HILO Software Documentation

Last updated: 16.03.2019

HILO App

The HILO app is designed as a user-friendly software to control a HILO spinning machine. The functionality is limited (for now) to controlling the thickness and amount of twist of spun yarn. Users can select a HILO device connected to their computer (via a USB serial port) or use a simulator. The app is written in Processing and can be launched from the Processing IDE, or exported as a stand-alone app for MacOS, Windows or Linux.

Keyboard Controls

The app is made to be used with the mouse or touchpad. A few keyboard shortcuts were left in, which can be helpful when using or debugging.

- **c** - connect to HILO
- **C** (uppercase, Shift + c) - disconnect from HILO
- **Spacebar** - start or stop spinning
- **l** (lowercase L) - print the list of available serial ports to the console (for debug)
- **I** (uppercase i, Shift + i) - load a new image (only in pattern mode)

Settings Files

The app is provided with a set of configuration files, within the **`data/settings/`** folder. The files are written in plain text using a JSON format. They can be opened and edited using a simple text editor (e.g. TextEdit, Notepad) and are easy enough to understand and modify. The original names and locations should be kept, as the software is looking for these exact files at the exact location. Descriptions of each file and contents follow.

App Settings

Filename: **`HILO_AppSettings.json`**

This file contains basic settings for the app's behavior and appearance. This includes the size of the viewport window and a reference for the file from which the app's text is loaded.

- **`prefPort`**: the name of the preferred serial port, within double quotes. For instance **`"COM4"`** on Windows or **`"/dev/tty.usbserial001"`** on Mac OS. You can leave it blank, as just a pair of double quotes **`""`**. Whenever you successfully connect to a HILO machine via serial port, the software sets that port as the preferred port and updated the settings file.

- **`fullscreen`**: if set to **`true`** (no quotes) the app will start in fullscreen mode and ignore the **`width`** and **`height`** settings. If set to **`false`** (no quotes) the app's window size is determined by **`width`** and **`height`**.
- **`langFile`**: the name of the file (in quotes) containing the app's text, by default **`"HILO_AppText_EN.json"`**. It would be possible to use a different file, so the app can be translated to other languages.
- **`width`** and **`height`**: positive integer numbers determining the size of the app's window, in pixels. If the **`fullscreen`** option is set to **`true`** then **`width`** and **`height`** are ignored.

Machine Settings

Filename: **`HILO_MachineSettings.json`**

This file contains settings for the HILO machine. Every time a connection is established, these values are set in the machine. The values can't be changed in the UI. They are pre-determined through tests using the in-house test app.

- **`stepsPerCm`**: number of steps in the drafting motor corresponding to one centimeter of yarn. A positive floating point number, such as **`12.34`** or **`8.0`**. You can determine this number by dividing the total amount of steps in a full rotation (typically 200) by the perimeter of the drafting roll, in centimeters.
- **`deliverySpeedSteps`**: the speed in steps per second (positive integer value, like **`300`**) for the delivery speed. Essentially, the speed at which the machine is running.
- **`draftingSpeedPercMax`**: the highest value for the speed of the drafting roll, as a percentage of the delivery speed. A positive integer between 0 and 100, like **`70`**. This corresponds to the drafting speed for thick yarn.
- **`draftingSpeedPercMin`**: the lowest value for the speed of the drafting roll, as a percentage of the delivery speed. A positive integer between 0 and 100, like **`20`**. This corresponds to the drafting speed for thin yarn.
- **`spindleSpeedStepsMax`**: the speed in steps per second (positive integer value, like **`700`**) for the spindle, corresponding to the maximum amount of twist in spin mode.
- **`spindleSpeedStepsMin`**: the speed in steps per second (positive integer value, like **`200`**) for the spindle, corresponding to the minimum amount of twist in spin mode.

App Text

Filename: **`HILO_AppText_EN.json`**

This file contains the text (*strings*) used in the HILO app, for the English language. The values are sufficiently self-explanatory. If you need to adjust the wording in the app, you can make changes to this file.

Note: making significant changes may break the UI layout, e.g., text may be misaligned, cut, or otherwise not properly displayed.

This file is referenced in the app settings file. You could create other app text files for other languages. To use one of them, you would have to replace the filename in the app settings. This cannot be changed from within the app itself.

Note: the character set loaded by the app to draw text is limited, so text in other languages (using “special” or language-specific characters) will not display properly. This can be improved.

Code Structure

The app is written as a Processing “sketch”, using the Processing IDE and framework, which is Java-based. It can be run from the Processing IDE or exported as a stand-alone app for Mac, Windows or Linux.

Processing organizes sketch code in “tabs”, which correspond to files in the project (extension *.pde*, Java code as plain text). When running/exporting, the contents of these tabs/files are all bundled into one Java class, with the same name as the sketch, and subclassing the **`PApplet`** class.

The code is commented. Each tab includes a description of it’s contents at the top. Classes and functions are also commented, with remarks when necessary; as are variable declarations or blocks thereof. Function bodies may include comments for operations which aren’t obvious or trivial.

For a quick overview of the overall structure of the app, we consider the app’s main functions (conforming to the typical structure of a Processing sketch), the HILO interface (which is used to connect to and control a HILO device) and the app’s UI.

Main Structure

A Processing sketch has an inherent structure and utility functions, as a subclass of PApplet. Code for initializing the app goes inside **`settings()`** and **`setup()`**. Code which runs every frame goes inside **`draw()`**. Keyboard and mouse events are handled in **`mousePressed()`**, **`mouseDragged()`**, **`mouseReleased()`**, **`mouseWheel()`** and **`keyPressed()`**. These can be found in the main tab, named **`HILO_App`**.

Most of the global variables and objects can also be found at the top of this tab, with the exception of UI variables and objects (those are in the **`UI_All_Layout`** tab). This includes:

- the objects which store the settings for the app and machine, and the app's text strings: **`appSettings`**, **`machineSettings`** and **`appStrings`**
- the handler objects for the UI and HILO machine event callbacks: **`appUICallbackHandler`** and **`appHILOCallbackHandler`**
- the object which controls a HILO device (machine or simulator) simply named **`hilo`**

Classes and methods pertaining to the user interface (UI) can be found in tabs prefixed with **`UI_`**, such as **`UI_BaseClasses`**. UI variables and objects, as well as the top-level methods for creating the layout and UI elements are in the tab **`UI_All_Layout`**. The handler object for UI events, however, is defined in the tab **`App_Callbacks_UI`**.

Classes and methods for interfacing with a HILO device (machine or software simulator) be found in tabs prefixed with **`HILO_`**, such as **`HILO_BaseClasses`**. The handler object for HILO events, however, is defined in the tab **`App_Callbacks_HILO`**, together with other functions that control or react to changes in the machine's state.

User Interface

The layout of the UI is based on a mixture of fixed values and reactive behaviour - that is, the position and size of some elements is calculated depending on the size of the app's window.

Some of the UI elements are based on visual assets (e.g. icons) which can be found in the folder **`data/icons/`** folder. There are different asset folders (**`small`**, **`medium`** and **`large`**) corresponding to different window sizes. The fonts used by the app can be found within the **`data/fonts/`** folder. Fonts are in a vector-format which (contrary to the icons) can be scaled for different sizes without a loss in quality.

The variables, objects and UI setup functions can be found in the **`UI_All_Layout`** tab; with special attention to the **`uiSetup()`** function, where UI elements are created and laid-out.

The app's UI is built from a series of UI elements. These are objects descending from class **`UIBasicElement`** or otherwise conforming to the **`UIElement`** interface (see the tab **`UI_BaseClasses`**). Each UI element can be assigned a callback handler (**`UICallbackHandler`** object or a descendant, see the same tab) which gets notified of changes in the element and acts accordingly. For instance: when a slider knob is moved, the slider element notifies its handler via the **`callbackUISliderDragged()`** callback, and the handler object changes the appearance of another UI element, or triggers an update on the HILO machine.

Classes defining UI elements can be found in tabs with the prefix **`UI_`** and a tab may contain more than one class - for instance **`UI_BaseClasses`** or **`UI_Text`**. The actual handler used by the app for UI events is an **`AppUICallbackHandler`** (extending the original **`UICallbackHandler`**) and can be found in the tab **`App_Callbacks_UI`**.

HILO Devices

A HILO device is represented as an abstract in the app, as a **`HILOInterface`** (found in the tab **`HILO_BaseClasses`**). This defines a common set of operations for any HILO device, and allows us to use an actual device (communicating through a serial port); or a “simulator” so that we can test and demonstrate the app without needing a machine or other hardware. In the future we may wish to define other devices such as remote/networked HILO machines and clusters, or different versions of the HILO machine.

The simulator acts for most (but not all) purposes like an actual machine. As mentioned above, it is meant to replace an actual machine during app testing and quick demos. The simulator class **`HILOSimulator`** is defined in the tab **`HILO_Simulator`**, whereas the actual HILO (which connects to a serial port and communicates with a machine using the HILO protocol) is implemented as class **`HILODevice`** in the tab **`HILO_Device`**.

A device or simulator should be assigned a callback handler (implementing the **`HILOCallbackHandler`** interface, see the **`HILO_BaseClasses`** tab) which gets notified of changes in the state of HILO and acts accordingly. The actual handler used by the app for HILO events is an **`AppHILOCallbackHandler`** (implementing the **`HILOCallbackHandler`** interface) and can be found in the tab **`App_Callbacks_HILO`**.

Testing App

The testing app is based on the second prototype for the HILO app, before UX research. It is developed for in-house use or demonstrations only and is not meant to be shared, exported or published. The test app is developed as a Processing sketch and can be run from within the Processing IDE. It requires the *controlP5* library, which can be downloaded through the *Contributions Manager* tool within the Processing IDE.

The test app was developed for in-house use and early demos at events, and is based on the pattern or effect use-case. The user loads an image, changes the image’s sampling (resolution) and the result is used to drive the HILO machine. The controls aren’t user-friendly but rather more specific, allowing for testing the machine with different settings and determine adequate configuration values for the user-friendly HILO App. You can control the machine’s delivery speed, spindle speed and elevator speed (in newer hardware). Speed values are in steps/second. The pixel length slider, in millimeters, will only be accurate if the hard-coded value **`ROLL_PERIMETER`** is accurate. Please refer to Settings (hard-coded) below.

In the older machine’s firmware, the spindle speed value in steps per second (in a range from 0 to 1000) is mapped to the range of the analog values in Arduino (0 to 255), to control the DC motor.

Keyboard Controls

- **c** - connect to HILO
- **C** (uppercase, Shift + c) - disconnect from HILO
- **p** - spin the next pixel and stop
- **Spacebar or P** (uppercase, Shift + p) - start spinning until stopped or reaching the end of the image
- **r** - reset the “shuttle” or cursor in the image to the starting position
- **Arrow keys** - move the “shuttle” or cursor in the image; left and right move the cursor to the previous and next pixel in the direction of spinning (instead of always left and right)
- **d** - show the original picture
- **D** (uppercase, Shift + d) - show a preview of the textile piece; currently broken
- **S** (uppercase, Shift + s) - upload the configuration to the machine; for debug purposes only, as the configuration is uploaded on connect and everytime a setting is changed in the UI

Settings (hard-coded)

As this is an in-house test Processing sketch, there is no configuration file. You are free to change/adjust some of the definitions within the code itself. At the top of the main tab you may find some of the most important ones: **`ROLL_PERIMETER`**, **`STEPPER_TOTAL_STEPS`** and **`STEPPER_MAX_SPEED`**. You should measure the perimeter of the delivery roll and write in an accurate number. These values are used to calculate the timing/size of each “pixel” spun by the machine.

In the first lines of the **`setup()`** function you can also change the width and height of the window, by editing the numbers inside the call to **`size()`**. If you want to run the sketch in full-screen, then replace **`size(width, height)`** with **`fullscreen()`**.

Serial Communication Protocol

The HILO machine’s “brain” is an Arduino-based board. The original prototype used an Arduino Uno model with a custom-made shield, whereas the new prototype uses an Arduino Mega 2560 with a RAMPS v1.4 shield. The machine can be controlled by sending simple text-based messages to it via the Arduino’s serial port (USB) running at a speed of 115200 Baud. This is commonly done by an app running a graphic user interface (GUI) with buttons and sliders, but you can also control the machine “directly” by opening a serial port terminal/monitor (such as the *Serial Monitor* tool in the Arduino IDE) and typing in the messages yourself.

The protocol defining the messages and behavior is rather simple. The user or app sends a command to the machine; and the machine replies with a conformation or error. For instance, the app sends a command to start spinning with the drafting speed at 40% of the delivery speed, and the machine confirms that it started spinning:

- app sends message **`[R,40]`** which means “run at 40% drafting speed”
- machine replies with **`[R]`** which means “I’m running”

Each message (command or reply) is contained in square brackets and consists of an upper-case character and one or two optional parameters (integer numbers).

Once a command is sent, the machine will typically reply with the same command (upper-case character), without parameters, as above. If an error occurs, it replies with an error message, which contains an error code number - for instance **`[E,6]`** when receiving an unknown command. Please refer to the firmware for each code’s description.

A machine programmed with a debug-enabled firmware will also send debug messages starting with the character **`X`** such as **`[X,handleCommandRun(): HILO is already spinning]`**. These are just meant to provide feedback when testing, and are otherwise inconsequential. The HILO app (or rather, the **`HILODevice`** class used in the app) prints these messages to the console.

Command	**Meaning and reply**	**Example** **(command/reply)**

`[R,n]`	Run the machine, with a drafting speed as a percentage of the delivery speed, as an integer number between 0 and 100. Can also be called while the machine is running, to change the drafting speed.
	Reply: **`[P]`** in case of success, or **`[E,n]`** in case of error (number or value of parameters, or the elevator is resetting).
		`[R,70]`
		`[R]`

`[S]`	Stop the machine. This can be used while the machine is spinning or resetting the elevator. **Reply:** **`[S]`**.	**`[S]`** **`[S]`**

`[D,n]`	Set the speed of the delivery motor(s) in steps/second.
	Reply: **`[D]`** in case of success, or **`[E,n]`** in case of error (number of parameters or value).
		`[D,300]`
		`[D]`

`[P,n]`	Set the speed of the spindle motors in steps/second.
	Reply: **`[P]`** in case of success, or **`[E,n]`** in case of error (number of parameters or value).
		`[P,550]`
		`[P]`

`[L,n]`	Set the speed of the elevator motor(s) in steps/second. If the value is negative, the elevator changes direction.	**`[L,-200]`**
		`[L]`

	Reply: `[L]` in case of success, or `[E,n]` in case of error (number or value of parameters, or the elevator is resetting).	

`[V]`	Reset the elevator to a position near the top.
	Reply: `[V]` in case of success, or `[E,n]` in case of error (the machine is currently running). Once the elevator is reset, the machine sends the message `[S]` (stopped)
		`[V]`
		`[V]`
		…
		`[S]`

`[A]`	Ping the machine, which can be used to check if the a HILO machine is connected to a serial port and responding properly. **Reply:** `[A]`.	`[A]` `[A]`

`[T]`	Request the machine's current state.
	Reply: either `[S]`, `[R]` or `[V]` when stopped, running or resetting the elevator (respectively).
		`[T]`
		`[S]`

`[I]`	Request information about the machine. Mostly useful for debug purposes, or to check which version of the firmware is running in a machine. **Reply:** a debug message like `[X,HILO Firmware 0.1.0. State S. Delivery speed -240. Spindle speed 700. Elevator height 0]`	

Firmware

The firmware for the HILO machine - that is, the software which runs on the machine itself - is developed using the Arduino IDE and meant to be used in Arduino-compatible boards. The firmware uses the AccelStepper library by Mike McCauley, which can be installed via the *Library Manager* tool in the Arduino IDE.

Support for Old Hardware

One of the firmware versions is written to support the original HILO prototype as well as the new machines. This inevitably makes the code harder to read, change and maintain, as the older machine has significant differences. As such, the open firmware version only supports the new machine.

When uploading the firmware to an old machine you should first uncomment the line containing **`#define OLD_HARDWARE`** found near the top of the main tab **`HILO_Firmware`** in the Arduino sketch. Throughout the code there are blocks delimited by code like **`#ifdef OLD_HARDWARE`** and **`#endif`** (these are compiler directives in C/C++) which indicate which parts of the code get compiled and uploaded to the Arduino board, depending on whether the line **`#define OLD_HARDWARE`** is commented or not.

Sketch Structure

The code is commented. Each tab includes a description of it's contents at the top. Variable declarations (or blocks thereof) and functions are also commented, with remarks when necessary. Function bodies include comments for operations which aren't obvious or trivial. The following is a quick description of the contents of each tab.

- **`HILO_Firmware`** is the sketch's main tab, which has the same name as the sketch. It contains definitions for general settings, for the communications protocol and global variables, followed by Arduino's setup() and loop() functions. In each iteration, the sketch reads bytes from the serial port, putting together a message which is parsed and handled when complete; and runs the machine's own loop, which reads sensors (if any) and runs the motors depending on its current state.
- **`CommandParser`** contains functions for parsing commands and values from a message, according to the communication protocol.
- **`MachineControl`** contains the pin definitions for hardware, and the variables and functions that control the machine's behaviour, that is, reading from sensors (if any) and controlling motor speeds. This includes the machine's main loop, in the function **`machineControlLoop()`**.
- **`Utils`** contains utility functions to print messages to the serial port formatted according to the protocol.

Start/stop Switch (optional)

The firmware also supports a start/stop switch, so that the machine can be controlled via a physical control, like a foot-switch. This is available only on the new hardware, but disabled by default to prevent erratic behavior from reading a floating pin (i.e. a pin that has nothing connected to it).

To use the switch, make sure the line containing **`#define USE_START_STOP`** (found near the top of the main tab **`HILO_Firmware`** in the Arduino sketch) is uncommented, before uploading the sketch.

The switch is assigned to pin 19 on the Arduino Mega, corresponding to the "Z positive" endstop on the RAMPS v1.4 shield. You can reassign it to another digital pin if you need (see the **`MachineControl`** tab). Keep in mind that the start/stop switch is considered pressed when the pin's digital value is **`HIGH`**.

Hardware

The HILO machine is currently controlled by an Arduino Mega 2560 board with a RAMPS v1.4 shield. The board controls five stepper motors (FIT0278) through five stepper driver modules (Pololu A4988), one for each motor. While it was technically feasible to control the two motors running the elevator using only a single driver, this caused the driver chip to heat to potentially dangerous temperatures. So it was preferred to spend an extra driver instead of risking hazards to the machine or its users. Nevertheless, it is a good idea to add heatsinks to the driver chips, and to keep cool air circulating along the whole circuit (e.g. by connecting a fan to the corresponding connector on the RAMPS shield).

Stepper Motor Connections

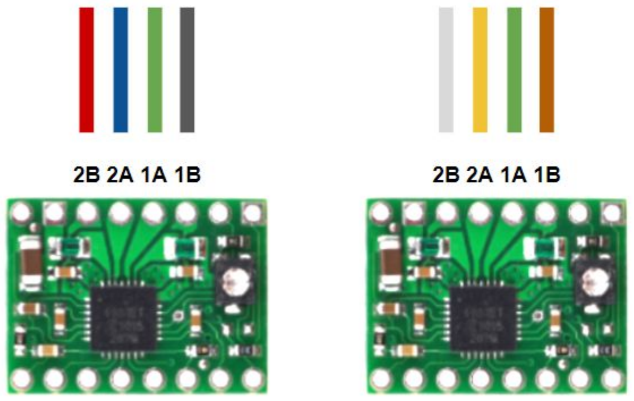
The RAMPS shield has sockets for five motor drivers. These are labelled X, Y, Z, E0 and E1. Since the RAMPS is originally meant for 3D printers, these would control the motors for the X, Y and Z axis (the latter with 2 motors in parallel) and two more motors for the extruder (E0 and E1).

In the HILO machine, these correspond to:

- X - drafting
- Y - delivery
- Z - spindle
- E0, E1 - elevator

The elevator motor connections are interchangeable, that is, it doesn't matter which motor is connected to which driver.

The stepper motor driver has four pins, broken out to the RAMPS shield. These are labelled, from left to right, **`2B 2A 1A 1B`**. Currently they are connected corresponding to the motor pins C D B A (as per the mechanical drawing of the motor). In what regards wiring color, with the existing cables and connectors, this makes **`2B 2A 1A 1B`** correspond to *red, blue, green, black*. Some of the cables have extensions, with different colors for the wires. These should be connected so that green connects with green; and so **`2B 2A 1A 1B`** correspond to *white, yellow, green, brown*.



Note: Connector Wire Colors

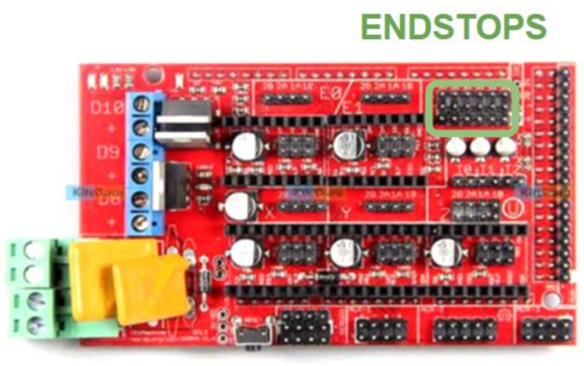
On the mechanical drawing document the wire colors indicated (in Chinese characters) for A C B D are respectively: black, green, red and blue. The same is true in online images of the product such as this one.

Curiously enough, the cables and connectors for motors we currently have are wired, for A C B D in black, red, green and blue. That is, red and green are swapped.

Please refer to the datasheet for the DFRobot FIT0278 stepper motors, and the A4988 Stepper Motor Driver Carrier.

Endstops and Switches

The RAMPS shield has dedicated connectors for endstop switches, which in 3D-printer hardware are used to sense when motors have reached the extremes of their axis. As such, there are six connectors, two for each end of the three axes.



The HILO machine's elevator has one endstop, at its top. When first connected or rebooted the machine will move the elevator up by default, when spinning.

When moving up, the elevator will reverse its direction once the endstop is pressed. When moving down, the elevator will reverse its direction after a set (empirically measured) amount of steps has been take (see the firmware).

The elevator endstop is connected to the **`X-`** (negative x-axis) connector. The firmware also supports an optional start/stop switch (see Firmware) which would then be connected to the negative z-axis connector (to the right of **`Z-`**).

Each of these are aligned vertically, and correspond from top to bottom to signal (marked **`S`**), ground/negative (**`-`**) and voltage/positive (**`+`**). They should be wired to the switch as pictured below.