

Using Background Programs

on the LI-6800 Portable Photosynthesis System



Version 1.4

LI-COR[®]

Using Background Programs on the LI-6800 Portable Photosynthesis System

for Bluestem OS™ version 1.4

LI-COR Biosciences

4647 Superior Street
Lincoln, Nebraska 68504
Phone: +1-402-467-3576
Toll free: 800-447-3576 (U.S. and Canada)
envsales@licor.com

Regional Offices

LI-COR Biosciences GmbH

Siemensstraße 25A
61352 Bad Homburg
Germany
Phone: +49 (0) 6172 17 17 771
envsales-gmbh@licor.com

LI-COR Biosciences UK Ltd.

St. John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
United Kingdom
Phone: +44 (0) 1223 422102
envsales-UK@licor.com

LI-COR Distributor Network:

www.licor.com/env/distributors

LI-COR®

Notice

The information in this document is subject to change without notice.

LI-COR MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. LI-COR shall not be held liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without prior written consent of LI-COR, Inc.

Microsoft, Windows, and Excel are registered trademarks of the Microsoft Corporation. Drierite is a registered trademark of the W.A. Hammond Drierite Company. Sorbead and Orange CHAMELEON are registered trademarks of BASF CATALYSTS LLC. Fetch is a registered trademark of Fetch Softworks LLC. Excelon Bev-a-line is a registered trademark of Thermoplastic Processes, Inc. Propafilm is a trademark of Innovia Films. All other trademarks and registered trademarks are property of their respective owners.

The LI-6800 is protected by patents 8,610,072, 8,692,202, and 8,910,506. Additional patents pending in the U.S. and other countries.

Printing History





Copyright © 2019, LI-COR, Inc. All Rights Reserved

Publication Number: 977-18536

Created on: Tuesday, January 28, 2020.

Notes on Safety

This LI-COR product has been designed to be safe when operated in the manner described in this manual. The safety of this product cannot be assured if the product is used in any other way than is specified in this manual. The product is intended to be used by qualified personnel. Read this entire manual before using the product.

Equipment markings:	
	The product is marked with this symbol when it is necessary for you to refer to the manual or accompanying documents in order to protect against injury or damage to the product.
	The product is marked with this symbol when a hazardous voltage may be present.
	The product is marked with this symbol if a Chassis Ground connection is required.
	The product is marked with this symbol to indicate that a direct current (DC) power supply is required.
WARNING	Warnings must be followed carefully to avoid bodily injury.
CAUTION	Cautions must be observed to avoid damage to your equipment.
Manual markings:	
Warning	Warnings must be followed carefully to avoid bodily injury.
Caution	Cautions must be observed to avoid damage to your equipment.
Note	Notes contain important information and useful tips on the operation of your equipment.

CE Marking:

This product is a CE-marked product. For conformity information, contact LI-COR Support at envsupport@licor.com. Outside of the U.S., contact your local sales office or distributor.

California Proposition 65 Warning

WARNING: This product contains chemicals known to the State of California to cause cancer and birth defects or other reproductive harm.

Federal Communications Commission Radio Interference Statement

WARNING: This equipment generates, uses, and can radiate radio frequency energy and if not installed in accordance with the instruction manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide a reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Waste Electronic and Electrical Equipment (WEEE) Notice

This symbol indicates that the product is to be collected separately from unsorted municipal waste. The following applies to users in European countries: This product is designated for separate collection at an appropriate collection point. Do not dispose of as household waste. For more information, contact your local distributor or the local authorities in charge of waste management.



Contents

Section 1. Things you should know

Nomenclature and symbols	1-1
Python	1-2
If you don't know Python	1-2
Pay attention to the hints	1-3
If you do know Python	1-6
Consider using VNC	1-8

Section 2. Overview

Work flows	2-3
A tour	2-4
Start an existing BP	2-4
Build a new BP	2-6

Section 3. Some examples

Early morning FoFm	3-1
Variations on AutoLog	3-5
Timing	3-5
Log until sundown	3-7
Varying the log interval	3-10
Monitoring match mode	3-18
Record and replay a time series	3-20

Section 4. Response curves

Basic	4-1
Multiple controls	4-5
Higher dimensions	4-8
Variable stability wait times	4-14

Section 5. Using dialogs

Section 6. Screen reference

The Open/New screen	6-1
The Build screen	6-2
The Set screen	6-5
The Start screen	6-6
The Monitor screen	6-8
Set screen interface tools	6-9
Simple objects	6-9
Control table	6-11
Data dictionary	6-12
Control dictionary	6-13
Status dictionary	6-13
Debug mode	6-14
Directory information	6-15
Saving BPs	6-17

Section 7. Step reference

# - comment	7-1
ASSIGN	7-1
Value or expression	7-1
Data Dictionary value	7-3
Status Dictionary value	7-4
(Advanced) Topic and Key	7-5
(Advanced) XML value	7-9
AUTOENV	7-10
BREAK	7-13
CALL and DEFINE	7-13
Variable scope	7-13
Passing by value or reference	7-13
DIALOG	7-15
Grid items	7-16
_dlg variables	7-20
Things to consider	7-21
EXEC	7-22
Local vs global	7-22
GROUP	7-23
IF, ELSE IF, ELSE	7-24
LOG	7-25
Open file	7-25
Record remark	7-26
Record data	7-26
Close file	7-26

LOOP	7-27
Count	7-27
Duration	7-28
List	7-29
File	7-29
PROPERTIES	7-31
RETURN	7-31
RUN	7-31
SETCONTROL	7-32
SHOW	7-33
TABLE	7-34
Structure of a TABLE variable	7-35
Custom executions	7-37
WAIT	7-37
Duration	7-38
Stability	7-38
Until	7-39
Event	7-41
WHILE	7-42

Appendix A. Control dictionary map

Appendix B. Status dictionary map

Appendix C. The list_utility module

Section 1.

Things you should know

A Background Program (BP) is a collection of steps that will execute on the LI-6800 console “in the background” to accomplish various tasks. These steps can be put together with tools provided on the console’s user interface.

While BPs can do the same things as conventional AutoPrograms, they have three significant advantages: 1) you can make your own BPs, 2) the scope of what can be done via a BP exceeds what can be done with an AutoProgram, 3) any number of BPs can be run simultaneously.

Some background information will help you be successful with background programs.

Nomenclature and symbols

We use some terminology and some highlights in this document that are explained here.

- AutoProgram - The traditional LI-6800 method of running automated logging, etc.
- BP - Background Program. The subject of this document
- **Open**, **New**, **Start**, etc. - Refers to a button on the interface.
- **Open/New**, **Build**, **Set**, **Monitor** - Refers to a tab label, or the screen associated with that tab label.
- LOOP, SETCONTROL, WAIT - A BP step.
- LOOP [Duration] - A BP step of a particular sub-type.

Python

Even through BPs are Python files, you do not need to be a Python programmer to build or run them, since the LI-6800 software provides a graphical user interface for doing those tasks. However, knowing a little bit of Python syntax can be very helpful if you wish to design your own BPs and take advantage of some powerful options that are available. The BP interface contains places where you are invited to type in values, expressions, variable names, etc., and even if you have no inclination to become a programmer, some simple knowledge can be very useful to you. For example, rather than entering in a list of setpoints for light intensity, you might instead type in something like `randomList(50,2000,15)`, which, when the program runs, will generate a randomized list of 15 linearly spaced set points ranging between 50 and 2000.

If you are not a programmer, the next section is for you.

If you don't know Python

The Python world consists of objects of various types, such as numbers, strings, lists, and so on. You create and keep track of your objects by assigning them to variables. Consider these simple examples of Python assignments:

- `a = 10` *a* points to an integer.
- `ByeBye = 'this is a string'` *ByeBye* points to a string.
- `c66 = ["hello", 55, ByeBye]` *c66* points to a list containing a string, number, and string.
- `dd = True` *dd* points to a boolean.
- `e_xy = ByeBye+" and " + c66[0]` *e_xy* will be the string 'this is a string and hello'.
- `fff = 'file '+str(a)` *fff* makes a string out of a number, so is = 'file 10'.
- `hh = a >15` and `dd` *hh* will be False, since *a* is not greater than 15.

Variable names should start with a letter (a-z or A-Z), and can contain numbers and underscores (`_`). Python is case sensitive. There are some keywords you should avoid when creating variable names (*Figure 1-1* on the facing page), but if you accidentally use one, you will be notified when you run the program and you can fix it.

The same name rule also applies to functions. A **function** is a named collection of instructions that can be invoked by using the name followed by a list of zero or more **arguments** (information passed to the function). Examples:

- `doSomething(a, 5, c66)` Calls the function *doSomething*, sending to it three arguments.
- `tryThis()` Runs the function *tryThis* which takes no arguments.
- `fff = 'file '+str(a)` The function *str()* makes a string out of a number, so *fff* becomes 'file 10' since *a* is 10.

Some functions only exist for a particular type of object. For those, we use a dot: *object.function()*. For example, strings have a *replace(x, y)* function that replaces all occurrences of *x* with *y* in that string. So, if the string pointed at by *fff* is 'file 10', then

`gg = fff.replace('file', 'cabinet')` *gg* becomes 'cabinet 10'

Keywords in Python programming language

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Figure 1-1. Avoid using these as variable names.

Pay attention to the hints

The interface for configuring steps in a BP frequently contain hints that tell you what is expected to configure that step. For example, the configuration interface for the BP step **ASSIGN**, which lets you create a variable and assign it to an expression (or other things), is shown in *Figure 1-2* on the next page.

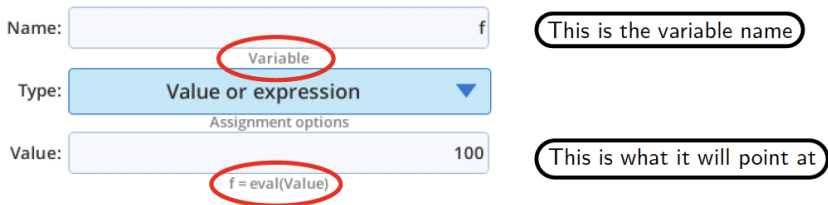


Figure 1-2. An example of an edit box requiring a variable name.

When an edit box has a hint with 'eval()' in it, it means that entry is going to be passed to Python's `eval()` function when the program runs. This allows your entry to be more complicated than, say, a numeric value. In the example above, the variable is going to be assigned to whatever the `eval()` returns, regardless of type. In other settings (Figure 1-3 below), a specific type is required, and that is reflected in the hint. In the first example below, the entry is 10. It could just as easily be an expression, like $8+2$, or (if you have defined variables x and y) $(x+y)/3.14159$, or anything that evaluates to something numeric.

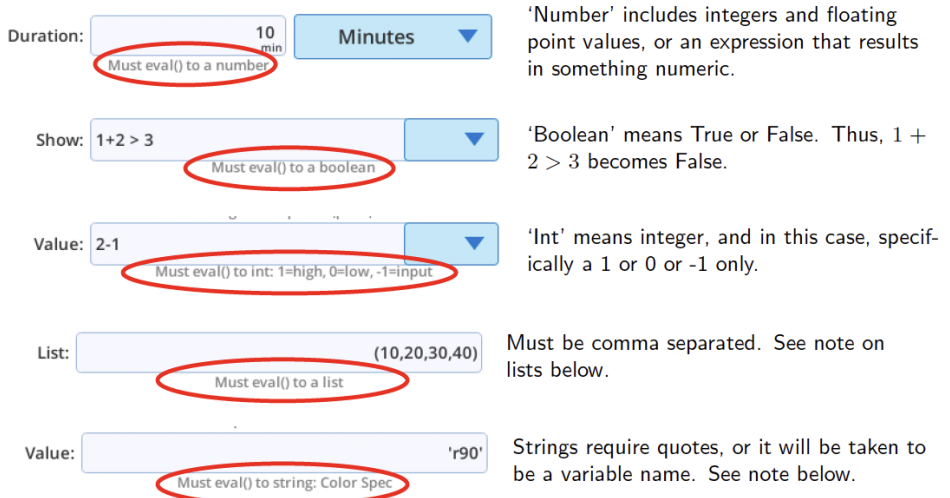


Figure 1-3. Examples of edit boxes with `eval()` in the hint.

Lists: "Must eval() to a list" means to enter a sequence that is comma separated. You can use `[]`, `()`, or leave them off:

```
[1,2,3] or (1,2,3) or 1,2,3
```

In Python, `[1, 2, 3]` is a list that can be modified; otherwise, it is a tuple, a list that cannot be modified. For lists you enter in a BP interface, that distinction is usually not important. To enter a single valued list, use a trailing comma.

```
[1,] or (1,) or 1,
```

Trailing commas like this `(1, 2, 3,)` are always allowed, regardless of the size of the list.

Strings: Take special note of the string example in *Figure 1-3* on the previous page. If you are being asked for a string, such as a color, or a file name, that will be passed to `eval()`, and you enter

```
/home/licor/myfile.txt
```

`eval()` will treat that entry like a variable name and you'll see an error. To avoid that, put single or double quotes around it.

```
'/home/licor/myfile.txt' or "/home/licor/myfile.txt"
```

The reason for this is to allow for variable names. For example, you might wish to programmatically name a new file each time through a loop, so you enter

```
'/home/licor/myfile'+str(count)+'.txt'
```

where *count* is some variable you have defined, giving you a new file each time through:

```
/home/licor/myfile0.txt  
/home/licor/myfile1.txt  
/home/licor/myfile2.txt
```

Failure to follow the guidance provided in the hint will usually result in an error (*Figure 1-4* on the next page).

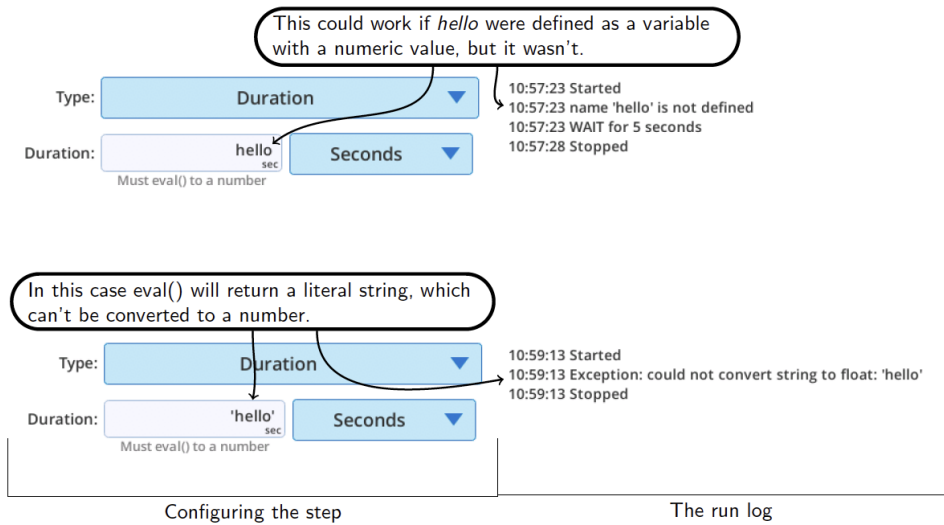


Figure 1-4. This is a one step BP, consisting of a WAIT step. Bad entries (left) and the resulting output in the BP's run log (text at right).

If you do know Python

BP files are Python files (.py). A simple one from one of the upcoming examples is shown in *Listing 1-1* on the facing page.

```

from bpdefs import ASSIGN, LOOP, SETCONTROL, LOG, Nothing, CheckBox,
Text, Button, DropDown, RadioBtns, EditText

steps=[
# Assign a variable to an expression: ASSIGN('varname',
exp="expression" [,dlg=Nothing()])
ASSIGN("logint",
      exp="lambda x: 30/(1+50*math.exp(-0.03*x))+0"),
# Assign a variable to an expression: ASSIGN('varname',
exp="expression" [,dlg=Nothing()])
ASSIGN("test",
      exp="lambda x: x if x >= 1 else 0"),
# Loop through a list: LOOP(list=itemList [,var=varname]
[,mininc=''])
LOOP(list="1500,50,1500",
      var="x",
      steps=(
          # Set a control:

```

```

SETCONTROL('target', 'value', 'eval' [,opt_target=''])
    SETCONTROL("Qin","x","float"),
    # Loop for a duration: LOOP(dur="float"
[,units='Seconds' Second|Minutes|Hours ] [,var='') [,mininc=''])
    LOOP(dur="15",
        units="Minutes",
        var="t",
        mininc="test(logint(t))",
        steps=(
            # Log a data record:
LOG([avg='Default'] [,match='Default'] [,flr='Default']
[flash='Default'])
            LOG(avg="Off",
                match="Off",
                flr="0: Nothing"),
        )
    ),
)
]

```

Listing 1-1. Listing of the file /home/licor/apps/examples/VariableLogInt.py.

A BP file contains one list, which is always named `steps`. Each item in `steps` is created using a constructor, with class names such as `ASSIGN`, `LOOP`, `LOG`, etc. The line before each constructor contains a comment, showing the options available for the version of constructor used.

The first thing to know is this: running a BP is not simply a matter of running the `.py` file; rather, the `.py` file is compiled (`eval()`) to obtain a list of program steps (`steps`). The thread that actually executes when a BP uses `steps` as data as it constructs itself.

A second thing to note is that because we are just constructing data in a list, variables and expressions have to exist at this stage as strings, sometimes even doubly-quoted strings. The first parameter in `ASSIGN`, for example, is destined to become a Python variable (but not yet), so at this stage, it is a quoted string. Similarly, what will be the object assigned to that variable also starts out as a string. In the two `ASSIGN`s above, the assigned objects will become lambda expressions. In summary, you have to keep in mind that you are writing code that will be sent to a Python's `eval()` or `exec()` statements twice: once to build the step, then later when the step processes its parameters.

Because BP files are .py files, they can be written and/or edited outside of the LI-6800 console's user interface using any text editor. Ideally, of course, you should use an editor designed for Python, to eliminate syntax errors in the editing stage.

Consider using VNC

The touch screen keyboard on the LI-6800 console does not lend itself well to typing long expressions and names that may be needed when developing BPs.

Consider connecting to the LI-6800 with a virtual network computing (VNC) viewer client when doing BP development. Clients are available for a wide range of platforms from <https://www.realvnc.com/en/connect/download/viewer/>

Using a VNC client on your computer allows you to use your computer's keyboard to enter those long expressions and names.

One hint for doing this is in *Figure 1-5* below. When the LI-6800 keyboard (or keypad) appears for an entry, the VNC client keyboard will not immediately be “activated”. To do this, you must click in the keypad entry area *somewhere to the left* of the entry's last character. (If the entry area is empty, you may have first click on a couple of keypad buttons to put some characters there, then click to the left of the last one.)

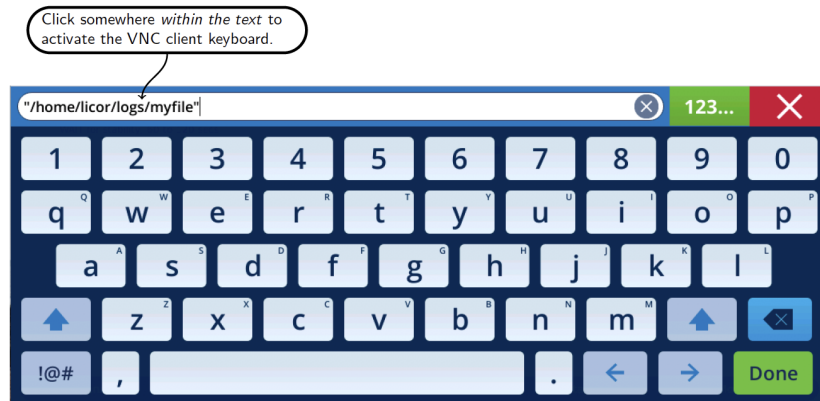


Figure 1-5. Click in the text to activate the VNC client keyboard.

Section 2.

Overview

As mentioned, a Background Program (BP) is a collection of steps that will execute on the LI-6800 console "in the background" to accomplish various tasks. These steps can be put together by the user with tools provided on the console's user interface.

Figure 2-1 below illustrates how to access the BP screens.

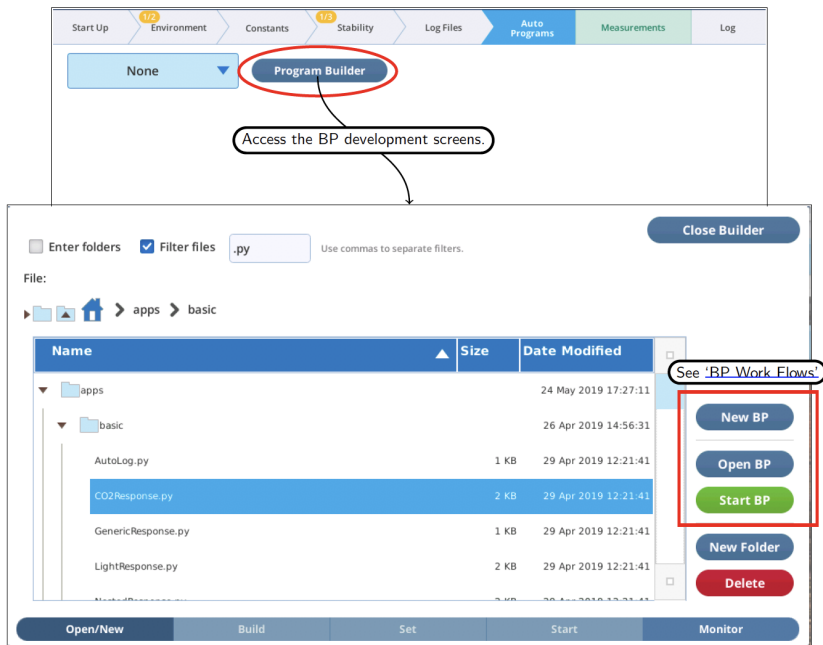


Figure 2-1. The **Open/New** tab of the Background Program interface.

The BP environment has five tabs along the bottom: **Open/New**, **Build**, **Set**, **Start**, and **Monitor**.

Open/New - Load BPs from the file system for running or editing.

Build - Assemble a list of program steps fom the menu on the left. Steps can be re-ordered, moved around, etc.

Set - Each program step has associated parameters that can be edited and adjusted as needed.

Start - At any point, you can try out all or part of your program. This screen shows what step is being executed, and shows messages and output generated by the BP as it runs. You can execute your program in real time or step by step.

Monitor - Similar to the Start screen, but for any running BP.

Work flows

Open/New

See *The Open/New screen* on page 6-1

Build

See *The Build screen* on page 6-2

Set

See *The Set screen* on page 6-5

Start

See *The Start screen* on page 6-6

Monitor

See *The Monitor screen* on page 6-8

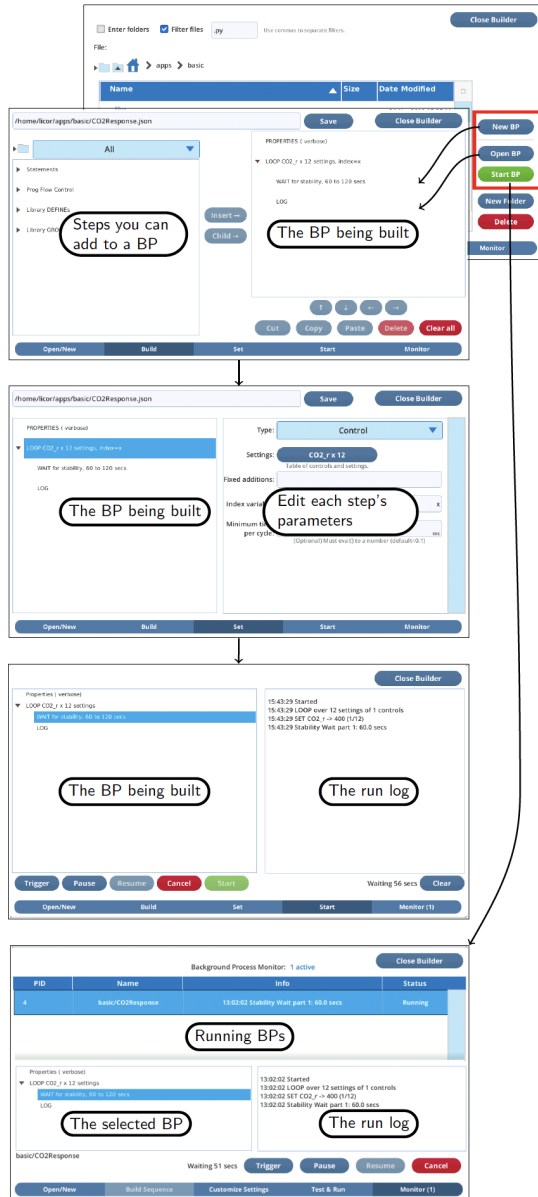


Figure 2-2. The BP workflows.

A tour

For this introductory tour, we will launch an existing BP and watch it a bit, then - while the first BP continues to run - we will build a second BP from scratch, and run it.

Start an existing BP

Tap **Program Builder** in the AutoProgram page (top of *Figure 2-1* on page 2-1). This will bring up the BP screens. Follow the steps in *Figure 2-4* on the facing page. (You don't need to have a log file open for this demo.)

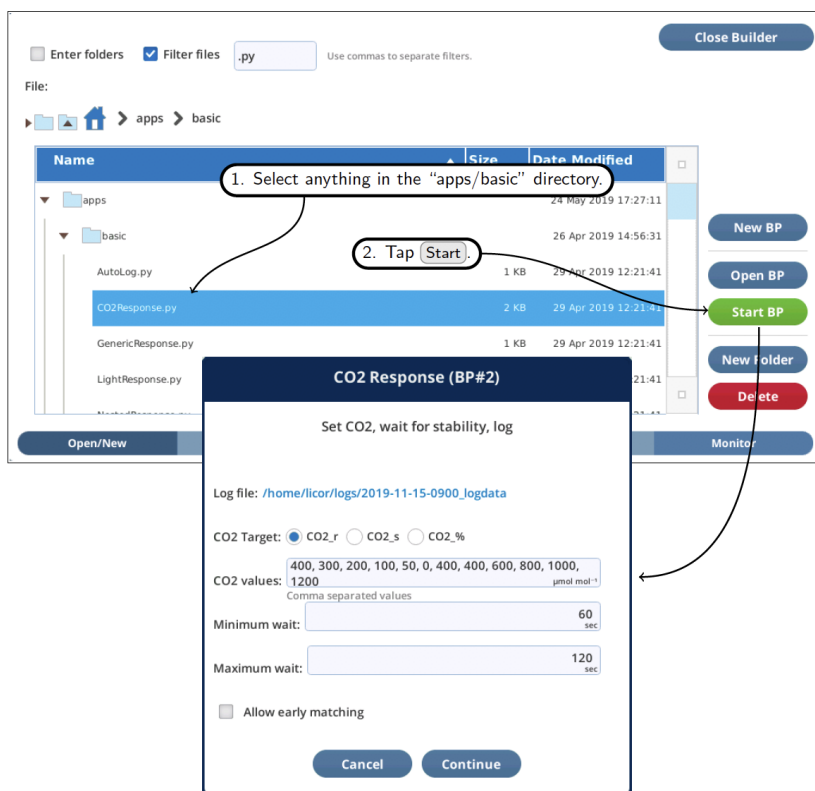


Figure 2-3. Launching the CO₂ response BP, which has an opening dialog.

The BPs in the apps/basic directory all have opening dialogs, allowing settings to be modified from the default. Tap **Continue** and you can then monitor the progress of the BP in the **Monitor** tab (*Figure 2-4* below).

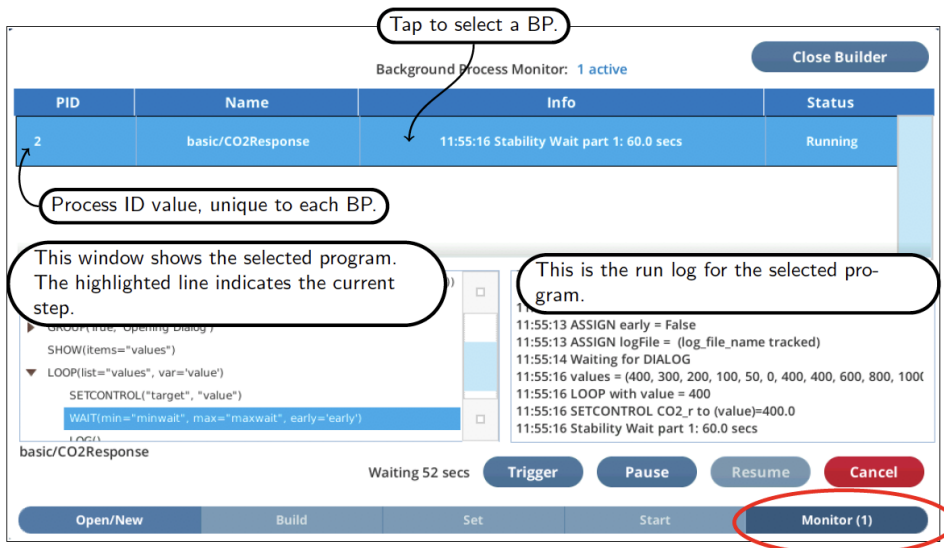


Figure 2-4. Launching and monitoring a BP.

Use **Trigger** to skip waits, try out **Pause** and **Resume**. Don't tap **Cancel** - we'll keep this BP running while we continue the tour, building a simple BP from scratch.

Build a new BP

Let's make a simple program to do the following: 1) Turn on the chamber fan, set it to 12,000 rpm. 2) Wait 30 seconds. 3) Turn the fan off.

To do this, following the steps:

- 1 Create an empty program.

Select the **Open/New** tab (lower left) and tap the **New** button (Figure 2-5 below).

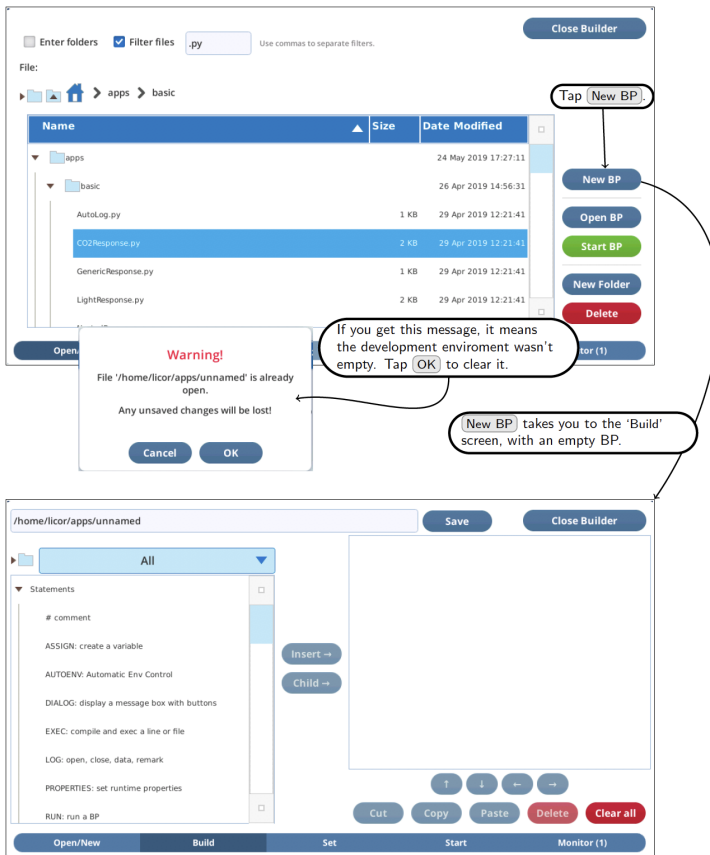


Figure 2-5. Preparing the build a BP from scratch.

- 2 Add four steps (PROPERTIES, SETCONTROL, WAIT, and SETCONTROL) to the program. To add a step, select it in the left-hand window, then tap **Insert** → to put a copy of it on the right. To arrange steps in the right window, use **↓** and **↑** at the bottom of the right view.

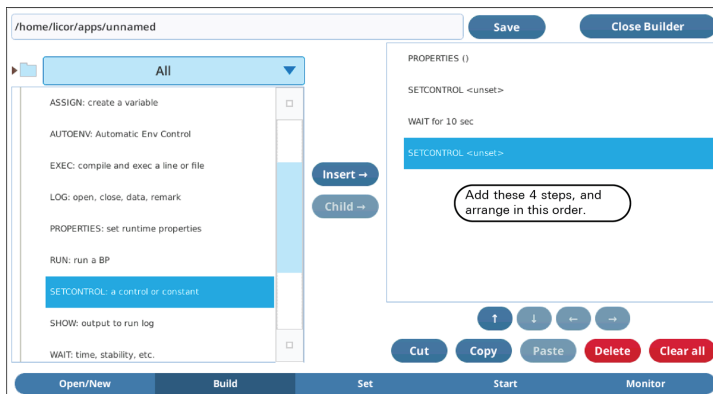


Figure 2-6. Building a simple program.

- 3 Now tap **Set**. This is the screen in which we configure the steps to do what we need. Our 4-step BP is on the left; tap a step to select it. On the right is the configuration interface for the selected BP step.

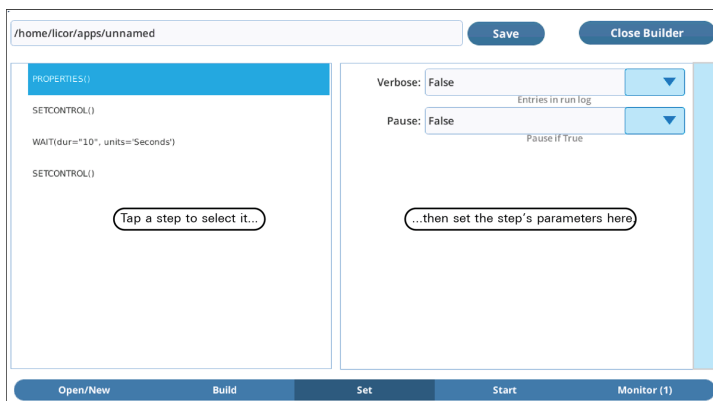


Figure 2-7. Setting the parameters for each step in the BP.

- Select each BP step, and configure it as shown in *Figure 2-8* below.
As you make edits on the right, the step summary on the left will change to reflect the changes.

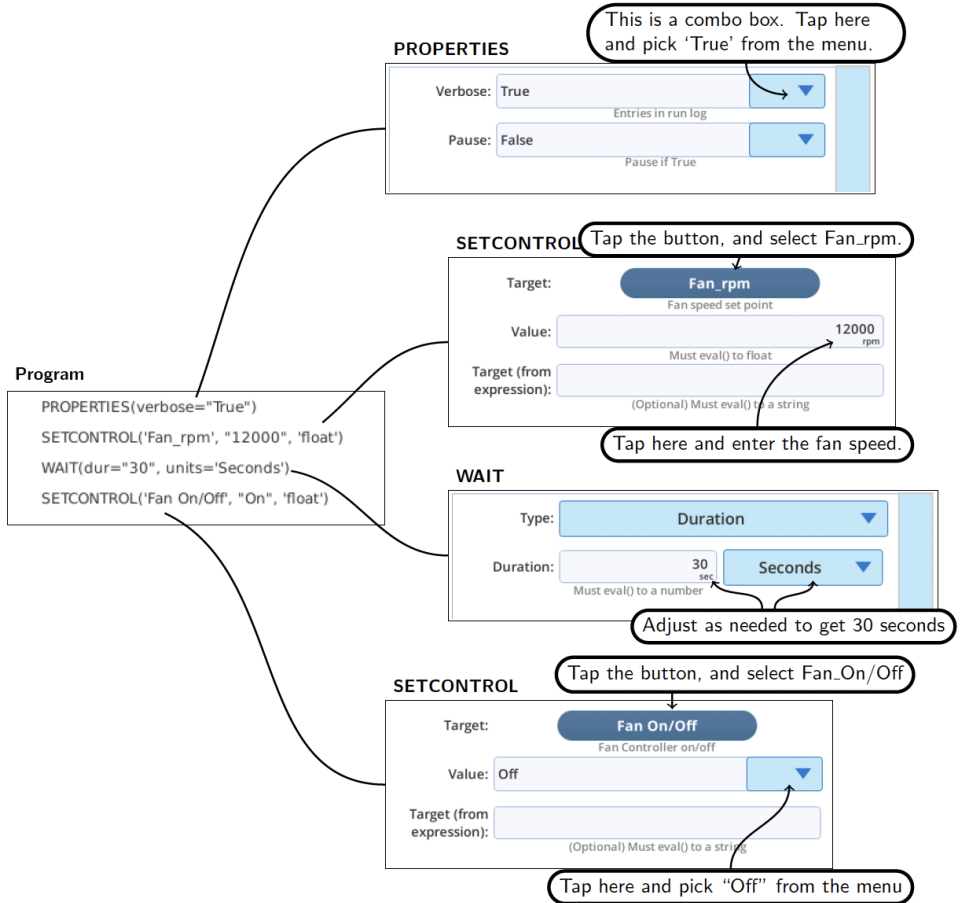


Figure 2-8. Setting the parameters for each step in the BP.

The **PROPERTIES** step is there so we can turn on “verbosity”, which means every step will produce some output in the run log when we run it.

- 5 Now tap **Start** and we will try out the program.

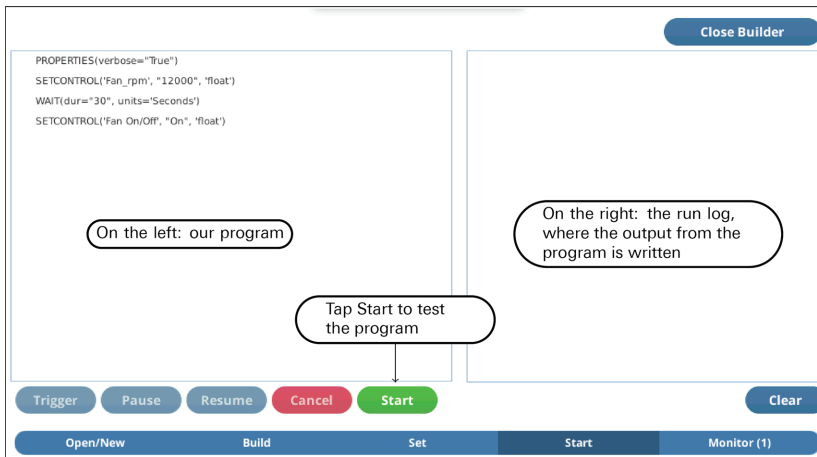


Figure 2-9. The **Start** screen allows you to test your program.

- 6 When you start running the program (tap **Start**), you should see things start to appear in the right-hand run log (Figure 2-10 below).

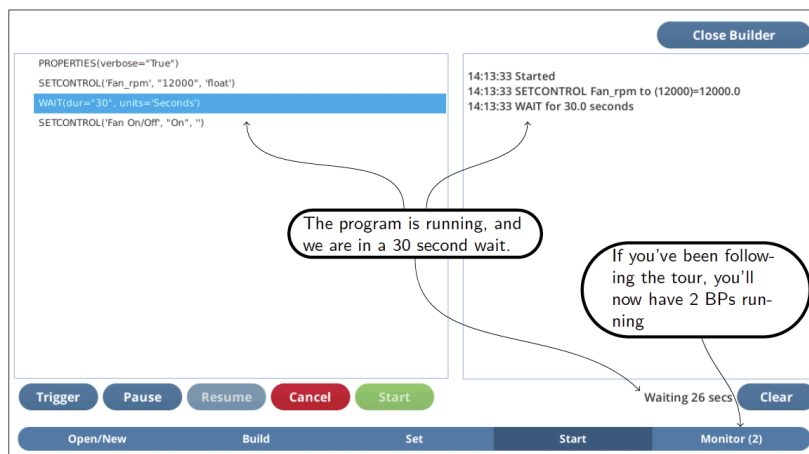


Figure 2-10. The step currently being executed in the program is highlighted in the left list.

7 Tap Monitor.

In our tour we should now have two BPs running (at least for the 30 seconds the latest one has the fan on), so we can view both in the Monitor (*Figure 2-11* below).

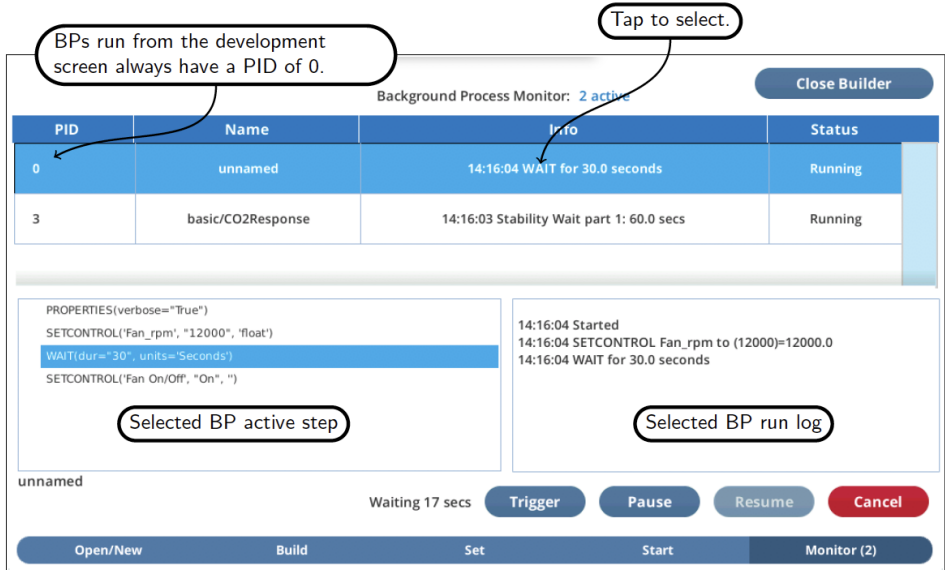


Figure 2-11. Multiple BPs running concurrently.

When a process ends (as our PID=0 will after 30 seconds), it will disappear from the monitor screen. If it was the selected BP when it ended, its run log will remain visible until you selected a different process to view.

- 8 Now tap the remaining process that we launched at the start of the tour, and see in the run log what it has been doing (Figure 2-12 below).

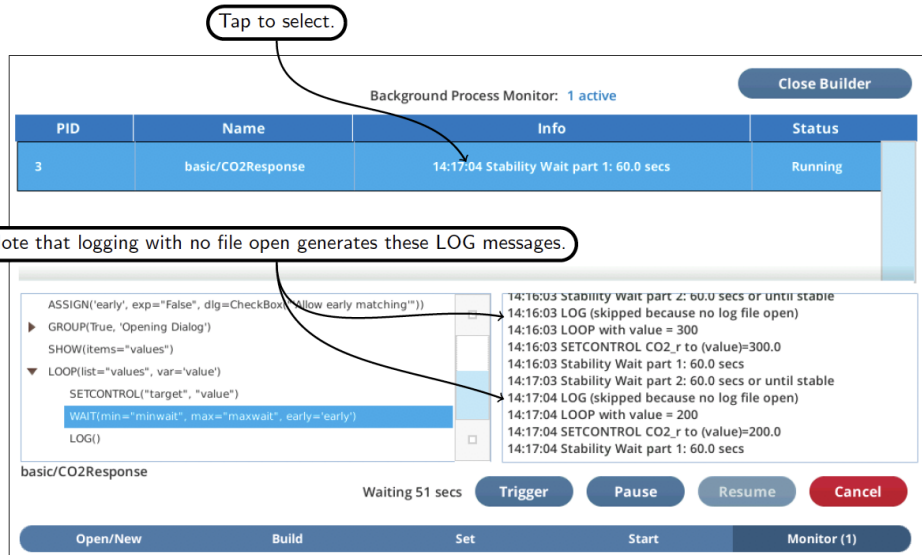


Figure 2-12. The program we first launched has been busy while we were the second program.

- 9 Tap **Cancel** to end that program, and our tour.

Section 3.

Some examples

In this section, we present some examples that show how to carry out a measurement with a background program.

Early morning FoFm

Suppose you need to make a dark adapted fluorescence reading on a leaf the first thing in the morning, but don't want to be there to do it. What you would rather do is start a BP the day before and put the instrument to sleep, leaving this BP running to do what needs to be done in the morning.

(A finished version of this program can be found in `/home/licor/apps/examples/EarlyMorningFo.py`). Specifically, we need the program to perform the following steps (each is followed by the actual BP step):

- 1** Wait until 5 a.m. (**WAIT**).
- 2** Get the instrument out of sleep mode (**SETCONTROL**).
- 3** Wait a few minutes to let things warm up (**WAIT**).
- 4** Open a log file (**LOG**).
- 5** Log, getting an FoFm (**LOG**).
- 6** Close the file (**LOG**).
- 7** Go back to sleep (**SETCONTROL**).

To assemble this BP, do the following:

- 1 Tap the **Open/New** tab, then **New**.
This will take you to the **Build** tab, with an empty list.
- 2 Insert a **PROPERTIES**.
Tap on **PROPERTIES** in the Steps section on the left, then tap **Insert→**.
- 3 Insert a **WAIT** below it (with **PROPERTIES** highlighted on the right, and **WAIT** highlighted on the left, tap **Insert→**).
This will become the item that waits until dawn.
- 4 Continue, until the list looks as shown in *Figure 3-1* below.

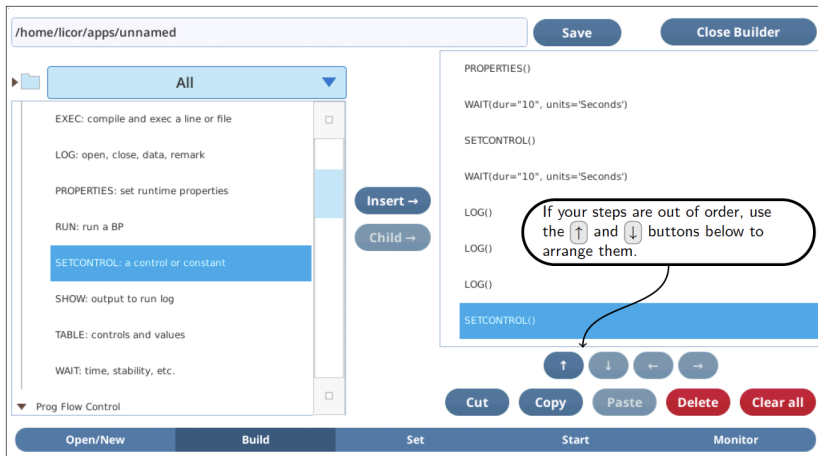


Figure 3-1. The program steps necessary - and not yet configured - for the early morning FoFm example.

- 5 Tap on the **Set** tab, and configure each step according to *Figure 3-2* on the facing page.

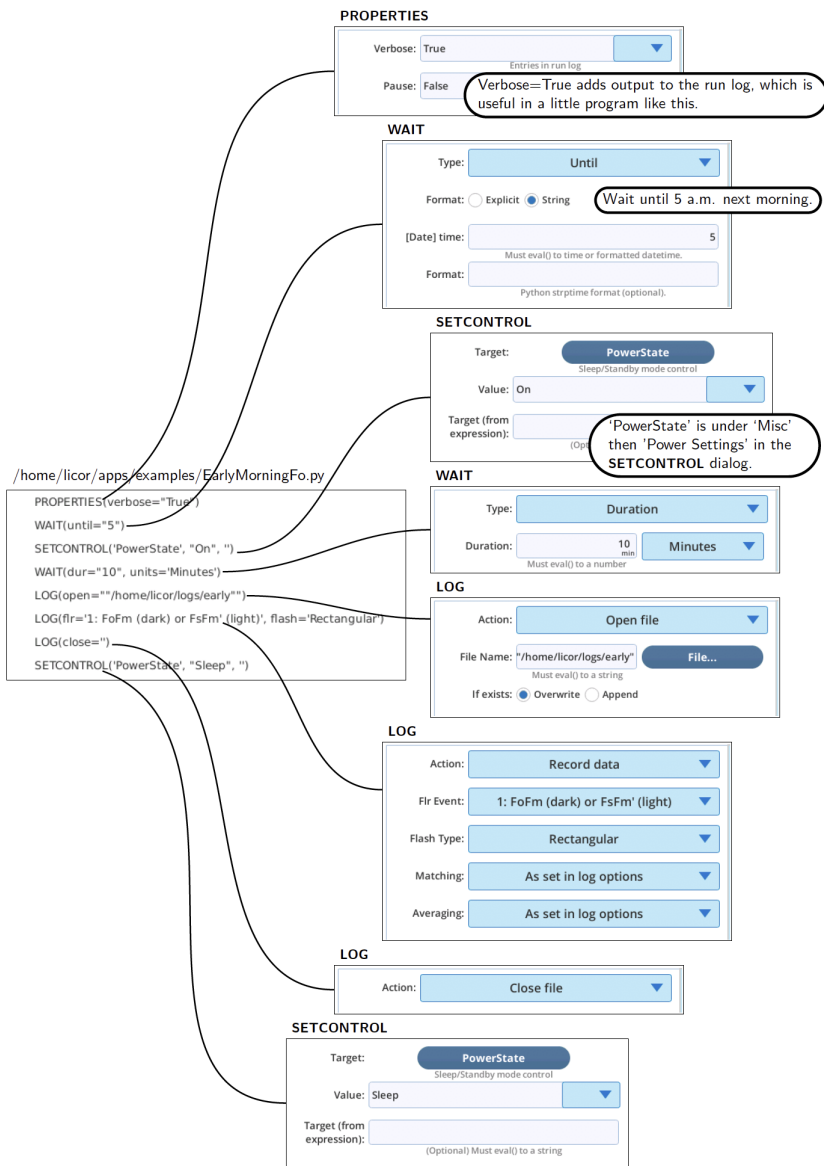


Figure 3-2. How each step is to be configured for the early morning FoFm example.

Note that we are doing three different things with LOG: open, write to, and close a file.

Test the program by tapping the **Start** tab, and then tap the **Start** button. Immediately we are waiting until 5:00 the next morning. Tap **Trigger** if you don't want to wait that long.

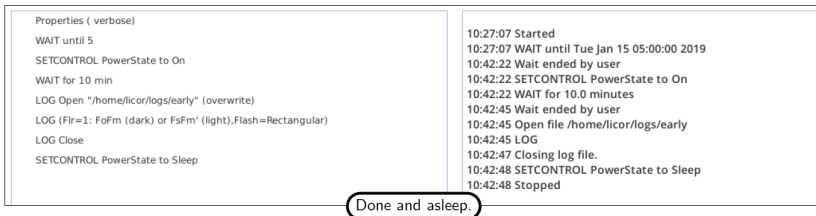
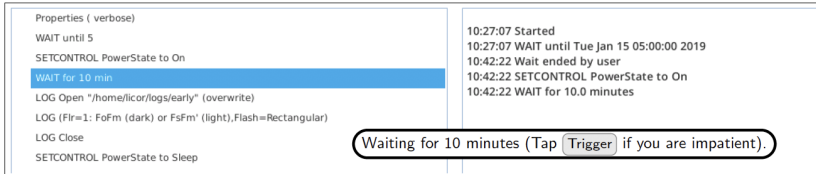
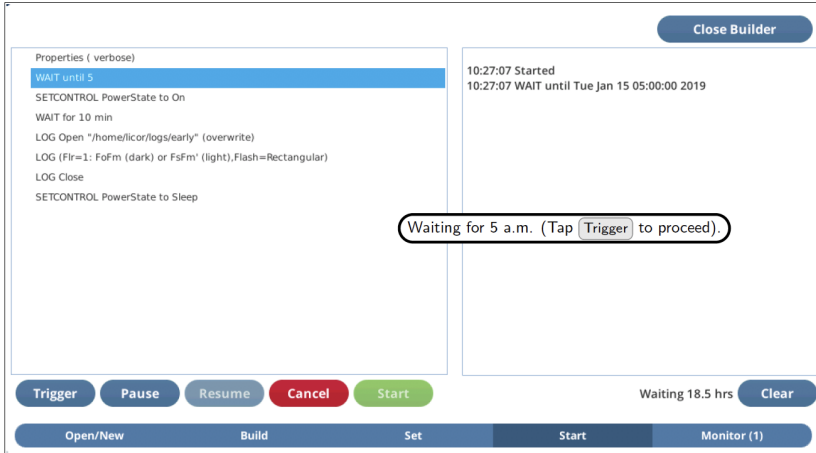


Figure 3-3. Test running the program.

Variations on AutoLog

Autologging (logging at regular intervals for some fixed amount of time) is very simple with a BP, but there are some useful variations on this theme that are good to know, including

- Log events can take variable amounts of time (matching, fluorometry, etc.), so how can we get precise log intervals?
- What if we want the log interval to change with time?
- What if we want to start or stop based on some environmental condition, rather than just the clock?

Timing

Let's start with a basic autologging program (*Figure 3-4* below) which does a LOOP lasting 10 minutes (autolog duration) that contains a LOG and a WAIT (the logging interval).

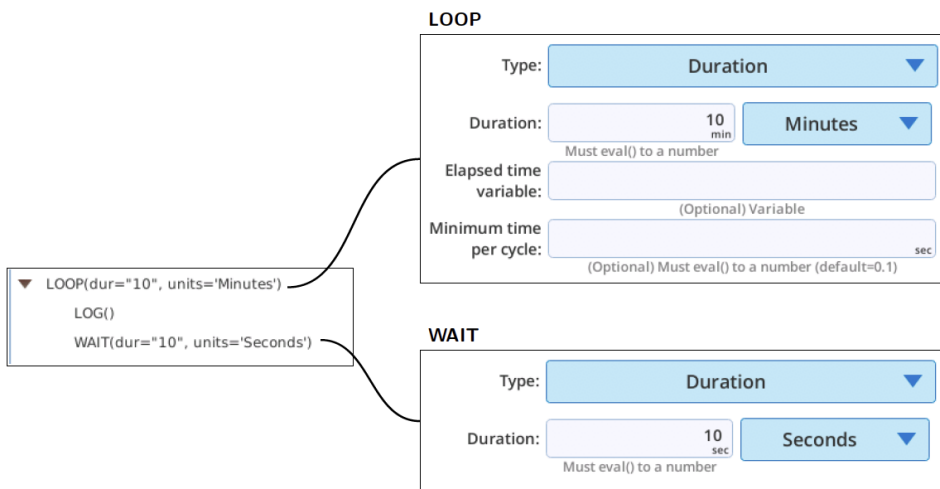


Figure 3-4. A simple autolog program.

A potential shortcoming of this program is that **it does not account for how long the LOG takes**, which might range from about half a second to many seconds, depending on **matching, fluorometer actions**, etc.. If you were trying to log every 1 or 2 seconds, then this program would be a little frustrating because the log interval might often be longer than that. We could explicitly time the log in our

program and adjust the wait time accordingly but **LOOP** provides an easier way to accomplish that (*Figure 3-5* below).

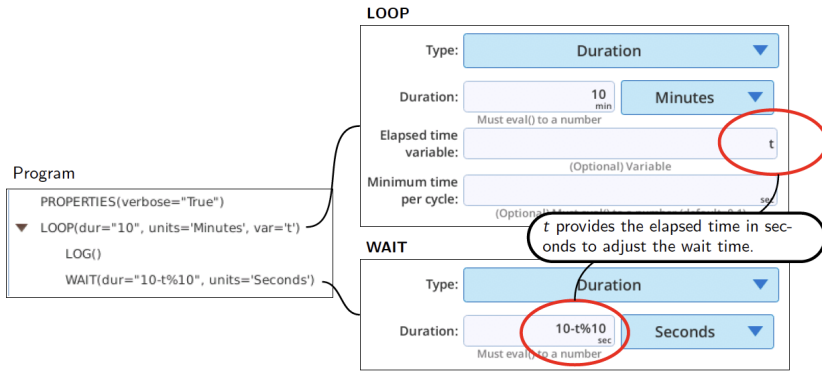


Figure 3-5. A more precise autolog using **LOOP**'s built-in timer.

Let's see this program in action. With a log file open, and a **PROPERTIES** statement added to get the actual time sent to the **WAIT** statements, we get *Figure 3-6* below.

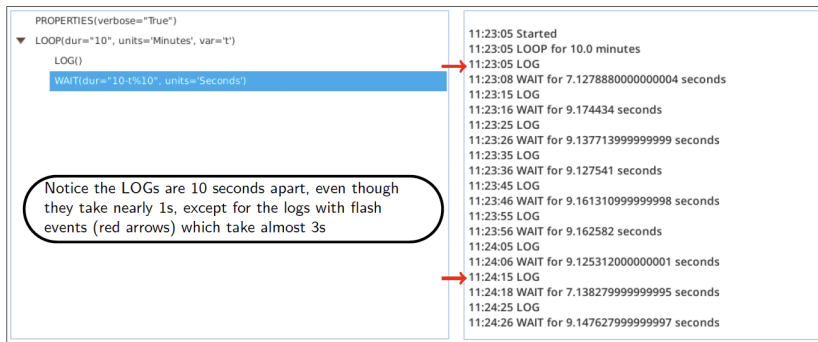


Figure 3-6. Testing the adjustable wait.

There is yet a third method, and it is even easier (*Figure 3-7* on the facing page). **LOOP** (and **WHILE**) have a minimum time per cycle parameter that can be specified to regulate how often the loop repeats. At the end of the loop, it does an implicit wait if necessary to enforce that timing so we can get rig of our **WAIT** statement.

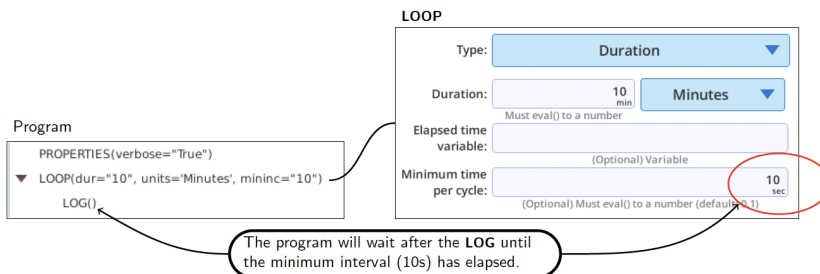


Figure 3-7. Replace the **WAIT** with the minimum interval specifier.

Another advantage of this last method is that both parameters for the autolog (duration and interval) are contained in the **LOOP** settings.

Log until sundown

Now that we know how to do an autolog program, let's try another variation, this time adjusting the duration. Instead of a fixed time duration, let's make it log until some environmental condition is met. Specifically, we'll make a program to log a data record every 10 minutes from the time we start the program until the sun goes down. We'll assume there is an external quantum sensor measuring light, and define sundown as when the PAR reading goes below $5 \mu\text{mol m}^{-2} \text{s}^{-1}$.

(A finished version of this program can be found in `/home/licor/apps/examples/LogTilDark.py`).

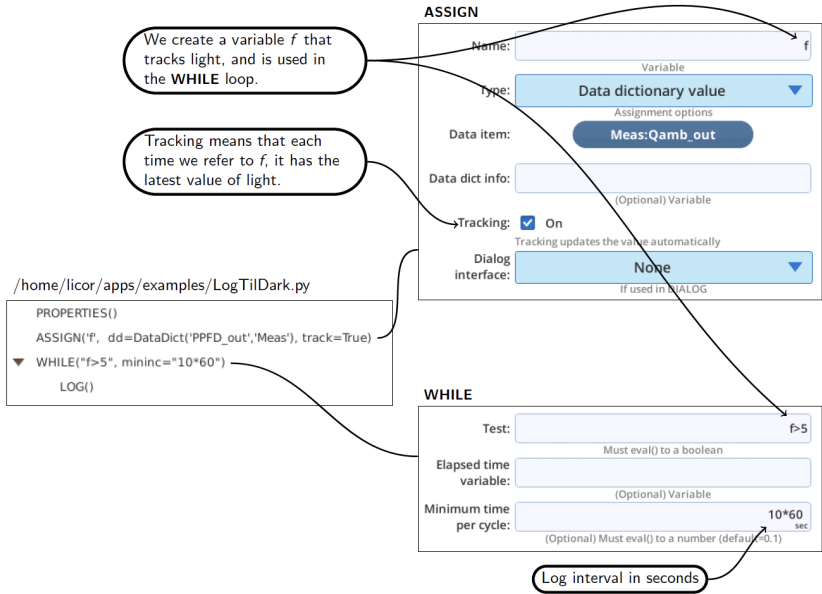


Figure 3-8. Configuring the program to log until dark.

Now let's test this program, making sure there is enough light on the external quantum sensor so it will act like daytime. To simulate sunset after a test loop or two, simply cover the sensor.

The screenshot shows the LogTilDark configuration interface. The top section is titled "Close Builder". Below it, the configuration is as follows:

```

PROPERTIES(verbose="True")
ASSIGN('f', dd=DataDict('PPFD_out','Meas'), track=True)
WHILE('f>5', mininc="10*60")
  LOG()

```

On the right, the execution log shows:

```

12:44:06 Started
12:44:06 ASSIGN f = 10.9392 (PPFD_out tracked)
12:44:06 WHILE (f>5)
12:44:06 LOG

```

Annotations in the image explain the state:

- A callout points to the `verbose="True"` property: "For testing, set 'Verbose' to True to get the output shown in this example."
- Another callout points to the `LOG()` line in the code: "The quantum sensor read 10.9, which is greater than 5, so it entered the loop, did a log, and is now waiting for 10 minutes."

At the bottom, there are control buttons: Trigger, Pause, Resume, Cancel, Start, and Clear. Below these are status buttons: Open/New, Build, Set, Start, and Monitor (1).

The second screenshot shows the program after a 10-minute wait. The configuration is the same, but the log has been updated:

```

PROPERTIES(verbose="True")
ASSIGN('f', dd=DataDict('PPFD_out','Meas'), track=True)
WHILE('f>5', mininc="10*60")
  LOG()

```

The execution log now includes:

```

12:44:06 Started
12:44:06 ASSIGN f = 10.9392 (PPFD_out tracked)
12:44:06 WHILE (f>5)
12:44:06 LOG
12:44:21 Wait ended by user
12:44:21 WHILE (f>5)
12:44:21 LOG
12:44:38 Wait ended by user
12:44:38 WHILE (f>5)
12:44:38 Stopped

```

Annotations in this screenshot explain the user interaction:

- A callout points to the `LOG()` line: "Pressed [Trigger] to end the 10 minute wait."
- Another callout points to the `Wait ended by user` log entries: "Before pressing [Trigger] again, we covered the sensor to make it dark. Since now $f < 5$, we left the WHILE loop, and the program ended."

Figure 3-9. Configuring LogTilDark.

Running this program with `Verbose=False` (in the `PROPERTIES` step) will decrease the output to just the start and stop message, and the log events in between. This is how a repetitive program like this should normally be run.

Varying the log interval

When recording the response of a leaf to an abrupt environmental change (e.g., light level), there might be multiple time scales you want to capture, from the very rapid (photosynthesis) to the very slow (stomatal conductance). While one option might be to log as fast as possible for a long time, a more efficient method is to start logging frequently and slow down as time goes by.

BPs provide a couple of ways to do that.

One method is to chain together a series of autoprogram loops, each with differing durations and wait intervals, such as log every 1 second for 1 minute, then every 5 seconds for 1 minute, then every 10 seconds for 3 minutes, then 60 seconds for 10 minutes.

Let's assemble a BP that does this, and in the process learn something about BP functions (CALL and DEFINE):

- 1 Tap the **Open/New** tab, then **New**.

This will take you to the **Build Sequence** tab, with an empty list.

- 2 Add a **CALL** (With **CALL** highlighted on the left, and tap **Insert**→.)

- 3 Add the function **DEFINE AutoLog** (With **CALL** highlighted on the right, scroll down to the **Library DEFINES** section of the box on the left, tap on **Define AutoLog**, and tap **Insert**→.)

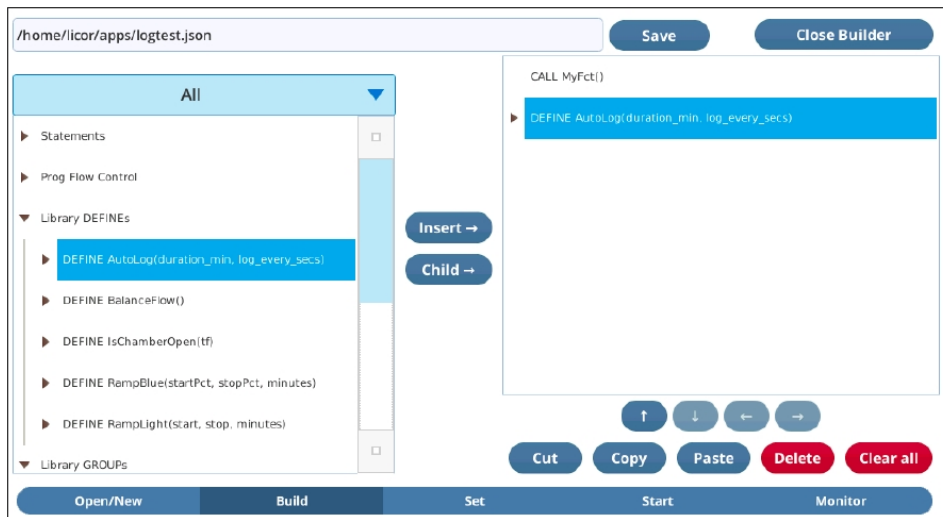


Figure 3-10. The first steps in building a four stage autolog program.

- 4 Change to the **Set** screen, and configure the **CALL** to call the correct subroutine *Figure 3-11* below.

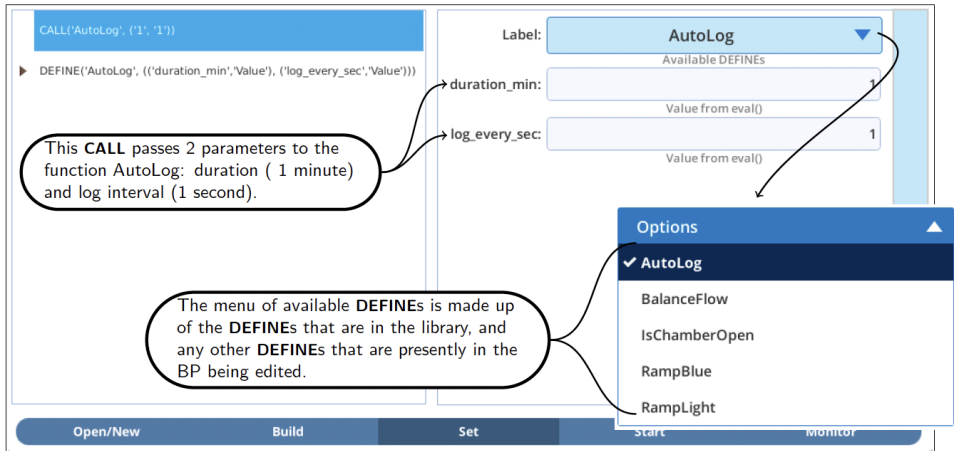
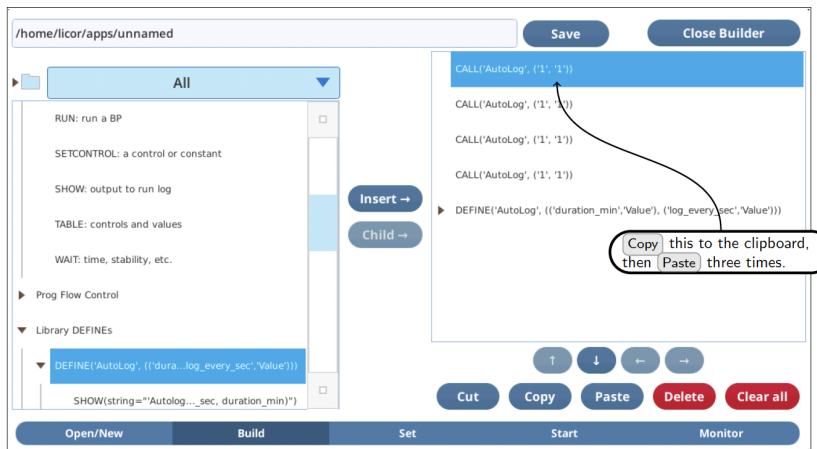


Figure 3-11. Configuring the first call to AutoLog.

- 5 Now change back to **Build**, highlight the **CALL** AutoLog on the right. Tap **Copy**, then, **Paste**, **Paste**, and **Paste**.



- 6 Now change back to **Set**, and adjust the configurations on the three new AutoLog CALLS.

```
CALL AutoLog(1, 1)
CALL AutoLog(1, 5)
CALL AutoLog(3, 10)
CALL AutoLog(10, 60)
▶ DEFINE AutoLog(duration_min, log_every_secs)
```

Figure 3-12. Adding the other CALLs.

Up to now, we haven't examined what a **DEFINE** looks like in a BP. Let's do that now, and see what is inside the AutoLog **DEFINE** (Figure 3-13 below).

The parameters for a **DEFINE** consist of a name, a parameter count, and names for each parameter. You can also specify if each parameter is passed in by value or reference (Step reference on page 7-1).

The screenshot shows a configuration window for an **AutoLog** **DEFINE**. It is divided into three main sections: **DEFINE**, **SHOW**, and **LOOP**.

- DEFINE Section:**
 - Name:** AutoLog
 - Number of Arguments:** 2
 - Argument #1:** duration_min (Variable), value
 - Argument #2:** log_every_sec (Variable), value
- SHOW Section:**
 - Type:** Formatted String (selected)
 - String:** 'Autolog: log every {0} seconds for {1} minutes'.format(log_every_sec, duration_min)
- LOOP Section:**
 - Type:** Duration
 - Duration:** duration_min (sec), Seconds
 - Elapsed Time Variable:** (Optional) Variable
 - Minimum time per cycle:** log_every_sec (sec), (Optional) Must eval() to a number (default=0.1)

Callouts provide additional context:

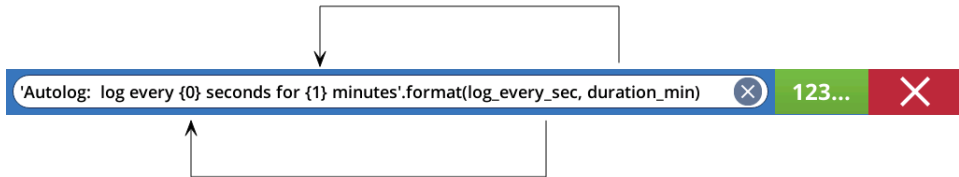
- A **DEFINE** lets you specify a name and parameter count (up to 5), with a local variable for each. AutoLog uses 2 parameters, named *duration_min* and *log_every_sec*.
- The passed-in values, contained in the variables *duration_min* and *log_every_sec*, are used in **LOOP**.
- SHOW** prints a message in the run log (Figure 32).

The background shows a code editor with the following code:

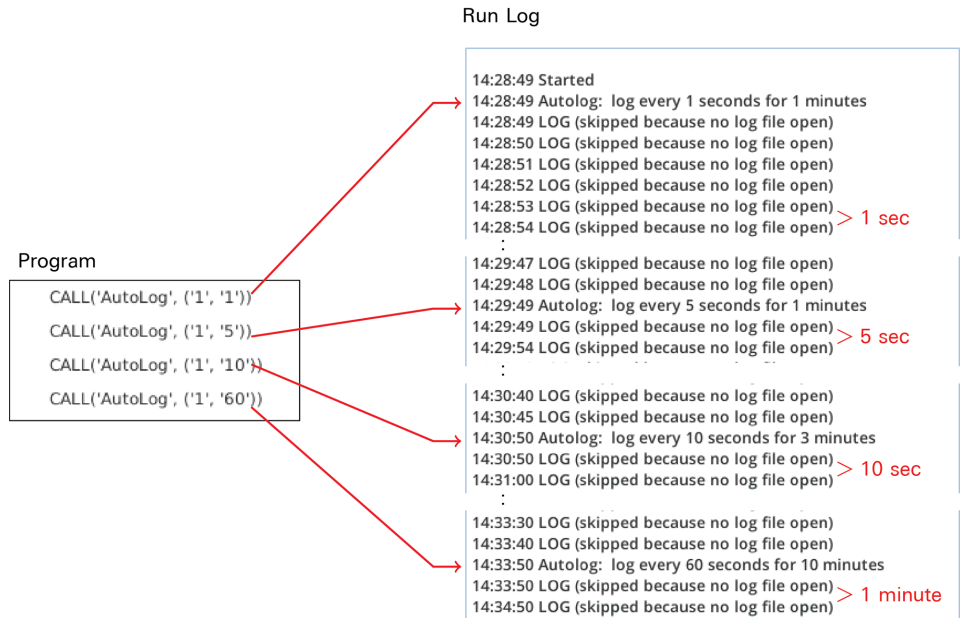
```
CALL('AutoLog', ('1', '1'))
CALL('AutoLog', ('1', '5'))
CALL('AutoLog', ('1', '10'))
CALL('AutoLog', ('1', '60'))
▼ DEFINE('AutoLog', (('duration_min', 'Value'), ('log_every_sec', 'Value')))
  SHOW(string="Autolog: log every {0}...format(log_every_sec, duration_min)")
  ▶ LOOP(dur="duration_min", units='Minutes', mininc="1")
```

Figure 3-13. How the AutoLog **DEFINE** is configured.

If you are wondering how the **SHOW** is printing the run log message you see in *Figure 3-14* below, it uses the built-in function `format()` for building a formatted string.



Because this AutoLog **DEFINE** is in the library, and we haven't changed anything about the copy we added to our BP, we don't really need it: We can delete it from our BP and the program will still work just fine.



*Figure 3-14. Run log for 4-step BP. Since the AutoLog **DEFINE** is in the library, it needn't be physically present in the BP.*

There is another approach to varying log intervals after an event: make a function that provides the log interval as a function of time after an event. Lets say we want

a logging interval that looks something like *Figure 3-15* below, which would be provided by a logistical expression such as

$$f(t) = \frac{30}{1+50e^{-0.03t}}$$

3-1

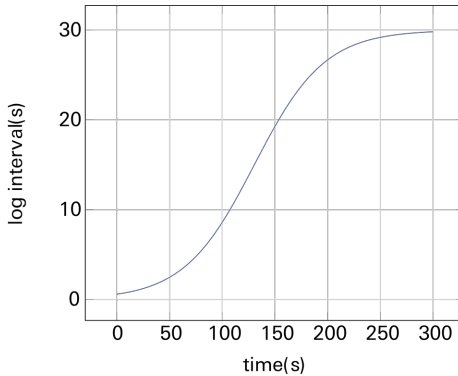


Figure 3-15. A prospective log interval function.

Implementing a function like this is very easy in a BP (*Figure 3-16* on the facing page). Creating a simple function is just like assigning a variable, but by preceding the definition with *lambda x:* (this is a Python construct), we declare *logint* to be a single parameter function, with *x* as the independent variable. The only thing in the loop is a **SHOW** statement to print out elapsed time and the computed loop trigger interval (formatted to show 3 significant digits).

Program

```

ASSIGN('logint', exp="lambda x:30/(1+50*math.exp(-0.03*x))")
LOOP(dur="15", units='Minutes', var='t', mininc="logint(t)")
SHOW(string="logint({0:1.3g})={1:1.3g}'.format(t, logint(t))")

```

ASSIGN

Name:

Variable

Type: **Value or expression**

Assignment options

Value:

logint = eval(Value)

Dialog interface: **None**

If used in DIALOG

The variable *logint* is assigned to a Python lambda expression.

We use *logint* here, and treat it like a function, *logint(t)*, where *t* is elapsed time in seconds.

LOOP

Type: **Duration**

Duration:

Must eval() to a number

Elapsed time variable:

(Optional) Variable

Minimum time per cycle:

(Optional) Must eval() to a number (default=0.1)

Figure 3-16. Program to test using a continuous function for log interval.

When run, this program produces the output as in Figure 3-17 below. The loop starts out running about every 0.5 s, and increases from there.

Program

```

ASSIGN('logint', exp="lambda x:30/(1+50*math.exp(-0.03*x))")
LOOP(dur="15", units='Minutes', var='t', mininc="logint(t)")
SHOW(string="logint({0:1.3g})={1:1.3g}'.format(t, logint(t))")

```

Run Log

```

10:32:06 Started
10:32:06 logint(0.0032)=0.588
10:32:06 logint(0.584)=0.598
10:32:07 logint(1.18)=0.609
10:32:08 logint(1.79)=0.62
10:32:08 logint(2.4)=0.631
10:32:09 logint(3.02)=0.643
10:32:10 logint(3.66)=0.655
10:32:10 logint(4.31)=0.668
10:32:11 logint(4.97)=0.681
10:32:12 logint(5.65)=0.694
10:32:12 logint(6.34)=0.709
10:32:13 logint(7.04)=0.723
10:32:14 logint(7.77)=0.739
10:32:14 logint(8.5)=0.755
10:32:15 logint(9.27)=0.772
10:32:16 logint(10)=0.79
:
10:33:08 logint(62.3)=3.44
10:33:12 logint(65.7)=3.77
10:33:15 logint(69.5)=4.16
10:33:20 logint(73.6)=4.62
10:33:24 logint(78.3)=5.19
10:33:29 logint(83.5)=5.89
10:33:35 logint(89.3)=6.78
10:33:42 logint(96.1)=7.9
10:33:50 logint(104)=9.36
:

```

Figure 3-17. Testing the timing.

At this point, we need to interject a bit of reality: the LI-6800 gets new data sets at 2 Hz, so it makes no sense trying to log any faster, and even if you could, you probably do not want to log two identical records. What we need, when we are logging rapidly, is a way to synchronize the log interval to when data is available.

Fortunately, there is a way to do that, and that is to set the 'minimum time per cycle' parameter in a LOOP to 0. Then, instead of waiting for the clock to tell it when to launch another cycle, it waits for a new data set to be ready.

Let's adapt the program to use 0 for any $\text{logint}(t)$ that computes to less than 1 second (Figure 3-7 on page 3-7).

ASSIGN

Name:	<input type="text" value="test"/>
Type:	Value or expression
Value:	$\text{lambda } x: x \text{ if } x \geq 1 \text{ else } 0$
Dialog interface:	None

Program

```

ASSIGN('logint', exp="lambda x:30/(1+50*math.exp(-0.03*x))")
ASSIGN('test', exp="lambda x: x if x >= 1 else 0")
LOOP(dur="15", units='Minutes', var='t', minInc="test(logint(t))")
SHOW(string="test(logint({0:1.3g}))={1:1.3g}'.format(t, test(logint(t)))")
        
```

Run Log

```

11:05:58 test(logint(13.5))=0
11:05:59 test(logint(14))=0
11:05:59 test(logint(14.4))=0
11:06:00 test(logint(14.9))=0
11:06:00 test(logint(15.5))=0
11:06:01 test(logint(16))=0
11:06:01 test(logint(16.4))=0
11:06:02 test(logint(17.1))=0
11:06:02 test(logint(17.4))=0
11:06:03 test(logint(18))=0
11:06:03 test(logint(18.4))=1.01
11:06:04 test(logint(19.4))=1.04
11:06:05 test(logint(20.4))=1.07
11:06:06 test(logint(21.5))=1.1
11:06:07 test(logint(22.6))=1.14
11:06:09 test(logint(23.7))=1.17
11:06:10 test(logint(24.9))=1.22
11:06:11 test(logint(26.1))=1.26
11:06:12 test(logint(27.4))=1.3
        
```

LOOP

Type:	Duration
Duration:	15 Minutes
Elapsed time variable:	t
Minimum time per cycle:	test(logint(t))

The function $\text{test}(x)$ returns x if $x > 1$, otherwise it returns 0, which means wait for the next available data set.

Waits for new data (about 0.5 sec)

Waits computed time, uses most recent data

We pass $\text{logint}(t)$ to the function $\text{test}()$.

Figure 3-18. Modifications to wait for data for log intervals <1 second.

Let's expand this program to do a couple of large step changes in light, and track the response with this timing function. The program is illustrated in *Figure 3-19* below, and is available at `/home/licor/apps/examples/VariableLogInt.py`.

```

/home/licor/apps/examples/VariableLogInt.py
ASSIGN('logint', exp="lambda x: 30/(1+50*math.exp(-0.03*x))+0")
ASSIGN('test', exp="lambda x: x if x >= 1 else 0")
LOOP(list="1500,50,1500", var='x')
  SETCONTROL('Qin', "x", 'float')
  LOOP(dur="15", units='Minutes', var='t', mininc="test(logint(t))")
    LOG(avg='Off', match='Off', flr='0: Nothing')

```

LOOP

Type: List

List: 1500,50,1500
Must eval() to a list or TABLE

Loop variable: x
Variable

Minimum time per cycle: sec
(Optional) Must eval() to a number (default=0.1)

SETCONTROL

Target: Qin
Light incident on leaf set point

Value: x
Must eval() to float $\mu\text{mol m}^{-2} \text{s}^{-1}$

Target (from expression):
(Optional) Must eval() to a string

LOG

Action: Record data

Flr Event: 0: Nothing

Matching: Off

Averaging: Off

Figure 3-19. Program to log at increasing intervals following an abrupt light change.

Monitoring match mode

Suppose you are measuring isotopes in the air stream that is coming from the sample cell gas analyzer. You will need to know when the system is in match mode to know when to ignore that measurement (match mode puts reference air to both cells). A simple BP can monitor the system, and set a spare DAC channel to signal when match mode is active.

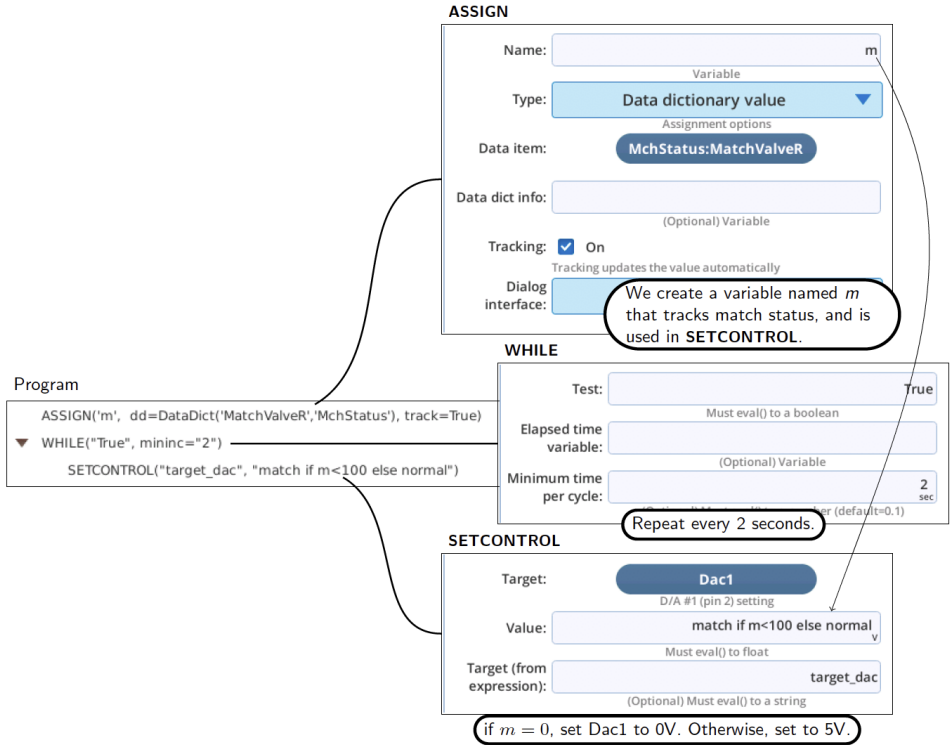


Figure 3-20. A simple program to signal when a match is active.


```

/home/licor/apps/utilities/MatchWatcher.py
ASSIGN('m', dd=DataDict('MatchWatcher', 'MchStatus'), track=True)
ASSIGN('m_last', exp="-1")
ASSIGN('target_dac', exp="Dac1", dlg=Dr..wn("D/A to control", ("Dac1", "..."))
ASSIGN('normal', exp="0", dlg=EditBox("...", "V", desc="0 to 5", checkable="0"))
ASSIGN('match', exp="5", dlg=EditBox("...", "V", desc="0 to 5", checkable="0"))
GROUP(True, 'Opening dialog')
  DIALOG(title="Match Wat...", sub="The... text="This prog...",...,var='button')
  IF('button == 'Cancel')
    RETURN()
  WHILE("True", mininc="2")
    WAIT(event="m != m_last")
    SETCONTROL('target_dac', "match if m<100 else normal")
    ASSIGN('m_last', exp="m")
  WAIT
  
```

ASSIGN

Name: Variable

Type: **Value or expression** (Assignment options)

Value: target_dac = eval(Value)

Dialog interface: **Dropdown list** (If used in DIALOG)

List label: Must eval() to a string

List items: Must eval() to list of strings

These ASSIGNs include information on their dialog interface: one uses a drop down list, and two use edit boxes.

DIALOG

Title: Must eval() to a string

Subtitle: Must eval() to a string

Text box: Must eval() to a string (Optional) Must eval() to a string

Grid items: (Optional) List of variables

Buttons: Must eval() to list of strings

Button response: Variable name

WAIT

Type: **Event**

Event: Must eval() to a boolean

WAIT uses the event option: it waits for *m* to change, then does SETCONTROL, updates *m_last*, and re-loops (WHILE), with a 2 second pause.

When the BP runs, the DIALOG step triggers this:

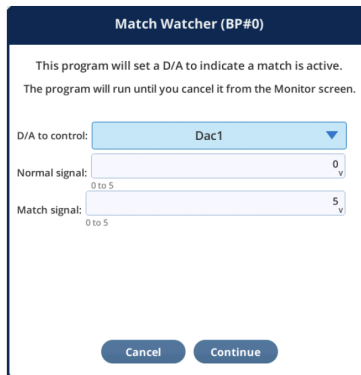


Figure 3-21. Listing of /home/licor/apps/utilities/MatchWatcher.py.

Record and replay a time series

This example has two parts. In Part 1, the objective is to **record a time series of light intensity to a file.** This would be useful, for example, in an understory with passing sunflecks. In Part 2, **use that file to drive a light source through the same time series of light intensities.**

For the data recording part, we won't use normal logging, as we just need a time stamp and light sensor value. A BP can do this for us easily, and *Figure 3-22* below illustrates a test version, since it outputs to the run log, and only runs for 10 seconds.

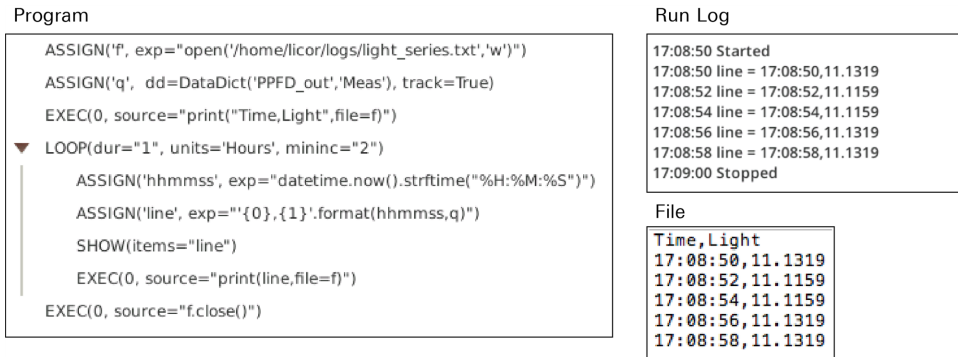


Figure 3-22. Program to record a light time series.

A discussion of the program steps follows:

- 1 Use **ASSIGN** to open a file (using the Python expression for this), assigning *f* to the file object.
- 2 Use **ASSIGN** to track the external quantum sensor with a variable named *q*.
- 3 Write a header ("Time,Light") to the file using a Python print statement inside of an EXEC step. Note the reference to the file (*file=f*) in the print.
- 4 Use **ASSIGN** to make a variable *hhmmss* that captures current time in hh:mm:ss format. (The Python *datetime* module is available and is used for this.)
- 5 Use **ASSIGN** to build the observation line for the file, held in variable *line*.
- 6 Use **SHOW** puts the line in the run log. (Strictly for debugging - we'll get rid of this in the final version.)
- 7 Write the line to the file using `print()`, putting it all in an **EXEC** statement.
- 8 When the loop is done, close the file with `f.close()`, inside an **EXEC**.

The real version of this program is in `/home/licor/apps/examples/GetTimeSeries.py`, minus the **SHOW** statement, and set to run for 1 hour.

Variations. For higher speed data, set the system averaging time to 0 (add a **SETCONTROL** before the **LOOP**, and target it to **SysConst:AvgTime**, which you'll find under **Constants**, then **System** in the **Control Dictionary**), and make the time/cycle in the **LOOP** equal to 0 (this triggers the cycle each time new data is ready.)

Part 2 is making a BP to use this file. All that is required is 2 steps (*Figure 3-23* below). The program is also available on `/home/licor/apps/examples/RunTimeSeries.py`.

LOOP

Type: File

File Name: `cor/logs/light_series.txt` File...

Must eval() to a string

Parsing: Yes Comma

Parse each line? Parsing delimiter

Skip lines: `1` Skip the label line

Must eval() to an integer

Line variable: `Data is in list x` x

Variable

Minimum time per cycle: `2 sec` Run every 2 seconds

(Optional) Must eval() to a number (default=0.1)

`/home/licor/apps/examples/RunTimeSeries.py`

```

LOOP(file="/home/licor/logs/light_serf...e, delim='Comma', skip="1", mininc="2")
  SETCONTROL('Qin', "x[1]", 'float')

```

SETCONTROL

Target: Qin

Light incident on leaf set point

Value: `x[1]` $\mu\text{mol m}^{-2} \text{s}^{-1}$

Target (from expression): `x[0] is time, x[1] is light value`

(Optional) Must eval() to a string

Figure 3-23. Program to drive the light source according to a time series file.

Question: How would you modify this program to log the leaf response while it replays the time series?

The answer: don't even try. Instead, run this program to drive the light source, and use a separate, concurrent autologging program (BP or conventional AutoProgram)

to do the logging. Trying to do both is easy with separate programs, but combining them into one is more difficult. This is the power of BPs - you can split complicated tasks into simple, independent pieces.

Variations on that theme: You could launch both the autolog and the light control programs from a third BP (use the **RUN** keyword and the program's file name), or you could launch the autolog BP from the light control BP, or vice versa.

Those are exercises left to the reader.

Section 4.

Response curves

In principal, generating a response curve consists of the following repetitive pattern: 1) set environmental conditions, 2) wait for stability, 3) log data.

BPs provide a fair amount of flexibility for accomplishing these tasks: Control setpoints can be hand-entered, read from a file, generated from an algorithm, and so on. Wait times can be fixed, or made to adapt to conditions.

The following sections illustrate some response curve options available with BPs.

Basic

A simple BP to generate a response curve for CO2 can be found in the Library GROUPS source, and is shown in *Figure 4-1* below. To change setpoints, edit the LOOP step, and to change target, edit the SETCONTROL step.

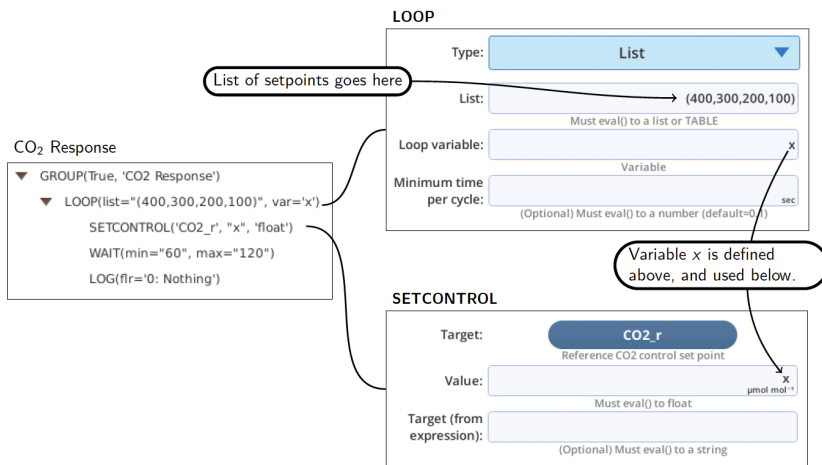


Figure 4-1. Single variable response using LOOP [List].

Since the list entry in LOOP is evaluated, you can use Python expressions there. For example, if you wanted integer setpoints every 100 ppm from 100 to 1000, you could make use of the `range()` method, which takes integer arguments of start, stop, and step.

List:
Must eval() to a list

which would (at run time) make the list

Another option is shown in *Figure 4-2* below, where we've added an EXEC step to make available some list generating utility methods available in the module `list_utility.py`. We use that capability to generate a floating point setpoint list given simply a start value, stop value, and number of desired points.

The screenshot shows a configuration window for a program. On the left, a tree view shows a group named 'CO2 Response' containing an EXEC step and a LOOP step. The EXEC step is configured with the file path `/home/licor/resources/lib/list_utility.py`. The LOOP step is configured with the expression `linearList(100,1000,10)`, which is circled in red. The EXEC step configuration includes fields for Source, File (checked), and Scope (Local selected). The LOOP step configuration includes fields for Type (List), List (linearList(100,1000,10)), Loop Variable (C), and Minimum time per cycle (0.1).

Figure 4-2. Adding linearList capability.

If you want randomized values, use `randomList()` instead of `linearList()`.

List:
Must eval() to a list

Table 4-1. Examples of `randomList()`, and `linearList()`.

Expression	Result
<code>linearList(1, 10, 4)</code>	<code>[1.0, 4.0, 7.0, 10.0]</code>
<code>linearList(5, -5, 6)</code>	<code>[5.0, 3.0, 1.0, -3.0, -5.0]</code>
<code>randomList(5, -5, 11)</code>	<code>[4.0, 3.0, -1.0, -3.0, 2.0, 1.0, -2.0, -4.0, -5.0, 0.0, 5.0]</code>

The sample program `/home/licor/apps/basic/GenericResponse.py` (Figure 4-3 below) illustrates the use of a **TABLE**, which makes it easy to coordinate targets and setpoints. **Table** entries do not by default pass through `eval()` (i.e., no expressions, no variables), so set points have to be explicitly entered.

Tapping the **TABLE** Settings button brings up this dialog, in which you can set controls (rows) and set points (columns).

Table

Name: Variable

Settings: Table of controls and settings.

Fixed additions: comma separated list

Dialog interface: If used in DIALOG

Item label: Must eval() to a string

program

```

PROPERTIES(verbose="True")
ASSIGN('minwait', exp="60", dlg=EditBox("Minimum wait", units="sec"))
ASSIGN('maxwait', exp="120", dlg=EditBox("Maximum wait", units="sec"))
ASSIGN('early', exp="False", dlg=CheckBox("Allow early matching"))
TABLE('table', <CO2_r x 4 settings>, dlg=Button("Controls and settings"))
GROUP(True, 'Opening Dialog')
  ASSIGN('logFile', sd="LOG.FileName", track=True, dlg=Text("Log file"))
  DIALOG(title="Generic R...", sub="", text="Set contr...", var='button')
  IF("button == 'Cancel'")
  LOOP(list="table", var='x')
    WAIT(min="minwait", max="maxwait")
    LOG()
  
```

Since **table** is a **TABLE**, the **LOOP** does an implicit **SETCONTROL** on the rows in the table, a column at a time.

LOOP

Type:

List: Must eval() to a list or TABLE

Loop variable: Variable

Minimum time per cycle: sec (Optional) Must eval() to a number (default=0.1)

WAIT

Type:

Minimum: secs Must eval() to a number

Maximum: secs Must eval() to a number

Early Matching:

Figure 4-3. The program `/home/licor/apps/basic/GenericResponse.py`.

The GenericResponse.py program does support an opening dialog (Figure 4-4 below), from which you can also edit the table.

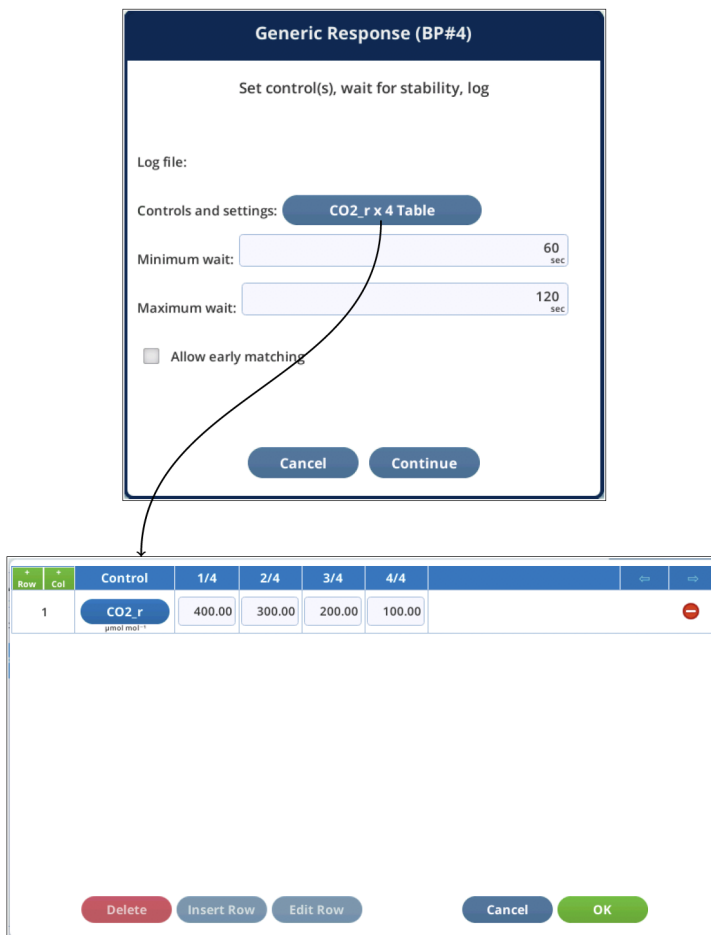


Figure 4-4. The opening dialog for /home/licor/apps/basic/GenericResponse.py.

Multiple controls

How to change multiple controls in a response loop? For example, how could you make a light response with the color becoming more red as intensity drops?

One method is to use the control table interface (such as with `/home/licor/apps/basic/GenericResponse.py`), and make the table look something like this (Figure 4-5 below):

Row	Col	Control	1/7	2/7	3/7	4/7	5/7	6/7	7/7	←	→
1		Qin <small>$\mu\text{mol m}^{-2}\text{s}^{-1}$</small>	1500	1000	750	500	250	100	50		⊖
2		Color_Qin	r80	r85	r88	r90	r92	r94	r95		⊖

Note: these are strings, but *don't* need to be quoted. The Control Table knows, based on the control, when to quote entries and when not to.

Delete Insert Row Edit Row Cancel OK

Figure 4-5 . Light curve with increasing fraction of red as light drops.

Another method is illustrated by the example program `/home/licor/apps/examples/LightColorCurve.py` (Figure 4-6 on the next page), which uses programmatically determined set points for intensity and color.

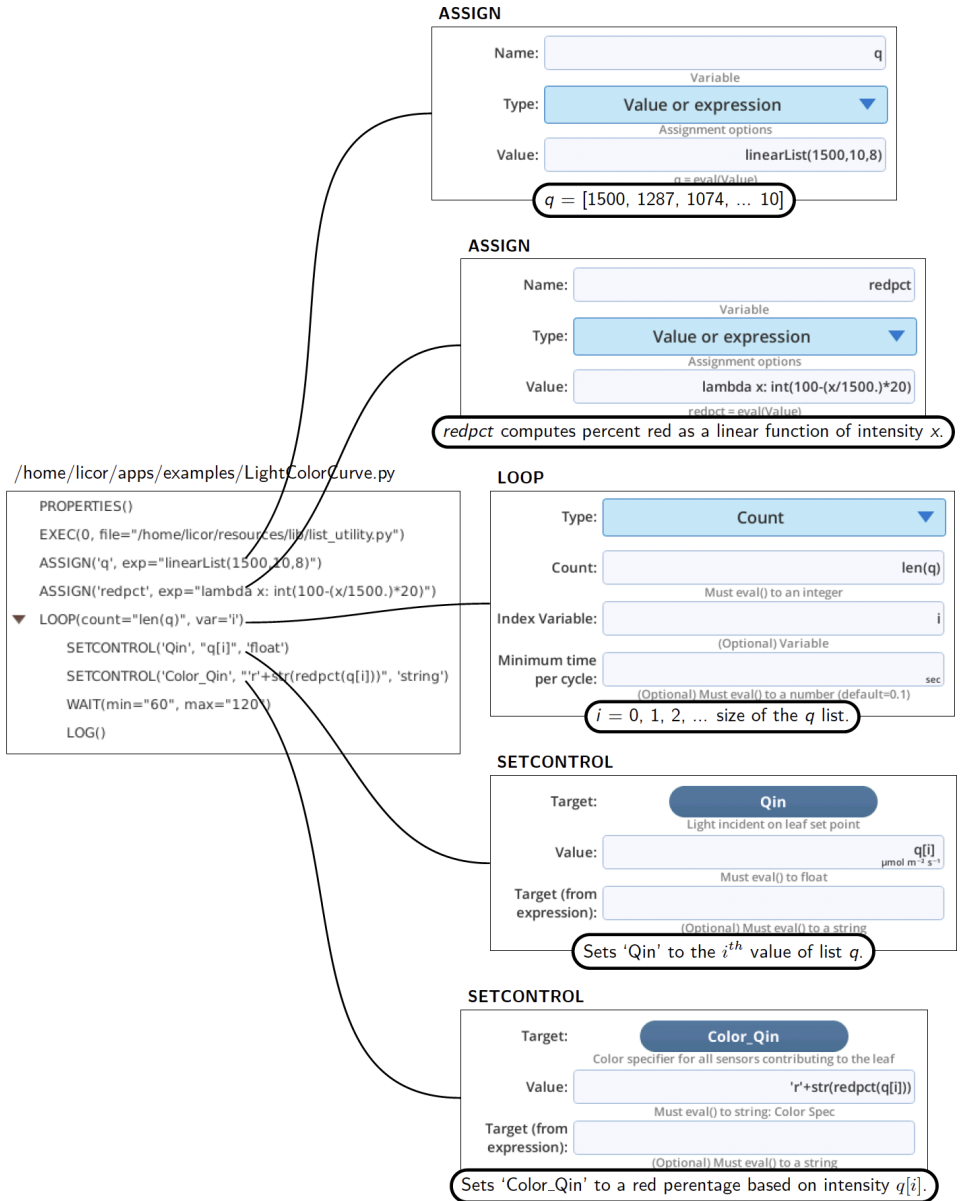


Figure 4-6. Programming color, from /home/licor/apps/examples/LightColorCurve.py.

If we test run this program in verbose mode, we can see the values it picks for the each setpoint (*Figure 4-7* below).

The screenshot displays a test runner interface with the following components:

- Properties Panel (Left):**

```

PROPERTIES(verbose="True")
EXEC(0, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('q', exp="linearList(1500,10.8)")
ASSIGN('redpct', exp="lambda x: int(100-(x/1500.)*20)")
LOOP(count="len(q)", var='i')
  SETCONTROL('Qin', "q[i]", 'float')
  SETCONTROL('Color_Qin', "'r'+str(redpct(q[i]))", 'string')
  WAIT(min="60", max="120")
LOG()

```
- Execution Log (Right):**

```

15:57:20 Started
15:57:20 exec(/home/licor/resources/lib/list_utility.py)
15:57:20 ASSIGN q = [1500.0, 1287.1400000000001, 1074.29, 861
15:57:20 ASSIGN redpct = <function <lambda> at 0xb524eb70>
15:57:20 LOOP 8 times, index=
15:57:20 SETCONTROL Qin to (q[i])=1500.0
15:57:20 SETCONTROL Color_Qin to 'r'+str(redpct(q[i]))=r80
15:57:20 Stability Wait part 1: 60.0 secs
15:57:26 Wait ended by user
15:57:26 Stability Wait part 2: 60.0 secs or until stable
15:57:26 LOG (skipped because no log file open)
15:57:26 SETCONTROL Qin to (q[i])=1287.14
15:57:26 SETCONTROL Color_Qin to 'r'+str(redpct(q[i]))=r82
15:57:26 Stability Wait part 1: 60.0 secs

```
- Control Labels:** Two labels, "Intensity" and "Color", are circled in black. Arrows point from these labels to the corresponding log entries: "Intensity" points to the log entry for setting Qin to 1500.0, and "Color" points to the log entry for setting Color_Qin to 'r'+str(redpct(q[i]))=r80.
- Control Panel (Bottom):** A row of buttons: Trigger, Pause, Resume, Cancel, Start, and Clear. Below this is a progress bar with labels: Open/New, Build, Set, Start, and Monitor (1). A "Waiting 51 secs" indicator is shown next to the Clear button.

Figure 4-7. Test running `/home/licor/apps/examples/LightColorCurve.py`.

Higher dimensions

Suppose you want to measure a response surface instead of a curve? For example, photosynthesis, conductance, etc. as a function of light and CO_2 . With just 2 independent variables, you could do nested control loops, as illustrated by `../basic/NestedResponse`, with a measurement strategy as illustrated by *Figure 4-8* below. Basically, we are measuring a light curve (lot of points) at a few different CO_2 concentrations.

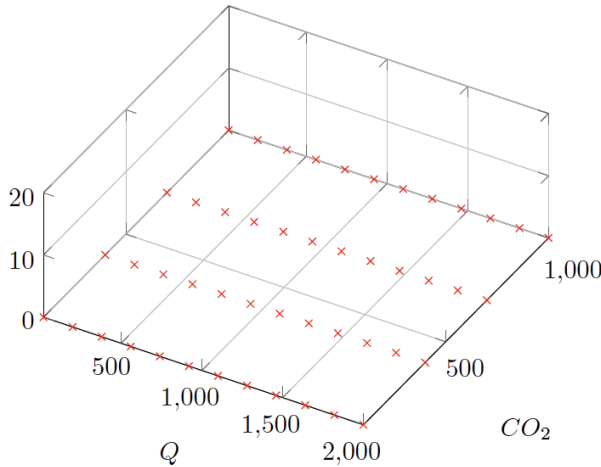


Figure 4-8. Measuring a response surface with 12 light setpoints at 4 different CO_2 setpoints. 48 total points.

This would provide four high resolution light curves, and 12 very crude CO_2 response curves. If you want to balance it out, you could work backwards: how much time do you want to spend maximum (say 2 hours), divide it by the average time to equilibrate at each point (say 5 minutes) to get the 24 total points, take a square root (≈ 5), leaving you with 5 light values and 5 CO_2 values. Not very satisfying.

Well, why not use 12 light and 12 CO_2 points, and just one loop, setting both each time (*Figure 4-9* on the facing page)? That way, we could be done in $12 \times 5 = 60$ minutes.

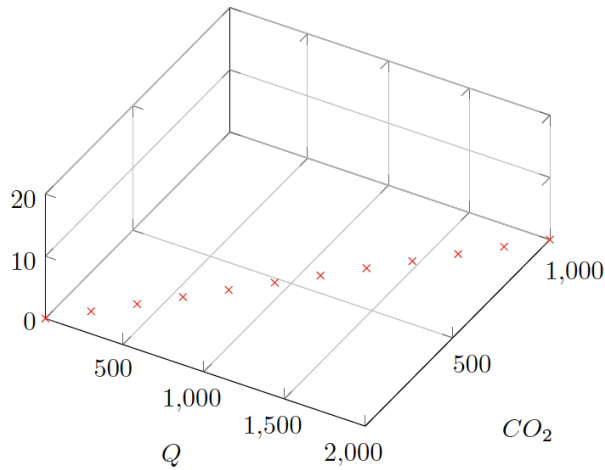


Figure 4-9. Trying to combine responses leaves you with a very poor sampling of the surface.

Plotting out that strategy (Figure 4-9 above) makes it clear why it is not helpful: every point we get is **on a unique light or CO₂ curve**, leaving us with little or no knowledge of what that response surface actually looks like.

But, suppose we use those 12 pairs of points, and randomize the order so that we have a very low correlation between them (Figure 4-10 below).

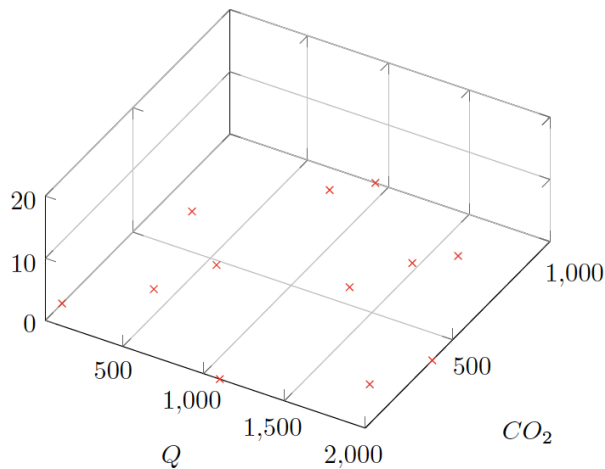


Figure 4-10. Randomizing the order of the Q and CO₂ setpoints to minimize the correlation between them greatly improves the chances of determining the response surface.

Now we are sampling across the light CO₂ space, and have a chance of getting a reasonable estimate of what the surface above it might be like.

This approach is easy to do with a BP. Let's start building a program to try this out (*Figure 4-11* below). We add the list `list_utility.py` module (an **EXEC** step), then make two lists of 12 evenly spaced setpoints: *q* (for light) from 50 to 1500, and *c* (for CO₂) from 50 to 1000). Another **EXEC** passes setpoint lists to `makeOrtho()` to shuffle them, getting back a list of lists, which we put into the original variables. We use **SHOW** to look at the values.

The screenshot shows a software interface for running a program. It is divided into three main sections:

- Program:** Contains the following code:


```
EXEC(1, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('q', exp="linearList(50,1500,12,rounded=0)")
ASSIGN('c', exp="linearList(50,1000,12,rounded=0)")
EXEC(0, source="(q,c)=makeOrtho((q,c))")
SHOW(items="q,c")
```
- EXEC:** Shows the configuration for the second EXEC step. The source is set to `(q,c)=makeOrtho((q,c))`. There are options for 'File' (unchecked, 'Source is a file') and 'Scope' (radio buttons for 'Local' and 'Global', with 'Local' selected).
- Run log:** Shows the execution output:


```
08:33:59 Started
08:33:59 q = [973.0, 50.0, 314.0, 709.0, 1105.0, 1236.0, 1368.0, 18
08:33:59 c = [741.0, 482.0, 309.0, 568.0, 827.0, 50.0, 914.0, 1000.0
08:33:59 Stopped
```

Figure 4-11. Creating orthogonal setpoint lists in a test program using the `makeOrtho()` function from the `list_utility` module.

Now all we need is to add a **LOOP** with **SETCONTROLS**, **WAIT**, and **LOG**, and we have the program `/home/licor/apps/examples/OrthoLightCO2.py` (*Figure 4-12* on the facing page).

Configure LOOP to the number of setpoints in the q (or c) list, and name the count index 'i'

```

/home/licor/apps/examples/OrthoLightCO2.py
PROPERTIES(verbose="True")
EXEC(1, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('q', exp="linearList(50,1500,12,rounded=0)")
ASSIGN('c', exp="linearList(50,1000,12,rounded=0)")
EXEC(0, source="(q,c)=makeOrtho((q,c))")
SHOW(items="q,c")
▼ LOOP(count="len(q)", var='i')
  SETCONTROL('Qin', "q[i]", 'float')
  SETCONTROL('CO2_s', "c[i]", 'float')
  WAIT(min="120", max="300")
  LOG()

```

LOOP

Type: **Count**

Count: **len(q)**
Must eval() to an integer

Index Variable: **i**
(Optional) Variable

Minimum time per cycle: **sec**
(Optional) Must eval() to a number (default=0.1)

SETCONTROL

Target: **Qin**
Light incident on leaf set point

Value: **q[i]**
Must eval() to float
 $\mu\text{mol m}^{-2}\text{s}^{-1}$

Target (from expression): **q[i]**
(Optional) Must eval() to a string

Configure each SETCONTROL to use the i^{th} value in its list (q[i] for light, c[i] for CO₂).

SETCONTROL

Target: **CO2_s**
Sample CO2 control set point

Value: **c[i]**
Must eval() to float
 $\mu\text{mol mol}^{-1}$

Target (from expression): **c[i]**
(Optional) Must eval() to a string

Figure 4-12. A program to measure a CO₂ and light response surface, using orthogonal setpoints.

There are two optional parameters provided by `makeOrtho()` (see *The list_utility module* on page 10-1), that are illustrated in the following example. Suppose you want three independent variables: light, CO₂, and temperature. Making big jumps in CO₂ or light is not a problem (other than longer leaf equilibration times), but with temperature, you would prefer to make the changes in a monotonic manner as much as possible just to minimize system equilibration time. How can that be done?

An example is `/home/licor/apps/examples/OrthoTempLightCO2.py` (Figure 4-13 on the next page).

/home/licor/apps/examples/OrthoTempLightCO2.py

```

PROPERTIES()
EXEC(1, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('temp', exp="linearList(15,30,12)")
ASSIGN('q', exp="linearList(50,1500,12,rounded=0)")
ASSIGN('c', exp="linearList(50,1000,12,rounded=0)")
EXEC(0, source="(temp,q,c)=makeOrtho((temp,q,...)")
SHOW(items="temp,q,c")
▼ LOOP(count="len(q)", var='i')
  SETCONTROL('Tleaf', "temp[i]", 'float')
  SETCONTROL('Qin', "q[i]", 'float')
  SETCONTROL('CO2_s', "c[i]", 'float')
  WAIT(min="120", max="300")
  LOG()

```

Run log (at start)

```

09:39:04 Started
09:39:05 temp = [15.0, 16.359999999999999, 17.73, 19.09, 20.44]
09:39:05 q = [182.0, 973.0, 314.0, 1500.0, 1368.0, 841.0, 577.0, 44
09:39:05 c = [1000.0, 395.0, 136.0, 568.0, 223.0, 655.0, 827.0, 50.0

```

ASSIGN

Name:

Variable

Type: **Value or expression**

Assignment options

Value:

temp = eval(value)

Temperature, 15 to 30, 12 values

EXEC

Source:

Must eval() to a string

File:

Scope:

SETCONTROL

Target: **Tleaf**

Leaf temperature set point

Value:

Must eval() to float

Target (from expression):

(Optional) Must eval() to a string

Figure 4-13. A program to measure a temperature, CO₂, and light response surface using orthogonal setpoints.

Changes to the previous program are:

(line 3, **ASSIGN**) - Added a variable (*temp*) that holds 12 temperature set points from 15 to 30.

(line 6, **EXEC**) - Added *temp* and some optional parameters to the *makeOrtho()* call, so it looks like this"

```
makeOrtho((temp,q,c), lock_index=0, outfile='/home/licor/logs/ortho3_values.txt')
```

`lock_index=0` tells *makeOrtho()* to not randomize the first (0th) list, which is *temp*.

`outfile='/home/licor/logs/ortho3_values.txt'` instructs *makeOrtho()* to also write its results to a file, so we can view or use these same setpoints later (*Listing 4-1* on the facing page).

```

corr_coeff= 0.076648340643
15.0 1105.0 568.0
16.36 182.0 827.0
17.73 1368.0 741.0
19.09 50.0 223.0
20.45 1500.0 309.0

```



```
21.82 314.0 482.0  
23.18 445.0 136.0  
24.55 973.0 1000.0  
25.91 841.0 50.0  
27.27 709.0 655.0  
28.64 577.0 914.0  
30.0 1236.0 395.0
```

Listing 4-1. Listing of /home/licor/logs/ortho3_values.txt.

(line 7, **SHOW**) - Added *temp* to the list.

(line 9, **SETCONTROL**) - Set left temperature to the i^{th} value of *temp*.

Variable stability wait times

Suppose you wish to do a light curve at a few different CO₂ concentrations by nested loops, with an outer loop that changes CO₂, and inner loop that changes light (the program /home/licor/apps/examples/Light_CO2_autofile.py will do this, putting each light curve in its own log file.) An important consideration with this method is the wait time for the first light value needs to be longer than normal, since the CO₂ will have just changed, and the light will have had a big change from the last value of the previous light curve. A method of accommodating that is shown in (Figure 4-14 below).

inner_table settings

row	Co	Control	1/7	2/7	3/7	4/7	5/7	6/7	7/7
1		Qin	1500.0	1250.0	1000.0	750.0	500.0	250.0	100.0
2		minWait	300.00	60.00					
3		maxWait	500.00	120.00					

Extra time for first setpoint, all others use 2nd value.

TABLE *inner_table*

Name:

Settings: **3 x 7**

Fixed additions: minWait,maxWait

minWait:

maxWait:

```

/home/licor/apps/examples/Light_CO2_autofile.py
PROPERTIES(verbose="True")
TABLE('outer_table', <CO2_x 4 settings>)
TABLE('inner_table', <Qin + 2 Items x 7 settings>)
LOOP(list="outer_table", var="outer_index")
  LOG(open="/home/licor/logs/co2_"+str(outer_index)*)
  LOG(rem="automatic file")
  LOOP(list="inner_table", var="inner_index")
    WAIT(min="minWait", max="maxWait")
    LOG(avg="On")
  LOG(close="")
    
```

The variables defined in *inner_table*

Figure 4-14. Adding wait time variables to a table.

The equivalent to *Figure 4-14* on the previous page without using **TABLE** could be done as illustrated in *Figure 4-15* on the next page. Here we take advantage of being able to easily generate set points (*linearList()*). *minWait* is a normal minimum wait time, and *firstWait* is the time to use on the first pass through the light curve each time. Maximum wait times are always 2 times the minimum.

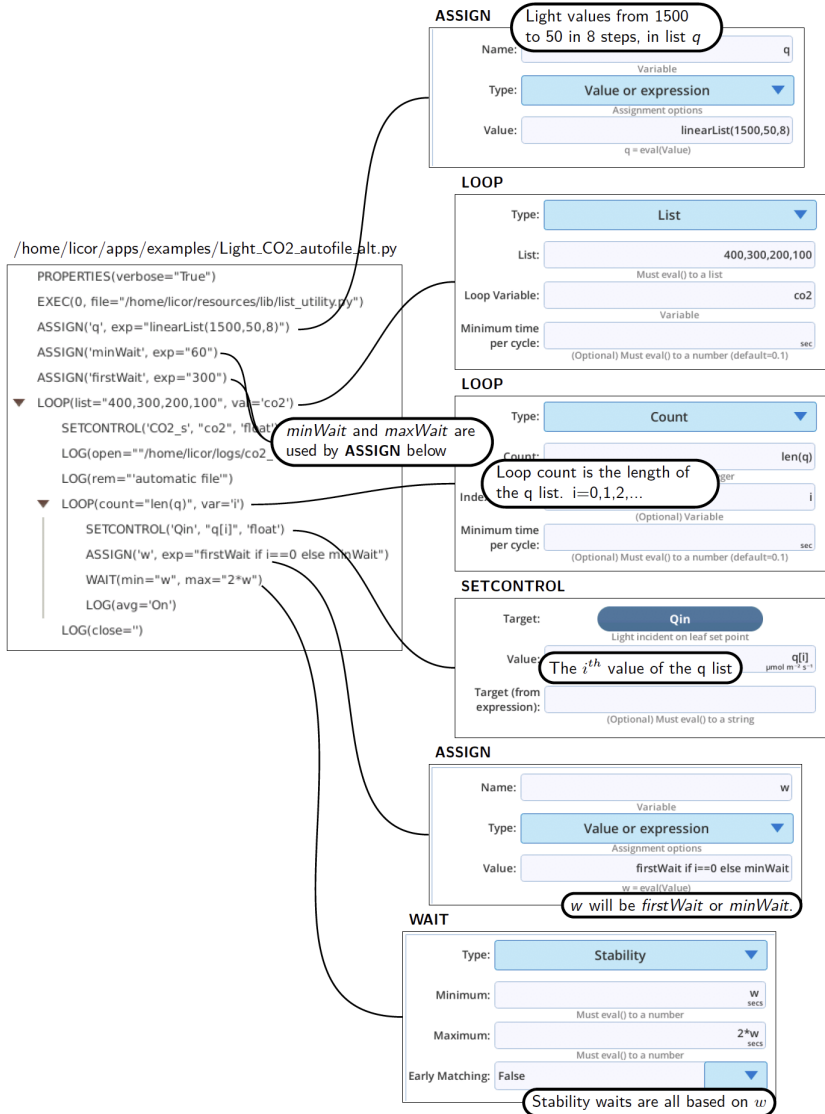


Figure 4-15. Nest control loop with computed wait times.

Section 5.

Using dialogs

It can be very helpful to put a user interface on a BP to guide the user, especially when the user is not the program designer. Consider, for example, a simple BP for doing a light response curve (*Figure 5-1* below).

```
PROPERTIES(verbose="True")
EXEC(1, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('start', exp="2000")
ASSIGN('stop', exp="10")
ASSIGN('count', exp="10")
ASSIGN('setpoints', exp="linearList(start,stop,count)")
▼ LOOP(list="setpoints", var='x')
    SETCONTROL('Qin', "x", 'float')
    WAIT(min="60", max="120")
    LOG()
```

Figure 5-1. A good candidate for a DIALOG front end.

This program has three parameters (starting setpoint, ending setpoint, and increment). If someone (not the designer) wants to use this program but needs to modify the settings, they have to do so in the BP editing environment, where - if they aren't sure of what they are doing - they could inadvertently render the program unworkable. It would be better if the program simply presented the user with something like *Figure 5-2* on the next page.

Light Response Curve (BP#0)

Automatically generates linearly spaced setpoints

Starting value: μmol m⁻² s⁻¹
Qin set point

Final value: μmol m⁻² s⁻¹
Qin set point

Number of set points:
Recommended: 8 to 12

Figure 5-2. A dialog front-end for the light response program.

Happily, setting up such a dialog for a BP is fairly simple, since there is a **DIALOG** step that does most of the work for you. If there are variables in your BP you want to edit, or if you want to let the user control the program's actions via buttons or checkboxes, there will be a few code additions needed to handle that. In general, however, there are three basic steps:

- 1 Insert a **DIALOG** step in the BP at an appropriate place in the execution flow that you want the dialog to appear.
- 2 If the dialog allows for editing program variables, revisit the **ASSIGN** steps for those variables and fill in the dialog-related information for whatever interface you choose.
- 3 If your program flow is going to depend on what button was pressed, or the state of edited variables, add that code.

How to make the BP in Figure 5-1 on the previous page produce the dialog in Figure 5-2 above is shown in Figure 5-3 on the facing page, with the program additions shown in the red box. We put the **DIALOG** step after count is assigned, but before setpoints is computed, since that uses the potentially edited values. We also are handling the **Cancel** button by **IF** and **RETURN** steps. To make *start*, *stop*, and *count* appear in the dialog, they are a) listed in **Grid** items in the **DIALOG** setup, and b) they have interfaces specified in their **ASSIGN** statements.

ASSIGN

Name: Variable

Type: Value or expression

Value: Assignment options

Dialog interface: Edit box ui Variable

Edit box label: 'Starting value' Checkable

Edit box width, units: Small Value becomes dict

Edit box description: 'Qin set point'

`/home/licor/apps/examples/LinearLightResponse.py`

```

PROPERTIES(verbose="True")
EXEC(1, file="/home/licor/resources/lib/list_utility.py")
ASSIGN('start', exp="2000", dlg=EditBox...desc="Qin set point", checkable="0")
ASSIGN('stop', exp="10", dlg=EditBox"...desc="Qin set point", checkable="0")
ASSIGN('count', exp="10", dlg=EditBox...commended: 8 to 12", checkable="0")
DIALOG(title="Light Res...", sub="", text="Automatic...",...,var='button')
IF("button == 'Cancel'")
RETURN()
ASSIGN('setpoints', exp="linearList(start,stop,count)")
LOOP(list="setpoints", var='x')
SETCONTROL('Qin', "x", 'float')
WAIT(min="60", max="120")
LOG()

```

ASSIGN

Name: Variable

Type: Value or expression

Value: Assignment options

Dialog interface: Edit box ui Variable

Edit box label: 'Final value' Checkable

Edit box width, units: Small Value becomes dict

Edit box description: 'Qin set point'

ASSIGN

Name: Variable

Type: Value or expression

Value: Assignment options

Dialog interface: Edit box ui Variable

Edit box label: 'Number of set points' Checkable

Edit box width, units: Small Value becomes dict

Edit box description: 'Recommended: 8 to 12'

DIALOG

Title: 'Light Response Curve'

Subtitle: 'Automatically generates linearly spaced setpoints'

Text box: (Optional) Must eval() to a string

Grid items: start,stop,count (Optional) List of variables

Buttons: 'Cancel', 'Continue'

Button response: button

The BP variables being edited in the dialog

When the user taps a button, the variable named *button* will assume the value of the tapped button's label ('Cancel' or 'Continue', in this case.)

Figure 5-3. The *LinearLightResponse* program uses a *DIALOG* to configure itself.

An exercise, let's add a "dark adapt" feature to this program. It's an on/off sort of thing, so would lend itself well to a check box in the dialog and a boolean variable in the program. Figure 5-4 on the next page illustrates one way to accomplish this:

- 1 Add a variable dark (upper green box), and ASSIGN it to False. Give it a check box interface.
- 2 Add dark to the list of editable items in DIALOG. Put it last if you want it at the bottom of the edit items in the dialog.
- 3 Add some code (lower green box) to perform a dark adaption if dark has been set to True. (We're ignoring the details of what exactly is in the Dark Adapt GROUP in this example).

The figure illustrates the configuration of a dialog box and the associated program code. It consists of three main components:

- Dialog Box (Top Left):** Titled "Light Response Curve (BP#0)", it contains input fields for "Starting value" (2000), "Final value" (10), and "Number of sets" (10). A checkbox labeled "Dark adapt before starting" is highlighted with a green circle.
- ASSIGN Panel (Top Right):** Shows the configuration for a variable named "dark". The type is set to "Value or expression" and the value is "False". The dialog interface is set to "Checkbox" with the label "'Dark adapt before starting'".
- Program Code (Middle Left):** A code editor showing the implementation of the dialog. Key sections are highlighted in green:


```

            PROPERTIES(verbose="True")
            EXEC(1, file="/home/licor/resources/lib/list_utility.py")
            ASSIGN('start', exp="2000", dlg=EditBox(...desc="Qin set point", checkable="0"))
            ASSIGN('stop', exp="10", dlg=EditBox(...desc="Qin set point", checkable="0"))
            ASSIGN('count', exp="10", dlg=EditBox(...commended: 8 to 12", checkable="0"))
            ASSIGN('dark', exp="False", dlg=CheckBox("Dark adapt before starting"))
            DIALOG(title="Light Res...", sub="", text="Automatic...", ..., var='button')
            IF("button == 'Cancel'")
            RETURN()
            IF("dark")
            GROUP(True, 'Dark Adapt')
            ASSIGN('setpoints', exp="linearList(start,stop,count)")
            LOOP(list="setpoints", var='x')
            SETCONTROL('Qin', "x", 'float')
            WAIT(min="60", max="120")
            LOG()
            
```

 A callout box points to the "GROUP(True, 'Dark Adapt')" line, stating: "The dark adapt routine is found in the Library GROUPs section of the 'Build' screen sources."
- DIALOG Panel (Bottom Right):** Shows the configuration for the dialog box. The title is "'Light Response Curve'", the subtitle is "'Automatically generates linearly spaced setpoints'", and the grid items include "start,stop,count,dark". The "dark" item is highlighted with a green circle.

Figure 5-4. Adding a dark adapt option to the opening dialog.

For more on using DIALOG, see DIALOG on page 7-15.

Section 6.

Screen reference

This section explains the various screens associated with building and running BPs, and related details.

The Open/New screen

Open/New is where you can launch a BP, load one and make changes, or start building one from scratch.

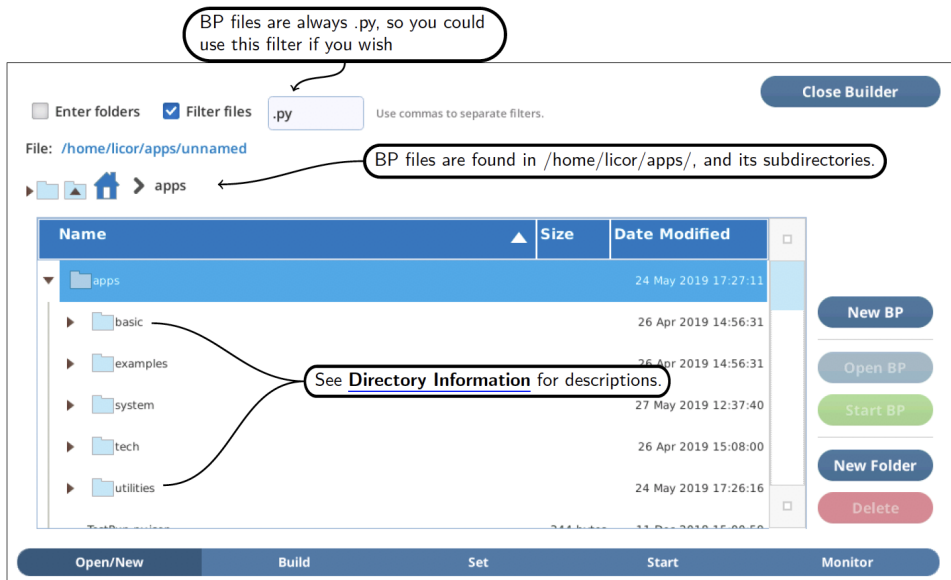


Figure 6-1. The **Open/New** screen lets you pick a BP, or start a new one. See [Directory information](#) on page 6-15 for information about folders.

- **New BP** clears BP steps, and takes you to the **Build** screen.
- **Open BP** reads BP steps from the selected file and displays them in the **Build** screen.
- **Start** launches the selected file. To view its progress, tap **Monitor**.

The Build screen

The **Build** screen is where you can add steps from the list on the left to a BP on the right, arranging them to build up the basic structure of your program. See *Table 6-1* on the facing page for a summary of the steps; more details on each step is available in *Step reference* on page 7-1. Pre-configured step collections (of type DEFINE and GROUP) are also available (*Table 6-2* on page 6-4).

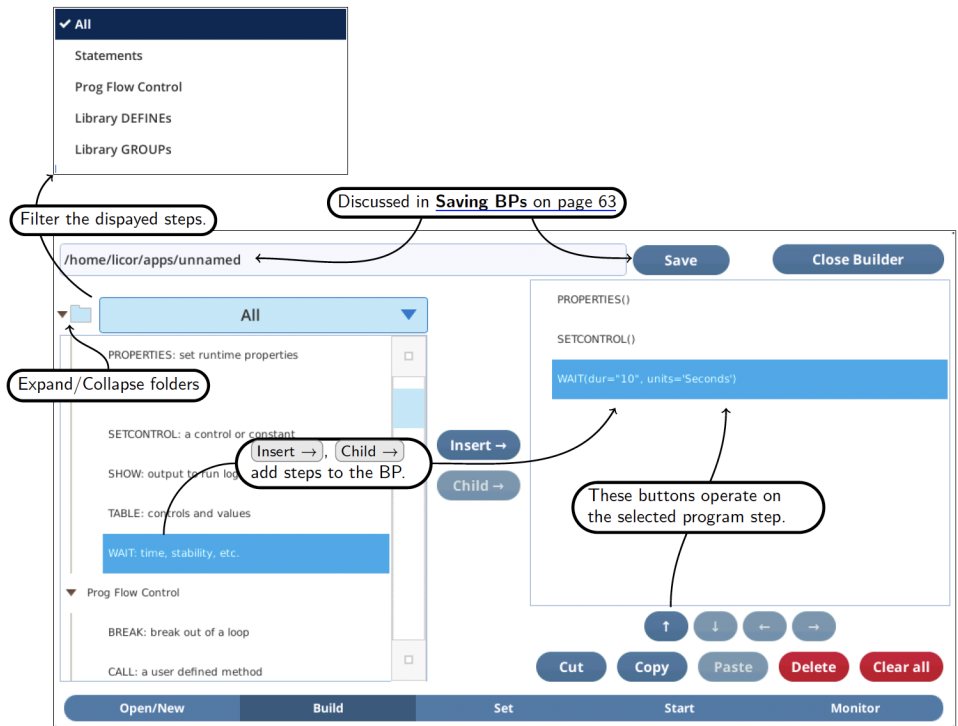


Figure 6-2. The **Build** screen for adding, removing, and rearranging steps in a program.

- **Insert** → copies selected item on left to below highlighted item on right.
- **Child** → appends selected item on left to list of children of highlighted parent item on right.
- **↓** and **↑** moves selected step (and all contained children, if any) up or down.
- **→** moves selected step to the bottom of the parent above it.
- **←** moves selected step out its parent and puts it below the parent.
- **Cut** deletes the selected step, and puts it in the clipboard.
- **Copy** copies the selected step to the clipboard.
- **Paste** inserts a copy of the clipboard into the program list.
- **Delete** deletes the selected step from the program list.
- **Clear all** clears the contents of the program list, changes name to `/home/licor/apps/unnamed.`

Table 6-1. The building blocks of a BP.

Group	Step	Description
Statements	#	A comment line, for making a BP more human-readable.
	ASSIGN	Create a local variable. Can be of any Python type.
	AUTOENV	Provides a method of configuring, starting, stopping any of the six AutoEnvs.
	DIALOG	Displays a dialog box on the console for user interactions.
	EXEC	Performs a Python <code>exec()</code> call on the string, or the specified file name.
	LOG	Log file control: open, close, add data, add a remark.
	PROPERTIES	Set verbosity, pause. Other properties to be added in the future.
	RUN	Launch a BP that is stored in the file system.
	SETCONTROL	Sets a control or constant.
	SHOW	Outputs information to the run log
	TABLE	A control table. Columns are set points and rows are controls.
	WAIT	Wait for some duration, or for stability, or a specific date and time

Table 6-1. The building blocks of a BP. (...continued)

Group	Step	Description
Prog Flow Control	BREAK	Exits a LOOP or WHILE .
	CALL	Calls a subroutine (DEFINE).
	DEFINE	Define a subroutine.
	GROUP	Contains a collection of steps.
	IF	Conditional branching. Has ELSE IF and ELSE options.
	LOOP	Loops over child program steps.
	RETURN	Exits a subroutine or the main program.
	WHILE	Loops while a condition is True.

Table 6-2. Pre-configured collections that can be added to a BP.

Group	Name	Description
Library	AutoLog	Logs at regular intervals for some fixed duration.
DEFINES	BalanceFlow	Adjusts flow rate until sample and reference exhaust flows balance (for the current pump speed).
	ChamberInfo	Gets model number and serial number for present chamber.
	ChamberStatus	Sets the passed in variable to True if sample cell flow is very low.
	RampBlue	Ramps the fraction of blue of the source color between two end points over some time period.
	RampLight	Ramps the light intensity of the source between two end points over some time period.
Library GROUPs	AutoLog	Logs for a fixed time at regular intervals.
	CO ₂ Response	A simple CO ₂ response loop.
	Dark Adapt	If a fluorometer is present, does a typical dark adapt routine.
	Dialog Example	Template for using a dialog.
	Dialog: Buttons only	Template for a simple dialog.
	Light Response	Does a simple light curve.
	Time examples	Useful time functions.
	Trigger User Prompts	Displays the user-prompts page.
	Wake if sleeping	Wakes the instrument if in sleep or standby

The Set screen

Each program step has attributes that define its behavior. **WAIT**, for example, can suspend the BP's operation for a fixed time duration, or until stability to be achieved, or until a specific time of day.

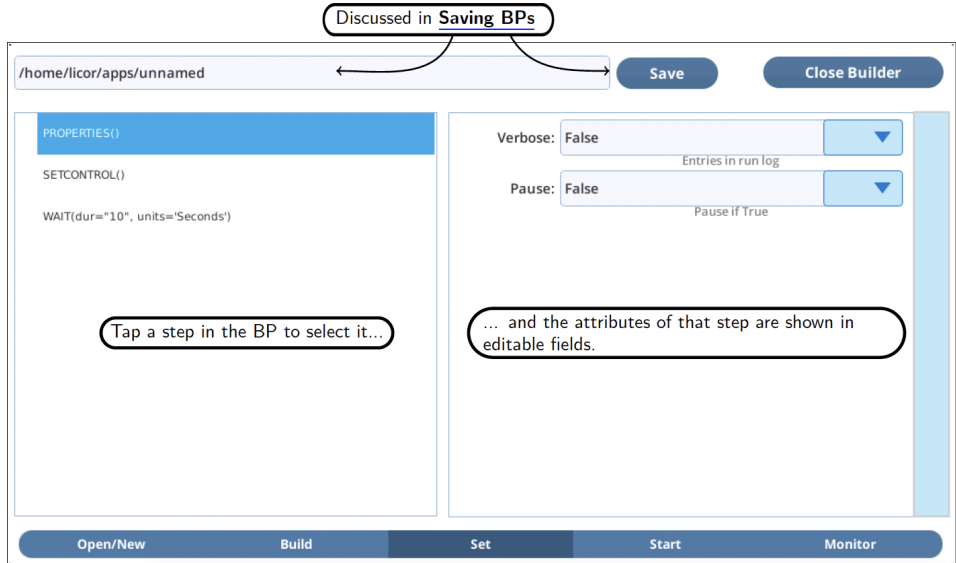


Figure 6-3. Use the **Set** screen to configure individual program steps. See *Saving BPs* on page 6-17 for more details.

See *Set screen interface tools* on page 6-9 for a discussion of the various interface tools that might appear on the right side of the **Set** screen.

In this document, we illustrate how to configure BPs in the **Set** screen as in *Figure 6-4* on the next page a listing on the left, and one or more setting screens (the right hand side of the **Set** screen) on the right.

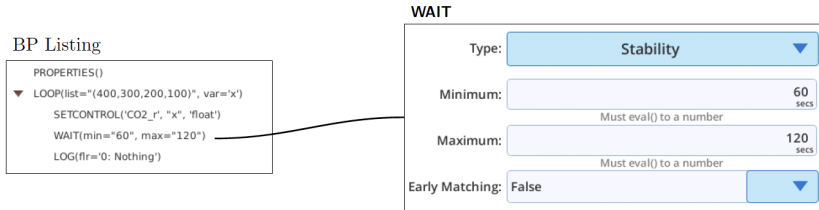


Figure 6-4. An example of how this documentation represents how to set the parameters for a step in a BP.

The Start screen

The **Start** screen provides a place to test run the BP being developed.

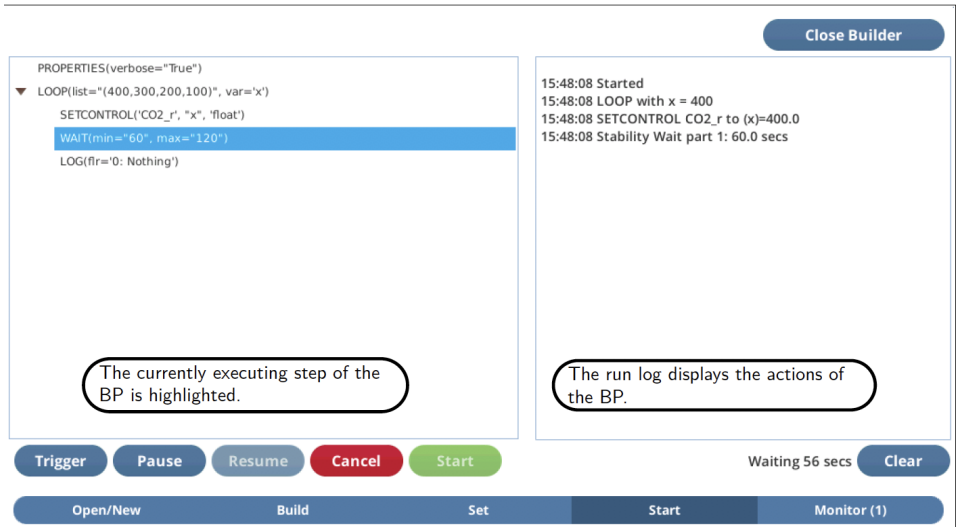


Figure 6-5. Running a BP.

Cancel stops the program, **Pause** pauses the program (enabling **Resume**, which will start it running again), **Trigger** terminates a **WAIT**.

If you **Trigger** when a program is paused, the program will remain in a paused state, but go on to the next step. It also will output any messages about that step even if the **Verbosity is False** (by default, but is set by **PROPERTIES**). More details are in *Debug mode* on page 6-14.

Clear clears the run log. Note that if you tap this accidentally, you can still see the entire run log for that program by going to the **Monitor** screen and - if it still running - selecting the BP there (it will have PID=0).

When a program is running in the **Start** screen, you can continue editing in the **Set** and (if available) **Build** screens. You are also allowed to press **Open BP**, loading a completely different program into the **Build** and **Set** screens. None of these changes will have any effect on the BP running in **Start**. Once it is finished, however, the program displayed in the **Start** will update to whatever changes you might have made.

The Monitor screen

The **Monitor** screen allows you to see all running BPs, and selectively monitor progress, or **Cancel**, **Pause**, etc. *Debug mode* on page 6-14 can be used, as well as the **Start** screen.

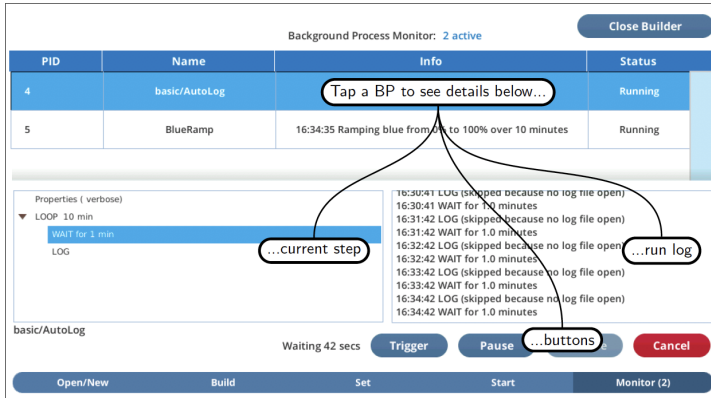


Figure 6-6. Running a BP.

There is a second method to access the Monitor Screen (Figure 6-7 below).

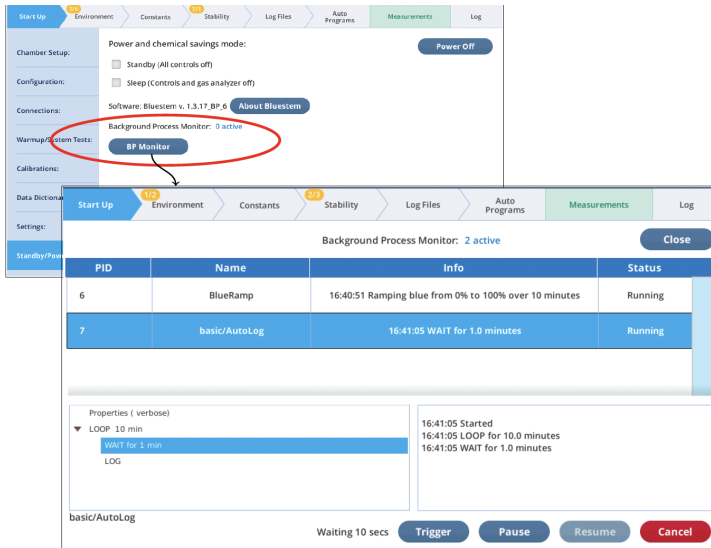


Figure 6-7. Access the Monitor Screen from Start Up.

Set screen interface tools

The interface tools for setting steps are explained below.

Simple objects

A dropdown menu displays a fixed list of choices when you tap it (*Figure 6-8* below). Note: There may be more items in the list than are shown; you won't know until you try scrolling (touch and drag) down.

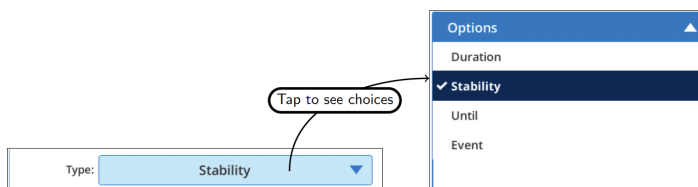


Figure 6-8. Drop down menus display fixed choices.

Tapping in an edit box will cause the full keyboard to appear (*Figure 6-9* below). See *Overview* on page 2-1.

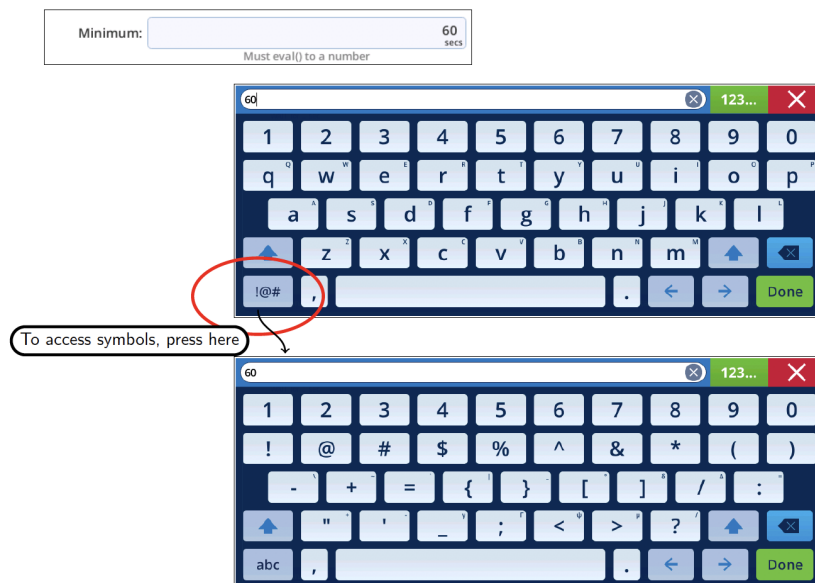


Figure 6-9. Edit boxes use the full screen keyboard.

The combo box is a combination of an edit box and a menu.

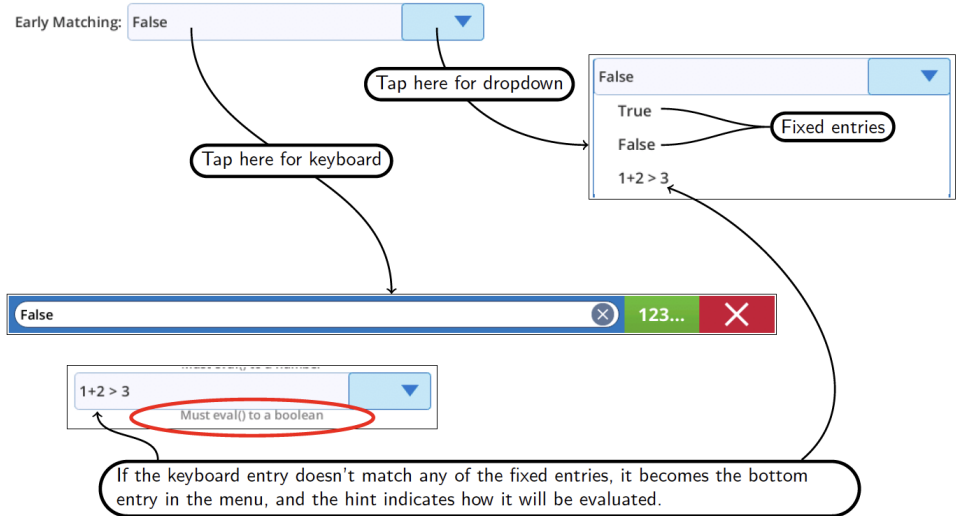


Figure 6-10. Combo box illustration.

Buttons are used to access the appropriate support dialog.

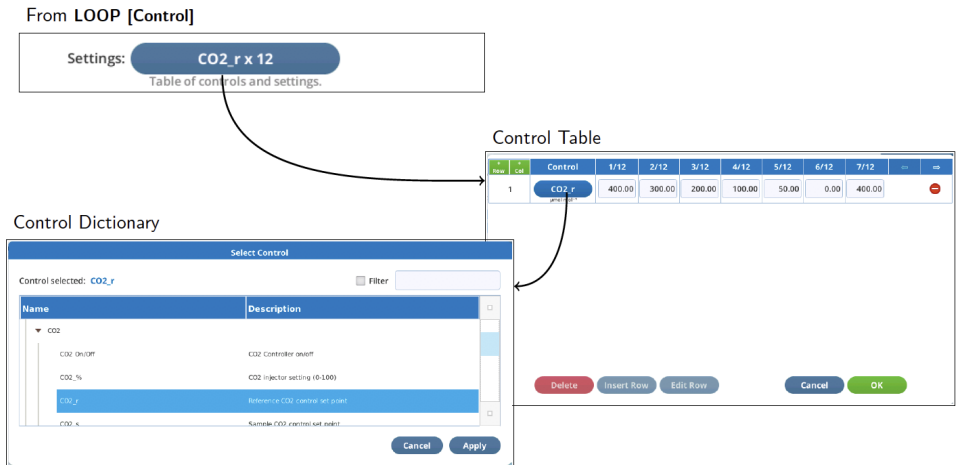


Figure 6-11. When editing BPs, buttons access dialogs.

File names are sometimes specified with a combination of button and edit box (Figure 6-12 on the facing page). Don't forget that strings should be quoted.

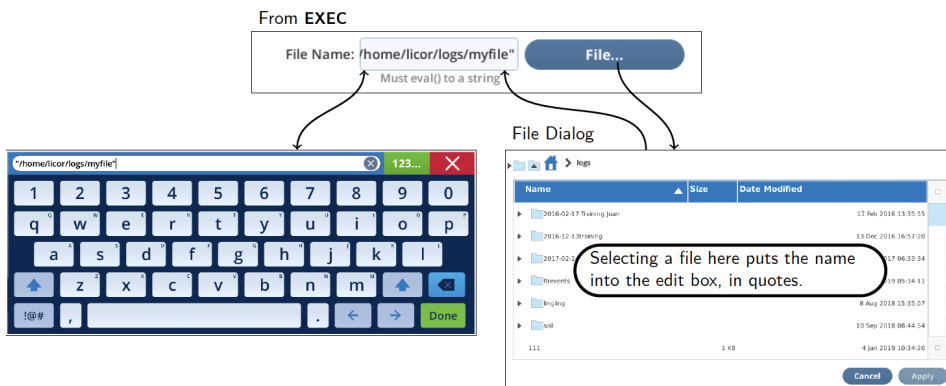


Figure 6-12. Picking a file name can be done by simply typing or picking (if the file exists), or any combination.

Control table

The **Control Table** window is used in the LOOP [Control] interface. It is a table whose rows correspond to controls, and whose columns correspond to set points. Thus, in the example below, Light and Fan speed are being set (together) to 5 set-points each.

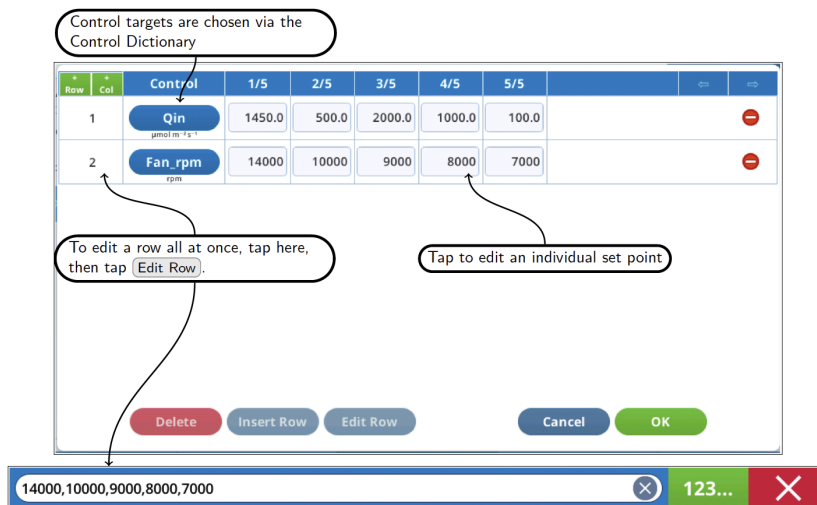


Figure 6-13. The Control Table dialog.

You can have **sparse entries in the table** (Figure 6-14 below), either by entering an empty entry for a cell, or by successive commas when editing a line. The rule is, if there is not a **valid entry for a control for that setpoint, no change will be made to that control.**



Figure 6-14. Missing values are allowed.

Row	Col	Control	1/9	2/9	3/9	4/9	5/9	6/9	7/9		
1		Qin <small>$\mu\text{mol m}^{-2} \text{s}^{-1}$</small>	1500	1250	1000	750	500	250	100		⊖
2		Color_Qin	r90								⊖
3		CO ₂ _s <small>$\mu\text{mol mol}^{-1}$</small>	410								⊖

Figure 6-15. Configured for a light curve at a particular color and CO₂ concentration.

Note that no Control Table entry is passed to Python's eval(), so you cannot use expressions or variable names.

Data dictionary

The Data Dictionary dialog is used in **ASSIGN** for assigning a local variable. It is a dialog similar to the one used in Data Dictionary in Start Up. Items are arranged by Group.

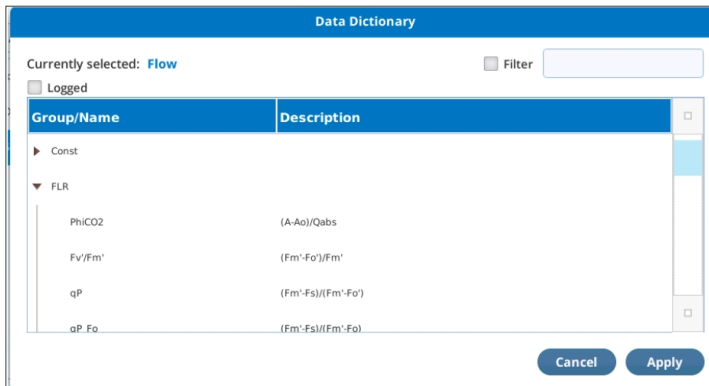


Figure 6-16. The Data Dictionary dialog.

Control dictionary

The Control Dictionary is used for selecting a control or constant that can be set. The **SETCONTROL** and **AUTOENV [Define]** steps use it. For contents, see *Control dictionary map* on page 8-1.

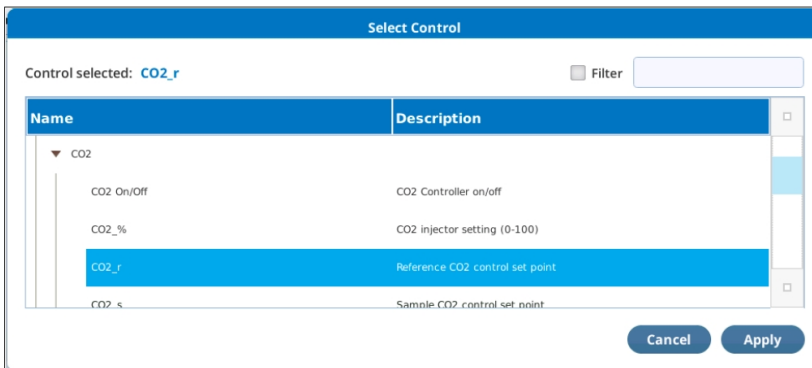


Figure 6-17. The Control Dictionary user interface.

Status dictionary

The Status Dictionary is used for selecting a system value (that is not in the Data Dictionary) to be monitored with the **ASSIGN** step. For contents, see *Status dictionary map* on page 9-1.

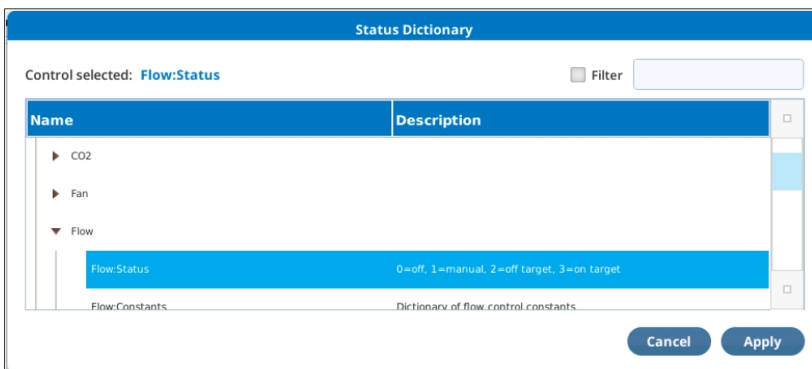


Figure 6-18. The Status Dictionary user interface.

Debug mode

When a program is in a paused state, **Resume** gets it running again. When a BP is paused, **Trigger** will only execute the next step, and the program remains paused. This allows you to slowly walk through your program (**Trigger**, **Trigger**, **Trigger**...) at your own pace. This is Debug Mode. While in this mode, each step produces output in the run log indicating what will happen next, even if Verbosity (in the **PROPERTIES** step) is **False**.

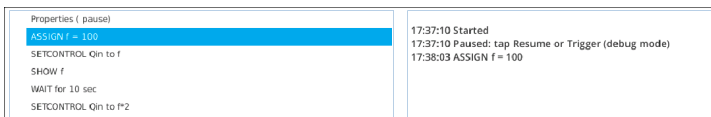
To put a running program into a paused state, tap **Pause**. You can also pause a program at a certain place by inserting a **PROPERTIES** step there, with **Pause = True**.

An example of operating in Debug mode follows, with the simple program shown. Note the first step is **PROPERTIES**, with **Pause** set to **True**.

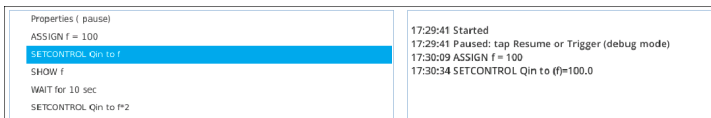
Tap **Start** and the BP immediately pauses because of the **PROPERTIES** statement with **Pause=True**.



Tap **Trigger**. The run log indicates it is about to do an **ASSIGN**. Note that the highlighted line and run log output are indications of what will happen when you tap **Trigger**.



Tap **Trigger**. The run log indicates it is about to do a **SETCONTROL**, setting *Qin* to *f*, which is **100**.



Tap **Trigger**. The run log indicates it is about to do **WAIT**. Tap **Trigger** again to actually begin the **WAIT**.

<pre>Properties (pause) ASSIGN f = 100 SETCONTROL Qin to f SHOW f WAIT for 10 sec. SETCONTROL Qin to f*2</pre>	<pre>17:29:41 Started 17:29:41 Paused: tap Resume or Trigger (debug mode) 17:30:09 ASSIGN f = 100 17:30:34 SETCONTROL Qin to (f)=100.0 17:30:55 f = 100 17:30:55 WAIT for 10.0 seconds</pre>
---	--

Tap **Trigger**. The run log indicates it is about to do SETCONTROL. Tap **Trigger** again to actually do the SETCONTROL.

<pre>PROPERTIES(pause="True") ASSIGN(f, exp="100") SETCONTROL("Qin", "f", "float") SHOW(items="f") WAIT(dur="10", units="Seconds") SETCONTROL("Qin", "f*2", "float")</pre>	<pre>16:00:58 Started 16:00:58 Paused: tap Resume or Trigger (debug mode) 16:01:47 ASSIGN f = 100 16:02:00 SETCONTROL Qin to (f)=100.0 16:02:34 f = 100 16:02:34 WAIT for 10.0 seconds 16:03:13 SETCONTROL Qin to (f*2)=200.0</pre>
--	---

Directory information

The "home" directory for BPs is `/home/licor/apps`. You are free to store your BPs here, or in any subdirectories you may care to make. The system does ensure that there are several subdirectories present (listed below), and populated with some files that are write protected: you will not be able to overwrite them, but you can delete them. (If you do accidentally delete one, it is automatically replaced the next time the instrument powers up.)

- `/home/licor/apps/basic`. Contains a basic set of programs.
- `/home/licor/apps/examples`. Examples used in this document.
- `/home/licor/apps/system`. This directory contains a number of programs that support some features of the user interface. It also contains a suite of tests used to verify BP operations.
- `/home/licor/apps/tech`. Contains useful tech BPs. In certain circumstances, you might be instructed to run one of these as part of technical support or troubleshooting.
- `/home/licor/apps/utilities`. Potentially useful BPs.

There are some other BP-related directories that are created and maintained by the system, and each contains one or more write-protected files. You are free to add appropriate files to these directories, once you understand how these files are used.

- `/home/licor/resources/defines`. These are BPs that contain one **DEFINE**. These files show up in the "Library Subroutines" portion of the source list (*Figure 6-19* on the next page).

- `/home/licor/resources/groups`. These are BPs whose first step is a **GROUP**. Files here show up in the **Library Groups** portion of the source list (Figure 6-19 below).
- `/home/licor/resources/lib`. Contains Python modules (.py files) that you might want to link to BPs via the **EXEC** step.

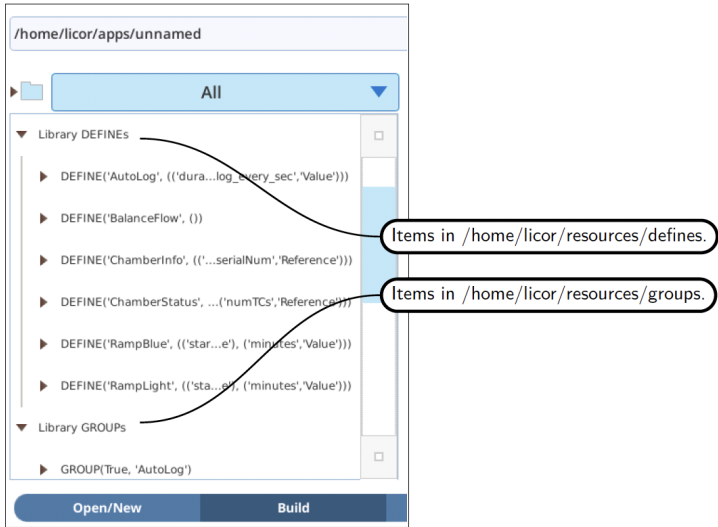


Figure 6-19. These items come from the file system, so you can add your own and make them available here (after a restart).

Saving BPs

BPs can be saved from either the **Build** screen or the **Set** screen.

The edit box at the top of either screen shows the name of the last loaded or saved file (`/home/licor/apps/unnamed` if you tapped **Clear All** in **Build** or **New** in **Open/New**), so if you tap **Save**, that file will be overwritten. An overwrite dialog is produced.

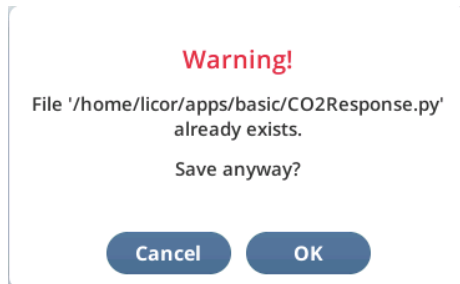


Figure 6-20. You are alerted when overwriting a file.

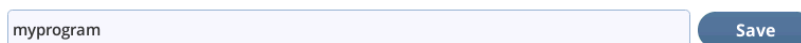
Factory supplied BPs (in the directories described in *Directory information* on page 6-15) are write protected, so if you wish to modify and save them, you will see an error message (Figure 6-21 below). Similarly, you will get an error if you try to write to a directory that is not there, or to one for which you do not have permission.



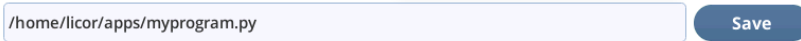
Figure 6-21. Errors saving a BP are reported.

Use the edit box to rename the BP and/or change its location. If you aren't sure of where to store a BP, follow the example in *Section 6.* on page 6-1: clear the whole entry, enter a name, and tap. If your name does not start out with `/home/licor/apps/`, the system will prepend it automatically. Also, if you don't end with `.json`, it is appended automatically. The final name is always reported back (unless there is an error).

- 1 Tap in the edit box, clear it, and type in a new name.



- 2 Once the new name is entered, tap **Save**.



The edit box now reports the full name actually used. Notice how the system conveniently supplies the base directory and/or the .py suffix if you leave it off.

Note: No quotes needed here: this is not processed at run time, but is live.

Note: You have to actually tap the **Save** button to save it. Just typing in a name on the keyboard dialog and tapping **Done** only sets the file name, it doesn't save it.

Section 7.

Step reference

This section provides a reference for steps used in background programs.

- comment

A comment is just that, and is there to clarify things for the user.



Figure 7-1. Comment.

ASSIGN

ASSIGN creates a local variable for your program, which you name in the **Name** field. There are numerous options:

Value or expression

The **Value** entry is evaluated with the Python `eval()` statement, and the result assigned to the variable specified in **Name** entry. The result of the `eval()` determines the Python type (str, float, int, list, etc.) of your variable. Values assigned to an expression have 5 interface options for appearing in **DIALOG**.

ASSIGN f = 100

The right hand side of the equation goes here.

Discussed in [Grid Items](#)

Name: Variable

Type: **Value or expression** Assignment options

Value: f = eval(Value)

Dialog interface: **None** If used in DIALOG

Dialog Interface Options

Checkbox (If used in DIALOG)

Checkbox label: Must eval() to a string

Dropdown list (If used in DIALOG)

List label: Must eval() to a string

List items: Must eval() to list of strings

Editbox (If used in DIALOG)

Editbox units: Must eval() to a string

Editbox label: Checkable Value becomes dict

Editbox description: Must eval() to a string

Text (If used in DIALOG)

Text label: Must eval() to a string

Radio buttons (If used in DIALOG)

List label: Must eval() to a string

List items: Must eval() to list of strings

Figure 7-2. ASSIGN to an expression. See Grid items on page 7-16 for more details.

Data Dictionary value

A local variable can be assigned to anything that can be found in the **Data Dictionary** (Figure 7-3 below). The assignment can be a 'snap shot' (capture the value, and keep it) or tracked (variable continually updated automatically).

You can connect to live or logged values.

The figure shows two parts of the software interface. The top part is the 'Data Dictionary' window, which has a blue header and a table of variables. The 'Flow' variable is selected. A callout bubble points to the 'Logged' checkbox, which is currently unchecked. The bottom part is a configuration dialog for a variable named 'f1'. It has several fields: 'Name' (f1), 'Type' (Data dictionary value), 'Data item' (Meas:Flow), 'Data dict info' (Optional) Variable, 'Tracking' (checked On), 'Dialog interface' (None), and 'Dialog Interface Option' (Text). A callout bubble points to the 'Tracking' checkbox with the text 'Note the tracking option.'

Group/Name	Description
Tair	Chamber air temperature
Fan_speed	Chamber fan rotation rate
ΔPcham	Chamber overpressure
Qamb_out	External ambient PPFID
Flow	Flow rate into chamber
Qamb_in	In-chamber ambient PPFID

Currently selected: **Flow** Filter

Logged

Name: Variable

Type: **Data dictionary value** Assignment options

Data item: **Meas:Flow**

Data dict info: (Optional) Variable

Tracking: On Tracking updates the value automatically

Dialog interface: **None** If used in DIALOG

Dialog Interface Option

Dialog interface: **Text** If used in DIALOG

Text label: Must eval() to a string

Note the tracking option.

Figure 7-3. ASSIGN to a Data Dictionary value.

Status Dictionary value

A local variable can be assigned to anything that can be found in the **Status Dictionary** (Figure 7-4 below). The assignment can be a 'snap shot' (capture the value, and keep it) or tracked (variable continually updated automatically).

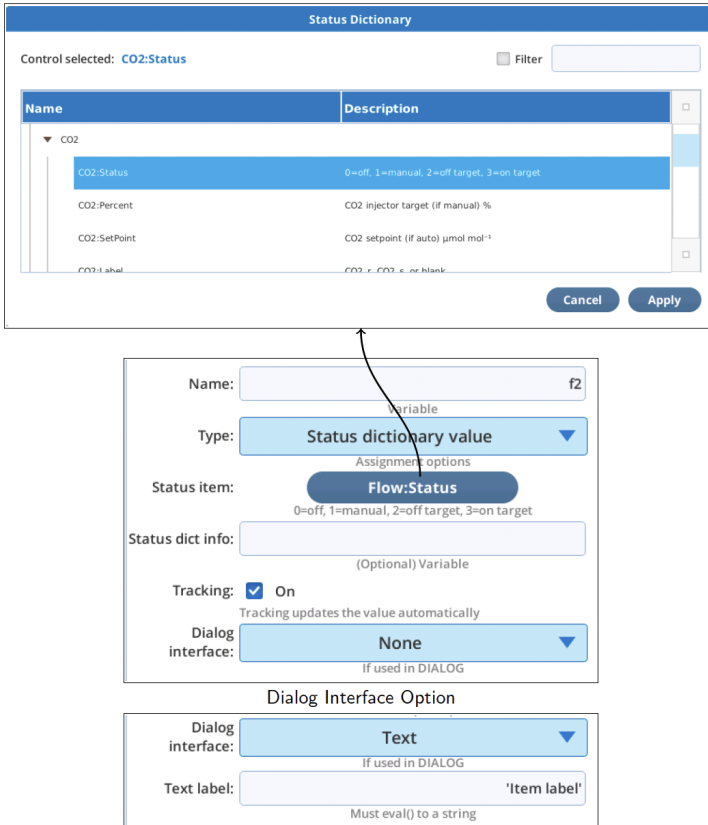


Figure 7-4. Status.

(Advanced) Topic and Key

The **Topic** and **Key** option provides a general purpose method to get to almost any item (or groups of items), even if they are not in the **Data** or **Status Dictionaries**. Most values of interest have a topic and name (key). Every item in the Data Dictionary also has a group and label. The system's Data Dictionary shows group, label, name, and topic (*Figure 7-5* below).

- **Topic:** the topic under which this item is published in internal LI-6800 communications. Except for its appearance in the Data Dictionary, topic is otherwise hidden from the user interface.
- **Name:** the unique (for that topic) identifier for this item.
- **Group:** A collection of items.
- **Label:** Often the same as Name, but can be different.

Group and Label are often used as identifiers (*Figure 7-5* below).

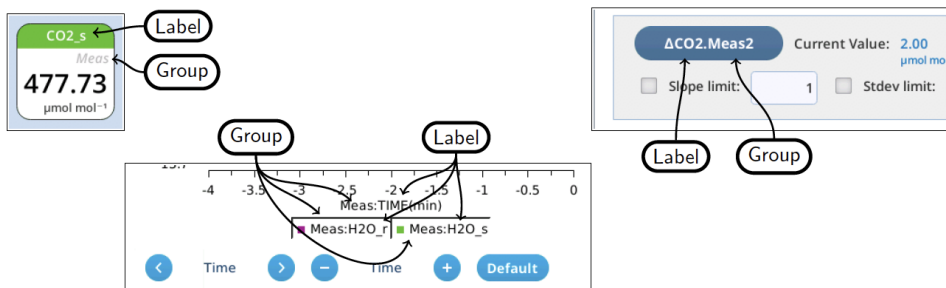


Figure 7-5. Group and label show up several places (and in different order) in the interface, including grids, graphs, and the stability screen.

The screenshot displays the 'Data Dictionary' section of a software interface. A table lists various data points with their group names and descriptions. The entry for 'Qamb_out' is highlighted in blue. To the right of the table, a detailed view for 'Qamb_out' is shown, including its label, group, name, units, topic, data source, and description. A red circle highlights the 'Label: Qamb_out', 'Group: Meas', and 'Name: PPFD_out' fields.

Group/Name	Description
H2O_r	Ref cell H2O concentration
H2O_s	Sample cell H2O concentration
Offset	Offset built-in to Tleaf
Offset2	Offset built-in to Tleaf2
Pa	Atmospheric pressure
Qamb_in	In-chamber ambient PPFD
Qamb_out	External ambient PPFD
TIME	Seconds since Jan 1970
Tair	Chamber air temperature

Qamb_out
Meas
11
 $\mu\text{mol m}^{-2} \text{s}^{-1}$

Label: Qamb_out
Group: Meas
Name: PPFD_out
Units: $\mu\text{mol m}^{-2} \text{s}^{-1}$
Topic: licor/li6850/output/DATA
Data source: live logged
Description: External ambient PPFD
Grids: 1,2

Figure 7-6. Name, Group, and Topic.

In Figure 7-6 above, we are looking at the entry for *Qamb_out*, but that is just its screen label. Its real name is *PPFD_out*, and it can be gotten via the topic (licor/li6850/output/DATA), or the group (Meas).

Figure 7-7 on the facing page shows how to use **ASSIGN [Topic and Key]** to access the entire collection of items in group **Meas**:

Program (specify the group)

```
ASSIGN(f, topic="Meas", track=True)
SHOW(string="json.dumps(f, indent=4)")
```

Program (specify the topic)

```
ASSIGN(f, topic="licor/li6850/output/DATA", track=True)
SHOW(string="json.dumps(f, indent=4)")
```

ASSIGN

Name: Variable

Type: **Topic and Key (Advanced)** Assignment options

Topic: Must eval() to a string

Key: Must eval() to a string

Tracking: On Tracking updates the value automatically

Dialog interface: **None** if used in DIALOG

Run Log (topic)

```
13:07:07 Started
13:07:07 {
  "Tb": 27.4039,
  "DIAG": 2,
  "Flow_r": 451.556,
  "Tleaf": 21.7575,
  "ADC_CH3": 1.87728,
  "abs_h_b": 0.0290961,
  "CO2_r": 124.622,
  "Td_r": -5.69505,
  "CO2_b": 124.622,
  "H2O_a": 4.40418,
  "Fan_speed": 0,
  "Tirga_block": 27.2366,
  "ADC_CH4": 1.86996,
  "ADC_CH5": 1.87266,
  "CO2_d": 125.137,
  "H2O_d": 0.1427,
  "ADC_CH8": 1.87064,
  "TIME": 1549393627.1,
  "ADC_CH2": 1.86855,
  "Vflow": 3.16947,
  "CO2_d": 2.85396,
  "Flow_b": 451.556,
  "Tleaf2": 997.715,
  "CO2_a": 125.829,
  "abs_h_a": 0.0303202,
  "Tchamber": 26.2518,
  "H2O_r": 4.12048,
  "H2O_s": 4.26318,
  "ADC_CH7": 1.86919,
  "GPI0": 265,
  "Flow_a": 0.379623,
  "CO2_s": 127.475,
  "Tchopper": 30,
  "Pchamber": -0.00541641,
  "Ta": 27.313,
  "e_r": 0.401727,
  "e_s": 0.415639,
  "PPFD_in": 0.740404,
  "CO2_s_d": 128.021,
  "abs_c_b": 0.0352488,
  "VPchamber": 2.54628,
  "H2O_b": 4.12048,
  "ADC_CH1": 4.9711,
  "Flow_s": 0.379623,
  "Pchamber": -0.00541641,
  "PPFD_out": 11.1801,
  "abs_c_a": 0.0360925,
  "Flow": 600.01,
  "Txchg": 25.1106,
  "Td_s": -5.24735,
  "ADC_CH6": 1.87314
}
13:07:07 Stopped
```

This can be a Topic (e.g. 'licor/li6850/output/DATA') or a Group (e.g. 'Meas')

Leave the 'Key' blank to get a dictionary of everything in the Group or Topic

Run Log (group)

```
12:52:40 Started
12:52:40 {
  "Offset2": 0,
  "Fan_speed": 0,
  "Tchamber": 26.3516,
  "H2O_r": 4.14767,
  "PPFD_in": 0.836346,
  "H2O_s": 4.31333,
  "Tleaf": 21.4279,
  "PPFD_out": 11.1801,
  "Tleaf2": 997.732,
  "Flow": 600.005,
  "CO2_s": 120.71,
  "Pchamber": -0.00468437,
  "CO2_r": 116.961,
  "Offset": 0,
  "TIME": 1549392760.1,
  "Press": 97.5152
}
12:52:40 Stopped
```

Figure 7-7. Leaving the 'Key' blank returns a dictionary of everything in the Group or Topic.

What can you do with this? Here is an example: the program

`/home/licor/apps/examples/LogMeas.py` illustrates how to log everything in the **Meas** group to a comma separated file, with elapsed time, as fast as new data sets are available.

```

/home/licor/apps/examples/LogMeas.py
GROUP(True, 'Time examples')
  # Useful functions
  ASSIGN('fct_now', exp="lambda : datetime.now()")
  ASSIGN('fct_elapsed', exp="lambda x:(datetime.now() - x).total_seconds()")
  ASSIGN('fct_hr', exp="lambda x: x.hour+x.minute/60.0+x.second/3600.")
  ASSIGN('fct_hms', exp="lambda x: x.strftime('%H:%M:%S')")
  EXEC('meas', topic="Meas", track=True)
  ASSIGN('f', exp="open('/home/licor/logs/mydata_meas.txt', 'w')")
  ASSIGN('labels', exp="sorted(list(meas.keys()))")
  EXEC(0, source="print('Elapsed',',',str(labe...")
  ASSIGN('start', exp="fct_now()")
  LOOP(dur="10", units="Seconds", mininc="0")
    ASSIGN('line', exp="str([meas[i] for i in labels])[1:-1]")
    EXEC(0, source="print(fct_elapsed(start),',',...")
  EXEC(0, source="f.close()")

```

EXEC Prints label line

Source: `print('Elapsed',',',str(labels)[1:-1], file=f)`
Must eval() to a string

EXEC Prints data line

Source: `print(fct_elapsed(start),',',line, file=f)`
Must eval() to a string

Figure 7-8. Logging the Meas group, with elapsed time.

The output of this program is shown in Listing 7-1 on the facing page.

```

Elapsed , 'CO2_r', 'CO2_s', 'Fan_speed', 'Flow', 'H2O_r', 'H2O_s', 'Offset',
'Offset2', 'PPFD_in', 'PPFD_out', 'Pchamber', 'Press', 'TIME', 'Tchamber',
'Tleaf', 'Tleaf2'
0.044217 , 162.356, 164.168, 0, 600.006, 4.26871, 4.24712, 0, 0, 0.811979,
11.1801, -0.00557787, 97.3788, 1549397993.6, 26.6202, 21.6749, 997.995
0.276225 , 162.335, 164.184, 0, 599.989, 4.26887, 4.247, 0, 0, 0.819595,
11.1801, -0.00569272, 97.3789, 1549397994.1, 26.6191, 21.686, 997.981
0.788974 , 162.331, 164.182, 0, 599.983, 4.26883, 4.24701, 0, 0, 0.830257,
11.1801, -0.00573881, 97.3794, 1549397994.6, 26.6181, 21.7068, 997.96
1.219423 , 162.33, 164.191, 0, 599.989, 4.26866, 4.24707, 0, 0, 0.853101,
11.1801, -0.00577292, 97.3796, 1549397995.1, 26.6173, 21.7342, 997.934
1.85875 , 162.337, 164.201, 0, 600.013, 4.26838, 4.24726, 0, 0, 0.869853,
11.1801, -0.00571798, 97.3801, 1549397995.6, 26.6169, 21.7587, 997.926
2.278488 , 162.359, 164.205, 0, 600.005, 4.26831, 4.24704, 0, 0, 0.86224,
11.1801, -0.0057185, 97.3801, 1549397996.1, 26.6169, 21.7815, 997.928
2.801033 , 162.367, 164.202, 0, 600.037, 4.26825, 4.24704, 0, 0, 0.854626,
11.1801, -0.00566928, 97.3802, 1549397996.6, 26.6169, 21.795, 997.928
3.281008 , 162.386, 164.218, 0, 600.009, 4.2684, 4.24721, 0, 0, 0.847013,
11.1801, -0.00566537, 97.3798, 1549397997.1, 26.6169, 21.8024, 997.926
3.809381 , 162.402, 164.22, 0, 600.015, 4.26823, 4.24701, 0, 0, 0.837876,
11.1801, -0.00561485, 97.3793, 1549397997.6, 26.6169, 21.7904, 997.924
4.261619 , 162.42, 164.214, 0, 600.008, 4.26816, 4.2471, 0, 0, 0.822646,
11.1801, -0.0055487, 97.3793, 1549397998.1, 26.6169, 21.7655, 997.922
4.794935 , 162.444, 164.221, 0, 600.03, 4.2682, 4.24693, 0, 0, 0.810461,
11.1801, -0.00549713, 97.3787, 1549397998.6, 26.6164, 21.7421, 997.929

```

```

5.244639 , 162.491, 164.239, 0, 600.011, 4.26834, 4.24663, 0, 0, 0.810461,
  11.1801, -0.00545026, 97.3785, 1549397999.1, 26.6156, 21.7197, 997.943
5.888604 , 162.547, 164.228, 0, 600.003, 4.26827, 4.24641, 0, 0, 0.807415,
  11.1801, -0.00548854, 97.3784, 1549397999.6, 26.6146, 21.6891, 997.945
6.246275 , 162.608, 164.239, 0, 599.998, 4.26845, 4.24642, 0, 0, 0.792185,
  11.1801, -0.00545416, 97.3788, 1549398000.1, 26.6135, 21.6532, 997.938
6.887768 , 162.666, 164.233, 0, 599.985, 4.26856, 4.24679, 0, 0, 0.778478,
  11.1801, -0.00550391, 97.3785, 1549398000.6, 26.6125, 21.6257, 997.939
7.343789 , 162.712, 164.229, 0, 600.01, 4.26845, 4.24664, 0, 0, 0.770865,
  11.1801, -0.0055724, 97.3789, 1549398001.1, 26.6117, 21.6039, 997.946
7.768044 , 162.746, 164.234, 0, 600.008, 4.2684, 4.24698, 0, 0, 0.766297,
  11.1801, -0.00558334, 97.3792, 1549398001.6, 26.6112, 21.5872, 997.95
8.230504 , 162.761, 164.235, 0, 599.988, 4.26836, 4.24708, 0, 0, 0.77391,
  11.1801, -0.00564896, 97.3795, 1549398002.1, 26.6109, 21.574, 997.953
8.756039 , 162.752, 164.235, 0, 599.962, 4.26828, 4.24725, 0, 0, 0.780001,
  11.1801, -0.00570001, 97.38, 1549398002.6, 26.6111, 21.5457, 997.95
9.268605 , 162.711, 164.223, 0, 599.997, 4.26825, 4.24755, 0, 0, 0.780001,
  11.1801, -0.00577735, 97.3802, 1549398003.1, 26.6117, 21.5073, 997.944
9.797924 , 162.667, 164.235, 0, 600, 4.26832, 4.24757, 0, 0, 0.781524, 11.1801,
  -0.00581069, 97.3799, 1549398003.6, 26.6126, 21.4895, 997.945

```

Listing 7-1. Listing of /home/licor/logs/meas values.txt.

(Advanced) XML value

There are a number of "lower-level" values that come from communication with the actual hardware devices (console, head, fluorometer). Many of these are repackaged and available through the **Data or Status Dictionaries, but not everything**. To access (get and set) these, it is necessary to use an XML interface.

The screenshot shows a configuration dialog with the following fields and options:

- Name:** A text input field containing the letter 'f'.
- Type:** A dropdown menu with 'XML value (Advanced)' selected. Below it, the text 'Assignment options' is visible.
- XML:** A text input field containing the string `'licor/li6850/cfg/pump/speed'`. Below it, the text 'Must eval() to a string' is visible.
- Tracking:** A checkbox labeled 'On' is checked. Below it, the text 'Tracking updates the value automatically' is visible.
- Dialog interface:** A dropdown menu with 'None' selected. Below it, the text 'If used in DIALOG' is visible.

Figure 7-9. Assigning a value using xml.

This XML entry `licor/li6850/cfg/pump/speed` is a shorthand for selecting the XML value at

```
<licor><li6850><cfg><pump><speed>value</speed></pump></cfg></li6850>
</licor>
```

AUTOENV

AUTOENV provides a means for a BP to configure and control any of the 6 Auto Controls. There are four sub-options ('Actions') for an AUTOENV step, and are illustrated in the figures below.

A BP can configure and start an AutoEnv, but note that once running, the AutoEnv is independent of the BP that started it; it will keep running when the BP ends, unless the BP stops it or the AutoEnv hits an end with AutoRepeat off.

With **Action** set to **Define** (Figure 7-10 below), you can configure everything relating to the Y-axis.

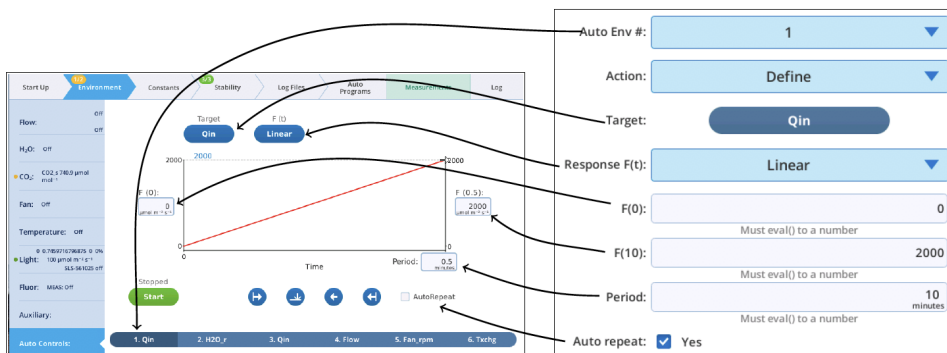


Figure 7-10. How the controls of AUTOENV [Define] map to the user interface.

With **Action** set to **Set time & direction** (Figure 7-11 on the facing page), you can set direction and location on the time axis.

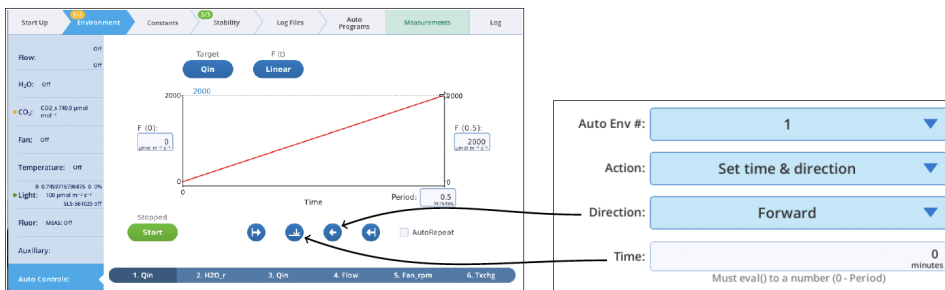


Figure 7-11. The AUTOENV [Set time & direction] controls set the direction and location (time).

With Action set to Start or Stop (Figure 7-12 below), you can turn the AutoEnv on or off.

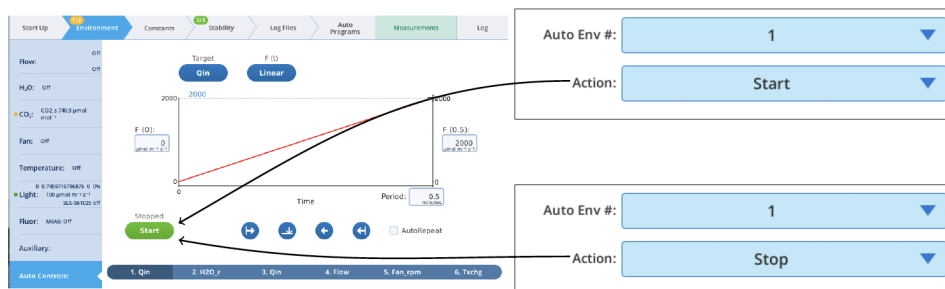


Figure 7-12. AUTOENV [Start] and [Stop].

Figure 7-13 on the next page illustrates the use of AUTOENV.

```

/home/licor/apps/system/tests/AUTOENV_test.py
ASSIGN('par', dd=DataDict('Qin','LeafQ'), track=True)
AUTOENV(1, target='Qin', f_of_t='Linear', range=("0","1000"), period="0.33")
AUTOENV(1, time="0", dir='Forward')
AUTOENV(1, start=1)
ASSIGN('avg', dd=DataDict('AvgTime','SysConst'))
SETCONTROL('SysConst:AvgTime', "0", 'float')
SHOW(string="The Test: PAR should incre...0 sec\nthen level off at 1000 till 40s")
LOOP(dur="40", units='Seconds', var='secs', mininc="3")
  SHOW(string="t={0} PAR={1}'.format(int(secs),int(par))")
SETCONTROL('SysConst:AvgTime', "avg", 'float')
SETCONTROL('Qin', "0", 'float')

```

```

Run log
11:47:51 Started
11:47:52 The Test: PAR should increase with time for 30 sec
then level off at 1000 till 40s
11:47:52 t=0 PAR=10
11:47:55 t=3 PAR=10
11:47:58 t=6 PAR=103
11:48:01 t=9 PAR=269
11:48:04 t=12 PAR=439
11:48:07 t=15 PAR=580
11:48:10 t=18 PAR=690
11:48:13 t=21 PAR=875
11:48:16 t=24 PAR=984
11:48:19 t=27 PAR=998
11:48:22 t=30 PAR=999
11:48:25 t=33 PAR=1000
11:48:28 t=36 PAR=1000
11:48:31 t=39 PAR=1000
11:48:34 Stopped

```

Figure 7-13. Running the AUTOENV test program.

Note that there is nothing in the AUTOENV user interface that actually captures the current state of any AutoEnv; it is write-only, basically. You can, however, get a dictionary of AutoEnv configuration information via `ASSIGN[Topic and Key]`, using the topic `licor/li6850/scripts/autoenv/x/constants` where `x` is 1, 2, ... 6.

BREAK

BREAK provides a way to exit from a **LOOP** or **WHILE** (Figure 7-14 below).

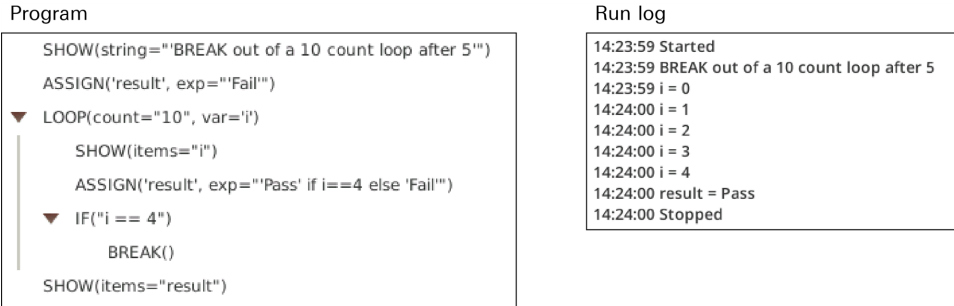


Figure 7-14. Using **BREAK** to exit a **LOOP**.

CALL and DEFINE

DEFINE is the BP equivalent of a subroutine or function. It defines a collection of program steps that can be called from anywhere (**CALL**), and have parameters passed to it.

Variable scope

User defined variables are local to the main program, or to the **DEFINE** function in which they are created (see **EXEC** on page 7-22 for a method to make variables that are global in scope).

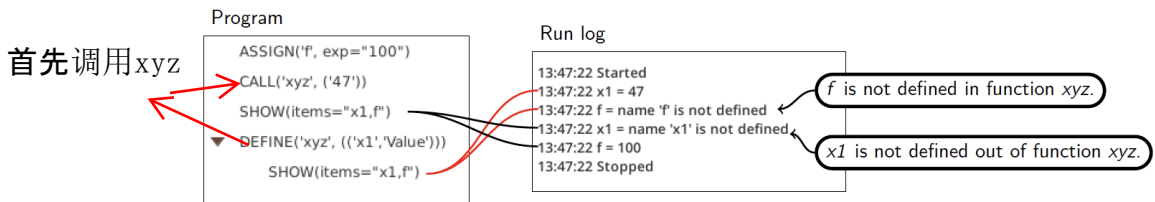


Figure 7-15. Illustration of the local scope of BP variables.

Passing by value or reference

Part of designing a **DEFINE** is specifying for each argument whether it is passed by value or reference.

Normally you pass by value, which means it is the value of the argument in the **CALL** statement that is important. You can use an expression or variable for that argument in the calling statement (e.g., **CALL** Something(x1, x2/16.2)).

Passing by reference forces you to use a variable name, and any changes to the value of that variable are "passed back" to the calling context. *Figure 7-16* below illustrates. The **DEFINE** *PassByTest* takes two arguments, *val* is pass-by-value, *ref* is pass-by-reference. The function doubles both values that are passed in. In the calling context, this does not affect the pass-by-value variable *a*, but does affect *b*, which gets the final value of *ref*.

```

/home/licor/apps/system/tests/PassByRefAndVal_test.py
ASSIGN('a', exp="100")
ASSIGN('b', exp="200")
CALL('PassByTest', ('a', 'b'))
SHOW(string="After PassByTest, a={0}, b={1}'.format(a, b)")
▼ DEFINE('PassByTest', (('val','Value'), ('ref','Reference')))
  SHOW(string="PassByTest 1: val={0}, ref={1}'.format(val, ref)")
  ASSIGN('ref', exp="ref*2")
  ASSIGN('val', exp="val*2")
  SHOW(string="PassByTest 2: val={0}, ref={1}'.format(val, ref))
        
```

DEFINE

Name:

Number of arguments:

Argument #1:

Argument #2:

Run log

```

14:27:10 Started
14:27:10 PassByTest 1: val=100, ref=200
14:27:10 PassByTest 2: val=200, ref=400
14:27:10 After PassByTest, a=100, b=400
14:27:10 Stopped
        
```

Not doubled.

Doubled.

Figure 7-16. Comparison of pass-by-value and pass-by-reference.

Thus, passing by reference is typically the method to use if you want to get some information back from a **DEFINE**, rather than just pass information to it.

DIALOG

DIALOG displays a dialog box for obtaining input from the user (*Figure 7-17* below)

The figure illustrates the configuration of a DIALOG step in a Business Process (BP). It is divided into three main parts:

- Dialog Box (Top):** A window titled "Nested Responses (BP#9)" with a subtitle "Set control(s), wait for stability, log". It contains a text box for the log file path, two buttons for "Outer loop control(s) and settings" and "Inner loop control(s) and settings", two input fields for "Minimum wait" (60 sec) and "Maximum wait" (120 sec), a checkbox for "Allow early matching", and "Cancel" and "Continue" buttons.
- Code Snippet (Middle):** A Python script for the DIALOG step:


```

/home/licor/apps/basic/NestedResponse.py

PROPERTIES(verbose="True")
ASSIGN('minwait', exp="60", dlg=EditBox("Minimum wait", units="sec"))
ASSIGN('maxwait', exp="120", dlg=EditBox("Maximum wait", units="sec"))
ASSIGN('early', exp="False", dlg=CheckBox("Allow early matching"))
TABLE('outer_table', <CO2_r x 4 settings>...("Outer loop control(s) and settings"))
TABLE('inner_table', <Qin x 3 settings>, d...n("Inner loop control(s) and settings"))
▼ GROUP(True, 'Opening Dialog')
  ASSIGN('logFile', sd='LOG:FileName', track=True, dlg=Text("Log file"))
  DIALOG(title="Nested Re...", sub="", text="Set contr...", var='button')
  ▼ IF("button == 'Cancel'")
    RETURN()
  ▼ LOOP(list="outer_table", var='outer_index')
    ▼ LOOP(list="inner_table", var='inner_index')
      WAIT(min="minwait", max="maxwait")
      LOG()

```
- DIALOG Setup Interface (Bottom):** A configuration window for the DIALOG step with the following fields:
 - Title: 'Nested Responses' (Must eval() to a string)
 - Subtitle: 'Set control(s), wait for stability, log' (Must eval() to a string)
 - Text box: (Optional) Must eval() to a string
 - Grid items: logFile, outer_table, inner_table, minwait, maxwait, early (Optional) List of variables
 - Buttons: 'Continue'; 'Cancel' (Must eval() to list of strings)
 - Button response: (Must eval() to list of strings) → button (Variable name)

Annotations in the figure explain that the variable `button` is defined in the DIALOG step and its value is set when the user presses the Continue or Cancel buttons.

Figure 7-17. The parts of a BP dialog, and how they are configured in a DIALOG step.

The DIALOG's setup interface (*Figure 7-17* above bottom) allows you to specify the dialog's parameters:

- **Title** should be a string, or string variable name. The system always appends " (BP#n)" to the title, where n is the BP's pid number.
- **Subtitle** should be a string or a string variable name.
- **Text box** (optional) should be a string or string variable. To force line breaks, embed a nn (backslash n) in the string.
- **Grid items** (optional) is a list of variables whose values can be edited in the dialog.
- **Buttons** is a list of button labels. In no list is specified, an "OK" button will be provided.
- **Button result name.** The BP variable that will contain the label of the button the user taps (closing the dialog). The **DIALOG** step will create this variable if it doesn't already exist.

While the dialog is displayed, the BP is in a wait state until the user presses one of the dialog's buttons.

Grid items

Grid items are items with a user interface, and can be added to a dialog by including one or more variable names in the **Grid items** list of the **DIALOG's** configuration. Each name is used to access the current value of the variable, and the Dialog interface information that was in the **ASSIGN** or **TABLE** setup for that variable.

A **Checkbox** (Figure 7-18 below) is an appropriate interface for a variable that can have a True or False value. The item label tells how the check box is to be labeled.

Figure 7-18. Check box.

A **Dropdown list** (Figure 7-19 on the facing page) is appropriate for letting the user select from a scrollable list.

Figure 7-19. Dropdown list.

An **Editbox** (Figure 7-20 below) is suitable for strings or values. There is also a checked option.

Figure 7-20. Edit boxes can be checked or unchecked.

The **Radio buttons** option (Figure 7-21 on the next page) is appropriate for letting the user select from a small selection. If your label and list of items is too long, it will not all be visible in the dialog box.

Where: Plot AD157 Plot AE200 Plot AE201 Greenhouse

ASSIGN

Name:	<input type="text" value="loc"/>	loc
	Variable	
Type:	<input type="button" value="Value or expression"/>	Value or expression
	Assignment options	
Value:	<input type="text" value="'Plot AD157'"/>	'Plot AD157'
	loc = eval(Value)	
Dialog interface:	<input type="button" value="Radio buttons"/>	Radio buttons
	If used in DIALOG	
List label:	<input type="text" value="'Where'"/>	'Where'
	Must eval() to a string	
List items:	<input type="text" value="'(Plot AD157','Plot AE200','Plot AE201','Greenhouse)'"/>	('Plot AD157','Plot AE200','Plot AE201','Greenhouse')
	Must eval() to list of strings	

Figure 7-21. Radio buttons list.

The **Text** dialog option (Figure 7-22 below) is not editable, but is useful for showing the current value. This is the only dialog option available for all **ASSIGN** items.

Current flow rate: 517.0 $\mu\text{mol s}^{-1}$

ASSIGN

Name:	<input type="text" value="f"/>	f
	Variable	
Type:	<input type="button" value="Data dictionary value"/>	Data dictionary value
	Assignment options	
Data item:	<input type="button" value="Meas:Flow"/>	Meas:Flow
Data dict info:	<input type="text"/>	(Optional) Variable
Tracking:	<input checked="" type="checkbox"/> On	Tracking updates the value automatically
Dialog interface:	<input type="button" value="Text"/>	Text
	If used in DIALOG	
Text label:	<input type="text" value="'Current flow rate'"/>	'Current flow rate'
	Must eval() to a string	

Figure 7-22. The text option shows current value.

The **Table** and **Text** summary dialog options (Figure 7-23 on the facing page) are available for variables assigned to the **TABLE**; one is editable, one is not.

Control table: **CO2_r x 4 Table**

	Control	Q/A	Q/A	Q/A	A/A
1	CO2_r	300.00	400.00	500.00	600.00

Control table: [['CO2_r', [300, 400, 500, 600]]]

TABLE

Name: Variable

Settings: **CO2_r x 4**
Table of controls and settings.

Fixed additions:

Dialog interface: **Table** (dropdown)
If used in DIALOG

Item label:
Must eval() to a string

TABLE

Name: Variable

Settings: **CO2_r x 4**
Table of controls and settings.

Fixed additions:

Dialog interface: **Text summary** (dropdown)
If used in DIALOG

Item label:
Must eval() to a string

Figure 7-23. Dialog options for TABLE.

`_dlg` variables

When you enable **Dialog interface** information in an **ASSIGN** or **TABLE** step, the system creates a second BP variable based on the **Name** field. For example, in *Figure 7-23* on the previous page, a variable named `table` is created that points to the control table, but because there is dialog interface information enabled, a second variable named `table_dlg` is created that contains that information. This auxiliary variable is always named for the Name entry, with `_dlg` appended.

`_dlg` variables can be accessed and manipulated, just like any other variable. You could include one in a **SHOW** statement, for example, to learn its structure. You can also create them explicitly, if you want to include an information in a Dialog for a variable that was not created by **ASSIGN** or **TABLE**.

To illustrate, the check box configuration in *Figure 7-18* on page 7-16 is doing this Python equivalent:

```
xyz = False
xyz_dlg = {'interface': 1, 'target': 'early', 'label': 'Allow early
matching'}
```

The drop down configuration in *Figure 7-19* on page 7-17 is doing this Python equivalent:

```
loc = 'Plot AD157'
loc = 'Plot AD157'
loc_dlg = {'interface': 3, 'values': ('Plot AD157', 'Plot AE200',
'Plot AE201', 'Greenhouse'),
'label': 'Measurement location', 'target': 'loc'}
```

The edit box configuration in *Figure 7-20* on page 7-17 is doing this Python equivalent:

```
abc = 4.5
abc_dlg = {'target': 'abc', 'description': 'Your best estimate',
'units': 'cm\u00b2',
'interface': 2, 'label': 'Damaged area', 'checkable': False,
'width': 0}
```

The checkable version of edit box does this:

```
abc = {'value': 4.5, 'checked': False}
abc_dlg = {'target': 'abc', 'description': 'Your best estimate',
'units': 'cm\u00b2',
```

```
'interface': 2, 'label': 'Damaged area', 'checkable': True, 'width': 0}
```

The radio button configuration in *Figure 7-21* on page 7-18 is doing this Python equivalent:

```
loc = 'Plot AD157'  
loc_dlg = {'interface': 8, 'values': ('Plot AD157', 'Plot AE200',  
'Plot AE201', 'Greenhouse'),  
'label': 'Where', 'target': 'loc'}
```

Things to consider

Some facts about **DIALOGS**:

- When a **DIALOG** statement is encountered in the BP, the dialog is presented. It remains on the screen until the user presses one of the displayed buttons. If no buttons are specified, there will be one **OK** button by default.
- While a **DIALOG** is being displayed, BP program execution remains on the BP's **DIALOG** step even if the user interacts with editable items (check boxes, edit boxes, etc.).
- BP variables editable by a **DIALOG** receive their updated values as they are edited. Including a Cancel button, for example, does not mean that pressing Cancel undoes any edits that may have been made. (You can support that behavior, of course, but you have to program for it outside of the **DIALOG** step.)

EXEC

EXEC performs the Python `exec()` function on the text in the 'Source' entry box, or on the file named by 'Source'. A simple example in *Figure 7-24* below illustrates: we use EXEC to define two variables.

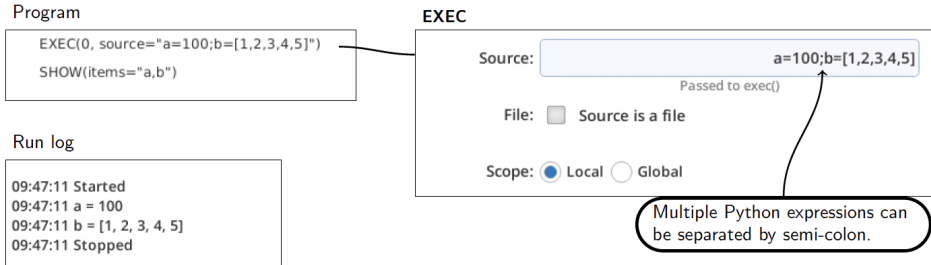


Figure 7-24. Defining two variables in an EXEC.

More complicated code can be put in a separate file and referenced by the EXEC. *Figure 7-25* below illustrates how to add this capability to a BP.

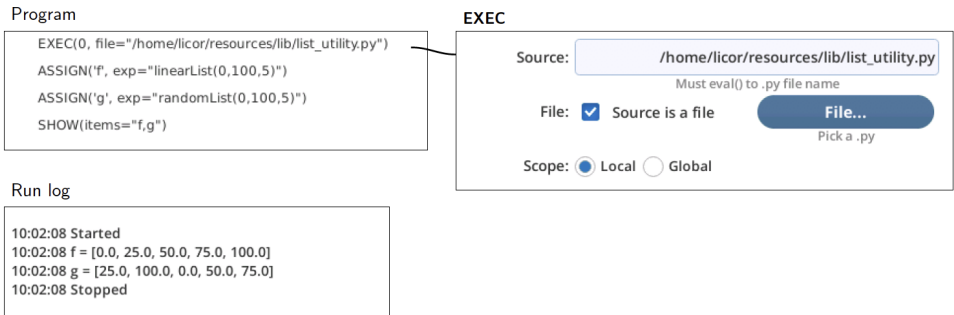


Figure 7-25. Adding extended functionality via EXEC.

Local vs global

If the **Scope** setting of an EXEC is **Local**, then variables defined in the EXEC are available only from the context in which the EXEC occurs. The **Global** option means whatever is defined in the EXEC is available anywhere (e.g., in any function) in the rest of the program. See *Figure 7-26* on the facing page.

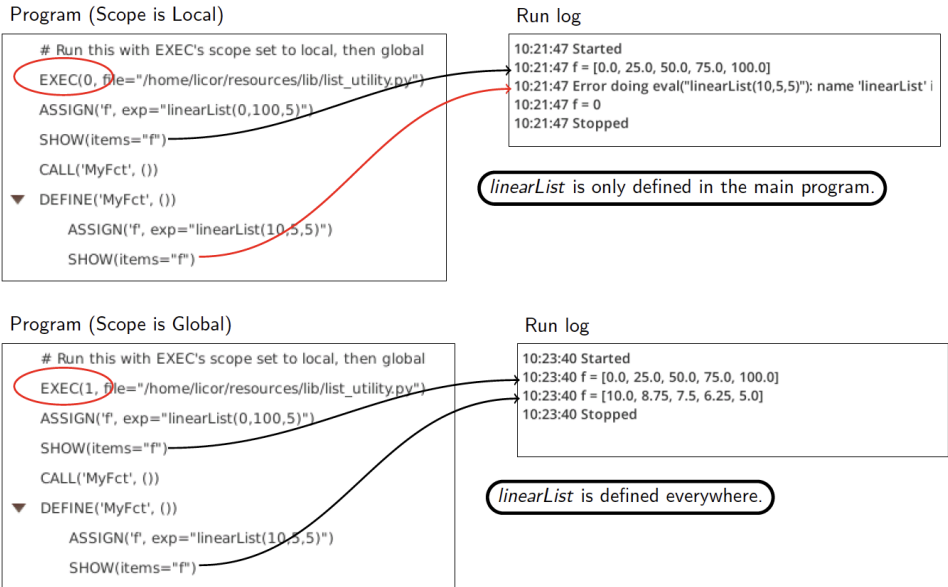


Figure 7-26 . Local (top) vs Global (bottom). The locally defined `linearList` is not available from within the `MyFct` subroutine.

GROUP

A **GROUP** is a container for program steps. It has an 'Enabled' property that determines whether or not the contained steps execute at run time. Thus, **GROUP** provides a convenient way to enable/disable sections of your program.

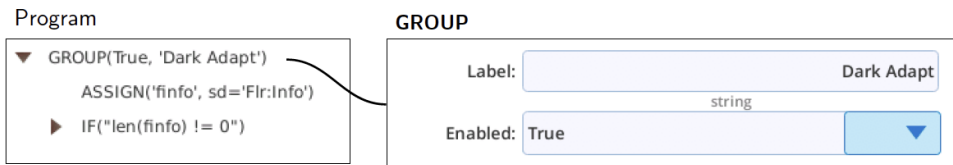


Figure 7-27. Example of an **GROUP** structure.

IF, ELSE IF, ELSE

The IF step can be made into any of three types: **IF**, **ELSE IF**, and **ELSE**.

Program

```

LOOP(list="(5,30,150,500)", var='f')
  IF("f<10")
    SHOW(string="{0} is < 10'.format(f)")
  ELIF("f<100")
    SHOW(string="{0} is < 100'.format(f)")
  ELIF("f<200")
    SHOW(string="{0} is < 200'.format(f)")
  ELSE()
    SHOW(string="{0} is an ELSE'.format(f)")
        
```

Run log

```

10:26:14 Started
10:26:14 5 is < 10
10:26:14 30 is < 100
10:26:14 150 is < 200
10:26:14 500 is an ELSE
10:26:14 Stopped
        
```

IF

Type: IF ELSE IF ELSE

Test: Must eval() to a boolean

ELSE IF

Type: IF ELSE IF ELSE

Test: Must eval() to a boolean

ELSE

Type: IF ELSE IF ELSE

Figure 7-28. Example of an IF structure.

While building a BP, there is nothing that ties any of these IF variations together, so it is up to you to arrange them appropriately.

- IF always marks the start of a new IF structure.
- After the IF, there can be as many ELSE IFs as you need.
- ELSE is optional, there can be only one, and it must come last.

If there is a misplaced ELSE or ELSE IF, then you will get an error message:

Program

```

▶ ELSE
        
```

Run Log

```

17:08:21 Error: ELSE or ELSE IF without IF
17:08:21 Stopped
        
```

LOG

The **LOG** statement cover all aspects of the normal LI-6800 log operations; it can open a log file, log a remark, log data, or close the file.

Open file

Figure 7-29 below illustrates opening a log file with a programmatically determined name. You can use **File...** to select an existing file, and modify the name in the edit box, or just type the name or expression into the edit box.

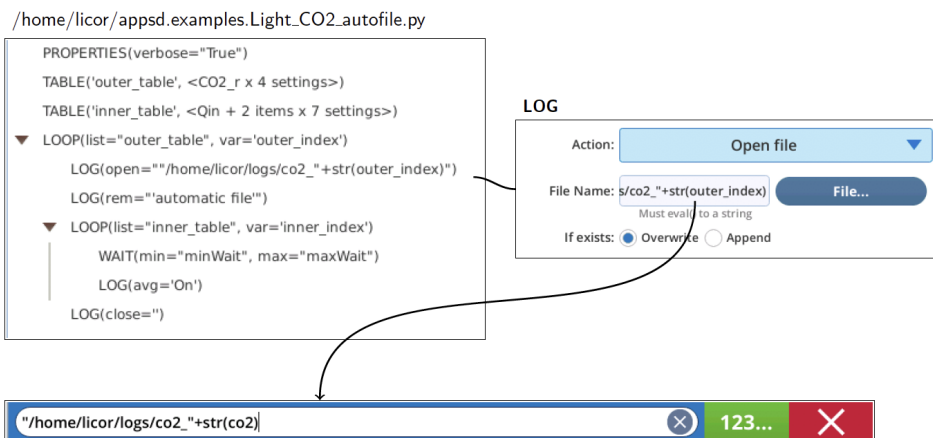


Figure 7-29 . Opening a log file with a programamntically determined name.

If a log file is already open when a **LOG [Open file]** is executed, the file will be closed first, and the new file opened.

Record remark

Adds a remark string to the file. This is skipped if no log file is open.

LOG

The screenshot shows a dialog box for the 'Record remark' action. It has two fields: 'Action:' with a dropdown menu set to 'Record remark', and 'Remark:' with a text input field containing the text 'automatic file'. Below the text input field, there is a small note that says 'Must eval() to a string'.

Figure 7-30. Logging a remark.

If you wish for some reason to retrieve the latest logged remark, it is available via **LOG>LastRem** in status dictionary.

Record data

LOG[RecordData] triggers a log event, or is just skipped if no log file is open. You can override the current log options for this event if you so choose.

LOG

The screenshot shows a dialog box for the 'Record data' action. It has five dropdown menus: 'Action:' set to 'Record data', 'Flr Event:' set to 'As set in flr log options', 'Flash Type:' set to 'As set in flr log options', 'Matching:' set to 'As set in log options', and 'Averaging:' set to 'On'.

Figure 7-31. LOG allows you to temporarily override log options.

Close file

The **LOG[Close file]** option takes no parameters. If no file is open when this is executed, it does nothing.

LOG

The screenshot shows a dialog box for the 'Close file' action. It has one dropdown menu: 'Action:' set to 'Close file'.

Figure 7-32. Closing a log file.

LOOP

LOOP contains steps that execute repeatedly. It comes in five types: Count, Duration, Control, List and File.

Count

If you wish to cycle a fixed number of times, use the **LOOP [Count]** option. You can define a variable to access the count (0, 1, ...). You can regulate the loop with 'Minimum time per cycle', otherwise it will be limited 0.1 seconds.

The image displays a software interface for configuring a LOOP step. It is divided into three main sections:

- Program:** Shows the configuration code: `LOOP(count="5", var='x')` and `SHOW(items="x")`. A curved arrow points from this code to the configuration window.
- Run log:** A text box containing the execution output:


```
12:56:22 Started
12:56:22 x = 0
12:56:22 x = 1
12:56:23 x = 2
12:56:23 x = 3
12:56:23 x = 4
12:56:23 Stopped
```
- LOOP Configuration Window:** A dialog box with the following settings:
 - Type:** A dropdown menu set to **Count**.
 - Count:** A text input field containing the value **5**. Below it is the text "(Must eval() to an integer)".
 - Index variable:** A text input field containing the value **x**. Below it is the text "(Optional) Variable".
 - Minimum time per cycle:** A text input field containing the value **0.1** with a "sec" label to its right. Below it is the text "(Optional) Must eval() to a number (default=0.1)".

Figure 7-33. Loop Count, 5 loops, index is x, and no timing regulation.

Duration

If you wish to cycle for some time duration, use the **LOOP [Duration]** option. You can define a variable that contains the number of seconds that program execution has been in the loop.

Program

```

LOOP(dur="5", units='Seconds', var='x', mininc="0")
  SHOW(items="x")
        
```

Run log

```

13:03:43 Started
13:03:43 x = 0.000213
13:03:43 x = 0.410153
13:03:44 x = 0.938826
13:03:44 x = 1.469727
13:03:45 x = 1.900386
13:03:45 x = 2.322318
13:03:46 x = 2.860055
13:03:46 x = 3.372189
13:03:47 x = 3.89833
13:03:47 x = 4.330667
13:03:48 x = 4.841978
13:03:48 Stopped
        
```

LOOP

Type: Duration

Duration: Seconds

Must eval() to a number

Elapsed time variable:

(Optional) Variable

Minimum time per cycle:

(Optional) Must eval() to a number (default=0.1)

Figure 7-34. LOOP [Duration], 5 seconds total, with timing via when new data is available (Min time = 0)

List

If you wish to cycle through a list of items of any type, use **LOOP [List]**. On each pass through, the associated variable that you name is assigned to an item from the list. It also works on **Table** variables.

Program

```
LOOP(list="(10,20+5,30,40)", var='x', mininc="1")
  SHOW(items="x")
```

Run log

```
13:10:19 Started
13:10:19 x = 10
13:10:20 x = 25
13:10:21 x = 30
13:10:22 x = 40
13:10:23 Stopped
```

LOOP

Type: **List**

List: (10,20+5,30,40)
Must eval() to a list or TABLE

Loop variable: x
Variable

Minimum time per cycle: 1 sec
(Optional) Must eval() to a number (default=0.1)

Figure 7-35. **LOOP [List]** example.

File

If you wish to process lines in a file, one at a time, use **LOOP [File]**. It a file, and executes the loop once for each linen in the file. You define a variable that will contain the line for each loop. If you wish, you can have the line parsed, in which case the variable will be a list of the items found.

`/home/licor/apps/system/tests/steps/LOOP_file_test.py`

```
GROUP(True, 'Tab delimited file')
  SHOW(string="** * Tab Delimited File * **")
  ASSIGN('file', exp="/home/licor/apps/system/tests/inputs/parsetest_tab.txt")
  SHOW(string="Parsing OFF")
  LOOP(file="file", var='x')
  SHOW(string="Parsing ON")
  LOOP(file="file", var='x', parse=True, delim='Tab')
  ASSIGN('tabok', exp="lastline[0] == '1...tline[2] == '1' and len(lastline)==3")
GROUP(True, 'Comma delimited file')
GROUP(True, 'Space(s) delimited file')
EXEC(0, source="BP.setResult('Pass' if tabok...)")
SHOW(items="tabok, commaok, spaceok, BP.getResult()")
```

LOOP

Type: **File**

File Name: file
Must eval() to a string

Parsing: Yes **Tab**
Parse each line? Parsing delimiter

Skip lines: 0
Must eval() to an integer

Line variable: x
Variable

Minimum time per cycle: 1 sec
(Optional) Must eval() to a number (default=0.1)

Figure 7-36. **Configuring LOOP [File]**.

See `/home/licor/apps/system/tests/LOOP_file_test.py`. This program parses three sample input files, listed below (spaces and tabs are shown as printable characters in the space and tab files). The files reside in `/home/licor/apps/system/tests/inputs/`.

parsetest_comma.txt

```
100,20,5
1000,21,4
1500,22,3
200,23,2
100,24,1
```

parsetest_space.txt

```
100_20_5
1000_21_4
1500_22_3
200_23_2
100_24_1
```

parsetest_tab.txt

```
100    20    5
1000   21    4
1500   22    3
200    23    2
100    24    1
```

Run log

```
16:58:37 Started
16:58:37 *** Comma delimited file ***
16:58:37 Parsing OFF
16:58:37 x = 100,20,5
16:58:38 x = 1000,21,4
16:58:38 x = 1500,22,3
16:58:38 x = 200,23,2
16:58:38 x = 100,24,1
16:58:38 Parsing ON
16:58:38 x = ['100', '20', '5']
16:58:38 x = ['1000', '21', '4']
16:58:38 x = ['1500', '22', '3']
16:58:38 x = ['200', '23', '2']
16:58:38 x = ['100', '24', '1']
16:58:38 Stopped
```

Run log

```
16:59:36 Started
16:59:36 *** Space(s) delimited file ***
16:59:36 Parsing OFF
16:59:36 x = 100 20 5
16:59:36 x = 1000 21 4
16:59:36 x = 1500 22 3
16:59:36 x = 200 23 2
16:59:36 x = 100 24 1
16:59:36 Parsing ON
16:59:36 x = ['100', '20', '5']
16:59:36 x = ['1000', '21', '4']
16:59:36 x = ['1500', '22', '3']
16:59:36 x = ['200', '23', '2']
16:59:36 x = ['100', '24', '1']
16:59:36 Stopped
```

Run log

```
17:00:05 Started
17:00:05 *** Tab Delimited File ***
17:00:05 Parsing OFF
17:00:05 x = 100    20    5
17:00:05 x = 1000   21    4
17:00:05 x = 1500   22    3
17:00:06 x = 200    23    2
17:00:06 x = 100    24    1
17:00:06 Parsing ON
17:00:06 x = ['100', '20', '5']
17:00:06 x = ['1000', '21', '4']
17:00:06 x = ['1500', '22', '3']
17:00:06 x = ['200', '23', '2']
17:00:06 x = ['100', '24', '1']
17:00:06 Stopped
```

Figure 7-37. LOOP [List] example.

Note that in each case, the unparsed line is always a string, and the parsed pieces are always strings. Turning a string into a number in Python is easy (`float()` function), but you may not need to. If you were using these values to set a control, the **SETCONTROL** controls will do the conversion for you.

PROPERTIES

The **PROPERTIES** lets you programmatically pause a BP, and also controls the BP's verbosity (run log output).

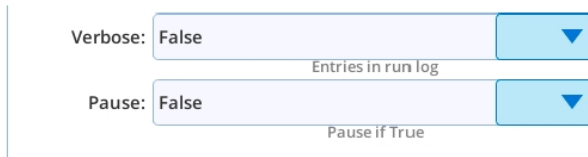


Figure 7-38. Using the **PROPERTY** step to pause a program.

Verbose, which defaults to **off**, controls how much gets written to the run log. With **Verbose** set to **False**, the only output to the run log will be start and stop messages, **SHOW** output, and warnings and errors. With **verbose** set to **True**, nearly every step will contribute to the run log.

If the **Pause** property is true, the BP will pause when it encounters that step while running. This is useful for debugging purposes (see *Debug mode* on page 6-14), or indefinitely pausing a program until the user wishes to resume.

RETURN

The **RETURN** step exits a **DEFINE**, or exits the main program.

There are no parameters associated with **RETURN**. If you wish to return a value from a **DEFINE**, use the pass-by-reference feature for a passed parameter (see *Passing by value or reference* on page 7-13).

RUN

RUN is how a BP can launch another BP.



Figure 7-39. A BP that launches two other BPs.

SETCONTROL

SETCONTROL allows you to set a control or system constant. The button in the **Target** line will access the control dictionary. The **Value** line will have slightly different looks depending upon the control.

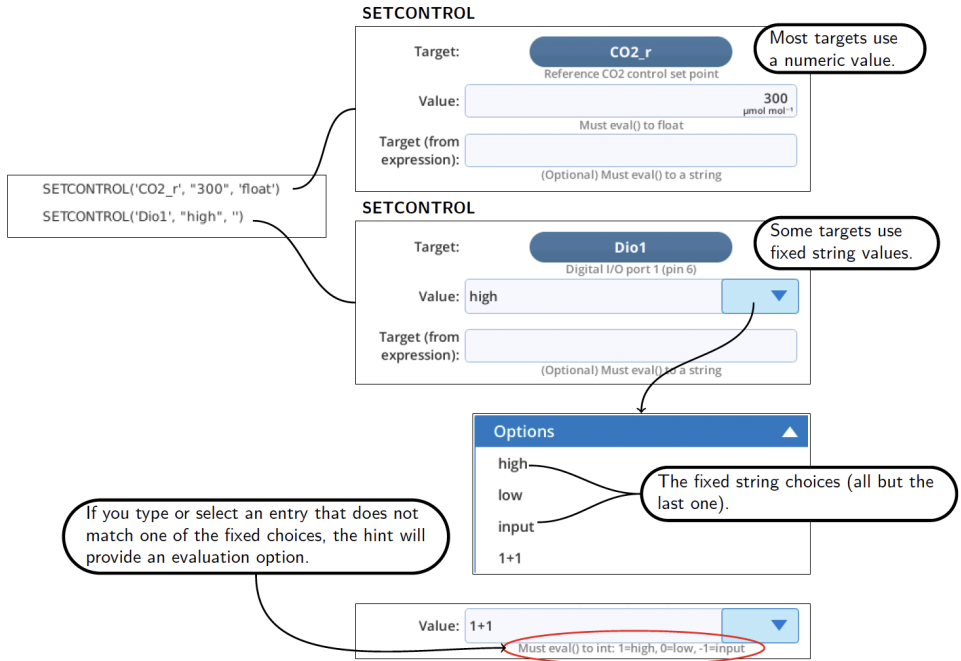


Figure 7-40. The SETCONTROL interface varies with target.

If you wish to select the target at run time instead of at design time, then use the Target (from expression) field. This will override whatever the button label is. For example, if you wish to read targets and values from a file, you could put a SETCONTROL in a loop that reads the `le`, and each pass through the loop sets target to `x[0]` and value to `x[1]`, assuming `x` held the parsed line, and each line held 2 items: target (string) and value (number or expression).

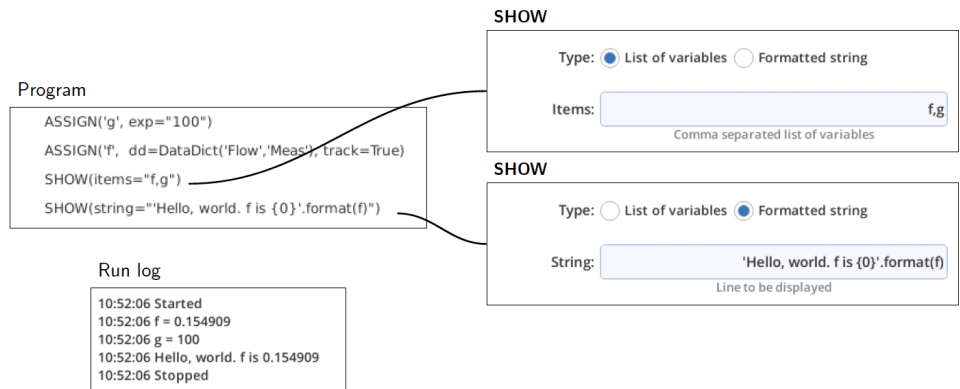
When the 'Value' is a ComboBox, as in the Dio1 example above, the non-fixed entry allows you to select the choice at run time, rather than design time. Table 7-1 on the facing page illustrates some entry choices for Dio1.

Table 7-1. Options for entering SETCONTROL Dio1.

Typed in Value	Result
high	At design time, same as selecting high from the menu.
'high'	At run time, evaluates to a string, so becomes high.
0	At run time, sets Dio1 like you had picked low.
xxx	At run time, assuming you have defined xxx, and it evaluates to 'high', 'low', 'input', 1, 0, or -1, then this is fine.
'in'+ 'put'	At run time, evaluates to a string ('input'), so same as input.
99-100	At run time, evaluates to -1, so same as input.

SHOW

SHOW prints to the run log. You can specify a list of variables, or specify a formatted string.

*Figure 7-41. Using SHOW.*

TABLE

A **TABLE** is a named control table that can be executed by making it the target of a **LOOP** [List].

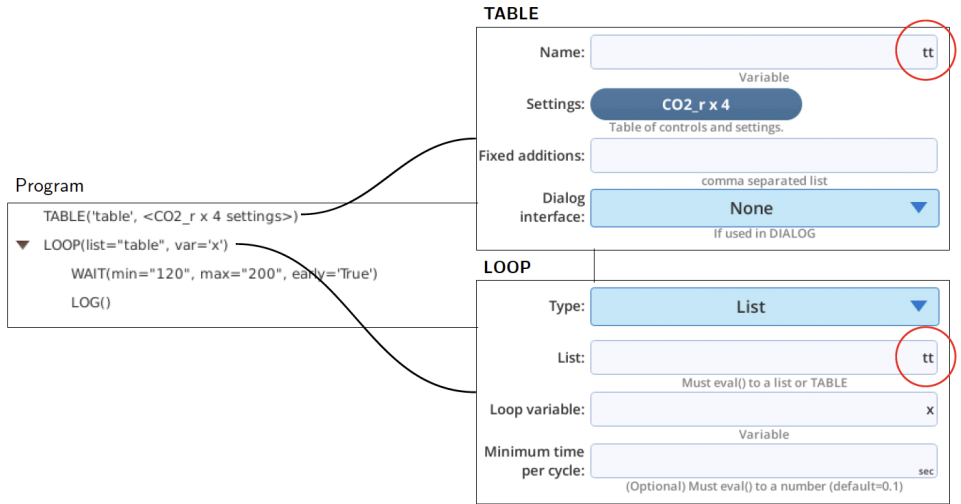


Figure 7-42. **TABLE** are executed in a **LOOP** [list].

The **LOOP** will loop over the number of settings found in the top row of the table. Each pass through, it sets the table's row targets (top to bottom), to the value in that row's column. If an entry is blank, it skips that target. Thus, if only the first column in the table for a particular row has an entry, that value will only be set once.

Structure of a TABLE variable

Consider a **TABLE** with two controls, one for light intensity, and one color, and an auxiliary constant for wait times (Figure 7-43 below).

Program

```
TABLE('table', <Qin + 2 items x 4 settings>)
```

TABLE

Name: Variable

Settings: Table of controls and settings.

Fixed additions: comma separated list

wait: units format

Dialog interface: If used in DIALOG

Row	Col	Control	1/4	2/4	3/4	4/4	
1		Qin <small>$\mu\text{mol m}^{-2}\text{s}^{-1}$</small>	1500.0	1000.0	500.0	100.0	⊖
2		Color_Qin	r60	r70	r80	r90	⊖
3		wait <small>mins</small>	2.00	3.00	4.00	5.00	

Delete Insert Row Edit Row Cancel OK

Figure 7-43. A table with two controls and an auxiliary constant.

A variable *table* is a Python dictionary with two keys:

- "values" - a list, each item is a tuple containing a string (the target) and a list (list of set points).
- "aux" - a dict, with keys corresponding to the strings (targets) in the "values" list. The keys for each item are:
 - "control": True or False
 - "staticmeta": True or False. If True, there will also be keys for "units" and "format".

```
{
  "values": [
    [
      "Qin", [1500, 1000, 500, 100]
    ],
    [
      "Color_Qin", ["r60", "r70", "r80", "r90"]
    ],
    [
      "wait", [2, 3, 4, 5]
    ]
  ],
  "aux": {
    "Color_Qin": {
      "control": True,
      "staticmeta": False
    },
    "wait": {
      "units": "mins",
      "format": ["f", 1, 2],
      "control": False,
      "staticmeta": True
    },
    "Qin": {
      "control": True,
      "staticmeta": False
    }
  }
}
```

Listing 7-2. An example of a TABLE variable.

Custom executions

If you need to customize how a **TABLE** is handled during run time, then you should start with *Figure 7-44* below which illustrates the functional equivalent of the standard processing that **LOOP [List]** provides.

Standard method of executing a **TABLE**

```
TABLE('table', <Qin + 2 items x 4 settings>)
▼ LOOP(list="table", var="")
    WAIT(dur="wait", units='Seconds')
```

Equivalent: /home/licor/apps/examples/table_execute.py

```
TABLE('table', <Qin + 2 items x 4 settings>)
SHOW(items="table")
▼ LOOP(count="len(table['values'])[0][1]", var='col')
    ▼ LOOP(list="table['values']", var='row')
        ASSIGN('target', exp="row[0]")
        ASSIGN('value_list', exp="row[1]")
        ▼ IF("col < len(value_list)")
            SHOW(string="{0} to {1}".format(target, value_list[col]))
            ▼ IF("table['aux'][target]['control']")
                SETCONTROL("target", "value_list[col]")
            ▼ ELSE()
                EXEC(0, source="BP.registerThis(target, valu...")
        WAIT(dur="wait", units='Seconds')
```

See the table definition.

Loop over the number of values in the first row. *col* = 0, 1, 2, ...

Loop over the rows. *row* = ["Qin", [1500, 1000, 500, 100]], ...

Enough values in this list?

Is this a control?

Set the value (*value_list[col]*) of a variable (*target* = "wait").

Run log

```
17:34:02 Started
17:34:02 table = {'aux': {'Color_Qin': {'control': True, 'staticmeta'
17:34:02 Qin to 1500
17:34:02 Color_Qin to r60
17:34:02 wait to 2
17:34:04 Qin to 1000
17:34:05 Color_Qin to r70
17:34:05 wait to 3
17:34:07 Qin to 500
17:34:07 Color_Qin to r80
17:34:07 wait to 4
17:34:09 Qin to 100
17:34:09 Color_Qin to r90
17:34:09 wait to 5
17:34:11 Stopped
```

The value of *table*, all on one line.

Figure 7-44. A template program for how to explicitly program the execution of a table.

WAIT

WAIT has three types: **Duration**, **Stability**, **Until** a time of day, and for an **Event** or condition. All but the last type of wait will put a countdown on the **Start** screen and/or **Monitor** screen (when selected).



Figure 7-45. Wait countdown.

Duration

WAIT [Duration] suspends execution for the specified amount of time.

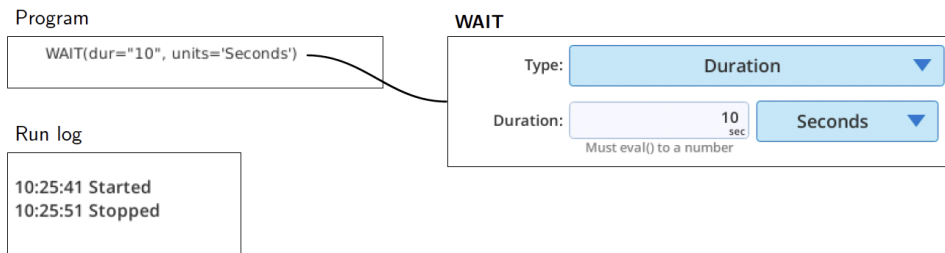


Figure 7-46. Wait duration.

Stability

WAIT [Stability] has two parts. Part 1 is the minimum time. From the end of that until the maximum time, the wait will be terminated if the current stability definition is met. In the interval between 30 seconds after the start of the wait, and 60 seconds before the maximum time, a match will occur if **Early matching** is **True**, and if the reference IRGAs (C_r and H_s) meet the following stability criteria:

$$\left| \frac{\partial C_r}{\partial t} \right| < 1 \frac{\mu\text{mol/mol}}{\text{min}}, \quad \left| \frac{\partial H_s}{\partial t} \right| < 0.1 \frac{\text{mmol/mol}}{\text{min}} \quad 7-1$$

The image shows a configuration window for a 'WAIT' step. The 'Type' is set to 'Stability'. The 'Minimum' is 60 seconds and the 'Maximum' is 120 seconds. The 'Early Matching' is set to 'False'. A line connects the 'WAIT(min="60", max="120")' command in the 'Program' box to the configuration window. Below, the 'Run log' shows the execution details.

Program

```
PROPERTIES(verbose="True")
WAIT(min="60", max="120")
```

WAIT

Type: **Stability**

Minimum: **60** secs
Must eval() to a number

Maximum: **120** secs
Must eval() to a number

Early Matching: **False**

Run log

```
17:57:07 Started
17:57:07 Stability Wait part 1: 60.0 secs
17:57:12 Wait ended by user
17:57:12 Stability Wait part 2: 60.0 secs or until stable
17:57:15 Wait ended by user
17:57:15 Stopped
```

Figure 7-47. Wait [stability] example.

Until

WAIT [Until] waits for a time of day. Figure 7-48 on the next page shows multiple ways to achieve the same thing.

Program

```
PROPERTIES(verbose="True")
WAIT(until=(15,30,0))
```

WAIT

Type: Until

Format: Explicit String

Time:
hh (0-23) mm ss

Date: Yes

Program

```
PROPERTIES(verbose="True")
WAIT(until=(15,30,0), date=(2019,6,11))
```

WAIT with explicit date

Type: Until

Format: Explicit String

Time:
hh (0-23) mm ss

Date: Yes Jun
yyyy Month dd

Program

```
PROPERTIES(verbose="True")
WAIT(until="15.5")
```

A variety of formats, including variable names, can be used here.

WAIT from string

Type: Until

Format: Explicit String

[Date] time:
Must eval() to time or formatted datetime.

Format:
Python strptime format (optional).

Run log

```
10:54:36 Started
10:54:36 WAIT until Tue Jun 11 15:30:00 2019
```

Figure 7-48. Wait [until] example.

If no date is specified and the specified time has passed when the statement executes, the program will wait until the next day.

The ‘String’ format (lower left in Figure 7-48 above) is an alternative way to specify time and date; the advantage is that allows you to programmatically specify it if you

choose. To include a date, specify the `strptime`¹ in the 'Format' box. Otherwise, just specify the string as decimal hours, or hh:mm, as illustrated in *Table 7-2* below.

Table 7-2. Some examples of specifying time and date. The format string is not really needed unless you wish to specify a date along with the time.

[Date] time	Format	Interpreted as
10		10:00:00 am
5.5		05:30:00 am
14:22		14:22:00 pm
8:30:6		08:30:06 am
6 Dec 2019 12:33:45	%d %b %Y %H:%M:%S	6 Dec 2019 12:33:45

Event

WAIT [Event] waits until the test you have specified becomes True. The example in *Figure 7-49* on the next page is as close as you may come to a game on the LI-6800. It simply times how long it takes you to get the flow rate² below 400.

¹See <http://pubs.opengroup.org/onlinepubs/007904975/functions/strptime.html>

²Cheat: you don't have to do this via software or touch screen: you could just reach over and unplug the hose going to the sensor head.

```

/home/licor/apps/system/tests/WAIT_event_test.py
ASSIGN('flow', dd=DataDict('Flow','Meas'), track=True)
SETCONTROL('SysConst:AvgTime', "0", 'float')
SETCONTROL('Pump', "auto", "")
SETCONTROL('Flow', "500", 'float')
SHOW(string="Waiting for flow to get above 400")
WAIT(event="flow > 400")
ASSIGN('start', exp="datetime.now()")
SHOW(string="** * Your Test: Make flow go below 100 * * ")
SHOW(string="You are being timed!")
# Flow ON
WAIT(event="flow < 100")
ASSIGN('YourTime', exp="(datetime.now() - start).total_seconds()")
SETCONTROL('SysConst:AvgTime', "4", 'float')
SHOW(string="Your time = {0} secs'.format('YourTime'))

```

Run log

```

11:31:05 Started
11:31:05 Waiting for flow to get above 400
11:31:08 ** * Your Test: Make flow go below 100 * *
11:31:08 You are being timed!
11:31:17 Your time = 8.652296 secs
11:31:17 Stopped

```

WAIT

Type: Event

Event: flow > 400
Must eval() to a boolean

WAIT

Type: Event

Event: flow < 100
Must eval() to a boolean

Figure 7-49. The Wait [Event] example.

WHILE

WHILE loops while the specified condition is True. The ‘Minimum time per cycle’ specifies the minimum time interval that will separate each cycle through the loop. Default is 0.1 seconds. Setting this to 0 means it will cycle when the next data set become available (nominally every 0.5 sec).

The example in *Figure 7-50* on the facing page will loop until the sample cell flow rate exceeds 20 $\mu\text{mol}/\text{sec}$, so if you have the chamber closed and the pump running before running this program, it will loop until you either open the chamber, or stop the flow, or 1 minute has expired.

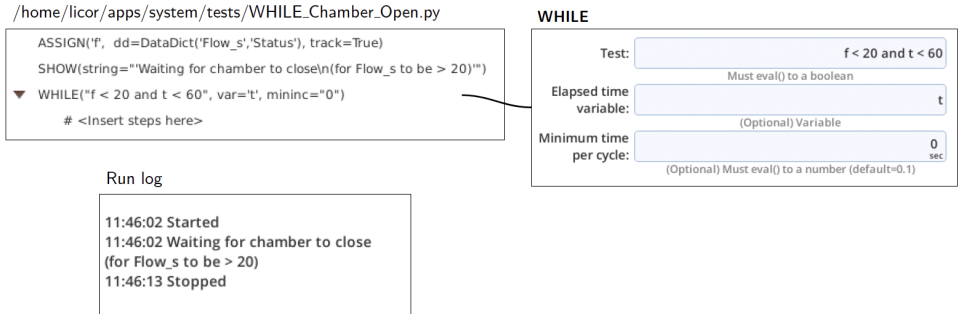


Figure 7-50. WHILE example.

Appendix A.

Control dictionary map

The table below shows the control that can be set via SETCONTROL

Type	Group	Name	Description
Auxiliary	Analog	AuxPwr	Voltage for the auxiliary power port in head
		Dac1	D/A #1 (pin 2) setting
		Dac2	D/A #2 (pin 3) setting
		Dac3	D/A #3 (pin 4) setting
		Dac4	D/A #4 (pin 5) setting
	Digital	ADC1Pullup	ADC Channel 1 pullup (pin 21)
		Dio1	Digital I/O port 1 (pin 6)
		Dio2	Digital I/O port 2 (pin 7).
		Dio3	Digital I/O port 3 (pin 8).
		Dio4	Digital I/O port 4 (pin 9).
		Dio5	Digital I/O port 5 (pin 10).
		Dio6	Digital I/O port 6 (pin 11).
		Dio7	Digital I/O port 7 (pin 12).
		Dio8	Digital I/O port 8 (pin 13).
		Excite5	5V Excitation (pin 25).
		Power12	12V Power (pin 23).
		Power5	5V Power (pin 14).

Type	Group	Name	Description	
Constants	Fluorometry	FLR:Adark	Dark photosynthetic rate	
		FLR:Fm	Value of dark adapted Fm	
		FLR:Fo	Value of dark adapted Fo	
		FLR:PS2/1	Photosystem distribution factor	
	Gas Exchange	Const:Geometry	Leaf type	
		Const:Custom	1-sided BLC for custom	
		Const:K	Stomatal ratio	
	Leaf Temp	Const:S	Leaf area	
		LTCConst:deltaTw	Wall temperature difference from air temp (for energy balance)	
		LTCConst:fT1	Fractional contribution of leaf thermocouple T1	
		LTCConst:fT2	fractional contribution of leaf thermocouple T2	
	Soil	LTCConst:fTeb	Fractional contribution of leaf energy balance	
		Soil:#Reps	Soil measurement rep count per measurement	
		Soil:CircPump_%	Circulation pump speed	
	System	Soil:Duration	Soil measurement duration	
		SysConst:AvgTime	System average time	
	User Defined	SysConst:Oxygen	Oxygen concentration	
		User:myfirst		
			User:mysecond	

Type	Group	Name	Description	
Env Controls	CO2	CO2 On/Off	CO ₂ Controller on/off	
		CO2_%	CO ₂ injector setting (0-100)	
		CO2_r	Reference CO ₂ control set point	
		CO2_s	Sample CO ₂ control set point	
	Color	Color_All	Color specifier for all sensors	
		Color_Con	Color specifier for light source attached to console	
		Color_Flr	Color specifier for fluorometer	
		Color_Head	Color specifier for light source attached to head	
		Color_Qin	Color specifier for all sensors contributing to the leaf	
		Fan	Fan On/Off	Fan Controller on/off
			Fan_%	Fan speed power (0=off, 100=full)
	Fan_blc		Boundary layer conductance set point (controlled by fan speed)	
	Flow	Fan_rpm	Fan speed set point	
		Flow	Flow rate to chamber	
		Flow On/Off	Flow Controller on/off	
		Flow_%	Flow split setting (% to leaf chamber)	
	H2O	Pump	Pump speed setting	
		Desiccant_%	Direct control of desiccant tube (0=bypass, 100=full scrub)	
		H2O_On/Off	H ₂ O Controller on/off	
		H2O_%	Direct H ₂ O control from full dry (-100) to full humidity (100)	
		H2O_r	H ₂ O Reference set point	
		H2O_s	H ₂ O Sample set point	
		Humidifier-%	Direct control of humidifier tube (0=bypass, 100=full wetting)	
		RH_air	Chamber relative humidity set point	
		SD_air	Saturation Deficit (air) set point	
		VPD_leaf	Vapor Pressure Deficit (leaf) set point	

Type	Group	Name	Description
	Light	Q_All	All light source(s) set point
		Q_Console	Light source attached to console set point
		Q_Flr	Fluorometer actinic set point
		Q_Head	Light source attached to head set point
		Qin	Light incident on leaf set point
	Pressure	Pressure	Chamber over-pressure pressure set point
		Pressure_On/Off	Pressure Controller on/off
		Pressure_%	Chamber over-pressure control setting
	Temp	Tair	Chamber air temperature set point
		Temp_On/Off	Temp Controller on/off
		Tleaf	Leaf temperature set point
		Txchg	Heat exchanger set point

Type	Group	Name	Description	
Flr Settings	Dark	DARK:After	Far red off time after actinic off	
		DARK:Before	Turn far red on time prior to actinic off	
		DARK:Duration	Dark pulse duration	
	Induction	DARK:FarRed target	Dark pulse far red target	
		IND:Duration	Induction flash duration	
	MPF	IND:Red target	Induction flash red target	
		MPF:Phase 1	MPF phase 1 duration	
		MPF:Phase 2	MPF phase 2 duration	
		MPF:Phase 3	MPF phase 3 duration	
		MPF:Ramp	MPF ramp control	
	Measure	MPF:Red target	MPF red target value	
		Meas:AverageTime	Averaging time for Fs and Fs'	
		Meas:DarkModRate	Modulation rate for dark measurements	
		Meas:FlashModRate	Modulation rate during flash events	
		Meas:LightModRate	Modulation rate for light measurements	
	RF	Meas:Modulation	Modulation control:	
		Meas:Recording	Turn flr recording on/off	
		RF:Duration	Rectangular flash duration	
	Log Options	Flr	RF:Red target	Rectangular flash red target value
			FlrOpt:Action	Fluorometer Action at Log
FlrOpt:Auto			Threshold for Automatic MPF	
Match		FlrOpt:FlashType	Fluorometer Flash Type	
		FlrOpt:MinFlash	Minimum flash interval	
		MchOpt:CO2Change	Match if CO2_r changed >	
		MchOpt:CO2Delta	Match if CO2_r - CO2_s <	
		MchOpt:Choice	Match when logging?	
Std		MchOpt:Elapsed	Match if elapsed time by	
		MchOpt:H2ODelta	Match if H2O_r - H2O_s <	
		MchOpt:H2OChange	Match if H2O_r changed >	
		LogOps:MakeExcel	Also create Excel log file	
		LogOps:AvgTime	Additional averaging time	
		LogOps:Beep	Beep on log	

Type	Group	Name	Description
Misc	Match Mode	Mch:AvgTime	Auto match stats averaging time
		Mch:CO2 limit	CO ₂ delta rate of change green light threshold
		Mch:H2O limit	H ₂ O delta rate of change green light threshold
		Mch:Mode	Match mode action
		Mch:Timeout	Auto mode timeout time
	Power Settings	PowerState	Sleep/Standby mode control

Appendix B.

Status dictionary map

The table below shows the status values that can be monitored via ASSIGN.

Type	Group	Name	Description
Auxiliary	Analog	AuxPwr	Voltage for the auxiliary power port in head
		Dac1	D/A #1 (pin 2) setting
		Dac2	D/A #2 (pin 3) setting
		Dac3	D/A #3 (pin 4) setting
		Dac4	D/A #4 (pin 5) setting
	Digital	ADC1Pullup	ADC Chan1 pullup (pin 21)
		Dio1	Pin 6: low, high, input
		Dio2	Pin 7: low, high, input
		Dio3	Pin 8: low, high, input
		Dio4	Pin 9: low, high, input
		Dio5	Pin 10: low, high, input
		Dio6	Pin 11: low, high, input
		Dio7	Pin 12: low, high, input
		Dio8	Pin 13: low, high, input
		Excite5	5V excitation (pin 25)
		GPIO	State summary (pins 13-6)
		GPIOdir	Direction summary (pins 13-6)
		Power12	12V power (pin 23)
		Power5	5V power (pin 14)

Type	Group	Name	Description
Env Controls	CO2	CO2:Label	CO2_r, CO2_s, or blank
		CO2:Percent	CO ₂ injector target (if manual) %
		CO2:Scrub	auto, on, off
		CO2:SetPoint	CO ₂ setpoint (if auto) mol mol
		CO2:Status	0=off, 1=manual, 2=off target, 3=on target
	Fan	Fan:Percent	Manual set pointmol s
		Fan:SetPoint	Automatic target (target units)
		Fan:SetPoint_rpm	Automatic target (rpm)
		Fan:Status	0=off, 1=manual, 2=off target, 3=on target
		Fan:Target	(if auto) RPM or BLC
	Flow	Flow:Percent	Flow setpoint (if manual) mol s
		Flow:Pump	auto, high, medium, low, minimum, off
		Flow:SetPoint	Flow setpoint (if auto) mol s
		Flow:Status	0=off, 1=manual, 2=off target, 3=on target
	H2O	H2O:PercentD	% Desiccant (manual)
		H2O:PercentH	% Humidifier (manual)
		H2O:SetPoint	H2O setpoint (target units)
		H2O:Status	0=off, 1=manual, 2=off target, 3=on target
		H2O:Target	H2O_r, H2O_s, RH air, etc.
		H2O:Teff	Coollest temp in chamber
		H2O:TlowLab	Label of coolest chamber temp

Type	Group	Name	Description
Env Controls (continued)	Light	Con:ColorMix	Mix resulting from color spec
		Con:ColorSpec	Color specification
		Con:Control	off, setpoint, percent or test
		Con:Info	Console light source info
		Con:Percent	Manual %s: red blue farred
		Con:Setpoint	Actinic setpoint mol m s
		Con:Status	Con: 0=off, 1=manual, 2=off target, 3=on target
		Con:Trans	Transmittance
		Flr:ColorMix	Mix resulting from color spec
		Flr:ColorSpec	Color specification
		Flr:Control	off, setpoint, percent or test
		Flr:Info	Flr info
		Flr:Percent	Manual %s: red blue farred
		Flr:Setpoint	Actinic setpoint mol m s
		Flr:Status	Flr: 0=off, 1=manual, 2=off target, 3=on target
		Flr:Trans	Transmittance
		Head:ColorMix	Mix resulting from color spec
		Head:ColorSpec	Color specification
		Head:Control	off, setpoint, percent or test
		Head:Info	Head lightsource info
		Head:Percent	Manual %s: red blue farred
		Head:Setpoint	Actinic setpoint mol m s
		Head:Status	Head: 0=off, 1=manual, 2=off target, 3=on target
		Head:Trans	Transmittance

Type	Group	Name	Description	
Env Controls (continued)	Pressure	Press:Percent	Press setpoint (if manual) mol s	
		Press:SetPoint	Flow setpoint (if auto) mol s	
		Press:Status	0=off, 1=manual, 2=off target, 3=on target	
	Temp	Temp:Hold	Tleaf control suspended (chamber open)	
		Temp:SetPoint	Automatic set point	
		Temp:Status	0=off, 1=manual, 2=off target, 3=on target	
		Temp:T2use	0=none or out, 1=in, 2=avg	
		Temp:Target	Txchg, Tair, Tleaf	
		Temp:TleafOp	Tleaf control option value	
		Temp:TleafOpID	Tleaf control option ID	
	Log Options	Flr	FlrOpt:Action	0=None, 1=Flash, 2=Flash+Dark
			FlrOpt:Auto	Threshold for Automatic MPF
			FlrOpt:FlashType	0=Auto, 1=RF, 2=MPF, 3=Ind
			FlrOpt:MinFlash	Minimum flash interval
Match		MchOpt:CO2Change	Match opt: CO2 changed >	
		MchOpt:CO2Delta	Match opt: CO2 <	
		MchOpt:Choice	Never match / Always match / Only match if	
		MchOpt:ChoiceNum	Match: 0=Never, 1=Always, 2=If	
		MchOpt:Elapsed	Match opt: elapsed time >	
		MchOpt:H2OChange	Match opt: H2O changed >	
		MchOpt:H2ODelta	Match opt: H2O <	
Standard		LogOps:MakeExcel	Also create Excel log file	
		LogOps:AvgTime	Additional averaging time	
		LogOps:Beep	Beep on log	

Type	Group	Name	Description
Matching	Auto Config	Mch:AvgTime	Auto match stats averaging time
		Mch:CO2 limit	CO ₂ delta rate of change green light threshold
		Mch:H2O limit	H ₂ O delta rate of change green light threshold
	Status	Mch:Timeout	Auto mode timeout time
		Mch:AutoFrac	Fraction complete of an automatic match
		Mch:LogLevel	Match label (shown on log button)
		Mch:Message	Result of previous match
		Mch:State	0=inactive, 1=manual, 2=automatic
		Mch:ddCdt	CO ₂ stability during match
		Mch:ddHdt	H ₂ O stability during match
Misc	Logging	LOG:FileName	Name of current log file
		LOG:FileTS	Timestamp of last opened log file
		LOG:IsFileOpen	Is a log file open?
		LOG:LastRem	Latest logged remark
		LOG:ObsCount	Observations logged
		LOG:State	Is a log event active?
	Power	PowerState	on, standby, sleep
		PowerValue	0=On, 1=Standby, 2=Sleep
	Stability	Stab:Stable	Number of items checked for stability
		Stab:State	Stability state: Stable/Total
		Stab:Total	Number of items stable

Appendix C.

The `list_utility` module

The following is a listing of the `list_utility.py` file on the LI-6800.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Thanks to Ari Kornfeld @ akorn@carnegiescience.edu

import numpy as np
from random import sample

def linearList(vv1, vv2, nn, rounded=2):
    # v1 - starting value,
    # v2 - ending value
    # n - number of values
    v1 = float(vv1)
    v2 = float(vv2)
    n = int(nn)
    v_range = [f for f in list(np.linspace(0, 1, num=n))]
    setpoints = [v1 + f*(v2-v1) for f in v_range]
    return list(np.around(setpoints, rounded))

def randomList(v1, v2, n, rounded=2):
    return sample(linearList(v1, v2, n, rounded=rounded),n)

def makeOrtho(listOfLists, lock_index=-1, max_cor=0.1, outfile=''):
    # listOfLists should contain 2 or more lists of setpoints. If
    # sizes not equal, smallest size is n
    # returns listOfLists with setpoint sorted to provide maximum
    # orthogonality (minimum correlation)
    # if you don't want one of the lists sorted, specify that index
    # as lock_index. 0 = first list, 1 = 2nd, etc.
    listCount = len(listOfLists)
```

```
n = np.min([len(x) for x in listOfLists]) # n is smallest list
size
print(n)
iter = 0
if max_cor < 0.05:
    max_cor = 0.05
orthogonal_enough = False
p = lambda i: sample(listOfLists[i], n) if i != lock_index else
listOfLists[i][0:n]
while not orthogonal_enough:
    result = np.matrix([p(i) for i in range(listCount)])
    #to test bad ortho: data.frame(T=T_range, C=C_range, Q = Q_
range)
    # now check that the variables aren't too strongly
correlated
    cor1 = np.corrcoef(result);
    np.fill_diagonal(cor1, 0.0) # we don't care about self
correlation
#     print(cor1)
    cc = np.max(np.abs(cor1))
#     print("cc=", cc)
    orthogonal_enough = (cc < max_cor)
    iter += 1
    if (iter > 500):
        max_cor = 0.2 # safety valve
print(iter, "iterations, cc=", cc)

if outfile != '':
    try:
        file = open(outfile, 'w')
        print('corr_coeff=', cc, file=file)
        for i in range(n):
            line = ''
            for j in range(listCount):
                line += str(result[j,:].tolist()[0][i])+' '
            print(line, file=file)
    except Exception as e:
        print('Exception in makeOrtho:', str(e))

    return [result[i,:].tolist()[0] for i in range(listCount)] #
return a list of lists
```

```
if __name__ == '__main__':  
    n=12  
    t = linearList(25, 15, 3) + linearList(17, 45, n-3) # make temp  
    efficient to work through - no big jumps, and start at ambient.  
    c = linearList(50, 1000, n, rounded=0) # test size mismatch  
    q = linearList(20, 2000, n, rounded=0)  
    (t,c,q) = makeOrtho((t,c,q), lock_index=0,  
outfile="/Users/jon/out.txt")  
    print(t)  
    print(c)  
    print(q)
```

Listing 10-1. Listing of /home/licor/resources/lib/list_utility.py.

LI-COR Biosciences

4647 Superior Street
Lincoln, Nebraska 68504
Phone: +1-402-467-3576
Toll free: 800-447-3576 (U.S. and Canada)
envsales@licor.com

Regional Offices**LI-COR Biosciences GmbH**

Siemensstraße 25A
61352 Bad Homburg
Germany
Phone: +49 (0) 6172 17 17 771
envsales-gmbh@licor.com

LI-COR Biosciences UK Ltd.

St. John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
United Kingdom
Phone: +44 (0) 1223 422102
envsales-UK@licor.com

LI-COR Distributor Network:

www.licor.com/env/distributors

977-18536 • 01/2020

The LI-COR logo is displayed in a bold, italicized, white sans-serif font against a solid blue background. The letters 'LI-COR' are connected, with a registered trademark symbol (®) positioned to the right of the 'R'.