LEonard

User Manual

PREMININARY

LECKY
ENGINEERING

Lecky Engineering, LLC
Software Rev 2022.11.9

November 9, 2022

Overview

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

LEonard allows you to use write programs in custom LEScript, Java, or Python to write control software and subroutines. 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.

LEScript is simpler and basic and is often more than enough for simple work cell coordination and data collection and is less intimidating to those who may not already have familiarity with Javas or Python.

An aside: LEonard File Structure

LEonard is aware of a “LeonardRoot” directory where it is installed. LEonard expects to be running out of LeonardRoot/LEonard/bin
For you this could be anywhere. Many customers use c:\LEonard.

Beneath this root, several directories are maintained:

- Code: Where the programs you create will be stored by default
    - Examples: Many LEonard example programs
    - Lib: Python and Java code libraries including cusom add-on by Lecky Engineering
- Config: Default location for all of the Devices files *.ldev
- Data: A good default directory for input and output data files
- DB: Leaonard database for Displays, Positions, Tools, and Variables. These are stored as simple XML.
- LEonard/bin: The location of the LEonard executables and support fiiles.
- Logs: Where the logfiles are created.
- 3rdParty: Where example programs for 3rd party devices such as the Universal Robot code and code for the LMI Gocator.

All About Leonard Devices

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

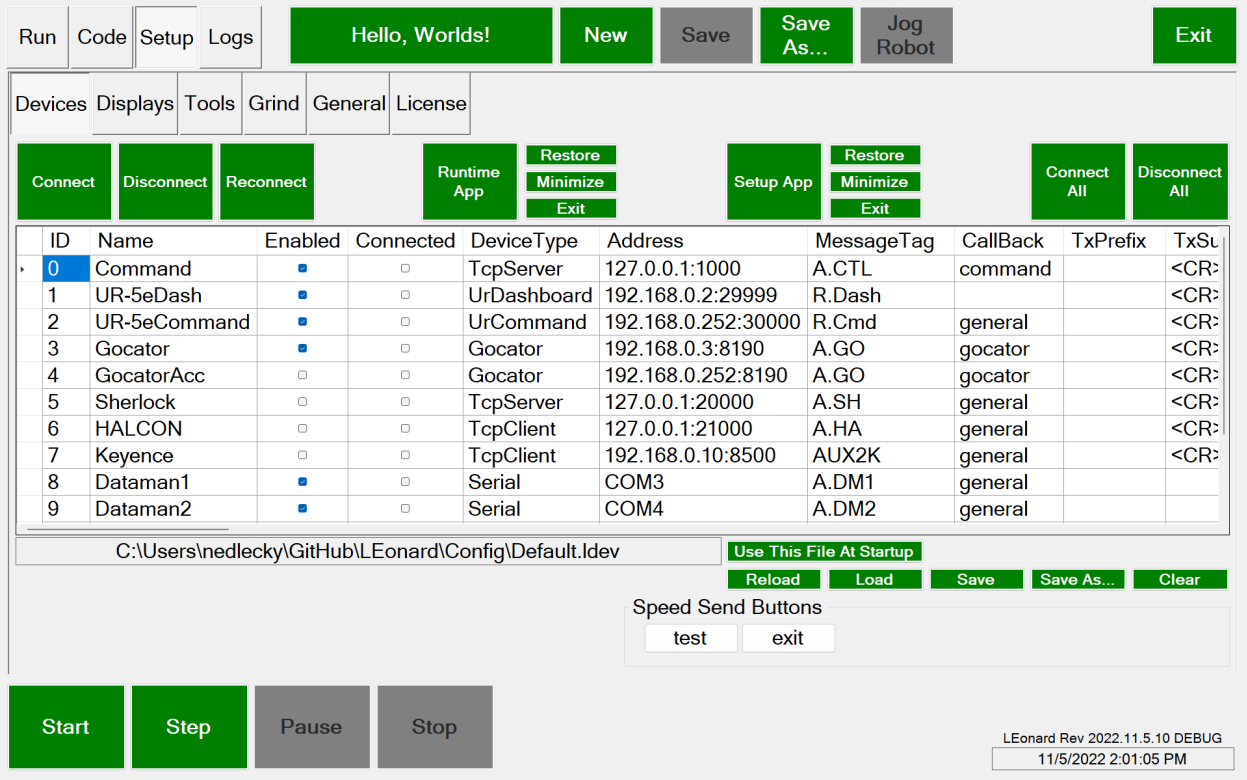


Figure 1 LEonard Setup | Devices Screen

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 nly 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. TCP Server
    b. TCP Client
    c. Serial
    d. Null
    e. Specialty
        i. Universal Robot Dashboard
        ii. Universal Robot Command
        iii. LMI Gocator Interface
5) Each device sets up a **CallBack**, 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

LEonard relies heavily on the use of a **LEonardStatement**.

When a **LEonardStatement** is analyzed, it is expected to look like 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
2) filename.py Execute an entire Python file
3) LE:script Execute any LEScript statement
4) JE:script Execute any Java statement
5) PE:script Execute any Python statement
6) SET varName value. Identical to varName=value
7) GET varName. LEonard reponds with the current value of variable ‘varName’
8) varName = value. LEonard stores ‘value’ into the variable named ‘varName’ for LEScript, Java, and Python

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.

A **LEonardMessage** is one or more **LEonardStatements**.
Single: LEonardMessage<TERM>
Double: LEonardMessage<SEP> LEonardMessage<TERM>
Mmore LeonardMessages can be added on.

The advantage of a multiple statement message is that all of the messages will be executed in rapid sequence as a unit.

LEonard Displays

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

However, LEonard can be used successfully on larger monitors. It is also designed to be resizable and to be usable even on big monitors for setup and other purposes.

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

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

LEonard Programming

LEonard provides three common programming languages to enable devices to be controlled and sequenced. So how do we do our Hello, World!

Well, let's first talk about printing to a console screen.

An Aside: The Print Command

LEonard provides a unified print console for debugging. Show or hide the console with the F12 key.

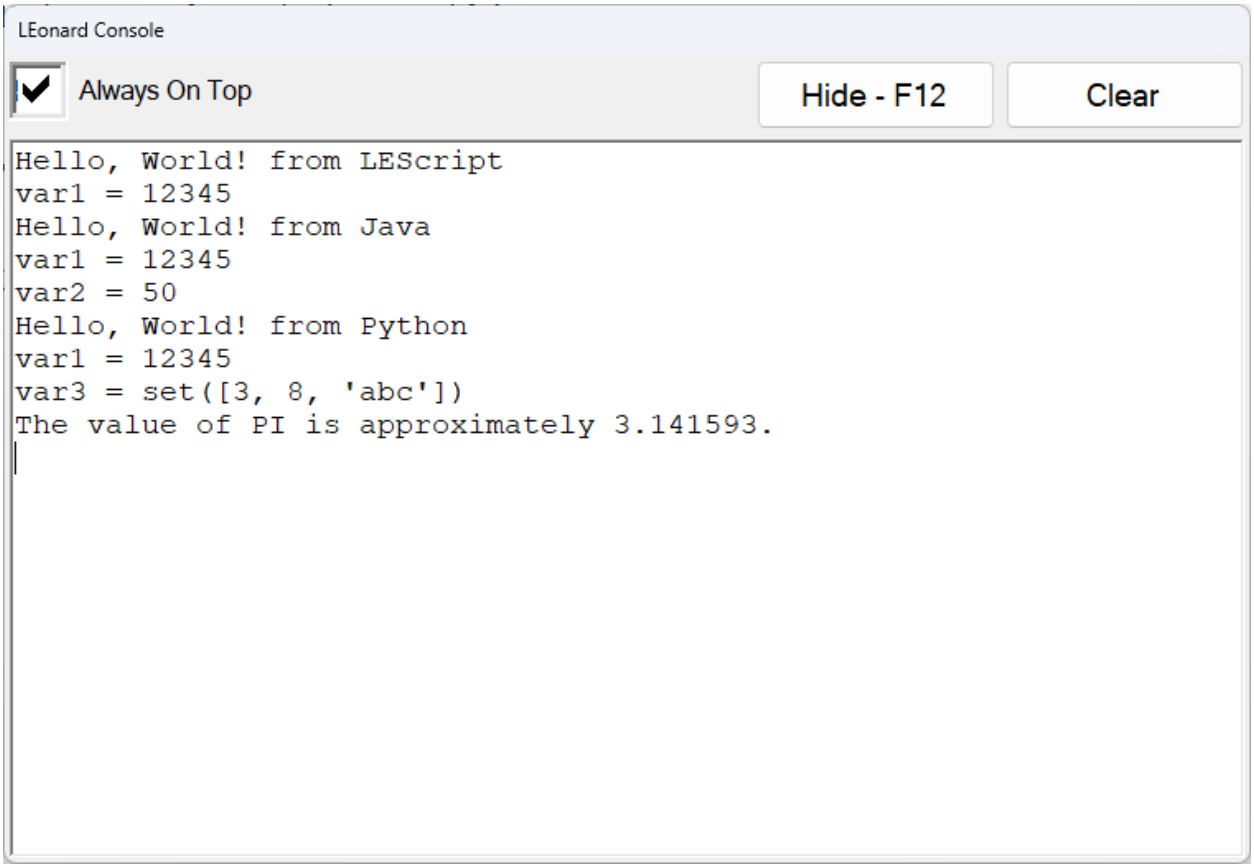


Figure 2 LEonard Console Window

Pressing F12 repeatedly shows and hides the console. It is always receiving and buffering lePrint command data.

The command leShowConsole(true|false) will also show or hide the console and works in any language.

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

OK, now we can try that Hello, World!

LEScript

LEScript is a simple scripting language 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, LEonard provides control using either Java or Python, discussed below.

```
# Hello, World! In LEScript
UsingLEScript()
lePrint(Hello, World!)
value = 13.25
lePrint(value = {value})

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

An exhaustive list of all the LEScript commands is located in **Appendix 1: LEScript Commands**

Java

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

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 program, commands are interpreted as LEScript until UsingJava() is encountered. Subsequent lines will be executed using Jint.
2. The LEScript command 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 LEonard::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
UseJava()
lePrint('Hello, World!')
value = 13.25 * 8.1
lePrint('value = ' + value)

Console:
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 is stable!

Specification:	
IronPython 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 Java execution can be triggered in LEonard

5. In a LEonard program, commands are interpreted as LEScript until UsingPython() is encountered. Subsequent lines will be executed in Python.
6. The LEScript command 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 “JE:javascript” The specified Python commands will be executed in Python.
8. The test area in the **Code** | **Python** tab. This provides a “sandbox” where Java programs can be created, edited, saved, and retrieved.

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

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

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

Theory of Operation

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

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

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

This maty 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.

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 program. This way, you can go fast when you need to, and when the error conditions don't create unsafe machine situations.

Since LEonard executes Java and Python line-at-a-time in the main run 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.

The Diagram

User Modes

There are 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 programs 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.

System Tabs

In the main LEonard screen, there are four Operation Tabs: **Run**, **Code**, **Setup**, and **Logs**. These screens 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 italics 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. Annunciators and control buttons appear on the Run Tab depending on what options are included and what kinds of devices you are connected to.

Run Tab- No Options

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

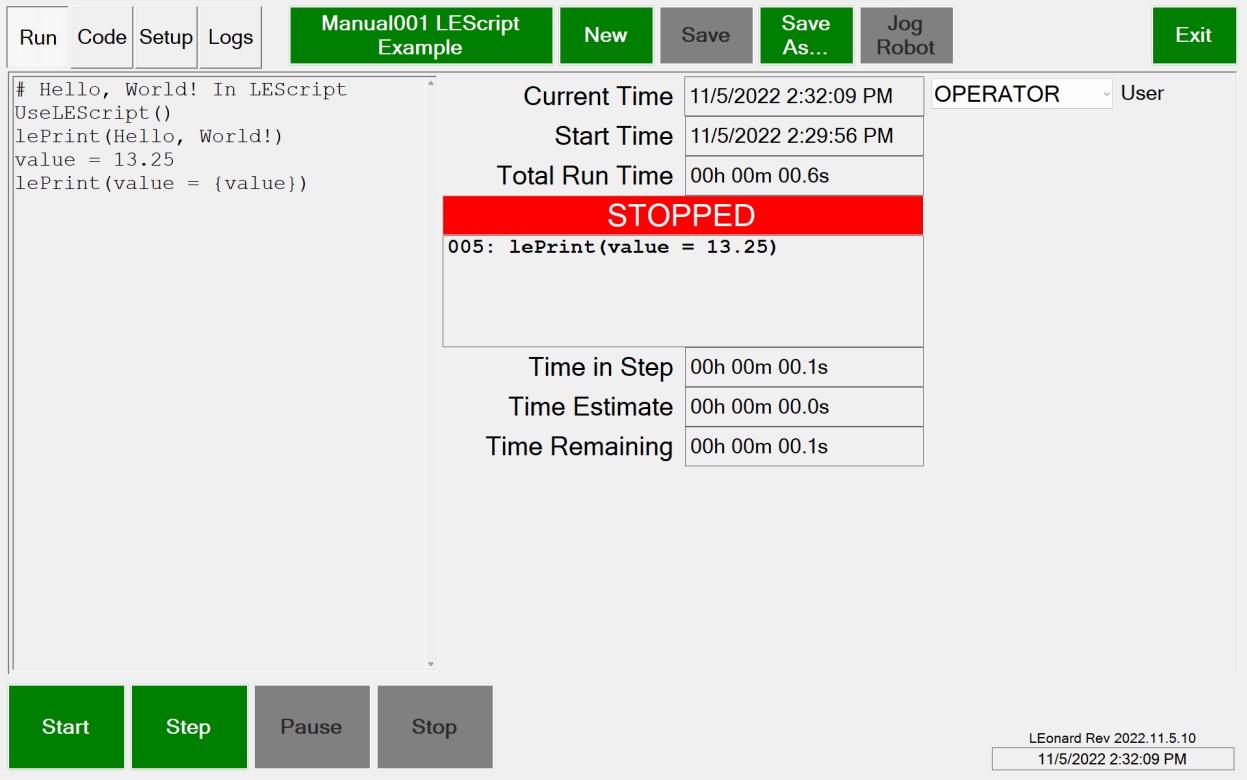


Figure 3 LEonard Run Screen with No Options Installed

A program is loaded using the **Program Name** button next to the **Logs** tab.
Program file operations in addition to **Load** are **New, Save, Save As.**
Set the User in the **User Dropdown.**

Run Tab- UR Dashboard Connection Only

Connecting a UR Robot Dashboard makes the robot control annunciators visible:

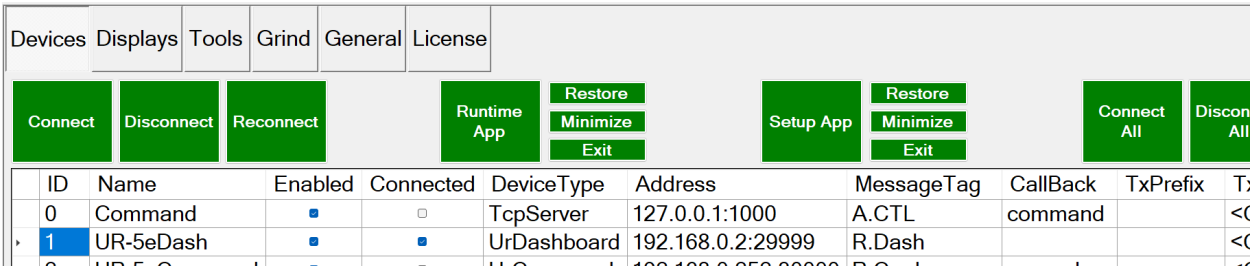


Figure 4 LEonard Connected to UR Dashboard

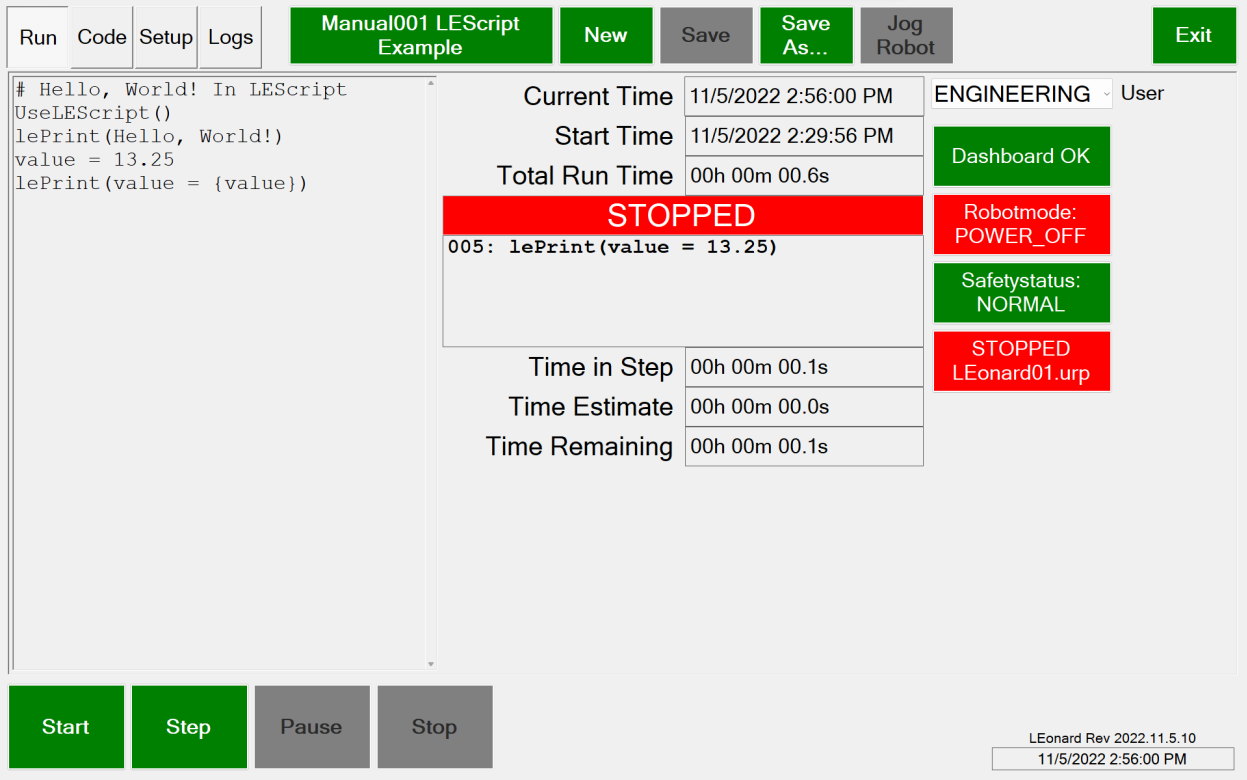


Figure 5 LEonard Main Screen when Connected to UR Dashboard

Dashboard OK: Shows connection status to the Dashboard (this is a TCP client).
Robotmode: Press this to cycle from POWER_OFF to BOOTING to IDLE to RUNNING
Safetystatus: Press this to clear robot stop conditions
STOPPED/PLAYING: Use this to toggle between a PolyScope program running or not on the robot. The program loaded is specified in the Device entry and is shown in this button.

Most of these options require that the UR is in Remote mode.

Run - UR Dashboard and Command Connection

With a UR Command interface also connected, the full UR control system is available. This requires that a special PolyScope program be installed and running on the robot. This program is provided by Lecky Engineering and can be modified by the user if you need special functions. Contact Lecky Engineering for assistance if you are new to UR programming!

The images below assume the optional UR Grinding package is also licensed.

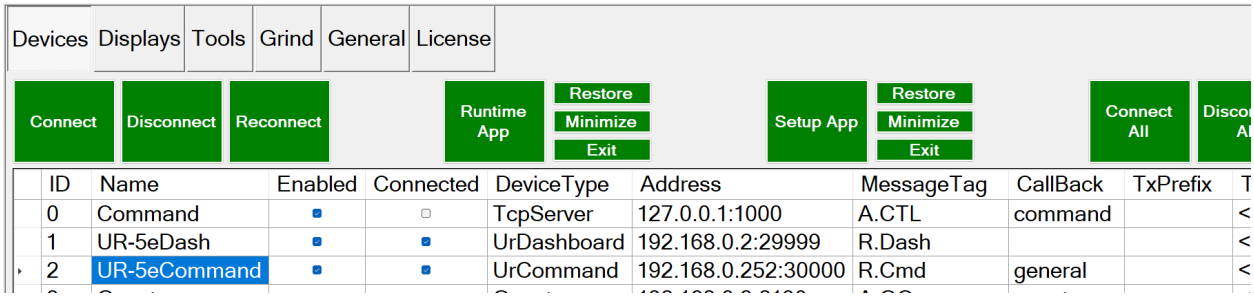


Figure 6 LEonard Connected to UR Dashboard and UR Command

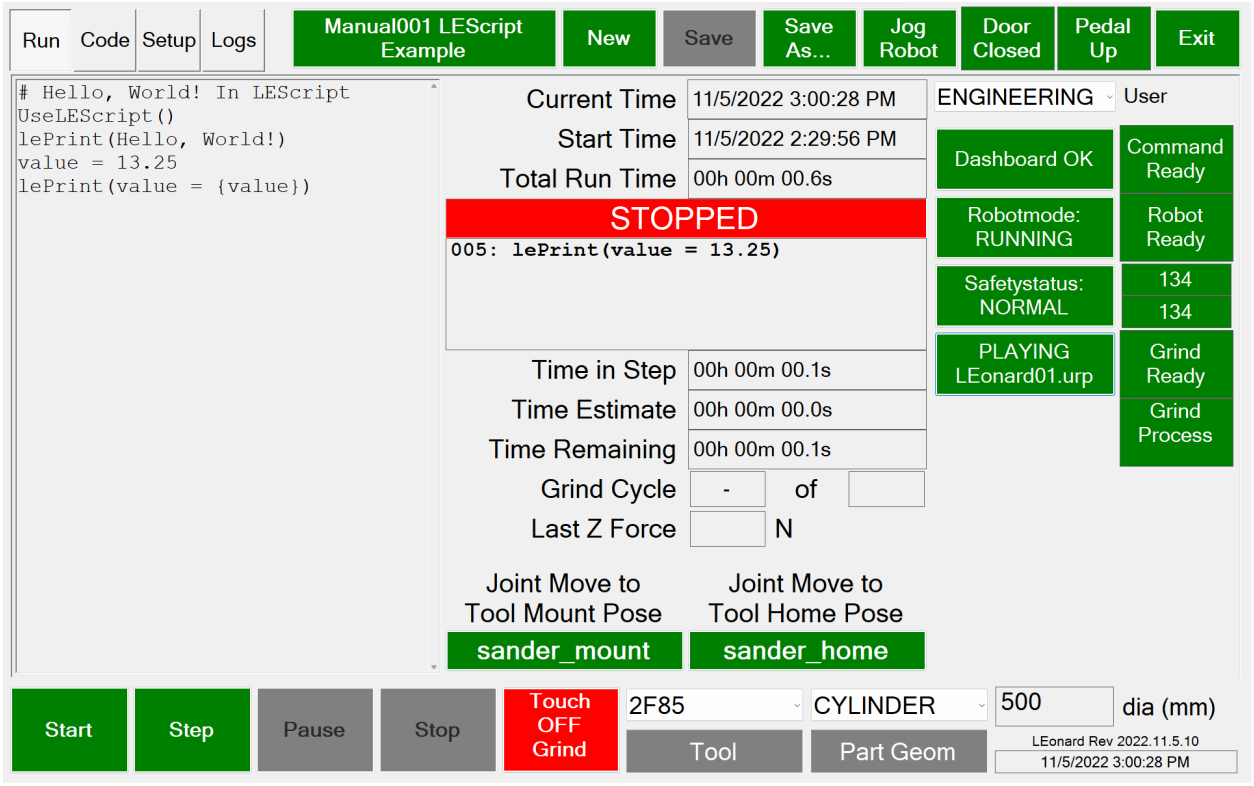


Figure 7 LEonard Main Screen when Connected to UR Dashboard and UR Command

With the optional Grinding Package enabled we get 2 additional buttons:
Grind Ready: Has the current grinding process completed?
Grind Process: Are we leaving the tool in contact with the part and expecting the next grind command withing a few seconds?

Universal Robot Option Controls

Command Ready: Shows status of UR Command connection (this is a TCP server).
Robot Ready: Shows if the robot is currently executing a LEonard command.
Command Index Numbers: Shows index number of last command sent and response ID of last response received. These should stay in lockstep and must be equal to allow the next command to be issued in the program.
Tools are selected in the **Tool Dropdown.** More information on Tools in in Setup - Tools

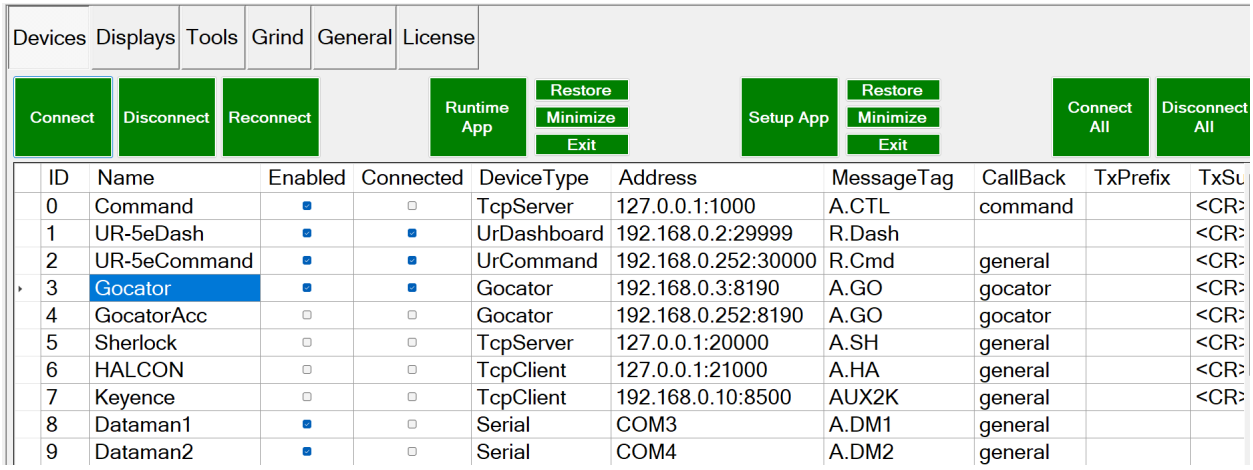
Part geometry in specified in **Part Geometry Dropdown:** The UR Command program supports surfaces that are FLAT, CYLINDER, or SPHERE
Tools have mount and home positions that can be moved into with the **toolname_mount** and **toolname_home** buttons.
Running a Program behaves as expected:
Start, **Stop**, **Step**, and **Pause / Continue**
Jog Robot is used to jog or freedrive to a defect. Jogging is described on the next page.

UR Grinding Option Controls

Set the grinding mode with the **Touch/Grind button** which cycles through **No contact, Touch Only, and Touch+Grind**
Protective Stops: These show up in (and may be cleared with) the SafetyStatus button
Door Status is monitored (IO is configured in the **Setup Tab**). Door Open is treated like **Pause**.
Footswitch Status is monitored (IO is configured in the **Setup Tab**).

Run - UR Dashboard and Command Connection plus LMI Gocator

Adding an LMI Gocator connection reveals the final (for now!) custom buttons:



	ID	Name	Enabled	Connected	DeviceType	Address	MessageTag	CallBack	TxPrefix	TxSu
	0	Command	■	□	TcpServer	127.0.0.1:1000	A.CTL	command		<CR>
	1	UR-5eDash	■	■	UrDashboard	192.168.0.2:29999	R.Dash			<CR>
	2	UR-5eCommand	■	■	UrCommand	192.168.0.252:30000	R.Cmd	general		<CR>
▸	3	Gocator	■	■	Gocator	192.168.0.3:8190	A.GO	gocator		<CR>
	4	GocatorAcc	□	□	Gocator	192.168.0.252:8190	A.GO	gocator		<CR>
	5	Sherlock	□	□	TcpServer	127.0.0.1:20000	A.SH	general		<CR>
	6	HALCON	□	□	TcpClient	127.0.0.1:21000	A.HA	general		<CR>
	7	Keyence	□	□	TcpClient	192.168.0.10:8500	AUX2K	general		<CR>
	8	Dataman1	■	□	Serial	COM3	A.DM1	general		
	9	Dataman2	■	□	Serial	COM4	A.DM2	general		

Figure 8 LEonard Connected to UR Dashboard/Command and LMI Gocator

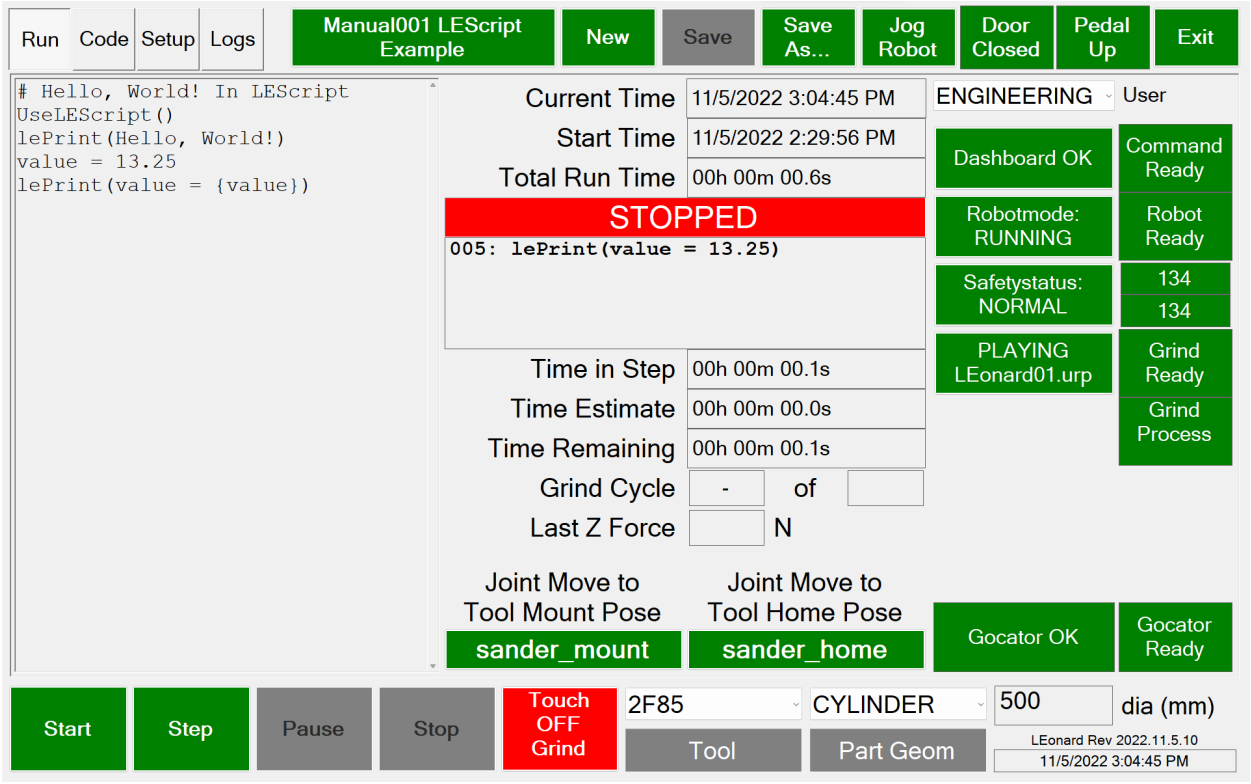
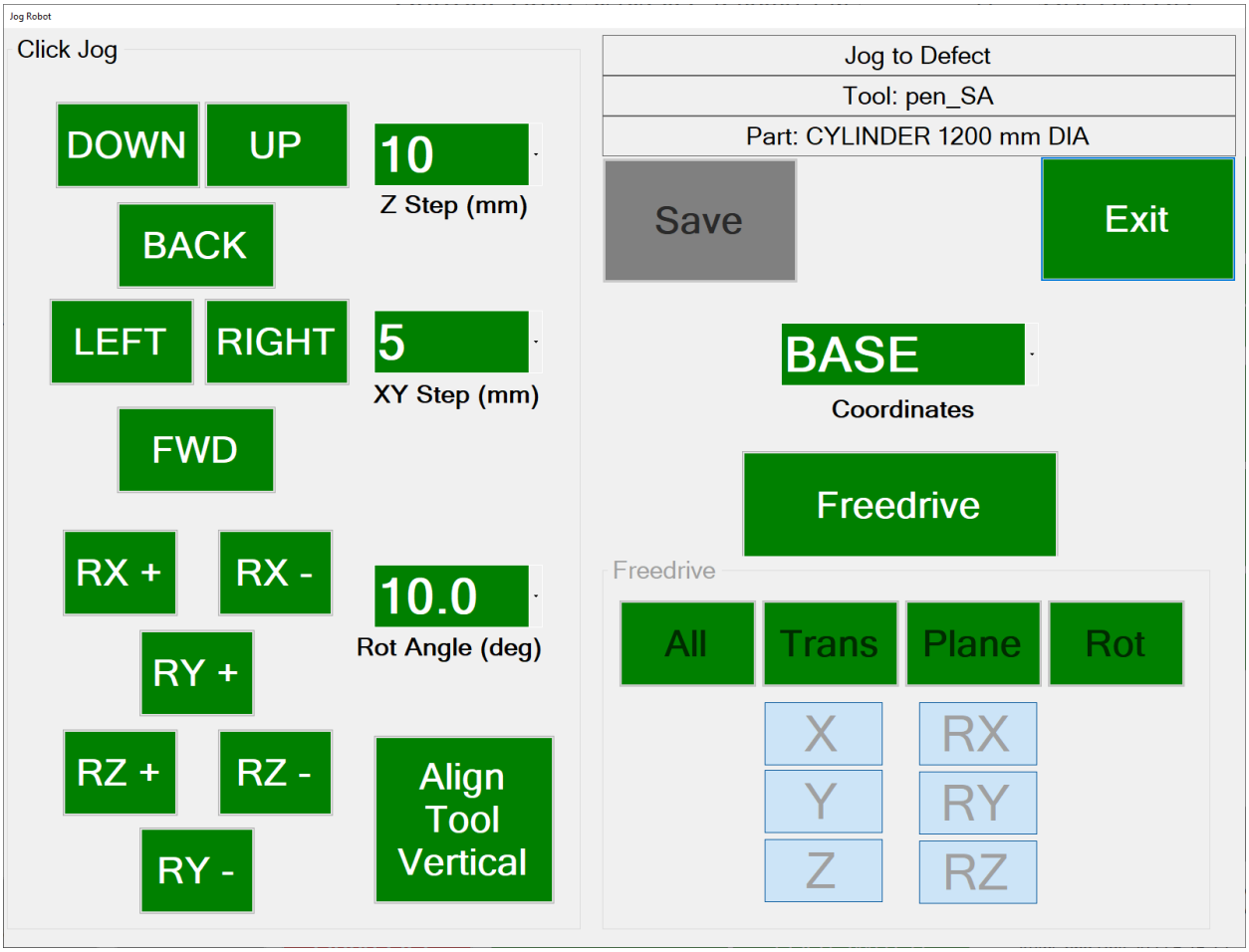


Figure 9 LEonard Main Screen when Connected to UR Dashboard/Command and LMI Gocator

Gocator OK: Is the connection with the Gocator (a TCP client) healthy?
Gocator Ready: Goes red when the Gocator is processing.

Robot Jogging in LEonard

Jogging opens a separate screen. Jogging can be done in **BASE** or **TOOL** coordinates, or relative to a **PART**. The buttons move the robot by the specified increment in Z, XY, or rotation. Holding a button down (mouse) or double-tapping and holding (tablet) makes the move repeat.



When jogging in **PART** mode, if a cylindrical or spherical geometry is selected, the tool will rotate around the center of the part instead of around the tool tip. This can be convenient for manually jogging to a defect using the touch screen instead of freedrive

Freedrive is supported in a manner identical to on the UR pendant. The X, Y, X, RX, RY and RZ buttons may be used to enable or disable freedrive in any desired axis. All, Trans, Plane, and Rot select pre-defined subsets of axes as on the UR.

Coordinate systems may be changed during freedrive and the tool will allow motion relative to the world, the tool, or the center of the part of part geometry is cylinder or sphere.
Press the Freedrive button again to turn freedrive mode off. Saving or exiting the dialog will also turn off freedrive.

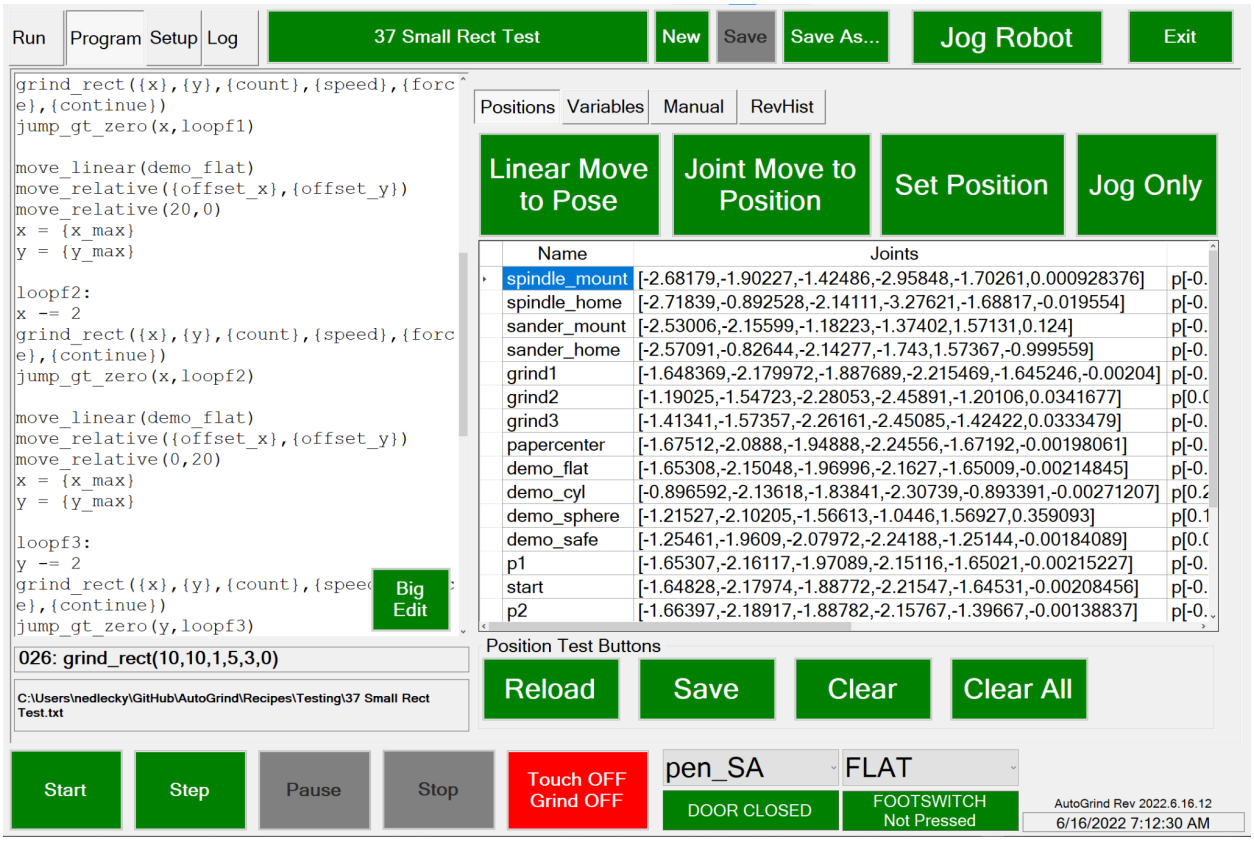
The **Freedrive footswitch** puts all 6 axes in freedrive whether this jog dialog is open or not.

Code Tab

The **Code Tab** has six subpages: **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,
Manual, to provide access to documentation, and **RevHist**, which displays what has been added in each version of the software. 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.



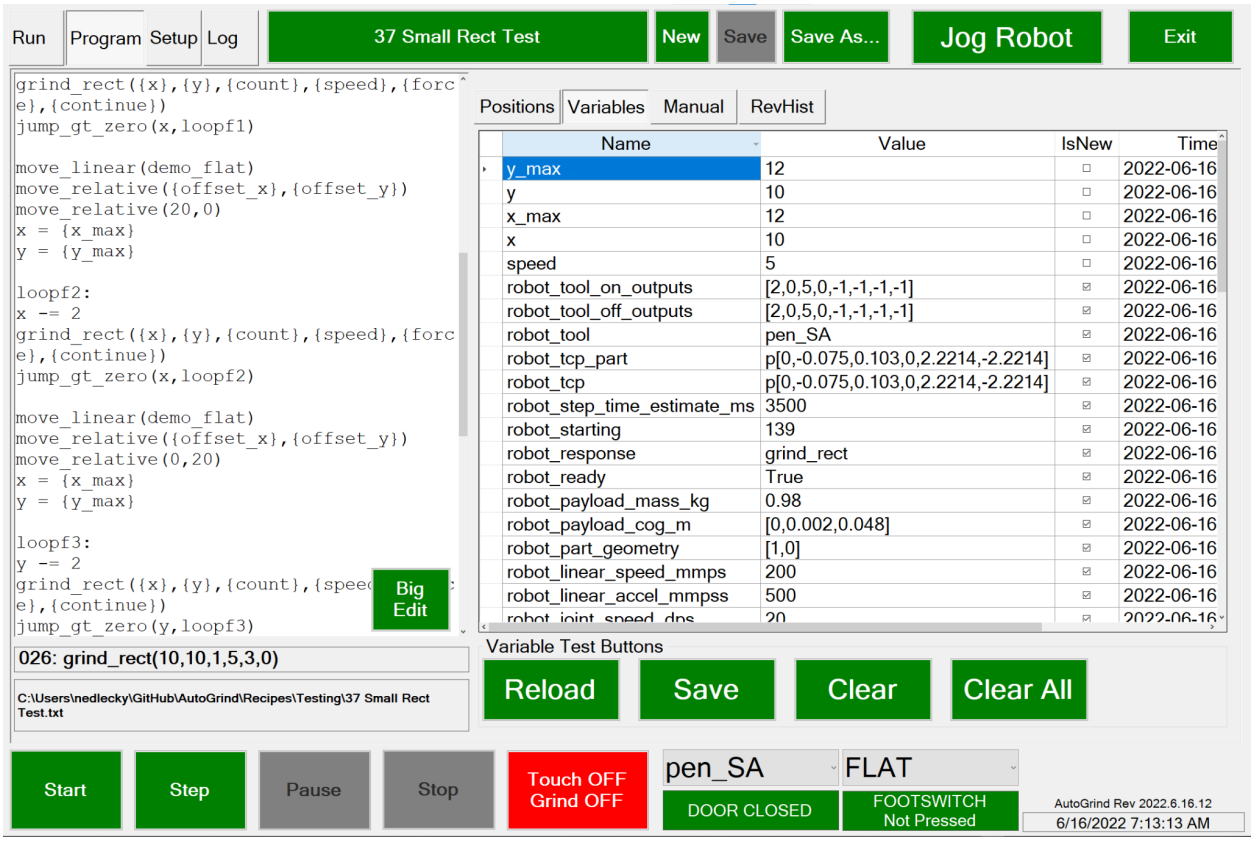
Positions can be saved manually (**Set Position**) or from the recipe with *save_position(name)*.

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.

Code | Variables

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



This tab shows all of 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 recipe *clear()* 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 may be saved or reloaded from Recipes/Variables.xml with the **Save** and **Reload** buttons. Variables are automatically saved on program exit and reloaded when the program starts.

Code | Java

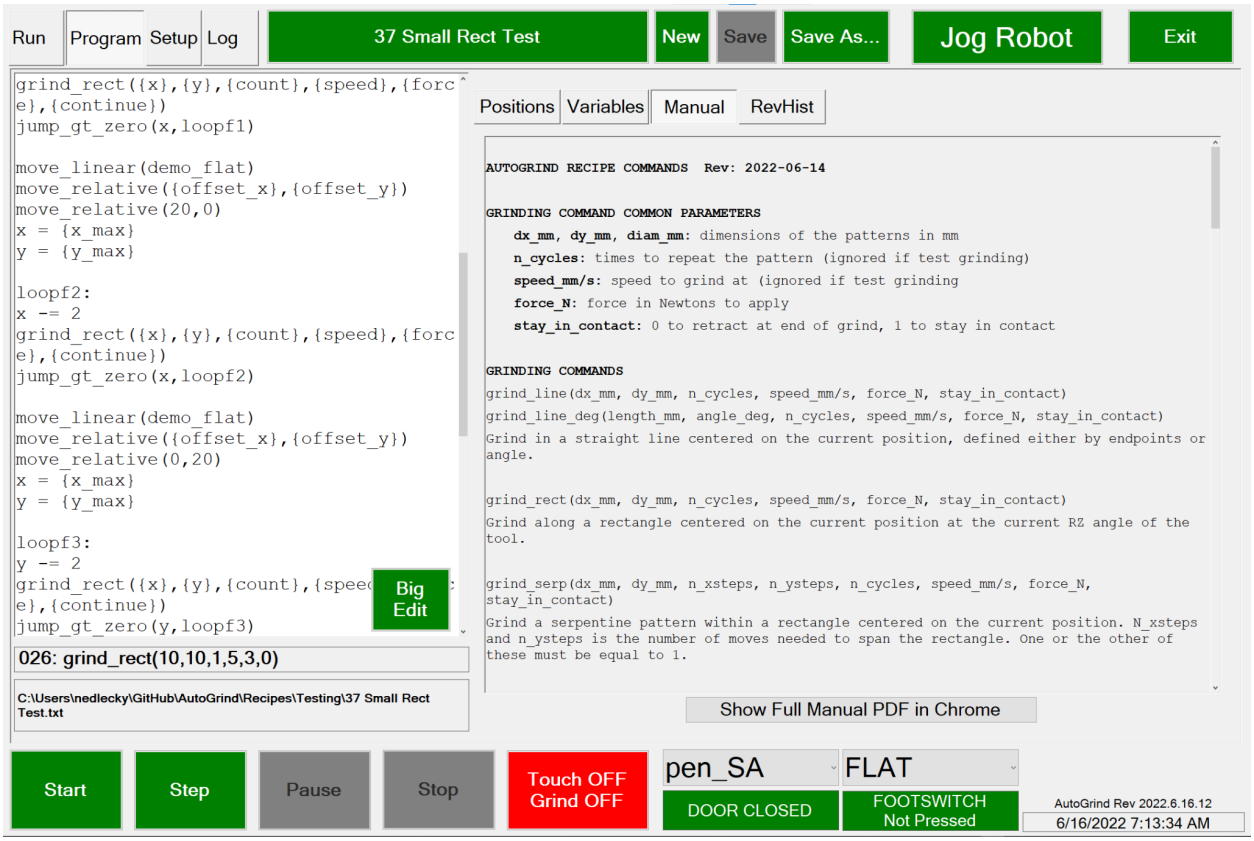
More here TODO

Code | Python

More here TODO

Code | Manual

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

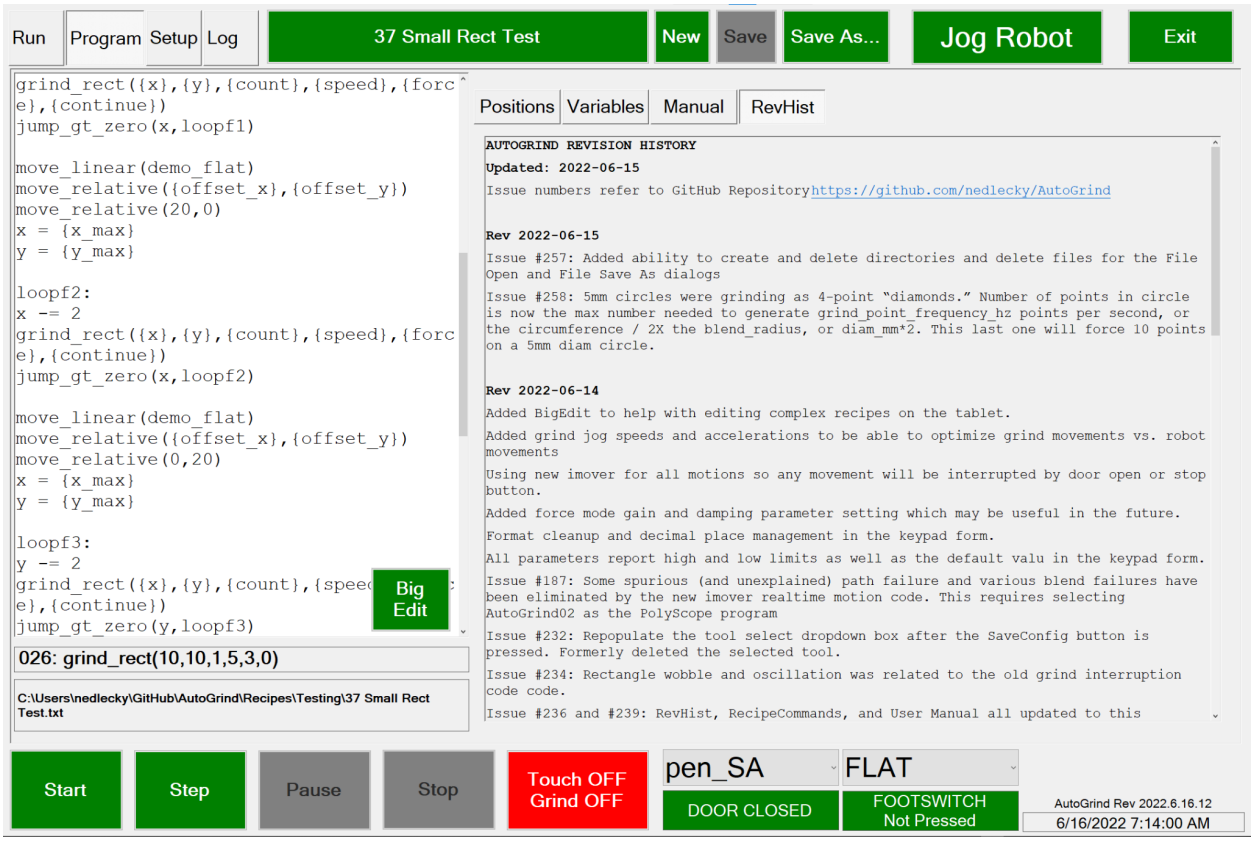


This tab displays the recipe commands to help you remember what each command does and how it is called..

The **Show Full Manual PDF in Chrome** button pulls up a PDF of the manual you are looking at in Chrome. It is assumed that Chrome is installed on the system.

Code | RevHist

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

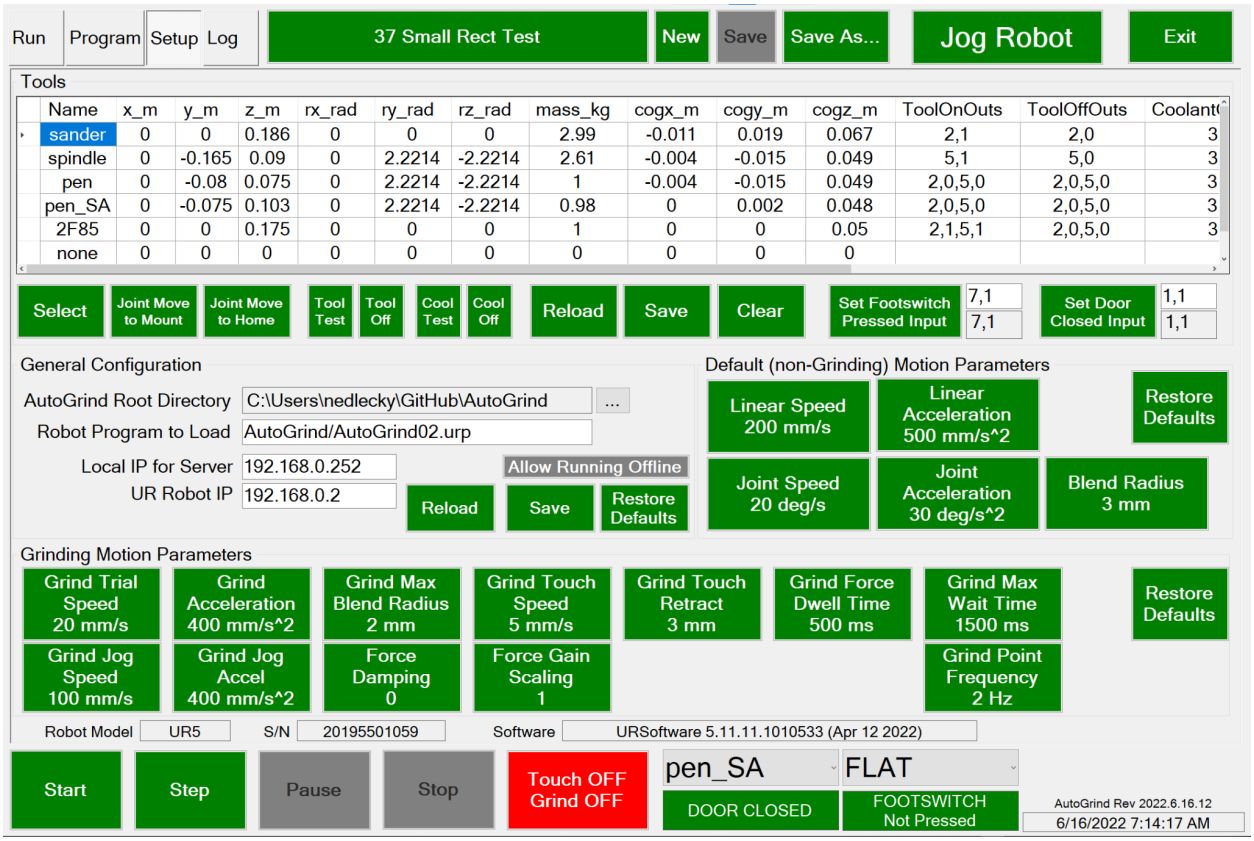


This tab displays the changes incorporated in the current (and prior) versions of the LEonard program. The issues addressed are described in more detail in the LEonard GitHub repository.

https://github.com/nedlecky/LEonard

Setup Tab

The **Setup Tab** is where all system configuration takes place.



Setup | Devices

The LEonard device list is a datafile create in the LEonardRoot/Data directory. You can have several device files and load different ones for different testing or operational situations.

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 fuile 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, They 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 simpe 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 TYCP 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.

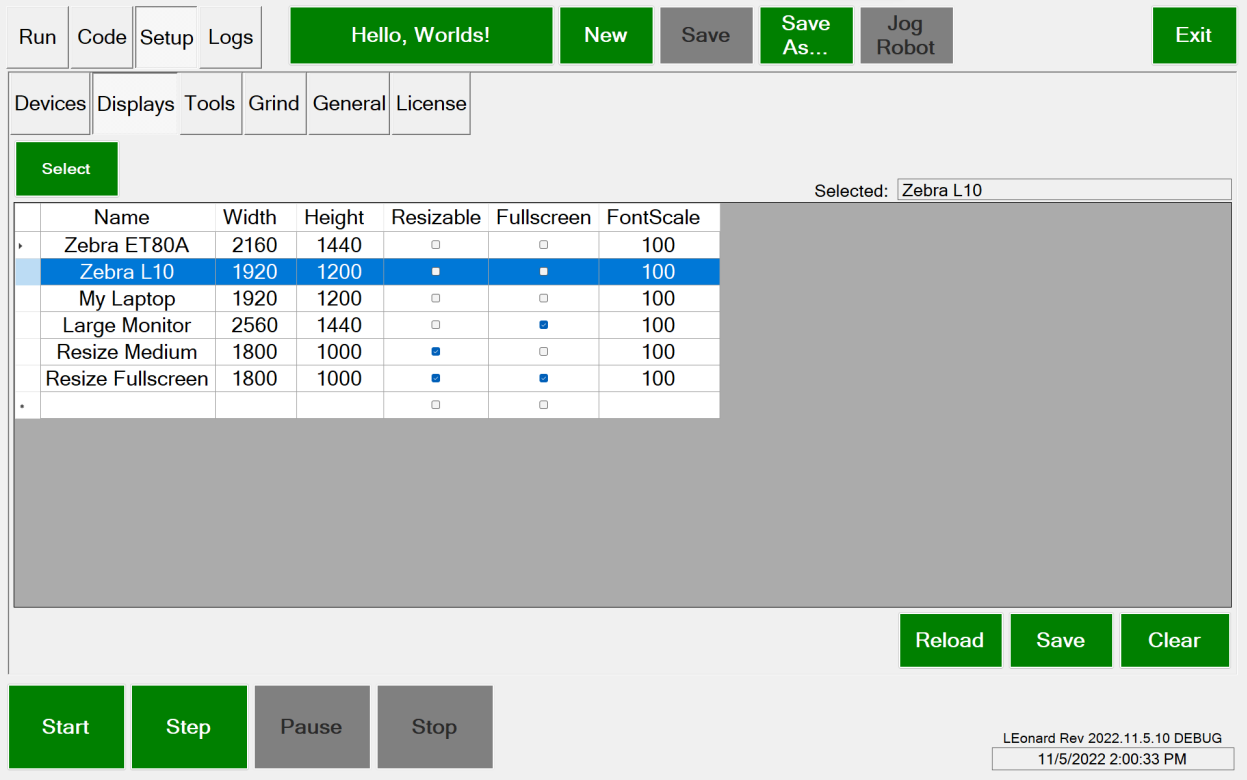


Figure 10 LEonard Setup | Displays Screen

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 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

Tools are defined in the Tool Table. Each contains the following information. These are saved in the Tools.xml file in LEonardRoot/Recipes 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. The system will use 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 homefor this tool. The system will use 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

First, there are settings governing grind operations. These are saved in the Variables.xml file in LEonardRoot/Recipes 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 touchoff. Recommended 5-10 mm/s
Grind Touch Retract: Distance robot retracts from part after touchoff.
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 system 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 speeds and accelerations used in jogging and non-grinding motion. These are saved in the Variables.xml file in LEonardRoot/Recipes and are sent to the robot whenever the software starts. New values are saved automatically. **Restore Defaults** sets all to standard values used in all testing.

Setup | General

1. **LEonard Root Directory:** Location where subdirectories Recipes and Logs will be created. Tools, Variables, and Positions are also stored here in the Recipes subfolder.
2. **Robot Program To Load:** Specifies the program on the UR that the UR will load and run when this software starts. Starting in version 2022-06-07, this should be **AutoGrind/AutoGrind02.urp**
3. **Local IP for Server:** This should be the IP address of the port on the host computer that is connected to the UR
4. **UR Robot IP:** The IP address that the UR is set to
5. **Allow Running Offline:** Testing only… will allow recipes to run with no UR attached

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.

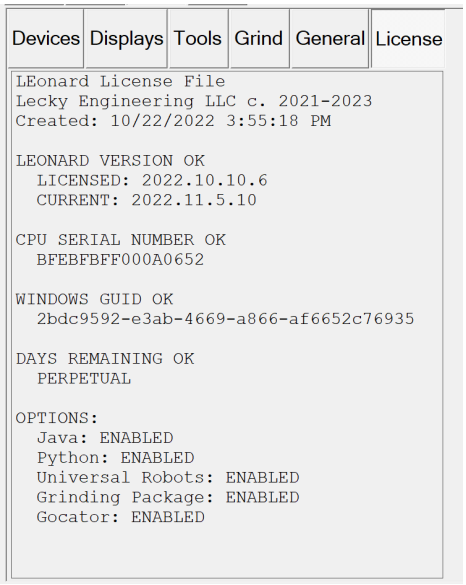


Figure 11 LEonard Setup | License Viewer

Access your license on the **Setup** | **License** tab. 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

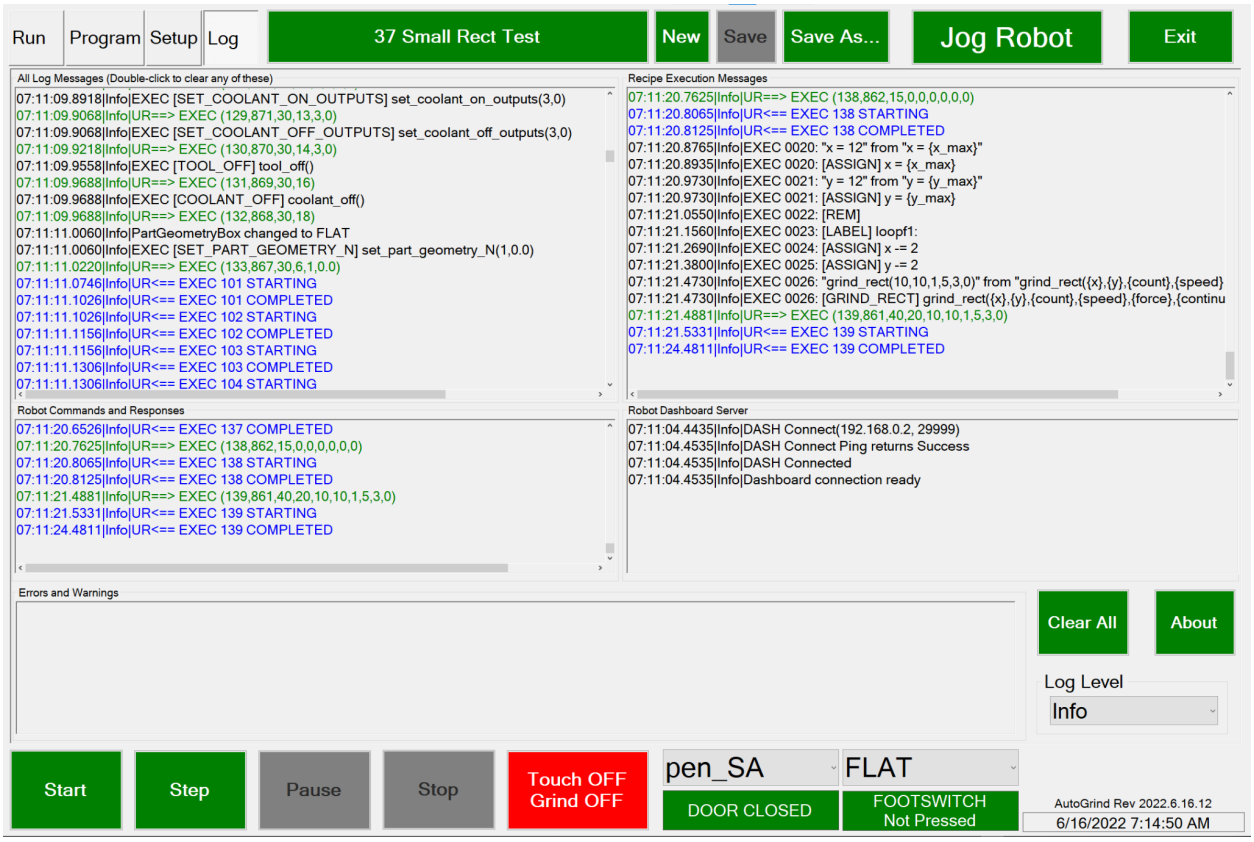
Contact Lecky Engineering to enable features and troubleshoot licensing issues on your system.

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 LEonardRoot/Logs directory, where up to forty 25MB files are archived. Information older than this 2GB total is automatically and silently deleted over time.



TODO THIS NEEDS UPDATING
In addition, some messages are copied for clarity to other boxes. The boxes are labeled with their respective data: Recipe Execution Messages, Robot Commands and Responses, Robot Dashboard (Control and Monitoring) Messages, and Errors/Warnings.
Any of the boxes can be cleared by double-clicking on them. All boxes can be cleared with **Clear All.** All of the messages are appended to the log files and are archived as described above.

LEonard Statements

Here is a comprehensive list of all LEonard statements that can be executed by the sequencer.

LElib Library, All Languages

The LElib Library provides common functions across all three languages.
Where differences exist, they will be highlights with the language matrix shown below.

LEScript	Java	Python
?	?	?

Variables and Data Structures

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.

Copying Variables Between LEonard and Java/Python

LEScript	Java	Python
	x	x

`string le_read_var(var_name)`

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

`le_write_var(string var_name, string value)`

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

`le_write_sysvar(string var_name, string value)`

Copies a variable from Java or Python to LEonard and marks it as a system variable. All LEonard variables are strings.
Read a file and process any lines that contain `variable_name = value`.

Limited LEScript Variable Handling

LEScript has a limited set of variable handling capabilities, described below:

LEScript	Java	Python
x		

`import_variables(filename)`

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

`clear()`

Deletes all variables except ones that are marked in the Variables Table as system variables. (Variables named robot_* and grind_* are automatically marked as system variables, otherwise you can use `le_write_sysvar(name, value)` from Java or Python to create them.

`LEScript Assignment`

LEScript supports updating variables using any of these basic operations. Variables can be inserted in 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
```

LElib.console: Console Control 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:	LE** message
Java:	JV** message
Python:	PY** message

`le_show_console(bool show?)`

Hides or shows the Console Window. The console is always open and accumulating any lePrint messages from any language.

`le_clear_console()`

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

`le_prompt(string message)`

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

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`

Comments in the sequencer are ignored as follows:

1. All blank lines are skipped
2. LEScript and Python statements ignore characters after "#" on any line
3. Java statements ignore any characters after ";" on any line

`Label_name:`

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 `jump_gt_zero(variable, labelName)` function since it does not presently have the ability to evaluate comparison conditions.

`jump(string labelName)`

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

`jumpif(bool condition, string labelName)`

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

`call(string labelName)`

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

`callif(bool condition, string labelName)`

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

`ret()`

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

`jump_gt_zero(string varName, string label)`

Only available in LEScript since comparisons aren't available there. Deprecated. Equivalent to `jumpif(varName > 0, labelName).`

`end()`

Halts execution of the sequencer.

`assert(bool condition)`

Testing support function.

1. In LEScript: used as `assert(varName, value).` The function checks to see if var_name == value and generates an error message if not.
2. In Python and Java, The function generates an error dialog if `condition != True.`

`sleep(double 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!

LElibUR Library for Universal Robots

These functions work with Universal Robots robotic systems. The commands fall into three categories

1. Dashboard
2. Command Interface to Lecky Engineering's LEonard PolyScope program
3. Grinding Package for force-controlled surface following

LElib.UR.dashboard: Commands to control the UR dashboard

The UR robot provides a dashboard interface that allows controlling the robot operation.

`string ur_dashboard(string message, int timeout_ms)`

Sends the command `message` to the currently selected Universal Robot dashboard connection and waits for up to `timeout_ms` milliseconds for a response.

Response:
LEScript: Any response received is placed in the variable `ur_dashboard_response`
Java, Python: Function returns any string received or and empty string.

The UR dashboard system provides many commands that are useful in loading, starting, and stopping the robot. The Run tab in LEonard uses `robotmode` to determine whether the robot has booted, and regularly sends `robotmode`, `safetystatus`, and `programstate` to keep an eye on the robot.

When you press the **Robot Mode** button, LEonard cycles through the robot modes as appropriate- RUNNING initiates sending `power off`. IDLE initiates sending `brake release`. And POWER_OFF initiates sending `power on`. This allows you to cycle through UR operating modes.
The **Safety Status** button sends `unlock protective stop` and `close safety popup` when the robot is in safety stop but not in E-Stop.

The **Program State** button toggles between sending `play` and `stop` to start and stop the loaded PolyScope program. The UR Dashboard device sends a `load jobFile` command when the UR connects with the dashboard to get your default PolyScope program loaded.

A comprehensive discussion of the dashboard interface is available on the UR website:
https://www.universal-robots.com/articles/ur/dashboard-server-e-series-port-29999/

LElib.UR.robot: The UR Robot PolyScope Interface

These commands interact with the Lecky Engineering PolyScope program used to support robot control and the grinding package.

`select_tool(string tool_name)`

Setup all of the necessary environment to be able to use `tool_name`. No motion is performed. Future tool moves, position moves, and grinds will assume this tool is attached.

`set_part_geometry(string FLAT|CYLINDER|SPHERE, double part_diam_mm)`

Future tool moves and grinds will assume the specified geometry.

`save_position(position_name)`

The current robot position is stored in the Positions Table as position_name.

`move_linear(position_name)`

The robot moves along a linear path to Position position_name.

`move_joint(position_name)`

The robot performs a joint move to Position position_name.

`move_relative(dx_mm, dy_mm)`

Move (dx_mm, dy_mm) relative to current tool position. If the part geometry selected is CYLINDER or SPHERE, robot moves along the part.

`move_tool_home()`

Perform a joint move to the home position associated with the current tool.

`move_tool_mount()`

Perform a joint move to the mounting position associated with the current tool.

`free_drive(0=OFF|1=ON)`

Turn robot free drive mode on or off.

The commands below provide a programmatic way to set the default motion parameters.

`set_linear_speed(speed_mm/s)`

Sets default linear speed used for robot linear moves.

`set_linear_accel(accel_mm/s^2)`

Sets default linear acceleration used for robot linear moves.

`set_joint_speed(speed_deg/s)`

Sets default joint speed used for robot joint moves.

`set_joint_accel(double accel_deg/s^2)`

Sets default joint acceleration used for robot joint moves.

`set_blend_radius(double blend_radius_mm)`

Sets default blend radius used in all robot moves.

`get_actual_tcp_pose()`

Ask the current robot to perform `get_actual_tcp_pose()` and return the value in the LEonard variable `actual_tcp_pose`.

`get_target_tcp_pose()`

Ask the current robot to perform `get_target_tcp_pose()` and return the value in the LEonard variable `target_tcp_pose`.

`get_actual_joint_positions()`

Ask the current robot to perform `get_actual_joint_positions()` and return the value in the LEonard variable `actual_joint_positions`.

`get_target_joint_positions()`

Ask the current robot to perform `get_target_joint_positions()` and return the value in the LEonard variable `target_joint_positions`.

`get_actual_both()`

Performs both `get_actual_joint_positions()` and `get_actual_tcp_pose()` on the current robot and return the values to the LEonard variables `actual_joint_positions` and `actual_tcp_pose`.

`get_target_both()`

Performs both `get_target_joint_positions()` and `get_target_tcp_pose()` on the current robot and return the values to the LEonard variables `target_joint_positions` and `target_tcp_pose`.

`movej(double j1, j2, j3, j4, j5, j6)`

Performs a `movej` to **joint positions** on the current robot as follows:

```
q = [j1, j2, j3, j4, j5, j6]
movej(q, a=robot_joint_accel_rpss, v=robot_joint_speed_rps)
```

`movel(double x, y, z, rx, ry, rz)`

Performs a `movel` to a **pose** on the current robot as follows:

```
p = p[x, y, z, rx, ry, rz]
movej(q, a=robot_joint_accel, v=robot_joint_speed)
```

`get_tcp_offset()`

Ask the current robot to perform `get_tcp_offset()` and return the value in the LEonard variable `tcp_offset`.

`movel_incr_base(double x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in base coordinates as in URScript:

```
local p0 = get_target_tcp_pose()
local p1 = p[x,y,z,dx,dy,dz]
local p2 = pose_add(p0, p1)
if p1[0] == 0 and p1[1] == 0 and p1[2] == 0:  # Rotational move
```

```
    movel(p2, robot_joint_accel_rpss, robot_joint_speed_rps)
  else:
    movel(p2, robot_linear_accel_mpss, robot_linear_speed_mps)
  end
```

`movel_incr_tool(double x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in TCP coordinates as in URScript:

```
  local p1 = p[x,y,z,rx,ry,rz]
  local p2 = pose_trans(get_target_tcp_pose(), p1)
  if p1[0] == 0 and p1[1] == 0 and p1[2] == 0:  # Rotational move
    movel(p2, robot_joint_accel_rpss, robot_joint_speed_rps)
  else:
    movel(p2, robot_linear_accel_mpss, robot_linear_speed_mps)
  end
```

`movel_incr_part(x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in PART coordinates. X and Y are interpreted based on `set_part_geometry(…)`. For cylinders, X is along the axis of the cylinder and Y is interpreted as a fixed-distance rotation about the cylinder.

`movel_single_axis(axis,value)`

Ask the current robot to move to its current pose with the coordinate `axis` changed to `value`.

`movel_rot_only(rx,ry,rz)`

Ask the current robot to move to its current pose with the new rotations `rx`, `ry`, and `rz`.

`movel_rel_set_tool_origin(double x,y,z,rx,ry,rz)`

`movel_rel_set_tool_origin_here()`

Sets a tool-coordinate origin for the current robot either to a specified pose or to the current robot position. Subsequent calls to `movel_rel_tool()` will move in tool coordinates relative to this origin.

`movel_rel_tool(x,y,z,rx,ry,rz)`

Move to a tool coordinate position that is relative to the `movel_rel_set_tool_origin`.

`movel_rel_set_part_origin(x,y,z,rx,ry,rz)`

`movel_rel_set_part_origin_here()`

Sets a part-coordinate (FLAT, CYLINDER, or SPHERE) origin for the current robot either to a specified pose or to the current robot position. Subsequent calls to `movel_rel_part()` will move in part coordinates relative to this origin.

`movel_rel_part(x,y,z,rx,ry,rz)`

Move to a part coordinate position that is relative to the `movel_rel_set_part_origin`.

`send_robot(param1, param2, …)`

Sends any command to the Lecky Engineering PolyScope program. Can be used to extend the LEonard-UR interface!

`set_output(int port, bool value)`

Set UR digital output `port` to `value`.

`robot_socket_reset()`

Commands the Lecky Engineering UR PolyScope program to reset (bounce) its socket connection to LEonard. Program must be running on the UR!

`robot_program_exit()`

Commands the Lecky Engineering UR PolyScope program to terminate. Program must be running on the UR!

Low-level Setup Calls

These are all called automatically by `select_tool()`, `set_part_geometry()`,and the `grind_xxx()` functions. They should be used through that high level interface except during testing or for special purposes!

`set_tcp(x,y,x,rx,ry,rz)`

A

`set_payload(mass_kg,cog_x_m, cog_y_m, cog_z_m)`

A

`set_door_closed_input(int dig_in, int state)`

Specifies what digital input and state is expected to signify that the door is closed to the current robot.

`set_footswitch_pressed_input(int dig_in, int state)`

Specifies what digital input and state is expected to signify that the footswitch is pressed on the current robot.

`set_tool_on_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `tool_on()` is executed on the current robot.

`set_tool_off_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `tool_off()` is executed on the current robot.

`set_coolant_on_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `coolant_on()` is executed on the current robot.

`set_coolant_off_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `coolant_off()` is executed on the current robot.

`tool_on()`

Performs the `tool_on` output list set in `set_tool_on_outputs()` on the current robot.

`tool_off()`

Performs the `tool_off` output list set in `set_tool_off_outputs()` on the current robot.

`coolant_on()`

Performs the `coolant_on` output list set in `set_coolant_on_outputs()` on the current robot.

`coolant_off()`

Performs the `coolant_off` output list set in `set_coolant_off_outputs()` on the current robot.

LElib.UR.grind: The UR grinding package

The grinding commands use a set of common parameters described below:

`dx_mm, dy_mm, diam_mm`**:** dimensions of the patterns in mm
`n_cycles`**:** times to repeat the pattern (ignored if test grinding)
`speed_mm/s`: speed to grind at (ignored if test grinding

`force_N`: force in Newtons to apply
`stay_in_contact`: 0 to retract at end of grind, 1 to stay in contact

```
grind_line(dx_mm, dy_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

```
grind_line_deg(length_mm, angle_deg, n_cycles, speed_mm/s,
force_N, stay_in_contact)
```

Grind in a straight line centered on the current position, defined either by endpoints or angle.

```
grind_rect(dx_mm, dy_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

Grind along a rectangle centered on the current position at the current RZ angle of the tool.

```
grind_serp(dx_mm, dy_mm, n_xsteps, n_ysteps, n_cycles,
speed_mm/s, force_N, stay_in_contact)
```

Grind a serpentine pattern within a rectangle centered on the current position. N_xsteps and n_ysteps is the number of moves needed to span the rectangle. One or the other of these must be equal to 1.

```
grind_poly(circle_diam_mm, n_sides, n_cycles, speed_mm/s,
force_N, stay_in_contact)
```

Grind along a polygon of n_sides inscribed in circle_diam_mm centered on the current position.

```
grind_circle(circle_diam_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

Grind along a circle centered on the current position.

```
grind_spiral(circle1_diam_mm, grind_circle2_diam_mm, n_spirals,
n_cycles, speed_mm/s, force_N, stay_in_contact)
```

Grind along a variable diameter circle centered on the current position. The circle goes from the first diameter to the second in n_spirals full revolutions.

```
grind_retract()
```

Ensure not in contact with the part. Happens automatically if a non-grind command is sent, if stop or pause is selected, or if grind_max_wait timer expires.

`grind_contact_enable(0=Touch OFF,Grind OFF|1=Touch ON,Grind OFF|2=Touch ON,Grind ON)`

Set the grinding mode programmatically as shown.

The commands below provide a programmatic way to set the grinding parameters.

`grind_touch_retract(touch_retract_mm)`

Set grind retract speed used after touchoff.

`grind_touch_speed(touch_speed_mm/s)`

Set speed used to go in for touchoff in Z.

`grind_force_dwell(dwell_time_ms)`

A dwell time performed when force mode is turned on to allow the robot to settle against the grind surface.

`grind_max_wait(max_time_before_retract_ms)`

If the tool is left in contact with the surface awaiting the next grind command, it will retract if this timeout is exceeded.

`grind_max_blend_radius(grind_blend_radius_mm)`

Sets the maximum blend radius that will be used in any pattern. This will be reduces for small geometries.

`grind_trial_speed(trial_speed_mm/s)`

Sets the speed used for “air grinding” when not in Touch + Grind mode.

`grind_linear_accel(accel_mm/s^2)`

Sets the linear acceleration used for grinding operations.

`grind_point_frequency(point_frequency_hz)`

Sets a point interpolation frequency used for complex figures. Obsolete.

`grind_jog_speed(trial_speed_mm/s)`

Sets the speed used when the grinding requires a robot move while not in contact with the part.

`grind_jog_accel(accel_mm/s^2)`

Sets the acceleration used for grinding m,oves not in contact with the part.

`grind_force_mode_damping(damping: 0.0 – 1.0)`

Sets the UR force_mode_damping parameter to assist in stabilizing force-mode performance.

`grind_force_mode_gain_scaling(scaling: 0.0 – 2.0)`

Sets the force_mode_gain_scaling parameter to assist in stabilizing force-mode performance.

Grind User Timers: Internal Use

Enabling these will time each grind operation and place it in a circular buffer of user_timers that can be returned to the variable list with `return_user_timers()`. Used primarily for internal testing.

`enable_user_timers(integer 0=off, 1=on)`

Turn the UR-internal user timers on or off.

`zero_user_timers()`

Zero all UR-internal user timers.

`return_user_timers()`

Return an array of timers. Each timer represents one grinding operation. Repeating the same grinding operations on different surface geometries can be used to validate Lecky Engineering's internal speed calibration system.

Example Recipes

Here are a few recipes that show the kinds of things that can be done in a recipe. The **Testing** subdirectory in the Recipes folder has many more complicated examples that you can examine (and run!).

These examples are shown in LEScript and require slight edits in Java or Python sequences.

Remove Current Tool

Just remove the current tool from the robot. As long as the one actually mounted is selected, this goes to the tool home followed by the mount/demount position and prompts the operator when it is time to remove.

```
# Remove Current Tool
# Go through demount procedure
# Assumes you have selected whatever tool is actually mounted!

prompt(Please confirm: you wish to demount {robot_tool}?)
```

```
move_tool_home()
move_tool_mount()
prompt(Please demount tool {robot_tool})

select_tool(none)
```

Install A Tool

This goes through prompting to mount a specific tool.

```
# Install 2F85
# Example to install a tool when none is currently installed
# We just select the new tool, move to the mount position, prompt
the operator, and move to tool_home

# Change to whatever tool you like
tool=2F85

# Operator confirmation
prompt(About to mount {tool})

# Mounting process
select_tool({tool})  # This only informs the robot what is
mounted

# This does the physical swap
move_tool_mount()
prompt(Please mount tool {tool})
move_tool_home()
```

Integrated Example

Here we start with the 2F85 tool ready to grind and swap tools and continue from the same location mid-recipe.

```
# Integrated Example
# Assumes we're where we want to grind initially but need to do a
tool swap mid-way

tool1=2F85
tool2=vertest

# Program assumes we are starting with tool1- verify internally
and with operator!
assert(robot_tool,{tool1})
```

```
prompt(Confirming tool {tool1} is currently mounted and you are
grinding on {robot_geometry})

# This will always be our grind_start position
save_position(grind_start)

# Do some grinding with tool1
move_linear(grind_start)
grind_rect(30,30,3,10,10,1)
grind_rect(20,20,3,10,10,1)

prompt(Ready to swap {tool1} to {tool2}?)
# Remove {tool1}
move_tool_home()
move_tool_mount()
prompt(Please remove {tool1})

# Install {tool2}
select_tool({tool2})
move_tool_mount()
prompt(Please install {tool2})
move_tool_home()

# Do some grinding with tool2
move_linear(grind_start) # Returns us to the starting position
grind_rect(30,30,3,10,10,1)
grind_rect(20,20,3,10,10,1)
```

Computed Concentric Circles

Here's a test recipe that grinds 3 concentric circles explicitly and in a loop, not lifting until the final one.

```
# 26 Concentric Circle Test

# Old school
grind_circle(30,2,0.9,10,1)
grind_circle(20,2,0.9,10,1)
grind_circle(10,2,0.9,10,0)

# Do it with a loop
size = 30
count = 2
speed = 0.9
force = 10
```

```
repeat:
grind_circle({size},{count},{speed},{force},1)
size -= 10
jump_gt_zero(size,repeat)
```

Lots of Grinds

By pre-teaching points and swapping geometries, a whole day's work could be done (other than tool swaps!)

```
# Test all the patterns on all the geometries

size1=40
size2=10
count=3
speed=5
force=10

select_tool(2F85)
cycleCount=0

redo:
move_linear(demo_flat)

set_part_geometry(FLAT,0)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,400.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
```

```
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,600.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,800.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,1000.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,400.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)
```

```
set_part_geometry(SPHERE,600.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,800.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,1000.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

cycleCount++
jump(redo)
```

Index