LEonard

User Manual

LECKY
ENGINEERING

Software Version 22.11.1.1

LEonard Software by Lecky Engineering

Document Version	Date	Major Additions
21.11.4.0	11/04/2021	Initial user interface and device management system, Java interpreter
22.04.1.0	04/01/2022	Universal Robot interface and grinding system, LEScript support
22.08.1.0	08/15/2022	LMI Gocator interface and demonstration
22.11.1.0	11/14/2022	Python support, screen sizing and display management
22.11.1.1	11/25/2022	Documentation fixes, small code cleanup and reorganization

CONTENTS

Overview

Welcome! LEonard is a work cell control program that maintains communication with all the devices in your industrial work cell and allows you to orchestrate their coordinated operation using simple scripting- just like a good orchestra conductor.

LEonard allows you to write work cell control Sequences in LEScript, Java, or Python. You can also use Java and Python to create complex programs and subroutines that the LEonard Sequence calls, so LEonard doesn't trap you in someone else's framework. The use of Java and Python opens the potential to use millions of lines of pre-existing code, as well as providing you with all the features of these rich programming environments.

LEonard is designed to talk to a variety of devices and allow you to coordinate all of them through a single script. LEonard's tested and validated devices include:

- Robots
  - Universal Robots
- Vision Systems
  - LMI Gocator
  - MVTec HALCON
  - Teledyne Sherlock
  - Cognex Insight
- ID Readers
  - Keyence
  - Cognex Dataman
  - Zebra FS40
- PLC Interfaces (In test)
  - SIPC
  - MODBUS
- External Devices
  - TCP Connections
  - Serial Data

Really, anything you can talk to with Java or Python is a candidate for adding in yourself! Lecky Engineering adds custom device interfaces at customer request. These are usually very simple code extensions to develop, often identical to or very similar to something we already have.

Ask us!

The LEonard Screen

Why does it look like this?

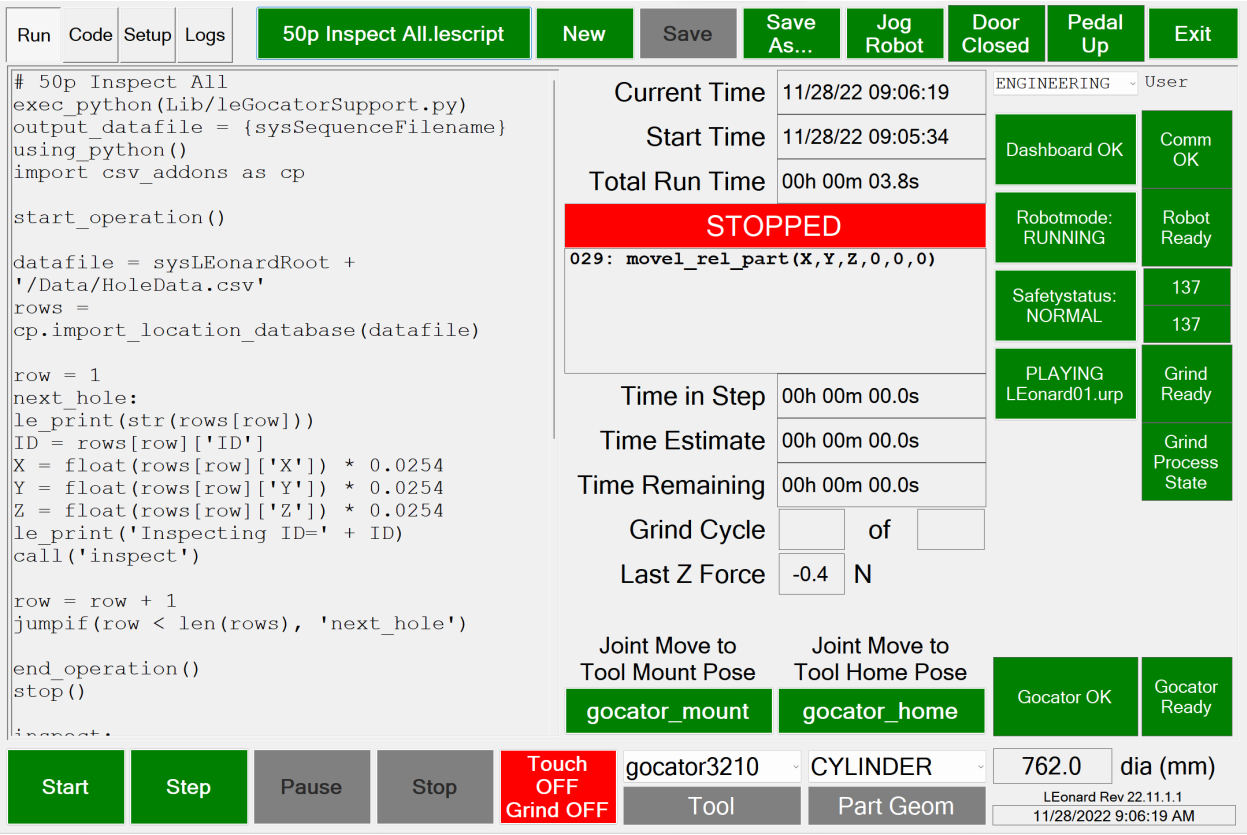


Figure 1 Typical LEonard Main Screen

LEonard is industrial control software and is designed to work well on standard 10” industrial touch screen tablets. Buttons are large and easy to read and special file open and save dialogs, as well as numeric entry, are handled in a touchscreen-friendly way.

LEonard can also be used comfortably on larger monitors. The screen, all the dialogs, and even the font sizes automatically scale as you use different screen sizes or resize the application window.

LEonard provides a standard database of various fixed screen sizes and behaviors, and you can add your own.

More information on LEonard displays is provided in the Setup | Displays section of the manual

An Introduction to LEonard Devices

LEonard maintains a list of devices that it communicates with. Devices are managed under the **Setup** | **Devices** tab.

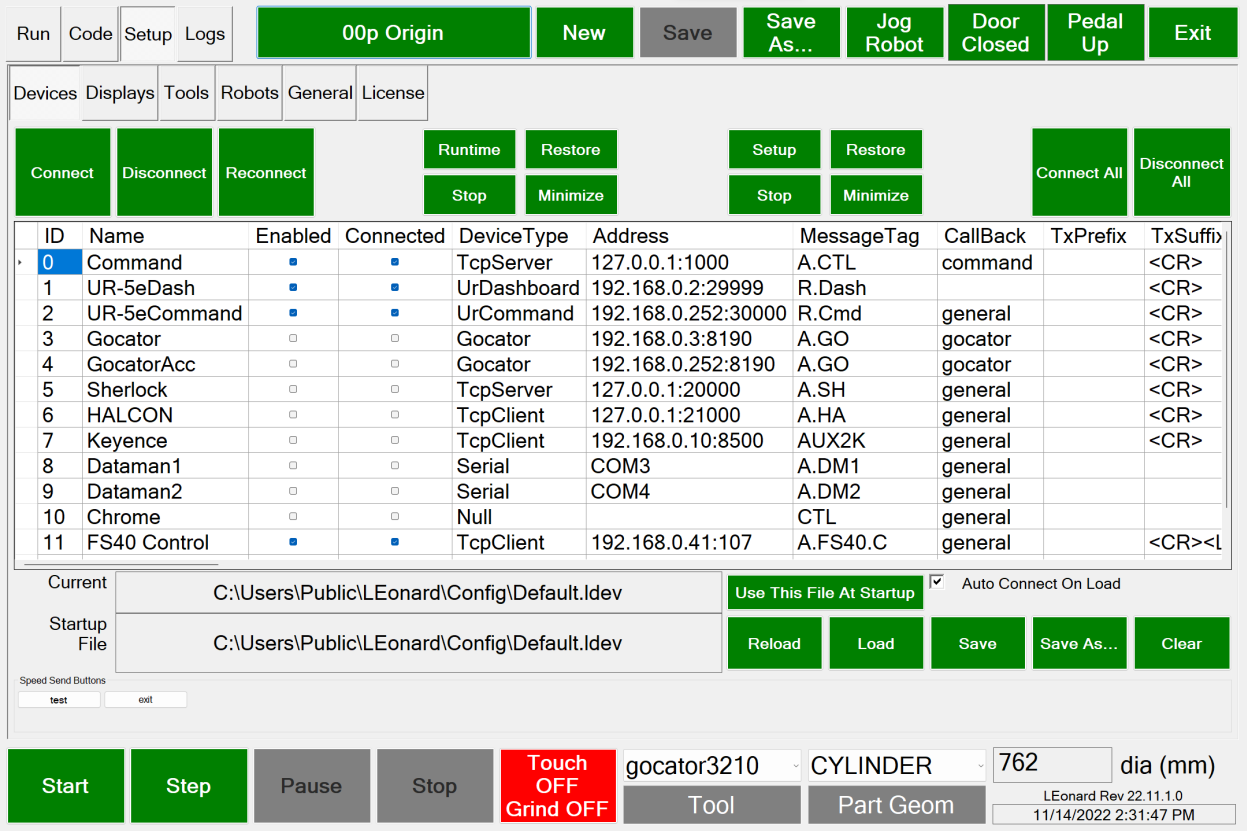


Figure 2 LEonard Setup | Devices Screen

Each device has many setup parameters that control how LEonard accesses and communicates with the device.

The device parameters are explained in more detail in Setup | Devices. It is sufficient to know the following:

1) The list of connected devices is stored in a device file. You can have many of these, but most users will need only one.
2) Each device has a unique ID number and a unique name.
3) Devices can be enabled or disabled… this affects which devices get connected by the **Connect All** button.
4) There are several types of devices:
    a. TcpServer
    b. TcpClient
    c. Serial
    d. Null

    e. Specialty
        i. Universal Robot Dashboard: UrDashboard
        ii. Universal Robot Command: UrCommand
        iii. LMI Gocator Interface: UrGocator
5) Each device sets up a **CallBack** function, which is code to handle messages from the device whenever they arrive.

The `general` CallBack is most common and is one of LEonard's key features and tricks! More information on the specifics of the Devices features of LEonard are available in Setup | Devices.

The LEonardStatement and LEonardMessage

Making LEonard send some particular message to some particular device doesn't take much fancy code. You just shoot the characters out some communication port and send some desired terminator.

The fancy part comes in dealing with what comes back. LEonard allows a CallBack function to be attached to any device, and most devices will be setup with the standard `general` CallBack.

The `general` CallBack knows how to interpret what it receives, whenever it receives it. This means that external devices can send messages to LEonard whenever they want.

The received messages need to be what is called a **LEonardMessage**… a series of **LEonardStatements** separated by some separation character.

When a **LEonardStatement** is analyzed, it is expected to be any one of the following. These are checked in sequence and the first match is handled and the other options untested.

1) `filename.js` Execute an entire Java file (wow!)
2) `filename.py` Execute an entire Python file (wow!)
3) `LE:script` Execute any LEScript statement
4) `JE:script` Execute any Java statement
5) `PE:script` Execute any Python statement
6) `SET var_name value` Set `var_name` = `value` all languages
7) `GET var_name`. Return the current value of variable `var_name`
8) `var_name = value` Stores `value` into `var_name` for all languages

That last one is simple and crucial: remote devices that can send results with `var_name=` in front of each one are instantly integrated into LEonard. This covers most barcode readers, vision systems, and even robots.

As you can see, external devices can ask LEonard to do very complicated or special things. This is what makes it possible to use LEonard to automate so many work cell scenarios.

So, the **LEonardMessage** is simply one or more of these **LEonardStatements** glued together with some sort of character separator, which is `#` by default.

Single: `LEonardMessage<TERM>`
Double: `LEonardMessage<SEP>LEonardMessage<TERM>`
More: `LEonardMessage<SEP>LEonardMessage<SEP> … <TERM>`

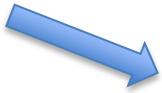
Why do this? The advantage of a multiple statement message is that all the received messages will be executed by LEonard in rapid sequence as a unit. This helps avoid certain kinds of machine race conditions.

What if I don't want this powerful receive capability on certain devices?
To use LEonardStatements, the `general` CallBack must be selected by the device. Not using a CallBack is the easiest way to prevent devices from asking LEonard to do fancy things when that's not needed or appropriate.

All About LEonard Sequence Programming

Theory of Operation

The Sequence



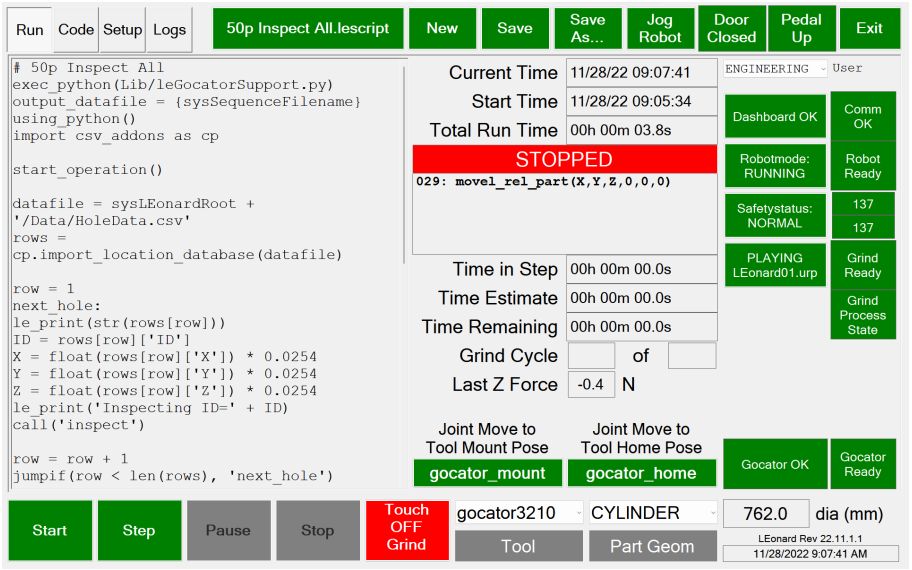


Figure 3 Where is the Main Sequence?

LEonard orchestrates the interoperation of many devices using a **Sequence**, a simple program script that can be written in LEScript, Java, Python, or a combination of all three.

Sequences can be stored as `.py`, `.js`, or `.lescript` file extensions and the default language will be assumed by extension. Sequences can all be multilanguage files, however, as controlled by the `using_language()` functions.

All three languages are provided to make things easier for you. Use what you like.

LEonard executes the main **Sequence** *one line at a time*. This permits monitoring, error recovery, and single stepping.

This may seem odd to those with a formal Computer Science background, but for reasons of safety and error-recovery, each line of a work cell Sequence needs to be handled all on its own.

This also allows single stepping through work cell operations, critical during debug and testing.

Line execution rate is slow, perhaps 10 lines per second. But work cell lines are asking robots and vision systems to do things that typically take many seconds so this doesn't matter.

Wait! What if I want to read the same barcode reader 10 times and return the highest confidence reading?

Just do it in Java or Python. Those languages run at full speed, you can build a function, and then you can just call it from your Sequence.

Large functions that are safe and which don't make robots fly around can be built into Java or Python procedures that you load at the start of the LEonard Sequence. This way, you can go fast when you need to if the error conditions don't create unsafe machine situations. Use your judgement. LEonard won't get in your way or second-guess you.

Now, since LEonard executes Java and Python line-at-a-time in the main Sequence window, creating functions and classes in Java or Python must be done in a separate stand-alone file and loaded in your main program. LEonard provides Java and Python sandboxes for writing and testing standalone code in the **Code | Java** and **Code | Python** tabs.

Hello, World!

How do we `printf("Hello, World!\n");` in a LEonard Sequence?

Well, first, about printing. LEonard provides a unified print console for debugging. Show or hide the console with the F12 key.

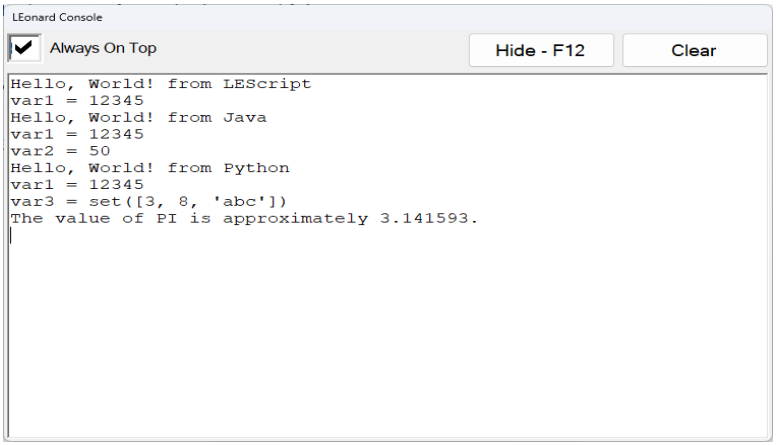


Figure 4 LEonard Console Window

Pressing F12 repeatedly shows and hides the console. It is always receiving and buffering print data. All print data goes to the log files, too, in case you ever need to look for that. (You can also use `le_log_error(message)` and `le_log_info(message)` to send things to the log files. More on that in the Logs tab!)

The Sequence command `le_show_console(True | False)` will also show or hide the console and works in any language.

To send data to the print console, use `le_print(message)` from any language.

OK, now we can try that Hello, World!

LEScript

LEScript is a simple scripting language created by Lecky Engineering that allows "programming-free" interaction with devices. Most typical work cells can be programmed in LEScript. If you need to compute values or get into deeper calculations or manage data structures, think about using Java or Python.

```
# Hello, World! In LEScript
using_lescript()  # Redundant since LEScript is default
le_print(Hello, World!)
value = 13.25
```

```
le_print(value = {value})
```

Console Output:

```
Hello, World!
value = 13.25
```

Java

LEonard provides a Java interface with full ECMA 5.1 compliance based on ***Jint***.

Java Specifications:

Jint Version:	2.11.58
Author:	Sebastian Ros
License:	https://raw.githubusercontent.com/sebastienros/jint/master/LICENSE.txt
Date Published:	11/27/2017
Project URL:	https://github.com/sebastienros/jint

There are several ways that Java execution can be triggered in LEonard.

1. In a LEonard Sequence, commands are interpreted as LEScript until `using_java()` is encountered. Subsequent lines will be executed using Jint.
2. The Sequence function `exec_java(filename)` will run a Java `.js` file as specified.
3. An external device that is using the `general` CallBack can return a Java request.
    a. `Filename.js` The specified file will be executed by Jint
    b. `JE:javascript` The specified Java commands will be executed by LEonard
4. The test area in the **Code | Java** tab. This provides a “sandbox” where Java programs can be created, edited, saved, and retrieved.

```
# Hello, World! In Java
using_java()
le_print('Hello, World!')
value = 13.25 * 8.1
le_print('value = ' + value)
```

Console Output:

```
Hello, World!
value = 107.32499999999999
```

Python

LEonard provides the open-source Iron Python implementation originally developed by Microsoft. This supports Python 2.7 environment and includes the entire standard library. Iron Python support for Python 3 is still in Beta and will be made available upon request if we feel it gets more stable! The Python standard library for 2.7 is quite complete. Import away!

Python Specifications:

Iron Python Version:	2.7.12
Author:	Iron Python contributors, Microsoft
License:	https://licenses.nuget.org/Apache-2.0
Date Published:	1/21/2022
Project URL:	https://ironpython.net/

There are several ways that Python execution can be triggered in LEonard.

5. In the LEonard Sequence, commands are interpreted as LEScript until `using_python()` is encountered. Subsequent lines will be executed in Python.
6. The Sequence function `exec_python(filename)` will run a Python `.py` file as specified.
7. An external device that is using the `general` CallBack can return a Python request:
    a. Send `Filename.py` The specified file will be loaded and executed in Python.
    b. Send `PE:pythonscript` The specified Python commands will be executed in Python.
8. The test area in the **Code** | **Python** tab. This provides a "sandbox" where Python programs can be created, edited, saved, and retrieved.

```
# Hello, World! In Python
using_python()
le_print('Hello, World!')
value = 13.25 * 8.1
le_print(str(value))

import math
var3 = {8, 'abc', 3}
le_print('var3 = ' + str(var3))
le_print('The value of PI is approximately
{0:.6f}.'.format(math.pi))
```

Console Output:

```
Hello, World!
107.325
var3 = set([3, 8, 'abc'])
The value of PI is approximately 3.141593.
```

The Main Leonard Tabs

Now you've got a feel for the overall approach to LEonard.

Let's go through each tab in the interface and describe what features are available.

In the main LEonard screen, there are four main operation tabs: **Run**, **Code**, **Setup**, and **Logs**. These tabs are described below.

Many functions can be manually activated with buttons or automatically activated with program commands. The convention in this manual is the used boldface for buttons and Tabs and `Courier Font` for program commands, as in **This Is a Button** and `this_is_a_code_function(param1)`.

The four main tabs will be described below. Here are some quick links if you want to jump!

Run Tab

The **Run** tab is where the bulk of program execution will typically be observed. Different annunciators that show status and control buttons appear on the Run Tab depending on what kinds of devices you are connected to.

For example, the raw LEonard screen looks like this, only showing the main program and providing access to Start, Stop, etc.

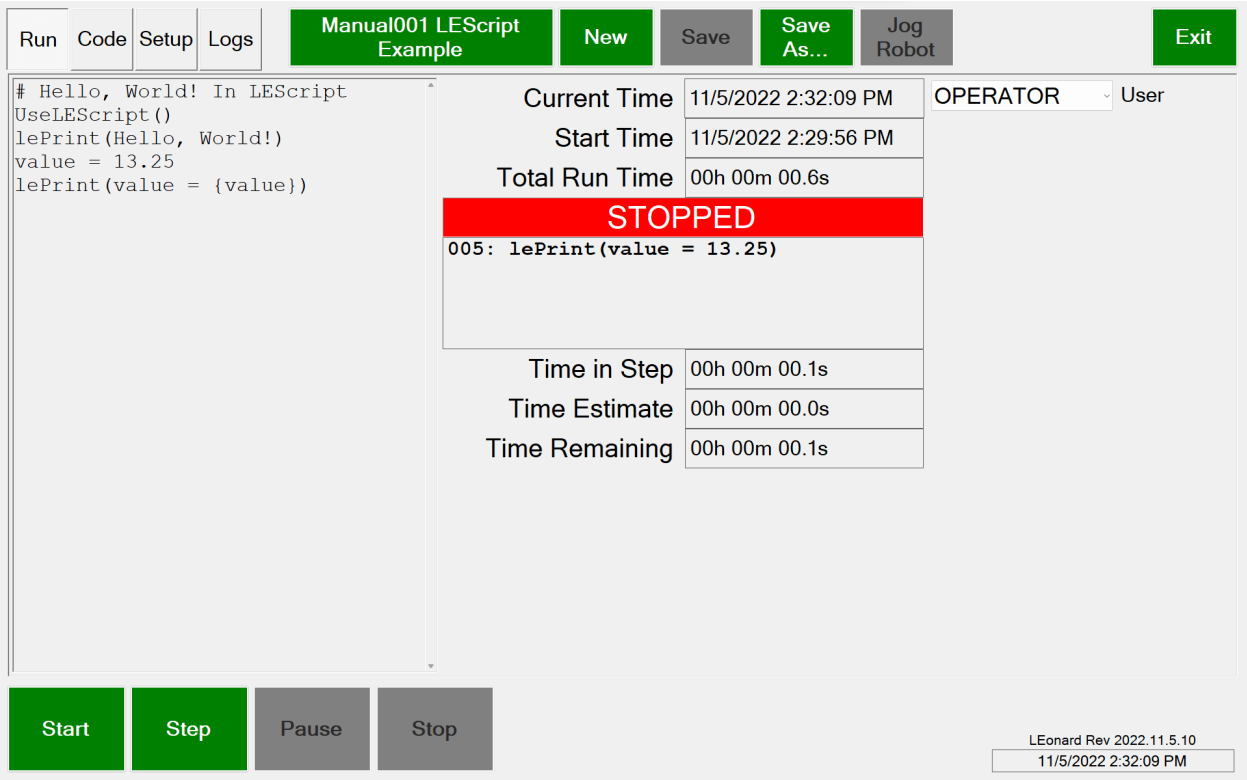


Figure 5 LEonard Run Screen with No Options Installed

A program is loaded using the **Program Name** button up next to the **Logs** tab.

Program file operations in addition to **Load** are **New, Save, Save As.**

To prevent unintentional setup changes or Sequence edits, LEonard provides three **User Modes**.

The user mode is selected using the **User** field in the upper-right corner of the **Run** tab.

1. **Operator** mode only permits loading and running existing Sequences from the **Run** tab. There is no access to the **Code** or **Setup** tabs.
2. **Editor** mode allows access to the **Code** tab to permit editing, but **Setup** is suppressed.
3. **Engineering** mode provides full access to all functions including **Setup**.

Entering Operator or Engineering mode requires a fixed password. By default, these are 9 and 99, respectively.

Code Tab

The **Code Tab** has five sub tabs: **Positions**, for teaching and manually moving to fixed positions, **Variables**, for monitoring or changing LEonard variables, **Java**, for writing and testing Java programs, **Python**, for writing and testing Python programs, and **Manual,** to provide access to documentation. The current software version is always displayed in the lower right of the screen.

Code | Positions

Below is the **Program Tab** when the Positions Subtab is selected.

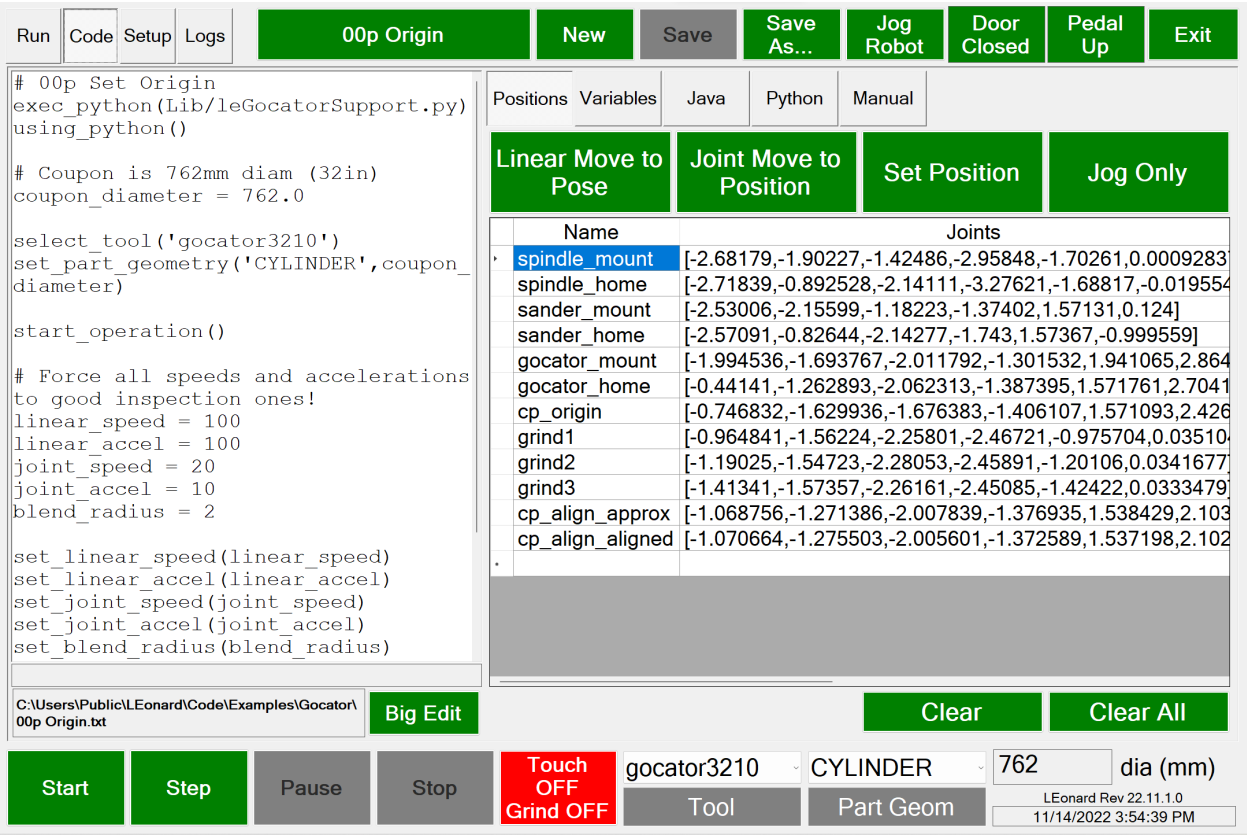


Figure 6 Code | Positions Tab

Positions can be saved manually (**Set Position**) or from the Sequence with *save_position(name)*.

You can manually move to Positions in Joint (**Joint Move To Position**) or Linear (**Linear Move To Pose**) paths. These can also be executed from a recipe with *move_linear(position)* or *move_joint(position).*

Jogging is used here for setting or updating named positions or just for moving the robot. This uses the standard Jog screen.

Big Edit opens up a full-screen editor to make editing complex recipes easier. If you have VS Code installed on your machine, it will use that instead. There is a VS Code workspace defined at `SysLEonardRoot/Code/LEonard-code.code-workspace` that you can edit to add other folders in if you are engaged in more complex projects.

Don’t want to default to VS Code? There’s an option to turn that off in **Setup | General**.

Code | Variables

Below is the **Program Tab** when the **Variables Subtab** is selected.

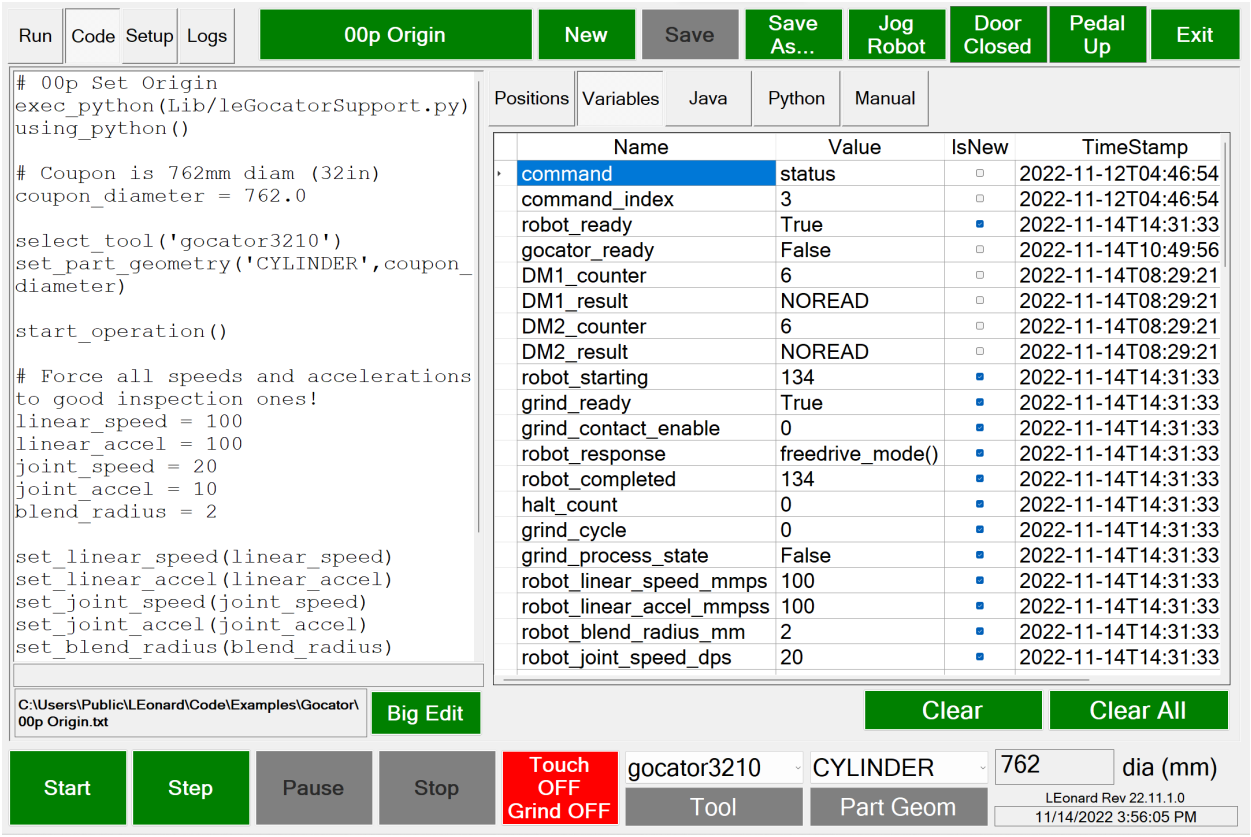


Figure 7 Code | Variables Tab

This tab shows all the local variables maintained in LEonard for internal, system, and user purposes. They can be edited here as well.

System variables will not be erased by the **Clear** button, or by the Sequence `le_clear_variables()` command.

The **TimeStamp** shows when the variable was last written.

IsNew indicates whether the variable has ever been examined by the program since the last write.

The variables are saved in `Variables.xml` in the `sysLEonardRoot/Config` directory with the **Save** and **Reload** buttons. Variables are automatically saved on program exit and reloaded when the program starts.

Code | Java

For testing out simple Java programs, the Code | Java tab allows loading, saving, and running Java code using the same Jint Java interpreter and environment used in Sequence execution.

Any code ideas you have can be tried out here prior to building them into an actual Sequence.

Figure 8 Code | Java Test Area

The buttons **New**, **Load**, **Save**, and **Save As…** operate as expected in creating, saving, and loading Java `.js` files.

The **Run** button will execute the Java program displayed immediately, even if LEonard is in the middle of running a Sequence. This is a powerful (maybe too powerful) debug and testing tool.

The bottom-left panel is a copy of any messages sent to `le_print` by Java. The right bottom-right panel shows a list of all Java variables created in Java or sent from LEonard.

Code | Python

For testing out simple Python programs, the Code | Python tab allows loading, saving, and running Python code using the same Iron Python interpreter and environment used in Sequence execution.

Any code ideas you have can be tried out here prior to building them into an actual Sequence.

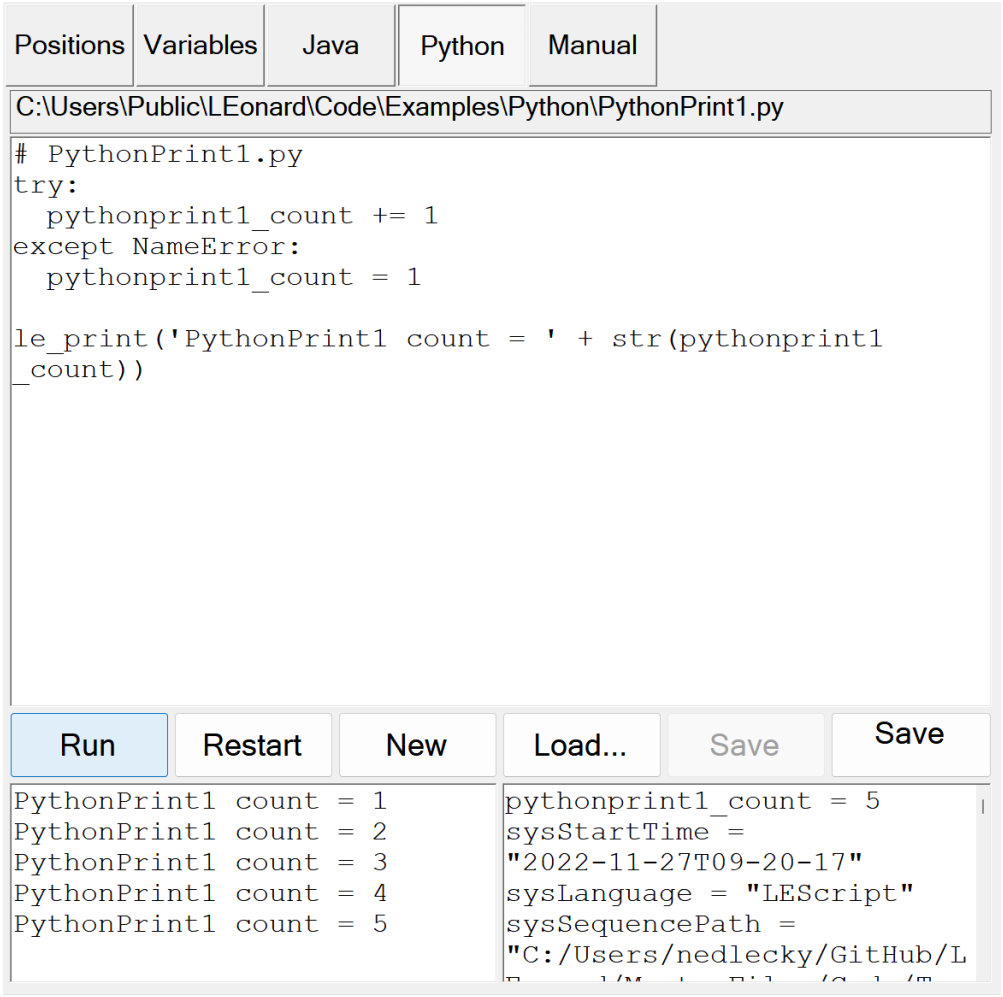


Figure 9 Code | Python Test Area

The buttons **New**, **Load**, **Save**, and **Save As…** operate as expected in creating, saving, and loading Java `.py` files.

The **Run** button will execute the Python program displayed immediately, even if LEonard is in the middle of running a Sequence. This is a powerful (maybe too powerful) debug and testing tool.

The bottom-left panel is a copy of any messages sent to `le_print` by Python. The right bottom-right panel shows a list of all Python variables created in Python or sent from LEonard.

Code | Manual

Below is the **Program Tab** when the **Manual Subtab** is selected.

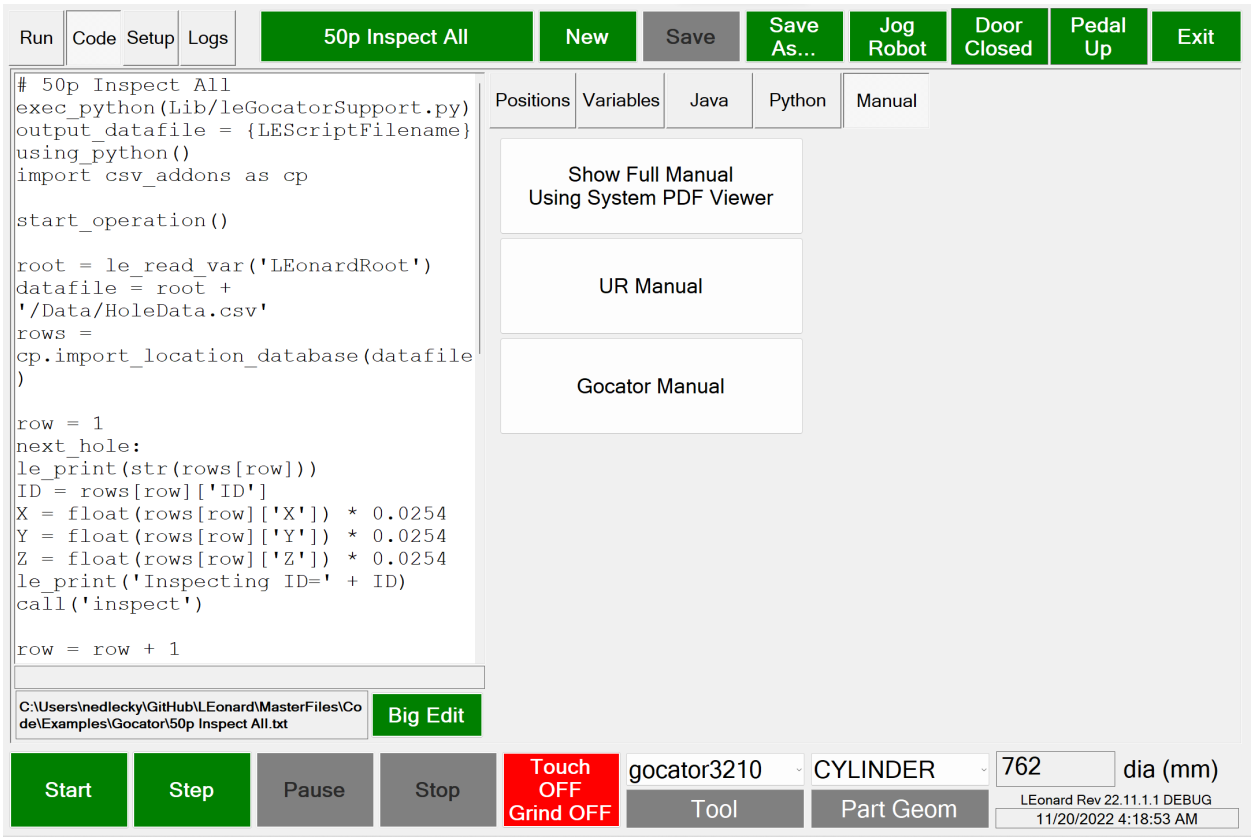


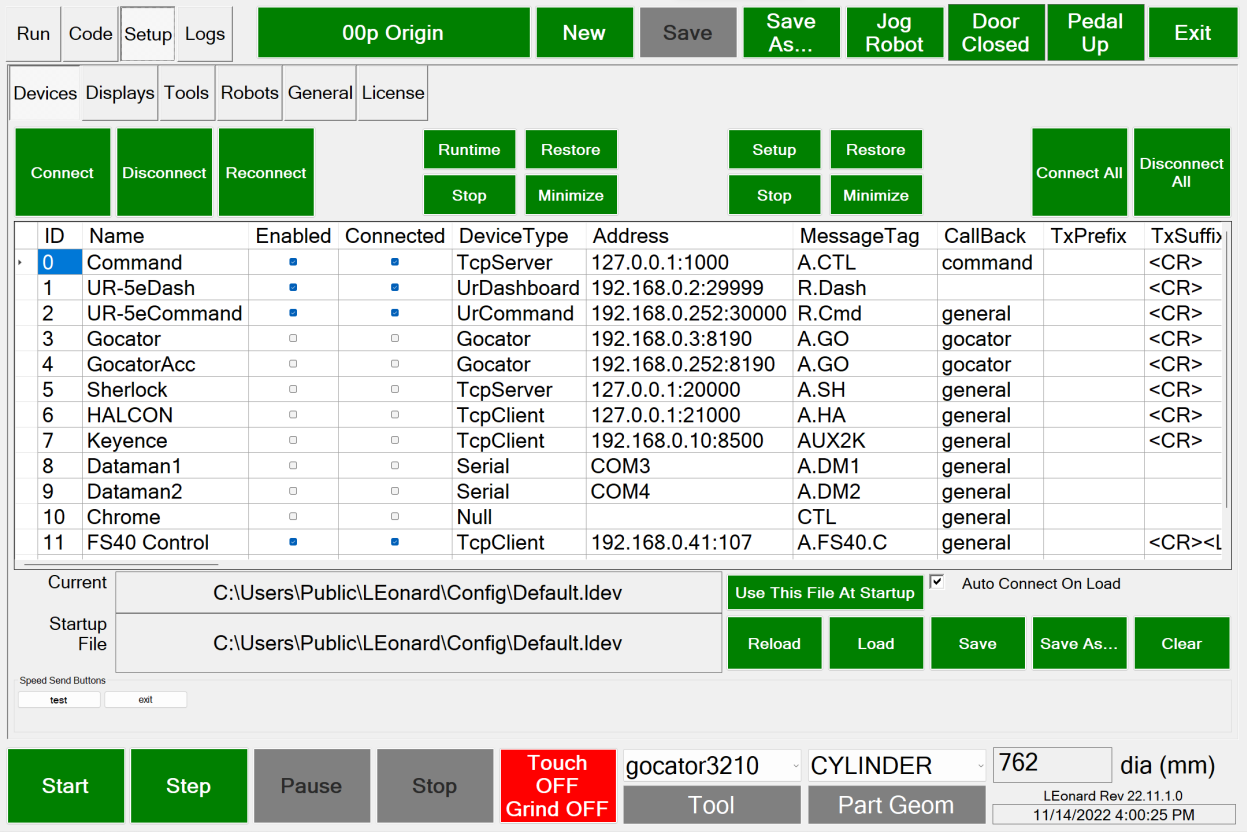
Figure 10 Code | Manual Tab

This tab provides access to all the LEonard manuals as PDF files. They are automatically opened on your computer by whatever default application Windows uses for PDF files.

There are also desktop shortcuts and Start Menu icons created for all the manuals in the installation process.

Setup Tab

The **Setup Tab** has six sub tabs: **Devices**, **Displays**, **Tools**, **Robots**, **General**, and **License**.



Setup | Devices

The LEonard Device list is a datafile created in the `sysLEonardRoot/Config` directory. You can have several device files and load different ones for different testing or operational situations. One or more Devices files can be created and managed in the **Setup | Devices** tab, shown below.

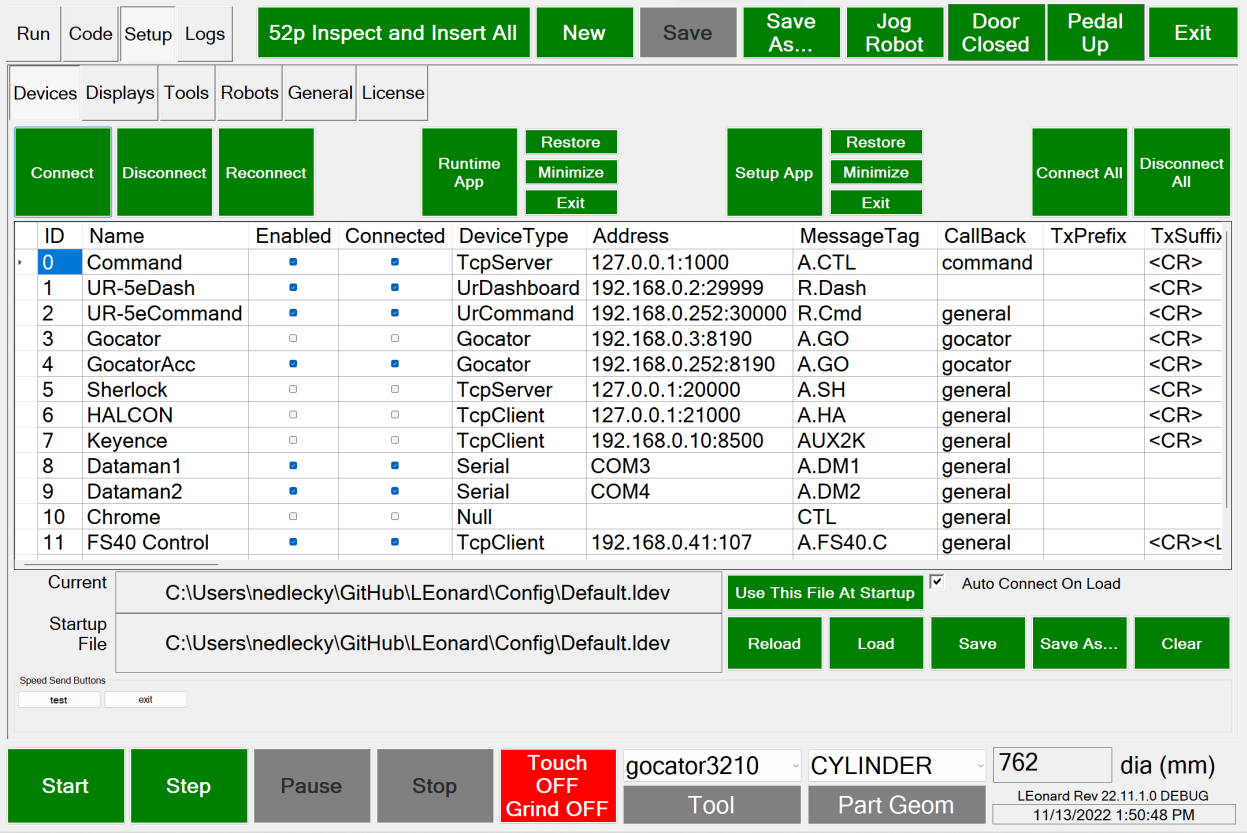


Figure 11 LEonard Setup | Devices tab

The **Reload**, **Load**, **Save**, **Save As..** and **Clear** buttons behave as expected. **Clear** also offers an opportunity to create a set of default devices spanning the set of common LEonard devices.

Press **Use This File At Startup** to copy the name of the current file into the **Startup File** field. This will cause that file to be loaded when LEonard starts. If you also select **Auto Connect On Load**, LEonard will attempt to connect to all of the enabled devices in the file at startup.

LEonard devices have the following parameters:

Field Name	Type	Description
ID	Integer	A unique ID assigned to the device
Name	String	A name to help you remember what the device does
Enabled	Boolean	Specifies whether the device should be automatically connected by the Connect All button.

		Devices are also automatically connected when the device file is loaded if **Auto Connect On Load** is enabled in **Setup | General**
Connected	Boolean	A Boolean value
DeviceType	String	One of several classes of devices, discussed below
Address	String	Either IP:Port or COMn
MessageTag	String	A string to be prepended to log messages to help identify messages from a particular device
CallBack	String	A CallBack function, described below.
TxPrefix <PREFIX>	String	A prefix of characters to be sent before each transmission. May include <CR> <LF>, and <CRLF>
TxSuffix <SUFFIX>	String	Characters sent at the end of each transmission. May include <CR> <LF>, and <CRLF>
RxTerminator <TERM>	String	Characters to be waited for to signify end of received message. May include <CR> <LF>, and <CRLF>
RxSeparator <SEP>	String	LEonard will parse (and execute atomically) multiple commands using command<SEP>command<TERM>
OnConnectExec	String	When the external device connects, any LEonardMessage specified here will be executed.
OnDisconnectExec	String	When LEonard initiates a disconnect, any LEonardMessage specified here will be executed.
RuntimeAutostart	Boolean	If true, Runtime Program will be started before connection is attempted. This can be used to start a background server needed by the device for operation.
RuntimeWorkingDirectory	String	The Runtime Program will be executed from this directory
RuntimeFilename	String	Specifies the filename of the Runtime Program
RuntimeArguments	String	Specifies arguments for the Runtime Program
SetupWorkingDirectory	String	Some devices have a Setup Program used to configure them. These can be specified here for directory, filename, and arguments
SetupFilename	String	Filename of the Setup Program
SetupArguments	String	Arguments for the Setup Program
SpeedSendButtons	String	If you'd like to be able to send simple commands for testing to the device, they may be entered here as command1|command2|command3|… A

		button will be created for each string between vertical bars!
JobFile	String	If a device needs to load a specific program to run it can be specified here. This is currently used by UrDashboard and Gocator to start the specified programs at connect time.
Model	String	If a device returns a model number, it will be entered here
Serial	String	If a device returns a serial number, it will be entered here
Version	String	If a device returns a software version number, it will be entered here

Device Types

LEonard device types must be one of the following items:

1. TcpServer Set up a TCP Server on Address:Port and wait for a connection.
2. TcpClient Immediately connect to a device on Address:Port.
3. Serial Connect to a device with Address = COMn using serial protocol over either a hard serial or a USB serial connection
4. UrDashboard Setup a TCP Client connection to a Universal Robot dashboard server.
5. UrCommand Setup a TCP Server for a Universal Robot PolyScope program to attach to
6. Gocator Setup a TCP Client appropriate for handling commands with an LMI Gocator.
7. Null Connect to nothing… but perhaps use the other features of a device!

Connect/Disconnect Execution

Just after a device connects or just before it is disconnected, LEonard can perform an operation. These operations are encoded in the **OnConnectExec** and **OnDisconnectExec** fields in the device.

Any **LEonardMessage** can be specified. Recall a multi-statement LEonard message can be encoded as:
LEonardMessage = LEonardStatement<SEP>LEonardStatement

Setup | Displays

LEonard maintains a database of standard display sizes in the **Setup | Displays** tab, shown below.

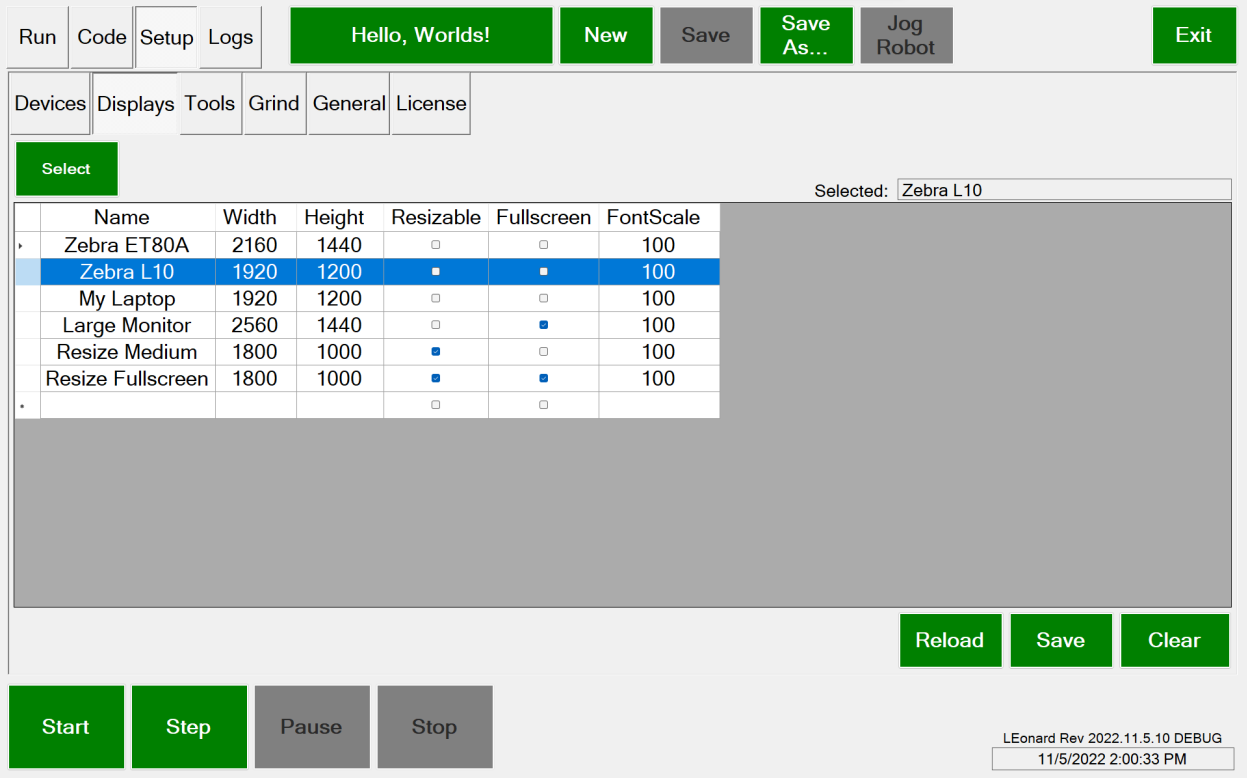


Figure 12 LEonard Setup | Displays Screen

All LEonard windows and dialogs are designed to be resizable. Fonts get larger or smaller as necessary. However, you arrange the dialog to use it, that is how it will appear the next time you see it.

You can create your own display setting, complete with Width, Height, whether the window is fixed size or resizable, whether it should expand to full screen, and an optional additional Font Scale parameter. The **FontScale** is applied to the main windows and all LEonard dialogs.

The display database is automatically saved by LEonard, but you can reload the old one if you've made a mistake. Clearing the displays will also offer an option to restore these defaults.

Whatever display is selected when LEonard exits will be restored at application startup.

Setup | Tools

Robot tools for are defined in the **Setup | Tool** tab, shown below.

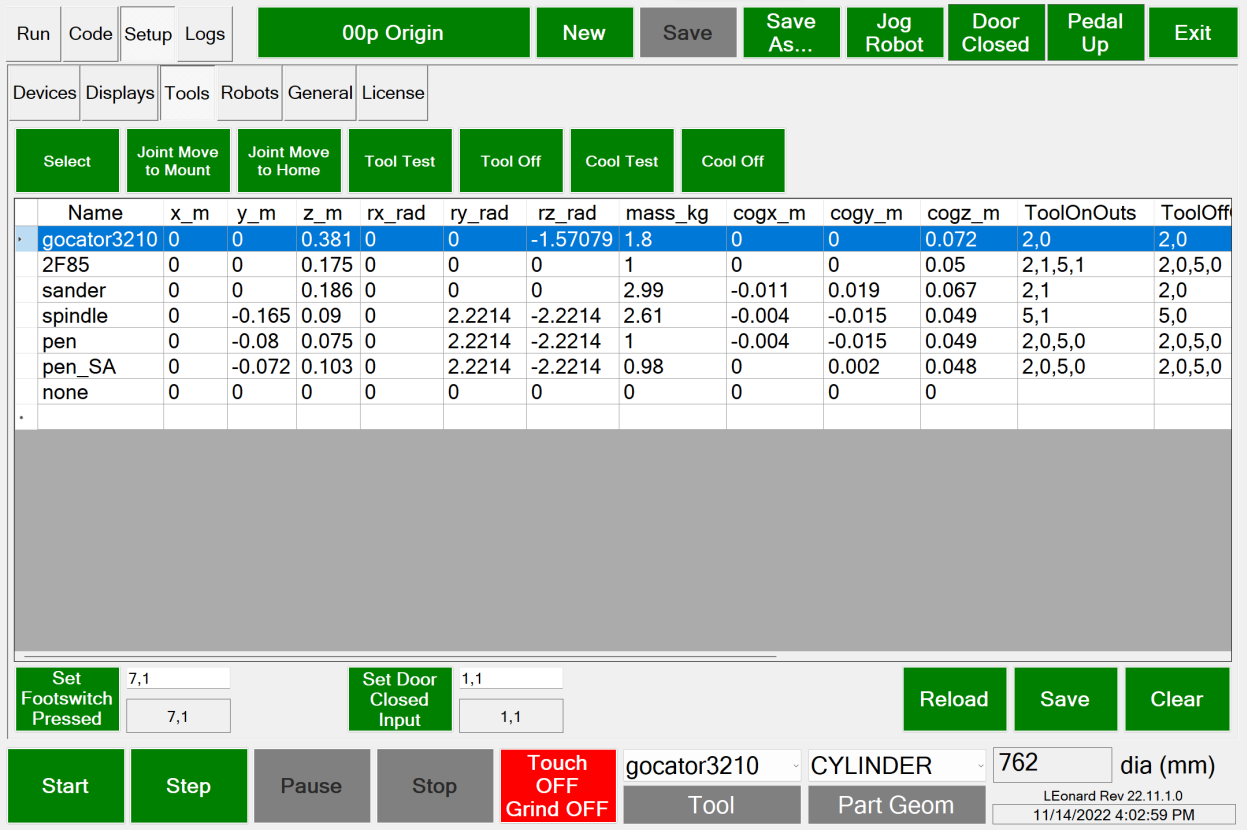


Figure 13 Setup | Tools Tab

Each tool entry contains the following information. These are saved in the `Tools.xml` file in `SysLEonardRoot/Config` and are loaded and saved automatically.

1. **Tool TCP:** This is a copy of what we would teach for the tool on the UR including x, y, z offset and rx, ry, rz orientation. Teaching these is best done on the UR and then the values simply copied to the entry in LEonard
2. **Mass and Center of Gravity:** Set these as you would on the UR. Accurate settings improves behavior when in freedrive mode.
3. **ToolOnOuts, ToolOffOuts:** This is a list of up to 4 digital IOs that need to be turned on or off to enable the tool. This is only done during a grind in **Touch ON Grind ON** mode. Examples: “1,1,3,1” implies that output 1 should be set to 1 and output 3 should be set to 1. “3,1” implies that output 3 should be set to 1
4. **CoolantOnOuts, CoolantOffOuts:** Similarly, these are digital output commands to be executed when grinding in **TouchOn Grind ON** mode.
5. **MountPosition:** This is a position recommended for installing/removing this tool. LEonard uses joint moves to approach the position with **Joint Move To Mount** or

move_tool_mount(). This must be a position that has been defined in the **Positions Table**.

6. **HomePosition:** This is a position recommended for home for this tool. LEonard uses joint moves to approach the position with **Joint Move To Home** or *move_tool_home()*. This must be a position that has been defined in the **Positions Table**.
7. **Tool Test, Tool Off** and **Cool Test, Cool Off**: These allow manually verifying the outputs for the currently selected tool.
8. **Set Footswitch Pressed Input:** This is defaulted to 7,1 meaning that input 7 goes high when the footswitch is pressed.
9. **Set Door Closed Input:** This is defaulted to 1,1 meaning that input 1 goes high when the door is closed.

Setup | Robot

Optional robot and grinding general settings are maintained in the Setup | Robots page, shown below.

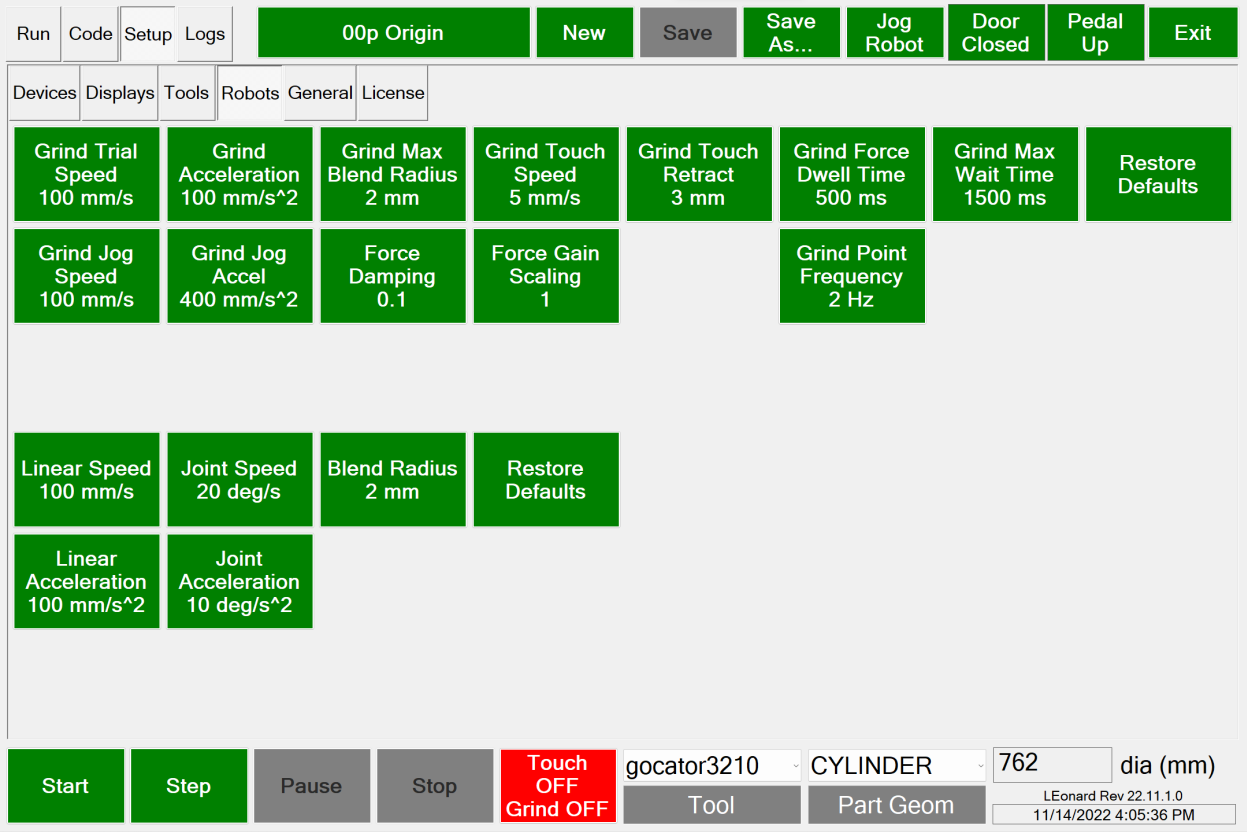


Figure 14 Setup | Robots Tab

First, there are settings governing grind operations. These are saved in the `Variables.xml` file in `SysLEonardRoot/Config` and are sent to the robot whenever the software starts. New values are saved automatically.

Grind Trial Speed: When not in **Touch On Grind On** mode, the grind patterns are limited to one cycle and are performed at this speed.

Grind Acceleration: Linear acceleration used during grinding
Grind Max Blend Radius: Maximum blend radius used during grinding. Recommended 2 mm
Grind Touch Speed: Speed robot advances toward part for touch off. Recommended 5-10 mm/s
Grind Touch Retract: Distance robot retracts from part after touch off.
Grind Force Dwell Time: How long robot waits after turning force-on to allow time for tool to settle against part
Grind Max Wait Time: Maximum time the robot will wait for the next grind command if a grind command ends with 1 (stay in contact with part)

Grind Jog Speed: Linear speed used for all grinding motions that are not in contact with the part other than the actual simulated grind which runs at **Grind Trial Speed.**
Grind Jog Accel: Linear acceleration used for all grinding motions that are not in contact with the part.
Grind Point Frequency: Used as a minimum frequency for points generated for circles and spirals.
Force Damping: May be useful for force mode tuning in the future. Calls the URScript function force_mode_set_damping() with a value between 0 and 1. The default is 0 and that is the only value that has been tested.
Force Gain Scaling: May be useful for force mode tuning in the future. Calls the URScript function force_mode_set_gain_scaling() with a value between 0 and 2. The default is 1.0 and that is the only value that has been tested.

Second, there are generally self-explanatory settings for default speeds and accelerations used in basic jogging and non-grinding motion. These are also saved in `Variables.xml`. New values are saved automatically. **Restore Defaults** sets all to standard values used in all testing.

Setup | General

Any system-wide setup is stored in **Setup | General**.

Currently, only the `sysLEonardRoot` directory is stored here and that is currently defaulted to `C:\Users\Public\LEonard`. This is a good place for LEonard files since any user will be able to access them there.

Setup | License

LEonard shows its licensing information in **Setup | License**. LEonard uses an encrypted license file that includes information about the CPU, Windows Version it is licensed on, and options that are included in the installation.

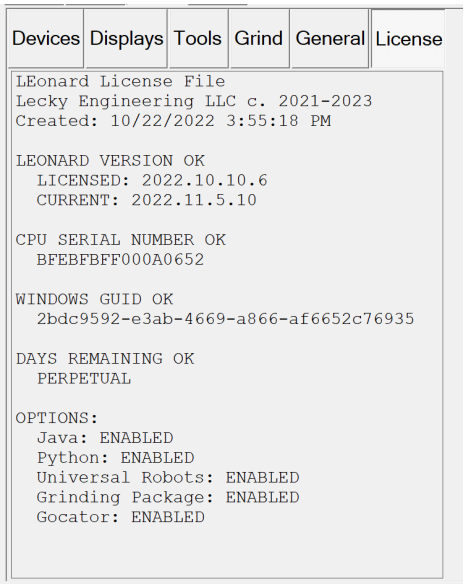


Figure 15 LEonard Setup | License Viewer

Licenses can be perpetual or have a time limit for trial purposes.

A given license file will only work on one CPU and one installation of Windows. Contact Lecky Engineering if you have other needs. We're very flexible.

Current options are:

- Java
- Python
- Universal Robots Support Package
- Grinding Package for Universal Robots
- LMI Gocator Support

Contact Lecky Engineering to enable features and troubleshoot licensing issues with your software.

Logs Tab

The Log Tab provides five windows where log messages are displayed. The level of detail in the messages is controlled by the Log Level setting:

- Error: only error messages are shown
- Warn: Error messages and Warnings are shown
- Info: All of the above, plus informational messages about execution. Default setting
- Debug: All of the above plus additional information that may be useful for debugging
- Trace: All of the above plus extremely verbose execution tracing

The **All Log Messages** box gets 100% of the generated messages. These messages are also written to log files in the `SysLEonardRoot/Logs` directory, where up to forty 25MB files are archived. Information older than this 2GB total is automatically and silently deleted over time.

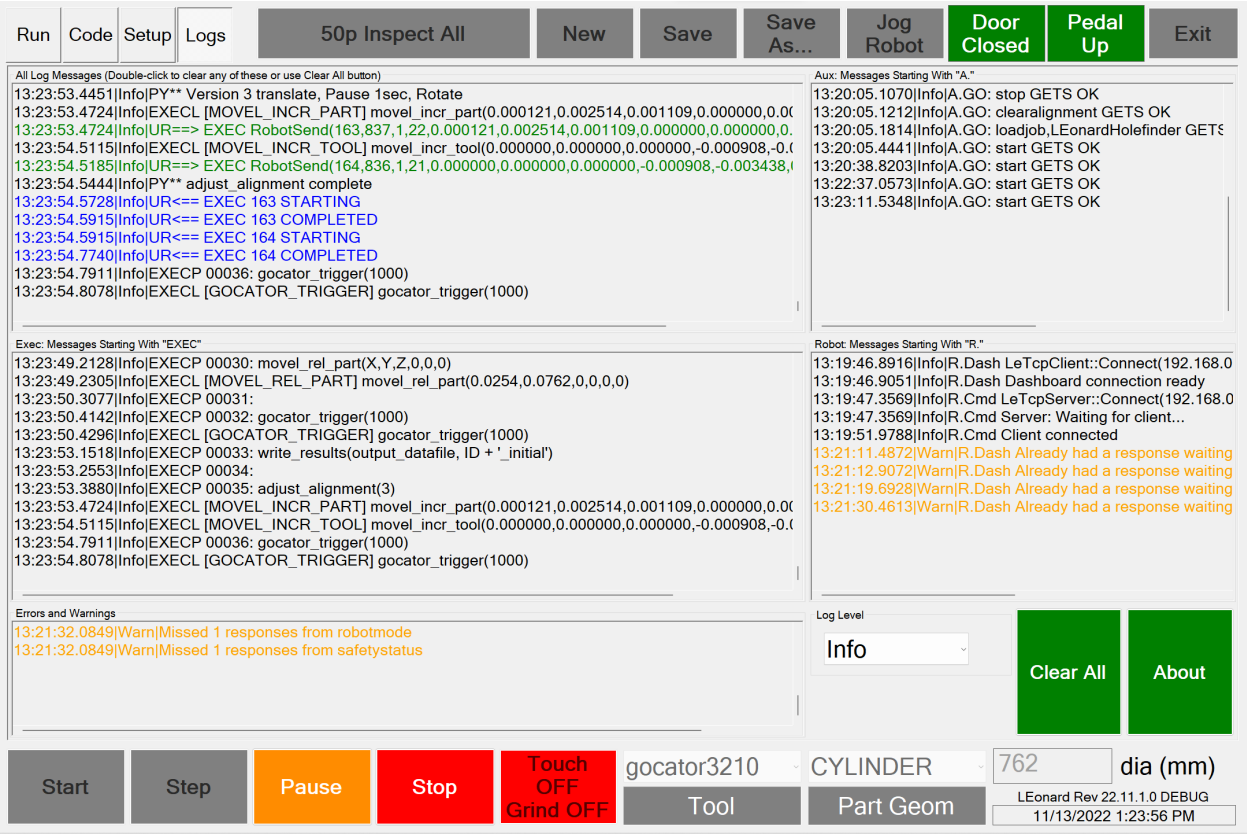


Figure 16 LEonard Logs Tab

In addition, some messages are copied for clarity to other boxes.

1. Any Warning or Error messages are duplicated in the Errors and Warnings box. Warnings are always displayed in Orange, and Errors in Red.
2. Any messages starting with the characters `EXEC` are duplicated in the “Exec: Messages Starting With EXEC” box. These messages are typically associated with Sequence line-

by-line execution. Execution messages generated by LEScript contains “EXECL”. Java and Python execution messages start with “EXECJ” and “EXECP”, respectively.

3. Messages beginning with `R.` are duplicated in the “Robot: Messages starting with R.” box. All devices have a **MessageTag** setup in their **Devices** entry and these characters are always prepended to their messages. Placing all the robot-related messages into a device containing a tag that starts with `R.` causes those messages to be duplicated here. This can help see robot-related issues with your Sequence.
4. Similarly, there is an “Aux: Messages Starting With A.” box. Any messages beginning with `A.` are duplicated here, so assign a **MessageTag** starting with `A.` to any devices whose messages you’d like to see here.

Any of the boxes can be cleared by double-clicking on them. All boxes can be cleared with **Clear All.** All the messages are appended to the log files and are archived as described above.

LEonard Function

You know all there is to know about the LEonard user interface. Congratulations!

What's left? All the possible Sequence functions and what they do.

This is the comprehensive list.

Certain options, such as Universal Robot or LMI Gocator support, enable other options which are documented in the **Using XXXX with LEonard** manual series.

LElib Standard Library, All Languages

The LElib Library provides common functions across all three languages.

Where differences exist, they are be highlighted with the language matrix shown below.

LEScript	Java	Python
?	?	?

LElib.language: Using Different Languages

By default, all Sequences are interpreted as LEScript. You can change this with the `using_xxx()` functions. The language that is currently being interpreted is always available in the system variable `sysLanguage`.

`using_lescript()`

Future sequence lines will be interpreted as LEScript. The variable `sysLanguage` is set to `LEScript` in LEScript, Java, and Python.

`using_java()`

Future sequence lines will be interpreted as Java. The variable `sysLanguage` is set to `Java` in LEScript, Java, and Python.

`using_python()`

Future sequence lines will be interpreted as LEScript. The variable `sysLanguage` is set to `Python` in LEScript, Java, and Python.

`exec_lescript(string filename)`

This command loads an entire LEScript Sequence and executes all lines sequentially. Recommended only for setup purposes to set variables. Should not try to execute long operations like robot moves, but it will if you ask it to!

`exec_java(string filename)`

This command loads an entire Java script file and executes it in a fashion identical to the way the **Code | Java** test area does.

`exec_python(string filename)`

This command loads an entire Python file and executes it in a fashion identical to the way the **Code | Python** test area does.

`execline_lescript(string line)`

This function can be used if you just want to execute one line of LEScript regardless of what language is selected.

`execline_java(string line)`

This function can be used if you just want to execute one line of Java regardless of what language is selected.

`execline_python(string line)`

This function can be used if you just want to execute one line of Python line regardless of what language is selected.

LElib.variables: System Variables

LEonard maintains all of its variables in the **Code | Variables** table. All LEonard variables are strings.

Whenever a LEonard variable is written, it is also copied to the Java and Python engines. This minimizes the need to explicitly copy variables from LEonard to Java or Python.

Several system variables are always available in all three languages:

`sysLEonardRoot`	The root directory where Code, Data, Logs, etc. are contained
`sysSequenceFilename`	The filename of the currently loaded sequence
`sysSequencePath`	The path where the Sequence file came from
`sysLanguage`	The name of the currently executing language
`sysStartTime`	A timestamp showing when Sequence started running.

LElib.variables: Interacting with Variables

This is the area where the three languages supported by LEonard differ the most!

For Java and Python, declaring and setting variables, using math, and creating class and structures works as expected in those languages.

Just remember that the Sequence executes line-by-line, so any multiline definitions need to be placed in a text file and either imported or loaded.

LEScript has a few specific variable handling functions which are described below.

`clear_variables()`

LEScript	Java	Python
X		

Deletes any variables not marked in the **Code | Variables** table as system variables.

Variables named robot_* and grind_* are automatically marked as system variables.
You can also use `le_write_sysvar(name, value)` from Java or Python, or `le_system_variable(name, True|False)`, to set whether a variable is a system variable.

`import_variables(string filename)`

LEScript	Java	Python
X		

Read a file and process any lines that contain `var_name = value`.

`system_variable(string var_name, bool is_system)`

LEScript	Java	Python
X		

Sets whether an existing variable is a system variable.

`le_random(int N, float low, float high)`

LEScript	Java	Python
X		

LEScript doesn't have direct access to the powerful random number facilities of Java or Python. We suggest you use them, but for a basic capability from native LEScript we have `le_random()`.

This function creates `N` variables between `low` and `high` and names them `rnd1`, `rnd2`, through `rndN`.

`LEScript Assignment`

LEScript supports updating variables using any of these basic operations. For `++`, `--`, `+=`, and `-=`, if the named variable does not exist, it is first created and initialized to 0.

Variables can be substituted into any LEScript command using the syntax `{var_name}`.

```
var_name = 12.3       var_name = {other_var_name}
var_name++            var_name--
var_name -= 17.5      var_name += 18
```

Copying Variables Between LEonard and Java/Python

When a variable is written in LEScript or assigned a value in a LEonardMessage, it is written to the global variable list of all three languages.

Explicitly rereading a variable or writing a variable back to LEScript either as a system variable or a standard variable is support for both Java and Python.

System variables differ from normal variables in that they are not erased by the `le_clear_variables()` function. All variables are written to the data file `sysLEonardRoot\Config\Variables.xml` when LEonard closes, and so are persistent.

`string le_read_var(string var_name)`

LEScript	Java	Python
	X	X

Copies a variable from LEonard to Java or Python. All LEonard variables are stored as strings! Note that when variables are written in LEonard they are automatically also copied to Java and Python.

`le_write_var(string var_name, string value)`

LEScript	Java	Python
	X	X

Copies a variable from Java or Python to LEonard. All LEonard variables are strings.

`le_write_sysvar(string var_name, string value)`

LEScript	Java	Python
	X	X

Copies a variable from Java or Python to LEonard and marks it as a system variable. All LEonard variables are strings.

LElib.console: Console Functions

`le_print(string message)`

Prints message to the Console Window. For Java and Python, it is also sent to the console test areas in **Code | Java** or **Code | Python** as appropriate. The messages are also logged to the logfile as:

LEScript: `LPR:message`

Java: `JPR:message`
Python: `PPR:message`

`le_show_console(bool show)`

Hides or shows the Console Window. The console is always open and accumulating any `le_print(…)` messages from any language.

`le_clear_console()`

Clears all the text from the Console Window, just like the **Clear** button in the console window itself.

LElib.log: Logging Functions

`le_log_info(string message)`

Send the message to the logging system as Info. The message will appear in the **Logs** tab and in the logfile.

`le_log_error(string message)`

Send the message to the logging system as Error. The message will appear in the **Logs** tab in red in the Error buffer and will also be sent to the logfile.

LElib.flow: Flow Control Functions

Flow control in line-by-line execution is implemented differently than it is in the Java or Python standards.

LEonard follows a convention in which lines can be given label names and then jumped to or called. This is consistent with line-by-line execution and is why LEonard is different in the main execution window.

Java and Python functions created in text files operate the way Java and Python always do!

The following sections list each flow control command.

`comments`

LEScript comments in the Sequence follow both the Java and Python conventions as follows:

1. All blank lines are skipped
2. Characters after “#” on any line are ignored
3. Characters after “//” on any line are ignored

Java and Python comments are defined as in the respective languages. Python statements ignore characters after “#” on any line and Java statements ignore any characters after “;” on any line.

`pause()`

Causes execution to pause, just like pressing **Pause** on the Run tab.

`pauseif(bool condition)`

Conditional `pause()`. If `condition==true`, causes execution to pause, just like pressing **Pause** on the Run tab.

`stop()`

Causes execution to stop, just like pressing **Stop** on the Run tab.

`stopif(bool condition)`

Conditional `stop()`. If `condition==true`, causes execution to stop, just like pressing **Stop** on the Run tab.

`prompt(string message)`

Puts up a dialog box containing `message` and pauses execution until the operator presses `Continue` or `Abort`.

`promptif(bool condition, string message)`

Conditional `prompt(message)`. If `condition==true`, puts up a dialog box containing `message` and pauses execution until the operator presses `Continue` or `Abort`.

`label_name:`

Yes, this works in all three languages! This helps support line-by-line sequencing and flow control, which Java and Python cannot do.

This structure associates a name with a line in the Sequence. Label names are alphanumeric, case-sensitive, and may include the ‘_’ character. Labels are found prior to execution and can be used as targets for `jump`, `jumpif, call,` and `callif` statements. LEScript provides the j`ump_gt_zero(variable, label_name)` function since it does not presently have the ability to evaluate comparison conditions.

`jump(string label_name)`

Causes execution to pass to the line containing `label_name:`

`jumpif(bool condition, string label_name)`

Performs a `jump` to the line containing `label_name`: if condition is true.

`call(string label_name)`

Causes execution to pass to the line containing `label_name:` Use `ret()` to return from the call. Call maintains a return stack (which is cleared when execution begins!) and can nest.

`callif(bool condition, string label_name)`

Performs a `call` to the line containing `label_name:` if condition is true.

`ret()`

Return execution from a `call(…)` or `callif(…)` to the line after the one that initiated the call.

`sleep(float timeout_s)`

Causes the Sequence to pause for `timeout_s` seconds. All other operations continue, so this is better to use than the built-in sleep functions in Java or Python!

`jump_gt_zero(string var_name, string label)`

Only available in LEScript since traditional Java and Python comparisons aren't available there. Equivalent to `jumpif(var_name > 0, 'label_name')` in either Java or Python.

`assertTrue(bool condition)`

Testing support function. Halts execution if condition is not true.

`assertFalse(bool condition)`

Testing support function. Halts execution if condition is true.

`assertEqual(string var_name, string value)`

Testing support function provided for use with LEScript. Halts execution if `var_name != value`.

`assertNotEqual(string var_name, string value)`

Testing support function provided for use with LEScript. Halts execution if `var_name == value`.

LElib.device: Device Control Functions

`le_connect(string device_name)`

Performs the **Connect** function on the specified device. Equivalent to selecting the corresponding row in the **Setup | Devices** table and pressing **Connect**.

`le_disconnect(string device_name)`

Performs the **Disconnect** function on the specified device. Equivalent to selecting the corresponding row in the **Setup | Devices** table and pressing **Disconnect**.

`le_connect_all()`

Performs the **Connect** function on all devices in the **Setup | Devices** table that are enabled. Equivalent to pressing **Setup | Devices | Connect All**.

`le_disconnect_all()`

Performs the **Disconnect** function on all connected devices in the **Setup | Devices** table that are enabled. Equivalent to pressing **Setup | Devices | Connect All**.

`le_send(string device_name, string message)`

Sends the specified `message` to `dev_name` including any terminators specified in the device entry. Device must be connected.

`string le_ask(string device_name, string message, int timeout_ms)`

Sends the specified `message` to `dev_name` including any terminators specified in the device entry. Device must be connected.

Waits up to `timeout_ms` for a response and returns it as follows:

1. LEScript. Any response is stored in the variable `le_ask_response`. If no response is received or the device is not connected, sets the return value to `Null`.
2. Java and Python. The received string is returned by the function.

LElib.infile: Using Input Files

Java and Python have extensive file I/O capabilities. This library is a simple set of functions that allow LEScript to read CSV files one line at a time. Deprecated.

LEScript	Java	Python
X		

`infile_open(string filename)`

Opens up file `SysLEonardRoot/Data/filename` for reading. The file is assumed to be in CSV format. Headers are skipped.

`infile_close()`

Closes any file that has been opened with `infile_open(…)`.

`infile_readline()`

A line is read from the file opened with `infile_open(…)`. Fields found on the line are stored in individual variables named `infile_p0`, `infile_p1`, etc. The values can be automatically scaled using `infile_scale(…)` below.

`infile_scale(int column, float scale, …)`

Causes any data subsequently read using `infile_readline()` from specified columns of the input file to be scaled by a value. For example, the line below would cause columns 2, 3, and 4 of the input data to be multiplied by 0.0254, for example to convert from inches to meters.

```
infile_scale(2,0.0254,3,0.0254,4,0.0254)
```

INDEX