WINDOWS SDK TUTORIALS

User Guide

Version 1.0.0

CONTENTS

For motor and programming environment set up see the Windows_QuickStart_Guide.

WARNING

Be aware that the shaft or motor will move during operation. This software will cause the motor to create forces and motion. Ensure the shaft and motor are mounted in a safe location and are not in danger of hitting people or property.

INTRODUCTION

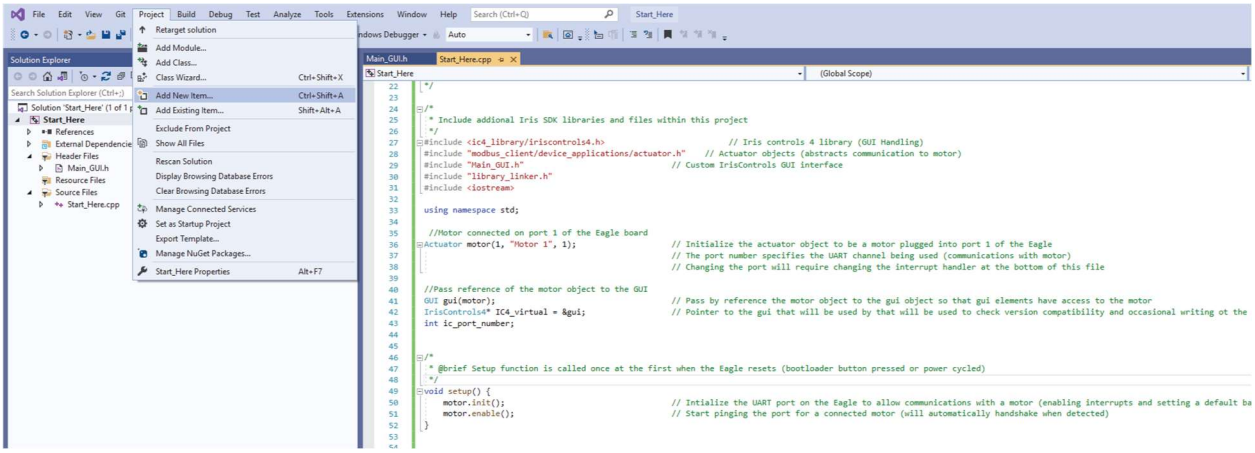
The goal of this guide is to guide developers through using the various features available on the Orca Series Motor and in the IrisControls4 library. Each tutorial is designed to begin where the previous one left off, so that they build on each other. While we encourage you to build each solution yourself, tutorial solutions are also included in this folder.

TUTORIAL 0: GENERAL OVERVIEW

A simple project is provided called “Start_Here”, that includes the basic project elements and will be the starting point of the first tutorial. The start here folder can be copied and renamed.

The project contains two files, start_here.cpp, which contains the main method and is the starting point of the application, and Main_GUI.h, which contains the IrisControls4GUI object.

The files can be selected from the Solution Explorer on the right, and navigated between by using the tabs along the top. If a new file is required in the project, select Project from the top menu, and then Add New Item.



The Start_Here.cpp 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.

Star_Here.cpp also prompts for the comport the user would like to use to communicate with IrisControls4. More information about how the Windows Virtual device communicates with IrisControls4 is available in the Windows QuickStart Guide, included in this repo.

Run this program by pressing the green triangle in the top bar. You can choose to run in either Debug mode or Release mode. More information about the two build configuration is available at

https://learn.microsoft.com/en-us/visualstudio/ide/understanding-build-configurations?view=vs-2022. Then run IrisControls4.exe. Select the COM port related to the Windows Virtual Device 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.cpp

```
/*
 * Include addional Iris SDK libraries and files within this project
 */
// Iris controls 4 library (GUI Handling)
#include <ic4_library/iriscontrols4.h>
// Actuator objects (abstracts communication to motor)
#include "modbus_client/device_applications/actuator.h"
// Custom IrisControls GUI interface
#include "Main_GUI.h"
//must be included in project to allow for library function implementations
#include "library_linker.h"
//used to accept console inputs from user.
#include <iostream>

using namespace std;

 //Motor connected on comport 67
// Initialize the actuator object to use a dummy comport
// Changing this port can be done using the set_new_comport function of the Actuator object.
Actuator motor(1, "Motor 1", 1);

// Pass by reference the motor object to the gui object so that gui elements have access to the motor
GUI gui(motor);
// Pointer to the gui that will be used by that will be used to check version compatibility and occasional writing to the console with errors
IrisControls4* IC4_virtual = &gui;
//comport used to communicate with iriscontrols4.
int ic_port_number;


/*
 * @brief Entry point for the application
 * Main loop that gets called continuously
 * Motor frames communication are done here, depending on the mode either Sleep, Force or Position will be commanded
 * Return frame contains information about motor position, force, temperature, power, errors.
 * Additional commands can be injected into the stream.
 */
int main() {

    cout << "Please enter the comport number you would like to use to communicate with Iris Controls and press enter: \n\n";
```

```
    //monitor for port number until one has been entered.
    while (ic_port_number == 0) {
        cin >> ic_port_number;
    }

    //call setup function, which will set up the connection to Iris Controls on the
selected comport
    IC4_virtual->setup(ic_port_number);

    //call setup and enable the motor
    //change the comport being used be the actuator object to the one that the RS422
cable is connected to
    motor.set_new_comport(67);
    //init the actuator object
    motor.init();
    // Start pinging the port for a connected motor (will automatically handshake when
detected)
    motor.enable();

    //main loop which will run while the application is open
    while (1) {
    // Parse incoming motor frames (Motor -> Windows Virtual Device)
        motor.run_in();
    // Send out motor frames (Windows Virtual Device -> Motor)
        motor.run_out();
    // Run IrisControls connectivity, GUI element interaction, serial parsing found in
Main_GUI.h
        gui.run();
    }
}
```

Main_GUI.h

```
//windows driver for iriscontrols4.
#include "ic4_library/device_drivers/windows/ic4_windows.h"
//IrisControls object
#include "ic4_library/iriscontrols4.h"
//actuator objects
#include "modbus_client/device_applications/actuator.h"
//needed for library function implementation
#include "library_linker.h"

/**
    @file Main_GUI.h
    @class GUI
    @brief Extension of the Windows device driver for the IC4 library, handles
connectivity with IrisControls4 and gui element interaction
*/
class GUI : public IC4_windows {
// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator& motor;
```

```
// This will keep track of the last update
    uint32_t gui_timer = 0;
// This is the time between updates in milliseconds (100 ms = 10 fps)
    uint8_t gui_update_period = 100;
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 receive feedback from the motor */
        Actuator& _motor
    ) :
        /* Initialization List */
        motor(_motor)
    {
        // Server name, changes the name in the initial connection message with
IrisControls
        set_server_name("Start Here");
        // Device ID, changes how the device will show up in the com port drop down
        set_device_id("Windows Virtual Device");
    }

    /*
        @Brief Handles connectivity and gui interaction with IrisControls
    */
    void run() {
     // Parses incoming serial communications (IrisControls -> Windows Virtual Device)
        check();
        switch (gui_frame_state) {
        case rx:
            // If connection times out, set disconnected
            if (is_timed_out()) {
                set_disconnected();
                reset_all();
            }
            break;

        case tx:
            // IRIS CONTROLS HAS ESTABLISHED CONNECTION
            if (new_connection()) {
                setup();
            }

            if (is_connected()) {                   // IRIS CONTROLS REGULAR UPDATES
                // When IrisControls is connected only update the gui at a specified
gui period (in milliseconds)
                // This dictates the frame rate of the GUI, if the gui period is too
large the gui might be choppy
                // If the period is too small there will be an unneeded amount of
serial traffic and could cause lag
                if ((uint32_t)(millis() - gui_timer) > gui_update_period) {
                    gui_timer = millis();

                    // Update GUI
                    frame_update();
```

```
                    // Transmit end of transmission message
                    end_of_frame();
                }
            }
            send();                                     // Sends anything in the transmit
buffer (Windows Virtual Device -> IrisControls)
            break;
        }//SWITCH
    }

private:

    /*
        @brief GUI initalization called from check for new IrisControls connection
    */
    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 Windows
Virtual Device
        print_l("Connected to IrisControls\r");
    }

    /*
        @brief action to be called every gui frame go here
        This is called inside the is_connected gui update loop
    */
    void frame_update() {
    }

    /** @brief Reset all gui elements */
    void hide_all() {

    }

    /** @brief Reset all gui elements */
    void reset_all() {

    }

};
```

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.

```
    /* 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),
        comport_select_panel(_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("Windows Device");
  }
```

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 run. We can see this change in the console output when a new connection is established with IrisControls4.

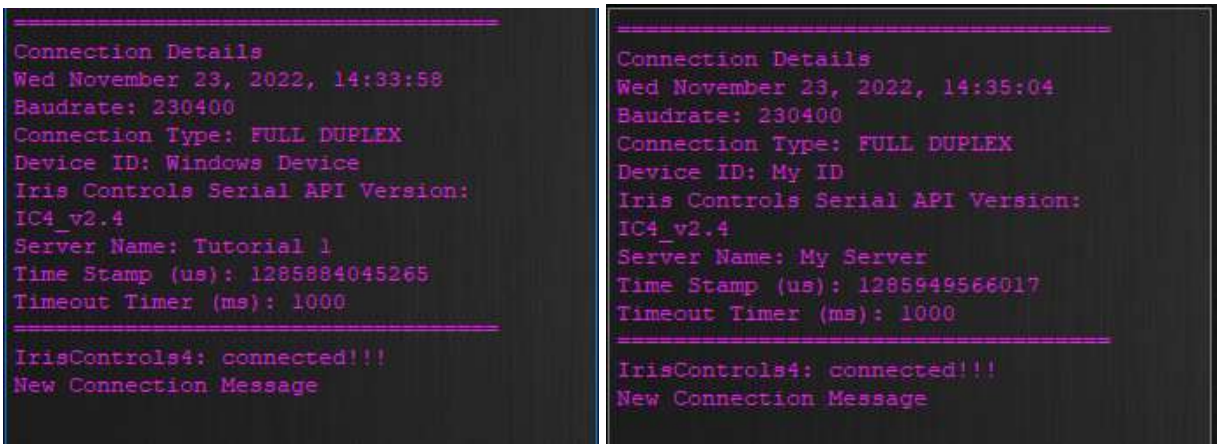


Figure 2: Change server name and device id

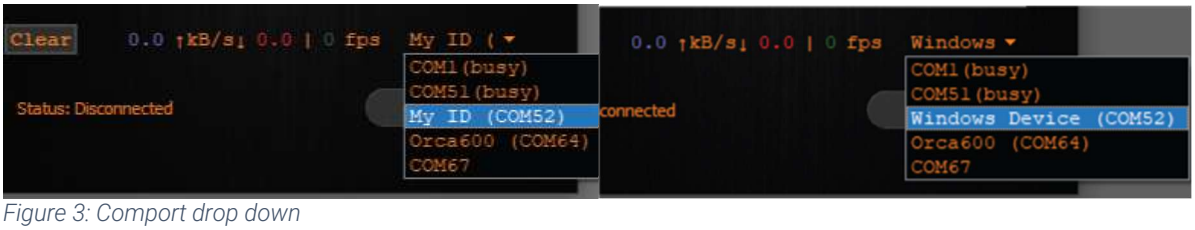


Figure 3: Comport drop down

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

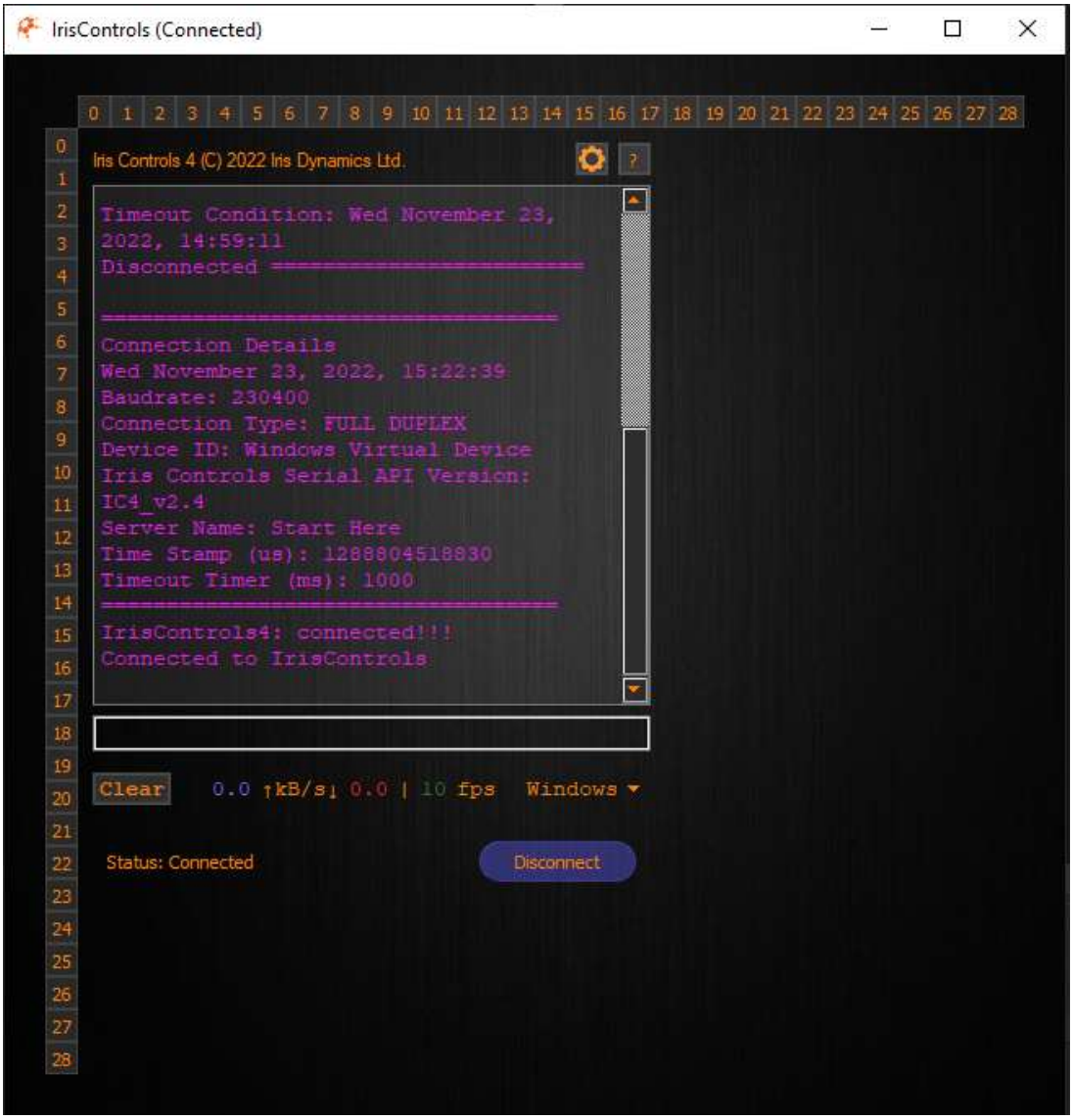


Figure 4: Turning grid guide on for a 30x30 window

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 Windows Virtual device and IrisControls4 has been established.

```
    void setup() {
        gui_set_grid(30, 30);                          // This will set the size of
the IrisControls window,
```

```
        print_l("Connected to IrisControls\r");         // Message is printed when
IrisControls establishes a connection with the Windows Virtual Device
    }
```

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

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

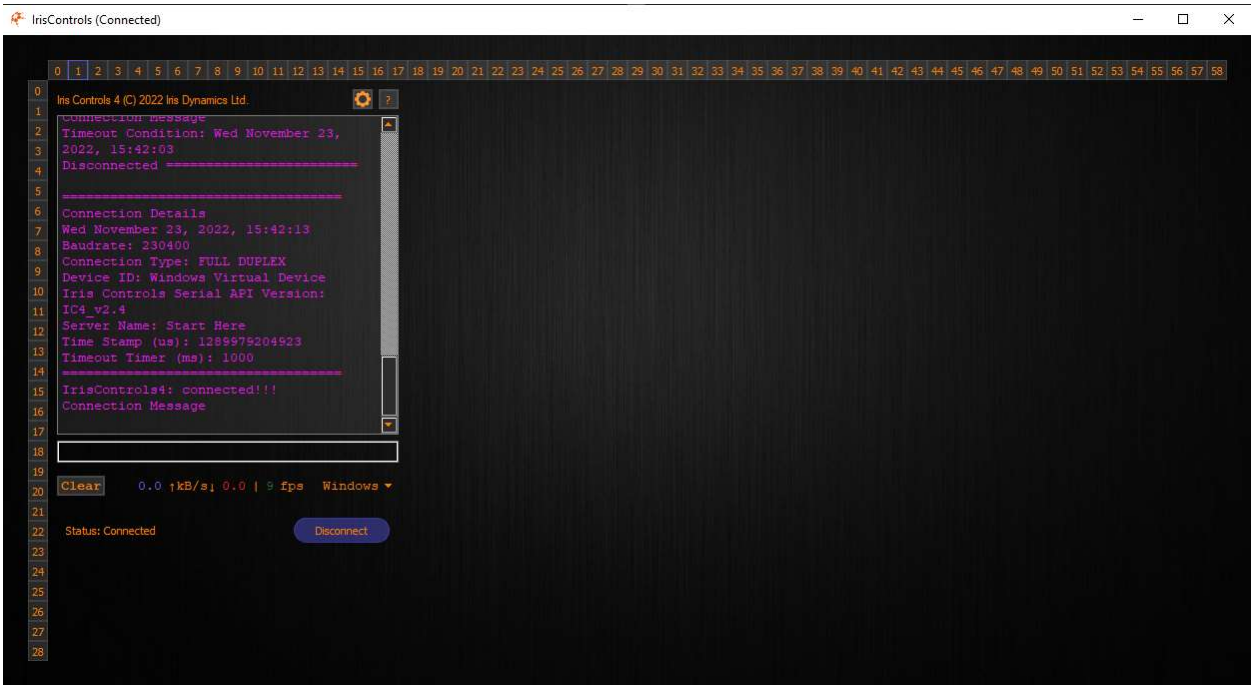


Figure 5: Update grid size and new connection message printed to console.

Step 3: Adding and Updating GUI elements

In our last example we hardcoded in the comport that was connected to the RS422 USB cable. Ideally, we would allow users to choose this comport each time the applications starts up, as this comport can change if a new cable or computer is used. The irisSDK_libraries submodule contains a file called Comport_Select.h, which can be used for this purpose.

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

The `add()` function used in this example takes the following arguments:

1.	A pointer to the first actuator of an array of actuators	(Actuator*)	_motor
2.	Anchor location on the y axis	(uint9_t)	3
3.	Anchor location in the x asix	(uint16_t)	20

The `Comport_Select` element includes a space for entering a comport, a Connect button and a Disable button. When the Connect button is pressed the comport associated with the passed in motor will be updated, and a connection will be attempted. When the Disable button is pressed the motor will be disabled and communications will be disconnected.

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.

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*:

4.	A string for the data element's name/label	(const char *)	"Position: "
5.	The row location	(uint16_t)	10
6.	The column location	(uint16_t)	19
7.	The row span	(uint16_t)	2
8.	The column span	(uint16_t)	10
9.	The initial value	(int)	0
10.	The denominator value	(uint16_t)	1
11.	A string for the units	(const char *)	"*mu*m"
12.	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.

```
    void setup() {
        gui_set_grid(30, 60);                           // This will set the size of
the IrisControls window,
        print_l("New Connection Message\r");            // Message is printed when
IrisControls establishes a connection with the device
        comport_select_panel.add(motor, 3, 20);
        position_element.add("Position: ", 10, 19, 2, 10, 0, 1, "*mu*m",
FlexData::UNITS); // 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.
    }
```

After building the project the GUI should look like the picture below. If you move the motor's shaft you should be able to see the position value update in the GUI.

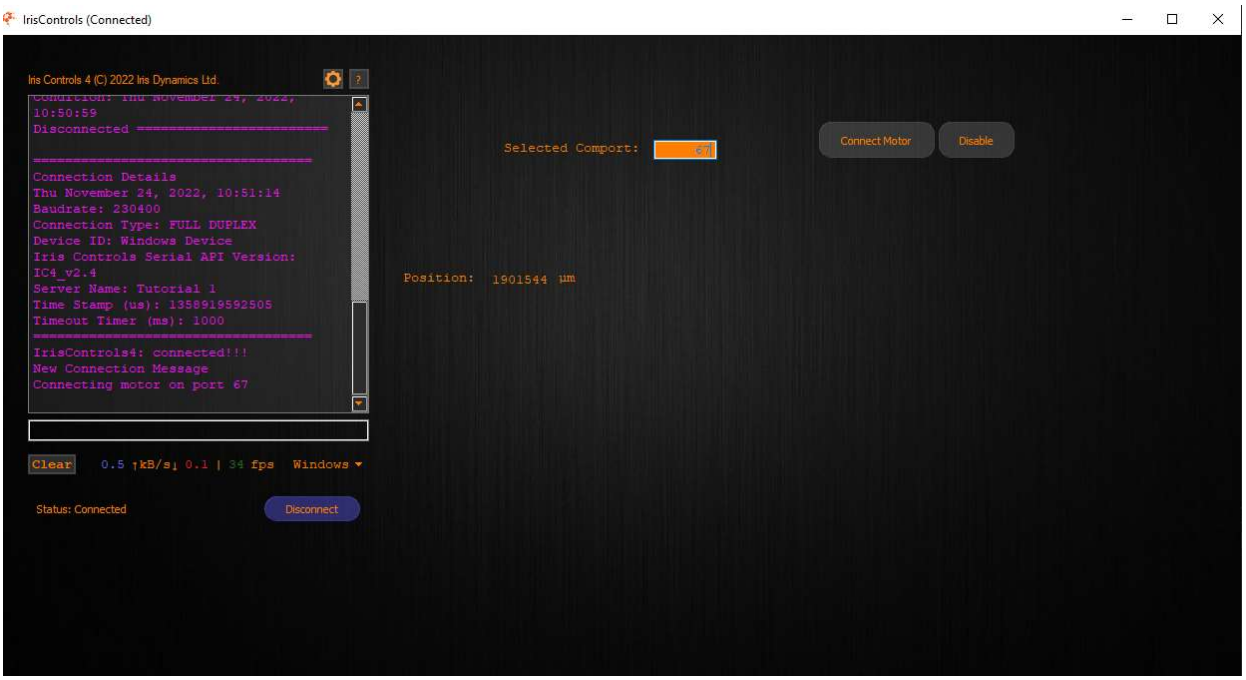


Figure 6: Display of Comport_Select and FlexData elements

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 Windows Virtual Device and IrisControls4 have established a connection, the FPS can be seen below the console in green.

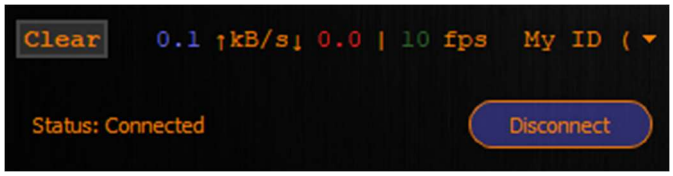


Figure 1: GUI frames per second display

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.

```
class Home_Page {
// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;
// Data element for displaying the motor's position
    FlexData position_element;
// Panel for selecting a comport to open a motor connection on
    Comport_Select comport_select_panel;

public:
// Boolean value tracks whether or not the page is active
    bool is_running = false;

    /* Constructor */
    Home_Page(
        /** 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),
        comport_select_panel(_motor)
    {}
```

There is also a `Comport_Select` elements used in the `Home_Page`.

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 initialization of the position element and comport select 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.

```
class Home_Page {
// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;
// Data element for displaying the motor's position
    FlexData position_element;
// Panel for selecting a comport to open a motor connection on
    Comport_Select comport_select_panel;

public:
// Boolean value tracks whether or not the page is active
    bool is_running = false;

    /* Constructor */
    Home_Page(
        /** 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),
        comport_select_panel(_motor)
    {}

void setup() {
            comport_select_panel.add(&page_elements, motor, 3, 20);

            // Initialize position data element
            position_element.add(&page_elements, "Position: ", 10, 19, 2, 10, 0, 1,
"*mu*m", FlexData::UNITS);  // Init element with a name, row and col anchors, row and
col spans, init value, denominator (can be used to divide a value for unit conversions,
using 1 has no effect), units, and configurations.

            // Initialize home page label element
            page_label.add(&page_elements, "<p_style=\"font-size:20px;\">Home</p>", 1,
19, 2, 9);
        is_running = true;
    }

    /**
         @brief Handles updating the position element
     */
    void run() {
        if (!is_running) return;
        comport_select_panel.run_gui();

        // Update position element with motor position
        position_element.update(motor->get_position_um());
    }
```

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 {
// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;
// Data element for displaying the motor's position
    FlexData position_element;
// Panel for selecting a comport to open a motor connection on
    Comport_Select comport_select_panel;

public:
// Boolean value tracks whether or not the page is active
    bool is_running = false;

    /* Constructor */
    Home_Page(
        /** 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),
        comport_select_panel(_motor)
    {}

void setup() {
            comport_select_panel.add(&page_elements, motor, 3, 20);

            // Initialize position data element
            position_element.add(&page_elements, "Position: ", 10, 19, 2, 10, 0, 1,
"*mu*m", FlexData::UNITS);  // Init element with a name, row and col anchors, row and
col spans, init value, denominator (can be used to divide a value for unit conversions,
using 1 has no effect), units, and configurations.

            // Initialize home page label element
            page_label.add(&page_elements, "<p_style=\"font-size:20px;\">Home</p>", 1,
19, 2, 9);
        is_running = true;
    }

    /**
         @brief Handles updating the position element
     */
    void run() {
        if (!is_running) return;
        comport_select_panel.run_gui();

        // Update position element with motor position
```

```
        position_element.update(motor->get_position_um());
    }

  /**
        @brief Hides all Home Page elements in GUI view
     */
    void hide() {
        page_elements.hide();

        is_running = false;
    }

    /**
        @brief Resets all Home Page elements in GUI view
    */
    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 <ic4_library/device_drivers/windows/ic4_windows.h>
#include "modbus_client/device_applications/actuator.h"
#include "Home_Page.h"                                      // Custom page displaying a
motor's position data

class GUI : public IC4_windows {

    Actuator* motor;                                        // Reference to an actor object
that will be passed in when this object is initialized
    uint32_t gui_timer = 0;                                 // This will keep track of the
last update
    uint8_t gui_update_period = 20;                         // This is the time between
updates in milliseconds (100 ms = 10 fps)
    Home_Page home_page;                                    // Define Home_Page object

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),
```

```
        home_page(_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 2");
        set_device_id("Windows Tutorial 2");
    }
```

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);                           // This will set the size of
the IrisControls window,
        print_l("New Connection Message\r");            // Message is printed when
IrisControls establishes a connection with the Windows device
        home_page.setup();
        reset_all();
    }
```

Now, if you build the project and connect with IrisControls4, 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_windows {

    Actuator* motor;                                    // Reference to an actor object
that will be passed in when this object is initialized
    uint32_t gui_timer = 0;                             // This will keep track of the
last update
    uint8_t gui_update_period = 20;                     // This is the time between
updates in milliseconds (100 ms = 10 fps)
    Home_Page home_page;                                // Define Home_Page object
    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);                           // This will set the size of
the IrisControls window,
        print_l("New Connection Message\r");            // Message is printed when
IrisControls establishes a connection with the Windows device
        home_page_btn.add("Home", 1, 26, 1, 2, 4);      // Initializes position page
button
        home_page.setup();
        reset_all();
    }
```

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() {
        //check if home page is toggled on
        if (home_page_btn.toggled()) {
            if (home_page_btn.get()) {
                home_page.setup();
            }
            else {
                home_page.hide();
            }
        }
        //call run on gui panels
        home_page.run();

    }

    /** @brief Hide all gui elements */
    void hide_all() {
        home_page.hide();
    }

    /** @brief Reset all gui elements */
    void reset_all() {
        home_page.reset();
    }
```

Build the project and reconnect with IrisControl4. Clicking the home button should not display and hide the position and comport elements.

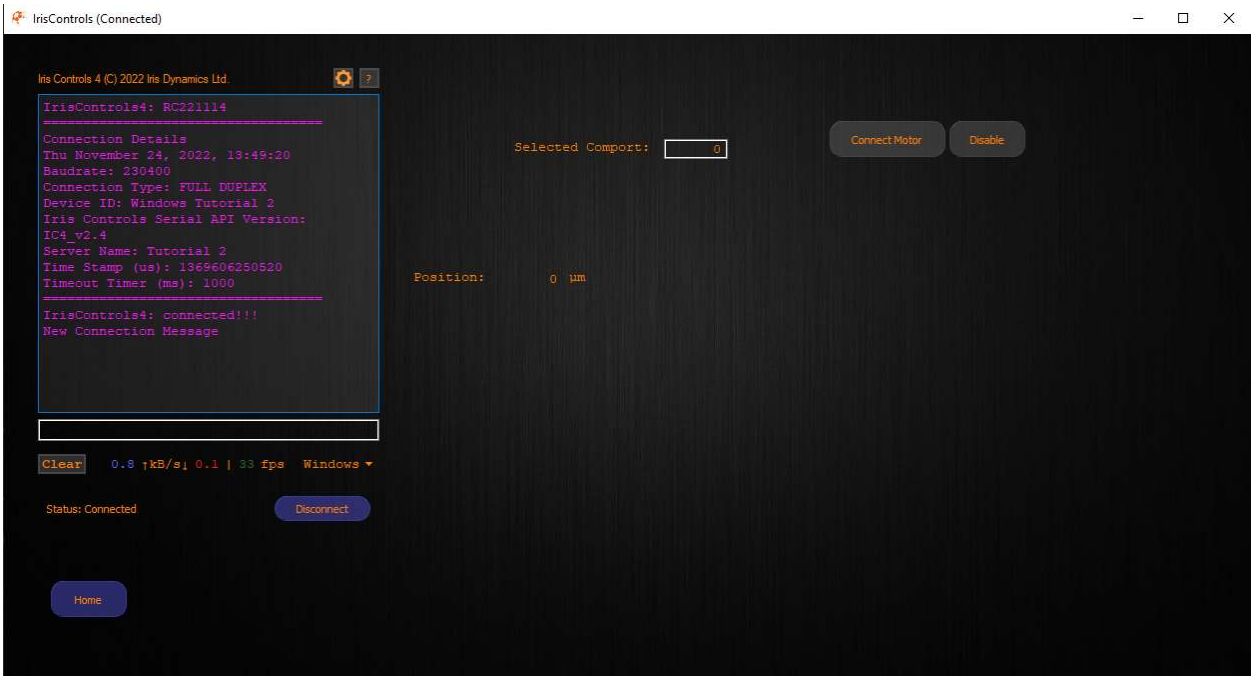


Figure 8: How Button for showing home page

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;                                    // Reference to an actuator
object that will be passed in when this object is initialized

    FlexLabel page_label;                               // Home page label element

    FlexData position_element;                          // Data element for displaying
the motor's position

    Comport_Select comport_select_panel;                // Panel for selecting a
comport to open a motor connection on
```

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.

```
    void setup() {
        if (first_setup) {
            first_setup = false;
```

```
            comport_select_panel.add(motor, 3, 20);

            // Initialize position data element
            position_element.add(&page_elements, "Position: ", 10, 19, 2, 10, 0, 1,
"*mu*m", FlexData::UNITS);  // Init element with a name, row and col anchors, row and
col spans, init value, denominator (can be used to divide a value for unit conversions,
using 1 has no effect), units, and configurations.

            // Initialize home page label element
            page_label.add(&page_elements, "<p_style=\"font-size:20px;\">Home</p>", 1,
19, 2, 9);
        }
        else {
            position_element.show();
            comport_select_panel.show;
        }
        is_running = true;
    }

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

Figure 2: 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;                                    // Reference to an actuator
object that will be passed in when this object is initialized

    FlexLabel page_label;                               // Home page label element

    FlexData position_element;                          // Data element for displaying
the motor's position

    GUI_Page  page_elements;                            // GUI_Page object to handle
displaying/hiding all elements in home page

    Comport_Select comport_select_panel;                // Panel for selecting a
comport to open a motor connection on
```

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. The `comport_select` element also has an `add()` function which takes a reference to a GUI_Page as an argument.

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.

```
/**
        @brief Handles initializing the home page
    */
    void setup() {
        if (first_setup) {
            first_setup = false;

            // Initialize GUI_Page object
            page_elements.add();

            comport_select_panel.add(&page_elements, motor, 3, 20);

            // Initialize position data element
            position_element.add(&page_elements, "Position: ", 10, 19, 2, 10, 0, 1,
"*mu*m", FlexData::UNITS);  // Init element with a name, row and col anchors, row and
col spans, init value, denominator (can be used to divide a value for unit conversions,
using 1 has no effect), units, and configurations.

            // Initialize home page label element
            page_label.add(&page_elements, "<p_style=\"font-size:20px;\">Home</p>", 1,
19, 2, 9);
        }
        else {
            page_elements.show();
        }
        is_running = true;
    }
```

```
    /**
         @brief Hides all Home Page elements in GUI view
     */
    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 “WindowsSDK_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 Windows Virtual device 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 "irisSDK_libraries/Motor_Plot_Panel.h"

class Home_Page {

// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;

// Motor plot object to display live motor data
    Motor_Plot motor_plot;

// Data element for displaying the motor's position
    FlexData position_element;

// GUI_Page object to handle displaying/hiding all elements in home page
    GUI_Page  page_elements;

// Panel for selecting a comport to open a motor connection on
    Comport_Select comport_select_panel;
```

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();

            comport_select_panel.add(&page_elements, motor, 3, 20);

            // Initialize motor plot
            motor_plot.add(&page_elements, motor, "Orca Motor", 13, 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;
        comport_select_panel.run_gui();
        // Update motor plot with motor data
        motor_plot.run_gui();
```

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 {

// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;

// Motor plot object to display live motor data
    Motor_Plot motor_plot;

// New dataset to add to plot: data will always take the new_data_slider slider value.
    Dataset new_data;

// Slider whose value is used as the new_data y-value
    FlexSlider new_data_slider;

// Home page label element
    FlexLabel page_label;

// Data element for displaying the motor's position
    FlexData position_element;

// GUI_Page object to handle displaying/hiding all elements in home page
    GUI_Page  page_elements;

// Panel for selecting a comport to open a motor connection on
    Comport_Select comport_select_panel;
```

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", 21, 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 build the project, you will see the slider value in the plot as displayed in the image below.

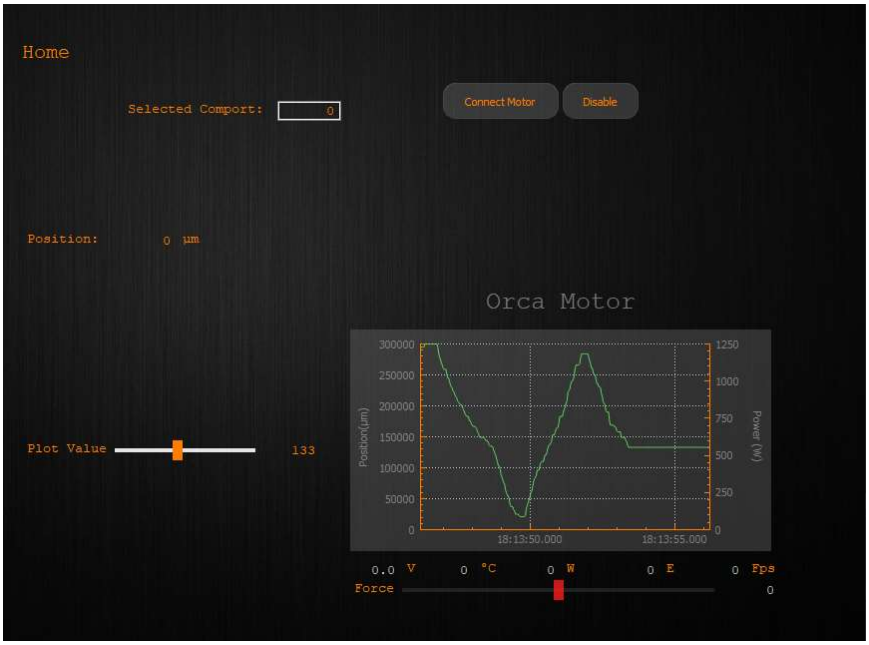


Figure 11: Slider value displayed on plot

Step 3: Connection Configuration

One of the values under the plot is the Fps which denotes the speed of messages between the Windows Virtual Device and Orca motor. This speed is determined during the handshake between the motor and device during initial connection. In this case default values for the baud rate and interframe delay are used.

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 before the loop begins in the application's `main()` function.

```
    //set up motor config
    motors[0].connection_config.target_baud_rate_bps = 1040000;
    motors[0].connection_config.target_delay_us = 0;
    motors[0].set_connection_config(motors[0].connection_config);
```

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.

```
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",
      with a white space char before each argument.
  */
        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.

```
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",
      with a white space char before each argument.
  */

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

Once you build the project and connect with IrisControls4, you can use the new command. If you enter “hello” into the console, “world” is printed in response.

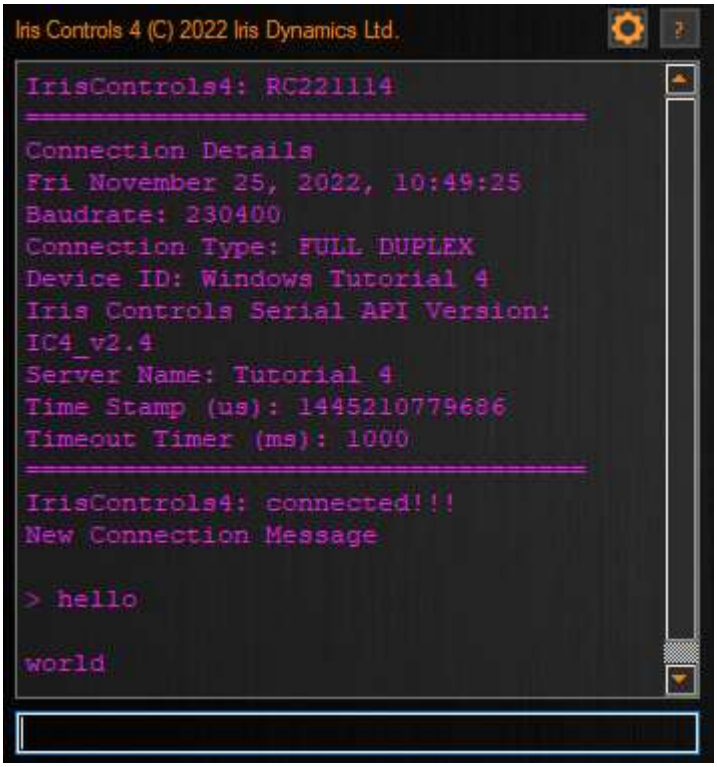


Figure 12: Hello world console parsing

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*.

```
// Command prints the values of the motors temperature, position, force, power , and
voltage.
        COMMAND_IS "get_data" THEN_DO
//Check to see if the motor is connected
            if (motor->is_connected()) {
// Print temperature from Actuator object
                print_l("\rTemperature (*deg*C): ");
                print_d(motor->get_temperature_C());
// Print position from Actuator object
                print_l("\rPosition (*mu*m): ");
                print_d(motor->get_position_um());
// Print force from Actuator object
```

```
                print_l("\rForce (mN): ");
                print_d(motor->get_force_mN());
// Print power from Actuator object
                print_l("\rPower (W): ");
                print_d(motor->get_power_W());
// Print voltage from Actuator object
                print_l("\rVoltage (mV): ");
                print_d(motor->get_voltage_mV());
            }
            else {
                print_l("\rPlease connect a motor to read data.");
            }
```

Rebuild the project 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 (by updating value in register 139)
        COMMAND_IS "max_temp" THEN_DO
            unsigned int arg_index = 0;
        if (motor->is_connected()) {
            uint16_t max_temp = parse_int(args, arg_index); // Get max temperature
value from argument list
            print_l("\rSetting max temp: ");
            print_d(max_temp);
            motor->set_max_temp(max_temp);
        }
        else {
            print_l("\rPlease connect a motor to set the 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 `main()` method loop.

```
public:
    int32_t target_force = 0;
// Define a target force variable. Initialize to 0, it will be updated if the user send
the force serial command "f"
```

Then in the loop in the `main()` method 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.

```
while (1) {
        //call run on the gui
        gui.run();

        //call run out and set_force_mN on the motor if it is enabled
        if (motors[0].is_enabled()) {
            motors[0].set_force_mN(gui.target_force);
            motors[0].run_out();
        }
        motors[0].run_in();
    }
```

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;
        if (motor->is_connected()) {
            target_force = parse_int(args, arg_index);
            print_l("\rTarget force: ");
            print_d(target_force);
            motor->set_mode(Actuator::ForceMode);
        }
        else {
            print_l("\rPlease connect a motor to set the target force.");
        }
```

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");
        //if a motor is connected, print any active errors
        if (motor->is_connected()) {
            IC4_virtual->print_l(motor->get_name());
            IC4_virtual->print_l("\rActive Errors: ");
            uint16_t active_sum = motor->get_errors();
            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
            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 rebuilding the project and connecting with IrisControls4, 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 of 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.

This example will require the use of a Signal_Generator object, which is found in the irisSDK_libraries folder. This object contains a run function, which should be called at the same frequency that run_in and run_out of the actuator object is called. It will also need to be references by the GUI objects, which run at a slower rate, so that the signals can be displayed graphically. So this reason we will declare the Signal_Generator object in the main cpp file of the project, and then pass it into the Main_GUI.h object, and form there pass into the position control page.

```
class Position_Control_Page {

// Reference to an actuator object that will be passed in when this object is
initialized
        Actuator& motor;
// Button for switching the motor's mode between 'Position' and 'Sleep'
        FlexButton sleep_btn;
// Motor plot object to display live motor data
        Motor_Plot motor_plot;
// Position Control page label element
        FlexLabel page_label;
// GUI_Page object to handle displaying/hiding all elements in position control page
        GUI_Page  page_elements;
// Boolean value tracks whether or not the page has been initialized yet
        bool first_setup = true;

    public:
        // Boolean value tracks whether or not the page is active
        bool is_running = false;

        /* Constructor */
        Position_Control_Page(
            /** 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,
            Signal_Generator& _signal_generator
        ) :
            /* Initialization List */
            motor(_motor),
```

```
            signal_panel(_signal_generator)
        {}

        /** @brief Handles initializing position control page elements */
        void setup() {
            if (first_setup) {

                // Update first_setup to indicate the setup for this page is complete
                first_setup = false;

                // Initialize GUI_Page object
                page_elements.add();

                // Initialize motor plot
                motor_plot.add(&page_elements, &motor, "Orca Motor", 1, 35);

                // Add position signal to plot
                position_signal.add(&motor_plot.plot, "Position Signal", "Time",
"*mu*m", Dataset::TIMEPLOT + Dataset::NONE);
                position_signal.set_max_data_points(25000);
                position_signal.set_colour(GREEN);
                position_signal.show();

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

            }
            else {
                page_elements.show();
            }
            is_running = true;
        }

            position_signal.add_data(IC4_virtual->system_time(),
(int)(signal_panel.get_target_value()));

            // Update plot with motor data
            motor_plot.run_gui();
        }

/** @brief Handles updating the position control page with motor data */
        void run() {
            if (!is_running) return;
            // Update plot with motor data
            motor_plot.run_gui();
        }

        /** @brief Hides all Position Control Page elements in GUI view */
        void hide() {
            if ( is_running ) {
                page_elements.hide();
                is_running = false;
            }
        }
```

```
        /** @brief Resets all Position Control Page elements in GUI view */
        void reset() {
            hide();
            first_setup = true;
        }

};
```

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. Initialize the `Position_Control_Page` object with the `Actuator` object in the GUI constructor

```
class GUI : public IC4_windows {
// Reference to an actor object that will be passed in when this object is initialized
    Actuator* motor;
// This will keep track of the last update
    uint32_t gui_timer = 0;
// This is the time between updates in milliseconds (100 ms = 10 fps)
    uint8_t gui_update_period = 20;
// Define Home_Page object
    Home_Page home_page;
// Define FlexButton element for hiding/displaying the home page
    FlexButton home_page_btn, pos_ctrl_page_btn;

public:
// Define Position_Control_Page object
    Position_Control_Page position_control_page;
// Define a target force variable. Initialize to 0, it will be updated if the user send
the force serial command "f"
   int32_t target_force = 0;


    /* 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),
        home_page(_motor),
        position_control_page(motor[0], _signal_generator)
    {
        /*  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 5");
        set_device_id("Windows Tutorial 5");
    }
```

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". To ensure each GUI page is reinitialized between connections to IrisControls4, the GUI `reset_all()` function should be called before the home page is initialized in the GUI `setup()` function.

```
    void setup() {
// This will set the size of the IrisControls window,
        gui_set_grid(40, 60);
// Message is printed when IrisControls establishes a connection with the Windows
device
        print_l("New Connection Message\r");
        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);
        reset_all();
// Initialize home page (landing page)
        home_page.setup();
    }
```

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.

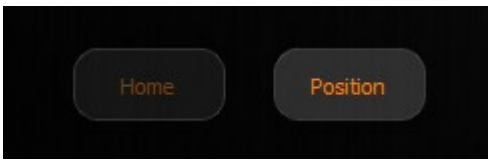


Figure 13: Disabled "Home" button

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() {
```

```
        //check if home page is toggled on
        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);
        }
        //call run on gui panels
        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. The signal panel can update an initial value until it can seamlessly convert into the specified signal. This is useful for position control as it prevents extreme changes in position when moving from some starting position into some signal. We will define a property `signal_init_value` (int32_t) for this that will be updated by the motor position. We will also define a property `signal_target_value` (int32_t) that we will use to update the motor's position.

```
class Position_Control_Page {
// Reference to an actuator object that will be passed in when this object is
initialized
        Actuator& motor;
// Button for switching the motor's mode between 'Position' and 'Sleep'
        FlexButton sleep_btn;
// Provides interface for entering signal type and parameters
        Signal_Panel signal_panel;
// Value used by the signal panel to track the motor's starting position, used to avoid
sudden extreme changes in position
        int32_t signal_init_value;
// Target position value
        int32_t signal_target_value;
// Dataset to add to plot. Data will always take the Signal_Panel signal value
        Dataset position_signal;
// Motor plot object to display live motor data
        Motor_Plot motor_plot;
// Position Control page label element
        FlexLabel page_label;
// GUI_Page object to handle displaying/hiding all elements in position control page
```

```
        GUI_Page  page_elements;
// Boolean value tracks whether or not the page has been initialized yet
        bool first_setup = true;

.
.
.
```

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. (optional) An initial value reference (&int32_t)
3. A y anchor (uint8_t)
4. An x anchor (uint8_t)
5. (optional) units (defaults to micrometers)

In the `Position_Control_Page setup()` function, initialize the `Signal_Panel`. Include the page's `GUI_Page` object as the `Signal_Panel add()` function's first parameter so it can handle hiding and displaying it along with the other page elements. The `signal_init_value` we just defined will be used as the initial value reference. Remember, we can give it the value of the motor's position by calling the `Actuator` object's `get_position_um()` function. Since we are updating the motor's position in micrometers, we do not need to specify a parameter for units and can use the default value.

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

The `Signal_Panel run()` function will manage the GUI elements on the panel and initialize signals when the "Start Signal" button is pressed. 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. We can do this in the `Position_Control_Page run()` function.

```
void run() {
            if (!is_running) return;

            // Update signal init value only if the signal panel is not paused
            if(!signal_panel.signal_generator.signal_paused){
                signal_init_value = motor.get_position_um();
            }

            // Update signal fields displayed depending on slider value
            signal_panel.run();

            // Update plot with motor data
            motor_plot.run_gui();
        }
```

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.

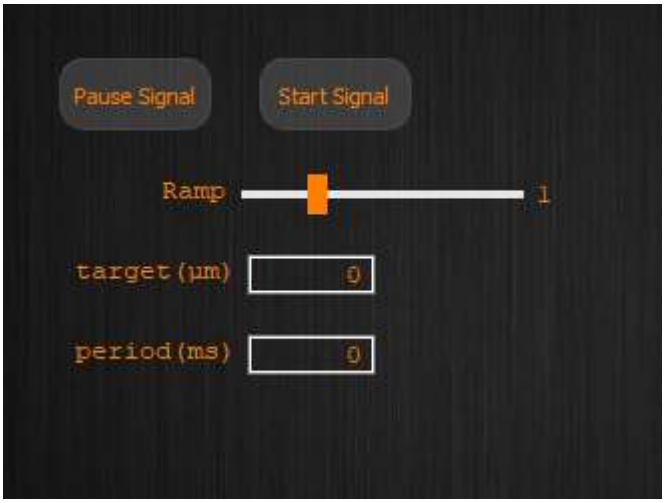


Figure 14: Signal Panel "Ramp" signal detected.

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 target position should be updated in the main loop, as this is being called at the same rate as run_in/run_out to the motors.

```
while (1) {
        //call run on the gui
        gui.run();

        // Set position using signal generator
        motors[0].set_position_um(signal_generator.run());

        //call run out and set_force_mN on the motor if it is enabled
        if (motors[0].is_enabled()) {
            motors[0].run_out();
        }
        motors[0].run_in();
    }
```

The `Signal_Panel run()` function generates the specified signal and returns the generated value we want to use to update the motor's position. To set the motor's target position, we call the `Actuator` object's `set_position_um()` function.

To pause the signal generation when the position control page is hidden, we can call the `Signal_Panel pause()` function from the `Position_Control_Page hide()` function.

```
        void hide() {
            if ( is_running ) {
                signal_panel.pause();
```

```
                page_elements.hide();
                is_running = false;
            }
        }
```

We can now also update the `position_signal` dataset with the current signal panel's target value and the IrisControls4 system time. This is done in the `run()` function of `Position_Control_Page.h` 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;

            // Update signal init value only if the signal panel is not paused
            if(!signal_panel.signal_generator.signal_paused){
                signal_init_value = motor.get_position_um();
            }

            // Update signal fields displayed depending on slider value
            signal_panel.run();

            // Update position signal dataset with signal value
            position_signal.add_data(IC4_virtual->system_time(),
(int)(signal_panel.signal_generator.get_target_value()));

            // Update plot with motor data
            motor_plot.run_gui();
        }
```

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: 10000. 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 10000. More will be discussed about PID tuning in future tutorials. 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, 10000);
```

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);
```

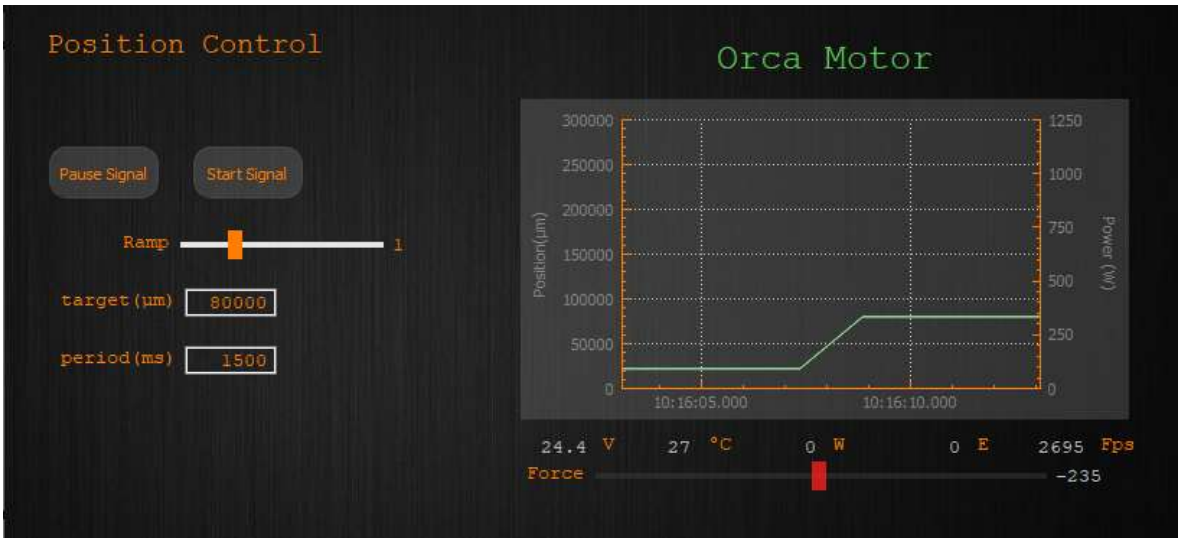


Figure 15: Motor following generated ramp signal

Now that the home page can update the motor's position by putting it in sleep mode, we should make sure there are no sudden changes in the motor's target position when the position control page is displayed. We can do this by updating the `signal_init_value` and `signal_target_value` with the motor's current position in the `setup()` function that runs when the page is displayed.

```
            else {
                signal_init_value = motor.get_position_um();
                signal_target_value = signal_init_value;
                page_elements.show();
            }
```

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.

TUTORIAL 6: PID GAIN TUNING

This tutorial demonstrates how a `Gain_Panel` object can be used to set the PID controller's proportional, integral, and derivative gain values as well as its maximum allowable force. This panel makes use of the Orca API `tune_position_controller()` function to update the values to the motor.

Step 1: Add Gain Panel

Include the `Gain_Panel` header file, `Gain_Panel.h`, at the top of `Position_Control_Page.h`

```
#include "irisSDK_libraries/Gain_Panel.h"
```

Declare a `Gain_Panel` object at the top of the `Position_Control_Page` class where the rest of the panels are declared.

```
class Position_Control_Page {
// Reference to an actuator object that will be passed in when this object is
initialized
        Actuator& motor;
// Interface for updating Motor PID gain and maximum force values
        Gain_Panel gain_panel;
```

Initialize the `Gain_Panel` by calling its `add()` function. The `Gain_Panel add()` function requires an `Actuator` object reference and a column and row anchor value. A `GUI_Page` object reference can also be provided optionally.

```
        void setup() {
            if (first_setup) {

                // Update first_setup to indicate the setup for this page is complete
                first_setup = false;

                // Initialize GUI_Page object
                page_elements.add();

                // Initialize gain panel
                gain_panel.add(&page_elements, &motor, 18, 19);
```

Next, call the `Gain_Panel run_gui()` function from the `Position_Control_Page run()` function. Here, the `signal_init_value` should also be updated with the motor's position if the signal panel is running.

```
        void run() {
            if (!is_running) return;

            // Update gain values if update button pressed
            gain_panel.run_gui();
```

Now, if you build the project and connect with IrisControl4, you should be able to see the gain panel below and signal panel on the position page.

Step 2: Add Sleep Button to Position Control Page

When tuning the position controller, it is possible to enter states of instability which can results in undesirable shaft behaviour. To mitigate this, we will add a "Sleep" button that puts the motor into sleep mode when it is toggled on. Start by declaring a `FlexButton` object for the sleep button in the `Position_Control_Page` class.

Initialize the sleep button by calling the `FlexButton add()` function. Do this in the `Position_Control_Page setup()` function. Notice the third argument, 0, indicates that the button will be toggled off by default.

```
        void setup() {
            if (first_setup) {

                // Update first_setup to indicate the setup for this page is complete
                first_setup = false;

                // Initialize GUI_Page object
                page_elements.add();

                // Initialize sleep button
                sleep_btn.add(&page_elements, "Sleep", 0, 3, 19 , 2, 4);
```

Update the `Position_Control_Page run_gui()` function to set the motor's mode based on the sleep button's toggle value.

```
void run() {
            if (!is_running) return;

            // Put motor into sleep mode if sleep button toggled on
            if(sleep_btn.get()){
                motor.set_mode(Actuator::SleepMode);
            }
            else{
                motor.set_mode(Actuator::PositionMode);
            }

            // Update gain values if update button pressed
            gain_panel.run_gui();

            // Update signal init value only if the signal panel is not paused
            if (!signal_panel.signal_generator.signal_paused) {
                signal_init_value = motor.get_position_um();
            }

            // Update signal fields displayed depending on slider value
            signal_panel.run();

            // Update position signal dataset with signal value
            position_signal.add_data(IC4_virtual->system_time(),
(int)(signal_panel.signal_generator.get_target_value()));

            // Update plot with motor data
            motor_plot.run_gui();
```

```
        }
```

Step 3: Use Default PID Gain Values

In this step we will find values for the PID controller's proportional gain, integral gain, and derivative gain. Proportional action is proportional to the distance between the motor's position and target position. The further the motor's current position is from its target position, the more force the controller will apply.

By increasing how frequently the motor plot is updated, we can see a more accurate plot of the motor's position. We can do this by decreasing the `gui_update_period` from 20ms to 5ms in Main_GUI.h.

```
uint8_t gui_update_period = 20;
```

You should see the default PID gain values in the panel as used in the previous tutorial: P gain = 200, I gain = 0, D gain = 0, and saturation = 10,000. Click the "Update" button to update each value in the Orca. Ensure that the Orca is connected to the Windows Virtual device and there are no errors. The easiest way to do this is to make sure the motor plot title is green.

A square wave signal will allow us to easily see the oscillation and overshoot of the motor's position in comparison to its target position. Using the default gain values, try running a square wave signal with a minimum target position of 10,000µm, a maximum target position of 30,000µm, and a period of 5,000ms. This will cause the shaft to change positions every 5 seconds.

Because the motor's movement will be approximately the same for each square wave cycle, after the first cycle we can stop the motor's movement and focus on the plot data. Once you have captured one square wave cycle, press "sleep" or "Pause Signal" and double-click on the motor plot to pause it. Using the scroll-wheel on your mouse, zoom in until you can distinguish between the motor's target and actual positions, as displayed in the screenshot below.

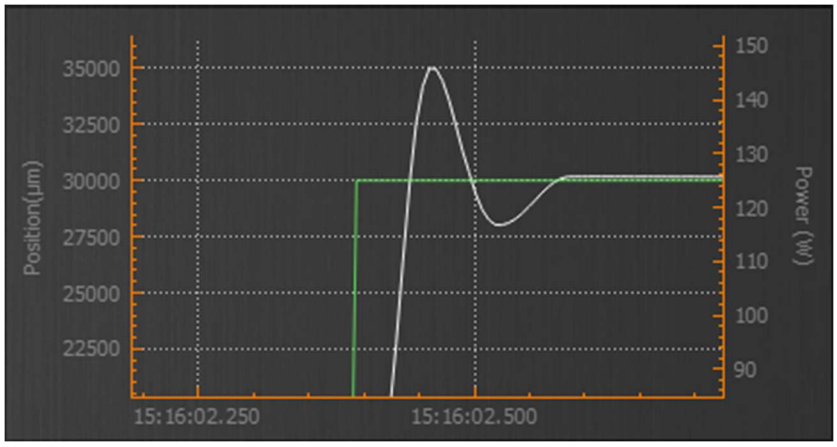


Figure16: Square Wave Response with Kp = 200

In the plot above, the motor's target position is the green line, and its actual position is the white line. You can see that the motor's position overshoots the target position. The next step in this

tutorial will demonstrate how to tune the gain values so that the motor's position more closely follows its target position.

Step 4: Tune PID Gain Values

There are multiple methods for tuning PID controllers and optimal tuning values will depend on the nature of the load and any disturbances experienced. While there are multiple formal tuning methods, we will simply demonstrate the effect of each gain to help us find a reasonable tuning. Note that the shaft used when writing this tutorial was mounted horizontally and did not need to overcome gravity when moving towards its target. The three gains we will be adjusting in this step are the gains for proportional action, integral action, and derivative action.

Derivative action is a force acting against the speed of the shaft. The action feels like a damper on the shaft and can prevent it from reaching high speeds. Note that this PID controller has two types of derivative control implemented: derivative of error, and derivative of velocity. The derivative of velocity is the derivative action we will be using in this tutorial.

Integral action takes any small errors in the position and accumulates action to correct them over time. Integral gain is inherently unstable and must be used in combination with proportional gain to prevent oscillation.

The following table summarizes how increasing each of the three gains independently will affect the motor's response to a disturbance.

Parameter	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor Change	Decrease	Decrease	No effect in theory	Improves if K_d is small

Effects of Increasing the Individual PID Gains

First, we should begin by increasing the proportional action so that the motor's position reaches its target more quickly. We can do this by increasing the proportional gain until oscillations can be observed in response to the step signal. The plot below shows the motor's response to a square wave after increasing the proportional gain, K_p to 800.

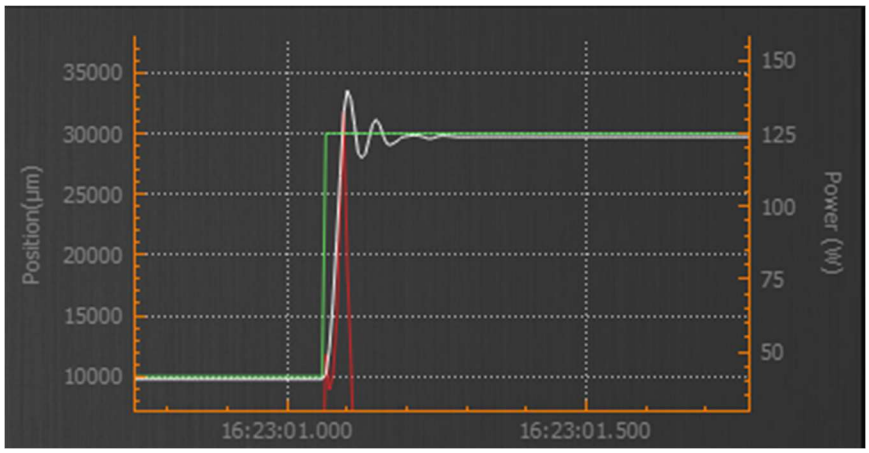


Figure 17: Square Wave Response with Kp = 800, K_d = 0, K_i = 0

This change causes the motor to reach its target more quickly, but it also causes a larger overshoot and more oscillations. The next step is to increase the derivative action to dampen the motor's response and reduce this overshoot. Increase the derivative gain until the oscillations go away and the response does not overshoot. The plot below shows the motor's response to a square wave after increasing the derivative gain, K_d to 30 (K_p is still 800).

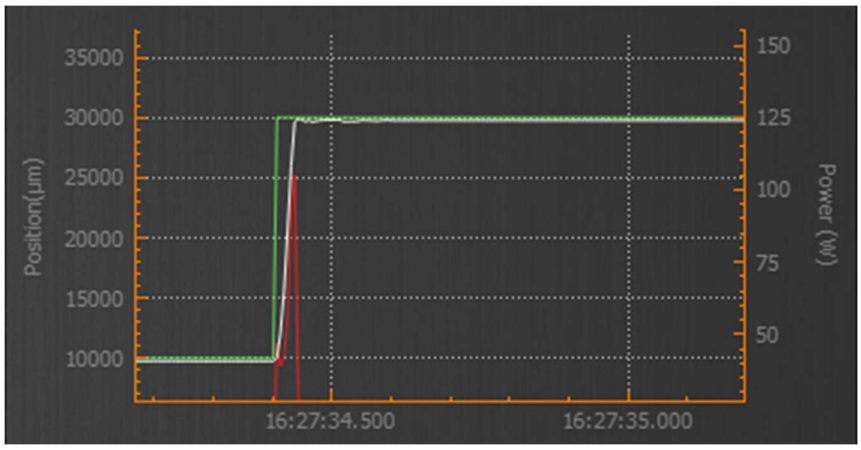


Figure 18: Square Wave Response with Kp = 800, K_d = 30, K_i = 0

Here, some tuning methods slowly increase K_p and K_d to find their maximum stable values. In our case, we can accept K_p = 800 and K_d = 30 as our stable values. If you zoom in on the above plot, you will see that the motor does not quite reach the target exactly but has a steady-state error of about 250µm. This can be seen in the zoomed-in plot below in figure 20.

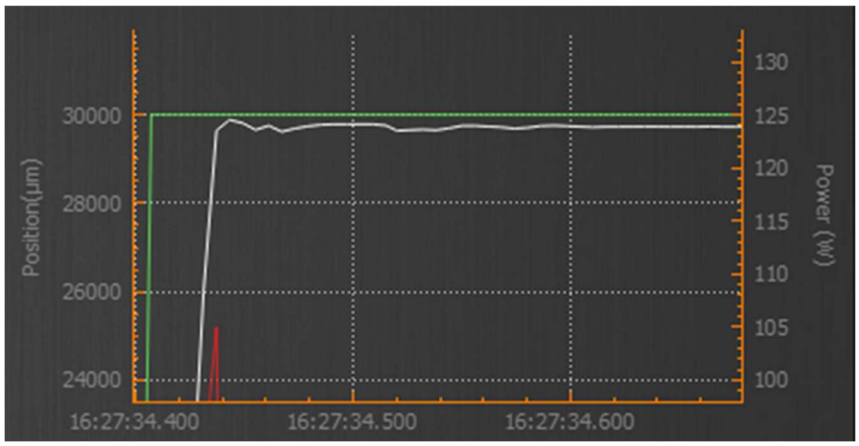


Figure 19: Square Wave Response Zoomed in with K_p = 800, K_d = 30, K_i = 0

Integral action can be used to mitigate this error and help ensure the motor's position reaches the target more closely. To do this we should increase the value of K_i. The plot below shows the motor's response to a square wave after increasing the integral gain, K_i, to 40 (K_p and K_d unchanged).

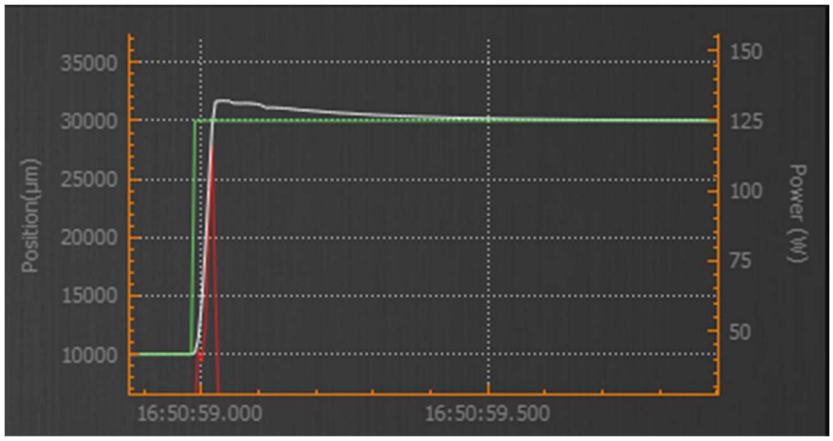


Figure 20: Square Wave Response Zoomed in with K_p = 800, K_d = 30, K_i = 40

You will notice that increasing the integral action causes a slight overshoot but also reduces the steady-state error from 250µm to about 16µm.

Experiment with the gain values for your setup to see what gives the best performance.

TUTORIAL 8: FORCE CONTROL

This tutorial demonstrates how to create a force control page and use it to update the motor's force. We will use the signal panel introduced in tutorial 6 as well as a force effect panel to generate a target force value

Step 1: Add Force Control Page

To make a new page for force control, create a new file called Force_Control_Page.h and copy the following code into it. You may notice that this class looks like the `Position_Control_Page` class without the `Gain_Panel` or target position `Dataset`. Instead, it has a `Dataset` property `force_data` that we will update later in this tutorial. The signal panel is also initialized differently here than how it was for position control. It has no initial signal value because force control does not require as smooth of a transition in target values the same way position control does. Additionally, it's initialization specifies millinewtons ("mN") to be used as units.

```
#ifndef _FORCE_CONTROL_PAGE_H_
#define _FORCE_CONTROL_PAGE_H_

#include "irisSDK_libraries/Signal_Panel.h"
#include "irisSDK_libraries/Motor_Plot_Panel.h"
#include "irisSDK_libraries/Force_Effect_Panel.h"

/**
    @file Force_Control_Page.h
    @class Force_Control_Page
    @brief Provides an interface for updating an Orca motor's force using a generated
signal and force effects
*/
class Force_Control_Page {

        Actuator& motor;
        Force_Effect_Generator& ff_gen;
// Interface for updating the motor's force using a generated signal and force effects
        Force_Effect_Panel force_effect_panel;
// Button for switching the motor's mode between 'Force' and 'Sleep'
        FlexButton sleep_btn;
// Provides interface for entering signal type and parameters
        Signal_Panel signal_panel;
// Dataset to add to plot. Data will always take the sum of the Signal_Panel signal
value and the Force_Effect_Panel value
        Dataset force_data;
// Motor plot object to display live motor data
        Motor_Plot motor_plot;
// Force Control Page label element
        FlexLabel page_label;
// GUI_Page object to handle displaying/hiding all elements on force control page
        GUI_Page  page_elements;
// Boolean value tracks whether or not the page has been initialized yet
        bool first_setup = true;

    public:
        // Boolean value tracks whether or not the page is active
        bool is_running = false;
```

```
        /* Constructor */
        Force_Control_Page(
            /** 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,
            Force_Effect_Generator& _ff_gen,
            Signal_Generator& sig_gen
        ):
            /* Initialization List */
            motor(_motor),
            ff_gen(_ff_gen),
            signal_panel(sig_gen)
        {}


        /** @brief Handles initializing force control page elements */
        void setup() {
            if (first_setup) {

                // Update first_setup to indicate the setup for this page is complete
                first_setup = false;

                // Initialize GUI_Page object
                page_elements.add();

                // Initialize force effect panel
                force_effect_panel.add(&page_elements, &ff_gen, 17, 19);

                // Initialize sleep button
                sleep_btn.add(&page_elements, "Sleep", 0, 3, 19, 2, 4);

                // Initialize signal panel
                signal_panel.add(&page_elements, 5, 19, "mN");

                // Initialize motor plot
                motor_plot.add(&page_elements, &motor, "Orca Motor", 1, 36);

                // Add target force dataset to plot
                force_data.add(&motor_plot.plot, "Target Force", "Time", "Force(mN)",
Dataset::TIMEPLOT + Dataset::NONE +  Dataset::SECONDARY_Y_AXIS);
                force_data.set_max_data_points(25000);
                force_data.set_colour(BLUE);
                force_data.show();

                // Update plot axis by replacing power with force
                motor_plot.plot.set_axes_labels(&force_data);
                motor_plot.plot.set_secondary_range(-15000, 15000);

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

            }
            else {
```

```
                page_elements.show();
            }
            is_running = true;
            motor.set_mode(Actuator::ForceMode);
        }


        /** @brief Handles updating the force control page with Orca data and running
the signal and force effect panels. */
        void run_gui() {
            if (!is_running) return;

            // Put motor into sleep mode if sleep button toggled on
            if(sleep_btn.get()){
                motor.set_mode(Actuator::SleepMode);
            }
            else{
                motor.set_mode(Actuator::ForceMode);
            }

             // Update force effect objects with slider values
             force_effect_panel.run();

            // Update signal fields displayed depending on slider value
            signal_panel.run();

            // Update target force dataset with target force value
            force_data.add_data(IC4_virtual->system_time(), ff_gen.get_target_force() +
signal_panel.get_target_value());

            // Update plot with motor data
            motor_plot.run_gui();
        }


        /* Hides all force control page elements in GUI view */
        void hide() {
            if ( is_running ){
             //   signal_panel.pause();           // Pause signal panel
                force_effect_panel.pause();     // Pause force effect panel
                page_elements.hide();           // Hide elements
                is_running = false;             // Set page status to not running
            }
        }


        /* Resets all force control page elements in GUI view */
        void reset() {
            hide();
            first_setup = true;
        }
};
```

Now, we can use this page in the main GUI by including the `Force_Control_Page` header file, `Force_Control_Page.h`, at the top of `Main_GUI.h`.

```
#include "Force_Control_Page.h"
```

This page requires a reference to a `Force_Effect_Generator` object, which we will use in the main loop. Declare this object in the main loop, and then pass it into the `Main_GUI` when it is created. The `Main_GUI` will then pass the `Force_Effect_Generator` to the `Force_Control_Page`.

As we did with the rest of the pages we have implemented, we will add a button for displaying the force control page. Declare both a `Force_Control_Page` and a `FlexButton` for this in the GUI class.

```
class GUI : public IC4_windows {
// Reference to an actuator object that will be passed in when this object is
initialized
    Actuator* motor;
// This will keep track of the last update
    uint32_t gui_timer = 0;
// This is the time between updates in milliseconds (100 ms = 10 fps)
    uint8_t gui_update_period = 20;
// Define Home_Page object
    Home_Page home_page;
// Define FlexButton element for hiding/displaying the home page
    FlexButton home_page_btn, pos_ctrl_page_btn, force_ctrl_page_btn;

public:
// Define Position_Control_Page object
    Position_Control_Page position_control_page;
    Force_Control_Page force_control_page;
// Define a target force
//Define a target force variable. Initialize to 0, it will be updated if the user send
the force serial command "f"

    int32_t target_force = 0;


     /* 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,
        Signal_Generator& _signal_generator,
        Force_Effect_Generator& _ff_generator
    ) :
        /* Initialization list */
        motor(_motor),
        home_page(_motor),
        position_control_page(motor[0], _signal_generator),
        force_control_page(motor[0], _ff_generator, _signal_generator)
    {
```

```
        /*  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 7");
        set_device_id("Windows Tutorial 7");
    }
```

Initialize the `Force_Control_Page` by calling its constructor from the `GUI` constructor with an `Actuator` object reference, a `Signal_Generator` reference, and a `Force_Effect_Generator` reference.

```
     /* 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,
        Signal_Generator& _signal_generator,
        Force_Effect_Generator& _ff_generator
    ) :
        /* Initialization list */
        motor(_motor),
        home_page(_motor),
        position_control_page(motor[0], _signal_generator),
        force_control_page(motor[0], _ff_generator, _signal_generator)
    {
        /*  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 5");
        set_device_id("Windows Tutorial 5");
    }
```

The `FlexButton` for toggling on the force control page must also be initialized. We can initialize it with the other page buttons by calling its `add()` function.

```
       // Initializes page buttons
        home_page_btn.add("Home", -1, 26, 1, 2, 4);
        pos_ctrl_page_btn.add("Position", -1, 26, 6, 2, 4);
        force_ctrl_page_btn.add("Force", -1, 26, 11, 2, 4);
```

Next, we should update the `GUI frame_update()` function to display the force control page when this button is toggled. We should also make sure the force control page is hidden when either the home page or position page button is toggled.

```
    void frame_update() {
        //check if home page is toggled on
        if (home_page_btn.pressed()) {
            position_control_page.hide();
            force_control_page.hide();
            home_page.setup();
            home_page_btn.disable(true);
```

```
            pos_ctrl_page_btn.disable(false);
            force_ctrl_page_btn.disable(false);
        }
        if (pos_ctrl_page_btn.pressed()) {
            home_page.hide();
            force_control_page.hide();
            position_control_page.setup();
            home_page_btn.disable(false);
            pos_ctrl_page_btn.disable(true);
            force_ctrl_page_btn.disable(false);
        }
        if (force_ctrl_page_btn.pressed()) {
            home_page.hide();
            position_control_page.hide();
            force_control_page.setup();
            home_page_btn.disable(false);
            pos_ctrl_page_btn.disable(false);
            force_ctrl_page_btn.disable(true);
        }
        //call run on gui panels
        home_page.run();
        position_control_page.run();
        force_control_page.run_gui();
    }
```

Update the `GUI hide_all()` and `reset_all()` functions to hide and reset the force control page along with the other pages.

```
/** @brief Hide all gui elements */
    void hide_all() {
        home_page.hide();
        position_control_page.hide();
        force_control_page.hide();
    }

    /** @brief Reset all gui elements */
    void reset_all() {
        home_page.reset();
        position_control_page.reset();
        force_control_page.reset();
    }
```

Lastly, we can update the IrisControls4 connection message to include a description of the force control page. This is the done in the `setup()` function of Main_GUI.h.

```
print_l("Home Page: Sleep mode with motor plot panel and slider to plot additional
data\r Position Page: Position mode configurable signal following\r Force Page: Force
mode configurable signal and force effect following\r");
```

If you build the project with these changes, clicking on the “Force” button will bring up the force page.

Step 2: Add Force Effect Panel

Now, we can make use of the force effect panel to update the motor's force. The force effect panel uses a combination of constant force, spring, and damping effects. Include the `Force_Effect_Panel` header file, `Force_Effect_Panel.h`, at the top of `Force_Control_Page.h`.

```
#include "irisSDK_libraries/Force_Effect_Panel.h"
```

Declare a `Force_Effect_Panel` object in the `Force_Control_Page`.

Next, initialize the panel by calling its `add()` function. The `Force_Effect_Panel add()` function requires three arguments and will accept a reference to a `GUI_Page` object as an optional fourth.

1. (optional) `GUI_Page` object that all panel elements will be added to
2. The `Actuator` object that enables connection to the Orca motor
3. A row anchor (uint8_t)
4. A column anchor (uint8_t)

The panel should be initialized in the `Force_Control_Page setup()` function.

```
        void setup() {
            if (first_setup) {

                // Update first_setup to indicate the setup for this page is complete
                first_setup = false;

                // Initialize GUI_Page object
                page_elements.add();

                // Initialize force effect panel
                force_effect_panel.add(&page_elements, &ff_gen, 17, 19);
```

If you build the project now you will see a force effect panel in the GUI that contains four sliders. The `Force_Effect_Panel` has a `run_gui` function which handles the panel's interface. It should be called every GUI update period from the `Force_Control_Page run_gui()` function.

```
        void run_gui() {
            if (!is_running) return;

            // Put motor into sleep mode if sleep button toggled on
            if(sleep_btn.get()){
                motor.set_mode(Actuator::SleepMode);
            }
            else{
                motor.set_mode(Actuator::ForceMode);
            }

             // Update force effect objects with slider values
             force_effect_panel.run();
```

Step 3: Update Target Force in Motor

We declared the Force_Effect_Generator object in the main cpp file so that we can call it's run method at the same frequency as the main loop. In the main loop we will check to see what mode the motor is in, and if it is in force made we will set update the motors force target force. We are using the sum of the force effect objects (constant force, spring force and damping force), returned by the `run()` method of the `Force_Effect_Generator` object, plus the signal value, returned in the `run()` function of the `Signal_Generator` object, as the target force.

```
while (1) {
        //call run on the gui
        gui.run();
        // Set position using position control page

        //call run out and set_force_mN on the motor if it is enabled
        if (motors[0].is_enabled()) {
            switch (motors[0].get_mode()) {
            case Actuator::PositionMode:
                motors[0].set_position_um(signal_generator.run());
                break;
            case Actuator::ForceMode:
                motors[0].set_force_mN(ff_generator.run() + signal_generator.run());
                break;
            }

            motors[0].run_out();
        }
        motors[0].run_in();
    }
```

If you build the project now you will be able to use the signal panel and force effect panel together to update the force of the motor.

Step 4: Add Target Force to Plot

As we did with the target position on the position control page, we can add the target force to the force control page's motor plot for better visibility. The `force_data Dataset` property has already been defined for this purpose. We can initialize it by calling it's `add()` function from the `Force_Control_Page setup()` function. Again, we pass it a reference to the `Motor_Plot plot`. This time, however, we should update the title and axes labels to reflect force control. Since force control uses much lower target values, we must also specify the `SECONDARY_Y_AXIS` configuration value to plot our data against the axis on the right-hand side of the plot. Like before, we should set the maximum number of stored data points using the `Dataset set_max_points()` function, and we can also set the color of the plotted data using the `Dataset set_colour()` function. Finally, we call the `Dataset show()` function to display the data on the plot.

```
                // Initialize motor plot
                motor_plot.add(&page_elements, &motor, "Orca Motor", 1, 36);

                // Add target force dataset to plot
                force_data.add(&motor_plot.plot, "Target Force", "Time", "Force(mN)",
Dataset::TIMEPLOT + Dataset::NONE +  Dataset::SECONDARY_Y_AXIS);
```

```
                force_data.set_max_data_points(25000);
                force_data.set_colour(BLUE);
                force_data.show();
```

Now that the force value will be displayed on the plot, we should make sure the plot has a proper range and axes labels. To update the axes labels, use the `FlexPlot set_axes_labels()` and pass a reference to the `force_data Dataset` as an argument.

We'll keep position as the primary (left-hand side) axes range and update the secondary one. To update the secondary range, use the `FlexPlot set_secondary_range()` function and pass the range minimum as the first argument and the range maximum as the second.

```
                // Add target force dataset to plot
                force_data.add(&motor_plot.plot, "Target Force", "Time", "Force(mN)",
Dataset::TIMEPLOT + Dataset::NONE +  Dataset::SECONDARY_Y_AXIS);
                force_data.set_max_data_points(25000);
                force_data.set_colour(BLUE);
                force_data.show();

                // Update plot axis by replacing power with force
                motor_plot.plot.set_axes_labels(&force_data);
                motor_plot.plot.set_secondary_range(-15000, 15000);
```

To update the `force_data Dataset` each GUI update period, we should change its value in the `Force_Control_Page run_gui()` to update it each GUI update period. As before, we can do this using the `Dataset add_data()` function which takes an x-axis value (time) and a y-axis value (target force). We can call `system_time()` for the x-axis value, and use the value of the `target_force` property for the y-axis value.

```
        void run_gui() {
            if (!is_running) return;

            // Put motor into sleep mode if sleep button toggled on
            if(sleep_btn.get()){
                motor.set_mode(Actuator::SleepMode);
            }
            else{
                motor.set_mode(Actuator::ForceMode);
            }

             // Update force effect objects with slider values
             force_effect_panel.run();

            // Update signal fields displayed depending on slider value
            signal_panel.run();

            // Update target force dataset with target force value
            force_data.add_data(IC4_virtual->system_time(), ff_gen.get_target_force() +
signal_panel.get_target_value());

            // Update plot with motor data
```

```
            motor_plot.run_gui();
        }
```

Now when the force control page is used to update the Orca motor' target force, you will see the target force value on the motor plot in green.

REVISION HISTORY

Version	Date	Author	Reason
1.0.0	November 2022	KC	Initial Draft