EAGLE SDK

Tutorial Guide

Version 1.0.0

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

CONTENTS

LIST OF FIGURES

For motor and programming environment set up see the Eagle Reference Manual.

Available at: https://www.irisdynamics.com/downloads/

WARNING

Be aware that the shaft or motor will move during operation. Ensure the shaft and motor are mounted in a safe location and are not in danger of hitting people or property.

TUTORIAL 0: GENERAL OVERVIEW

A simple project is provided called "Start_Here", that includes the basic elements that any project will require and will be the starting point of the first tutorial. The start here folder can be copied and renamed (ensure the .ino file inside is renamed with the same name as the parent folder).

The project contains two files Start_Here.ino, which is the main program file, and Main_GUI.h which contains the IrisControls4 GUI object.

The files in a project can be navigated between by using the tabs in the top left corner. If a new file is required in the project, click the drop down arrow on the right of the window and select new tab. Then name the file "newfilename.h" or "newfilename.cpp".

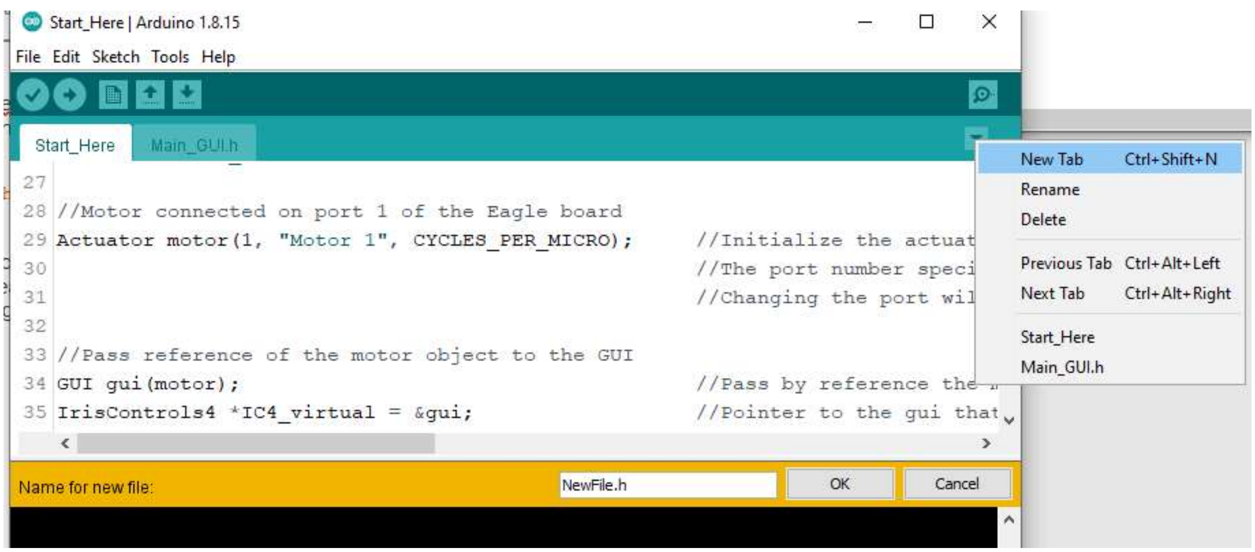

Figure 1:Files in project

The .ino file contains the required components for setting up and maintaining communication with a motor as well as initialization of the GUI object. The Main_GUI.h file contains the GUI object which is used for establishing and maintaining connection to the IrisControls4 windows application.

Flash this this code to the Eagle by pressing the check mark button in the top left then pressing the button on the bootloader. Then run IrisControls4.exe. Select the COM port related to the Eagle from the drop-down menu. Once connected the IrisControls4 window will be mostly empty except the console window which will display connectivity information and a welcome message.

Changing and interacting with the different section of the code will be gone over in detail in the following tutorial, but for now here is an overview of the functionality of different code sections.

Start_Here.ino

```
//This section is where libraries are included
#include <iriscontrols4.h>
#include <modbus_lib.h>
#include "client/device_applications/actuator.h"
#include "Main_GUI.h"

/*Defining of the Actuator object that will be used for motor communications
This will allow communication to a motor plugged in to UART port 1*/
Actuator motor(1, "Motor 1", CYCLES_PER_MICRO);

/*Defining the GUI object that will handle interaction with the IrisControls app*/
GUI gui(motor);
IrisControls4 *IC4_virtual = &gui;

/*This function runs once when the Eagle is reset and will initialize motor
communications */
void setup() {
    motor.init();
    motor.enable();
}

/*This function gets called continuously after the first setup(), this is where
enqueueing new messages to send and parsing of received messages is performed, this
is also where the GUI's run() function is called that will handle GUI
interaction/updates*/
void loop() {
    motor.run_in();
    motor.run_out();
    gui.run();
}

/*Interrupts handling for UART 1 (the same port used when defining our actuator
object)*/
void uart1_status_isr() {
    motor.isr();
}
```

Main_GUI.h

```
//Include guards
#ifndef MAIN_GUI_H_
#define MAIN_GUI_H_

//Library inclusion
#include "device_drivers/k20/ic4_k20.h"
#include "client/device_applications/actuator.h"

//Object that extends the Eagle IrisControls device driver
class GUI : public IC4_k20 {

    Actuator& motor;
    uint32_t gui_timer = 0;
    uint8_t gui_update_period = 100;



    public:
    //Constructor that gives this object access to our motor
    GUI(
       Actuator& _motor
    ):
       motor(_motor)                  //Initialization list
    {
       set_server_name("Tutorial 1");
       set_device_id("Eagle");
    }

    //Called repeatedly from .ino loop()
    void run() {
       check();
       switch (gui_frame_state) {
          case rx:                    //Serial receiving state
             if ( is_timed_out() ) {
                set_disconnected();
                reset_all();
             }
          break;

          case tx:                    //Serial transmission state
             if (new_connection()) {
                setup();              //initialize IrisControls GUI
             }

             if (is_connected() ) {
                if((uint32_t)(millis()- gui_timer) > gui_update_period){
                   gui_timer = millis();
                   frame_update();
                   // Transmit end of transmission message
                   end_of_frame();
                }
             }
           send();
           break;
       }//SWITCH
  }

  private:
```

```
  //Initialize and populate the IrisControls GUI
  void setup() {
     gui_set_grid(30, 30);
     print_l("Eagle Connected to IrisControls\r");
  }

  //Updates to gui elements happen
  void frame_update(){

  }

  //Hide all gui elements
  void hide_all(){
  }

  //Reset all gui elements
  void reset_all(){
  }

};

#endif
```

TUTORIAL 1 IRISCONTROLS4 ENVIRONMENT SETUP

Step 1: GUI Constructor

Open the Main_GUI.h file and navigate to the GUI object constructor. The GUI requires a reference to an Actuator object and will set the server's name and device ID upon construction.

```
public:
    //Constructor
    GUI(
    /* Parameter for constructing the GUI object is a reference to a motor. This
    will allow this object to have access to control and feedback from the motor.*/
        Actuator& _motor

    ):
    // Initialization list
        motor(_motor)
    {
    /* Server name, changes the name in the initial connection message with
    IrisControls. Device ID, changes how the device will show up in the com port
    drop down*/
        set_server_name( "Tutorial 1" );
        set_device_id( "Eagle" );
    }
```

Change the string passed to `set_server_name()` to change the name of the server. Change the string passed to `set_device_id()` to change the ID of the device.

This code can now be recompiled and flashed onto the Eagle. We can see this change in the console output when a new connection is established with IrisControls4.

Figure 2: Change to server name and device id

The device ID, as well as the COM port number of the device, is seen by clicking on the drop down above the connection button.

Figure 3: COM port drop down

Step 2: New Connection with Iris Controls

While still connected to IrisControls4, type the command `"guide_on"` into the console. You will be able to see the numbered rows and columns of the GUI grid. The (0, 0) point is in the top left corner of the window. Enter the command `"guide_off"` to hide these rows and columns again. This is a helpful tool to use when determining placement of GUI elements

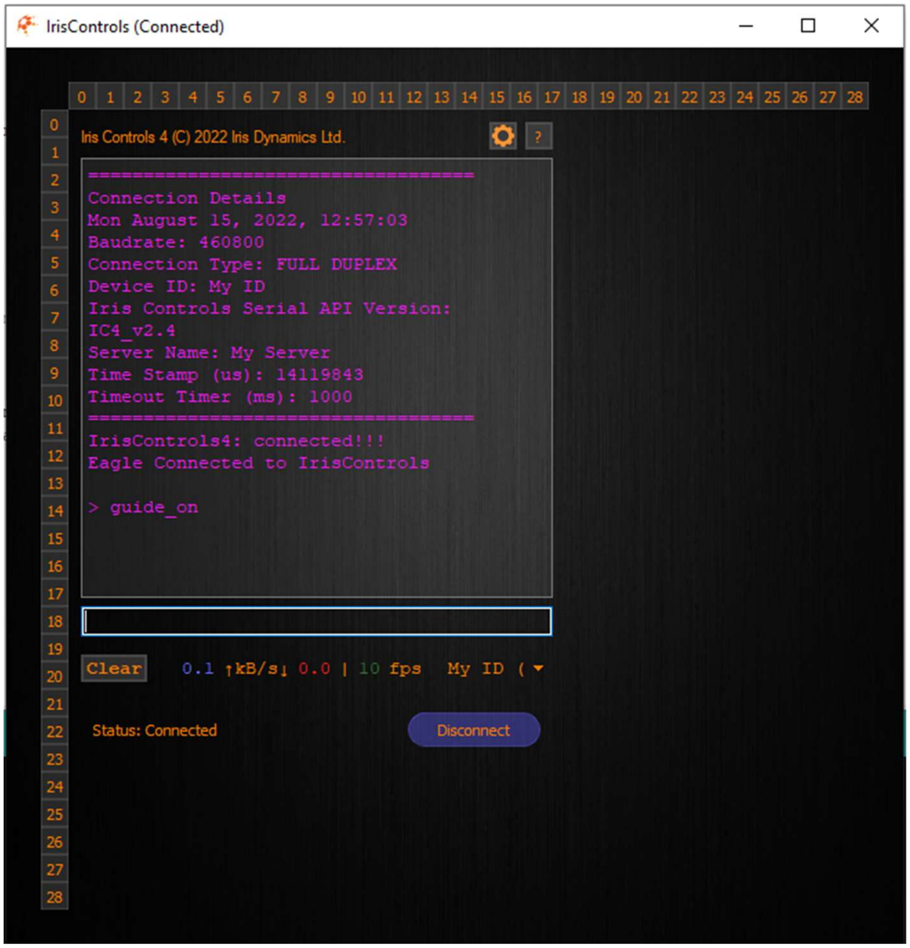

Figure 4: Turning grid guide on for 30 x 30 window

The grid size is established in the `setup()` function of the GUI object. This is where the GUI grid size is currently set to 30x30 units and this is also where a message is printed to the console when the connection between the Eagle and IrisControls4 has been established.

```
void setup() {
   // This will set the size of the IrisControls window
   gui_set_grid(30, 30);
   // Message is printed when IrisControls establishes a connection with the Eagle
   print_l( "Eagle Connected to IrisControls\r" );
}
```

Now, change the GUI grid size by changing the arguments passed to `gui_set_grid()`. The first argument is the height (number of rows) of the GUI and the second argument is its width (number of columns). Decrease the GUI height to 30 units and increase the GUI width to 60 units.

The message printed upon connection can also be change by passing a different string to the `print_l()` function.

```
void setup() {
   // This will set the size of the IrisControls window
   gui_set_grid(30, 60);
   // Message is printed when IrisControls establishes a connection with the Eagle
   print_l( "New Connection Message\r" );
}
```

After recompiling the project, the change will be seen in the IrisControls4 window.

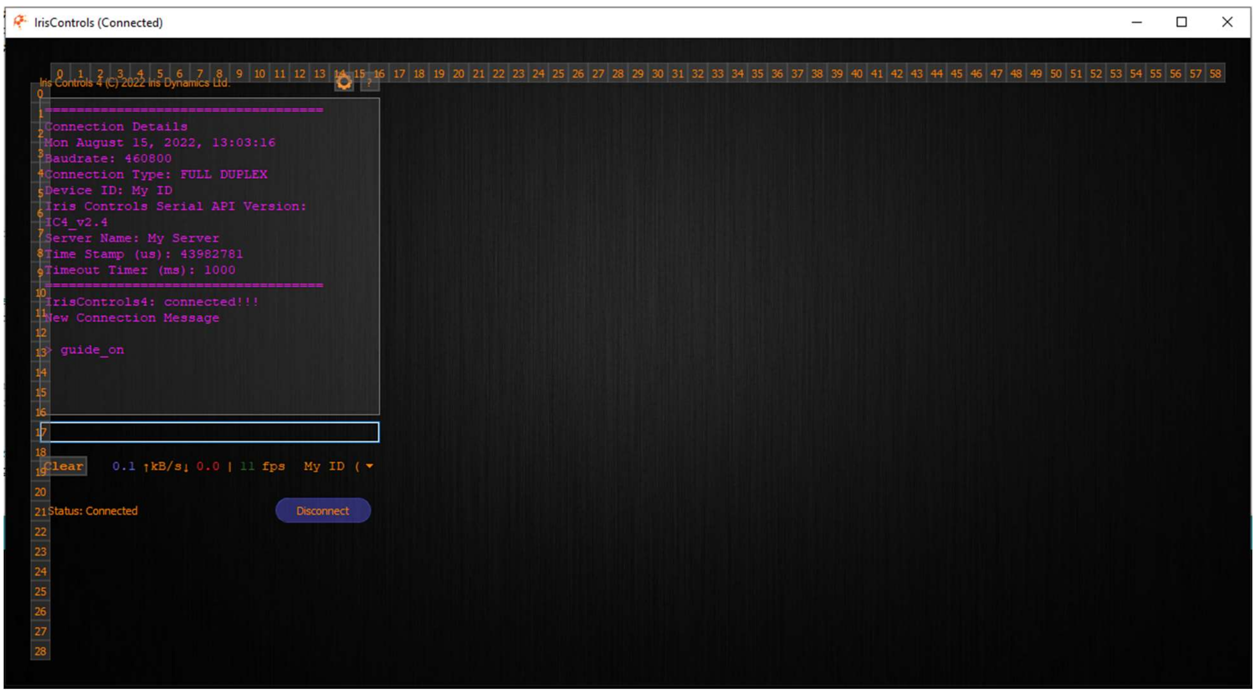

Figure 5: Update grid size and new connection message printed to console

Step 3: Adding and Updating GUI elements

We will now add a GUI element to display the motor's position. The `FlexData` element type is well suited for this as it displays a labelled data with the ability to add units.

```
class GUI : public IC4_k20 {
   /* Reference to an actor object that will be passed in when this object is
   initialized*/
   Actuator& motor;
   // Data element for displaying the motor's position
   FlexData position_element;
```

Initialize the `FlexData` element in the `setup()` function by calling its `add()` function. It will be initialized once a connection with IrisControls4 is established.

The `add()` function takes the following arguments*:

1.	A string for the data element's name/label	(const char *)	"Position: "
2.	The row location	(uint16_t)	5
3.	The column location	(uint16_t)	19
4.	The row span	(uint16_t)	2
5.	The column span	(uint16_t)	10
6.	The initial value	(int)	0
7.	The denominator value	(uint16_t)	1
8.	A string for the units	(const char *)	"*mu*m"
9.	Configuration	(uint16_t)	FlexData::UNITS

** Additional information about the GUI element configuration options and the IrisControls4 API can be found at https://wiki.irisdynamics.com/index.php?title=IrisControlsAPI4_Overview*

The value of the `FlexData` element will need to be continuously updated with position data from the motor (`Actuator` object). The `frame_update()` function is called periodically to update the GUI, so this is where we should update the `FlexData` element. This can be done by calling the `FlexData`'s `update()` function and passing it the value we want to use to update the element. The motor's most recent position can be retrieved using the `Actuator` object's `get_position_um()` function.

```
/*GUI initalization called from check for new IrisControls connection in run function
tx state*/
void setup() {
   //This will set the size of the IrisControls window
   gui_set_grid(30, 60);
   // Print on establishing a connection with IrisControls
   print_l("New Connection Message\r");
   /* Init data element with a name, row and col anchors, row and col spans, init
   value, denominator (can be used to divide a value passed to the element, using 1
   has no effect), units, and configurations*/
   position_element.add("Position: ", 5, 19, 2, 10, 0, 1, "*mu*m", FlexData::UNITS);
}

/*Action to be called every gui frame go here. This is called inside the is connected
gui update loop*/
void frame_update() {
   // Update position element with motor position
   position_element.update(motor.get_position_um());
}
```

After compiling and uploading this code, the GUI should look like the below picture. If you move the shaft, you should be able to see the position value update in the GUI.

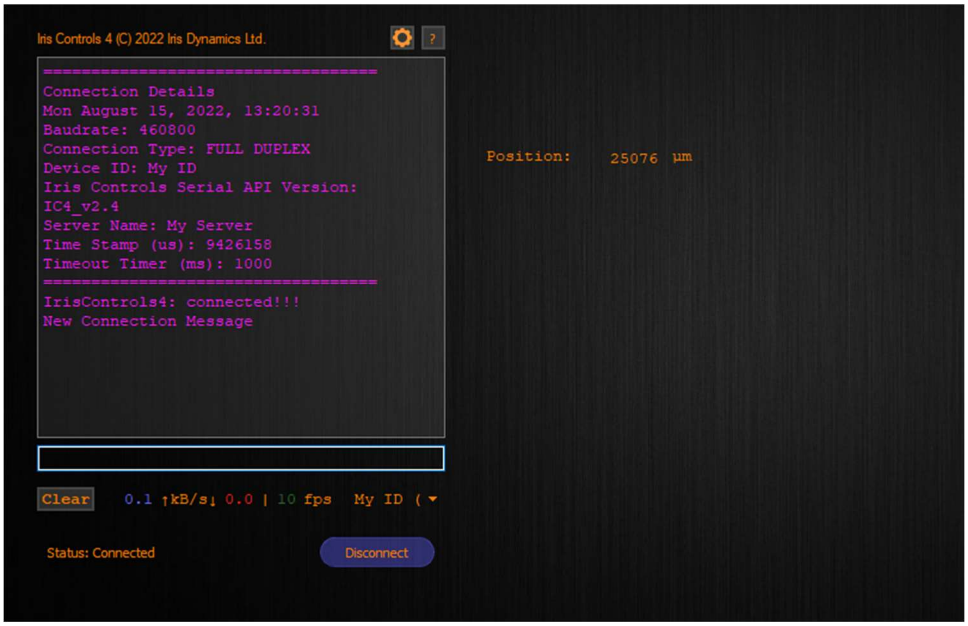

Figure 6: Display of position FlexData element

The static variable `gui_update_period` is the time between updates in milliseconds which dictates the GUI frame rate. It is currently given the value of 100 milliseconds which results in transmitting 10 frames per second (FPS). When the Eagle and IrisControls4 have established a connection, the FPS can be seen below the console in green.

Figure 7: GUI frames per second display

This value can be adjusted to 20 ms and the resulting FPS will change to 50.

TUTORIAL 2: IRISCONTROLS4 GUI PAGES

This tutorial picks up where tutorial one left off and demonstrates how to populate the IrisControls4 GUI with additional elements, hide/display these elements, and how to group elements to create "pages". GUI pages allow for GUI elements that should be shown at the same time to be grouped together for improved readability, organization, and to minimize serial traffic.

Step 1: Create Home Page Class

To do this let's start by creating a new file named "`Home_Page.h`", that contains a class called `Home_Page`.

We can move the position element declaration into the `Home_Page` class. We also need to define an `Actuator` object property. Similarly as to how the `Actuator` object was passed to the GUI class's constructor, we need the GUI class to then pass that `Actuator` object to the `Home_Page`'s constructor so that it can be used to update the position element. We can also add a Boolean property `is_running` that can be used to determine when the page is active and only perform the `run()` function when it is.

```
#ifndef _HOME_PAGE_H_
#define _HOME_PAGE_H_

class Home_Page {
        Actuator& motor;
        FlexData position_element;

    public:
        bool is_running = false;
        //Constructor
        Home_Page(
            Actuator& _motor
        ):
            motor(_motor)
        {}
};
#endif
```

Next, we will define two functions: `setup()` and `run()`. `setup()` will be called from the GUI class's own `setup()` function and will initialize the position element. The `run()` function will be called from the GUI class's `frame_update()` function, and will update the position element with the current position of the motor.

Move the position element initialization from the GUI class to `Home_Page`'s `setup()` function and move our code to update the element from the GUI class to the `run()` function.

```
#ifndef _HOME_PAGE_H_
#define _HOME_PAGE_H_

class Home_Page {
       Actuator& motor;
       FlexData position_element;

    public:
       bool is_running = false;   //indication of the page state
        //Constructor
       Home_Page(
```

```
            Actuator& _motor
        ):
            motor(_motor)
        {}

        // Initializes all home page elements and puts the motor into sleep mode
        void setup() {
            // Initialize position element
            position_element.add("Position:   ",   5,   19,   2,   10,   0,   1,   "*mu*m", 
FlexData::UNITS);
            is_running = true;
        }

        // Handles updating home page elements with motor data
        void run() {
            if (!is_running) return;
            // Update position element with motor position
            position_element.update(motor.get_position_um());
        }
};
#endif
```

In the `Home_Page` class, we can add a private `bool first_setup` to keep track of whether the page elements have been initialized yet. This way, the `setup()` function will only initialize these elements once then simply display them during any following calls. As well we will need a `hide()` function to hide the contents of the page. An additional `reset()` function can be used when a disconnection from IrisControls4 occurs. These functions will also make changes to the `is_running` property value. We can ensure that the run function only proceeds if the page's `is_running` property is `true`. Update the `Home_Page` class as the following code block shows.

```
class Home_Page {
    Actuator& motor;
    FlexData position_element;
    bool first_setup = true;
  public:
    bool is_running = false;
    //Constructor
    Home_Page(
      Actuator& _motor
    ):
      motor(_motor)
    {}

   void setup() {
       if (first_setup) {
           first_setup = false;
           position_element.add("Position:   ",   5,   19,   2,   10,   0,   1,   "*mu*m", 
FlexData::UNITS);
       }
       else {
           position_element.show();
       }
       is_running = true;
   }

   void run() {
```

```
        if (!is_running) return;
        position_element.update(motor.get_position_um());
    }

    void hide(){
        position_element.hide();
        is_running = false;
    }

    void reset() {
        hide();
        first_setup = true;
    }
```

Now that the `Home_Page` class is made; we can initialize a `Home_Page` object from the GUI class. All the code from the GUI class that was moved to the `Home_Page` class will also be replaced with calls to the new `Home_Page` functions. To use the `Home_Page` object, include the file `Home_Page.h` at the top of `Main_GUI.h`.

Define a `Home_Page` object in the GUI class and initialize the object with the `Actuator` object in the GUI constructor.

```
#include "device_drivers/k20/ic4_k20.h"
#include "client/device_applications/actuator.h"
#include "Home_Page.h"              //include new file

class GUI : public IC4_k20 {

        Actuator& motor;
        uint32_t gui_timer = 0;
        uint8_t gui_update_period = 20;
        Home_Page home_page;         //Define object

    public:
        //Constructor
        GUI(
            Actuator& _motor
        ):
            // Initialization list
            motor(_motor),
            home_page(_motor)        //Object initialization
        {
            set_server_name( "Tutorial 2" );
            set_device_id( "Eagle" );
        }
```

Now you can add a home page `setup()` function call to the GUI `setup()` function, and a home page `run()` function call to the GUI `frame_update()` function.

To ensure pages are re-initialized after disconnecting from IrisControls4, `reset_all()` should be called upon establishing a connection. The home page `setup()` function should also be called so that it is displayed when the connection is established.

```
void setup() {
   gui_set_grid(30, 60);
   print_l( "Eagle Connected to IrisControls\r" );
   reset_all();
   home_page.setup();
}

void frame_update() {
   home_page.run();
}
```

Now, if you compile and flash the code, the position element is displayed in the GUI as before.

Step 2: Use FlexButton Element to Hide/Display Home Page

The home page we just created can be hidden or displayed from the GUI by toggling a button element. Which can be done by defining and initializing a `FlexButton`.

```
class GUI : public IC4_k20 {

        Actuator& motor;
        Home_Page home_page;
        FlexButton home_page_btn;
```

The FlexButton's `add()` function takes 6 arguments:

1. A name (const char *)
2. An initial toggle value (1 = toggled on, 0 = toggled off, -1 = not toggleable)

3. & 4. Row and column anchors
5. & 6. Row and column spans

This can be added to the GUI `setup()` function as it will persist regardless of the displayed page.

```
void setup() {
   gui_set_grid(30, 60);
   print_l( "Eagle connected to IrisControls\r" );
   home_page.setup();
   home_page_btn.add( "Home" , -1, 26, 1, 2, 4);
}
```

Next, we will want to use this button for hiding and displaying the home page. In the GUI class `frame_update()` function, we can manage the current page being displayed. The `FlexButton pressed()` function will return true when a button is clicked. In this case if the page is already running, it can be hidden. Otherwise, the page should be setup (initialized or displayed). We will also place the Home Page's `hide()` function in the GUI `hide_all()` function, and the Home Page's `reset()` function in the GUI `reset_all()` function.

```
void frame_update() {
   if (home_page_btn.pressed()) {
      if (home_page.is_running) home_page.hide();
         else home_page.setup();
      }
   home_page.run();
}

void hide_all(){
   home_page.hide();
}

void reset_all(){
   home_page.reset();
}
```

Compile and upload the code and reconnect to IrisControls4.exe. Clicking the home button should now hide and display the position element.

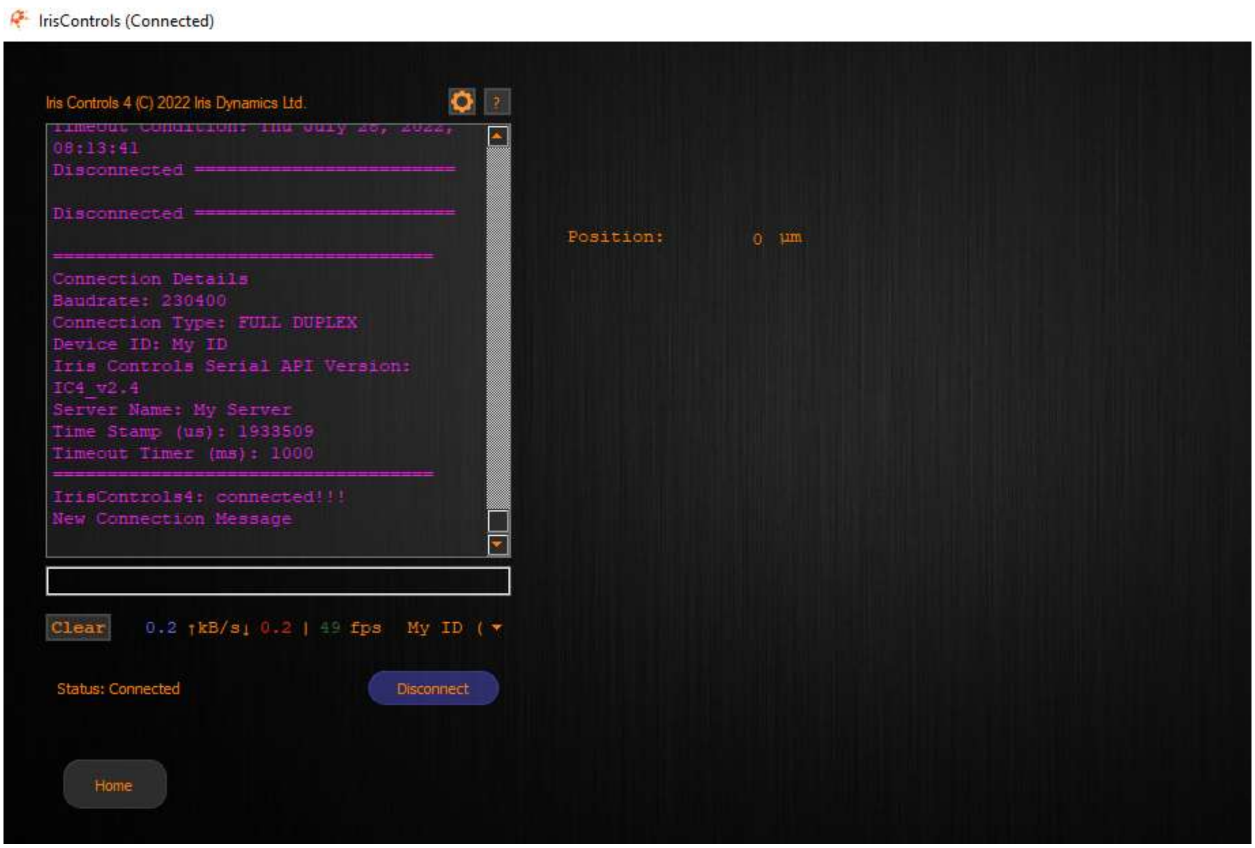

Figure 8: Home Button for showing home page

Step 3: Additional Home Page Element

Now that we have a functional home page class, we can add other elements to the page. Let's add a `FlexLabel` element to `Home_Page`.

```
class Home_Page {
   Actuator& motor;
   FlexData position_element;
   FlexLabel page_label;
```

A `FlexLabel` is initialized when its `add()` function is called. The first argument this function takes is a name (const char *) which can be styled by HTML tags. The second and third arguments are the column and row anchors. The fourth and fifth arguments are the column and row spans.

The label must also be displayed in the `Home_Page setup()` function and hidden in the `Home_Page hide()` function.

```
//show all elements on this page
void setup() {
   if (first_setup) {
      first_setup = false;
      position_element.add("Position: ", 5, 19, 2, 10, 0, 1, "*mu*m",
FlexData::UNITS);
      page_label.add( "<p_style=\"font-size:20px;\">Home</p>" , 1, 19, 2, 9);
     }
     else {
        position_element.show();
        page_label.show();
     }
     is_running = true;
}

//hide all elements on this page
void hide(){
   position_element.hide();
   page_label.hide();
   is_running = false;
}
```

Figure 9: Home Page label with different font size

Step 4: Use GUI_Page Object to Clean up Home Page

If we were to continue adding elements this way, each new element would require manually adding a function call to the `hide()` and `setup()` functions to hide and display it. To avoid this, we can use a `GUI_Page` object. If we define a `GUI_Page` object as the parent of all elements in the

`Home_Page` class, the `GUI_Page` object can handle hiding/displaying all elements with a single function call, which will also minimize the serial traffic.

```
class Home_Page {
   Actuator& motor;
   FlexData position_element;
   FlexLabel   page_label;
   GUI_Page page_elements;
```

To add elements to the GUI_Page object, the object must first be initialized by calling its `add()` function which takes no arguments. To link elements to a GUI page object, each `FlexElement` has an alternative `add()` function that takes a reference to a GUI_Page object as the first parameter of the function.

Now that the Home_Page Flex elements are linked to the new GUI_Page object, it can be used to replace the multiple show and hide calls.

```
//show all elements on this page
void setup() {
   if (first_setup) {
      first_setup = false;
      page_elements.add();
      position_element.add(&page_elements, "Position: ", 5, 20, 2, 13, 0, 1, "*mu*m",
FlexData::UNITS);
      page_label.add(&page_elements, "<p_style=\"font-size:20px;\">Home</p>" , 1, 19,
2, 9);
    }
    else {
        page_elements.show();
    }
    is_running = true;
}

//hide all elements on this page
void hide(){
    page_elements.hide();
    is_running = false;
}
```

Using GUI pages makes it easy to link together groups of associated GUI elements and improve organization and readability. For additional examples of GUI elements (buttons, sliders, graphs, labels) and different configuration possibilities, see the “EagleSDK_GUI_example” in the “examples” folder.

TUTORIAL 3: MOTOR INFORMATION DISPLAY

In the included GUI_Panels library there are some objects that can be used in your project that contain a set of related GUI elements. The Motor_Plot object is one that creates a visual display of the information returned to the Eagle from the Orca motor.

Step 1: Add Motor_Plot Object

To access the `Motor_Plot` object, include `Motor_Plot_Panel.h` in `Home_Page.h`.

Now, a `Motor_Plot` object can be defined in the `Home_Page` class.

```
#include "Motor_Plot_Panel.h"

class Home_Page {

    Actuator& motor;

    Motor_Plot motor_plot;
```

The `Motor_Plot` is initialized by calling its `add()` function which requires four to five arguments:

1. (optional) The `GUI_Page` object that all panel flex elements will be added to
2. The `Actuator` object that enables connection to the Orca motor
3. name (const char *)
4. row anchor (uint16_t)
5. column anchor (uint16_t)

```
void setup() {
   if(first_setup){
      first_setup = false;
      // Initialize GUI_Page object
      page_elements.add();
      // Initialize motor plot
      motor_plot.add(&page_elements, &motor, "Orca Motor", 1, 35);
```

Initializing `Motor_Plot` enables you to see it in the GUI. The `Motor_Plot run()` function handles updating the panel with data from the motor. Call the `Motor_Plot run()` function from the `Home_Page run()` function to ensure the plot is consistently updated.

```
void run() {
   if (!is_running) return;
   // Update motor plot with motor data
   motor_plot.run();
```

Now move the motor's shaft to see the position data in the plot follow the motor's position value.

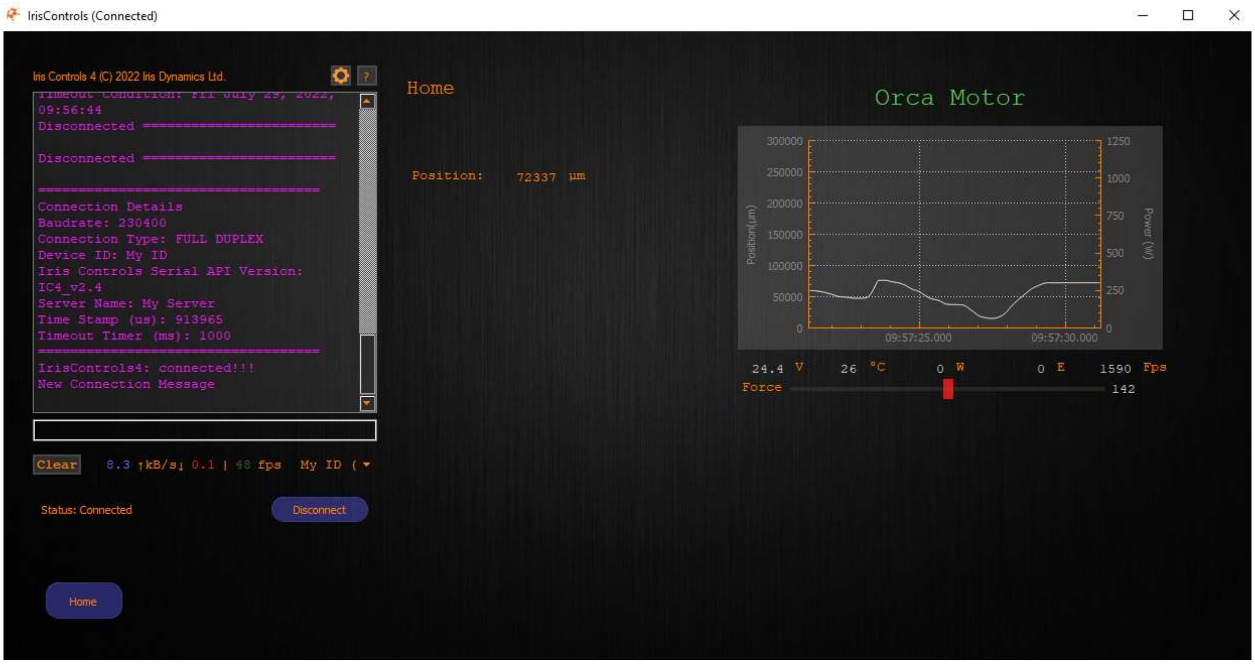

Figure 10: Plot displaying moving shaft position

Since the `Motor_Plot` object was initialized with a `GUI_Page` object, that `GUI_Page` object handles hiding / displaying the `Motor_Plot`.

The plot itself displays the position of the motor in micrometers with a white line as well as the power of the motor in watts (W) with a red line. From left to right just below the plot, values for the motor's voltage, temperature, power, error codes, and frames per second (fps) are displayed. Below these is a slider that displays the value of the motor's sensed force. Double clicking the plot pauses it and allows you to scroll up, down, left, or right (click and drag the mouse) or zoom in and out. If the title of the plot is green, a connection to an Orca motor has been successfully established. If it is gray, no connection has been established. If it is red, a connection has been established but the motor has active errors. The numerical error code value below the plot can tell you what the error is. See the Orca API User Guide for a list of error codes and their corresponding descriptions.

Step 2: Add Dataset to Plot

Additional data can be added to the plot by using the `Dataset` object from the IrisControls4 API. We can also add a slider to the GUI whose value will be used to update the new `Dataset`. Start by declaring a `Dataset` and a `FlexSlider` in the `Home_Page` class.

```
class Home_Page {

        Actuator& motor;
        Motor_Plot motor_plot;
        Dataset new_data;
        FlexSlider new_data_slider;
```

In the `Home_Page setup()` function, initialize both elements by calling their `add()` functions. The `FlexSlider add()` function requires ten arguments and can optionally take a `GUI_Page` as its first argument.

1. Name (const char *)
2. Row anchor (uint16_t)
3. Column anchor (uint16_t)
4. Row span (uint16_t)
5. Column span (uint16_t)
6. Minimum slider value (int)
7. Maximum slider value (int)
8. Initial slider value (int)
9. Denominator value (uint16_t)
10. Configuration value (uint16_t)

The denominator argument can be used for unit conversion. All values sent to the slider will be divided by the denominator while all values coming from the slider will be multiplied by the denominator. We can use 1000 for the denominator, this way the slider range will be 1000 times less than our plot axis range.

The configuration value we need to include is `FlexSlider::ALLOW_INPUT`. This enables the ability for the user to drag the slider. See the IrisControlsAPI4 Wiki to read the complete list of available configuration values.

The `Dataset add()` function has four required arguments:

1. Reference to `FlexPlot` object that will display the data
2. Name (const char *)
3. X-axis label (const char *)
4. Y-axis label (const char *)
5. Configuration value (uint16_t)

Note that these axes labels will not be displayed unless the `Motor_Plot` function `plot.set_axes_labels()` is called with the `Dataset` object provided as an argument.

The last argument is a configuration value. If `Dataset::TIMEPLOT` is set, `Dataset` expects all data point x-values to be a time in microseconds since device boot. If `Dataset::NONE` is set, data points will not be marked and values will be displayed by a continuous line. See the IrisControlsAPI4 Wiki to read the complete list of available configuration values.

The `Dataset set_max_data_points()` function can be used to set the maximum number of datapoints that IrisControls4 saves before it deletes old data to make room for the new data (default value is 100,000). The `Dataset set_colour()` function can be used to set the color used to plot the data.

The Dataset object also has a `show()` and `hide()` function that are used to display or remove the data from its associated plot.

```
// Add new dataset to plot
new_data.add(&motor_plot.plot, "New Data", "Time", "Slider Value", Dataset::TIMEPLOT
+ Dataset::NONE);
new_data.set_max_data_points(25000);
new_data.set_colour(GREEN);
new_data.show();

// Init slider for dataset values
new_data_slider.add(&page_elements, "Plot Value:", 8, 19, 1, 15, 0, 300, 0, 1000,
FlexSlider::ALLOW_INPUT);
```

To update the value being plotted, call the `Dataset` object's `add_data()` function from the `Home_Page run()` function. `add_data()` takes two arguments: an x-value and a y-value of the data point to plot. Since this is a time plot, the x-value is the current time. The y-value is the current value of the `FlexSlider`, which we can get by calling the `FlexSlider get()` function.

```
// Update new dataset
uint64_t now = IC4_virtual->system_time();    // Get current time
new_data.add_data(now, new_data_slider.get());
```

Now when you compile and upload the code, you will see the slider value in the plot as displayed in the image below.

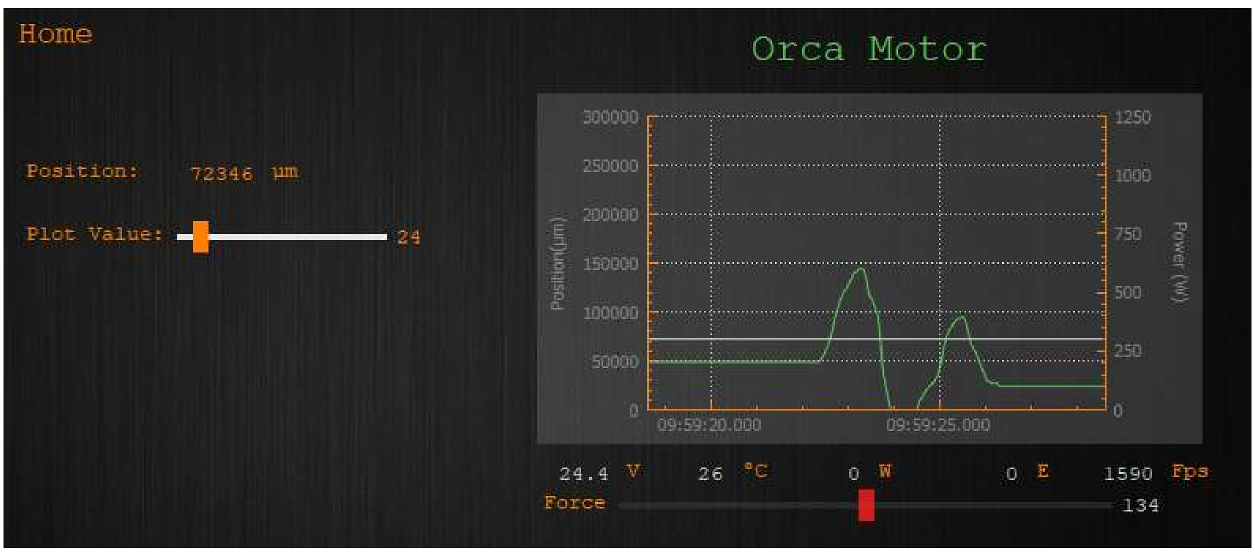

Figure 11: Slider value displayed on plot

Step 3: Connection Configuration

One of the values under the plot is the Fps which denotes the speed of messages between the Eagle controller and Orca motor. This speed is determined during the handshake between the motor and Eagle during initial connection. In this case default values for the baud rate and interframe delay are used. This results in a stream frequency of approximately 1590 successful message responses per second from the motor.

The connection configuration can be updated in the code by declaring a `ConnectionConfig` struct (defined in `Actuator` class), updating its properties, and passing it as an argument to the `Actuator` object function `set_connection_config()`.

The `ConnectionConfig` baud rate is defined by its `target_baud_rate_bps` property. Here, it is set to 1040000, the maximum allowable rate. The target delay is the time between a message being received and the next message sending out, it can be set as low as 0. Then, the `ConnectionConfig` can be updated in an `Actuator` object by passing it to the `Actuator` class function `set_connection_config()`. This values will only be used if set before call the the Actuator `enable()` function, so we will place it in the `setup()` function of the .ino file.

```
Actuator::ConnectionConfig connection_config;

void setup() {
    motor.init();
    connection_config.target_baud_rate_bps = 1040000;
    connection_config.target_delay_us = 0;
    // Update the configuration in the Actuator object
    motor.set_connection_config(connection_config);
    motor.enable();
}
```

Once the code is compiled and uploaded, the updated stream frequency should be displayed below the bottom right corner of the plot. Updating the target baud rate has increased the stream frequency from 1590 bits per second to 2676 bits per second.

Figure 12: Increased FPS from new connection_config values

TUTORIAL 4: SERIAL COMMANDS

Serial commands enable the user to send commands through the IrisControls4 console. This tutorial will go over how to implement custom commands and start moving the motor.

Warning: Ensure the motor is secured and will not hit people or property if the shaft loses control.

Step 1: Define parse_app(char * cmd, char * args)

`parse_app(char * cmd, char * args)` is a function recognized by the IrisControls4 message parser as an optional parser that can be implemented on the application layer. You can declare this function in your code and implement your own command handling here. Commands can be implemented between `START_PARSING` and `FINISH_PARSING` using the macros `COMMAND_IS` and `THEN_DO`. Add the `parse_app()` function to the GUI class.

```
/* @Brief Parsing of serial messages at application layer */
int parse_app(char * cmd, char * args) {

   START_PARSING

   /* Command handling format:
   COMMAND_IS "<command name>" THEN_DO
   <code to handle command>
   Note: all arguments following the command name are stored in the char pointer
   "args" which is delimited by whitespace characters.*/

   FINISH_PARSING
}
```

Step 2: Add Hello World Command

For our first command, we want to implement a command that prints “world” when “hello” is written to the console. Here, the command name is “hello”, and the code to handle the command is a single print statement with the string “world”. The ‘\r’ is a carriage return which will create a new line in the console.

```
/* @Brief Parsing of serial messages at application layer */
int parse_app(char * cmd, char * args) {

   START_PARSING

   // Command "hello" prints "world" to IC4 console.
   COMMAND_IS "hello" THEN_DO
      print_l("world\r");            // Print "world" to console

   FINISH_PARSING
}
```

Once you compile and flash these changes, you can use the new command. If you enter “hello” into the console, “world” is printed in response.

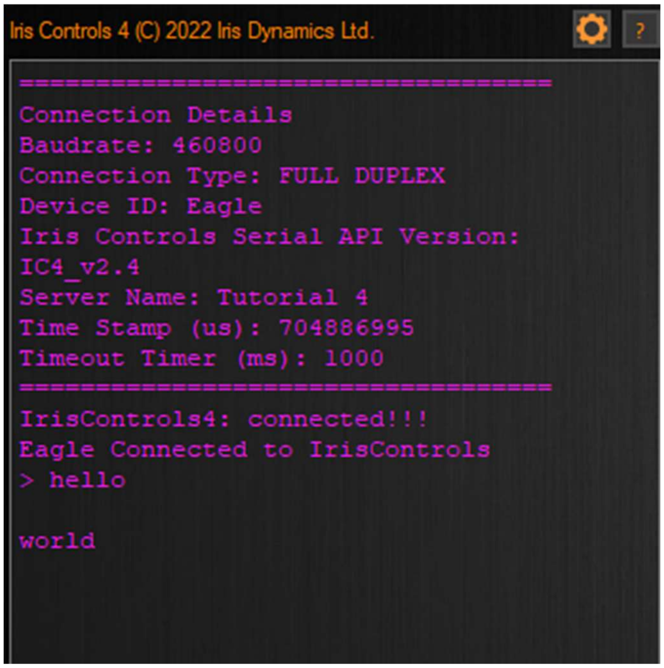

Figure 13: Hello world console parsing

Step 3: Add Command for Retrieving Motor Data

Now let's add another command "get_data", to print latest values for the motor's temperature, position, force, power, and voltage.

The `Actuator` object provides several get functions for retrieving this data. To retrieve the motor's temperature in Celsius, we can call the `Actuator` object's `get_temperature_C()` function. In order to print a string to the console use the `print_l()` function while the `print_d()` function is used for printing integers. Some symbols are also available such as *deg* and *mu*.

```
    // Prints motor's temperature, position, force, power, and voltage.
    COMMAND_IS "get_data" THEN_DO
       print_l( "\rTemperature (*deg*C): " );
       print_d(motor.get_temperature_C());

       print_l( "\rPosition (*mu*m): " );
       print_d(motor.get_position_um());

       print_l( "\rForce (mN): " );
       print_d(motor.get_force_mN());

       print_l( "\rPower (W): " );
       print_d(motor.get_power_W());

       print_l( "\rVoltage (mV): " );
       print_d(motor.get_voltage_mV());
```

Recompile and upload the code and reconnect to IrisControls4. Running the "get_data" command should print each of the specified data values for the motor.

Step 4: Add Command for Updating Register Value

In this step we will implement a command that takes an argument to update the motor's maximum allowable temperature. The motor will not output any force if the maximum temperature is reached until it has cooled down.

The Actuator object provides set commands that enable updating the motor's registers. For example, we can update the motor's maximum allowable temperature value by calling the Actuator object's `set_max_temp()` function. This function takes one 16-bit unsigned integer argument for the temperature value. Refer to the Orca API Reference Manual for all set functions.

Command arguments are stored in a char pointer called `args`. Each argument contained in `args` has a white space char before it. The last character in `args` is the return character, '\r'. The `parse_int()` function is a helper function for parsing integer arguments. The first argument it takes is the argument list, `args`. The second argument takes a reference to an argument index integer which is incremented by the function. For example, `parse_int(args, arg_index)` will return the first integer argument in `args` and calling `parse_int(args, index)` a second time would return the second if it was present, and so on.

```
// Command message of "max_temp 50" will give the motor a maximum allowable temperature
of 50 degrees Celsius
COMMAND_IS "max_temp" THEN_DO
   unsigned int arg_index = 0;
   // Get max temperature value from argument list
   uint16_t max_temp = parse_int(args, arg_index);
   print_l( "\rSetting max temp: " );
   print_d(max_temp);
   // Set max temp in motor
   motor.set_max_temp(max_temp);
```

Step 5: Add Command for Updating Motor Force

Let's start moving the motor! As of now the motor has been in "Sleep" mode we will need to move it to "Force" mode and continually set the force value to ensure the timeout does not expire which returns the motor to "Sleep" mode. We will need to add a public property to the GUI object that will be used for setting the target force for that it can be retrieved from the .ino main loop.

```
public:
  int32_t target_force = 0;
```

Then in the loop() function in the .ino file we will add a call to the Actuator object set_force_mN() function and pass the target_force as an argument. The call to set the force or position of the motor should be called in this loop to ensure the highest frequency of updates to the motor.

```
void loop() {
    motor.set_force_mN(gui.target_force);
    motor.run_in();
    motor.run_out();
    gui.run();
}
```

This line has no effect while the motor is in “Sleep” mode so let’s use a serial command to put the motor into “Force” mode and update the target force value.

```
// Command message of "f 1000" will give a target force of 1000 to the motor.
COMMAND_IS "f" THEN_DO
   unsigned int arg_index = 0;
   target_force = parse_int(args, arg_index);
   print_l("\rTarget force: ");
   print_d(target_force);
   motor.set_mode(Actuator::ForceMode);
```

First try commanding a force of 0. This will remove the damping force that are present in “Sleep” mode by moving it in to “Force” mode. Next, try commanding a small force by passing a value of 1000. The shaft will move on its own to one end. Now try -1000. This will move the shaft in the opposite direction. Try increasing this value to see how the force the motor exerts increases.

Note: If the shaft is mounted vertically a larger force might be required to overcome gravity.

Step 6: Add Command for Reading Motor Errors

The last command we will implement in this tutorial will be an error command. When “error” is typed into the console, the motor’s active errors should be displayed. The meaning of each error code will also be displayed. We will implement the command such that it handles the following error codes:

1	Configuration invalid
32	Force control clipping
64	Max temp exceeded
128	Max force exceeded
256	Max power exceeded
512	Low shaft quality
1024	Voltage invalid
2048	Comms timeout

The sum of all active error codes can be retrieved using the `Actuator` object function `get_errors()`. Because the error code values increase exponentially, we can find each error code contained in the summation quite easily. If the largest error code value is larger than the active errors sum, it is contained in the sum, otherwise it isn’t. If an error code is contained in the sum, it must be subtracted from the sum. The remaining sum is compared to the next highest error code. This process is repeated for all error codes. The following code block implements this command.

```
COMMAND_IS "error" THEN_DO
// Print descriptions of error codes
IC4_virtual->print_l("Error   Flags:\r1-configuration   invalid\r32-force   control
clipping\r64-max temp exceeded\r128-max force exceeded\r256-max power exceeded\r512-
low shaft quality\r1024-voltage invalid\r2048-comms timeout");
IC4_virtual->print_l("\r\r");

// Get name and errors from Actuator object
IC4_virtual->print_l(motor.get_name());
uint16_t active_sum = motor.get_errors();
```

```
// Define error code to search for in the active error sum
uint16_t error_code_list[] = {2048, 1024, 512, 256, 128, 64, 32, 1};
int error_index = 0;

// Compare active error sum to each error code in list
IC4_virtual->print_l("\rActive Errors: ");
while( error_index < 8 ){
   uint16_t error_code = error_code_list[error_index];
   if(active_sum >= error_code){
      // If active error sum contains error code, print error code and subtract it from the active error sum
      IC4_virtual->print_d(error_code);
      IC4_virtual->print_l(", ");
      active_sum -= error_code;
   }
   error_index += 1;
}
IC4_virtual->print_l("\r\r");
```

After compiling and uploading these changes, running an “error” command will print any active errors in the motor to the console.

TUTORIAL 5: POSITION CONTROL

This tutorial demonstrates how to set the motor's position to follow a generated signal using the motor's position mode.

Step 1: Add Position Control Page

There are a couple libraries available that provide interfaces for updating the motor's position to follow a signal and update the PID gains. To make space for these interfaces, we should add another GUI page for position control. Set up a position control page class that takes an `Actuator` object as a constructor argument, like we did with the home page class. Again, use a `GUI_page` object to handle hiding / displaying the contents of the page. This page should also have a `Motor_Plot` as well as a `Dataset` that will be used to plot the target position signal. We will not require the `FlexSlider` or `FlexData` objects.

```
#ifndef _POSITION_CONTROL_PAGE_H_
#define _POSITION_CONTROL_PAGE_H_

class Position_Control_Page {

        Actuator& motor;
        Dataset position_signal;
        Motor_Plot motor_plot;
        FlexLabel page_label;
        GUI_Page  page_elements;
        bool first_setup = true;

    public:
        bool is_running = true;

        //Constructor
        Position_Control_Page(
            Actuator& _motor
        ):
            motor(_motor)
        {}

   void setup() {
      if (first_setup) {
         // Initialize GUI_Page object
         page_elements.add();
         motor_plot.add(&page_elements, &motor, "Orca Motor", 1, 35);

         page_label.add(&page_elements,       "<p_style=\"font-size:20px;\">Position
Control</p>", 1, 19, 2, 10);

         // Add position signal to plot
         position_signal.add(&motor_plot.plot,  "Position  Signal",  "Time",  "mm",
Dataset::TIMEPLOT + Dataset::NONE);
         position_signal.set_max_data_points(25000);
         position_signal.set_colour(GREEN);
         position_signal.show();
      }
      else {
         page_elements.show();
      }
      is_running = true;
   }
```

```
    void run() {
       if (!is_running) return;
       motor_plot.run();
    }

    //hide all elements on this page
    void hide() {
       page_elements.hide();
       is_running = false;
    }

    void reset() {
       hide();
       first_setup = true;
    }
};

#endif
```

In Main_GUI.h, include Position_Control_Page.h.

```
#include "Position_Control_Page.h"
```

When adding a new GUI page, we should also add a new page button that can be used to display the page. Declare a `Position_Control_Page` object and a `FlexButton` for the new page. The `Position_Control_Page` will update every GUI update period. Updating the motor's position from within the `Position_Control_Page run()` function would be too infrequent and the motor would experience jerky motion. Instead, we should update the position from the main `loop()` function in the .ino file. To do this, make sure the `Position_Control_Page` object is declared publicly. Initialize the `Position_Control_Page` object with the `Actuator` object in the GUI constructor.

```
class GUI : public IC4_k20 {

        Actuator& motor;
        uint32_t gui_timer = 0;
        uint8_t gui_update_period = 20;
        FlexButton home_page_btn, pos_ctrl_page_btn;

public:
        Home_Page home_page;
        Position_Control_Page position_control_page;

        int32_t target_force = 0;

        //Constructor
        GUI(
            Actuator& _motor
        ):
            motor(_motor),                                  // Initialization list
            home_page(motor),
            position_control_page(motor)
        {
            set_server_name("Tutorial 5");
            set_device_id("Eagle");
        }
```

The position control page `FlexButton` object must also be initialized. This button can be placed just to the right of the home button.

Now that we have two pages to click between instead of just one that is being toggled, we can make use of the GUI element disable function. This function can either allow or disallow user input to the GUI element. When this function is called from a `FlexButton` instance, it makes the button clickable or unclickable. Passing a value of "true" to this function makes it unclickable, while "false" makes it clickable. After initializing each button, we should make the home page button unclickable since the home page is displayed by default. We can do this by calling its `disable()` function and passing it the value "true".

```
void setup() {
   gui_set_grid(30, 60);
   print_l("New Connection Message\r");
   home_page.setup();
   home_page_btn.add("Home", -1, 26, 1, 2, 4);  // Initializes position page button
   pos_ctrl_page_btn.add("Position", -1, 26, 6, 2, 4);
   home_page_btn.disable(true);
}
```

Here, we would like each button to toggle on when clicked, after which it becomes unclickable until the other button has been clicked. When one button is clicked and toggled on, the other button is immediately toggled off and becomes clickable.

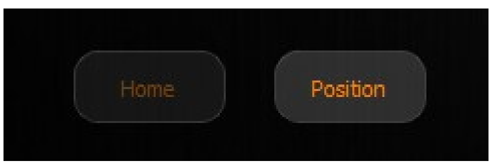

Figure 14: Disabled "Home" button

Our `frame_update()` function logic must be updated to handle two different buttons and pages. When switching between different GUI pages, if a page button is pressed, any other pages should be hidden before that page should be displayed. Additionally, the pressed page button should be disabled while all other page buttons should be enabled.

```
void frame_update() {
   if (home_page_btn.pressed()) {
      position_control_page.hide();
      home_page.setup();
      home_page_btn.disable(true);
      pos_ctrl_page_btn.disable(false);
   }
   if (pos_ctrl_page_btn.pressed()) {
      home_page.hide();
      position_control_page.setup();
      home_page_btn.disable(false);
      pos_ctrl_page_btn.disable(true);
   }
   home_page.run();
   position_control_page.run();
}
```

As we did with the `Home_Page` class before, call the `GUI_Page add()` function and call the `Motor_Plot add()` function to initialize each object. The `GUI_Page` object should also be passed to the `Motor_Plot add()` function.

Step 2: Add a Signal Panel

Start by including the signal panel library, Signal_Panel.h, in the Position_Control_Page.h file.

The `Signal_Panel` object will need to be declared within the `Position_Control_Page` class. Since we plan on updating the position from the main `loop()` function in the .ino file, the `Signal_Panel` object must be declared publicly.

As done with the previous panels, initialize the `Signal_Panel` by calling its `add()` function. The `Signal_Panel add()` function requires:

1. (optional) A `GUI_Page` object reference (GUI_Page&)
2. An initial value reference (&int32_t)
2. A y anchor (uint8_t)
3. An x anchor (uint8_t)

Since we are using the signal panel to update the motor's position, the initial value should be the starting position of the motor. We will make a property `signal_init_value` (int32_t) that will be updated by the motor position. When the signal panel is updated with a new signal, it starts generating signal values from the initial value, so that the motor doesn't experience an extreme position change when moving from its current position into the signal. Remember, the motor's position can be retrieved by calling the `Actuator` object's `get_position_um()` function.

```
#include "Signal_Panel.h"

class Position_Control_Page {

        Actuator& motor;
        Dataset position_signal;
        Motor_Plot motor_plot;
        FlexLabel page_label;
        GUI_Page page_elements;
        bool first_setup = true;
        int32_t signal_init_value;

    public:
        Signal_Panel signal_panel;
        .
        .
        .
```

Update the `setup()` function to include the signal panel to be added to our `GUI_page`. By doing so it will show and hide with the other elements on the position page. The `Signal_Panel add()` function optionally takes a `GUI_Page` object reference as the first argument and requires a row and column anchor value.

```
signal_panel.add(&page_elements, &signal_init_value, 5, 19);
```

The `Signal_Panel run()` function will manage the GUI elements on the panel. The `signal_init_value` should also be updated before each call to the `Signal_Panel run()` function as this is the value that the `Signal_Panel` object will use when initializing a new signal.

```
void run() {
   if (!is_running) return;
   // Update signal fields displayed depending on slider value
   signal_init_value = motor.get_position_um();
   signal_panel.run();
   // Update plot with motor data
   motor_plot.run();
}
```

The GUI will now have a display of a slider that can move through different signal types and provide input locations for the relevant initialization parameters.

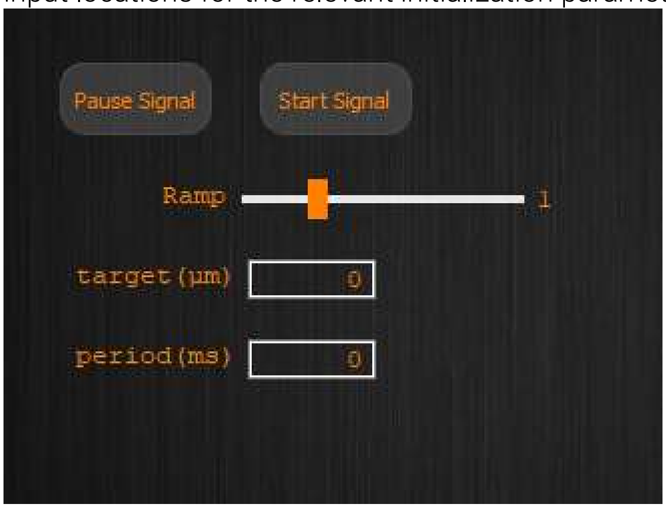

Figure 15: Signal Panel "Ramp" signal selected

Step 3: Generate Signal and Enter Position Mode

As of now, our signal panel interface is displayed, and a user can select between different signal types with various parameters. However, the signal is not yet being generated. The generation of the signal is separate from the GUI as it must be run very quickly it is done in the main `loop()` function in the .ino file. The `Signal_Panel generate()` function handles generating the specified signal. It also returns the generated value which can be used to update the motor's position. To set the motor's target position, we call the `Actuator` object's `set_position_um()` function.

```
void loop() {
   motor.set_force_mN(gui.target_force);
   motor.set_position_um(gui.position_control_page.signal_panel.generate());
   motor.run_in();
   motor.run_out();
   gui.run();
}
```

We can now also update the `position_signal` dataset with the current signal panel's target value and the IrisControls4 system time. The `Signal_Panel get_target_value()` function can be used to get the current value of the signal. The value will be plotted in micrometers.

```
void run() {
    if (!is_running) return;
        signal_panel.run_gui();
        signal_init_value = motor.get_position_um();
        position_signal.add_data(IC4_virtual->system_time(),
(int)(signal_panel.get_target_value()));
        motor_plot.run();
}
```

Until the motor's mode has been set to position mode the above command will be ignored. We can move the motor into position mode and set up the position controller at the bottom of the position control page setup function. For now, we will use a PID tuning of P:200, I:0, D:0, Saturation: 5000. This can be set by calling the `Actuator` object's `tune_position_controller()` function. This will use a spring like control that will have a maximum force value of 5000. More will be discussed about the PID tuning in the next tutorial. Make these changes in the `Position_Control_Page setup()` function such that this code runs regardless of the value of `first_setup`.

```
is_running = true;
motor.set_mode(Actuator::PositionMode);
motor.tune_position_controller(200,0,0,5000);
```

Now that we are putting the motor into position mode on the position page, the motor will remain in position mode while on the home page. To prevent this we can add a similar line in the home page `setup()` function to enter sleep mode.

```
motor.set_mode(Actuator::SleepMode);
```

The motor's positions should now follow the target signal.

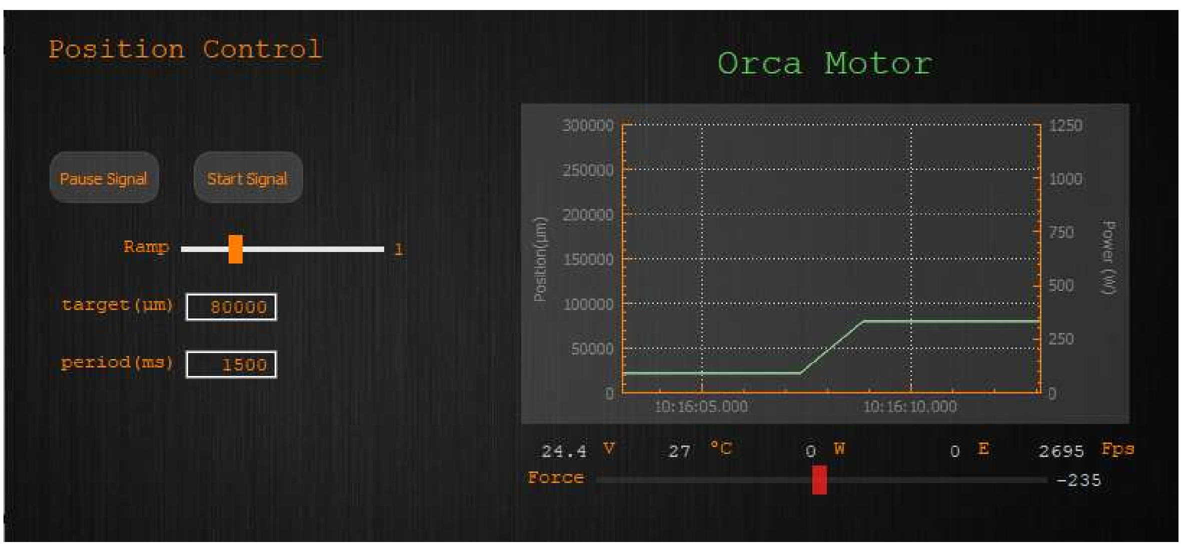

Figure 16: Motor following generated ramp signal

The signal panel provides functionality for the following signal types.

None: The target position will be the motor's current position.

Ramp: Move at a constant rate from current position to the target position over the set period

Square: Switch between a maximum and minimum target position with the set signal period

Triangle: Triangle waveform that moves at a constant rate between the maximum and minimum position with the set signal period.

Sine: Sinusoidal waveform with a specified signal amplitude and offset with the set signal period. The offset in this case is the minimum value of the signal offset from 0 so the center point of the signal would otherwise be the amp value.

REVISION HISTORY

Version	Date	Author	Reason
1.0.0	August 2022	RM, MA	Initial Release tutorials 0 - 5