FLXLab 2.3

An experiment generator for the free world

©2009 Todd R. Haskell

API

Contents

1 Overview **3**

1.1 What you need . . . 3

1.2 What you need to know . . . 3

1.3 Structure of this document . . . 3

2 How FLXLab works with modules **4**

2.1 How modules are loaded . . . 4

2.2 How modules add functionality to FLXLab . . . 5

3 How to write a module **5**

3.1 Organization of directories and files . . . 5

3.2 Master source files . . . 5

3.3 Ordinary source files . . . 7

3.4 Initialization functions . . . 7

3.5 Data messages . . . 7

3.6 Adding commands . . . 9

3.7 Adding event types . . . 10

3.8 Building your module . . . 12

4 FLXLab Internals **13**

4.1 The FlxObject Class . . . 13

4.2 The dependency hierarchy . . . 14

4.3 Updating . . . 15

4.4 Static and dynamic objects . . . 17

4.5 Modules, scopes, and scope exit hooks . . . 20

4.5.1 Beginning and ending a scope . . . 21

4.5.2 Deleting a scope . . . 22

1 Overview

FLXLab has an inherently modular design. The core program only provides the basic "engine" for running scripts. All other features are provided by add-on modules. For example, there are separate modules that handle presentation of graphical stimuli, text, and sounds. Another module handles dialog boxes. The start-up screen is implemented as a separate module as well.

Because of this design, it is relatively easy to write new modules that additional features. For example, you could write a module that allows FLXLab to interact with a particular piece of hardware. This has been done for the EyeLink eyetracking system. You could also write a module in order to conduct experiments with a complex trial structure that is not easily implemented in ordinary scripts.

1.1 What you need

- The source code distribution of FLXLab (flxlab-VERSION-src.zip).
- The FLXLab development package (flxlab-VERSION-devel.zip).
- An appropriate command-line compiler and the associated tools for your system. If you use Linux, you should already have these. If you use Windows, you will need to download and install them. The Windows version of FLXLab was built with the MinGW port of the gcc compiler and the MSYS shell, available at http://www.mingw.org.
- If you want to access any of the features that depend directly on Allegro (primarily graphics and text presentation), you may need the DirectX SDK for Allegro, available at http://alleg.sourceforge.net/wip.html. The current version of Allegro as of the time this was written uses DirectX 7. Make sure to download the version for MinGW.

1.2 What you need to know

- A basic knowledge of how to use FLXLab to create an experiment, including such concepts as events, conditions, objects, etc.
- A working knowledge of the C++ programming language.
- How to build applications using command-line programs such as make and gcc/g++.

1.3 Structure of this document

This document is divided into three parts. Section 2 provides a description of how FLXLab works with modules. Section **??** describes how to write, build and

install your own module. Section ?? provides additional technical details about how FLXLab works that are useful for creating more complex modules.

2 How FLXLab works with modules

2.1 How modules are loaded

Within the main FLXLab installation directory, there are two sub-directories that are important for the use of modules. The first of these is called `config`. This directory contains one script file for each module. For example, the graphics module has a configuration script called `graphics_config.flx`. When FLXLab starts up, it reads and executes each script it finds in the `config` directory.

The configuration script for a module typically does three things:

1. Loads any modules that the current module depends on.
2. Loads the current module itself.
3. Carries out any necessary configuration of the current module.

For example, here are the contents of the file `startscreen_config.flx`, which controls the loading of the `startscreen` module:

```
UseModule graphics
UseModule gui

UseModule startscreen

UseEditor "NotePad"
```

The `startscreen` module depends on the `graphics` module and the `gui` module, so the first two lines make sure those modules are loaded before the `startscreen` module itself is loaded. This is necessary because the configuration scripts can be read and executed in any order. It's okay to call `UseModule` on a module more than once; if the module is already loaded, the command will simply be ignored.

Next, the `startscreen` module itself is loaded.

Finally, the module is configured. The start screen provides a button for editing a script; the `UseEditor` command is provided by the `startscreen` module, and specifies what program should be run to do this editing.

The second sub-directory that is important for the use of modules is called `modules`, and it contains the actual module files. Note that the name of the module file is determined by adding `flx` to the beginning and `.dll` to the end

of the module name (for Windows) or `libflx` to the beginning and `.so`) to the end (in Linux). Thus, in Windows the file containing the `graphics` module is called `flxgraphics.dll`.

2.2 How modules add functionality to FLXLab

When a module is loaded using `UseModule`, FLXLab calls an initialization routine within that module. Depending on the module, the initialization routine can do any or all of the following:

- Add a new command. For example, the `startscreen` module adds the command `UseEditor`.
- Add a new variable. For example, the `graphics` module adds the variables `black`, `white`, `red`, etc., corresponding to predefined colors.
- Add a new condition. For example, the typedgui module adds the `key`, `mouse`, and `joystick` conditions.
- Add an event hook. An event hook is a function that gets called before, after, or during a particular type of event. For example, the `graphics` module adds an event hook called `clear_screen` that gets called prior to the beginning of each trial.

A module may optionally define a cleanup routine which is called when the module is unloaded.

3 How to write a module

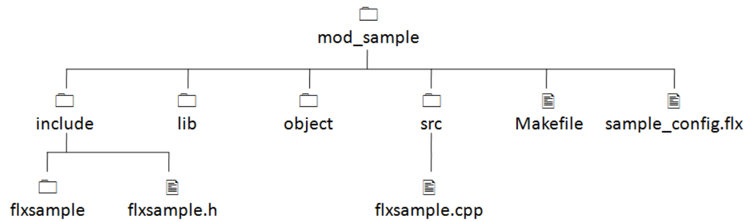
3.1 Organization of directories and files

By convention, the name of the directory containing the source code for a module is formed by adding `mod_` to the beginning of the module name. Thus, the directory containing the source code for the `text` module is `mod_text`. The development package contains source code for a sample module in a directory called `mod_sample`. Figure 1 shows the organization of directories and files for this module.

3.2 Master source files

Each module has a "master" source file, which is named by adding `flx` to the beginning and `.cpp` to the end of the module name. For the sample module, it is called `flxsample.cpp`. This module contains the master initialization function that gets called when the module is loaded, along with a cleanup function that

Figure 1: The layout of a source code directory for a module



is called when the module is unloaded. The cleanup function is optional. By convention, the actual functionality of the module is provided by other source files; in this case they would be `sample_commands.cpp` and `sample_events.cpp`. Each such file should contain its own initialization function. The master initialization function should call the file-specific initialization functions. By convention, these functions are named by adding `flx_` to the beginning of the file name, and replacing `.cpp` with `_init` at the end. So, the initialization function for `sample_commands.cpp` would be `flx_sample_commands_init`. The following shows the contents of `flxsample.cpp` to illustrate how this all works:

```
#define FLX_INCLUDE_MODULE_HEADER
#include <flxbase.h>

extern void flx_sample_commands_init(void);
extern void flx_sample_events_init(void);

/*****************************************************************************/

extern "C" {

FLX_START_MODULE_INIT

  string cur_function="sample_module_init";

  flx_data->write_message(FLX_DATAINFO,cur_function,"Initializing sample module");
  flx_sample_commands_init();
  flx_sample_events_init();

FLX_END_MODULE_INIT

}

/*****************************************************************************/
```

For most simple modules, you can create your own master source file by editing this one as follows:

1. Edit the `extern` declarations near the beginning of the file to correspond to the file-specific initialization functions.
2. Edit the lines where `flx_sample_commands_init` and `flx_sample_events_init` are actually called to call your own functions instead.
3. Change the three locations where you find the word "sample" in a string constant to be the name of your module.

3.3 Ordinary source files

By convention, the code that actually does the "work" of the module is contained in one or more separate source files (rather than being included in the master source file). For a simple module, you may only need one such source file. For example, the startscreen module has only one ordinary source file, called `startscreen.cpp`. For more complicated modules, you may need several ordinary source files.

3.4 Initialization functions

The structure of each ordinary source file is similar. The file should contain a file-specific initialization function, named as described above. The following shows what this function looks like for `sample_commands.cpp`.

```
void flx_sample_commands_init(void){
  string cur_function="flx_sample_commands_init";

  flx_add_command("Power",Power);
  flx_add_command("Factorial",Factorial);

} /* flx_sample_commands_init */
```

3.5 Data messages

In many of the code snippets provided in this document, you can find lines like the following:

```
flx_data->write_message(FLX_DATAINFO,cur_function,"Initializing sample module");
```

These lines send messages to the data file. Within FLXLab, data files are represented by a class object. The variable `flx_data` always points to the current

data object. To send a message to the data file, you can use the `write_message` member function. This function takes three arguments. The first argument is a constant that indicates the type of message. These correspond to the types that can be used with the `RecordToDataFile` command, as shown below:

Message type	Constant for use with `write_message`	Purpose of message
ERROR	FLX_DATAERROR	To indicate to the user that an error has occurred
DATA	FLX_DATADATA	To record data from an experiment
INFO	FLX_DATAINFO	To record information about setup and configuration, such as the day and time a script was executed, the screen resolution and color depth, whether millisecond or microsecond timing is being used, or that a particular module has been loaded
EVENT	FLX_DATAEVENT	To indicate that an event is being executed
SCRIPT	FLX_DATASCRIPT	To indicate that a script command is being executed
HOOK	FLX_DATAHOOK	To indicate that a hook function is being called
DEBUG	FLX_DATADEBUG	To provide low-level diagnostic or debugging information
DDEBUG	FLX_DATADDEBUG	To provide even more detailed debugging information
DDDEBUG	FLX_DATADDDEBUG	To provide extremely detailed debugging information

The use of some of these message types is discussed further in the following sections.

The second argument is a string which indicates the function that is generating the message. In most FLXLab code, a string called `cur_function` is defined at the beginning of each function, and initialized to contain the name of the function. This variable is then passed any time the `write_message` function is called. This approach makes it easier to do things like change the name of a function, combine two functions into one, or split a single function into two functions, since all that is required is to adjust the definition of `cur_function` accordingly.

The third argument is a string which contains the message itself.

Messages generated with the `write_message` function are stored in a message queue. Periodically, FLXLab examines each message in the message queue and determines whether it should be written to the data file or discarded. This decision is determined by the type of the message. If that message type is set to be recorded, it is written to the data file. Otherwise, the message is

discarded. Whether a particular message type is recorded or not is controlled by the RecordToDataFile command.

Note that this decision is based on the recording settings at the time the message queue is processed, not the settings at the time the message was originally generated. In general, the message queue is processed at the beginning and end of a *session*, and at the end of each execution of a TrialEvent. A session begins the first time a compound event is executed. Events of type TrialEvent, BlockEvent, ExperimentEvent and GroupingEvent are all compound events. The session ends when that same compound event finishes executing. If you want the message queue to be processed at some other time, you can force this to happen with the flush member function, which takes no arguments:

```
flx_data->flush()
```

3.6 Adding commands

The initialization function shown earlier adds two commands to FLXLab, Power and Factorial. Commands are added with the flx_add_command function. This function takes two arguments. The first argument is the string that is used to invoke the command in a script file. This string should not contain any white space. The second argument is the name of a function that will be called when the command is executed.

The definition for the Power function is shown below. This function takes value and raises it to the power specified by exponent.

```
bool Power(long *value,long *exponent){
  string cur_function="Power";
  long base, i;

  if(*exponent>=0){
    base=*value;
    *value=1;
    for(i=0;i<*exponent;i++){
      *value*=base;
    }
    flx_data->write_message(FLX_DATASCRIPT,cur_function,"Raising variable to the "+flx_conve
    return true;
  } else {
    flx_data->write_message(FLX_DATAERROR,cur_function,"Exponent for this command must be no
    return false;
  }

} /* Power */
```

Functions that are to be used as commands will be referred to as command

functions. Command functions must always return a `bool`. The function should return `true` if the command was successful, and `false` if it was not. If the function returns `false`, FLXLab will display an error message indicating that the command has failed.

Command functions can take from 0 to 4 arguments. All arguments must be pointers. This allows a command function to modify one or more of the arguments passed to it, as the `Power` function modifies the `value` argument. The type being pointed to must be one that FLXLab knows how to work with. The basic types FLXLab understands are `long`, `bool`, `float`, and `string`. In addition, arguments can be FLXLab-defined types such as `FlxEvent`, `FlxCondition` and `FlxGraphicsObject`, which are discussed below.

The arguments passed to the function correspond to the arguments provided with the command in the script file. For each type that can be used as an argument, there is a function that takes the string that appears in the script, and converts it to a pointer to the appropriate type. If the argument is a variable, FLXLab converts it to a pointer to the value of the variable. If the argument is a literal value, FLXLab essentially creates an anonymous variable with that value, and passes a pointer to it. Thus, a command function can always safely modify any argument passed to it, though the effects of the that modification may not be evident in the case of an anonymous variable. For example, both of the following instances of the `Power` command will execute without errors:

```
Counter foo 2
Power foo 2
Power 2 2
```

In the first case, the result of the computation will be stored in the variable `foo`. In the second case, however, there is no way to access the result of the computation.

Each command function should generate a message of type `FLX_DATASCRIPT` upon successful execution of the command. This message should generally convey some information about what the command has done, as in the `Power` function shown above. A command function should also generate a message of type `FLX_DATAERROR` if execution of the command was unsuccessful. This message should provide information about why the command failed. Command functions may optionally generate messages of one of the debugging types to provide detailed information about what the function is doing.

3.7 Adding event types

To add a new type of event to FLXLab, you first need to define a class derived from `FlxEvent`, as in this example taken from `sample_events.cpp`:

```
class IncrementEvent : public FlxEvent {
  long *d_counter;
public:
  IncrementEvent(string name,long *counter) : FlxEvent(name), d_counter(counter) {}
  ~IncrementEvent(void) {}
  void execute(void);
};

void IncrementEvent::execute(void){
  string cur_function="IncrementEvent::execute";

  flx_data->write_message(FLX_DATAEVENT,cur_function,d_name);
  this->do_generic_event_processing();
  (*d_counter)++;
}
```

All events must have, at a minimum, a constructor, a destructor, and a member function called `execute` which is called when the event is to be executed. Every `execute` function should do at least two things. The first is to make a call to `write_message` indicating that that an event is being executed. The second is to call the member function `do_generic_event_processing`. This function does some internal housekeeping that needs to be done for all events.

Once you have defined your event class, you need to define a command function that creates an instance of this class. All command functions that create an event should take the name of the event as the first argument. You may optionally use additional arguments to specify particular characteristics of the event, as with the variable `counter` in the example below:

```
bool NewIncrementEvent(string *name,long *counter){
  string cur_function="NewIncrementEvent";
  IncrementEvent *iep;

  flx_data->write_message(FLX_DATASCRIPT,cur_function,"Creating new IncrementEvent '"+*name-
  iep=new IncrementEvent(*name,counter);
  flx_add_scalar_source(iep,counter);
  return true;

} /* NewIncrementEvent */
```

All command functions that create an event need to do two things. First, as with all command functions, make sure your command function generates a message indicating what the command is doing. Second, create an instance of your class. As long as your class is derived from the type `FlxEvent`, the class constructor will take care of registering the new event with FLXLab so that it can be referred to in the script.

In some cases your event class may have data members which are set based on arguments to the command. In this example, the class `IncrementEvent` has a data member called `d_counter`, which is set to be the second argument passed to `NewIncrementEvent`. In FLXLab, this sort of relationship between two objects is called a dependency: The behavior of the event pointed to by `iep` depends on the value pointed to by `counter`. The function `flx_add_scalar_source` tells FLXLab about the existence of such a dependency. The first argument should be a pointer to your event; the second argument should be a pointer to the value it is dependent on. The second argument can be a pointer to any basic type used by FLXLab, i.e., `long`, `bool`, `float` and `string` (even though `string` is technically a class type). There is another function `flx_add_object_source` that can be used when there is a dependency between an event and a FLXLab-defined type, such as another event. Dependencies are discussed in much more detail in a separate section below.

Finally, you need to register your command function with FLXLab. This should be done in the initialization function for the source file where your event class is defined. This is done in exactly the same way as for any other command function, as shown below:

```
void flx_sample_events_init(void){
  string cur_function="flx_sample_events_init";

  flx_add_command("IncrementEvent",NewIncrementEvent);

} /* flx_sample_events_init */
```

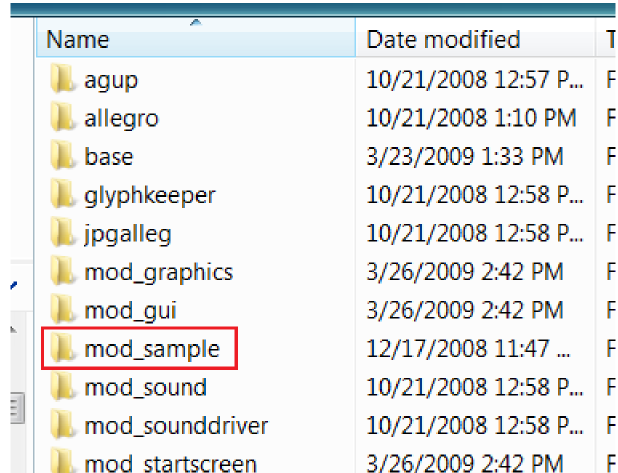
3.8 Building your module

In order to actually build your module, you need have a file named `Makefile` in the module directory. Since all modules are built in essentially the same way, most makefiles for modules simply execute a default module makefile. Thus, the contents of `Makefile` for `mod_sample` are just this:

```
include ../Makefile.modules
```

For this to work, you will need to put your module directory inside the FLXLab source directory, alongside the directories for the basic modules provided with FLXLab, as shown in Figure 2. As long as you have already built the basic FLXLab program, so that the necessary libraries have been created, you should be able to build your module by simply typing `make` on the command line. If the module builds successfully, you can then install it by `make install`. Note that this will install it in the same place as the rest of the FLXLab program. It should also install the config file for your module (as long as you have named it properly) so that your module will be automatically loaded when you run FLXLab.

Figure 2: Placing a new module directory inside the FLXLab source directory



For a simple module, such as a module to let FLXLab interface with a particular piece of hardware, this is probably all the information you need to know. If you want to write a more complex module, or if you just want to better understand how FLXLab works, the remainder of this document provides a detailed description of some of the internal functioning of the program.

4 FLXLab Internals

This section of the document describes some of the internal workings of FLXLab. Knowing this information is not necessary for writing simple modules, but may be useful for troubleshooting, debugging, or if you want to write more complex or sophisticated modules. This section only describes certain aspects of the program; additional information may be added to this section with future releases.

4.1 The FlxObject Class

To provide a consistent interface for working with objects and variables created by a script, FLXLab makes use of a class called FlxObject. Events, graphics objects, positions, colors, lists, etc. are all represented by classes derived from this base class. All objects derived from FlxObject have several common properties that can be accessed through the FlxObject interface:

- A name, which can be used to refer to the object in a script.

- A value, which is what gets passed to a command when the name is used as an argument.
- A list of dependencies (discussed further below).
- A scope name (also discussed below).
- An update function (also discussed further below).

FLXLab maintains a list of all FlxObjects that are created. The constructor for FlxObject adds the object being created to this list. This functionality is inherited by any derived classes. For example, creating an object of type IncrementEvent (as discussed above) will result in that object being added to the list.

Once an object is in the list, you can look it up based on its name using the `flx_get_object` function. The most common reason for looking up an object in the list is in order to convert an argument in a script into a pointer that can be passed to the command function. In this context, we don't simply want any object derived from FlxObject that has a given name. Rather, we are looking for an object of a particular type. This is handled by making `flx_get_object` a template function, where you specify the type you are looking for as the template parameter, like this:

```
FlxEvent *ep;

ep=flx_get_object<FlxEvent>("my_event");
```

If there is an object in the list named `my_event`, and this object is a FlxEvent or can be cast to that type, the function will return a pointer to that object. If there is no object with that name, or if there is such an object but it cannot be cast to FlxEvent, the function will return NULL.

4.2 The dependency hierarchy

In a typical experiment, the stimulus presented to a subject varies from trial to trial. For example, suppose the stimulus consists of a word which is different on each trial. One would usually put the word to be presented on each trial in a column of the stimulus list, and then create a TextObject which is linked to this column. The TextObject would then be attached to a DisplayEvent which handles the actual drawing of the word on the screen.

In FLXLab, this state of affairs is captured by the notion of a *dependency*. Consider the following excerpt from a script:

```
LabelListColumn 1 stimulus_word
```

```
TextObject stimulus $stimulus_word

DisplayEvent show_stimulus
AddObject stimulus
```

In order to know what to draw on the screen, the `show_stimulus` event needs to access the object `stimulus`. Thus, we say that `show_stimulus` is dependent on `stimulus`. Note that a dependency relationship is directional; we refer to the object which needs to access another object as the dependent, and the object that is accessed as the source. In this example, `show_stimulus` would be the dependent, and `stimulus` would be the source. Note that there is another dependency in this script as well: `stimulus` is dependent on `stimulus_word`. FLXLab keeps track of dependencies by the use of a dependency hierarchy. This is implemented by having each FlxObject store a list of any dependent objects and a list of any source objects.

You can tell FLXLab about a dependency relationship using the two functions `flx_add_scalar_source` and `flx_add_object_source`. Both commands take a pointer to the dependent as the first argument. The dependent must be a class derived from `FlxObject`. The second argument is a pointer to the source. Use `flx_add_scalar_source` if the source is one of FLXLab's basic types, i.e., `long`, `bool`, `float` or `string`. Use `flx_add_object_source` if the source is a class derived from `FlxObject`, like a `FlxEvent` or `FlxGraphicsObject`. Both functions will cause the dependent to get added to the list of dependents for the source, and the source to get added to the list of sources for the dependent.

As a final observation, note that we can distinguish between a direct source and an indirect source. This distinction is analogous to the distinction between a direct base class and an indirect base class in C++, i.e., an indirect source is a source of a source. In the example above, we would say that `stimulus` is a direct source of `show_stimulus`, while `stimulus_word` is an indirect source.

4.3 Updating

The primary reason for having a dependency hierarchy has to do with updating objects. All objects derived from FlxObject have a member function called `update`. Updating is used in part to ensure that changes in a source object are reflected in the behavior of any dependent objects. For example, the value of `stimulus_word` will change on each trial. The `stimulus` object is responsible for actually drawing the bitmap of the word, so it's behavior must also change. Finally, `show_stimulus` is responsible for creating the overall bitmap of the entire display, and so this must also change on each trial. Thus, whenever a change occurs to an object, we need to make sure the `update` function gets called for each of its dependents. The dependency hierarchy provides a way to do that.

The actual updating gets carried out by a member function called `update_object_and_dependents`.

This function does two things:

1. Builds a list of all the objects that need to be updated.
2. Calls the update function for all of these objects, in order.

For purposes of step 1, we start off with an empty list. This list is then passed to a member function called `add_to_update_list`. This function also does two things:

1. Adds the current object to the end of the list. If it is already in the list, move it to the end.
2. Calls `add_to_update_list` on all the dependents of the current object.

Let's look at how this works with a concrete example. Consider the following excerpt from a script:

```
StimulusList my_stimuli "stimuli.txt"
LabelListColumn 1 red
LabelListColumn 2 green
LabelListColumn 3 blue

DefineColor my_color $red $green $blue

RectangleObject my_rect
Color my_color
```

We can represent the dependency relationships in this situation as follows:

Source	Dependent
my_stimuli	red
my_stimuli	green
my_stimuli	blue
red	my_color
green	my_color
blue	my_color
my_color	my_rect

The initial change in this scenario is that the stimulus list advances to the next item in the list. Thus, we call `update_object_and_dependents` for `my_stimuli`, and it in turn calls `add_to_update_list` to build the list of objects that needs to be updated. This will first add `my_stimuli` to the list. It will then proceed to recursively call `add_to_update_list` for `red`, `green` and `blue`. When it is called for `red`, it will be recursively called for `my_color`, and so on. After we're done with the process for `red`, the update list will look like this:

my_stimuli
red
my_color
my_rect

If we just proceeded to do the process for `green` next, we would end up with a list that looks like this:

my_stimuli
red
my_color
my_rect
green
my_color
my_rect

We really only need to update each object once, after all of its sources have been updated. Thus, if `add_to_update_list` gets called for an object that is already in the update list, instead of adding the object to the list again, it gets moved to the end of the list. The result is that the final update list actually looks like this:

my_stimuli
red
green
blue
my_color
my_rect

Once the update list has been generated, the `update` member function is called for each object, in order. This ensures that moving to the next item in the stimulus list results in the appropriate change in the color of `my_rect`.

4.4 Static and dynamic objects

For purposes of updating, FLXLab distinguishes between two types of objects. Static objects do not change from trial to trial, while dynamic objects do change from trial to trial. The updating process works somewhat differently for the two types of objects.

Internally, FLXLab defines the difference between the two types of objects as follows. Technically, it is not advancing to the next item in the stimulus file that triggers the updating described above. Rather, FLXLab defines a special object called `flx_dependency_root`, and calls `update_object_and_dependents` for this object at the beginning of each trial. Thus, making `flx_dependency_root` a source for an object will cause that object to be updated before each trial. The current stimulus list always has `flx_dependency_root` as a source. An object is considered dynamic if it has `flx_dependency_root` as a source, either direct or indirect. Otherwise, it is considered static.

Let's take another look at the script commands discussed above in the context of updating; they are repeated here for convenience.

```
StimulusList my_stimuli "stimuli.txt"
LabelListColumn 1 red
LabelListColumn 2 green
LabelListColumn 3 blue

DefineColor my_color $red $green $blue

RectangleObject my_rect
Color my_color
```

The object `my_stimuli` is dynamic, since a stimulus list always has `flx_dependency_root` as a direct source, as discussed above. The objects `red`, `green` and `blue` will also be dynamic, since they have `flx_dependency_root` as an indirect source. The same applies to `my_color` and `my_rect`.

Now consider a slightly different set of script commands:

```
DefineColor my_color 70 140 210

RectangleObject my_rect
Color my_color
```

In this case both `my_color` and `my_rect` would be static, since they only depend on constants.

The general rule for updating is that dynamic objects are only updated at the beginning of each trial, while static objects are typically updated when there is a change in their list of sources. This is the sort of situation that occurs when you use the `AddObject` command:

```
DisplayEvent show_stimulus
AddObject stimulus
```

The object `stimulus` gets added as a source of `show_stimulus`. If `show_stimulus` is determined to be static, then `update_object_and_dependents` will be called on `show_stimulus` as the last step in executing the `AddObject` command.

This situation also occurs when you set the properties of a graphics object:

```
RectangleObject my_rect
Color red
```

The object `red` will be added as a source of `my_rect`; if `my_rect` is static, this will trigger an update. In either of these situations, adding a source will not trigger an update if the object with the new source is dynamic.

Note that an object can change from static to dynamic during the course of a script, as illustrated by the following sequence of commands:

```
StimulusList my_stimuli "stimuli.txt"
LabelListColumn 1 red
LabelListColumn 2 green
LabelListColumn 3 blue

DefineColor my_color $red $green $blue

RectangleObject my_rect
Size 100 200
Color my_color
```

After the completion of the `Size` command, `my_rect` is static, as its only sources are the two constants `100` and `200`. Thus, it will get updated at that point. However, after the completion of the `Color` command, `my_rect` becomes dynamic, since it now has a stimulus list as an indirect source. Thus it will not be updated again at that point; rather, it will be updated the next time a TrialEvent is executed.

The distinction between static and dynamic objects helps address two issues. For objects which are dependent on the contents of a stimulus file, it doesn't make sense to update them when a source is added, because they will only need to be updated again when the value of that source changes at the beginning of the first trial. On the other hand, for an object that does not change from trial to trial, such as a fixation cross that is displayed for each trial, it doesn't make sense to update it with every trial, because such updating would be redundant. Furthermore, for events that occur outside of a trial, like an event that displays instructions read from a text file, it is actually necessary to update it before any TrialEvent occurs, or the instructions won't get displayed.

Note that updating of static objects is not automatic. Rather, you need to call the function `flx_process_dependencies` on an object after adding a source in order to trigger an update. For example, the command function `JoinStrings` includes the following sequence of function calls:

```
flx_add_scalar_source(js,string_ptr);
flx_process_dependencies(js);
```

The function `flx_process_dependencies` will trigger an update if `js` is static, and do nothing otherwise.

It isn't always necessary to call this function after adding a source. You only need to call it if adding the source could potentially result in incorrect behavior of the object or its dependents unless they are updated. For example, for the function that creates objects of type `IncrementEvent`, which was discussed

earlier, adding the amount to increment by as a source doesn't require the object itself to be updated, nor will it cause problems with any dependents.

4.5 Modules, scopes, and scope exit hooks

In FLXLab, the lifetime of a particular command, variable, object, etc. is determined by its *scope*. Scopes are used to group together commands, variables and objects that should be deleted at the same time. Note that unlike in programming languages such as C, scopes do not determine the visibility of a particular object; all objects can be accessed by any script from the point where they are created until they are deleted.

The various scopes used by the program form a hierarchy. At the broadest level is a scope called `BASE`. This scope begins when FLXLab starts up, and is deleted when the program exits. Each script file, including configuration scripts, defines another scope. Finally, each module also defines a scope. As an example, a scope hierarchy for running the reaction time I demo is shown below:

```
BASE
  graphics_config.flx.1
    graphics
  gui_config.flx.2
    gui
  sound_config.flx.3
    sound
    soundriver
  sounddriver_config.flx.4
  startscreen_config.flx.5
    startscreen
  text_config.flx.6
    text
  startscreen_dialog
    startscreen_script
      reaction_time_I.flx.7
```

When the program starts up, it finds the script `graphics_config.flx` in the `config` directory and executes it. This begins a scope called `graphics_config.flx.1`, which is contained within the scope `BASE`. (For scopes associated with a particular script file, the name of the scope is the name of the script file plus a numeral. The numeral increases by one for each script file read. This ensures that even if the same script file is read twice, it will result in two different scopes.) This is the scope for any variables or objects created by the configuration script. The graphics configuration script calls `UseModule` to load the graphics module, and when that module loads it defines another scope called `graphics`, which is contained within the scope `graphics_config.flx.1`. This is the scope of any commands, variables, etc. that are added by the module. Once the graphics

module is loaded, the scope `graphics` ends. Any additional objects or variables created by the configuration file will again have scope `graphics_config.flx.1`. At the end of the configuration script the scope `graphics_config.flx.1` ends, and the current scope reverts back to `BASE`.

The primary purpose of scopes is to provide a mechanism to clean up the variables objects, variables, and commands created by scripts, modules and configuration files. For example, if a module `awesome` adds a new command `CoolEvent`, then that command needs to be removed when the module is unloaded. This is accomplished by keeping a list of all commands along with the scope for each one. Thus, `CoolEvent` will be associated with the scope `awesome`. When the scope `awesome` is deleted, FLXLab will automatically remove all commands that are associated with that scope. In most cases, the module itself doesn't need to do any cleanup. The same applies to variables, conditions, etc. For the most part, FLXLab creates and ends scopes automatically, and so you don't usually need to worry about this. However, in some cases it may be useful to know how the process actually works.

4.5.1 Beginning and ending a scope

There are four ways in which a new scope begins. First, a new scope automatically begins every time a script file is read. The scope name is the name of the script file plus a numeral, e.g., `graphics_config.flx.1`. All objects, variables, etc. created by the script file will have this scope. The scope ends when the end of the script file is reached. Note that this applies both to regular script files and to configuration files.

A new scope also automatically begins every time a module is loaded. The scope name is the same as the module name, i.e., the scope for the module `mod_graphics` is `graphics`. All objects, variables, etc. created by the module will have this scope. The scope ends once the module has finished loading.

A new scope can also be created by an explicit call to the function `flx_begin_scope`. This function takes one argument, a string indicating the name of the scope. For example, the startscreen module creates a scope called `startscreen_dialog` in this way. This provides a mechanism for modules to control the lifetime of objects created via source code (as opposed to by a script). A scope created in this way does not automatically revert to the previous scope; it will continue until explicitly ended by calling `flx_end_scope` with the name of the scope as an argument.

Finally, a new scope can be created explicitly by the user using the `BeginScope` command, which also takes the name of the scope as an argument. A scope created in this way also does not automatically revert to the previous scope; you need to explicitly end it with `EndScope`, again providing the name of the scope as an argument.

4.5.2 Deleting a scope

When a scope is deleted, all commands, variables and objects associated with that scope are deleted as well. There are three basic ways in which a scope is deleted. First, it can be deleted explicitly with the function `flx_delete_scope`. This would be the usual way of deleting a scope that you create yourself in your code.

Second, deleting a scope also deletes all scopes nested inside it, in the reverse order in which they were created. This is how the FLXLab program cleans up after itself when it is exiting; the scope `BASE` is deleted, and since all other scopes are nested inside it, those scopes are deleted as well. This is also how FLXLab cleans up after a script has been run via the start screen; the startscreen is associated with a scope called `startscreen_script`. The scope for a script run via the startscreen is nested inside this scope. Once the script is done running, the scope `startscreen_script` is deleted, which causes the scope for the script to be deleted as well.

The third way in which a scope can be deleted is with the `EndScope` command. This command both ends and deletes the current scope.