

FLXLab 1.9 User's Guide

(c) 2006 Todd R. Haskell

July 20, 2006

Contents

1	Change Log - version 1.7 to version 1.9	3
1.1	Major Changes	3
1.2	Minor Changes and Bug Fixes	4
1.3	Backwards Compatibility	4
2	Introduction	4
2.1	What is FLXLab?	4
2.2	What are the advantages of FLXLab?	4
2.3	How can I get started using FLXLab?	5
3	The User Interface	5
4	The Demo Experiments: A Tutorial	6
4.1	Reaction Time I	6
4.1.1	Commands	6
4.1.2	Basic Events	7
4.2	Reaction Time II	8
4.2.1	Aborting an experiment	10
4.2.2	Blank lines and comments	10
4.2.3	Using a stimulus list	10
4.2.4	Positions	10
4.2.5	Displaying text	11
4.2.6	Delays of variable duration	11
4.2.7	Variables and literal values	11

4.2.8	Changing attributes of shapes	12
4.2.9	Defining a reference point for measuring reaction times	12
4.2.10	Recording data to a file	12
4.2.11	Determining the number of trials from the stimulus file	12
4.3	Lexical Decision	13
4.3.1	Getting input from the user	14
4.3.2	Combining strings of text	15
4.3.3	Generating paths to files on different operating systems	15
4.3.4	Specifying the data file	15
4.3.5	More advanced commands for text: Reading text from files, text boxes, and text events	15
4.3.6	Changing the font	16
4.3.7	Clearing the screen	16
4.3.8	Recording which key was pressed	17
4.3.9	Experiment events	17
4.4	Masked Priming	17
4.4.1	Synchronizing events with the display refresh	19
4.5	Temporal Order Judgment	20
4.5.1	A more complex experiment structure	21
4.5.2	Playing sounds	22
4.6	Picture Naming	22
4.6.1	Presenting images from picture files	24
4.6.2	Recording sound	24
4.6.3	Using the voice key	24
4.7	Sentence Production	25
4.7.1	Defining new colors	28
4.7.2	Intermixing events and commands	28
4.7.3	Selecting objects and changing the screen color	29
4.7.4	A more complex use of pictures: Image pattern objects	29
4.7.5	Using spaces or tabs in the stimulus file	29
4.8	A Psychophysical Experiment	30
4.8.1	Using command events	32
4.8.2	Using macros	33
4.8.3	Using counters	33

4.8.4	Conditions for executing commands	33
4.9	An Arithmetic Experiment	34
4.9.1	Collecting a response of more than one key	35
4.9.2	Updating objects and events during a trial	35
4.9.3	Using whenever conditions	37
4.9.4	Condition precedence	37
5	How FLXLab works: Going beyond the demos	38
5.1	Events, sub-events and conditions	38
5.2	More complex sequences of events	39
5.2.1	Events that only occur under certain conditions	39
5.2.2	Events that depend on the participant's response	41
5.3	Conditions	42
5.3.1	Basic Conditions	42
5.3.2	Condition Modifiers	43
5.4	Built-in Variables	44
5.5	Trials, Stimulus Lists, and Updating	44
5.6	Macros	45
6	Command Reference	45

1 Change Log - version 1.7 to version 1.9

1.1 Major Changes

- Much improved handling of fonts. The new `AddFontDirectory` command allows you to make use of many (if not all) of the fonts installed on your system, rather than just the three provided with FLXLab. This includes Unicode fonts, which means that you can easily present stimuli in Chinese, Japanese, and other languages that use a non-Roman writing system. See section 4.3.6 and the description of the `AddFontDirectory` command for details.
- It is now possible to record responses consisting of multiple key presses, as well as a single key, using the `input` variable. See section 4.9.1 as well as the description of the `input` variable in section 5.4 for details.
- A new condition modifier `whenever` has been added to provide more precise control over execution of events. See section 4.9.3 as well as the description in section 5.3.2 for details.

- For speed and efficiency, FLXLab usually only updates graphics objects (text, pictures, etc.) at the beginning of a trial. The current version adds a new event type `UpdateEvent` which allows you to update such objects at any point during a trial. See section 4.9.2 as well as the description of the `UpdateEvent` command for details.

1.2 Minor Changes and Bug Fixes

The `key` variable now always holds the last key pressed, as described in the user guide. In earlier versions, it held the last key that triggered a condition, which may or may not be the same thing.

1.3 Backwards Compatibility

The fonts distributed with FLXLab have been given new names, to avoid potential confusion with other fonts on your system. The old and new names correspond as follows: Bookman has become URWBookman, Courier has become URWNimbus, and Palatino has become URWPalladio. Scripts which use the `Font` command to select a specific font will need to be modified to use the new names. See section 4.3.6 for additional details. Otherwise, any script written for version 1.7 should also work in version 1.9.

2 Introduction

2.1 What is FLXLab?

FLXLab is a program for conducting computer-based experiments in psychology and related fields. It can present stimuli in the form of graphics, text, or sound, and can record responses to those stimuli, either manual (i.e., pressing a key) or verbal.

2.2 What are the advantages of FLXLab?

There are a number of existing programs, both free and commercial, for conducting experiments. Depending on your needs, one of these programs might work better than FLXLab. However, FLXLab offers several advantages over many of these programs.

- FLXLab may be freely used, copied, and redistributed, subject to the terms of the license at the end of this document.
- FLXLab is available for several different operating systems, and can be easily extended to others. See the Getting Started guides for the particular

operating system you are interested in.

- FLXLab has a highly modular design; additional capabilities (e.g., input from a joystick, support for special types of experiments) can be added simply by installing an appropriate module.
- FLXLab is an “open source” program. This means that the computer code used to create the program is available for anyone who wants it, which makes it easy to add new features (see the separate Programmer’s Guide for more information).
- FLXLab provides more sophisticated control over stimulus presentation and data collection than many existing programs. In particular, FLXLab provides powerful graphics capabilities.

2.3 How can I get started using FLXLab?

Installation instructions are available for each operating system that FLXLab runs on. Once FLXLab is installed, you can try out the various demo experiments that are provided. The following section describes how to use the user interface to load and run scripts, including the demo scripts. The scripts themselves are described in detail later in this guide. Following the description of the demos, there is a reference section that provides detailed information on important FLXLab concepts and descriptions of all FLXLab commands.

3 The User Interface

FLXLab provides a basic graphical user interface to facilitate working with scripts. If you run FLXLab directly you will be asked whether you want to create a new script file (equivalent to the **New** option below) or select an existing file (equivalent to the **Select** option below). The file thus created or selected becomes the current script file. If you run FLXLab by, e.g., double-clicking on a script file (or providing the script file as a command line argument in Linux), that file becomes the current script file. The name of the current script file is always displayed in the main FLXLab window.

Once the main FLXLab window is displayed, you can select among the following choices.

New Displays a dialog box allowing the user to enter a name for a new script file. Once a name has been chosen, the FLXLab window is hidden, and the new file is opened in a text editor program (typically WordPad for Windows and emacs for Linux, although you can customize this). When you want to return to the FLXLab program, simply quit the text editor.

Select Displays a dialog box allowing the user to select a script file to work with.

Edit Opens the current script file in a text editor. As with editing a new file, when you want to return to the FLXLab program, simply quit the text editor.

Run Executes the current script file.

Quit Exits the FLXLab program.

If you prefer not to make use of the user interface, you can disable it by going into the directory where FLXLab was installed, and removing the file `launcher_config.flx` from the `config` directory (it is recommended you simply move this file someplace else, in case you want to re-enable the interface later).

4 The Demo Experiments: A Tutorial

This section of the user guide introduces the basic principles of designing an experiment in FLXLab, using the set of demo experiments provided with the program as examples.

FLXLab experiments are controlled by a *script*. A script is a text file containing commands that set up the experiment. Instructions for installing FLXLab and executing an experiment script are contained in the Getting Started guide for your operating system. Once FLXLab is installed, you should be able to execute each of the demo experiments contained in the `demos` directory. The sections below review the script files for each of these experiments, and explain how the script files work. They are organized starting with the simplest demo and progressing to increasingly more complicated experiments.

4.1 Reaction Time I

The Reaction Time I demo experiment functions as follows. Each trial begins with a one-second pause, after which a small rectangle appears in the middle of the screen. It remains on the screen until a key on the keyboard is pressed, at which point it disappears. This cycle repeats five times. The text of the script file is reproduced in Table 1. Note that the numbers on the left are for reference only, and are not part of the script.

4.1.1 Commands

Each line of the script contains one FLXLab command. The name of the command is always the first word on the line (e.g., `DelayEvent`, `DisplayEvent`,

Table 1: Script for Reaction Time I

```

1 DelayEvent pause 1000
2 RectangleObject rectangle
3 DisplayEvent stimulus
4 AddObject rectangle
5 WaitEvent wait_for_key "until key any"
6 TrialEvent trial
7 AddEvent pause
8 AddEvent stimulus
9 AddEvent wait_for_key
10 BlockEvent reaction_time "repeat 5"
11 AddEvent trial
12 Start reaction_time

```

etc.). The remaining words on the line are *arguments* to the command. Arguments are additional information that affects how the command functions. Commands may have anywhere from zero to four arguments; in some cases one or more arguments may be optional. If an argument contains more than one word, it must be surrounded by quotes to tell FLXLab to treat it as a single unit.

4.1.2 Basic Events

The building blocks of an experiment in FLXLab are *events*. “Simple” events are those that control stimulus presentation and data collection. In this experiment, there are three such events. One of them controls the pause at the beginning of each trial. This event is created with the `DelayEvent` command, which takes two arguments (line 1). The first argument specifies a name that will be used to refer to this event (here, `pause`). The second specifies the duration of the delay in milliseconds (here, 1000, or one second).

A second event controls the presentation of the rectangle. The first thing we need to do is to create a *display object* which will be shown on the screen. This is accomplished on line 2 with the `RectangleObject` command, which creates a rectangle named `rectangle`. The actual event is created with the `DisplayEvent` command, which just takes one argument, the name of the event. By itself, this event would display nothing on the screen. To tell FLXLab to display the rectangle, we need to associate it with the `stimulus` event. This is accomplished with the `AddObject` command on line 4.

The third simple event is created with the `WaitEvent` command on line 5. The first argument is the name of the event; the second specifies what we’re waiting for, in this case for the user to press a key on the keyboard.

At the next level up, events can be combined into a *trial*. This kind of event can be created with the `TrialEvent` command. The argument to this command specifies the name of the event. Lines 7, 8 and 9 then use the `AddEvent` command to add the events `pause`, `stimulus`, and `wait_for_key` to `trial`.

One more level up, trials can be combined to form *blocks*. This kind of event can be created with the `BlockEvent` command (line 10). Again, the first argument to this command is the name of the event. The second argument (`'repeat 5'`) indicates that the block will cycle through five times. We then add `trial` to our block event `reaction_time`, again using the `AddEvent` command.

Note that there is an even higher level of event called an *experiment*; we will encounter this in some of the more complicated demo experiments described below.

Finally, we need to tell FLXLab to start the event `reaction_time`; this is done with the `Start` command on line 12.

This script contains many of the major elements of an experiment, but it is not a very good experiment. The timing of the rectangle is predictable, and the script does not actually record the reaction times. These and other issues are addressed in the next demo experiment.

4.2 Reaction Time II

The second version of the reaction time experiment involves a somewhat more sophisticated procedure. Each trial begins with a one-second pause, as before. This is followed by a fixation cross in the middle of the screen. After a delay of variable duration (either 800, 1000, 1200, 1400 or 1600 ms), a small filled rectangle appears in one of eight positions around the screen. Again, the task is to press a key on the keyboard as soon as the rectangle appears. After the experiment, there will be a file called `data.txt` in the `reaction_time_II` directory that contains the reaction times for each trial, measured from the onset of the rectangle. The text of the script file is reproduced in Table 2.

Table 2: Script for Reaction Time II

```
1  # set up the stimulus list
2  StimulusList stimulus_list stimulus_list.txt
3  LabelListColumn 1 stimulus_onset_time
4  LabelListColumn 2 stimulus_position
5
6  # define the positions at which the rectangle can appear
7  DefinePosition 1,1 25% 25%
8  DefinePosition 1,2 50% 25%
9  DefinePosition 1,3 75% 25%
10 DefinePosition 2,1 25% 50%
11 DefinePosition 2,3 75% 50%
```

```

12 DefinePosition 3,1 25% 75%
13 DefinePosition 3,2 50% 75%
14 DefinePosition 3,3 75% 75%
15
16 # set up the delay between trials
17 DelayEvent intertrial_delay 1000
18
19 # create an event to display a fixation cross
20 TextEvent fixation_cross +
21
22 # the delay between the fixation cross and the
23 # rectangle; the duration of this delay is variable
24 # and is taken from the stimulus file
25 DelayEvent stimulus_onset_delay $stimulus_onset_time
26
27 # create an event to display the rectangle; the
28 # position of the rectangle is variable and is
29 # taken from the stimulus file
30 RectangleObject rectangle
31 Filled
32 DisplayEvent stimulus
33 ResetDataTime
34 AddObject rectangle $stimulus_position
35
36 WaitEvent wait_for_key "until key any"
37
38 # create an event to record the reaction time
39 DataEvent record_rt
40 DataColumn $stimulus_onset_time
41 DataColumn $stimulus_position
42 DataColumn $time
43
44 # add all the events together to make a trial
45 TrialEvent trial
46 AddEvent intertrial_delay
47 AddEvent fixation_cross
48 AddEvent stimulus_onset_delay
49 AddEvent stimulus
50 AddEvent wait_for_key
51 AddEvent record_rt
52
53 # continue to do trials until we get to the end
54 # of the stimulus list
55 BlockEvent reaction_time "until list end"
56 AddEvent trial
57

```

```
58 # start the experiment
59 Start reaction_time
```

4.2.1 Aborting an experiment

This experiment is considerably longer than Reaction Time I, and you may be wondering if there is a way to stop the experiment in the middle. The answer is yes: All you have to do is type Control-Q. Note that this won't work if you have a dialog box open on the screen (as occurs in some of the later demo experiments); you'll have to close it first.

4.2.2 Blank lines and comments

One of the major differences between this script and the reaction time I script is that this script contains numerous blank lines and lines beginning with a #. Lines beginning with # are called *comments*, and allow you to make notes, explain the purpose of particular commands, or anything else you'd like. Both blank lines and comments are ignored by FLXLab.

4.2.3 Using a stimulus list

In this experiment, both the timing of the stimulus and its position vary from trial to trial. This information is stored in a separate text file, in this case a file called `stimulus_list.txt`. This file contains one line for each trial of the experiment. There are two columns in the file. The first specifies the time from the onset of the fixation cross to the onset of the rectangle. The second specifies the position of the rectangle.

Lines 2-4 in the script configure FLXLab to use this stimulus file. The `StimulusList` command indicates that we will be using a stimulus list called `stimulus_list` read from the file `stimulus_list.txt`. The two `LabelListColumn` commands define labels we can use to refer to values in the first and second columns of the file.

4.2.4 Positions

By default, display objects are presented at the center of the screen. However, it is possible to present them at some other location by the use of *positions*. There are nine pre-defined positions: `topleft`, `top`, `topright`, `left`, `center`, `right`, `bottomleft`, `bottom`, and `bottomright`. Additional positions can be defined using the `DefinePosition` command. The first argument of this command is the name to be given to this position (e.g., the position defined on line 7 is given the name `1,1`). The next two arguments indicate the horizontal and vertical coordinates of the position. These may be specified either as a percentage of the

screen width, as done here, or as absolute pixel values. The use of percentages is generally preferred, as it ensures that stimuli appear in the same relative locations even if you change the monitor you are using.

To specify that a particular display object should appear at a particular position, we use a second, optional argument to the `AddObject` command, as is done on line 34. Note that `$stimulus_position` is a label for the second column in the stimulus file (see section 4.2.7 for an explanation of the `$`). If you look in the stimulus file, you will see that this column contains entries like `2,3`, `3,1`, `3,2`, etc. Thus, to determine the position of the rectangle on any given trial, FLXLab looks at the appropriate entry from the stimulus file.

4.2.5 Displaying text

The fixation cross in this experiment is implemented as a plus sign. This requires a new kind of event, created using the `TextEvent` command (see line 20). As with the `DisplayEvent` command, the first argument specifies the name for the event (here, `fixation_cross`. There is also a second argument that specifies the text to be displayed. Remember that if the text is more than one word, you must surround it with quotes. Here this text is provided explicitly in the script file; it is also possible to place the text in the stimulus file so that it changes from trial to trial. This is done in the same way as for positions or times (discussed next).

4.2.6 Delays of variable duration

Line 25 of the script defines a delay event for the time between the onset of the fixation cross and the onset of the rectangle. To make this delay variable, we place the duration of the delay in the stimulus file, and use the label for this column (`$stimulus_onset_time`) in place of the literal time (compare the second argument to the `DelayEvent` commands on lines 17 and 25). See section 4.2.7 for an explanation of the `$`.

4.2.7 Variables and literal values

Note that the column label on line 25 is preceded by a `$`. This signals to FLXLab that we are providing the name of a variable that contains the appropriate value, rather than providing the value directly. Without this convention, strings like `1000` (line 17) or `+` (line 20) would be ambiguous: Should they be taken as the names of a variables, or as the actual values themselves? Of course, `1000` and `+` are rather odd names for a variable, but there are also variables with names like `time` (line 42), in which case distinguishing between a variable and a literal value is not so easy.

The `$` should be used with any variables created by the `LabelListColumn` command; in this script, these are `stimulus_onset_time` and `stimulus_position`.

It must also be used with “built-in” variables such as `time` (see line 42).

4.2.8 Changing attributes of shapes

In the Reaction Time I experiment, the stimulus rectangle was a small, black outline. It is possible to change various attributes of shape objects, including their size, their color, the weight of the line used to draw them, and whether or not they are filled. This last possibility is utilized on line 31, where the `Filled` command specifies to use a filled rectangle rather than the default outline. Other attributes can be changed with the `Size`, `Color`, `BoxColor`, and `LineWidth` commands; more details are provided in the description of later scripts and in the reference section. Note that in addition to rectangles, you can create ellipses with the `EllipseObject` command.

4.2.9 Defining a reference point for measuring reaction times

In recording reaction times in our experiment, we want those times measured from the onset of the rectangle stimulus. By default, FLXLab measures reaction times from the beginning of the trial. You can indicate that you want reaction times measured relative to a specific event using the `ResetDataTime` command, as is done on line 33.

4.2.10 Recording data to a file

For our experiment to be useful, we have to be able to record the reaction times to a data file for analysis. Once you have run the experiment, you will find a file called `data.txt` in the `reaction_time_II` directory. This file contains three columns. The first two are the same as the stimulus file. The last is the reaction time for each trial.

This information is recorded to the file by means of the `DataEvent` command (line 39). This command defines an event that will write one line to the data file. The contents of that line are specified using one or more `DataColumn` commands. Lines 40 and 41 indicate that the presentation time and position of the rectangle stimulus should be written to the file; recall that `stimulus_onset_time` and `stimulus_position` were used to label these values in the stimulus file. Line 42 uses the special value `time`, which always refers to the current elapsed time (relative to the beginning of the trial or an event identified with the `ResetDataTime` command).

4.2.11 Determining the number of trials from the stimulus file

When a stimulus file is used, it is convenient to have the experiment simply run until all the lines in the stimulus file have been used. This is achieved in the current experiment by using ‘`until list end`’ as the second argument to

the `BlockEvent` command on line 55. This simply indicates that FLXLab should keep cycling through the block until the end of the stimulus list is reached.

4.3 Lexical Decision

This experiment introduces a new procedure: Lexical decision. On each trial, a string of letters is presented on the screen, and the participant has to decide whether or not it is an actual word of English, and press one of two keys accordingly. In addition, two more improvements are made. At the beginning of the experiment, a dialog box is presented for the experimenter to enter in the subject ID, which is used to create distinct data files for each subject. Second, instructions are presented on the screen prior to the beginning of the actual lexical decision trials. The text of the script file is reproduced in Table 3.

Table 3: Script for Lexical Decision

```
1 # Get the subject id from the user
2 EditDialog subject_id "Enter subject id:" subject01
3 # Use the subject id to generate the name of the data file
4 JoinStrings data_file "data_files $path_separator $subject_id .txt"
5 # Set the name of the data file
6 UseDataFile $data_file
7
8 # Create labels for the three columns in the stimulus file
9 StimulusList stimulus_list stimulus_list.txt
10 LabelListColumn 1 item_number
11 LabelListColumn 2 stimulus_item
12 LabelListColumn 3 item_type
13
14 # Use 36 point Courier throughout the experiment
15 Font Courier 36
16
17 # This creates an event to display the instructions
18 LoadTextFromFile instructions_text instructions.txt
19 TextBoxEvent instructions $instructions_text
20
21 WaitEvent wait_for_key "until key any"
22
23 TextEvent warning_signal ****
24
25 DelayEvent pause 500
26
27 ClearScreenEvent clear_screen
28
29 TextEvent stimulus $stimulus_item
30 # Reaction times will be relative to the onset of the stimulus
```

```
31 ResetDataTime
32
33 DataEvent record_response
34 DataColumn $item_number
35 DataColumn $stimulus_item
36 DataColumn $item_type
37 DataColumn $key
38 DataColumn $time
39
40 TrialEvent trial
41 AddEvent warning_signal
42 AddEvent pause
43 AddEvent clear_screen
44 AddEvent pause
45 AddEvent stimulus
46 AddEvent wait_for_key
47 AddEvent record_response
48
49 BlockEvent mainblock "until list end"
50 AddEvent trial
51
52 ExperimentEvent lexical_decision
53 AddEvent clear_screen
54 AddEvent instructions
55 AddEvent wait_for_key
56 AddEvent mainblock
57
58 Start lexical_decision
59
```

4.3.1 Getting input from the user

It is often convenient to collect certain information at the beginning of the experiment, such as what the subject ID should be or which list to use. This can be achieved with the `EditDialog` command, which displays a dialog box where the user can enter in information. The first argument to this command is a name which you can use to refer to whatever the user entered in. The second argument is a prompt which will be displayed in the dialog box. The third argument is the default text, which the user can either accept as is or edit. See line 2 of the script for an example of how to use this command to obtain a subject ID.

4.3.2 Combining strings of text

Now that we have the subject ID, we would like to use it to generate the name for a file to store that subject's data. This can be done with the `JoinStrings` command. The first argument to this command is a name which can be used to refer to the combined strings. The second argument is a series of strings (surrounded by quotes) to combine. To create the name of the data file, we take the name of the data directory (`data_files`), add the special variable `path_separator` (see section 4.3.3), the subject ID, and the suffix `.txt` to indicate that it is a text file (see line 4 of the script).

4.3.3 Generating paths to files on different operating systems

The special term `path_separator` represents whatever character separates directory names in a path on your system - on Unix/Linux it is `/`, in Windows it is `\`. By using `path_separator` rather than the actual character used on your system, you can ensure that your scripts will work correctly on any operating system that FLXLab runs on. Taking the use of the `JoinStrings` command on line 4 as an example, if the subject ID was `subject01`, and FLXLab is being run on Windows, this would yield `data_files\subject01.txt`. We could then refer to this combined string with the name `data_file`.

4.3.4 Specifying the data file

By default, FLXLab will record data to the file `data.txt`. Each time you run the experiment, this file will be replaced. To specify a different data file, you can use the `UseDataFile` command. This command takes one argument, the name of the file (see line 6). This name will be the value of `data_file`, e.g., in Windows `data_files\subject01.txt`.

4.3.5 More advanced commands for text: Reading text from files, text boxes, and text events

In the Reaction Time II demo, we learned how to display text on the screen. This script introduces several new commands for working with text. The `LoadTextFromFile` command takes two arguments: A name to be used to refer to the text, and a file to read the text from. It is used on line 18 of the script to load the text of the instructions. The name `instructions_text` can then be used any place a text argument is expected (e.g., with a `TextEvent` command or the `TextBoxEvent` command, discussed below).

If a large amount of text is in the file, it may not all fit on one line of the screen. With a text event, the text will simply be cut off at the edge of the screen. If you want the text to wrap around instead, you can use the `TextBoxEvent` command (see line 19).

4.3.6 Changing the font

Like shape objects, text objects have a color attribute which may be changed with the `Color` command. However, they also have several attributes of their own. Font face and size may be specified with the `Font` command (see line 15). The font face is specified with the first argument. In general, you would just use the name of the font as it appears in the font menu of a word processor, with the spaces removed. Thus, Franklin Gothic Medium is `FranklinGothicMedium`. To specify bold or italic, append this to the name of the font, e.g., `BookAntiquaBold`, `FranklinGothicMediumItalic`, `GautamiBoldItalic`. Note that `Bold` should come before `Italic` in this case.

FLXLab is distributed with three font faces: URW Bookman, URW Nimbus (similar to Courier), and URW Palladio (similar to Palatino). The default font is URW Palladio. However, FLXLab should be able to utilize any PostScript or TrueType font on your system. To access these fonts, you can use the `AddFontDirectory` command. In Windows, placing the following line in the configuration file for the fonts module should allow you to access many (though not all) of the fonts on the system:

```
AddFontDirectory \windows\Fonts
```

If you downloaded and installed the Windows version of the program, this has probably already been done for you. In Linux, things are a bit trickier, because there generally isn't a single directory where most fonts are stored. You may have to look around a bit to see what is available and where it is located.

Once you have chosen a font face, the font size is specified with the second argument to `Font`, e.g.,

```
Font BookAntiquaBold 36
```

Because FLXLab uses PostScript and TrueType fonts, it is generally possible to have any font size you like, though some will look better than others. You can also display any character you like, including Unicode characters. In order for FLXLab to recognize Unicode characters in text files (e.g., script files or stimulus lists), the files should be saved in the UTF-8 format.

Note that setting the font before any text objects are defined, as is done here, changes the default font, which will then be used for all text objects unless otherwise specified.

4.3.7 Clearing the screen

FLXLab automatically clears the screen at the beginning of each trial. However, if you want to clear the screen at some other point, you can do this with the `ClearScreenEvent` command (see line 27). This command simply creates a special kind of event that clears the screen.

4.3.8 Recording which key was pressed

Since the participant can press one of two keys in this experiment, we would like to know which key they pressed. We can do this with a special value similar to `time` (as in Reaction Time II and line 38 of the current experiment), called `key` (see line 37). This holds the key that last triggered a key condition (as in ‘‘when key any’’).

4.3.9 Experiment events

Because we want to present instructions prior to the main experiment, we need a level of events above that of a block. This is achieved with the `ExperimentEvent` command (see line 52). Events may be added to an experiment in the same way they are added to a block. Thus, our experiment begins by clearing the screen, after which the instructions are displayed. FLXLab will then wait until a key is pressed, after which it begins the actual block of trials.

4.4 Masked Priming

This experiment is a variant on the lexical decision demo. On each trial, the participant sees a mask for 500 ms, followed by a prime word for 35 ms, followed by the target. The target stays on the screen until a response has been made.

Table 4: Script for Masked Priming

```
1 # Get the subject id from the user
2 EditDialog subject_id "Enter subject id:" subject01
3 # Use the subject id to generate the name of the data file
4 JoinStrings data_file "data_files $path_separator $subject_id .txt"
5 # Set the name of the data file
6 UseDataFile $data_file
7
8 # Create labels for the three columns in the stimulus file
9 StimulusList stimulus_list stimulus_list.txt
10 LabelListColumn 1 item_number
11 LabelListColumn 2 prime_item
12 LabelListColumn 3 target_item
13 LabelListColumn 4 item_condition
14 LabelListColumn 5 item_type
15
16 # Use 36 point Courier throughout the experiment
17 Font Courier 36
18
19 UseMicroseconds
20
```

```

21 ClearScreenEvent clear_screen
22
23 # This creates an event to display the instructions
24 LoadTextFromFile instructions_text instructions.txt
25 TextBoxEvent instructions $instructions_text
26
27 WaitEvent wait_for_key "until key any"
28
29 TextEvent mask #####
30 WaitForRefresh
31 ResetEventTime
32
33 TextObject prime_text $prime_item
34 DisplayEvent prime
35 Color white
36 AddObject prime_text
37
38 TextObject target_text $target_item
39 DisplayEvent target
40 Color white
41 AddObject target_text
42 # Reaction times will be relative to the onset of the target
43 ResetDataTime
44
45 DataEvent record_response
46 DataColumn $item_number
47 DataColumn $target_item
48 DataColumn $item_condition
49 DataColumn $item_type
50 DataColumn $key
51 DataColumn $time
52
53 TrialEvent trial "until event record_response"
54 AddEvent mask
55 AddEvent prime "when time 500000"
56 AddEvent target "when time 533333"
57 AddEvent record_response "when key any"
58
59 BlockEvent mainblock "until list end"
60 AddEvent trial
61
62 ExperimentEvent lexical_decision
63 AddEvent clear_screen
64 AddEvent instructions
65 AddEvent wait_for_key
66 AddEvent mainblock

```

```
67
68 Start lexical_decision
69
```

4.4.1 Synchronizing events with the display refresh

For paradigms like masked priming, where it is important to precisely control the amount of time the prime is visible, it is useful to be able to synchronize stimulus presentation with refresh of the display. Typical computer displays are refreshed somewhere between 60 and 100 times a second. Using the 60 Hz rate as an example, this means a refresh occurs every 16.67 ms. Regardless of when you tell FLXLab to present a stimulus and when you tell it to replace it with something else, these actions will only take place when the next refresh occurs. Thus, if your refresh rate is 60 Hz, your prime will actually be visible for a multiple of this number, i.e., 16.67 ms, 33.34 ms, etc.

If you tell FLXLab to display the prime for a multiple of the refresh rate (or very close to it), then it will be visible for that amount of time, whether or not you synchronize with the start of the refresh cycle. However, there is some possibility that “tearing” will occur, i.e., that the top half and the bottom half of the prime will be out of sync with each other. You can avoid this by using the `WaitForRefresh` command to tell a display event to wait for the beginning of the refresh cycle before drawing to the screen (see line 30). Note that at this time, this command only works consistently in the Windows version of FLXLab.

Once you’ve synchronized with the display refresh once, you can usually keep your stimuli in sync with the refresh simply by ensuring the pauses between them are multiples of the refresh rate. In this experiment, we synchronize with the display once (for the mask), and then just precisely control the time between presentation of the mask and presentation of the prime and target. Because the refresh cycle often is not an even number of milliseconds, it’s usually a good idea to specify times in microseconds (millionths of a second) in such experiment; see lines 55 and 56. To make FLXLab interpret times as microseconds, use the `UseMicroseconds` command. We also want to measure our times relative to the onset of the mask, because the amount of time we have to wait for the beginning of the refresh cycle can vary from trial to trial; to do this, we use the `ResetEventTime` command with the `DisplayEvent` that shows the mask (line 31). This basically means that when the `mask` event executes, it resets the timer that measures how long the trial has been running for, and which is used on lines 55 and 56 to determine when the prime and target should be displayed. See Section 5 for a more detailed description of how this works.

One final comment: Inserting a `DelayEvent` between stimuli is not a good way to ensure a precise interval between them. This is because drawing a display to the screen takes time - usually several milliseconds. This incidental delay is added onto the intentional delay created by your `DelayEvent`. So if you put a `DelayEvent` of 50 ms between two stimuli, the actual interval between them

might be more like 54 ms. Specifying times relative to a fixed point in the trial, as done here, is a better approach.

4.5 Temporal Order Judgment

This script implements an auditory temporal order judgment task. On each trial, the participant hears two very brief tones that are separated by a variable interval (in the case, anywhere from 0 to 50 ms in 5 ms increments). The tones differ slightly in pitch. The task is to decide whether the low tone preceded the high tone or vice versa, and press one of two keys accordingly. The text of the script file is reproduced in Table 5.

Table 5: Script for Temporal Order Judgment

```
1 # Get the subject ID and specify the data file
2 EditDialog subject_id "Please enter the subject id:" subject01
3 JoinStrings data_file "data_files $path_separator $subject_id .txt"
4 UseDataFile $data_file
5
6 # Define a block to display the instructions
7
8 ClearScreenEvent clear_screen
9
10 LoadTextFromFile instructions_text instructions.txt
11 TextBoxEvent instructions $instructions_text
12
13 WaitEvent wait_for_key "until key any"
14
15 BlockEvent do_instructions
16 AddEvent clear_screen
17 AddEvent instructions
18 AddEvent wait_for_key
19
20 # Define a trial
21
22 StimulusList stimulus_list stimulus_list.txt
23 LabelListColumn 1 order
24 LabelListColumn 2 interval
25
26 DelayEvent intertrial_interval 1000
27
28 JoinStrings sound_file "sound_files $path_separator $order $interval .wav"
29 PlaySoundEvent tones $sound_file
30
31 DelayEvent pause 500
32
```

```

33 TextEvent prompt "lo-hi hi-lo"
34
35 DataEvent record_response
36 DataColumn $order
37 DataColumn $interval
38 DataColumn $key
39
40 TrialEvent trial
41 AddEvent intertrial_interval
42 AddEvent tones
43 AddEvent pause
44 AddEvent prompt
45 AddEvent wait_for_key
46 AddEvent record_response
47
48 # Define a block of trials
49
50 BlockEvent mainblock "until list end"
51 AddEvent trial
52
53 # Define a block for displaying a thank you message
54
55 TextEvent thank_you "Thank you for your help! (press any key to exit)"
56
57 BlockEvent do_thank_you
58 AddEvent clear_screen
59 AddEvent thank_you
60 AddEvent wait_for_key
61
62 # Combine the three blocks into an experiment
63
64 ExperimentEvent temporal_order_judgment
65 AddEvent do_instructions
66 AddEvent mainblock
67 AddEvent do_thank_you
68
69 Start temporal_order_judgment

```

4.5.1 A more complex experiment structure

Although the term *block* is conventionally used to refer to a block of trials, in FLXLab a block event is simply a way to combine a number of events into one unit. The current script creates three block events which are combined into one experiment event. The first block displays the instructions, the second block runs the trials, and the third block displays a thank you message.

4.5.2 Playing sounds

FLXLab can play sounds that are stored in a WAV file (currently there is no support for other file formats). This is achieved with the `PlaySoundEvent` command (see line 29). This command takes two arguments: The first is the name for the event, the second is the file to load the sound from. In this experiment, there are 22 different sound files, corresponding to the 22 conditions of the experiment (11 different intervals between the tones x 2 tone orders). These sound files are contained in the `sound_files` directory. The name of the appropriate file is constructed out of the entries in the stimulus file by use of the `JoinStrings` command, as seen on line 28.

Note that it would also be possible to have only two sound files, one low tone and one high tone, and use a delay event to create the interval between them. However, there are often slight delays associated with beginning to play a sound. One reason for this is that sound cards generally don't start playing a sound immediately, but wait until they've received a certain amount of sound data (this helps avoid so-called *underruns*). There are also sometimes delays associated with activating the sound hardware. Thus, the method used in the current script may provide more precise control over the interval between the tones.

4.6 Picture Naming

This script implements a basic picture naming task. On each trial, the participant sees a warning signal, followed by a picture. The task is to say what the picture is. As soon as the participant starts speaking, a voice key is triggered, removing the picture from the screen. The spoken response is recorded to an audio file, and the latency to start speaking is recorded to the data file. The participant is given 2 seconds to finish their response, and then the next trial commences. Note that you will need to connect a microphone to your computer and ensure that recording is functioning properly in order for this experiment to work. The text of the script file is reproduced in Table 6.

Table 6: Script for Picture Naming

```
1 EditDialog subject_id "Please enter the subject id:" subject01
2 JoinStrings data_file "data_files $path_separator $subject_id .txt"
3 UseDataFile $data_file
4
5 StimulusList stimulus_list stimulus_list.txt
6 LabelListColumn 1 picture_name
7
8 ClearScreenEvent clear_screen
9
10 LoadTextFromFile instructions_text instructions.txt
11 TextBoxEvent instructions $instructions_text
```

```

12
13 WaitEvent wait_for_key "until key any"
14
15 # Use a large font for the warning signal to make the cross more visible
16 TextEvent warning_signal +
17 Font Courier 72
18
19 DelayEvent pause 500
20
21 # Set up the file that spoken responses will be recorded to
22 JoinStrings sound_file "sound_files $path_separator $subject_id - $picture_name .wav"
23 # An event to record the spoken responses
24 RecordSoundEvent toggle_recording $sound_file
25
26 # Create a string with the file name that images will be loaded from
27 JoinStrings picture_file "images $path_separator $picture_name .bmp"
28 # An event to present the images
29 ImageEvent stimulus $picture_file
30 ResetDataTime
31
32 WaitEvent wait_for_voice "until voice_key"
33
34 # The data file records the name of the picture and the latency for the
35 # voice key to trigger
36 DataEvent rt_data
37 DataColumn $picture_name
38 DataColumn $time
39
40 DelayEvent record_response 2000
41
42 TrialEvent trial
43 AddEvent warning_signal
44 AddEvent pause
45 AddEvent clear_screen
46 AddEvent pause
47 # Turn audio recording on here
48 AddEvent toggle_recording
49 AddEvent stimulus
50 AddEvent wait_for_voice
51 AddEvent rt_data
52 AddEvent clear_screen
53 AddEvent record_response
54 # Turn audio recording off here
55 AddEvent toggle_recording
56
57 BlockEvent mainblock "until list end"

```

```
58 AddEvent trial
59
60 ExperimentEvent picture_naming
61 AddEvent clear_screen
62 AddEvent instructions
63 AddEvent wait_for_key
64 AddEvent mainblock
65
66 Start picture_naming
```

4.6.1 Presenting images from picture files

FLXLab can display images stored in several formats, including BMP (Windows bitmaps), LBM, PCX, and TGA. This is achieved with the `ImageEvent` command (see line 29), which is roughly analogous to the `PlaySoundEvent` command. This command takes two arguments. The first is the name for the event, the second is the file to load the image from. The name of the file is constructed on line 27 using the `JoinStrings` command.

4.6.2 Recording sound

FLXLab can record sound from a microphone or other external source, and write the sound data to a WAV file. Sound recording is controlled by the `RecordSoundEvent` command (see line 24). This command takes two arguments. The first is the name for the event, the second is a file to write the sound data to. Sound recording events are a bit unusual in that they work like a switch. The first time the event occurs, it turns sound recording on (line 48). The second time it occurs, it turns recording off again, and writes the sound data to the file (line 55). In this script, the sequence is as follows: Turn recording on, display the stimulus, wait for the voice key to trigger, write the speech initiation latency to the data file, clear the screen, wait two seconds, then turn recording off.

4.6.3 Using the voice key

FLXLab provides a software voice key. To use the voice key, you must first start recording sound. If you don't actually want to record the sound to a file, you can use a variant of the `RecordSoundEvent` command called `VoiceKeyEvent`, which does not require a file name. The two types of events work exactly the same except a voice key event discards the sound data at the end of the trial.

You can check whether the voice key has triggered in a similar way as you can check whether a key has been pressed, using phrases like "until voice_key" and "when voice_key". Note that the underscore is necessary here, as `voice_key` is one unit. In the current script, we define an event `wait_for_voice` that simply pauses until the voice key has been triggered (line 32).

The voice key is auto-calibrating, meaning that it adjusts its sensitivity on each trial based on the level of background noise. Thus, the voice key should work correctly regardless of the sound input level or the background noise level, provided there is a sufficient signal-to-noise ratio.

4.7 Sentence Production

This experiment is a more elaborate variation on Picture Naming designed to elicit simple sentences. On each trial, the participant sees a warning signal, followed by a set of four pictures in four different colors. Shortly after the pictures appear, a white rectangle briefly appears around one of the pictures (the “target picture”). The task is to indicate the color of the picture using a complete sentence. For example, if the target picture was a red umbrella, the participant would say “The umbrella is red.” On half the trials the target picture will occur alongside the same picture in a different color, in which case the participant must disambiguate which picture is being referred to, e.g., “The umbrella above the airplane is red.” The spoken response is recorded to an audio file. For purposes of illustration, when a key on the keyboard is pressed, the expected response is shown on the screen. After two seconds, the experiment advances to the next trial. The text of the script file is reproduced in Table 7.

Table 7: Script for Picture Naming

```
1   ### GENERAL CONFIGURATION
2
3   # Configure the data file
4   EditDialog subject_id "Please enter the subject id:" subject01
5   JoinStrings data_file "data_files $path_separator $subject_id .txt"
6   UseDataFile $data_file
7
8   # Configure the stimulus list
9   StimulusList stimulus_list stimulus_list.txt
10  LabelListColumn 1 trial_id
11  LabelListColumn 2 picture1_name
12  LabelListColumn 3 picture1_position
13  LabelListColumn 4 picture1_color
14  LabelListColumn 5 picture2_name
15  LabelListColumn 6 picture2_position
16  LabelListColumn 7 picture2_color
17  LabelListColumn 8 picture3_name
18  LabelListColumn 9 picture3_position
19  LabelListColumn 10 picture3_color
20  LabelListColumn 11 picture4_name
21  LabelListColumn 12 picture4_position
22  LabelListColumn 13 picture4_color
23  LabelListColumn 14 target_response
```

```

24
25 # Define the positions where the pictures will be displayed
26 DefinePosition P0 33% 33%
27 DefinePosition P1 67% 33%
28 DefinePosition P2 33% 67%
29 DefinePosition P3 67% 67%
30
31 # Define the colors that the pictures will be displayed in (red, green and
32 # blue are pre-defined)
33 DefineColor yellow 180 180 0
34 DefineColor purple 180 0 180
35
36 ##### DISPLAY THE INSTRUCTIONS
37
38 ClearScreenEvent clear_screen
39
40 LoadTextFromFile instructions_text instructions.txt
41 TextBoxEvent instructions $instructions_text
42
43 WaitEvent wait_for_key "until key any"
44
45 BlockEvent show_instructions
46 AddEvent clear_screen
47 AddEvent instructions
48 AddEvent wait_for_key
49
50 Start show_instructions
51
52 ##### CHANGE THE SCREEN COLOR TO BLACK
53 SelectObject screen
54 Color black
55
56 ##### CONFIGURE THE EXPERIMENT ITSELF
57
58 # Use a large font for the warning signal to make the cross more visible
59 TextEvent warning_signal +
60 Font Courier 72
61 Color white
62
63 DelayEvent pause 500
64
65 # Set up the file that spoken responses will be recorded to
66 JoinStrings sound_file "sound_files $path_separator $subject_id - $trial_id .wav"
67 # An event to record the spoken responses
68 RecordSoundEvent toggle_recording $sound_file
69

```

```

70 # Create strings with the file names that images will be loaded from
71 JoinStrings picture1_file "images $path_separator $picture1_name .bmp"
72 JoinStrings picture2_file "images $path_separator $picture2_name .bmp"
73 JoinStrings picture3_file "images $path_separator $picture3_name .bmp"
74 JoinStrings picture4_file "images $path_separator $picture4_name .bmp"
75
76 # Create display objects for each of the pictures
77 ImagePatternObject picture1 $picture1_file
78 Color $picture1_color
79 ImagePatternObject picture2 $picture2_file
80 Color $picture2_color
81 ImagePatternObject picture3 $picture3_file
82 Color $picture3_color
83 ImagePatternObject picture4 $picture4_file
84 Color $picture4_color
85
86 # Create a display event to show the pictures
87 DisplayEvent stimulus
88 AddObject picture1 $picture1_position
89 AddObject picture2 $picture2_position
90 AddObject picture3 $picture3_position
91 AddObject picture4 $picture4_position
92 MarkStartTime
93
94 # Create display events to show and then hide the flash
95 RectangleObject cue_on_object
96 Size 170 170
97 Color white
98 DisplayEvent cue_on
99 AddObject cue_on_object $picture1_position
100
101 RectangleObject cue_off_object
102 Size 170 170
103 Color black
104 DisplayEvent cue_off
105 AddObject cue_off_object $picture1_position
106
107 # A display event to show the target response
108 TextObject target_object $target_response
109 Color white
110 DisplayEvent show_target_response
111 DefinePosition screen_bottom 50% 90%
112 AddObject target_object screen_bottom
113
114 # The data file just records the name of the picture and the target
115 # response

```

```

116 DataEvent record_trial_info
117 DataColumn $trial_id
118 DataColumn $target_response
119
120 DelayEvent long_pause 2000
121
122 TrialEvent trial
123 AddEvent warning_signal
124 AddEvent pause
125 AddEvent clear_screen
126 AddEvent pause
127 # Turn audio recording on here
128 AddEvent toggle_recording
129 AddEvent stimulus
130 AddEvent pause
131 AddEvent cue_on
132 AddEvent pause
133 AddEvent cue_off
134 AddEvent wait_for_key
135 AddEvent show_target_response
136 # Turn audio recording off here
137 AddEvent toggle_recording
138 AddEvent record_trial_info
139 AddEvent long_pause
140
141 BlockEvent sentence_production "until list end"
142 AddEvent trial
143
144 Start sentence_production

```

4.7.1 Defining new colors

FLXLab pre-defines five colors: `white`, `black`, `red`, `green`, and `blue`. You may define additional colors using the `DefineColor` command (lines 33 and 34). The first argument to this command is the name to use for this new color. The next three arguments are values between 0 and 255 that indicate the intensity of red, green, and blue in the new color. Yellow is a blend of equal parts red and green; purple is a blend of equal parts red and blue. Once a color is defined in this way, it may be used just like the pre-defined colors.

4.7.2 Intermixing events and commands

The current script illustrates another possibility for how an experiment may be structured. An initial block event is created that displays the instructions, but rather than integrating it into a larger experiment event, it is executed

immediately (line 50). After the instructions have been displayed, the script continues with defining the experiment itself, which is then executed on line 140. The reason for doing this has to do with the commands on lines 53 and 54, which are discussed next.

4.7.3 Selecting objects and changing the screen color

In the current experiment, we want to present the instructions as black text on a white background. However, the colors of the pictures are easier to see against a black background. To achieve this, we change the color of the screen after the instructions are presented but before the experiment starts. This is the reason for intermixing events and commands, as discussed above. To change the color of the screen, we must first tell FLXLab that we'd like to work with the screen object. This is done with the `SelectObject` command, which takes one argument, the name of a display object (line 53). On line 54, we then change the color of this object to black, which gives us a black background during the experiment itself. Note that you can select any object, not just the screen. By default, objects are selected when you create them, and remain selected until you explicitly select another object or create a new object.

4.7.4 A more complex use of pictures: Image pattern objects

In addition to presenting pictures exactly as they are stored in a picture file, FLXLab can also use pictures as templates. Anything that is light in the picture will appear in the background color, and anything that is dark in the picture will appear in the color you specify. If the picture is essentially black-and-white, this has the effect of colorizing the picture. To do this, use the `ImagePatternObject` command (lines 77, 79, 81, and 83), which has the same form as the `ImageObject` command. Here, the color to use is taken from the stimulus list, and is different for each object and for each trial (see lines 78, 80, 82, and 84).

4.7.5 Using spaces or tabs in the stimulus file

By default, spaces and/or tabs are used in a stimulus file to separate columns. If you want to use spaces or tabs within a column, you can do so by putting double quotes around the entire column. This is done in the stimulus file for the current demo for the column containing the target response (see line 23). If you look at the stimulus file, you will see that this column contains entries like "the ring is blue" and "the eye above the ring is red", with the quotes indicating that the entire sentence is to be treated as a unit.

4.8 A Psychophysical Experiment

The goal of this experiment is to determine the discrimination threshold of the subject for differences in light intensity. According to Weber's Law, the discrimination threshold should get larger as the level of intensity increases; for simplicity, this script just measures the threshold at a particular point along this continuum.

The basic structure of a trial is as follows. A trial begins with a blank screen for 1500 ms. A gray square is then presented for 500 ms, followed by 1000 ms more of blank screen, followed by another gray square. The second gray square remains on the screen until the subject presses a key on the keyboard. The task is to press the 'f' key if the two squares were different shades of gray, and the 'j' key if they were the same shade of gray. Half the time the squares are the same shade, half the time they are different (this is specified in the stimulus file).

The two squares start out being clearly different shades. Each time the subject makes a correct response, they become more similar. Each time the subject makes an incorrect response, they become more distinct. Near the discrimination threshold, making the two grays more similar should result in an error, after which making the two grays more distinct again should result in a correct response. The experiment ends when two of these alternations have occurred. The intensity of both rectangles is recorded to the data file after each trial.

This is the least elegant of the demo scripts; it is somewhat awkward to implement this kind of experiment in FLXLab. However, it does demonstrate some of the more advanced capabilities of the program, and for this reason it is included here. The text of the script file is reproduced in Table 8.

Table 8: Script for Psychophysics Experiment

```
1  #### DEFINE SOME VARIABLES
2
3  Counter rect1_lum 76
4  Counter rect2_lum 176
5  DefineColor rect1_color $rect1_lum $rect1_lum $rect1_lum
6  DefineColor rect2_color $rect2_lum $rect2_lum $rect2_lum
7
8  StimulusList stimuli stimulus_list.txt
9  LabelListColumn 1 condition
10
11 Counter direction_of_change
12 Counter switch_counter
13
14 #### DEFINE THE EVENTS THAT MAKE UP A TRIAL
15
16 DelayEvent intertrial_interval 1500
17
```

```

18 RectangleObject rect1
19 Size 100 100
20 Color rect1_color
21 Filled
22
23 DisplayEvent stimulus1
24 AddObject rect1
25
26 DelayEvent view_stimulus 500
27
28 ClearScreenEvent clear_screen
29
30 DelayEvent interstimulus_interval 1000
31
32 RectangleObject rect2
33 Size 100 100
34 Color rect2_color
35 Filled
36
37 DisplayEvent stimulus2
38 AddObject rect2
39
40 WaitEvent wait_for_key "until key any"
41
42 DefineMacro DecreaseContrast
43 If "$direction_of_change equals 1" "Increment $switch_counter 1"
44 If "$direction_of_change equals 0" "Increment $direction_of_change -1"
45 If "$direction_of_change equals 1" "Increment $direction_of_change -2"
46 Start record_data
47 Increment $rect1_lum 3
48 Increment $rect2_lum -3
49 EndMacro
50 CommandEvent decrease_contrast DecreaseContrast
51
52 DefineMacro IncreaseContrast
53 If "$direction_of_change equals -1" "Increment $switch_counter 1"
54 If "$direction_of_change equals 0" "Increment $direction_of_change 1"
55 If "$direction_of_change equals -1" "Increment $direction_of_change 2"
56 Start record_data
57 Increment $rect1_lum -6
58 Increment $rect2_lum 6
59 EndMacro
60 CommandEvent increase_contrast IncreaseContrast
61
62 DataEvent record_data
63 DataColumn "rect1"

```

```

64 DataColumn $rect1_lum
65 DataColumn "rect2"
66 DataColumn $rect2_lum
67 DataColumn $direction_of_change
68 DataColumn $switch_counter
69
70 ##### BUILD THE TRIAL EVENT AND START THE EXPERIMENT
71
72 TrialEvent trial
73 AddEvent intertrial_interval
74 AddEvent stimulus1
75 AddEvent view_stimulus
76 AddEvent clear_screen
77 AddEvent interstimulus_interval
78 AddEvent stimulus1 "when $condition equals true"
79 AddEvent stimulus2 "when $condition equals false"
80 AddEvent wait_for_key
81 AddEvent decrease_contrast "either both $key equals j and
    $condition equals true or both $key equals f and $condition equals false"
82 AddEvent increase_contrast "either both $key equals f and
    $condition equals true or both $key equals j and $condition equals false"
83
84 BlockEvent psychophysics "until either $switch_counter equals 4 or list end"
85 AddEvent trial
86
87 Start psychophysics

```

4.8.1 Using command events

It is fairly straightforward to execute one of two events depending on the whether the subjects' response is correct or incorrect; this is illustrated on lines 81 and 82. The value of `$condition` is taken from the stimulus file, and is `true` if the two squares are the same shade, `false` if they're not. But what if you actually want to change the properties of some object, rather than execute an event? Generally this means you will need to run a command. But how can you run a command while you're in the middle of a trial?

This can be done using `CommandEvents`, as illustrated on lines 50 and 60. For example, line 50 creates a `CommandEvent` called `decrease_contrast`. Every time this event is executed, it runs the command `DecreaseContrast` (this is actually a macro; see the following section).

While we're talking about interleaving events and commands, take a look at lines 46 and 56. Here we execute an event in the middle of a sequence of commands, in order to record some information to the data file.

4.8.2 Using macros

In the current case, we don't just want to execute a single command depending on the subject's response, but several commands. To do this, we make use of macros. The command `DecreaseContrast` is actually a macro, defined on lines 42 to 49. This macro groups together six separate commands that should be executed every time the subject makes a correct response. Once the macro is defined, every time `DecreaseContrast` is executed, all six of these commands will be executed.

4.8.3 Using counters

FLXLab allows you to use counters to keep track of information during the course of the experiment. The current experiment utilizes four counters. Two of them (`$rect1_lum` and `$rect2_lum`) keep track of the current level of luminence associated with each of the two squares. These counters are created on lines 3 and 4, and then used to create a color variable for each rectangle on lines 5 and 6. Their values are updated using the `Increment` command on lines 47-48 (for a correct response) or 57-58 (for an incorrect response). Note that the amount by which the contrast is increased for an incorrect response is twice as large as the amount by which the contrast is decreased for a correct response; this sort of asymmetry is common for psychophysical experiments. For your reference, the values of both counters are written to the data file during each trial.

The other two counters (`$direction_of_change` and `$switch_counter`) are used to keep track of the subject's performance; this is discussed more fully in the following section.

4.8.4 Conditions for executing commands

We've already seen many examples of specifying specific conditions for the execution of an event. You can also specify conditions for executing a command, using the `If` command. This command takes two arguments; the first one is a condition, the second is a command to be executed if and only if the condition is satisfied.

Let's explore how this command is used in the current experiment. The crucial thing we need to keep track of is alternations between correct and incorrect responses. We use the `$direction_of_change` counter to do this. We'll set the value of this counter to -1 every time we decrease the contrast between the two rectangles, and 1 every time we increase it. This is done on lines 44-45 and 54-55. Lines 44 and 54 are needed to handle the first trial, when the value of `$direction_of_change` is 0 (because no responses have yet been made).

Now that we have this counter set up, we need one more counter to keep track of every time `$direction_of_change` goes from -1 to 1 or vice versa (i.e., a correct response is followed by an incorrect response, or an incorrect response is followed

by a correct response). This is done with the `$switch_counter`. It starts at 0, and is simply incremented by 1 each this occurs. The experiment stops when this counter reaches 4 (two complete alternations), or we get to the end of the data file (very unlikely, unless the subject has especially keen eyesight).

The values of both these counters are also written to the data file during each trial, to help you see how the experiment works.

4.9 An Arithmetic Experiment

This experiment illustrates how to collect responses from the user consisting of more than a single key press. Such responses could be words, sentences, or - as in this experiment - numbers. On each trial, a simple arithmetic problem is presented on the screen, either addition, subtraction, multiplication, or division. For example, on the first trial of the demo, the participant sees `32+45=`. The task is to type in the correct answer to this problem, then press the enter key. The text the participant types is shown on the screen as they type it. The backspace key may be used to correct mistakes. Once the enter key is pressed, another problem is shown. The data file records the problem number, the problem, the correct answer, and the answer the participant provides. The text of the script file is reproduced in Table 9.

Table 9: Script for Arithmetic Experiment

```
1 StimulusList stimulus_list stimulus_list.txt
2 LabelListColumn 1 item_number
3 LabelListColumn 2 problem
4 LabelListColumn 3 answer
5
6 Font URWNimbus 72
7
8 ClearScreenEvent clear_screen
9
10 JoinStrings problem_and_response_text "$problem $input"
11 TextBoxObject problem_and_response $problem_and_response_text
12 Size 40% 10%
13 Justification LEFT
14 Filled
15 BoxColor white
16 DisplayEvent show_problem_and_response
17 AddObject problem_and_response
18
19 UpdateEvent update_problem_and_response input
20
21 GroupingEvent update
22 AddEvent update_problem_and_response
```

```

23 AddEvent show_problem_and_response
24
25 DataEvent record_response
26 DataColumn $item_number
27 DataColumn $problem
28 DataColumn $answer
29 DataColumn $input
30
31 TrialEvent trial "until event record_response"
32 AddEvent clear_screen
33 AddEvent show_problem_and_response
34 AddEvent record_response "when key enter"
35 AddEvent update "whenever key any"
36
37 BlockEvent arithmetic "until list end"
38 AddEvent trial
39
40 Start arithmetic

```

4.9.1 Collecting a response of more than one key

Many of the previous demos have made use of the `key` variable, which always holds the last key that was pressed. In this case, however, we want to know all the characters that were typed on the current trial. This information is stored in the `input` variable (see line 29). If the user presses the backspace key at some point during the trial, this has the effect of deleting the last character stored in `input`, as would be expected. Any presses of the enter key are not included in `input`, as this key is usually used to indicate that the response is completed, and is not part of the response itself. Note that we still have to explicitly check for a press of the enter key (line 34) to know when to end the trial and record the response.

4.9.2 Updating objects and events during a trial

There are many actions that FLXLab must carry out for each trial of an experiment, some of which can take a measurable amount of time. These include loading images or sounds from files, as well as drawing text and graphics to be presented on the screen. If FLXLab carried out these actions only when they became necessary (i.e., during the middle of the trial), this could disrupt the timing of the trial. To avoid this, FLXLab carries out as much preparation as possible before the trial begins. This works well in the usual case where the text or image to present or the sound to play is already known at the start of the trial. However, in the current case we would like to display the characters that the user types, which naturally can't be known in advance. This situation is handled with the `UpdateEvent` command, which allows you to explicitly update

particular objects or events during a trial.

To understand how this works, you first need to understand the notion of a dependency. Consider the following lines from the script for the Picture Naming demo:

```
# Create a string with the file name that images will be loaded from
JoinStrings picture_file "images \${path\_separator} \${picture\_name} .bmp"
# An event to present the images
ImageEvent stimulus \${picture_file}
```

The name of the picture to display is taken from the variable `picture_name`, the value of which is obtained from the stimulus file. This is used to construct a variable `picture_file`, which contains the path to the file containing the picture. Thus, `picture_file` depends on `picture_name`, in that every time the value of `picture_name` changes, the value of `picture_file` needs to change as well.

The variable `picture_file` is used in turn to create an `ImageEvent` named `stimulus`. Thus, `stimulus` depends on `picture_file`, in that every time the value of `picture_file` changes, the image to be shown by `stimulus` needs to change. This results in a chain of two dependencies, such that `stimulus` indirectly depends on `picture_name`.

FLXLab keeps track of all the dependencies in an experiment, and uses this information in the updating process that occurs at the beginning of each trial. In the Picture Naming demo, at the beginning of each trial FLXLab will update all variables, objects and events that depend, either directly or indirectly, on `picture_name`. This ensures that the correct image is displayed.

The `UpdateEvent` command provides a way to explicitly trigger this updating process. In the current demo, we display on the screen the problem (contained in the variable `problem` followed by whatever the user has typed in thus far (contained in the variable `input`); the combination of these two things is stored in the variable `problem_and_response_text` (see line 10). This value is used to construct a `TextBoxObject` (line 11) which in turn is incorporated into a `DisplayEvent` (line 16). We need to make sure that the `DisplayEvent` gets updated every time `input` changes. We use the `UpdateEvent` command to do this (line 19), specifying `input` as the second argument. Every time this event is executed, it will cause the entire chain of dependencies associated with `input` to be updated. This ensures that the text on the screen is updated after every key press. Note that we don't use the `$` sign with the name of the variable here, i.e., `input` rather than `input`, because we're referring to the variable itself, not its value.

4.9.3 Using whenever conditions

The current demo also illustrates one way to use the condition modifier `whenever`. During each trial we need to keep checking for two things: Whether the user has pressed a key other than the enter key, in which case we should update the display, and whether the user has pressed the enter key, in which case we should record the response and proceed to the next trial. We use conditions (lines 34 and 35) for this purpose.

Most conditions will only evaluate to true once during a trial, when the criteria for the condition are first satisfied. For example, in the Masked Priming demo, we display the prime and target at specific times:

```
AddEvent mask
AddEvent prime "when time 500000"
AddEvent target "when time 533333"
```

This will cause the event `prime` to execute as soon as at least 500000 microseconds have elapsed since the onset of the mask. The next time the condition `"when time 500000"` is checked, it will evaluate to false, even though it is still the case that at least 500000 microseconds have elapsed. This prevents the prime from being displayed over and over again.

In the current case, however, we want to update the text on the screen every time a key is pressed, not just the first time. The `whenever` modifier (line 35) allows us to do this. The condition `whenever key any` will evaluate to true each and every time a key is pressed.

4.9.4 Condition precedence

In the current demo, we want to do different things depending on whether the enter key or some other key is pressed. To implement this, we place the condition checking for the enter key one line before the condition checking for any key (lines 34 and 35). This ensures that any time a key is pressed, FLXLab checks to see if it is the enter key before updating the text on the screen.

Once a key press satisfies some condition, it is considered “used up” and cannot satisfy any further conditions. That is, a press of the enter key will satisfy the condition on line 34, but not line 35. If we had put these two lines in the opposite order, the condition checking for the enter key would never be satisfied, because any such key press would always satisfy the more general condition (`whenever key any`) first.

5 How FLXLab works: Going beyond the demos

The demo experiments described above should provide enough information about how FLXLab works to design many simple experiments. However, if you need to design experiments very different from the demos, or you want to better understand how the scripts work, this section describes FLXLab in more detail.

5.1 Events, sub-events and conditions

FLXLab events can be divided into two types. *Simple* events carry out some action, such as presenting something on the screen (`DisplayEvent`), pausing for a specified duration (`DelayEvent`), or writing some information to the data file (`DataEvent`). *Compound* events control how one or more simple events take place. FLXLab has three commands for creating compound events: `TrialEvent`, `BlockEvent`, and `ExperimentEvent`.

Every compound event has what is called a *continue condition*. We have already encountered a couple such conditions, e.g., "`repeat 5`" and "`until list end`". The complete set of available conditions is described in section 5.3. The continue condition for a compound event is specified as the second argument when the event is defined, e.g., `BlockEvent main_block "until list end"`. When a compound event starts, it initiates an *event loop*. By loop is meant that FLXLab repeatedly cycles through a particular set of steps. The first step in this loop is that the continue condition for the event is checked. Using the example above, FLXLab would check if it has reached the end of the stimulus list. If it has, then the event immediately ends. If not, FLXLab continues on with the next step; once it has completed all the steps (one "pass" through the loop), it will come back and start going through the steps again, beginning with checking the continue condition.

If you don't explicitly specify the continue condition, a default continue condition is used. This default condition is "`repeat 1`". This means that FLXLab will go through the event loop once and then stop. You may specify any number you like after `repeat`, e.g., "`repeat 1000`" will cause FLXLab to go through the event loop 1000 times before stopping. Usually for trials and experiments the default continue condition is what you want (i.e., go through the entire event once).

If the continue condition is satisfied, FLXLab begins the second step, which is checking if any of the *sub-events* should be executed. Events become sub-events of a compound event by the use of the `AddEvent` command. Consider this snippet of the script from Reaction Time I:

```
TrialEvent trial
```

AddEvent pause

This says that the event `pause` is a sub-event of the event `trial`. Sub-events are also associated with a condition, called a *trigger condition*. If the condition is satisfied, the event will be executed, otherwise it will not. Again, there is a default condition if you don't specify one. For trial events and experiment events, the default condition is "`repeat 1`", which means the sub-event will be executed just once, the first time through the event loop. This is usually what you want.

For block events, the default trigger condition is `repeat` (without specifying a number), which means the sub-event will be executed on every pass through the event loop. Again, this is usually what you want. Consider the following snippet, used in several of the demo scripts:

```
BlockEvent mainblock "until list end"  
AddEvent trial
```

No trigger condition is specified for `trial`, so FLXLab uses the default trigger condition `repeat`. This means that `trial` will be executed on each pass through the event loop, until the end of the stimulus list is reached.

When there is more than one sub-event, FLXLab will check the trigger condition for the first sub-event, execute it if appropriate, then check the trigger condition for the next sub-event, execute it if appropriate, and so on. The order in which the sub-events are checked and executed is determined by the order in which they were added by the `AddEvent` command.

5.2 More complex sequences of events

All of the demo scripts utilized trials in which the order of the events was fixed, and all of the events took place on every trial. Let's consider how we can implement a trial that presents a warning message if (and only if) the participant takes too long to respond. We'll use the Lexical Decision demo as the template.

5.2.1 Events that only occur under certain conditions

In this example, we'll modify the Lexical Decision demo to give feedback to the participant if they are too slow. First, we need an event to actually present this feedback

```
TextObject too_slow_object "TOO SLOW!"  
TextColor red  
DisplayEvent too_slow_warning  
DefinePosition warning_position 50% 65%  
AddObject too_slow_object warning_position
```

This will display the text “TOO SLOW!” in red slightly below where the stimulus appears. These commands can be added to the script right before the definition of the `trial` event.

Now, we’ll revise the structure of our trial so that this warning message will be displayed if the participant takes too long to respond:

```
TrialEvent trial "until event record_response"  
AddEvent warning_signal  
AddEvent pause  
AddEvent clear_screen  
AddEvent pause  
AddEvent stimulus  
AddEvent record_response "when key any"  
AddEvent too_slow_warning "when time 2500"
```

To understand how this works, let’s first consider what would happen if we didn’t specify the continue condition `"until event record_response"` for the event `trial`. By default, FLXLab would just make one pass through the event loop (as if we had specified `"repeat 1"`). Immediately after the `stimulus` event, FLXLab would check if a key has been pressed, to determine if the event `record_response` should be executed. Presumably the participant has not had time to press a key yet, so FLXLab will just continue on and check if the time has reached 2500 ms, to know whether the event `too_slow_warning` should be executed. Only 1000 ms or so have elapsed since the start of the trial at this point, so FLXLab won’t execute that event, either. Now the end of the event loop is reached, and FLXLab goes back and checks the continue condition of `trial` again. Since by default this is `"repeat 1"`, FLXLab will note that it has already made one pass through the event loop, and end the trial. The participant will hardly have a chance to see the stimulus!

To avoid this problem, we need to use a different continue condition, `"until event record_response"`. This tells FLXLab to keep cycling through the event loop until the event `record_response` takes place. Since the default trigger condition for sub-events of a trial is `"repeat 1"`, the sub-events without an explicit trigger condition will only be executed on the first pass through the event loop, in the order they were added to the trial, i.e., `warning_signal`, then `pause`, then `clear_screen`, then `pause`, then `stimulus`. On subsequent passes through the event loop, these events will just be ignored, while FLXLab continues to check if a key has been pressed or the time has reached 2500 ms.

You might be wondering why we didn’t have to do this in any of the demos. There, we used an event like `wait_for_key` to delay the event `record_response` (and therefore the end of the trial) until a key has been pressed. FLXLab does allow you to wait for one of two things to happen in this way (see also section 5.3 below):

```
WaitEvent wait_for_key_or_timeout "until either key any or time 2500"
```

This is useful if you want to implement a “timeout” where the experiment continues on to the next trial even if the participant hasn’t made a response. However, we want to do different things depending on whether a key was pressed or the time reached 2500 ms, so this won’t work for us.

This example illustrates a contrast between what could be called a “fixed” trial structure and a “free” trial structure. In a fixed trial structure, the conditions for executing the sub-events and the sequence in which they should be executed is implied by the way in which they are added to the trial. However, we could also spell all this out, as in this alternative version of the `trial` event from Reaction Time I:

```
TrialEvent trial "until event wait_for_key"  
AddEvent pause "when time 0"  
AddEvent stimulus "when event pause"  
AddEvent wait_for_key "when event stimulus"
```

If we specify explicit trigger conditions in this way, we could just as well add the sub-events in the reverse order:

```
TrialEvent trial "until event wait_for_key"  
AddEvent wait_for_key "when event stimulus"  
AddEvent stimulus "when event pause"  
AddEvent pause "when time 0"
```

It is also possible to mix the two approaches, as was done in the example above.

5.2.2 Events that depend on the participant’s response

In this example we’ll modify the Lexical Decision demo so it gives feedback if the participant makes a mistake. First, we have to create an event to provide the feedback. We can’t simply present something on the screen and then move on to the next trial, as the participant won’t have a chance to read it. We also need to insert a delay (here, one second). In other words, we need to execute two separate events. To treat these events as a unit, we’ll create a new `BlockEvent`. For completeness, we’ll also clear the screen before presenting the feedback.

```
TextEvent feedback_message "Sorry, wrong answer"  
  
DelayEvent feedback_pause 1000  
  
BlockEvent error_feedback  
AddEvent clear_screen  
AddEvent feedback_message  
AddEvent feedback_pause
```

Now we need to tell FLXLab to execute this event if the participant makes an incorrect response. The first step here is to tell FLXLab what the incorrect response is. To do this, we add another column to the stimulus list, with the key (f or j) corresponding to the incorrect answer (this will be f for words and j for non-words). We also need to tell FLXLab about this new column:

```
LabelListColumn 4 incorrect_key
```

This would be added right after the other `LabelListColumn` commands. Finally, we add our feedback event to the trial structure:

```
TrialEvent trial
AddEvent warning_signal
AddEvent pause
AddEvent clear_screen
AddEvent pause
AddEvent stimulus
AddEvent wait_for_key
AddEvent record_response
AddEvent error_feedback "when $key equals $incorrect_key"
```

More details on the `equals` condition are provided in section 5.3.

5.3 Conditions

As is clear from the preceding portion of this guide, the use of conditions is pervasive in setting up experiments in FLXLab. Although many different conditions have been encountered already, this section gives a complete list of the conditions available in the core part of FLXLab. Note that add-on modules may provide additional condition options as well.

Note that when a condition is tested, it will evaluate to either true or false. Optional items are indicated with brackets.

5.3.1 Basic Conditions

repeat [*n*] Evaluates to true the first *n* times it is checked. If *n* is omitted, always evaluates to true.

time *t* Evaluates to true the first time it is checked after the time elapsed since the start of the current `TrialEvent`, `BlockEvent`, `ExperimentEvent`, or `GroupingEvent`, or since the last event tagged with the `ResetEventTime` command, reaches *t*. Note that this `time` refers to event timer, and is not necessarily the same as the `time` variable, which refers to the data timer.

- event *e*** Evaluates to true the first time it is checked after the event named *e* has been executed.
- list end** Evaluates to true if the end of the current stimulus list has been reached, or if there is no stimulus list.
- a* equals *b*** Evaluates to true if *a* equals *b*. Note that *a* must be a variable, though *b* can be either a variable or a literal value.
- key *k*** Evaluates to true the first time it is checked after the key *k* has been pressed. If *k* is **any**, then evaluates to true if any key has been pressed. Note that once a key condition evaluates to true, the corresponding key press is discarded, and will not satisfy any other conditions.
- mouse *b*** Evaluates to true the first time it is checked after mouse button *b* has been pressed, where *b* is **left**, **right**, or **center**. If *b* is **any**, then evaluates to true if any mouse button has been pressed.
- joystick *b*** Evaluates to true the first time it is checked after joystick button *b* has been pressed, where *b* is a number specifying a particular button. If *b* is **any**, then evaluates to true if any joystick button has been pressed.
- voice_key** Evaluates to true the first time it is checked while sound input is detected. Note that unlike **time**, **event** and **key**, it is not enough that sound was detected at some point in the past; it must be detected at the time the condition is checked.
- refresh** Evaluates to true if a screen refresh is about to begin, false otherwise.

5.3.2 Condition Modifiers

These are not conditions by themselves, but allow you to modify how basic conditions work, or combine them into more complex conditions.

- until *condition*** Evaluates to true until the first time *condition* evaluates to true.
- when *condition*** Equivalent to just *condition*; it is included to make scripts more readable.
- after *condition*** Evaluates to false until the first time *condition* evaluates to false.
- whenever *condition*** Evaluates to true whenever *condition* is true. This differs from **when** in that it can evaluate to true more than once.
- not *condition*** Evaluates to true if *condition* evaluates to false, and vice versa.
- both *condition1* and *condition2*** Evaluates to true if both *condition1* and *condition2* evaluate to true.

either *condition1* or *condition2* Evaluates to true if either *condition1* or *condition2* evaluate to true.

5.4 Built-in Variables

FLXLab maintains the following list of variables, the values of which may be used in conditions, written to data files, etc.

joystick - Holds the number of the last joystick button that was pressed.

key - Holds the last key that was pressed.

input - Holds all characters that have been entered since the beginning of the trial, except for the enter key, which is ignored, and backspace, which has the usual effect of deleting the previous character.

mouse - Holds the last mouse button that was pressed (**left**, **right**, or **center**).

time - Holds the time since the beginning of the current Trial, Block, Experiment, or GroupingEvent, or since the last event tagged with **ResetDataTime**. Note that there are two separate timers, corresponding to 'data' and 'event' time. Normally they are the same, but they can end up being different if you use the **ResetDataTime** or **ResetEventTime** commands.

5.5 Trials, Stimulus Lists, and Updating

Some commonly used events require a certain amount of preparation time. For example, before a **DisplayEvent** can be executed, the bitmap to be shown on the screen must be drawn. If an image is to be presented, it must be loaded from a file. If a sound is to be played, it must also be loaded from a file. All of these things take time, and can potentially impact the timing of an experiment. Ideally, then, such preparations would be carried out before the experiment starts executing. On the other hand, certain aspects of an event may vary from trial to trial (for example, the particular image to be displayed), or even depend on what happens during the previous trial (as in the psychophysics demo). In this case, preparations cannot be made far in advance.

To deal with these issues, FLXLab uses an intelligent update algorithm. If the properties of an event will not vary during the experiment, it is prepared for execution when it is created and updated whenever it is modified. If some of these properties will vary during the experiment (e.g., because they are linked to columns in the stimulus file), then the event is updated prior to the start of each trial. This ensures that any variation in timing affects only the intertrial interval, and not the trial itself.

Advancing to the next item in the stimulus list forms part of this pre-trial update. This means that if a certain property of an event is linked to a column

in the stimulus list, it will be undefined until the beginning of the first trial, and will change only at the beginning of each successive trial.

5.6 Macros

Macros allow you to use a single name to execute a commonly needed set of commands. The basic form of a macro definition is as follows:

```
DefineMacro name
command 1
command 2
...
command N
EndMacro
```

Once a macro has been defined, you can use *name* like any other command, and it will cause commands 1 through N to be executed.

For more flexibility, you can also use macros with arguments. For example, let's suppose you want to have a macro called `ScreenColor` for setting the color of the screen. Here's how you could do it:

```
DefineMacro ScreenColor
SelectObject screen
Color $1
EndMacro
```

Now if you execute the command `ScreenColor red`, FLXLab will replace every instance of `$1` in the macro with `red` before executing the commands. Macros can have up to four arguments, referred to by `$1`, `$2`, `$3`, and `$4`. It is important to note that macro arguments work by simply substituting text. Consider the following version of our `ScreenColor` macro:

```
DefineMacro ScreenColor
SelectObject screen
Color $1$2
EndMacro
```

If you now execute the command `ScreenColor dark magenta`, FLXLab doesn't care if `dark` and `magenta` are colors or even variables. It simply sticks these two strings together to form `darkmagenta`, and uses this as the argument for the `Color` command. As long as `darkmagenta` is a color, you won't get any errors.

6 Command Reference

AddEvent *event condition* Makes *event* a sub-event of the currently selected event, with the trigger condition *condition*. The currently selected

event must be a TrialEvent, BlockEvent, or ExperimentEvent. The trigger condition may be omitted, in which case it defaults to **repeat 1** (if the currently selected event is a TrialEvent or ExperimentEvent) or **repeat** (if the currently selected event is a BlockEvent).

AddFontDirectory *path* Adds all compatible fonts in *path* to the list fonts that may be used when displaying text.

AddObject *object position alignment* Adds *object* to the the currently selected graphics object, which must be either a DisplayEvent or an Object-Box. The *position* argument indicates where on the screen the object should be drawn. The *alignment* argument specifies what part of the object should be aligned with the position. For example, `AddObject foo top top` will position the object so that its top edge is aligned with the top of the screen. You may omit the *alignment* argument or both the *position* and *alignment* arguments; the default values for both are **center**.

AddSearchDirectory *path* Adds *path* to the list of directories that FLXLab searches in to find things like script files, image files, sound files, and modules.

AlertDialog *message* Displays a dialog box containing *message* on the screen.

Antialiased *setting* Sets text antialiasing for the current graphics object to *setting*, which should be either **true** or **false**. The *setting* argument may be omitted, in which case it defaults to **true**. If no graphics object is currently selected, sets the default value for antialiasing. The initial default value is **true**. If you are writing black text onto a white background or vice versa, antialiasing will generally improve the appearance of your text. However, if either your text or your background is colored, it can give odd results, so you might want to turn it off. The reason for this is that FLXLab draws the text before a trial begins, but the color of the background may depend on events during the trial. Thus, FLXLab makes an educated guess about the color of the background based on the color of the text: With dark text it assumes a white background, with light text it assumes a black background. If these assumptions are very wrong, then antialiased text won't look very good.

AppendData *setting* By default, when FLXLab opens a data file, it will overwrite any previously existing file with the same name. This command allows you to specify that new data should be appended to the existing file instead. The *setting* should be either **true** or **false**. If *setting* is omitted, it defaults to **true**.

BlockEvent *name condition* Creates a new BlockEvent named *name*, with the continue condition *condition*. If *condition* is omitted, it defaults to **repeat 1**. A BlockEvent will continue cycling through its sub-events as long as the continue condition evaluates to true.

BoxColor *color* Some graphics objects consist of a box or rectangle along with something displayed inside it (e.g., a `TextBoxObject`). This command allows you to set the color of the rectangle part of the object.

BufferData *setting* By default, FLXLab stores up messages sent to the data file, and only actually writes them to disk at certain times (e.g., at the end of every trial). This is done because excessive disk access during a trial can affect the timing of events. However, when a script is not behaving as expected, it can sometimes be useful to write debugging messages to the data file as soon as they are generated, which is what this command allows you to do. The *setting* argument should be either `true` or `false`; if it is omitted, it defaults to `true`.

ClearScreenEvent *name* Creates an event named *name* to clear the screen to the background color. By default, FLXLab does this at the start of every trial, but if you want to do it at other times, you will need to use a `ClearScreenEvent`.

Color *color* Sets the color of the currently selected graphics object to *color*. If no graphics object is currently selected, sets the default color.

CommandEvent *text* Creates an event to execute the command specified in *text*.

Counter *name value* This command allows you to create an integer variable that can be used as an argument to commands, just like any other variable. The variable will be called *name*, and you can specify its initial value with the *value* argument. If *value* is omitted, it defaults to 0. Together with the `Increment` command, this provides more sophisticated control over the execution of an experiment. For example, you could create a counter to keep track of the number of trials where a subject has made the correct response, and stop the experiment when this reaches a certain value. It can also be used in psychophysical experiments; see the demo on this.

DataColumn *value* Adds *value* to the list of columns associated with a `DataEvent`.

DataEvent *name* Creates an event named *name* to record information to the data file. The particular information to record is specified with one or more `DataColumn` commands.

DataTimeStamp *setting* FLXLab has the capability to preface each line in the data file with the time since the program began running. This command allows you to turn this feature on. The *setting* should be either `true` or `false`. If *setting* is omitted, it defaults to `true`.

DefineColor *name red green blue* Defines a new color named *name* with *red*, *green* and *blue* specifying the color in RGB format. These values should be between 0 and 255. FLXLab provides five predefined colors: white (255 255 255), black (0 0 0), red (255 0 0), green (0 255 0), and blue (0 0 255).

- DefineMacro** *name* Begins the definition for a macro called *name*. See the section on macros for more details.
- DefinePosition** *name x y* Defines a new position named *name*. The values for *x* and *y* can be specified in terms of pixels (e.g., 362) or as a percentage of the screen size (e.g., 30%).
- DelayEvent** *name duration* Creates a new event called *name* that will delay the execution of the next event by *duration* milliseconds. This is the primary way to create an interval between two stimuli.
- DisplayEvent** *name position alignment* Creates a new event called *name* for presenting graphics, text, or images on the screen. Use the `AddObject` command to add graphics objects to the display thus created. Usually you can omit the *position* and *alignment* arguments; see the description of `ObjectBox` for details on what they are used for.
- EditDialog** *variable_name message default_text* Displays a dialog box with the prompt given by *message*, and box which the user can enter text into. You can specify the initial contexts of this box with *default_text*. When the dialog is closed by clicking “OK”, the text in this box will be placed in a variable called *variable_name*.
- EllipseObject** *name* Creates a new `EllipseObject` called *name*. You can specify the properties of the ellipse using the `Size`, `Color`, and `Filled` commands.
- EndMacro** Ends the definition of a macro. See the section on macros for more details.
- ExperimentEvent** *name condition* This command is a synonym for `GroupingEvent`.
- Filled** *setting* This specifies whether the current graphics object should be drawn as an outline or filled. The *setting* argument should be either `true` or `false`. It may also be omitted, in which case it defaults to `true`. The current graphics object must be a `RectangleObject`, `EllipseObject`, `TextBoxObject`, `ObjectBox`, or `DisplayEvent`. If no graphics object is currently selected, sets the default value for filling shapes. The initial default value is `false`.
- Font** *typeface size* Sets the font for the current graphics object. The *typeface* argument should be the name of the font with spaces removed, optionally followed by one or both of the modifiers `Bold` or `Italic`, e.g., `BookAntiquaBoldItalic`. FLXLab is distributed with three fonts, URW Bookman, URW Nimbus, and URW Palladio. Other fonts on your system can be accessed using the `AddFontDirectory` command. The size can be any positive integer, and is specified in points. If no graphics object is currently selected, sets the default font. The initial default font is 48 point URW Palladio.

GroupingEvent *name condition* Creates a new GroupingEvent named *name*, with the continue condition *condition*. If *condition* is omitted, it defaults to repeat 1. A GroupingEvent will continue cycling through its sub-events as long as the continue condition evaluates to true.

If *condition command* If *condition* evaluates to true, executes *command*, otherwise does nothing.

ImageEvent *name image_file* Creates a DisplayEvent that will display the image contained in *image_file*. Use this if you only need to display one image at a time. This command is defined as a macro.

ImageObject *name image_file* Creates a graphics object named *name* that will draw the image contained in *image_file*. Use this if you need to combine several images into one display.

ImagePatternEvent *name image_file* Creates a DisplayEvent that will use the image contained in *image_file* as a template (see ImagePatternObject). Use this if you only need to display one image at a time. This command is defined as a macro.

ImagePatternObject *name image_file* Creates a graphics object named *name* that will use the image contained in *image_file* as a template. Areas that are light in this image will be drawn in the background color; areas that are dark will be drawn in the color of the object (this can be set with the Color command).

Increment *variable value* Increments the integer variable *variable* by *value*, which can be positive or negative.

JoinStrings *name string_sequence separator* Creates a new variable named *name* that holds the string resulting from concatenating all the strings in *string_sequence*. For example, suppose the name of your data file is formed by appending .txt to the value of the variable \$subject_id. You could build this file name with the following command:

```
JoinStrings data_file "$subject_id .txt"
```

Note that the sequence of strings must be surrounded by quotes, so that it gets treated as a single argument. By default, the strings are simply concatenated together. However, if you want to insert a certain character between them, you can do that by providing the optional *separator* argument, e.g.,

```
JoinStrings data_file "$subject_id data.txt" " "
```

If the subject id was subject1, this would result in the string subject1 data.txt.

Justification *value* Sets the justification for any text associated with the current graphics object. The *value* argument should be LEFT, CENTER, or RIGHT. If no graphics object is currently selected, sets the default justification. The initial default is CENTER.

LabelListColumn *column label* Specifies that *label* will be used to refer to the current value in the column indicated by *column*. The *column* argument should be a number between 1 and the number of columns in the current stimulus list file.

LineObject *name start end* This is one of two methods FLXLab provides to draw a line. Creates a graphics object named *name* that will draw a line from the position indicated by *start* to the position indicated by *end*. The start and end points must be positions that were previously defined by DefinePosition, or one of the pre-defined positions (e.g., topleft, top, center, bottomright, etc.).

LineWidth *width* Sets the line width of the current graphics object to *width* pixels. If no graphics object is currently selected, sets the default line width. The initial default line width is 1. This command can be used with a LineObject, VectorObject, RectangleObject, EllipseObject, TextBoxObject, or ObjectBox.

LoadTextFromFile *variable file_name* Creates a variable named *variable* which holds the entire contents of the file indicated by *file_name*. Note that *file_name* can be a variable itself, such that you can have the text change from trial to trial.

ObjectBox *name position alignment* An ObjectBox is a container that can be used to group multiple graphics objects together, much like a GroupingEvent allows you to group multiple events together. Normally, you do not need to create or modify an ObjectBox explicitly, as it is implicitly created whenever you create a DisplayEvent. However, you may sometimes want a finer degree of control over how a display is constructed, hence this description.

By default, ObjectBoxes are the same size as the screen. However, you can resize an ObjectBox using the Size command, in which case you might want to specify where on the screen to put the display. This can be done with the *position* and *alignment* arguments, which work in the same way as for the AddObject command. Either *alignment* or both *position* and *alignment* can be omitted; both arguments default to center. Because DisplayEvents contain an ObjectBox, all of this applies to DisplayEvents as well.

You can use the AddObject command with ObjectBoxes, just like with DisplayEvents. ObjectBoxes may also be nested, i.e., you can add one or more ObjectBoxes to a DisplayEvent or even to another ObjectBox. This facilitates positioning graphics objects relative to a subpart of the

full screen. It also helps you reuse a collection of objects when building up more complex displays.

PlaySoundEvent *name sound_file* Creates an event called *name* which will play the sound contained in *sound_file*. Note that *sound_file* can be a variable, so it's possible to have the sound file change from trial to trial.

ReadScriptFile *file_name* Reads the script file specified by *file_name* and executes the commands contained therein.

RecordBits *bits* Specifies how many bits per sample to use when recording sound with a RecordSoundEvent. Typical values are 8, 16 and 24. More bits means better quality but bigger sound files. If no event is currently selected, sets the default number of bits per sample. The initial default is 16.

RecordBufferSize *size* Specifies the size in bytes of the memory buffer to use when recording sound with a RecordSoundEvent. If no event is currently selected, sets the default number of channels. The initial default is 500,000 bytes (about 488Kb). This buffer should be large enough to hold the longest sound you might record on any given trial. For example, if you want to be able to record 10 seconds of sound in stereo at a 22050 sampling rate with 24 bit samples, this would require $10 * 2 * 22050 * 24 / 8 = 1323000$ bytes (the dividing by 8 being to convert bits to bytes).

RecordChannels *channels* Specifies how many channels to use when recording sound with a RecordSoundEvent. Typical values are 1 (mono) and 2 (stereo). If no event is currently selected, sets the default number of channels. The initial default is 1.

RecordRate *rate* Specifies how many samples per second to use when recording sound with a RecordSoundEvent. In theory you can specify any rate you want, but most sound cards only support certain rates. CD quality is 44100; 22050 is noticeably lower quality, but adequate for most purposes. If no event is currently selected, sets the default sampling rate. The initial default is 22050.

RecordSoundEvent *name file_name* Creates an event called *name* which will record sound data to the file indicated by *file_name*. Note that this event works like a switch: The first time it is executed, it starts recording. The next time it is executed, it stops recording and writes the sound data to a file. Note that *file_name* can be a variable, so it's possible to record to different files on each trial.

RecordToFile *message_type setting* FLXLab has the capability of writing a great deal of information to the data file in addition to the data. This command allows you to configure what types of information are recorded. The *message_type* argument must be one of the types listed below. The

setting argument should be either `true` or `false`; it may also be omitted, in which case the default is `true`. By default, recording of `ERROR` and `DATA` messages is turned on. Note that you can turn each type of information on or off separately.

<code>ERROR</code>	Error messages.
<code>DATA</code>	Data, i.e., what is written by <code>DataEvents</code> or the <code>WriteData</code> command.
<code>INFO</code>	Records general information about the status of the program.
<code>EVENT</code>	Records a line to the data file every time an event takes place.
<code>SCRIPT</code>	Records a line to the data file every time a script command is executed.
<code>HOOK</code>	Records a line to the data file whenever a hook function executes (see section on hook f
<code>DEBUG</code>	Debugging information
<code>DDEBUG</code>	More detailed debugging information
<code>DDDEBUG</code>	Very detailed debugging information

RectangleObject *name* Creates a new `RectangleObject` called *name*. You can specify the properties of the ellipse using the `Size`, `Color`, and `Filled` commands.

RefreshRate *rate* Attempts to set the rate at which the screen is refreshed. The *rate* argument should be specified in Herz, i.e., refreshes per second. This may or may not work with your operating system and video card. If you really need to change the refresh rate, there's usually a way to do this through the operating system (i.e., in Windows it's via the Control Panel).

ResetDataTime By default `FLXLab` measures reaction times from the start of the current compound event (typically this means from the beginning of the trial). Oftentimes it is more useful to measure from a particular point in the trial. This command can be applied to any event to indicate that the clock used to measure reaction times will be reset to 0 when that event executes.

ResetEventTime By default `FLXLab` measures time within a compound event from the start of that event. Sometimes it is convenient to measure it relative to some sub-event (e.g., when you want the compound event to end 2000 ms after a certain sub-event takes place). This command can be applied to any event to indicate that the clock used to measure time within a compound event will be reset to 0 when that event executes.

SelectEvent *event* Selects the event indicated by *object*, so that commands affecting events can be applied to it. Events are automatically selected when they are created; you only need this if you have created another event since then, and now want to go back and modify the first event.

SelectObject *object* Selects the graphics object indicated by *object*, so that commands affecting graphics objects can be applied to it. Graphics objects are automatically selected when they are created; you only need this if you

have created another graphics object since then, and now want to go back and modify the first object.

Size *x y* Specifies the size of the currently selected graphics object. The arguments may either be a number of pixels (e.g., 362) or a percentage of the screen size (e.g., 30%). If no graphics object is currently selected, sets the default size for graphics objects. The initial default size is 10 pixels by 10 pixels. This command will work for the same types of objects as with `SetFilled`.

Start *event* Initiates execution of the event specified by *event*.

StimulusList *name file_name* Creates a new stimulus list called *name* and links it to the file *file_name*.

TextBoxEvent This is basically a combination of a `TextBoxObject` and a `DisplayEvent`; executing the event *name* will display the text box.

TextBoxObject *name text* This is basically a combination of a `TextObject` and a `RectangleObject`. The major difference from a `TextObject` is that the default justification is `LEFT` and that text extending beyond one line will wrap around, within the confines of the box. You can specify the size and color of the box using the `Size` and `BoxColor` commands. Also note that the box is filled by default, although you can change this with the `Filled` command.

TextColor Some graphics objects consist of a box or rectangle along with some text displayed inside it (e.g., a `TextBoxObject`). This command allows you to set the color of the text part of the object.

TextEvent *name text* This is basically a combination of a `TextObject` and a `DisplayEvent`; executing the event *name* will display the text.

TextObject *name text* Creates a graphics object named *name* for displaying *text*. The properties of the text can be set with the `Font` and `Color` commands.

TrialEvent *name [condition]* Creates a new `TrialEvent` named *name*, with the continue condition *condition*. If *condition* is omitted, it defaults to `repeat 1`. A `TrialEvent` will continue cycling through its sub-events as long as the continue condition evaluates to true. A `TrialEvent` differs from an `ExperimentEvent` primarily in that certain actions are carried out before and after each trial. Before each trial `FLXLab` advances to the next item in the stimulus file, clears the screen and updates the bitmaps associated with each of the `DisplayEvents` in the trial, among other things. See the section on updates for more details on this. At the end of each trial, `FLXLab` updates the data file (see `BufferData` for an explanation). `FLXLab` also resets the state of any input devices (e.g., keyboard and mouse) both before and after each trial.

UpdateEvent *name object_name* Creates a new UpdateEvent named *name*, which will update the object *object_name* when it is executed. The object would usually be some sort of graphics object, but can also be a variable or an event.

UseDataFile *file_name* Specifies which file to write data to.

UseMicroseconds By default, times in FLXLab are specified in milliseconds. Depending on your computer and operating system, you may also be able to specify times in microseconds (millionths of a second). This command turns this feature on. Note that this affects both times that you specify (e.g., the interval between stimuli) as well as times that FLXLab reports (e.g., reaction times in the data file).

UseMilliseconds If you have previously turn microsecond timing on with UseMicroseconds, you can turn it off again with this command.

UseModule *module* Attempts to load the add-on module *module*. A module named *foo* corresponds to a file named `mod_foo.dll` (Windows) or `mod_foo.so` (Linux). FLXLab comes with a number of add-on modules which are loaded by default; this command is mostly useful if you want to use a 3rd party module.

VectorObject *name angle length* This is one of two methods FLXLab provides to draw a line. Creates a graphics object named *name* that will draw a line starting at a given point, going in the direction indicated by *angle* for the distance indicated by *length*. The angle should be in degrees, and the length in pixels.

VoiceKeyEvent *name* This is the same as RecordSoundEvent, but the file name can be omitted. Use this kind of event if you want to record sound so that you can use the voice key, but don't want to save the sound data to a file.

VoiceKeySensitivity *value* This allows the user to fine-tune how sensitive the voice key is, i.e., how great an increase in sound amplitude is necessary to trigger the voice key. The default value is .3; larger values make the voice key more sensitive, smaller values make it less sensitive.

WaitEvent *name condition* Creates an event named *name* which, when executed, waits for *condition* to evaluate to true. This is typically used to pause the program until the user does something, such as press a key.

WaitForRefresh *setting* Specifies whether a DisplayEvent should attempt to synchronize with the screen refresh. The *setting* argument should be either true or false. It may also be omitted, in which case it defaults to true. If no event is currently selected, sets the default value for this property. The initial default value is false, i.e., FLXLab does not attempt to synchronize with the screen refresh. Currently, synchronizing with the refresh works well under Windows and quite poorly under Linux.

WaitUntilFinished *setting* When a `PlaySoundEvent` executes, it can either play the entire sound before continuing to the next event in the trial, or allow the next event in the trial to execute while the sound plays in the background. This command allows you to choose which behavior you want. The *setting* argument should be either `true` or `false`. It may also be omitted, in which case it defaults to `true`. The currently selected event must be a `PlaySoundEvent` for this command to work. If no event is currently selected, sets the default value for this property. The initial default value is `true`.

WriteData *message* Writes the text in *message* to the data file.