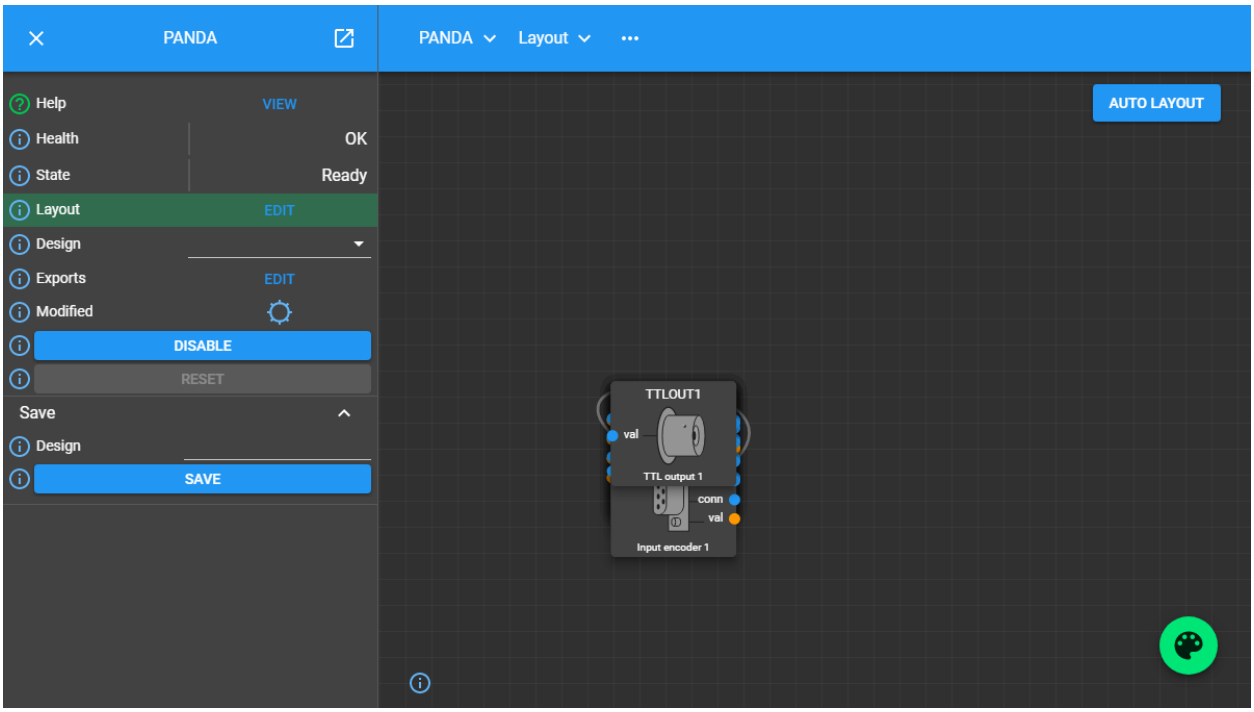

Fig. 5: The layout for the PANDA block

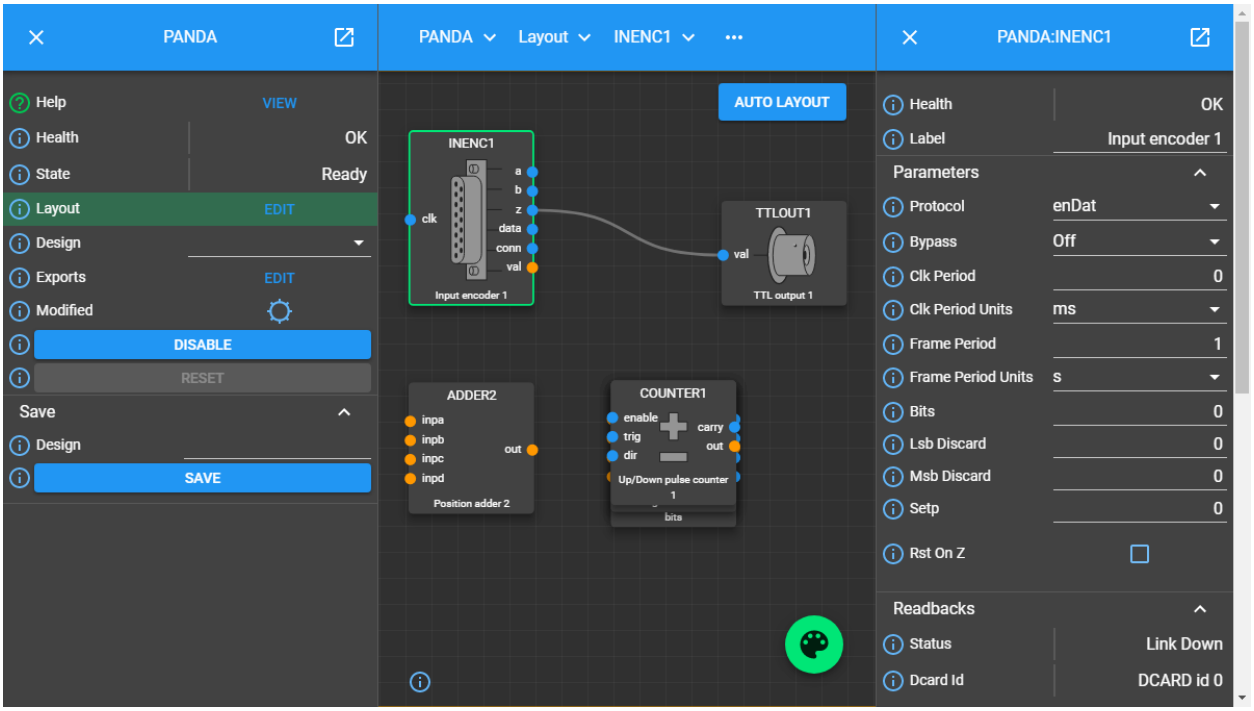

Fig. 6: Moving blocks around and seeing the details for a child block.

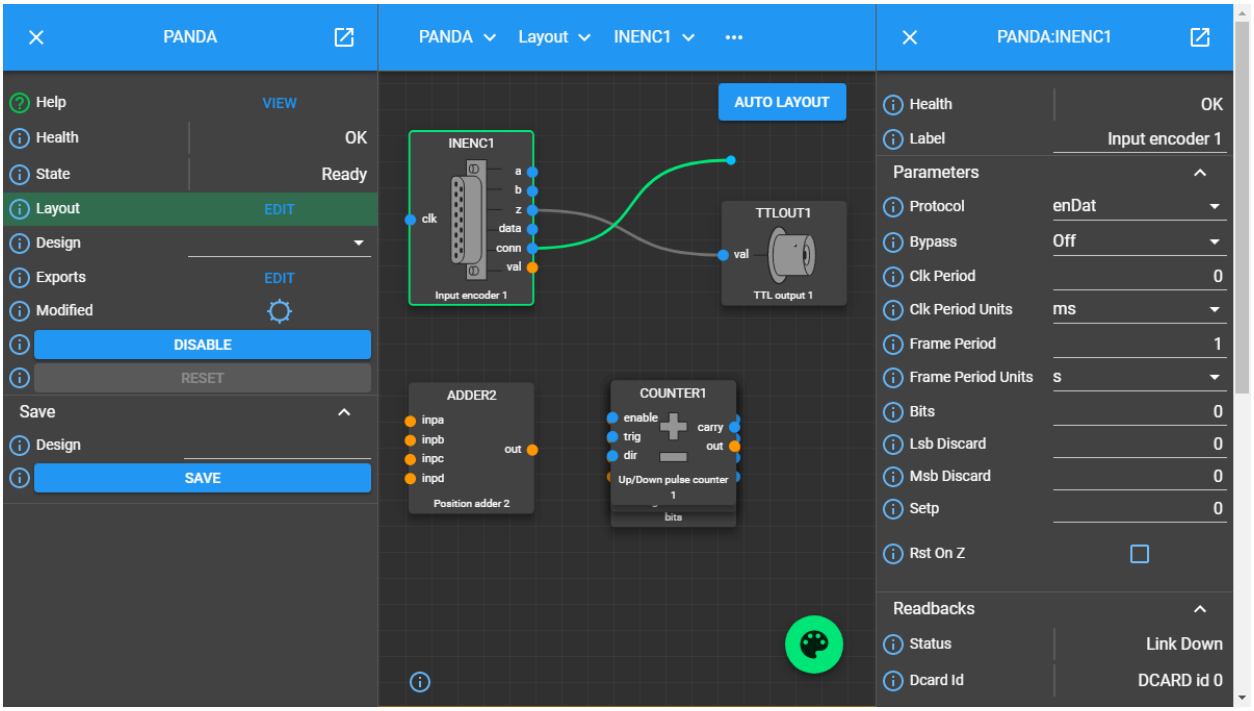

Fig. 7: Creating a new link

CHAPTER 4

User Interface Overview

The user interface provides a fully interactive environment for designing, configuring and managing components and the links between them that together define the underlying control system.

4.1 User Interface Components

The user interface consists of four core components. The content of each compoent changes dynamically to support the activity you are undertaking:

Looking at each component in more detail:

Component	Description
Navigation Bar	The *'navigation bar'* at the top of the screen provides the ability to move through the currently open Design, selecting design elements at increasingly deep levels of implementation beginning at your selected `Root Block`. In doing so it provides a breadcrumb-like map of where you currently are within the Design.
Left-hand Panel	The *'left-hand panel'* provides general information about the `Parent Block` currently forming the central focus of interest within the user interface.
Central Panel	The *'central panel'* displays information about an Attribute selected from the Parent Block presented in the 'left-hand panel'. If **Layout** is selected from a `Parent Block` then the *Layout View* is displayed. In *Attribute View* a *plot* or *table* of Attribute value data against time is displayed.
Right-hand Panel	The *'right-hand panel'* provides detailed information about the Block, Attribute or Link currently in focus. For example, if the 'left-hand panel' represents a `block` the 'right-hand panel' reflects an `attribute` within that Block or a `link` associated with it. If the 'left-hand panel'represents the `Parent Block` then the 'right-hand panel' may contain details of an Attribute associated with that block or a `Child Block` contained in the Parent Block's Design.

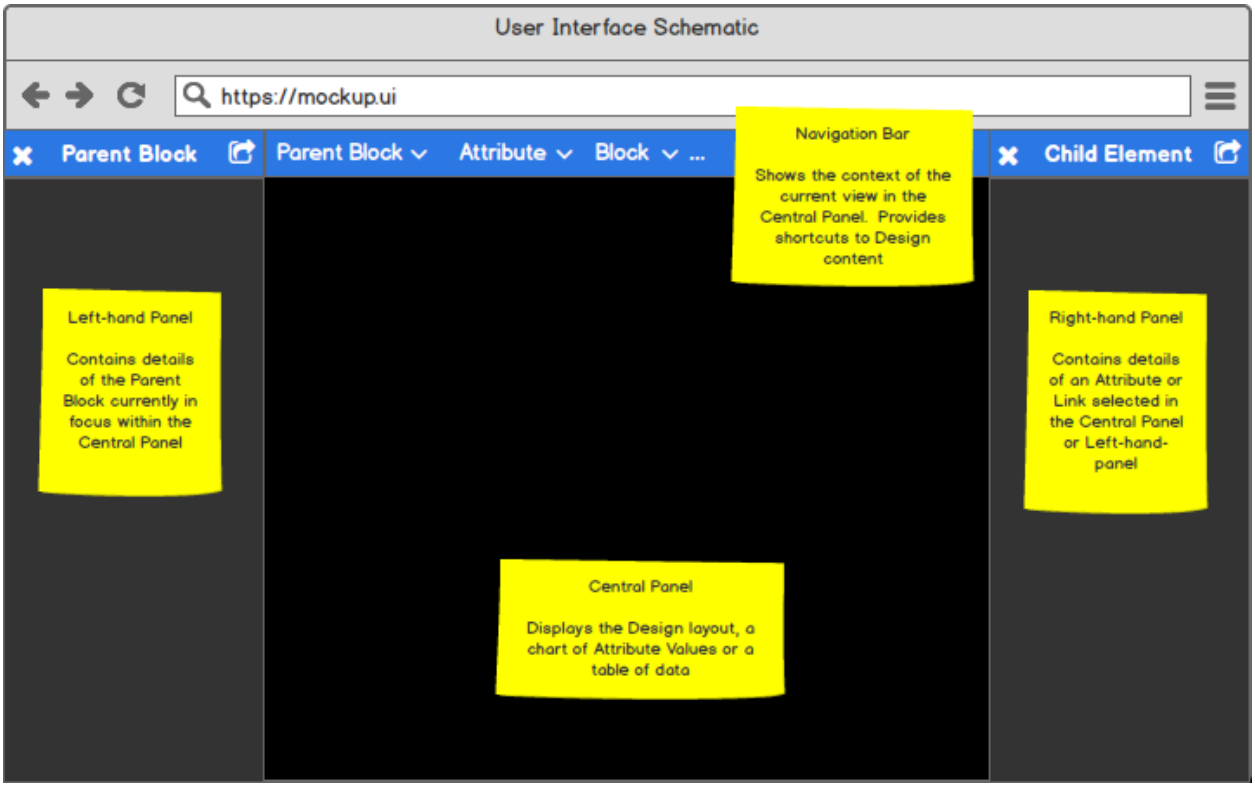

Fig. 1: User interface overview

4.2 Principle User Interface views

To support design, configuration and management activies the user interface provides two principle views into the underlying control system:

View	Content
Layout View	Provides an interactive environment for designing and configuring your control system through `block`, `attribute` and `link` specification. The resulting `flowgraph` provides a visual representation of the formal `design`.
Attribute View	Provides details of a single Attribute, including the ability to view graphical representation of the Attribute's value within the control system over time.

Tip: Remember that the location of information, and its nature, will depend on the context in which you are viewing it. In summary:

- A Block is *always* displayed in the left-hand panel. This may represent a Parent Block or a Child Block.
- Attribute meta-data is always displayed in the right-hand panel.
- Link information is always displayed in the right-hand panel.

4.3 Layout View

The Layout View is used to create, modify and manage the overall Design of your control system.

The Layout View is accessed via the **'View'** or **'Edit'** button associated with the *'Layout'* Attribute on a `Parent Block`.

Once the `layout` is displayed individual `blocks` within it can moved around the screen by clicking and dragging them to your desired location. Any `links` will be dynamically re-routed to accommodate the new location. Note there is also the ability to automatically optimise the Layout via the **'Auto Layout'** button within the 'central panel'.

A typical example of the view you may expect to see is shown below.

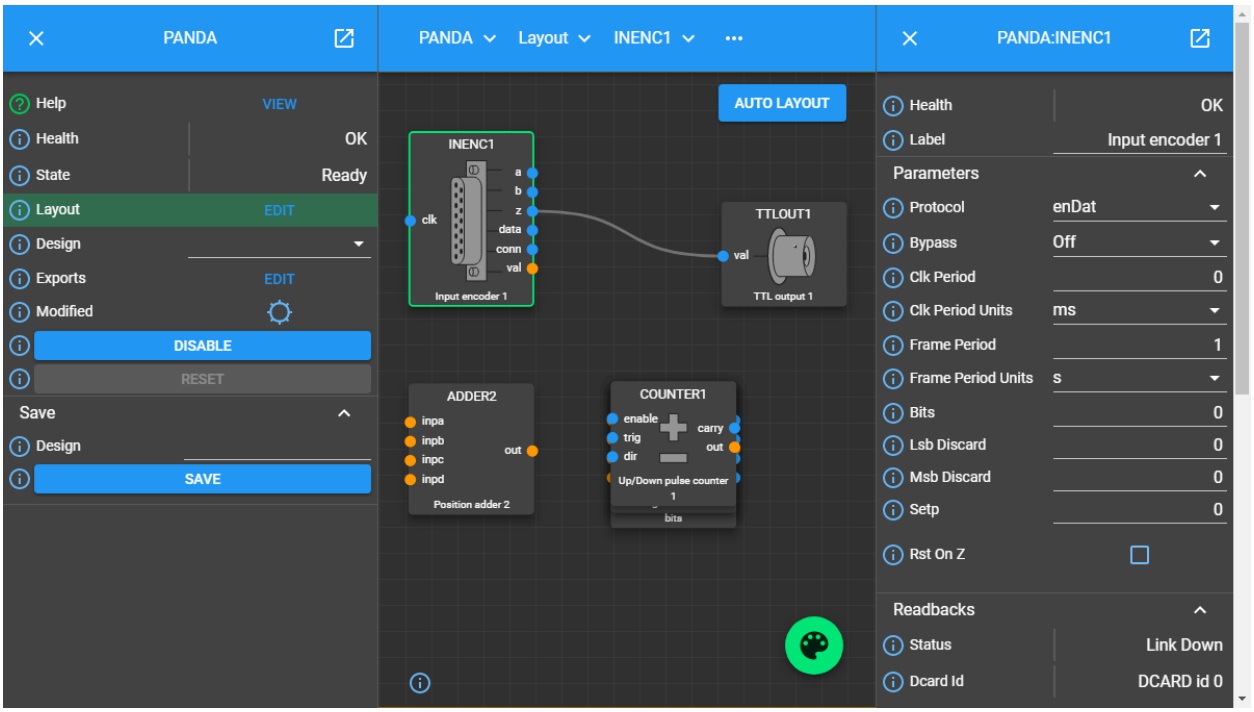

Fig. 2: Example layout

In this example we see:

- Summary information about the 'PANDA' system displayed in the 'left-hand panel'.
- The Design of the 'PANDA' presented in the central `layout` panel. Note the 'CLOCKS' `block` is highlighted.
- Detailed information about the 'CLOCKS' Block in the 'right-hand' panel, including all of its pre-defined `attributes`.
- The 'navigation bar' denoting that we are viewing the 'CLOCKS' Block via the layout of the 'PANDA' System.

4.4 Attribute View

The user interface automatically transitions to Attribute View when an Attribute is selected from either the 'left-hand panel' or 'right-hand panel'.

- If the Attribute is selected from the 'left-hand panel' more detailed information about that Attribute is displayed in the 'right-hand panel'.

- If the Attribute is selected from the 'right-hand panel' the *Block* represented in the 'right-hand panel' is transferred to the 'left-hand panel' as the new focus of interest, with more detailed information about the selected Attribute now presented on the right.

In both cases the 'central panel' presents a view of the Attribute's value against time. This may represent a constantly changing value, for example a calculated data value updated every 2ms, or a periodically changing boolean on/off status indicator that only changes every 10h. Two representations of the Attribute value are available and can be selected by choosing the appropriate option at the bottom of the 'central panel' thus:

- Plot - presents the Attribute Value as a line chart, displaying value against time. This graphical view is interactive and as a user you have the ability to undertake basic activities within the chart including panning, zooming and exporting for offline use. See *Monitoring Attribute Values* for further information.
- Table - presents the Attribute Value as a series of rows in a table. Each row represents the value at a different time point.

For example, viewing the plot associated with the 'Val' Attribute of the 'PANDA Input Encoder' Block:

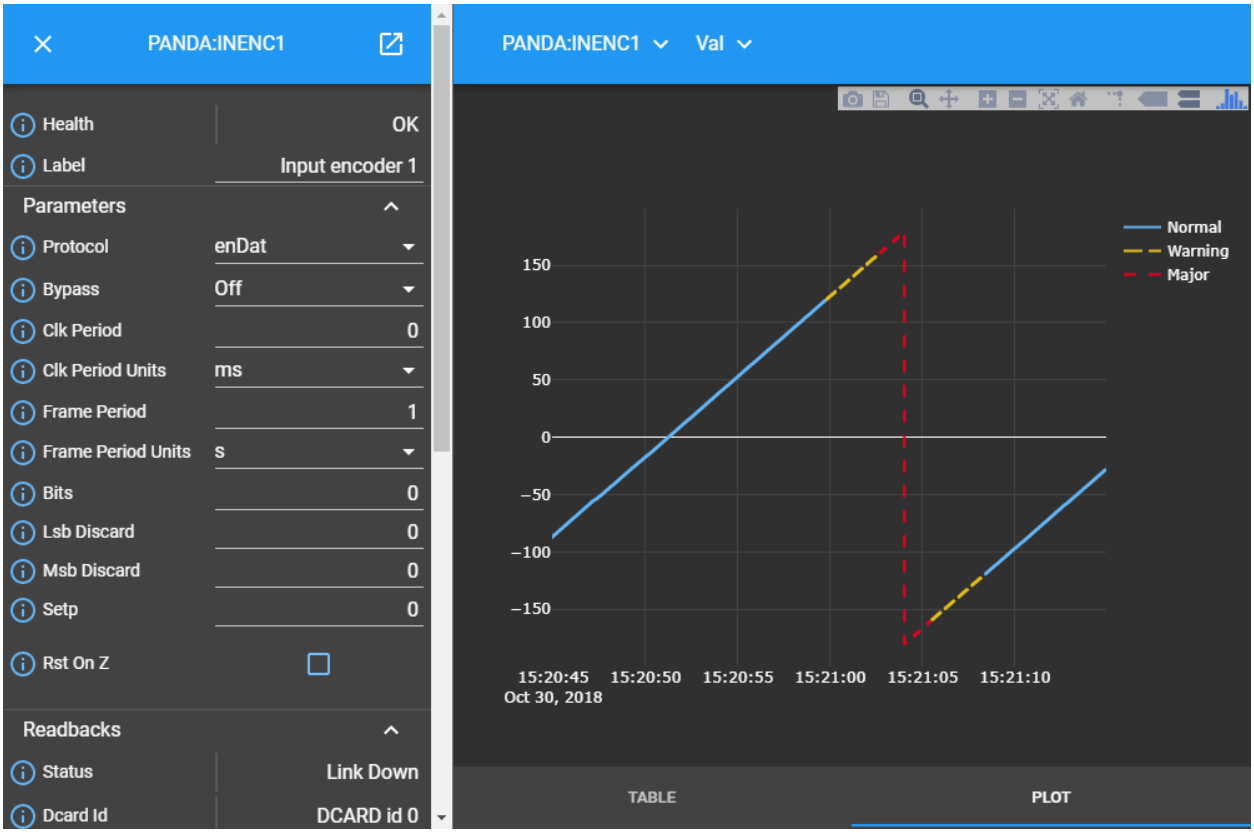

Fig. 3: Example plot showing continuously recorded data

Note: Since we are now in 'Attribute View' the left-hand panel contains details of the 'Input Encoder 1' Block not the 'PANDA' Parent Block.

4.5 Panel Popping

Under normal use the 'left-hand panel' contains summary information about the current *block* in focus and the 'right-hand panel' detailed information relating to an *attribute* or *method* associated with that Block. In complex systems it may be desireable to display information about a number of connected Blocks to track how each updates as data moves through the system they represent. This can be achieved by *'popping'* the Block Information Panel via the icon in the top left-hand corner of the panel. This causes the Block Information Panel to open in its own independent window. Multiple panels can be opened in the same way.

Connectivity to the underlying system is maintained meaning each independent window is updated in response to activity within the control system. Similarly, manual updates to any Attribute within an independent Information Panel is reflected back to the control system in the same way as occurs when the Information Panel is integrated with the main user interface. For example:

Fig. 4: Example of multiple Block Information Panels popped into an independent display

In this image, which spans two monitor screens, we see three Child Blocks ('CLOCKS', 'COUNTER1' and 'BITS') associated with the 'PANDA' Parent Block *popped* into individual windows and displayed alongside the overall 'PANDA' Layout.

CHAPTER 5

Working With a Design

A *design* forms the heart of your system implementation. It provides an interactive graphical representation of your system, helping you build and manage:

- *Blocks* representing hardware components, logic gates, etc.
- The *connectivity* between blocks, in terms of input (sink) ports and output (source) ports.
- The *attributes* associated with blocks and links.
- The *methods* available to influence behaviour within blocks.

A Design is created in the user interface *Layout View*.

5.1 Adding a Block to a Design

A *block* is added to a *design* by dragging and dropping it from the 'Block Palette' into the *Layout View* as follows:

1. Select the **'Palette'** icon at the bottom of the Layout Panel. The Block Palette opens containing the set of blocks currently available to you.
2. Identify the Block you wish to add. By hovering over it the mouse pointer changes from an arrow to a hand.
3. Click the left mouse button to select the Block and while holding down the mouse button drag the Block into the Layout Panel.
4. When you reach your desired location for the Block within the Layout Panel release the mouse button.

The Block Palette icon is replaced by a full representation of the selected Block, showing:

- The Block name (shown relative to its *Parent Block*).
- An optional, configurable descriptive label (initially containing default text).
- *Source Ports* responsible for transmitting output from the Block, including their type.
- *Sink Ports* responsible for receiving input to the Block, including their type.

After adding a Block to the Layout Panel it can be selected by hovering over it and clicking the left mouse button. Upon selection the *Block Information Panel* presenting each *attribute* and *method* available to that Block is displayed in the right-hand panel of the web interface.

Note: On intially adding a new Block to your Design it is configured according to its pre-defined default settings retrieved from the underlying Design Specification of that Block.

5.2 Removing a Block from a Design

If a *block* has been added to a *design* erroneously, or is no longer required within the current Design it can be removed in one of two ways:

1. *By dragging it to the Bin:*
    1. Select the Block to be removed by hovering over it and clicking the left mouse button. Upon selection a **'Bin'** icon is displayed at the bottom of the Layout Panel.
    2. While holding down the left mouse button drag the Block over the **'Bin'** icon. The icon is highlighted.
    3. Release the left mouse button.
2. *Hitting the Delete or Backspace Key:*
    1. Select the Block to be removed by hovering over it and clicking the left mouse button. The selected Block is highlighted.
    2. Hit the *Delete* key or *backspace* key on your keyboard.

Note: Upon removing a Block from your Design all *Source Port* and *Sink Port* links associated with it are automatically removed.

5.3 Working with the Block Palette

The Block Palette contains a list of each *block* available to a *design* based on pre-defined constraints imposed by the underlying hardware infrastructure associated with the system.

When a Block is selected from the Block Palette for inclusion in a Design it is removed from the Block Palette to ensure it is not included more than once. If all Blocks of a particular type have been added to a Design it is not possible to add any more as the underlying hardware implementation will not be able to represent them.

If a Block is *removed* from a Design it is immediately available again for selection in the Block Palette.

5.4 Specifying Block Attributes

The behaviour of a *block* is defined via its *attributes*. Attributes are pre-defined based on the function of the Block and may include default values providing a starting point for later implementation-time customisation. A full list of the attributes associated with each Block available from the Block Palette can be found in the documentation associated with that Block.

5.4.1 Types of Attributes

Four types of *attribute* are available, and a *block* may support zero or more of these depending on its purpose. These are summarised as follows:

Type	Description
Input Attribute	An Attribute identifying the source of data that will be received into a *block* via a *Sink Port* with the same name.
Output Attribute	An Attribute identifying the value (or stream of values) that will be transmitted out of a *block* via a *Source Port* with the same name.
Parameter Attribute	An Attribute that can be set by a user while configuring a Block, ultimately influencing the behavior of that Block.
Readback Attribute	An Attribute whose value is set automatically by a process within the execution environment. Readback attributes cannot be set manually via the user interface.

Attributes whose value can be set at design-time are denoted by a highlight below the attribute value field.

5.4.2 Obtaining Information About an Attribute

Information about an individual Attribute can be obtained by selecting the *information* icon associated with it from the *Block Information Panel*. This opens the Attribute Information Panel within the right-hand panel of the user interface.

For each Attribute the following information is displayed:

- The fully qualified path to the Attribute allowing it to be uniquely identified within the Design.
- Basic meta-data about the Attribute including it's type, a brief description of its purpose and whether it is a writeable Attribute.
- Details of the *Attribute state* associated with the Attribute, including severity of any issues and any corresponding message.
- Timestamp details showing when the Attribute was last updated.

Attribute meta-data and alarm state information is derived from pre-configured content provided within the underlying Block specification.

5.4.3 Setting a Block Attribute

Parameter, Input and Output Block attributes are set via the *Block Information Panel* associated with the Block you wish to configure.

The way in which an Attribute is set within the user inferface reflects the nature of that Attribute based on its defintion in the underlying Block specification. This can also provide clues on whether the Attribute is editable or not. The user interface provides the following approaches:

5.4.4 View/Edit Button

Provides the ability to modify a *complex Attribute*. Selecting the button opens configurable content in the central panel. Upon completion of changes the overall complex Attribute must be saved. If the Attribute is modifiable the text reads 'Edit', otherwise it reads 'View'

GET SCREENSHOT

5.4.5 Dropdown List

Provides the ability to select a value from a list of pre-defined values appropriate to the Attribute within is current Block context. Upon selection the Attribute value field updates to reflect the selected value.

GET SCREENSHOT

5.4.6 Text Input

Provides a 'free text' field accepting any alphanumeric string. Attributes that have been edited but not yet submitted are shown in the . Press the *Enter* key to submit the value. Upon successful submission the *edit* icon is replaced by the default *information* icon.

GET SCREENSHOT

5.4.7 Checkbox

Provides the option to switch on or switch off the action performed by the Attribute. If the checkbox is empty the Attribute is *off*.

GET SCREENHOST

To configure an Attribute:

1. Select the Block you wish to configure by clicking on it within the Layout Panel. The selected Block will be highlighted and the `Block Information Panel` associated with it displayed on the right-hand panel of the user interface.
2. Find the Attribute you wish to configure in the list of available Attributes.
3. Edit the Attribute value field as necessary based on the update process associated with the update approach described above.

Note: No data type validation is performed on manually entered values within the user interface. Validation is performed upon receipt by the backend server. If an invalid format is detected a *Warning* icon is presented in the user interface.

During the process of submitting a new Attribute value a *spinning* icon is displayed to the left of the modified Attribute. For more information on the process this represents see *Attribute Change Lifecycle*.

Upon successful submission the icon associated with the modified Attribute reverts to the *information* icon.

In case of submission failure an *attribute update error* icon is displayed next to the modified Attribute.

5.4.8 Exporting Attributes

The user interface presents a heirarchical view of the overall system, with one or more `Parent blocks` encapsulating increasingly deeper levels of your Design. By default at the top level of your `design` you will only see attributes associated with Parent blocks but it might be an underlying attribute within a Child Block that influences the behaviour of its parent. To mitigate this scenario every Parent Block provides the option to **Export** one or more Attributes from its children so they are displayed within the Parent Block.

In doing so it becomes possible to monitor, and potentially utilise, crucial Attributes implemented deep within a Design at increasingly abstracted levels of detail.

To specify an Attribute for export:

1. Identify the Attribute you wish to monitor outside the current layout level within the overall Deisgn. Note its source (in the format `BlockName.Attribute`).
2. Within the Parent Block describing the Layout select the **'View'** option associated with the 'Exports' Attribute.
3. When the Export Table is displayed select the first available blank row. If no blank rows are available select the option to add a new row.
4. In the 'Source' column select the drop-down menu option and find the Attribute you wish to export in the list of Attributes available.
5. In the 'Export' column enter the name of the Attribute as you would like it to appear when exported to its Parent Block. Leave the 'Export' field blank to display the default name of the Attribute. User specified display names must be specified in `camelCase` format, for example *myAttribute*.

Note: The `camelCase` naming convention is required to ensure an appropiate Attribute label can be generated in the Parent *`Block Information Panel`*.

Once successfully exported the Attribute appears within the 'Exported Attributes' section of the Parent *`Block Information Panel`* in the left-hand panel of the user interface.

Previously specified Attributes can be edited at any time within the Export Table following a similar process.

Any number of Attributes can be exported from Child Blocks to their overall Parent Block.

The order in which exported Attributes appear within their Parent Block mirrors the order in which they were added to the export specification. If you require a specific order to be displayed in the user interface:

1. With the Export Table displayed select the Edit icon associated with an existing Attribute or *Information* icon associated with a new Attribute. The information panel associated with the Attribute is displayed on the right-hand side.
2. To insert a new Attribute *above* the current one select the **'Insert row above'** option.
3. To insert a new Attribute *below* the current one select the **'Insert row below'** option.
4. On selecting the appropriate insert option a new row is added to the Export Table.
5. An existing Attribute can also be re-ordered by moving it up and down the list of attributes via the **'Move Up'** or **'Move Down'** option associated with it.

Attributes that have previously been exported can be removed from the Parent Block by deleting them from the Parent Block's export table. To remove an exported Attribute:

1. Identify the attribute to be removed.
2. Within the Parent Block containing the Attribute select the **'View'** option associated with the 'Export' Attribute.
3. Identify the line in the export table representing the Attribute to be removed.
4. Select the information icon assoicated with the Attribute. It's information panel is displayed on the right-hand side.
5. Select the **'Delete'** option associated with the **'Delete row'** field.

To complete the export process the export specification defined within the Export Table must be submitted for processing and recording within the overall system Design. To submit your export specification:

1. Select the **'Submit'** option at the bottom of the Export Table.
2. Refresh the Parent Block in the left-hand panel and confirm that the exported Attribute(s) have been promoted to the Parent Block or removed attributes are no longer visible.

Changes to the export specification can be discarded at any time throughout the modification process without impacting the currently recorded specification. To discard changes:

1. Select the **'Discard Changes'** option at the bottom of the Export Table.

5.4.9 Local vs. Server Parameter Attribute State

The underlying physcial hardware infrastructure described by your virtual representation is defined and configured based on the content of the Design specification saved behind the graphical representation you interact with on screen. Only when modified content is submitted and recorded to the Design specification is the change effected in physical hardware. It is therefore crucial to understand the difference between 'local' attribute state and 'server' attribute state, particularly for `Parameter Attributes` that can be modified directly within the user interface.

Local Attribute state represents the staus of a Parameter Attribute that has been modified within the User Inferface but not yet submitted for inclusion in the underlying Design specification. As such the modified value has no effect on the currently implemented hardware solution. Locally modified attributes are denoted by the 'edit' status icon next to the Attribute name within their `Block Information Panel`. A Parameter Attribute enters the 'local' state as soon as its incumbent value is changed in any way (including adding content to a previously empty Attibute value field) and will remain so until the 'Enter' key is pressed, triggering submission of content to the server. If the server detects an error in the variable content or format it will return an error and the variable will remain in 'local' state until the issue is resolved. Details of the mechanism of submitting modified content is described in the *Attribute Change Lifecycle* section below.

Once a Parameter Attribute has been successfully recorded it is said to be in the 'server' attribute state, denoting that it has been saved to an underlying information server used to host the Design specification. Attributes in 'server' state are reflected in the underlying hardware implementation and will be utilised by the system during exection of the hardware design. 'Server' state attributes are denoted by the 'information' status icon.

The following diagram shows the process involved in modifying a Parameter Attribute, mapping 'local' and 'server' states to the activities within it. Note also the inclusion of Attribute state icons as displayed in the user interface to denote the state of the Parameter Attribute as activities are completed.

Fig. 1: Attribute change lifecycle workflow

Tip: Do not confuse 'local' and 'server' Attribute state with a 'saved' Design. *Saving a Design* via a Parent Block 'Save' method does not result in all locally modified Attribute fields being saved to that Design. Only Attributes already in the 'server' state will be included when the overall Design is saved. Similarly, modified Attributes now in the 'server' state will not be stored permenantly until the overall Design has been saved.

5.4.10 Attribute Change Lifecycle

Attributes values modified via a `Block Information Panel` are recorded as part of the overall `design`. We refer to the combined submission and recording processes as a *'put'* action (as in 'we are putting the value in the attribute').

Once the 'put' is complete the Attribute value takes immediate effect, influencing any executing processes as appropriate from that point forward. If an error is detected during the 'put' process it is immediately abandonded and the nature of the error reflected back to the user interface.

The round-trip from submission of a value via the user interface to its utilisation in the execution environment takes a small but non-deterministic period of time while data is transferred, validated and ultimately recorded in the Design. Attribute modification cannot therefore be considered an atomic process.

Within the user interface the duration of this round-trip is represented by a *spinning* icon in place of the default information icon upon submission of the Attribute value. Once the change process is complete the spinning icon reverts to the default *information* icon. This reversion is the only reliable indication that a value has been recorded and is now being utilised.

Tip: Remember the three rules of Attribute change:

- Changing an Attribute value in the user interface has no impact on the underlying physical system until it has been 'put'.
- Once the 'put' process is complete the change takes immediate effect.
- Changes to an Attribute will not be stored permenantly unless the overall Design has been *saved*. Only those Attribute values that have been 'put' on the server will be recorded in the saved Design.

5.5 Complex Attributes

An Attribute associated with a Block may itself represent a collection of values which, when taken together, define the overall Attribute. For example, the Sequencer Block type contains a single Attribute defining the sequence of steps performed by underlying hardware when controlling motion of a motor.

The collection of values required by the Attribute are presented in the user interface as an Attribute Table. The template for the table is generated dynamically based on the specification of the Attribute within its Block. For details of utilising the table associated with a specific Attribute refer to the technical documentation of its Block.

An example of an Attribute Table for the 'Sequencer' Block associated with a 'PANDA' Parent Block is shown below:

5.5.1 Identifying Table Attributes

A Table Attribute can be identifed by the *View/Edit Button* associated with it. Selecting the button opens the Attribute Table within the central panel of the user interface.

5.5.2 Specifying Attribute Table Content

Upon opening an Attribute Table you are presented with details of the content of that Attribute, and the ability to define values. Like Attributes themselves these values may be selected from a list of pre-defined options, selectable enable/disable options, or text/numerical inputs.

After adding values the content of the table must be submitted for processing and recording within the overall system Design. To submit an Attribute Table:

1. Select the **'Submit'** option at the bottom of the Attribute Table.

Updates and changes within the table can be discarded at any time throughout the modification process without impacting the currently recorded specification. To discard changes:

1. Select the **'Discard Changes'** option at the bottom of the Attribute Table.

5.5.3 Static vs. Dynamic Attribute Tables

Depending on the specification of a table-based Attribute in its underlying Block the Attribute Table presented may be static or dynamic in nature.

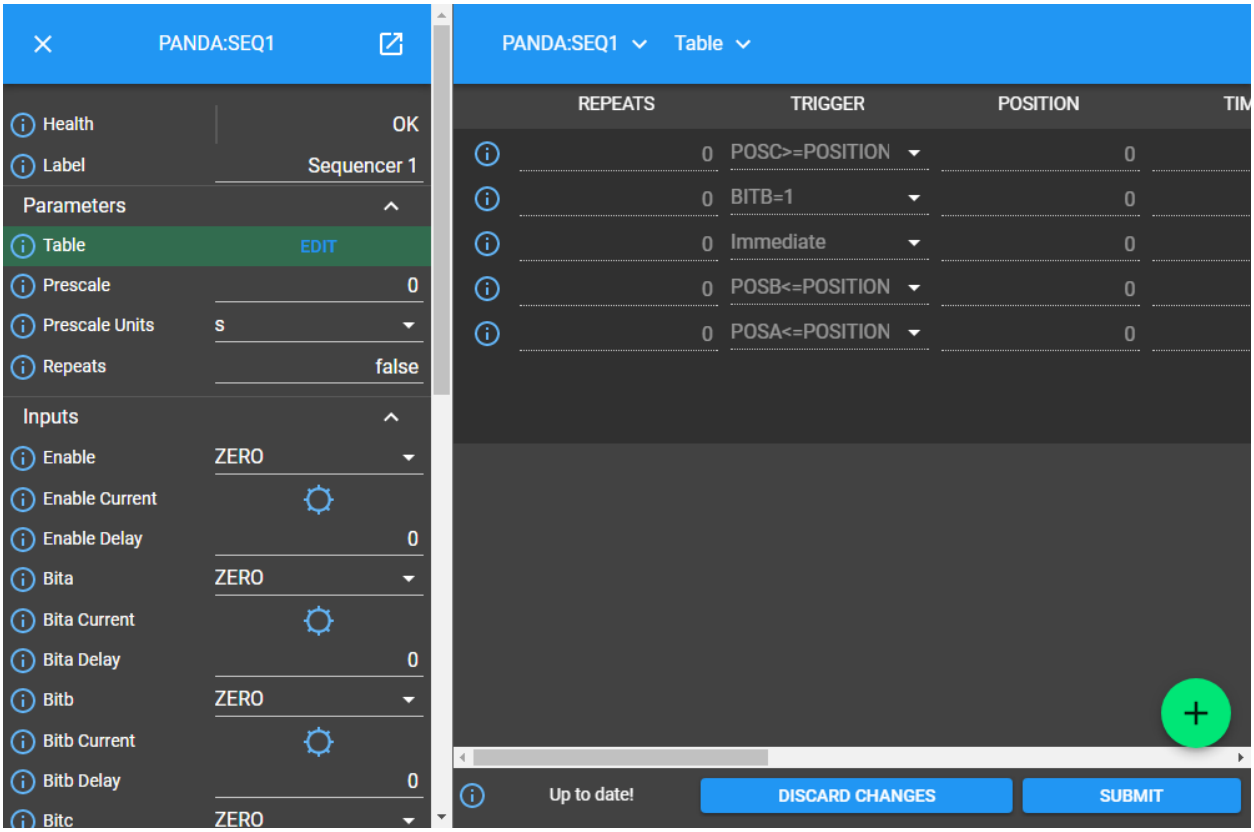

Fig. 2: Example Attribute Table associated with a complex Attribute

Static Attribute Tables contain a pre-defined number of columns and rows describing the information required for that Attribute. All fields must be completed in order to fully define the Attribute.

Dynamic Attribute Tables contain a pre-defined number of columns but allow for a varying number of rows. At least one row must be present to define the Attribute but typically more will be required to fully describe its behaviour.

New rows are added to the table in one of two ways:

- To add a new row to the end of the table select the **'Add'** option below the current last row entry. A new row is created.
- If the order in which table entries are specified is important (for example in the case of describing a sequence of activities), rows can be added before or after previously defined rows as follows:
    1. With the Attribute Table displayed select the 'edit' icon associated with an existing row entry or *information* icon associated with a new row. The information panel associated with the row is displayed on the right-hand side.
    2. To insert a new row *above* the current one select the **'Insert row above'** option.
    3. To insert a new row *below* the current one select the **'Insert row below'** option.
    4. An existing row can also be re-ordered by moving it up and down the list of attributes via the **'Move Up'** or **'Move Down'** option associated with it.

Rows that have been previously specified can be removed by deleting them from the Attribute Table. To remove a row:

1. Identify the row to be removed.
2. Select the *information* icon assoicated with the row. It's information panel is displayed on the right-hand side.
3. Select the **'Delete'** option associated with the 'Delete row' field.

5.6 Working with Block Methods

While Block `attributes` define the *behaviour* of a Block, `Methods` define the *actions* it can perform.

A Method in represented in the user inferface as a button, labelled with the name of the action that will be performed. The Method will only be executed if the button is pressed on the user interface.

A Method may require input parameters defining how the action is to be enacted. For example, the 'Save' Method associated with the Design within a `Parent Block` requires a single input parameter - the name of the file to which Design information is stored. Method parameters:

- Can be edited directly via the `Block Information Panel`.
- Exist in 'local' state until the button associated with the Method is pressed.
- Should be considered as properties of the Method they are associated with rather than entities in their own right. Method parameters are never recorded on the server or saved within the persistent Design specification.

A full list of the Methods available within each Block and details of their Method parameters can be found in the documentation defining that Block.

5.6.1 Obtaining information about Method execution

Selecting the 'Information' icon associated with a Block Method displays two sources of information relating to the Method:

- The right-hand panel displays details about the Method including a description of its purpose and the parameters it requires to execute successfully.

- The central panel shows a log recording each instance of Method execution within your current session. This includes the time of submission and completion, the status of that completion (e.g. success or failure) and any alarms associated with that status. Selecting the Method parameter name from the table header opens further information about that parameter in the 'Right-hand panel'.

5.7 Block Ports

If their purpose demands it Blocks are capable of *receiving* input information via one or more *Sink Ports* and *transmitting* information via one or more *Source Ports*.

A list of the Source ports and Sink ports associated with a Block can be found in the documentation for that Block.

To aid the design process ports are colour coded to denote the type of information they transmit (*Source Ports*) or receive (*Sink Port*). These are summarised below:

Port Type	Key
Boolean	Blue
Int32	Orange
Motor	Pink
NDArray	Brown

Transmission of information between a Source Port on one Block to a Sink Port on a second Block is achieved via a *link*. For further information about working with links see *Linking Blocks* below.

5.8 Linking Blocks

Blocks are connected to one another via *Links*. A Link joins a *Source Port* from one Block to a *Sink Port* on another. Both ports must be of the same type. The ports available to a Block and their specification are defined in the documentation for that Block.

5.8.1 Creating a Block Link

To create a Link between two blocks:

1. Select the *Source Port* or *Sink Port* representing one terminus of the link you wish to make by hovering over the Port on the Block. The Port will be temporarily highlighted.
2. Click the left mouse button and while holding it down drag the Link to the Port representing the other terminus of the link you wish to make. The target port will be temporarily highlighted.
3. Release the mouse button. If the *Link constraints* defined below have been respected the Link is displayed within the Design Layout.

Note: If an error occurs during the creation process details are displayed at the bottom of the Layout panel.

Tip: To confirm the Connection has been created correctly select the Link by clicking on it. The Link is highlighted to denote selection and the Link information panel opens in the right hand panel displaying the name of the *Source Port* and *Sink Port* associated with the Link.

5.8.2 Interrogating Link Attributes

A *link* does not possess attributes of its own, but selecting it within a *layout* displays information about its *source port* origin and *sink port* target in the right-hand panel of the user interface.

To interrogate the attributes associated with the Link you have created:

1. Hover over the Link of interest. The Link changes colour to denote that it may be selected.
2. Click the left mouse button to select the Link. A Link Information Panel open in the right-hand panel of the user interface.

Caution: It is possible to modify the Source and Sink associated with the Link from the Link Information Panel. Do so cautiously as this will change how blocks are connected in the overall Design without any acknowledgement that a change has occurred.

5.8.3 Removing a Link

If a *link* has been added to a *design* erroneously, or is no longer required within the current Design it can be removed in one of two ways:

1. *Hitting the 'Delete' or backspace key:*
    1. Hover over the Link of interest. The Link changes colour to denote that it may be selected.
    2. Click the left mouse button to select the Link. The Link is highlighted.
    3. Hit the *Delete* or *backspace* key on your keyboard. The Link is removed from the Design Layout.
2. *Via the Link Information Panel:*
    1. Hover over the Link of interest. The Link changes colour to denote that it may be selected.
    2. Click the left mouse button to select the Link. A Link Information Panel open in the right-hand panel of the user interface.
    3. Select the **'Delete'** button in the Link Information Panel. The Link is removed from the Design Layout.

5.8.4 Constraints When Using Links

Links are subject to the following constraints:

- A *sink port* can only accept a single Link.
- Multiple links can originate from a *source port*, connecting multiple Blocks to that Source Port.
- Links can only be used to connect a *source port* and a *sink port* of the same logical type (e.g. boolean, int32). Port types are specified in the documentation associated with the Block of interest, and colour coded within the Design Layout to aid identification of similarly typed ports.

5.9 Saving a Design

You can save your Design at any time during the creation or modification process, and we recommend you do so regularly.

To save a Design:

1. Navigate to the *Root Block* representing the highest level of the Design you wish to save.
2. Navigate to the 'Save' Attribute Group at the bottom of the left-hand panel. Expand it if necessary.
3. Enter a descriptive name for the Design in the 'Design' field. Note this will be used later to identify existing Designs available for use.

   Tip: To save your Design with the same name as the currently open Design leave the 'Filename' field blank.

4. Select the **'Save'** button. The information icon to the left of the button will spin to denote the save is in progess, returning to the information icon when the Design is saved.

Note: If an error is detected during the save process a red warning icon is displayed next to the button.

5.10 Opening an Existing Design

A *parent block* may facilitate multiple *designs*, each reflecting operation of that Block within different scenarios. Only a single Design can be utilised at any given time. By default this is the Design that is open at the time of system execution.

When a *parent block* is opened a list of all *Designs* within it is available via the 'Design' Attribute displayed in the left-hand panel. Selecting a pre-existing Design results in the Design being presented in the central Layout panel.

To open an existing Design:

1. Navigate to the *parent block* represening the highest level of the system you wish to use.
2. Navigate to the 'Design' Attribute and select the dropdown arrow to display the list of available Designs.
3. Select the Design you wish to use.
4. Select the *View/Edit Button* associated with the 'Layout' Attribute.

Tip: If no previously saved designs exist the 'Design' Attribute list will be empty.

CHAPTER 6

Monitoring Attribute Values

The user interface provides a near-realtime graphical representation of values associated with an *attribute* against time. Attribute values can be represented graphically as a *chart* or numerically as a *table*.

6.1 Working With Charts

Plotting data in a chart begins at the time the Attribute is selected (via the 'information' icon associated with it in its *block*). The data displayed is a 'for information only' representation of the Attribute's characteristics for the duration over which the plot remains open.

The User Interface renders every attribute as a chart. In doing so it handles the different Attribute formats available in different ways. In all cases the X axis represents time.

Attribute Format	Plot Type
Numerical data	Continuously recorded data is displayed as a traditional scatter plot with Attribute values on the Y axis.
Text data	Displayed in a similar way to discrete numberical data but in this case the Y axis representing each unique text string specified.
Boolean data	Representing on/off states within Attribute settings. The Y-axis represents the on or off setting.

6.1.1 Interactive Chart Functionality

Once displayed you have a number of tools at your disposal to support interaction with the chart. Hovering the mouse pointer over the chart displays the chart menu bar:

Fig. 1: Chart options bar

The following options are available:

Option	Description
Download Plot as PNG	Provides the ability to download a snapshot of the chart for use outside the user interface.
Edit in Chart Studio	Generates a snapshot of the data behind the plot and exports it to the online Chart Studio tool.
Zoom	Allows you to select a rectangular bounding box around a feature of interest and zoom in on it. The level of zoom dynamically adjusts according to the size of the bounding box (smaller box = higher zoom) with X and Y axes dynamically scaling to reflect the granularity of the resulting plot. Note that while zoomed updates to the selected area pause, instead presenting a snapshot at the time of zoom.
Pan	Provides the ability to pan the chart through horizontal and vertical axes. Note that unlike **Zoom** options panning does not pause automatic updates, thus when panning back through time the plot may appear blank.
Zoom in/out	similar to the manual **Zoom** option described above but automatically focussed on most recent data. As noted above zooming pauses automatic chart updates, presenting the area of zoom only.
Autoscale	Automatically scales the plot to include all data available since the Attribute was originally selected.
Home (Reset Axes)	Resets the chart back to its default scale after **Zoom** and **Pan** actions, restarting automatic updates of the chart as it does so.
Toggle Spike Lines	Click this option once to overlay vertical and horizontal guide lines when hovering over a data value. Click a second time to disable overlay.
Show closest data to hover	Displays the Attribute value for the closest data point to the current cursor location.
Compare data on hover	Displays *all* Attribute values at the time point represented by the cursor position (see notes on *Data Retrieval and Chart Update Frequency* below).

6.1.2 Enhanced Chart Interactivity

While the chart interface provides basic interactive exploration capabilities it is largely limited to panning and zooming. Further interactive capabilities are directly available via the '**Edit in Chart Studio**' option in the chart menu bar. Selecting this option automatically exports the data associated with the current on-screen chart and opens it in the online Chart Studio tool. Further information on the use of Chart Studio is available in its own online documentation.

Note: The external charting tool provided through the '**Edit in Chart Studio**' option is provided by a third party organisation. Functionality and support for this tool is not the responsibility of the MalcolmJS team.

6.1.3 Exporting a Chart For External Presentation

A snapshot of a chart can be taken at any time via the '**Download Plot as PNG**' option in the chart menu bar. The snapshot, representing the content displayed on screen at the time of selection, is automatically saved to the 'Downloads' folder of your local storage device.

6.1.4 Data Retrieval and Chart Update Frequency

1. Data is supplied to the chart interface at a frequency of up to 20Hz.

2. On screen graphics update at a pre-defined temporal frequency of 1 second and cannot therefore be regarded as a true realtime display. Each 1 second update plots *all* data supplied at 20Hz frequency during preceeding second.
3. On zooming into a plot the plot update process is paused to enable inspection of the area of interest. Automatic updates resume on returning to the default view state.

6.2 Working with Numerical Tables

A table present Attribute values in numerical form. Each row in the table represents a change in value, with each row presenting:

- The time at which the Attribute value changed.
- The value of the Attribute after the change.
- The status of the Attribute against parameterised alarm states.

For example:

The data displayed is a 'for information only' representation of the Attribute's characteristics for the duration over which the table remains open.

6.2.1 Data Retrieval and Table Update Frequency

1. Data is supplied to the table at a frequency of up to 20Hz.
2. On screen updates are set to a pre-defined temporal frequency of 1 second.
3. Each 1 second update includes *all* data supplied at 20Hz frequency during the preceeding second.
4. Newest data is presented at the *bottom* of the table.

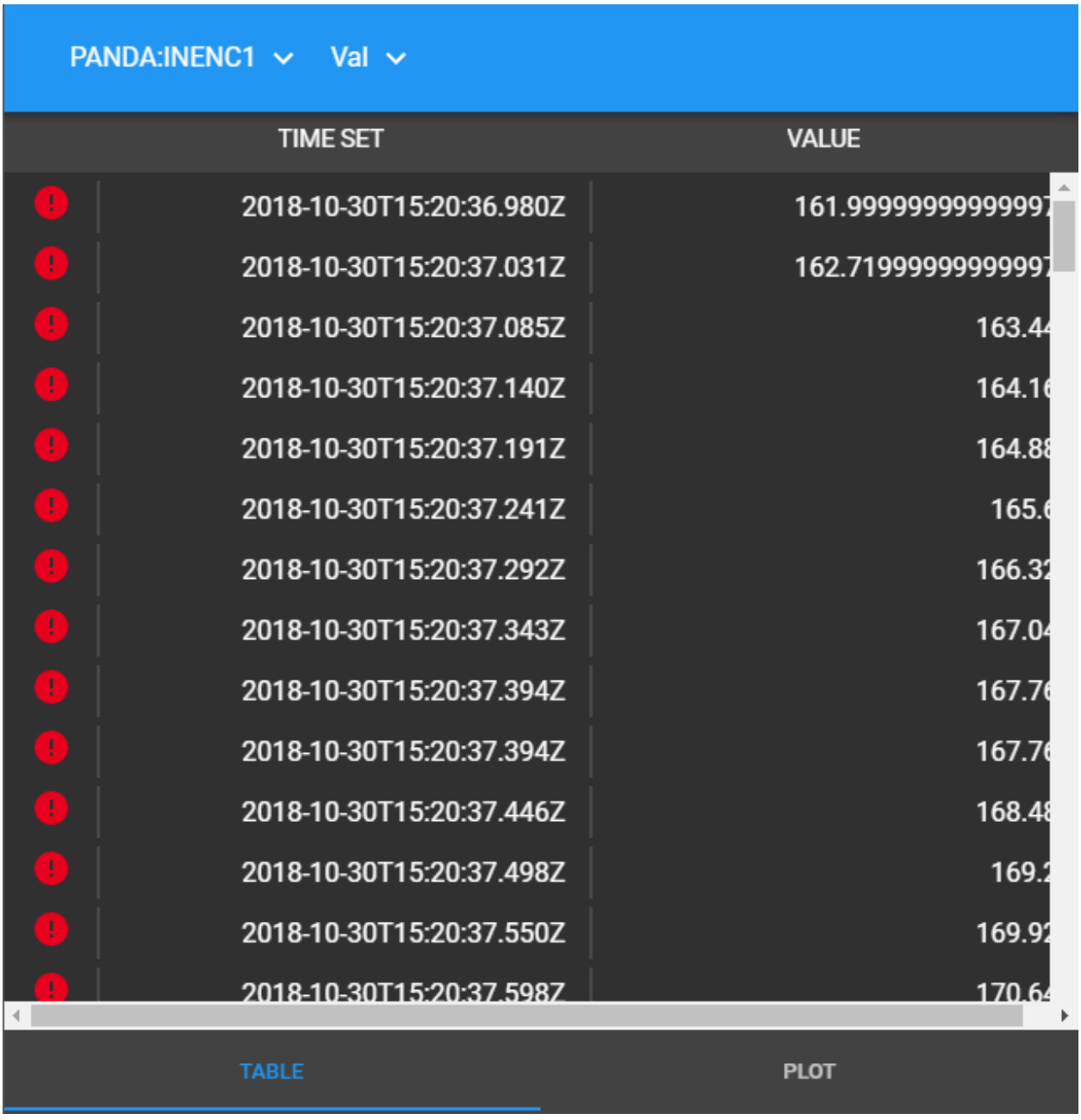

Fig. 2: Attribute values presented in tabular formal

CHAPTER 7

Understanding Attribute State

At any given time an Attribute can be in one of several states. State is dependent upon a range of factors including:

- The value associated with the Attribute.
- The pre-defined permissible operating range/threshold appropriate to the specification of the Attribute.
- The workflow that the Attribute is currently involved in.
- The context of the overall control system.

State information is presented in the user interface as a symbol displayed to the left of the Attribute name, with the following states represented:

7.1 Normal State

Data associated with the Attribute is within an acceptable operating theshold or range. For a `Parameter Attribute` this also means the value has been successfully *put* onto the underlying server.

7.2 Processing State

Data has been submitted to the underlying server or a request has been made to retrieve information from the underlying server. In both cases a process has been triggered and a response is a awaited.

GET IMAGE

7.3 Locally Edited State

Data within the user interface has been changed but not yet *put* to the underlying server. Consequently the local edit will have no effect on execution of the system and will not be saved as part of the *Design*.

7.4 Update Error State

The Attribute value *put* to the underlying server has not been accepted. This typically occurs because the value has failed the validation test associated with the Attibute as defined in its Block specification.

GET IMAGE

7.5 Warning State

An issue has been detected and requires investigation. This state is typically triggered when an *Input Attribute*, *Output Attribute* or *Readback Attribute* data is deemed to be outside normal operating parameters but is still considered acceptable.

7.6 Error State

An issue has been detected resulting in an error being reported by the underlying server. This state is typically triggered when *Input Attribute*, *Output Attribute* or *Readback Attribute* data is deemed to be outside acceptable operating conditions and immediate action is recommended.

7.7 Invalid State

The overall Block context has changed since the user interface was last accessed. Data displayed may no longer be accurate or consistent with the current *design*

7.8 Disconnected State

Communication with the Block hosting the Attribute has been lost by the underlying server. Immediate investigation is recommended.

GET IMAGE

Note: Operating ranges and threshold values are not specified directly within the user interface but instead via configuration of individual Attributes in the underlying Block Specification. These settings are then reflected into the User Interface. See specific Block documentation for additional information.

7.9 Presenting Status Information

Within Block Information Panels (presented in either the left-hand or right-hand panels) status information is displayed to the left of each Attribute via the corresponding state icon (see table above).

When presenting historical Attribute value data via the *Attribute value table* view corresponding icons are displayed against each row of data.

When presenting historical Attribute value data via the *Attribute Chart* view the line colour denotes the alarm state.

CHAPTER 8

Architecture

8.1 Context

Malcolm provides a higher level abstraction over EPICS for monitoring and controlling devices (e.g. motor controllers or area detectors). **MalcolmJS** is a user interface for Malcolm that allows blocks to be wired together and more complex behaviours to be programmed into the hardware (e.g. custom scan patterns).

Fig. 1: MalcolmJS context within its wider environment

The overall use case for MalcolmJS is that engineers will be using it to configure PANDA boxes from a laptop, this will involve wiring blocks together, tweaking inputs and monitoring outputs. This may involve working with physical hardware so as much information needs to be available on the screen as possible whilst minimising scrolling (this helps to reduce context switching).

The other minor use case is to allow engineers to view block attributes on their phone, this will allow them to monitor outputs and tweak inputs without having to go back to their laptop.

8.2 Containers

The container view for MalcolmJS breaks down in to the main user interface categories as well as the interface for interacting with Malcolm.

Fig. 2: MalcolmJS container view

Malcolm Interface:- The malcolm interface will allow MalcolmJS to be decoupled from Malcolm by isolating the code that interacts with the socket layer. This will also be the point at which we can integrate the Malcolm information with the Redux state that can be used by the presentation components.

The secondary reason for doing this is so that the Malcolm related code could be released as a library for others to create their own React dashboards from Malcolm data.

Block Pane:- This presentational layer container will contain all the React components for displays information about the details for a block, i.e. information about the attributes it has. There will be quite a large collection of components in here for the various attributes, e.g. number inputs, LED status, combo selections, etc. The attributes will roughly break down into summary information, parameters, views and methods.

View Pane:- When the user selects a view options, e.g. layout, table, etc. from the block pane then MalcolmJS will populate a central panel with the details about that view. For example, if they select layout then they will get a flow diagram showing the various blocks that are currently connected.

Navigation:- The navigation container has all the presentational components for showing how far down the block tree a user is and allow them to select different levels as well as different views easily from one place.

Attribute Tracking:- Currently there is no functionality in Malcolm to track the history of a numeric attribute over time, therefore to display a graph of that attribute we need to track its values over the time that the browser is open.

Connection Monitoring:- Malcolm works on a subscribe/unsubscribe model - in order to clean up connections we need to have sufficient monitoring. This will also making it easier to have a status indicator for connections as well as handling connection errors.

One of the best ways to think about the high level structure for MalcolmJS is to consider the high level user interface mock ups. They show the main sections of the interface and thus how the application breaks down into containers.

Fig. 3: MalcolmJS desktop view

The user interface breaks down in to three main components:

- the parent block (the left panel)
- the layout view (the central panel)
- the child block (the right panel)

Users typically navigate to a top level block and then navigate down the block structure depending on their needs. They can change the view in the middle panel for any that are available in the parents block details panel in the views section.

Once they click on a block in the central panel then it will display information about that child block in the right hand panel. This information will include attributes, and may additionally include views and methods of its own. If the user clicks on the view button for the child's layout attribute then the contents of the child panel become the contents of the parent panel and the user can navigate one more level down the tree.

On a multi-monitor system it can be useful to keep track of block details for various blocks in the tree, this is why there is a pop-out button. The user can choose to show the attributes for a particular block in a standalone window. It it worth noting that this is also what the mobile view would display.

Fig. 4: MalcolmJS mobile view

The mobile view is simply intended to give engineers a quick view on the information being output from devices whilst adjusting them, it is not meant to be a full MalcolmJS environment. Never-the-less, they can click on an attribute for more information and it will close the draw for the attributes and show graphs/tables/etc. in the central panel just like the desktop version.

8.3 Components

- **Malcolm Interface** The Malcolm Interface consists of all the parts needed to integrate it in to the Redux life-cycle. It is better described by the diagram below:

Fig. 5: MalcolmJS component view

Fig. 6: Malcolm redux integration

By ensuring that components need to create actions using the Malcolm Action Creators and processing data through the Malcolm reducer ensures there is a consistent data model for components to use. Messages intended for the socket are intercepted in the middleware and responses from the socket are injected back into the cycle as a new action in the Malcolm Socket handlers.

- **Attribute Details** All of the information for rendering the MalcolmJS interface is returned as part of the web-socket response, part of the metadata describes the widget to show for the attribute in question. This means that we are going to need a collection of components to represent these widgets, e.g. a numerical label, an LED indicator, etc.

  The attributes break down in to three main areas; parameters, views and methods. Views contain buttons to populate the central panel with information about that view, e.g. the layout, a table of information, etc. The method components lists the methods on the block and allows the user to click on a button to call them as well as displaying what other inputs from the block are used.

- **View Details** The view details provides more information about the parent block being viewed, while there are component and table views, the main view is the layout view. The layout view shows how the current block is wired to child blocks, it allows new connections to be made, and allows a user to add new child blocks.

  The user can drag blocks around in the layout view as well as connect output ports to input ports.

8.4 Deployment View

The primary deployment scenario is where MalcolmJS is packaged up with Malcolm and served from a web server inside Malcolm on the same PANDA box. The websocket connection is to the same server, so there should be no cross-origin issues. During development we'll introduce a proxy when connecting to a test instance.

Fig. 7: Serving MalcolmJS from the same server as Malcolm

8.5 Technologies

The technology stack selection has been based on the principles of:

- Making use of **Open Source Software**
- Selecting modern, well supported frameworks to ensure long term sustainability (within the bounds of the previous principle)
- Fitting in with Diamond processes where there are clearly defined technology choices for consistency

8.5.1 By Component

- **MalcolmJS redux components** A set of components for handling socket communication with Malcolm that intercepts messages intended for Malcolm and sends them, as well as injecting responses back into the

Redux one-way data flow.

- **MalcolmJS attribute components** A set of presentation only react components that could be distributed as an npm package for other people to develop MalcolmJS dashboards with.
- **Remaining MalcolmJS presentation components** The other container components needed to layout the MalcolmJS site and wire the presentational compoenents up to the MalcolmJS redux components.

8.5.2 Tools

- Create React App for the initial site template
- Jest unit testing and coverage
- Cypress end-to-end testing
- React Storybook
- React Storybook Info addon
- ESLint with the AirBnB rule set
- Prettier code styling
- Husky for pre-commit hooks
- Travis for continuous integration
- Github releases for uploading build artifacts back to Github
- Waffle.io for Agile tracking
- Codecov for tracking code coverage
- Github for version control and issue management
- Sphinx for document building

8.5.3 Languages

- Javascript
- reStructuredText
- bash
- yaml

8.5.4 Frameworks

- React
- Redux
- React-Router
- Redux-thunk
- Socket.io
- Material UI

8.6 Quality

8.6.1 Coding Standards

Static code analysis is done by running ESLint against the code with the AirBnB rule set. Code styling is done with Prettier to avoid debates on code styling. These are both enforced as pre-commit hooks with the `--fix` option turned on so as much as possible is automatically fixed. This ensures the static code analysis violations remain at zero unless explicitly ignored.

Unit testing is all done with Jest which provides code coverage information using the `--coverage` flag, this generates an LCOV report with all the coverage information. The coverage information is tracked on CodeCov, where during this phase of development, all the information is on the version 1 branch.

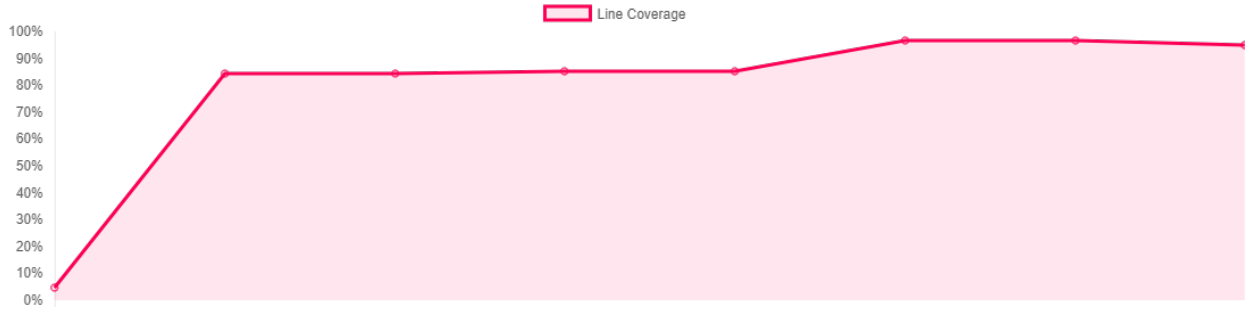

Fig. 8: Unit test coverage for the last 6 months on branch `version1`

All code including documentation should be peer-reviewed, as such all work must be done on a branch and a pull request created in order to review the code before merging into the main branch (during this phase of the project it is `version1`).

Branches should use the naming convention `feature/{descriptive name}-#{issue number}`. By adding the issue number to the end it allows the waffle.io integrations to automatically move cards on the agile boards when particular activities are in progress/completed.

When creating a pull request you should also add the comment

```
connect to #{issue number}
```

to the description to link the pull request to the issue.

Pull requests are gated so the automated build in Travis needs to succeed and the reviewer should also take note of the impact on code coverage. The aim is to maintain a high level of coverage (e.g. over 90% is good) but whilst being pragmatic, it is not an exercise in getting a high number but rather making sure the new code is sufficiently tested for maintainability.

8.6.2 Security

There are no current security restrictions on MalcolmJS as it has to be able to communicate with a Malcolm enabled device which are all inside the Diamond internal network and so MalcolmJS will also have to be accessed from inside the network. Once inside the network anyone is allowed the configure the Malcolm settings.

8.6.3 Testing

As much effort as possible should be made to automate unit, integration and system testing. MalcolmJS will use as much unit testing as possible, as well as running end-to-end tests against a test server that mimics the socket responses. This should mean very few system tests are needed as we can expand the socket responses of the test server to cover these cases. Where system tests are needed then they will need to be done manually as they will need to be run against an actual PANDA box but could still be based off scripted tests (e.g. using cypress).

Given we are developing a website, usability testing will also be important so we should plan to get some engineers to do some testing and gather their feedback.

One of the big issues with versions prior to version 1 was performance and the time taken to re-render updates. We should also put additional effort into performance testing to make sure the page is at least usable (i.e. it doesn't need to be lightning fast but shouldn't freeze up, it should at least indicate to the user it is still responding but could be waiting for a response).

8.6.4 Attitude Towards Bugs and Technical Debt

Bugs severely affect the maintainability of the system, as far as is practical we should seek to have a zero bug system - this means that when a bug is identified then it gets prioritised to the top of the backlog and dealt with as soon as possible.

This approach should ensure that the number of bugs doesn't become un-manageable and then ignored because they seem unsolvable.

The same approach should be employed with technical debt, we should seek to minimise technical debt so the system is more maintainable. This should allow us to develop faster because we aren't weaving new features into an existing fragile system. The one caveat with this is that a level of pragmatism needs to be taken depending on the timescales and progress needed for the project, but remembering that every un-addressed issue will slow the project down at some point in the future.

CHAPTER 9

Architecture Background

9.1 Architectural Constraints

The following constraints have been placed on the system

Constraint	Architectural impacts
The system must only use open source technologies	The technologies chosen should be open source and suitable for deployment in a Linux production environment to ensure that there are not any ongoing licensing costs.
The system must communicate with Malcolm using websockets	The PANDA boxes that MalcolmJS will communicate with are set up with a socket server. This is because it can take time to perform an action on the equipment as well as being able to subscribe to a stream of updates
The development will be done using Scrum	Manual testing effort is limited so we should make as much use of automated tests as possible
The system needs to be fully contained and not rely on any internet connection	MalcolmJS will be deployed alongside malcolm and served from the PANDA box, this may mean the client has no internet connection (e.g. if a laptop is plugged in directly to the equipment), therefore all dependencies (e.g. fonts) need to be part of the deployment package.

9.2 System Qualities

Quality	Note
Scalability	There is no scalability requirements in the normal sense as the system will be deployed from a single PANDA box and have very few clients attaching to it. Where it does need to be scalable is in the number of socket messages it can track, there may be a lot of attributes that are being watched and thus lots of Redux state updates, we need to ensure the the site refreshes reasonably and remains usable.
Security	MalcolmJS currently has no security constraints as the site will be deployed from a PANDA box inside the Diamond network. Even when externally hosted, all of the information comes from the websocket connection in Malcolm (which is inside the Diamond network). There are no user access requirements as it is open to anyone inside Diamond.
Testability	Given MalcolmJS is a react site revolving around a component based approach with a clear separation between the state and the presentation, it is important to ensure that each is independently testable to ensure that developers can work independently. This is also important considered that the malcolm related code may be split out into a separate library at a later time.
Usability	Usability is very important for MalcolmJS because it is a user interface for Malcolm. The site should be intuitive to learn with indicators/hints to what operations are happening (e.g. waiting for a socket reply). One key quality here is that the user should not have to scroll around for a lot of data; one usage scenario is when an engineer is configuring equipment and has MalcolmJS running on their laptop, they need to see the Malcolm output without switching between the equipment and the laptop.
Maintainability	MalcolmJS will continue to be developed beyond this intial stage, possibly by the open source community or Diamond partners. Given this, MalcolmJS should be developed against all of the appropriate coding standards and that ease of maintenance is considered during development.

9.3 Engineering Principles

Principle	Rational	Architectural impacts
Separation of concerns	The application should be developed with clear boundaries between different areas	The system will use the architectural layering defined to separate the data model (Malcolm information) from the presentation
Use of frameworks and libraries	The use of standard frameworks and libraries, where appropriate, will enable efficient development and ensure that industry best practice is followed.	Where possible, third party libraries and frameworks should be used to facilitate development. They should be carefully evaluated prior to use to ensure that they are under active development, have clear documentation and a large community.
Error handling	The system needs to ensure that unexpected errors do not stop the system working entirely, and should make the user aware of what has gone wrong and how to report it	Each component will use a consistent approach to error handling.
Logging	All components should implement logging appropriate to the components function.	Logging should be considered as a first class concern for all components since they will require different strategies. For example, it will be important to know how the malcolm communications fail in case it needs hardware support, whereas react errors are less important.
Automated release	The system will need to be regularly and and easily deployed to reduce maintenance	MalcolmJS should be designed to be deployed as a single package with releases being published to github automatically when tagged.

9.4 Architectural Styles

Style	Description
Modular architecture	So that the Malcolm related logic can be kept in one place we should adopt a modular design for the data related operations as well as the presentational components so we can later choose to release the widgets as a component library for others to make Malcolm dashboards.
One way data flow	As in now standard in React sites we will use a one way data flow paradigm to help separation of concerns, testability and performance

CHAPTER 10

Architectural Decision Record

1. Malcolm protocol related code should be grouped together

Status: Active

Context: Other groups may wish to generate their own dashboards from Malcolm outputs and so we may want to release the Malcolm related code as an `npm` package

Decision: Any communication with Malcolm via the socket should be isolated from the rest of the application, and code related to that should be stored together.

Consequences: A single redux reducer should be used for processing Malcolm results and adding it to the Redux store, presentational components should make use of that information from the store. Malcolm related actions should be built with action creators that are stored with the Malcolm code and components should import those action creators rather than building actions themselves.

2. Attribute components should be grouped together

Status: Active

Context: Other groups may wish to use the MalcolmJS widgets in their dashboards and so we may want to release the attribute components as an `npm` package (probably part of the same package as the Malcolm related code).

Decision: Attribute compoenents should be grouped together in the codebase and not rely on other parts of the code.

Consequences: Attribute components should be as stateless as possible (see below).

3. Attribute components should be as stateless as possible

Status: Active

Context: Other groups may wish to use the MalcolmJS widgets in their dashboards and so we may want to release the attribute components as an `npm` package (probably part of the same package as the Malcolm related code).

Decision: Attribute components should be isolated from the rest of the code base and all interactions should be done by passing in props and actions.

Consequences: Attribute compoenents should be developed with Storybook to ensure they are properly decoupled from the rest of the application.

CHAPTER 11

Sequence Diagrams

11.1 Connecting to the websocket

Fig. 1: Connecting to the websocket

11.2 Getting Block Details

Fig. 2: How the block information is gathered

11.3 Loading the layout

Fig. 3: Building the layout view

11.4 Running a method

Fig. 4: Running a method

CHAPTER 12

Setting up a Development Environment

12.1 Code Development

Requirements:

- node >= 8.5.0

To develop MalcolmJS you need a stable version of node (>=8.5.0 but it was developed against 8.9.0) and your OS needs to be able to run Chromium (for the end-to-end tests) - this means RHEL 7 or above but Windows has also been used as a development environment.

You will also need a git client of some form and then you can clone the code from the MalcolmJS github page

Navigate to the root of the repository and run

```
npm install
```

This will install all dependencies and development dependencies to run and develop MalcolmJS.

The available commands for developing with are listed in `package.json` as well as being documented in the *Maintenance* chapter; all commands are runnable from the shell using node so should be cross platform.

However, the main commands needed to develop are:

```
npm start
npm run storybook
npm test
npm run e2e
```

While in theory you can develop the code in any text editor, it is recommended that some form of IDE is used with syntax support for JavaScript and React. Both Webstorm and Visual Studio Code were used during this latest phase of development.

12.2 Documentation Development

The documentation is all in reStructuredText. The documentation can be edited in any text editor but some editors provide extra support for viewing a preview of the result (e.g. Sublime Text).

The final documentation is built for Read the Docs using Sphinx.

12.3 Setting up a virtual environment

12.3.1 In a Diamond environme

[To be written]

12.3.2 In an environment you control

To build the documentation locally you need to have a virtual environment set up:

```
pip install virtualenv
```

To make it easier to work with virtual environments you should install `virtualenvwrapper` on linux or `virtualenvwrapper-win` on Windows

```
pip install virtualenvwrapper
```

or on windows

```
pip install virtualenvwrapper-win
```

Then make a new virtual environment:

```
mkvirtualenv malcolmjs-docs
```

Change in to the root of the code and connect the codebase with the virtual environment:

```
cd {root of code base e.g. C:\code\malcolmjs}
setproject dir .
```

Install the dependencies:

```
pip install -r ./docs/source/requirements.txt
```

To deactivate the environment run:

```
deactivate
```

12.4 Running a docs build

To drop into the virtual env to run the build

```
workon malcolmjs-docs
npm run build-docs
```

The same build step can be done by running the `Makefile` in the root of the repo or manually running the command `sphinx-build -b html docs/source docs/build/html`

You can then open `.\docs\build\html\index.html` in a browser to view the results.

CHAPTER 13

Developer Workflow

13.1 Workflow

1. Select a story from the waffle board.
2. Pull the latest version of the `version1` branch
3. Create a branch from there called `feature/{insert general name of story}-#{story number}`
4. Put the story into `In Progress` on the waffle board
5. Repeatedly during development:
    1. Develop code/write tests
    2. Ensure all tests pass
    3. Run linter and fix issues
    4. Commit
    5. (optionally) push to `origin`
6. Merge `origin/version1` into your feature branch.
7. Run unit tests
8. Check impact on coverage
9. Run e2e tests
10. Push to `origin`
11. Make a pull request
12. Put the story into `Review` on the waffle board
13. Get the pull request reviewed
14. Wait for the automated build on Travis

15. Once a pull request is approved then it can be merged back into `origin/version1`
16. Move the story to `Review complete`
17. Wait for the merge build to complete on Travis
18. Move the story to `Done`

13.2 Story State Transitions

13.3 Branching Strategy

13.4 Code Development and Testing

13.5 Preparing for a Pull Request

13.6 Pull Request Procedure

When a branch is ready to be merged back in then you should create a pull request in GitHub. The description should contain:

- A general description of the changes in the pull request
- A link for waffle to connect the PR to the issue (`connect to #{issue number}`)
- A method for testing the changes, particularly if they are visual in nature (e.g. describe which StoryBook story to look at or which url to visit). You may also want to consider adding a screenshot of the changes.

If this is your first time reviewing then you will need a development environment set up first, see *Setting up a Development Environment*

Before approving, a reviewer should:

- review the code
- ensure the build passes
- assess the impact on code coverage
- run the method for testing as described by the developer as assess the visual impact of the changes

13.7 Monitoring Progress in a Sprint

CHAPTER 14

Testing Strategy

14.1 Scope

In-Scope:

- Functional and non-functional user story testing
- Unit, integration, and system testing for the developed code
- UI testing for usability and performance scenarios
- User Acceptance testing against a PandA simulator
- Deployment testing against a PandA
- **Compatibility testing on the below-listed browsers and devices**
  - Firefox (Must have)
  - Chrome, (Should have)

Out-of-Scope:

- Security testing
- Penetration testing
- Browsers that are not included in the in-scope list.
- Touch interactions (at least at this point of the project)

14.2 Strategy

The test strategy for each sprint is split into development testing, system testing, and user acceptance testing; they are carried out by various members of the team in different project phases.

Builds will progress through the various stages when they succeed at all previous levels. For example, builds passing all unit, integration and system tests will be deployed to a PandA for user acceptance tests, etc.

Fig. 1: Progression through the testing stages

Gnerally the number of tests and time for execution should follow the pyramid of testings shown in the diagram below and explained in this blob post by Uncle Bob https://codingjourneyman.com/2014/09/24/the-clean-coder-testing-strategies/

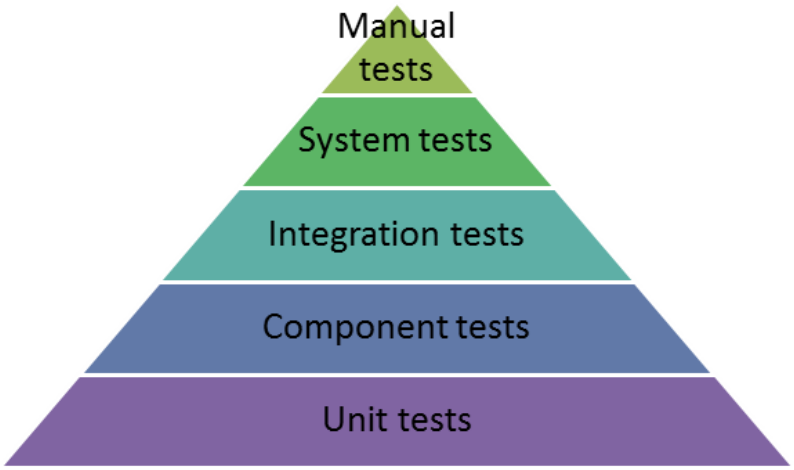

Fig. 2: Number of tests for each stage of testing

14.3 Development Testing

Development testing is managed and maintained by the developers. A Test Driven Development (TDD) approach should be practised by developers for every story developed. Developers play a key role in the testing process in addition to the tester; unit and integration testing are the sole responsibility of developers and an integral part of the development activity.

- Automated Unit and Integration tests will be created and monitored around all the features where business logic materialises.
- Formal and informal peer reviews are performed throughout the project to ensure coding standards and good practice guidelines are followed, primarily through pull requests.

14.4 System Testing

Without a dedicated tester we are aiming to create a set of automated regression tests that run using the production build. These automated tests are written using the Cypress test framework.

One advantage of automating the end-to-end (e2e) tests is that they can be run as part of every build rather than waiting for a deployment of the system.

Part of the system testing will also be **unscripted testing**, whereby during development or whilst looking over the system general; if we spot a bug then it should be reported.

14.5 Defect Management

If the story is still open then the developer will be informed of the defect and will be expected to fix it before the story is closed. If the bug is found after a story is closed then a new bug should be recorded in the backlog and prioritised accordingly.

14.6 User Acceptance Testing

In Sprint review meetings, the project team will demonstrate the user story implementation in the Test environment. Business users will be talked through the acceptance criteria for each committed story in the sprint using the latest release build. Post-review meeting, the business users may perform their user acceptance testing by downloading and running MalcolmJS to evaluate the functionality and usability of the application. Any issue or suggestions identified in the user evaluation period shall be communicated to the development team for further action; most likely story creation and prioritisation.

14.7 Deployment Testing

At periodic intervals we will deploy to a real PandA to evaluate the performance of MalcolmJS. Before MalcolmJS is released to users, the project team will carry out non-destructive sanity testing to ensure build correctness.

CHAPTER 15

Code Structure

This section seeks to give a description of the code structure so new developers can get aquainted with how the main components of the user interface link to the folders in the code base.

The diagram below shows a general view of the MalcolmJS interface:

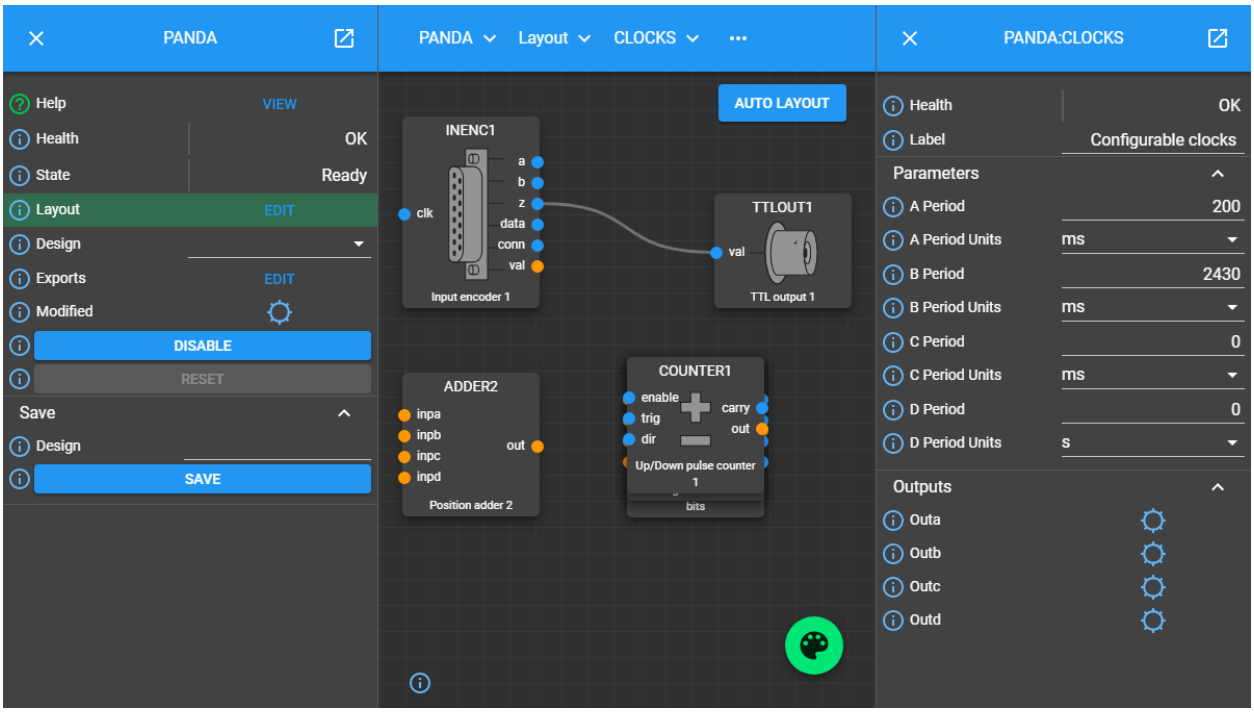

Fig. 1: An example of the MalcolmJS interface

The main components of this page are the **block details** on the left and right which are held in a **drawer container**, the **nav bar** at the top of the page and the **layout view** in the middle pane. We will explore this components in more detail in the following sections.

15.1 How to link UI components to backend actions/reducers

In order to understand how the state and business logic links to the UI components we first need to revisit how data flows in a React-Redux system.

The main data contruct of a Redux system is the Store that contains a state (see `src/testState.utilities.js` for the store structure), the state defines the UI and what is visible. A user then interacts with the UI and triggers actions which are sent to reducers, these reducers then update the state in the store.

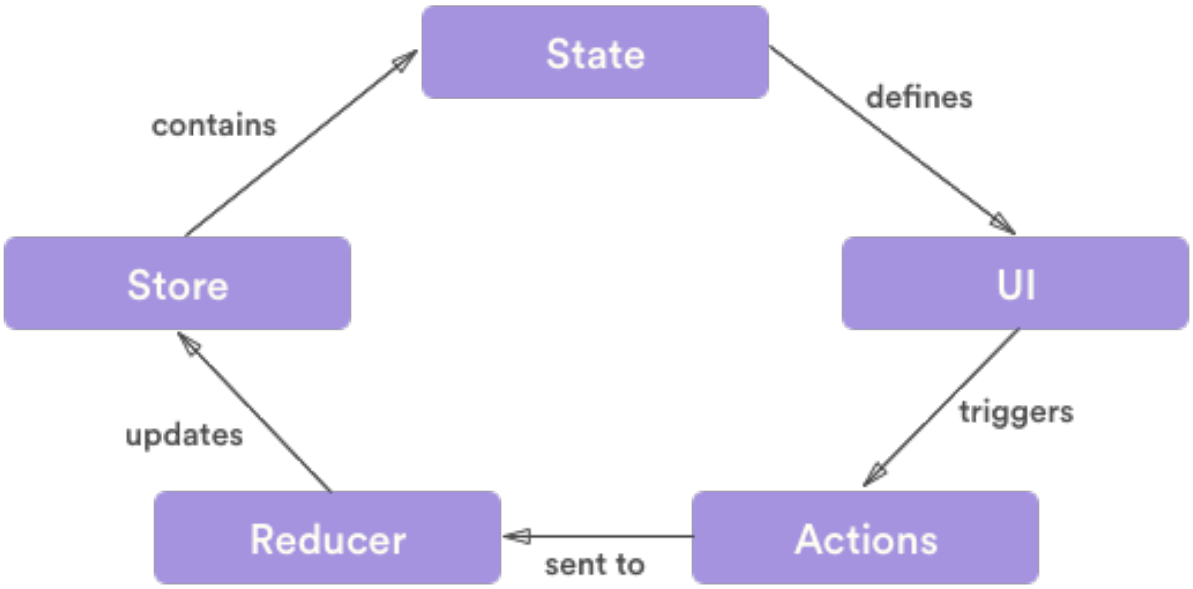

Fig. 2: How data flows round a react-redux system

In order to find out what updates to the store happen because of interactions with the UI it is important to follow this one-way data flow round the loop. From a component you will be able to find the actions that it fires (typically within the `mapDispatchToProps` method), you'll then look for the reducer that handles the type in that action, from that reducer you can see the changes that are made to the state.

Each action should be of the form:

```
const actionType = 'A_CONSTANT_STRING_FOR_THE_ACTION';

const action = {
    type: actionType,
    payload: {
        someContents: {}
    }
};

const reducer = (state, action) => {
    switch (action.type) {
        case actionType:
            // do something with the actionType action
    }
};
```

It is important that the action does not contain a direct string for the type, instead it should be referenced from a constant. This then enables IDEs to search for all usages and it will immediately highlight the reducers that the action is handled in.

Another way of linking the UI components to the actions and reducers is by the naming convention, general a layout component will have a matching set of actions and a reducer; for example, the `layout` component has actions in `layout.actions.js` and a reducer in `layout.reducer.js`.

15.2 Drawer Container

The main container component for MalcolmJS is the drawer container, this component holds the drawers on either side and the main panel in the middle behind the drawers.

The drawer container itself does not have much functionality, it simply holds other components.

15.3 Block Details

The block details component is again another container component but this time specifically focused on an individual block. The block details component pulls together all the attributes on a block and works out if they should be displayed in groups, as methods, or if they are a top attribute.

15.4 Attribute Details

Each attribute is displayed in an attribute details component, this component is responsible for working out which widget to use for the given attribute.

The main structure of the attribute details component is the alarm status, the label and then the widget for that attribute.

15.5 Methods

Methods are special attributes that get displayed with their own component because of the extra complexity around handling their state for the inputs as well as tracking the outputs returned by Malcolm.

The method component generally consists of a set of inputs (again selected using the same mechanism as in the attribute details) and then a run button. If the method returns outputs then they will be shown below the run button when they are received.

15.6 NavBar

The main method of navigating around in MalcolmJS is through the NavBar, this component along the top of the page is responsible for displaying the tree of blocks/attributes/info that is currently being viewed; each of those nodes of the tree are represented by a NavControl.

The left most drop down is special because it contains the top level list of blocks from the `.blocks` output from Malcolm.

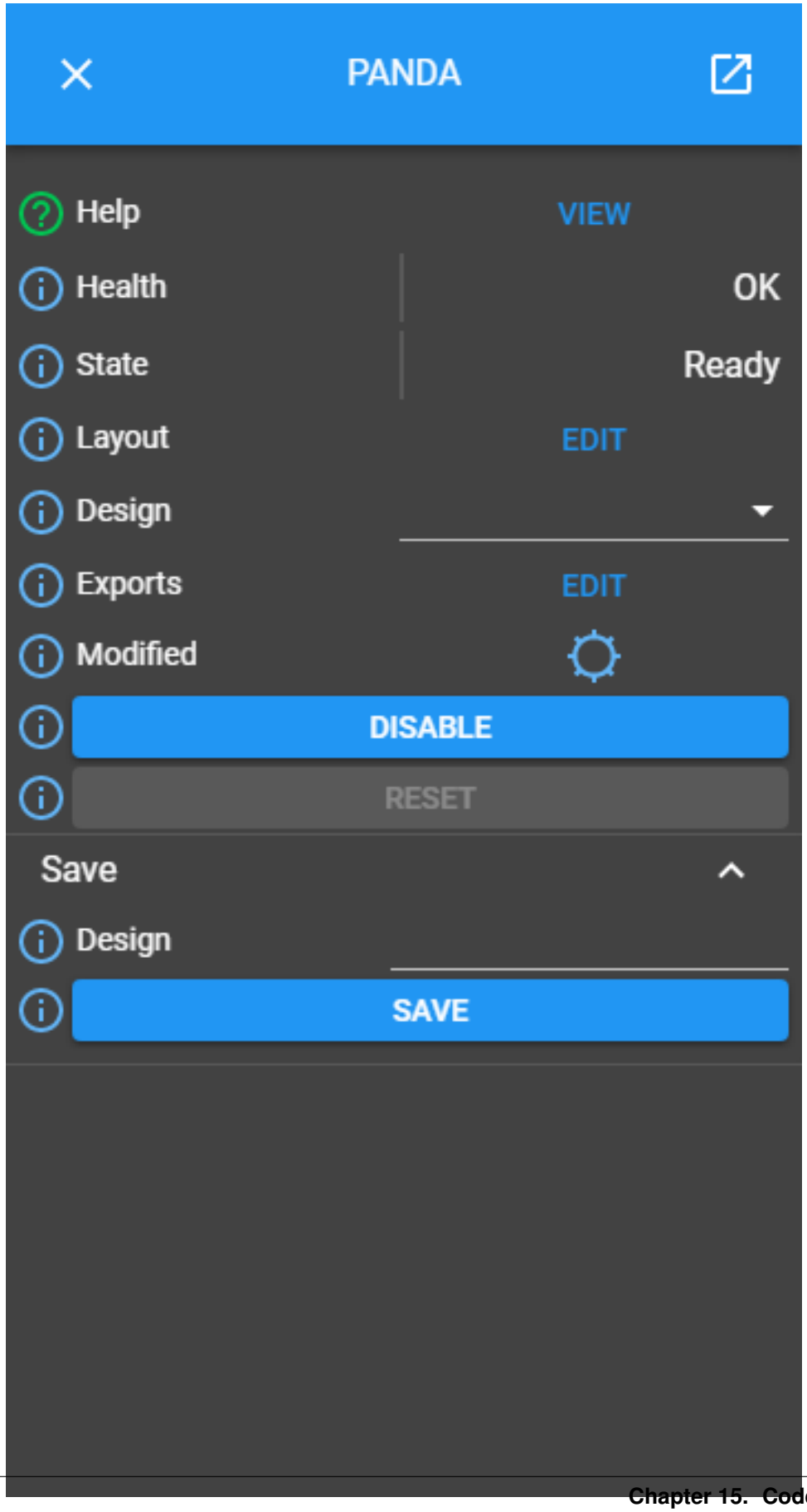

Fig. 3: The block details panel in the main UI

Fig. 4: The attribute details inside the block details

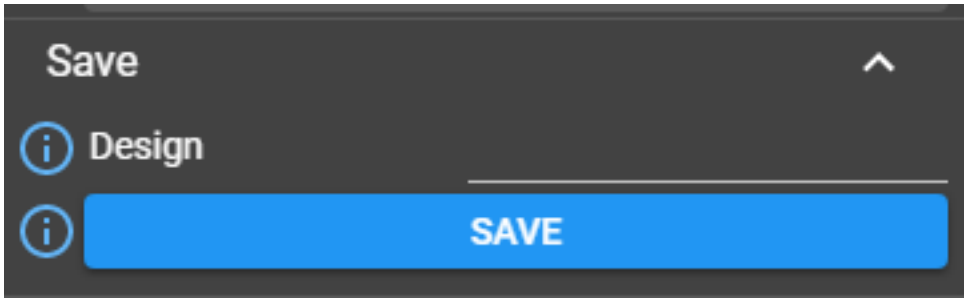

Fig. 5: A method inside the block details

15.7 NavControl

Each NavControl represents a node of the navigation tree that has been previously visited, it provides a user with a mechanism of jumping directly back to that node or alternatively selecting a new child of that node in the menu to the right of the nav control.

15.8 Layout Component

The layout folder in the code base contains everything related to displaying the layout of Malcolm blocks. In the example below we can see various blocks that each have ports and those ports have links joining them together.

The blocks, ports and links all have folders with corresponding code.

15.9 Malcolm

Generally all of the code in the Malcolm folder defines the business logic for the MalcolmJS system, the idea being to keep as much of the Malcolm related code together so it can potentially be distributed as a separate library.

The top level of this folder contains the top level MalcolmJS redux types, the socket handling and various utility functions.

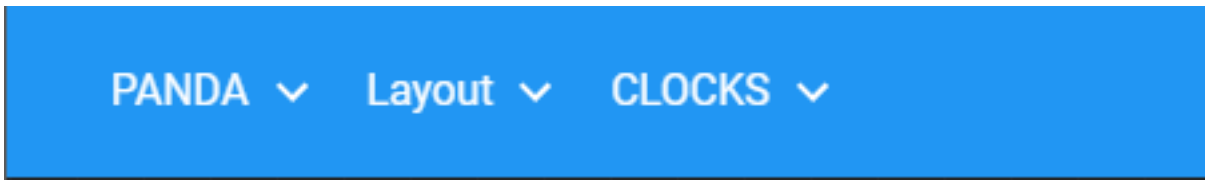

Fig. 6: An example of the navbar

Fig. 7: A navcontrol inside a navbar

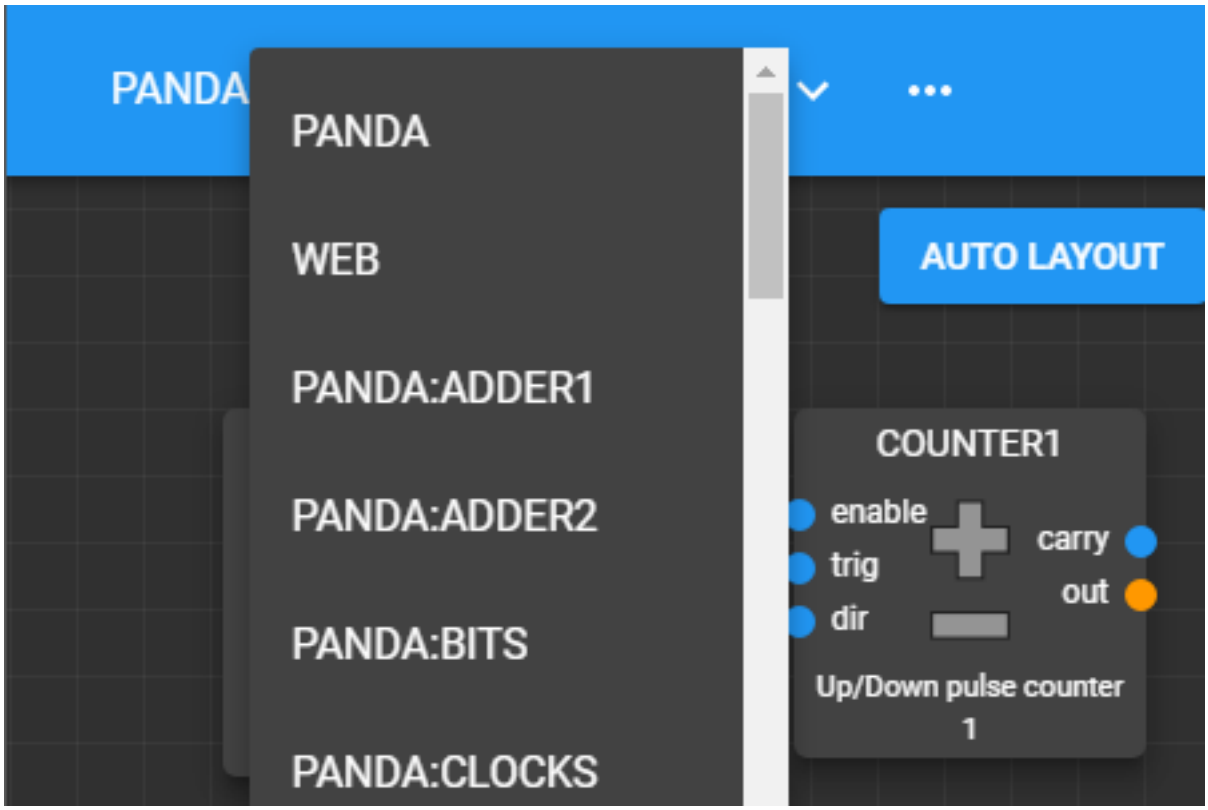

Fig. 8: The menu of children to navigate to

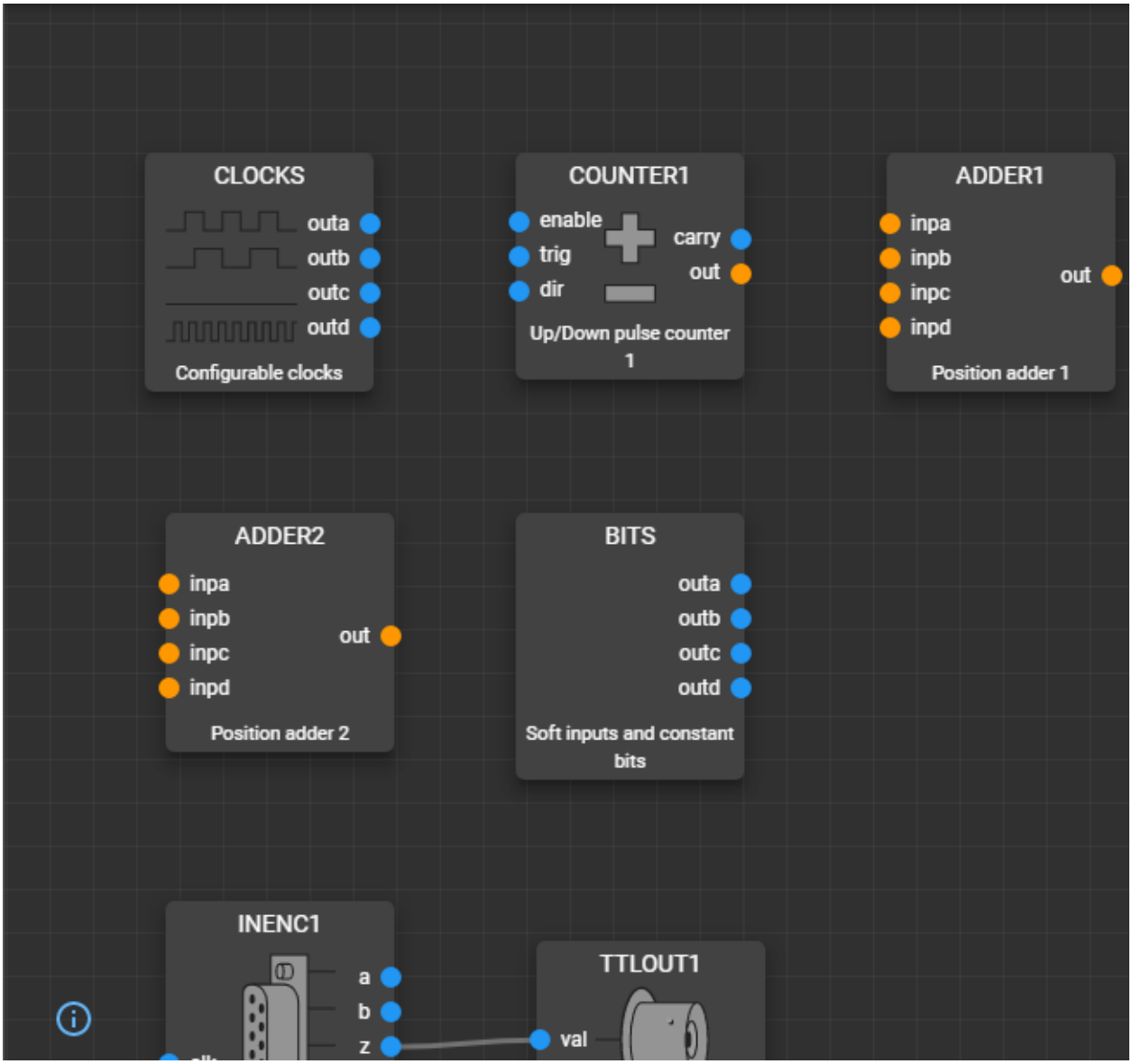

Fig. 9: An example of the layout component

15.9.1 actions

The actions folder contains all the actions for performing malcolm related Redux operations, e.g. sending a message back to Malcolm, calling a Malcolm method or changing the layout of the blocks.

15.9.2 reducers

The reducers folder has all the reducers that mutate the state related to malcolm operations.

15.9.3 middleware

The middleware folder has all the code related to sending messages to the Malcolm websocket.

15.9.4 malcolmHandlers

The malcolm handlers relate to the malcolm socket handler in the root malcolm folder and provide the logic for processing attributes, etc.

15.9.5 MalcolmWidgets

The folder contains all the UI components for displaying attributes, however these components are more generally used throughout the UI to show inputs for methods as well as controls inside table cells.

To review these components individually you can run Storybook with `npm run storybook` to explore these components outside of the MalcolmJS interface.

CHAPTER 16

Maintenance

16.1 Useful URLs

Code	https://github.com/dls-controls/malcolmjs
Agile board	https://waffle.io/dls-controls/malcolmjs
Travis	https://travis-ci.org/dls-controls/malcolmjs
Coverage	https://codecov.io/gh/dls-controls/malcolmjs/branch/version1
Docs	http://malcolmjs.readthedocs.io/en/latest/
PyMalcolm Docs	http://pymalcolm.readthedocs.io/en/latest/

16.2 NPM commands

The main commands you are most likely to need during development are:

```
npm [run] start
npm [run] test
npm run e2e
npm run e2e:interactive
npm run lint:js
npm run storybook
```

`precommit`	the npm action performed before committing that runs the linter and the Prettier style fixer.
`server`	runs a test server with canned responses simulating a connection to Malcolm using websockets.
`server:dev`	the same as `server` but running using `nodemon` for live reload whilst developing the test server component
`start`	runs both the `server:dev` target and `react-scripts start` for live reloading of the MalcolmJS site.
`build`	create a production optimized build of the site
`test`	run the tests and report coverage
`test:watch`	the same as `test` but watches for file changes and re-runs automatically
`eject`	**This should be avoided at all costs so we can update create-react-app** a standard create-react-app command that generates all the config files and webpack settings as standalone files if custom editing needs to be done - this should be avoided unless absolutely necessary/delayed as long as possible
`cy:open`	open the cypress interactive runner
`cy:run`	run the cypress tests in a headless mode
`e2e:serve`	create a production build and serve both MalcolmJS and the websocket connection from the same process (otherwise the end-to-end tests don't exit without a fault on Travis)
`e2e:interactive`	runs `e2e:serve`, waits for the server to start and then runs `cy:open`
`e2e`	runs `e2e:serve`, waits for the server and then runs `cy:run`
`lint:js`	runs ESLint on the javascript files in `src/`
`lint:css`	runs stylelint on the css files in `src/`
`storybook`	runs React Storybook for interactive development of the presentational components
`build-storybook`	builds a static site for the presentation components that can be hosted as a styleguide.
`build-docs`	builds the documentation using sphinx

16.3 Editing documentation

16.3.1 User Guide

Any new features in the UI should be documented in the User Guide, this should be done as part of the story so we can build a user guide over time.

16.3.2 SMG

The diagrams are all made with draw.io and the files exported as svg so they can be versioned in this code base and edited by others. If you edit the diagrams you should export a copy of the updated svg (with "Include a copy of my diagram" checked) back to the `architectural_diagrams` folder.

16.4 The malcolm development server

The MalcolmJS code base comes with a development server to simulate the messages that go to and from Malcolm.

To run the server you can run the `server` or `server:dev` npm commands, the server will be started as part of the `npm start` target.

This server only has very limited capability by returning canned responses captured from using PyMalcolm, as well as handling unsubscribing. For a more realistic scenario, MalcolmJS should be tested against a real instance of PyMalcolm.

More request/response pairs can be added by adding more files to `canned_data` as long as the names start `request_` and `response_` (see below).

16.5 Running pyMalcolm + PandA simulator & Generating canned data

As mentioned above, it is possible to run up a simulator for a PandABox and a corresponding pyMalcolm server to provide a more accurate and complete testing environment. It is also necessary to do this to generate updated canned data responses for the simple dev server (required when pyMalcolm or the PandA software is updated).

16.5.1 Simulator

In order to run the simulator, there are several things required:

1. PandABlocks-* config/binary files [server, FPGA, 2nd-FPGA, rootfs, webcontrol]
2. Latest version of pyMalcolm (clone from https://github.com/dls-controls/pymalcolm)
3. YAML config file for pyMalcolm

The appropriate versions of the PandABlocks-* components should be obtained from Tom Cobb; once they have been placed on a local machine, the simulator can be started by running `<path to pandablocks>/PandABlocks-server/simserver -f <path to pandablocks>/PandABlocks-FPGA` from a terminal.

Once the simulator is running (should display "Server started" in last line of terminal), we can now start the pyMalcolm server; this should be done by running `<path to pymalcolm>/pymalcolm/malcolm/imalcolm.py <path to yaml>/<YAMLNAME>.yaml` from a terminal. The YAML file for pyMalcolm should contain the following:

```
- pandablocks.blocks.pandablocks_manager_block:
    config_dir: /tmp
    mri: PANDA
    hostname: 127.0.0.1

- web.blocks.web_server_block:
    mri: WEB
```

16.5.2 Canned Data

Once the PandA simulator and pyMalcolm are running, it is possible to generate a fresh set of canned data. To do this, first copy the CannedData.json file from *<malcolmJS root>/server/canned_data/Pymalcolm_design* into the config directory as specified in the yaml file (for the above config the folder would be */tmp/PANDA*). Then, visit the page http://localhost:8008/gui/PANDA ; once loaded select 'CannedState' from the design field to load in the canned data design. Finally, run the canned_data_generator.py python script (located in *<malcolmJS root>/server/pyScripts*). This will delete the contents of the *<malcolmJS root>/server/canned_data* folder and generate new json files for a set of pre-programmed blocks, subscribing to their meta and all their attributes. Currently, these blocks are: ["PANDA", "PANDA:TTLIN1", "PANDA:INENC1", "PANDA:LUT1", "PANDA:SEQ1"], in addition to the list of all blocks available on the PANDA simulator.

CHAPTER 17

Performance

17.1 React Performance Tools

Profiling of MalcolmJS is mainly done using the performance tools in the browser, currently this is only supported in Chrome, Edge and IE as React uses the User Timing API but they expect more browsers to follow.

React has a good page describing how to optimise performance at https://reactjs.org/docs/optimizing-performance.html#profiling-components-with-the-chrome-performance-tab

The important point to note is that all of the React events are under the `User Timing` section. We have also added middleware (outlined here https://medium.com/@cvitullo/performance-profiling-a-redux-app-c85e67bf84ae) to allow us to track the time for redux actions.

An example of the profiling output is shown below:

17.2 Why-did-you-update Middleware

In order to identify unnecessary renders we have included the `why-did-you-update` npm package, this is a piece of redux middleware that identifies when the props of a component haven't actually changed.

It adds a significant performance cost and so the code to include it is commented out in `index.js` but can be uncommented on an adhoc basis to run a check on why renders are happening.

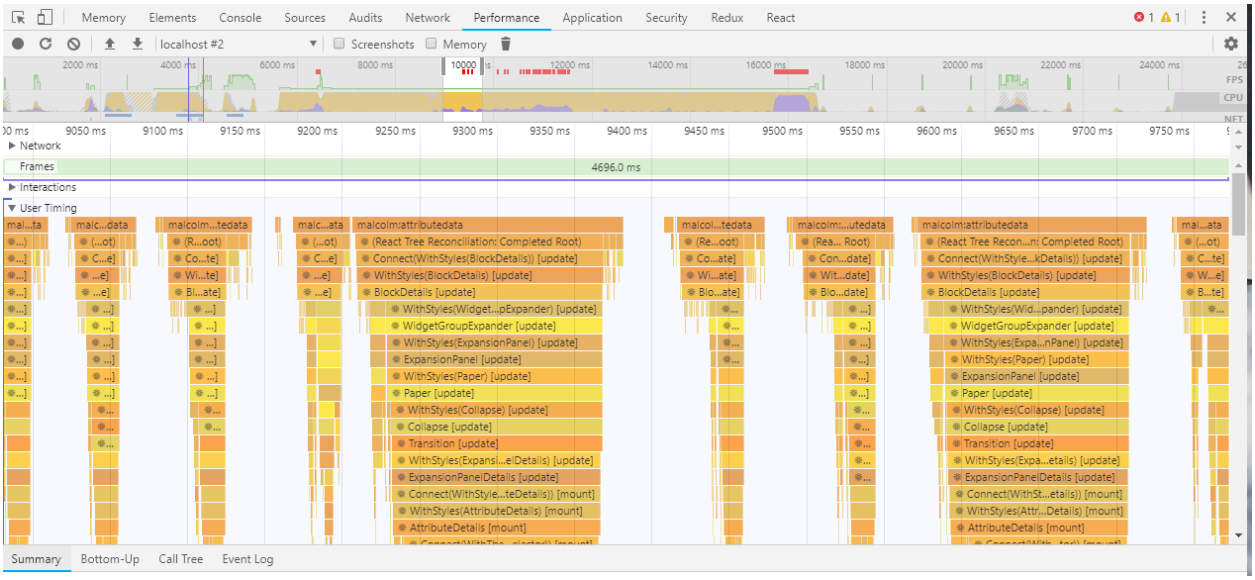

Fig. 1: An example of the profiling output when loading MalcolmJS

CHAPTER 18

Deployment

The continuous integration environment using Travis automatically deploys the built code bundle to Github when a particular commit is tagged. Therefore, to release MalcolmJS all you need to do is tag a commit and push that tag back to the server.

Once the resulting build is complete then the release will appear in the list of MalcolmJS releases.

Tags should take the form major.minor.patch `#-#-#`, the artifacts then use `git describe` to append the number of commits since the tag as well as the SHA if relevant.

The resulting code package can then be downloaded and bundled into a deployment of PyMalcolm. Alternatively you can statically serve the bundle locally using something like the npm package `http-server`.

18.1 Authentication

The recommended way to authenticate is to use a GitHub OAuth token. It must have the `public_repo` or `repo` scope to upload assets. Instead of setting it up manually, it is highly recommended to use `travis setup releases`, which automatically creates and encrypts a GitHub oauth token with the correct scopes.

This results in something similar to:

```
deploy:
  provider: releases
  api_key:
    secure: YOUR_API_KEY_ENCRYPTED
  file: "FILE TO UPLOAD"
  skip_cleanup: true
  on:
    tags: true::
```

If doing this from within Diamond, one should generate keys using the `dls-controls-user` GitHub account (login details available from Controls password vault). Travis CLI tools can be obtained by loading the Ruby module with `module load ruby`.

CHAPTER 19

Styling

19.1 Material-UI Theming

A useful feature of Material-UI is the provision of theme objects; these allow various styling parameters to be defined at a top level and passed down to components which need them.

At its core, a theme is defined as being either light or dark. It then defines colours in a palette object; this palette contains a set of primary, secondary and error colours, each of these having a light, main, dark and contrast text colour. It is possible to let these colour be entirely auto-generated (specifying whether the theme is to be light or dark), entirely manually defined or partially auto-generated (any of the sub-palettes will be auto-generated if not defined, and specifying a main colour for each will allow the other colours in the sub-palette to be auto-generated from this).

It is also possible to specify custom parameters as part of the theme by passing them to createMuiTheme (in the same object and at the same level as palette).

```
const theme = createMuiTheme({
    palette: {
        type: 'dark',
    },
    alarmState: {
        warning: '#e6c01c',
    },
});
```

More information can be found here.

CHAPTER 20

Glossary

These are some commonly used terms in this documentation.

Attribute A property of a *block*.

Attributes are divided into four categories based on their purpose:

- *Parameter Attribute*
- *Input Attribute*
- *Output Attribute*
- *Readback Attribute*

Attribute Group A logically grouping of related *attributes* within a *block*.

Within the user interface these are represented as a collapsable block within the overall *Block Information Panel*.

Block The graphical manifestation of a component within a *design*, encapsulating its attributes, methods and connectivity to other blocks.

Blocks may represent, for example:

- Input and output controllers (interfaces to the FPGA).
- Configurable clocks.
- Logic lookup tables and logic gates.

A Block is defined my its underlying Block specification, which is interrogated by malcolm and reflected into the user interface. For example, the list of supported blocks available for a PandA device is described in PandABlocks-FPGA documentation.

Block Information Panel Panel displayed within the user inferface containing details of the *attributes* and *methods* associated with the currently selected *block* within the *layout*.

Child Block A *block* within the *layout* of a *Parent Block*.

A Child Block may itself represent a Parent Block if its own functionality can be further decomposed.

Design The technical definition of the overall system, or a component within it, describing the *blocks* it contains, their *attributes* and the *links* between them. A Design is represented as a *Parent Block* within the user interface.

Designs are presented graphically as a *layout* within the 'Layout Panel' on the web interface allowing a user to build, configure and manage the system represented by that Design.

Design Element A generic term for any *block*, *attribute* or *link* currently forming the focus of interest within the *layout* view of the PandABox User Interface.

Flowgraph The graphical representation of a *design* showing the *Design Element* within the Control System as presented within the user interface 'Layout View'.

Input Attribute An Input Attribute identifies the value (or stream of values) that will be received into a *block* via a *Sink Port* on the Block to which the attribute relates. There is a 1:1 mapping between Input Attribute and Sink Port.

Input Port Synonym for *Sink Port*.

Layout The graphical representation of a *design* within the web interface showing the *blocks* within the Design and the *links* between them based on the selected *Root Block*.

Link The mechanism of transferring content from a *Source Port* in one *block* to a *Sink Port* in a second Block. Links can only be made between ports of the same logical type (e.g. Boolean -> Boolean, int32 -> int32).

Method Defines an **action** that can be performed by a *block* in support of the purpose of that block.

Output Attribute An Output Attribute identifies the value (or stream of values) that will be transmitted via a *Source Port* out of the *block* to which the attribute relates. There is a 1:1 mapping between Output Attribute and Source Port.

Output Port Synonym for *Source Port*.

Parameter Attribute An attribute whose value can be set by a User within a *block* in order to influence the behaviour of that *block*.

Parent Block A *block* aggregating one-or-more *Child Blocks* each performing an action or activity in support of its parent's functionality.

Parent blocks, together with their attributes and methods are typically presented in the left-hand panel of the web interface when open in Layout View.

Readback Attribute An Attribute whose value is set automatically by a process within the execution environment. Readback attributes cannot be set manually via the User Interface.

Root Block The outermost entity defining the content presented within the user interface. If the outermost Block representing a *design* is selected this encapulates the entire *design*, from where a user can 'drill down' to an area of interest. Otherwise the Root Block represents any configured *block* within the *design*.

Source Port A port on a *block* responsible for transmitting data generated within that Block.

Every Source Port within a Block has a pre-defined type as described in the Block specification.

Sink Port A port on a *block* responsible for accepting data for utilisation within that Block.

Every Sink Port within a Block has a pre-defined type as described in the Block specification.

- genindex

class _

Needed so that genindex is created

Index

Symbols

A

B

C

D

F

I

L

M

O

P

R

S