Programming EyeLink® Experiments in Windows (Python)

Beta Version 0.9

*** IMPORTANT: This is a Beta Version of the EDF Access API documentation. Great effort has been made to ensure that this release is as functional and as bug free as possible. ***

Table of Contents

1. Introduction

Performing research with eye-tracking equipment typically requires a long-term investment in software tools to collect, process, and analyze data. Much of this involves real-time data collection, saccadic analysis, calibration routines, and so on.

The EyeLink® eye-tracking system is designed to implement most of the required software base for data collection and conversion. It is most powerful when used with the Ethernet link interface, which allows remote control of data collection and real-time data transfer. The PyLink toolkit includes Pylink module, which implements all core EyeLink functions and classes for EyeLink connection and the eyelink graphics, such as the display of camera image, calibration, validation, and drift correct. The EyeLink graphics is currently implemented using Simple Direct Media Layer (SDL: www.libsdl.org).

1.1 Organization of This Document

Chapter 2 of this document introduces the standard form of an EyeLink experiment and some important EyeLink messages. This will help in understanding the sample code, and is also valuable to programmers in understanding how to write experiment software, and how to port existing experiments to the EyeLink platform. Chapter 3 documents the common functions and classes used in the pylink module. You will rarely need other functions or classes than those listed here. Finally, a detailed analysis of a sample program and code follows. This sample contains code that can be used for almost any experiment type, and can be used as a starting point for your own experiments.

1.2 Getting Started

Please refer to the EyeLink II Installation manual for instructions on how to set up the Subject PC for use with the API. For Python programming, you may refer to the book "*Python in a Nutshell*" by Alex Martelli or any other books on this topic.

2. EyeLink Programming Conventions

The Pylink library contains a set of classes and functions, which are used to program experiments on many different platforms, such as MS-DOS, Windows, Linux, and the Macintosh. Some programming standards, such as placement of messages in the EDF file by your experiment, and the use of special data types, have been implemented to allow portability of the development kit across platforms. The standard messages allow general analysis tools such as EDF2ASC converter or EyeLink Data Viewer to process your EDF files.

2.1 Outline of a Typical Windows Experiment

To help in understanding the sample code, we first introduce the standard form of an EyeLink experiment. This is also valuable to programmers in understanding how to write experiment software, and how to port existing experiments to the EyeLink platform. A typical experiment using the EyeLink eye tracker system will use some variation of this sequence of operations:

- Initialize the EyeLink system, and open a link connection to the EyeLink tracker.
- Check the display mode, create a full-screen window, and initialize the calibration system.
- Send any configuration commands to the EyeLink tracker to prepare it for the experiment
- Get an EDF file name, and open an EDF data file (stored on the eye tracker)
- Record one or more blocks of trials. Each block typically begins with tracker setup (camera setup and calibration), and then several trials are run.
- Close the EDF data file. If desired, copy it via the link to the local computer.
- Close the window, and the link connection to the eye tracker.

For each trial, the experiment will do these steps:

- Create a data message ("TRIALID") and title to identify the trial
- Create background graphics on the eye tracker display
- Perform a drift correction, or display a fixation target
- Start the EyeLink recording to the EDF file
- Display graphics for the trial. (These may have been prepared as a bitmap before the trial).
- Loop until the trial time is up, a button is pressed on the tracker, or the recording is interrupted from the tracker. The display may be changed as required for the trial (i.e. moving a target to elicit saccades) in this loop. Real-time eye-position and saccade/fixation data are also available for gaze-contingent displays and control.
- Stop recording, and handle any special exit conditions. Report trial success or errors by adding messages to the EDF file.
- Optionally, play back the data from the trial for on-line analysis.

This sequence of operations is the core of almost all experiments (see Chapter 4 for an example illustrating the above concept). A real experiment would probably add practice trials, instruction screens, randomization, and so on.

During recording, all eye-tracking data and events are usually written into the EDF file, which is saved on the eye tracker's hard disk, and may be copied to the Subject PC at the end of the experiment. Your experiment will also add messages to the EDF file to identify trial conditions and to timestamp important events (such as subject responses and display changes) for use in analysis. The EDF file may be processed directly using the EyeLink DataViewer application, or converted to an ASC file and processed by your own software.

2.2 Standard Messages

Experiments should place certain messages into the EDF file, to mark the start and end of trials. These will enable the SR Research viewing and analysis applications to process the files. Following these standards will also allow your programs to take full advantage of the ASC analysis toolkit.

Text messages can be sent via the toolkit to the eye tracker and added to the EDF file along with the eye data. These messages will be time stamped with an accuracy of 1 millisecond from the time sent, and can be used to mark important events such as display changes.

Several important messages have been defined for EDF files that are used by analysis tools. The "TRIALID", "SYNCTIME", and "TRIAL OK" messages are especially important and should be included in all your experiments.

- "DISPLAY_COORDS" or "RESOLUTION" followed by four numbers: the left, top, right, and bottom pixel coordinates for the display. It should be one of the first messages recorded in your EDF file. This gives analysis software the display coordinate system for use in analysis or plotting fixations. This is not used to determine the size in visual degrees of saccades: angular resolution data is incorporated in the EDF file for this purpose.
- "FRAMERATE" followed by the display refresh rate (2 decimal places of accuracy should be used). This is optional if stimulus presentation is not locked to the display refresh. Analysis programs can use this to correct stimulus onset time for the vertical position on the stimulus
- "TRIALID" followed by data on the trial number, trial condition, etc. This message should be sent **before** recording starts for a trial. The message should contain numbers ant text separated by spaces, with the first item containing up to 12 numbers and letters that uniquely identify the trial for analysis. Other data may follow, such as one number for each trial independent variable.
- "SYNCTIME" marks the zero-time in a trial. A number may follow, which is interpreted as the delay of the message from the actual stimulus onset. It is suggested that recording start 100 milliseconds before the display is drawn or unblanked at zero-time, so that no data at the trial start is lost.
- "DISPLAY ON" marks the start of a trial's display, and can be used to compute reaction time. It may be preceded by a number may follow, which is interpreted as the delay of the message from the actual stimulus onset. It is used like the "SYNCTIME" message, and may be used in addition to it.
- "TIMEOUT" marks the end of a trial when the maximum duration has expired without a subject response. This can also be used when the trial runs for a fixed time. It is suggested that recording continue for 100 milliseconds after a timeout, in case a fixation or blink has just begun.
- "ENDBUTTON" followed by a number marks a button press response that ended the trial. This can be used in place of the EyeLink button box for local key responses. It is suggested that recording continue for 100 milliseconds after the subject's response.
- The optional message "TRIAL_RESULT" followed by one or more numbers indicates the result of the trial. The number 0 usually represents a trial timeout, -1 represents an error. Other values may represent button numbers or key presses.
- "TRIAL OK" after the end of recording marks a successful trial. **All data** required for analysis of the trial (i.e. messages recording subject responses after the trial) must be written **before** the "TRIAL OK" message. Other messages starting with the word "TRIAL" mark errors in the trial execution.

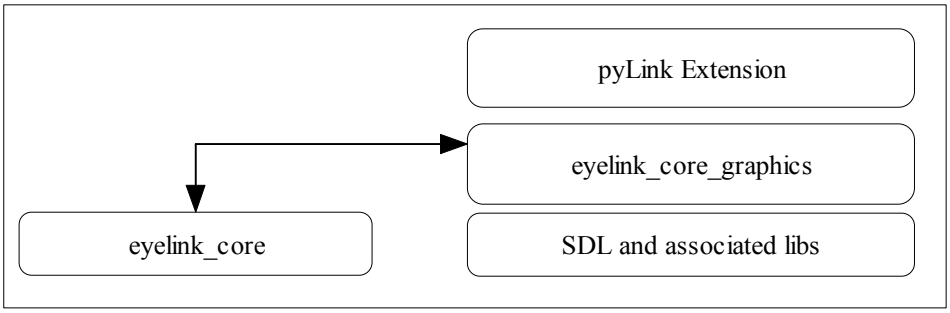
Messages can also be used to timestamp and record events such as calibrations, start and end of drawing of complex displays, or auxiliary information such as audio capture recording indexes. Be careful not to send messages too quickly: the eye tracker can handle about 20 messages every 10 milliseconds. Above this rate, some messages may be lost before being written to the EDF file.

For the ease of data analysis with EyeLink Data Viewer, special viewer message can be added to the EDF file. Examples of these commands include defining the image to overlay, creating interest areas, and specifying trial variables. See "EyeLink Data Viewer User's Manual" for details.

3. Overview of PyLink module

The interaction of the EyeLink tracker and your experiment is fairly sophisticated: for example, in the Setup menu it is possible to perform calibration or validation, display a camera image on the Subject PC to aid in subject setup, and record data with the experiment following along by displaying proper graphics. Large files can be transferred over the link, and should the EyeLink tracker software terminate, the experiment application is automatically terminated as well. Keys pressed on the Subject PC keyboard are transferred to the EyeLink PC and operate the tracker during setup, and buttons pressed on the tracker button box may be used to control the execution of the experiment on the Subject PC.

Pylink, a Python wrapper for Eyelink core API, implements all of the functions and classes for EyeLink connection and graphics, such as the display of camera image, calibration, validation, and drift correct. Each of the above operations required just one line of code in your program. Almost all of the source code you'll write is for the experiment itself: control of trials, presentation and generation of stimuli, and error handling.



The current chapter provides an overview of the classes in the pylink modules, followed by a detailed description of all functions in the pylink module and all classes in this module.

3.1 Classes in PyLink module

Class Name	Parent	Description
EyelinkListener		Eyelink base class that talks directly to the C api. The constructor of this class only, initialize the eyelink connection. However, it does not connect. This class is very useful for all broadcast programs.
Eyelink	EyelinkListener	This class extends off of EyelinkListener. If any address is given, the constructor will connect to the address, otherwise, it connects to the tracker at 100.1.1.1 with sub net mask 255.255.255.0
EyeLinkAddress		Class holds addresses to eyelink nodes
EyelinkMessage	EyelinkAddress	Class used to send and receive messages between eyelink nodes
DisplayInfo		Class used to contain information on a display, including width, height, color bits, and refresh rate;
SampleData		Sample data for left and right eye
Sample		Entire sample data returned by getFloatData
IOEvent		Returned by getFloatData when ever there is an IOEvent
ButtonEvent	IOEvent	Returned by getFloatData whenever there is a Button Event.
MessageEvent		Returned by getFloatData when ever there is a Message Event
StartFixationEvent		Class used to represent start fixation events.
StartSaccadeEvent		Class to represent Start Saccade event
StartBlinkEvent		Class to represent Start Blink event
EndFixationEvent		Class to represent End fixation event. This

		also contains the start fixation data.
EndSaccadeEvent		Class to represent End Saccade event. This also contains the start saccade data.
EndBlinkEvent		Class to represent End Blink event. This also contains the start Blink data.
FixUpdateEvent		Class to represent the fix update event.
ILinkData		Class to represent tracker status information such as time stamps, flags, tracker addresses and so on.

3.2 PyLink Module functions

The PyLink module has the following functions. These functions are mainly responsible for the timing (delay, setting application priority, and getting the current time, etc), calibration settings, and other experimental preparation routines.

3.2.1 alert

alert(*message*)

This method is used to give a notification to the user when an error occurs.

Parameters
<message>: Text message to be displayed.

Return Value
None

Remarks
This function does not allow printf formatting as in c. However you can do a formatted string argument in python.

This is equivalent to the C API **void alert_printf(char *fmt, ...);**

3.2.2 beginRealTimeMode

beginRealTimeMode(*delay*)

Sets the application priority and cleans up pending Windows activity to place the application in realtime mode. This could take up to 100 milliseconds, depending on the operation system, to set the application priority.

Parameters
<delay> an integer, used to set the minimum time this function takes, so that this function can act as a useful delay.

Return Value
None

This function is equivalent to the C API **void begin_realtime_mode(UINT32 delay);**

3.2.3 closeGraphics

closeGraphics()

Notifies the eyelink_core_graphics to close or release the graphics.

Parameters
None

Return Value
None

This is equivalent to the C API **void close_expt_graphics(void);**

3.2.4 currentDoubleUsec

currentDoubleTime()

Returns the current microsecond time (as a double type) since the initialization of the EyeLink library.

Parameters
None.

Return Value
A float data for the current microsecond time since the initialization of the EyeLink library

Remarks:
Same as currentUsec except, this function can return large microseconds. That is the currentUsec can return up to 2147483648 microseconds starting from initialization. The currentDoubleUsec can return up to 36028797018963968 microseconds.

This is equivalent to the C API **double current_double_usec(void);**

3.2.5 currentTime

currentTime()

Returns the current millisecond time since the initialization of the EyeLink library.

Parameters
None.

Return Value
Long integer for the current millisecond time since the initialization of the EyeLink library

This function is equivalent to the C API **UINT32 current_time(void);**

3.2.6 currentUsec

currentUsec()

Returns the current microsecond time since the initialization of the EyeLink library.

Parameters
None.

Return Value
Long integer for the current microsecond time since the initialization of the EyeLink library

This is equivalent to the C API **UINT32 current_usec(void);**

3.2.7 endRealTimeMode

endRealTimeMode()

Returns the application to a priority slightly above normal, to end realtime mode. This function should execute rapidly, but there is the possibility that Windows will allow other tasks to run after this call, causing delays of 1-20 milliseconds.

Parameters

None

Return Value
None

This function is equivalent to the C API **void end_realtime_mode(void);**

3.2.8 flushGetkeyQueue

flushGetkeyQueue()

Initializes the key queue used by getkey(). It may be called at any time to get rid any of old keys from the queue.

Parameters
None

Return Value
None

This is equivalent to the C API **void flush_getkey_queue(void);**

3.2.9 getDisplayInformation

getDisplayInformation()

Returns the display configuration.

Parameters
None

Return Value
Instance of DisplayInfo class. The width, height, bits, and refresh rate of the display can be accessed from the returned value.

For example.
display = getDisplayInformation()
print display.width, display.height, display.bits, display.refresh

3.2.10 msecDelay

msecDelay(*delay*)

Does a unblocked delay using currentTime().

Parameters
<delay>: an integer for number of milliseconds to delay.

Return Value
None

This is equivalent to the C API **void msec_delay(UINT32 n);**

3.2.11 openGraphics

openGraphics(dimension, bits);

Opens the graphics if the display mode is not set. If the display mode is already set, uses the existing display mode.

Parameters
<dimension>: two-item tuple of display containing width and height information.

<bits>: color bits.

Return Value
None or run-time error.

This is equivalent to the SDL version C API **INT16 init_expt_graphics(SDL_Surface * s, DISPLAYINFO *info).**

3.2.12 pumpDelay

pumpDelay(*delay*)

During calls to msecDelay(), Windows is not able to handle messages. One result of this is that windows may not appear. This is the preferred delay function when accurate timing is not needed. It calls pumpMessages() until the last 20 milliseconds of the delay, allowing Windows to function properly. In rare cases, the delay may be longer than expected. It does not process modeless dialog box messages.

Parameters
<delay>: an integer, which sets number of milliseconds to delay.

Return Value
None

Use the This is equivalent to the C API **void pump_delay(UINT32 delay);**

3.2.13 SetCalibrationColors

setCalibrationColors(*foreground_color, background_color*)

Passes the colors of the display background and fixation target to the eyelink_core_graphics library. During calibration, camera image display, and drift correction, the display background should match the brightness of the experimental stimuli as closely as possible, in order to maximize tracking accuracy. This function passes the colors of the display background and fixation target to the eyelink_core_graphics library. This also prevents flickering of the display at the beginning and end of drift correction.

Parameters
<*foreground_color*>: color for foreground calibration target.
<*background_color*>: color for foreground calibration background. Both colors must be a three-integer (from 0 to 255) tuple encoding the red, blue, and green color component.

Return Value
None

This is equivalent to the C API **void set_calibration_colors(SDL_Color *fg, SDL_Color *bg);**

Example:

```
setCalibrationColors((0, 0, 0), (255, 255, 255))
```

This sets the calibration target in black and calibration background in white.

3.2.14 SetCalibrationSounds

setCalibrationSounds(*target, good, error*)

Selects the sounds to be played during do_tracker_setup(), including calibration, validation and drift correction. These events are the display or movement of the target, successful conclusion of calibration or good validation, and failure or interruption of calibration or validation.
Note: If no sound card is installed, the sounds are produced as "beeps" from the PC speaker. Otherwise, sounds can be selected by passing a string. If the string is "" (empty), the default sounds are played. If the string is "off", no sound will be played for that event. Otherwise, the string should be the name of a .WAV file to play.

Parameters
<target>: Sets sound to play when target moves;
<good>: Sets sound to play on successful operation;

<error>: Sets sound to play on failure or interruption.

Return Value
None

This function is equivalent to the C API **void set_cal_sounds(char *target, char *good, char *error);**

3.2.15 setCameraPosition

setCameraPosition(*left, top, right, bottom*)

Sets the camera position on the display computer. Moves the top left hand corner of the camera position to new location.

Parameters
<left>: x-coord of upper-left corner of the camera image window;
<top>: y-coord of upper-left corner of the camera image window;
<right>: x-coord of lower-right corner of the camera image window;
<bottom>: y-coord of lower-right corner of the camera image window.

Return Value
None

3.2.16 setDriftCorrectSounds

setDriftCorrectSounds(*target, good*, *setup*)

Selects the sounds to be played during doDriftCorrect(). These events are the display or movement of the target, successful conclusion of drift correction, and pressing the ESC key to start the Setup menu.

Note:
If no sound card is installed, the sounds are produced as "beeps" from the PC speaker. Otherwise, sounds can be selected by passing a string. If the string is "" (empty), the default sounds are played. If the string is "off", no sound will be played for that event. Otherwise, the string should be the name of a .WAV file to play.

Parameters
<target>: Sets sound to play when target moves;
<good>: Sets sound to play on successful operation;
<setup>: Sets sound to play on ESC key pressed.

Return Value
None

This function is equivalent to the C API **void set_dcorr_sounds(char *target, char *good, char *setup);**

3.2.17 setTargetSize

setTargetSize(*diameter, holesize*)

The standard calibration and drift correction target is a disk (for peripheral delectability) with a central "hole" target (for accurate fixation). The sizes of these features may be set with this function.

Parameters
<diameter>: Size of outer disk, in pixels.
<holesize>: Size of central feature, in pixels. If holesize is 0, no central feature will be drawn. The disk is drawn in the calibration foreground color, and the hole is drawn in the calibration background color.

Return Value
None

This function is equivalent to the C API **void set_target_size(UINT16 diameter, UINT16 holesize);**

3.3 EyeLinkListener Class

EyeLinkListener class implements most of the core EyeLink interface. This includes the simple connection to the eye tracker, sending commands and messages to the tracker, opening and saving a recording file, performing calibration and drift correction, real-time access to tracker data and eye movement events (such as fixations, blinks, and saccades), as well as other important operations.

An instance of EyeLinkListener class can be created by using the class constructor function. For example,

```
try:
    EYELINK = EyeLinkListener()
except:
    EYELINK = None
```

The following sections list the methods of the EyeLinkListener class. All of the methods should be called in the format of: EYELINK.functionName(parameters), where EYELINK is an instance of the EyeLinkListener class.

3.3.1 abort

abort()

Places EyeLink tracker in off-line (idle) mode.

Parameters
None

Return Value
0 if mode switch begun, else link error

Remarks
Use before attempting to draw graphics on the tracker display, transferring files, or closing link. Always call waitForModeReady() afterwards to ensure tracker has finished the mode transition. This function pair is implemented by the EyeLink toolkit library function setOfflineMode().

This function is equivalent to the C API **INT16 eyelink_abort(void);**

3.3.2 acceptTrigger

acceptTrigger()

Triggers the EyeLink tracker to accept a fixation on a target, similar to the 'Enter' key or spacebar on the tracker.

Parameters
None.

Return Value
NO_REPLY if drift correction not completed yet
OK_RESULT (0) if success
ABORT_REPLY (27) if 'Esc' key aborted operation
-1 if operation failed
1 if poor calibration or excessive validation error.

This function is equivalent to the C API **INT16 eyelink_accept_trigger(void);**

3.3.3 applyDriftCorrect

applyDriftCorrect()

Applies the results of the last drift correction. This is not done automatically after a drift correction, allowing the message returned by **getCalibrationMessage()** to be examined first.

Parameters
None

Return Value
0 if command sent ok, else link error.

This function is equivalent to the C API **INT16 eyelink_apply_driftcorr(void);**

3.3.4 breakPressed

breakPressed()

Tests if the program is being interrupted. You should break out of loops immediately if this function does not return 0, if getkey() return TERMINATE_KEY, or if isConnected() method of the class returns 0.

Parameters
None.

Return Value
1 if CTRL-C is pressed, terminalBreak() was called, or the program has been terminated with ALT-F4;
0 otherwise

Remarks
Under Windows XP, this call will not work in realtime mode at all, and will take several seconds to respond if graphics are being drawn continuously. This function works well in realtime mode under Windows 2000.

This function is equivalent to the C API **INT16 break_pressed(void);**

3.3.5 broadcastOpen

broadcastOpen()

Allows a third computer to listen in on a session between the eye tracker and a controlling remote machine. This allows it to receive data during recording and playback, and to monitor the eye tracker mode. The local computer will not be able to send commands to the eye tracker, but may be able to send messages or request the tracker time.

Parameters
None

Return Value
0 if successful.
LINK_INITIALIZE_FAILED if link could not be established.
CONNECT_TIMEOUT_FAILED if tracker did not respond.
WRONG_LINK_VERSION if the versions of the EyeLink library and tracker are incompatible

Remarks
This may not function properly, if there are more than one Ethernet cards installed.

This function is equivalent to the C API **INT16 eyelink_broadcast_open(void);**

3.3.6 close

close()

Sends a disconnect message to the EyeLink tracker.

Parameters
None.

Return Value
0 if successful, otherwise linkerror.

This function is equivalent to the C API **INT16 eyelink_close(int send_msg);** with send_msg parameter 1**.**

3.3.7 closeDataFile

closeDataFile()

Closes any currently opened EDF file on the EyeLink tracker computer's hard disk. This may take several seconds to complete.

Parameters
None.

Return Value
0 if command executed successfully else error code.

This function is equivalent to the C API **int close_data_file(void);**

3.3.8 commandResult

commandResult()

Check for and retrieves the numeric result code sent by the tracker from the last command.

Parameters
None

Return Value
NO_REPLY if no reply to last command
OK_RESULT (0) if OK
Other error codes represent tracker execution error.

This function is equivalent to the C API **INT16 eyelink_command_result(void);**

3.3.9 dataSwitch

dataSwitch(*flag*)

Sets what data from tracker will be accepted and placed in queue. Note: This does not start the tracker recording, and so can be used with broadcastOpen(). It also does not clear old data from the queue.

Parameters
<flags> bitwise OR of the following flags:
RECORD_LINK_SAMPLES - send samples on link
RECORD_LINK_EVENTS - send events on link

Return Value
0 if no error, else link error code

This function is equivalent to the C API **INT16 eyelink_data_switch(UINT16 flags);**

3.3.10 doDriftCorrect

doDriftCorrect(x, y, draw, allow_setup)

Performs a drift correction before a trial.

Parameters:
<x, y>: Position (in pixels) of drift correction target.
<draw>: If 1, the drift correction will clear the screen to the target background color, draw the target, and clear the screen again when the drift correction is done. If 0, the fixation target must be drawn by the user.
<allow_setup>: if 1, accesses Setup menu before returning, else aborts drift correction.

Return value
0 if successful, 27 if 'Esc' key was pressed to enter Setup menu or abort

This is equivalent to the C API **int do_drift_correct(int x, int y, int draw, int allow_setup);**

3.3.11 doTrackerSetup

doTrackerSetup(position)

Switches the EyeLink tracker to the Setup menu, from which camera setup, calibration, validation, drift correction, and configuration may be performed. Pressing the 'Esc' key on the tracker keyboard will exit the Setup menu and return from this function. Calling exitCalibration() from an event handler will cause any call to doTrackerSetup() in progress to return immediately.

Parameters
None.

Return Value
None

This is equivalent to the C API **int do_tracker_setup(void);**

3.3.12 drawCalTarget

drawCaltarget()

Allow the normal calibration target drawing to proceed at different locations.

Parameters
Position: A tuple in the format of (x, y), passing along the position of drift correction target. X and y are in screen pixels.

Return Value
None

This is equivalent to the C API **INT16 CALLTYPE set_draw_cal_target_hook(INT16 (CALLBACK * erase_cal_target_hook)(HDC hdc), INT16 options);**

3.3.13 echo_key

echo_key()

Checks for Windows keystroke events and dispatches messages; similar to getkey(), but also sends keystroke to tracker.

Parameters
None

Return Value
0 if no key pressed, else key code TERMINATE_KEY if CTRL-C held down or program has been terminated.

Remarks
Warning: Under Windows XP, this call will not work in realtime mode at all, and will take several seconds to respond if graphics are being drawn continuously. This function works well in realtime mode under Windows 2000.

This function is equivalent to the C API **unsigned echo_key(void);**

3.3.14 escapePressed

escapePressed()

This function tests if the ESC key is held down, and is usually used to break out of nested loops.

Parameters
None.

Return Value
1 if ESC key held down 0 if not

Remarks
Under Windows XP, this call will not work in realtime mode at all, and will take several seconds to respond if graphics are being drawn continuously. This function works well in realtime mode under Windows 2000.

This function is equivalent to the C API **INT16 escape_pressed(void);**

3.3.15 exitCalibration

exitCalibration()

This function should be called from a message or event handler if an ongoing call to doDriftCorrect() or doTrackerSetup() should return immediately.

Parameters
None.

Return Value
None.

This function is equivalent to the C API **void exit_calibration(void);**

3.3.16 eyeAvailable

eyeAvailable()

After calling the waitForBlockStart() method, or after at least one sample or eye event has been read, this function can be used to check which eyes data is available for.

Parameters
None.

Return Value
LEFT_EYE (0) if left eye data
RIGHT_EYE (1) if right eye data
BINOCULAR (2) if both left and right eye data
-1 if no eye data is available

This is equivalent to the C API **INT16 eyelink_eye_available(void);**

3.3.17 flushKeybuttons

flushKeyButtons(*enable_buttons*)

Causes the EyeLink tracker and the EyeLink library to flush any stored button or key events. This should be used before a trial to get rid of old button responses. The <enable_buttons> argument controls whether the EyeLink library will store button press and release events. It always stores tracker key events. Even if disabled, the last button pressed and button flag bits are updated.

Parameters
Sets to 0 to monitor last button press only, 1 to queue button events.

Return Value
Always 0

This is equivalent to the C API **INT16 eyelink_flush_keybuttons(INT16 enable_buttons);**

3.3.18 getButtonStates

getButtonStates()

Returns a flag word with bits set to indicate which tracker buttons are currently pressed. This is button 1 for the LSB, up to button 16 for the MSB. Note: Buttons above 8 are not realized on the EyeLink tracker.

Parameters
None

Return Value
Flag bits for buttons currently pressed.

This function is equivalent to the C API **UINT16 eyelink_button_states(void);**

3.3.19 getCalibrationMessage

getCalibrationMessage()

Returns text associated with result of last calibration, validation, or drift correction. This usually specifies errors or other statistics.

Parameters
None.

Return Value
Message string associated with result of last calibration, validation, or drift correction.

This function is equivalent to the C API **INT16 eyelink_cal_message(char *msg);**

3.3.20 getCalibrationResult

getCalibrationResult()

Checks for a numeric result code returned by calibration, validation, or drift correction.

Parameters
None

Return Value
NO_REPLY if drift correction not completed yet
OK_RESULT (0) if success
ABORT_REPLY (27) if 'Esc' key aborted operation
-1 if operation failed
1 if poor calibration or excessive validation error.

This function is equivalent to the C API **INT16 eyelink_cal_result(void);**

3.3.21 getCurrentMode

getCurrentMode()

This function tests the current tracker mode, and returns a set of flags based of what the mode is doing. The most useful flag using the EyeLink experiment toolkit is IN_USER_MENU to test if the EyeLink Abort menu has been activated.

Parameters
None.

Return Value
Set of bit flags that mark mode function:
IN_DISCONNECT_MODE if disconnected

IN_IDLE_MODE if off-line (Idle mode)
IN_SETUP_MODE if in Setup-menu related mode
IN_RECORD_MODE if tracking is in progress
IN_PLAYBACK_MODE if currently playing back data
IN_TARGET_MODE if in mode that requires a fixation target
IN_DRIFTCORR_MODE if in drift-correction
IN_IMAGE_MODE if displaying grayscale camera image
IN_USER_MENU if displaying Abort or user-defined menu

This is equivalent to the C API **INT16 eyelink_current_mode(void);**

3.3.22 getDataCount

getDataCount(*samples, events*)

Counts total items in queue: samples, events, or both.

Parameters
<samples>: if non-zero count the samples.
<events>: if non-zero count the events..

Return Value
Total number of samples and events is in the queue.

This function is equivalent to the C API **INT16 eyelink_data_count(INT16 samples, INT16 events);**

3.3.23 getEventDataFlags

getEventDataFlags()

Returns the event data content flags.

Parameters
None.

Return Value
Possible return values are a set of the following bit flags:

Constant Name	Value	Description
EVENT_VELOCITY	0x8000	Has velocity data
EVENT_PUPILSIZE	0x4000	Has pupil size data
EVENT_GAZERES	0x2000	Has gaze resolution
EVENT_STATUS	0x1000	Has status flags
EVENT_GAZEXY	0x0400	Has gaze x, y position
EVENT_HREFXY	0x0200	Has head-ref x, y position
EVENT_PUPILXY	0x0100	Has pupil x, y position
FIX_AVG_ONLY	0x0008	Only average data to fixation events
START_TIME_ONLY	0x0004	Only start-time in start events
PARSEDBY_GAZE	0x00C0	Events were generated by GAZE data
PARSEDBY_HREF	0x0080	Events were generated by HREF data
PARSEDBY_PUPIL	0x0040	Events were generated by PUPIL data

This is equivalent to the C API **UINT16 eyelink_event_data_flags(void);**

3.3.24 getEventTypeFlags

getEventTypeFlags()

After at least one button or eye event has been read, can be used to check what type of events will be available.

Parameters
None.

Return Value
Possible return values are a set of the following bit flags:

Constant Name	Value	Description
LEFTEYE_EVENTS	0x8000	Has left eye events
RIGHTEYE_EVENTS	0x4000	Has right eye events
BLINK_EVENTS	0x2000	Has blink events
FIXATION_EVENTS	0x1000	Has fixation events
FIXUPDATE_EVENTS	0x0800	Has fixation updates
SACCADE_EVENTS	0x0400	Has saccade events
MESSAGE_EVENTS	0x0200	Has message events
BUTTON_EVENTS	0x0040	Has button events
INPUT_EVENTS	0x0020	Has input port events

This is equivalent to the C API **UINT16 eyelink_event_type_flags(void);**

3.3.25 getFloatData

getFloatData()

Reads data of a specific type returned by getNextData. If this function called multiple times without calling getNextData(), the same data is returned.

Parameters
None

Return Value
None if no data available. Otherwise, a valid data is returned. The returned data type can be:
1. Sample
2. StartBlinkEvent
3. EndBlinkEvent
4. StartSacadeEvent
5. EndSacadeEvent
6. StartFixationEvent
7. EndFixationEvent
8. FixUpdateEvent
9. IOEvent
10. MessageEvent

This function is equivalent to the C API **INT16 eyelink_get_next_data(void *buf);**

3.3.26 getkey

getKey()

Returns the key pressed.

Parameters
None

Return Value
0 if no key pressed, else key code. TERMINATE_KEY if CTRL-C held down or program has been terminated.

Remarks
Warning: This function processes and dispatches any waiting messages. This will allow Windows to perform disk access and negates the purpose of realtime mode. Usually these delays will be only a few milliseconds, but delays over 20 milliseconds have been observed. You may wish to call escapePressed() or breakPressed() in recording loops instead of getkey() if timing is critical, for example in a gaze-contingent display. Under Windows XP, these calls will not work in realtime mode at all (although these do work under Windows 2000). Under Windows 95/98/Me, realtime performance is impossible even with this strategy.
Some useful keys are:
CURS_UP
CURS_DOWN

CURS_LEFT
CURS_RIGHT
ESC_KEY
ENTER_KEY
TERMINATE_KEY
JUNK_KEY

This function is equivalent to the C API **unsigned getkey(void);**

3.3.27 getLastButtonPress

getLastButtonPress()

Reads the number of the last button detected by the EyeLink tracker. This is 0 if no buttons were pressed since the last call, or since the buttons were flushed. If a pointer to a variable is supplied the eye-tracker timestamp of the button may be read. This could be used to see if a new button has been pressed since the last read. If multiple buttons were pressed since the last call, only the last button is reported.

Parameters
None.

Return Value
Two-item tuple, recording the button last pressed (0 if no button pressed since last read) and the time of the button press.

This function is equivalent to the C API **UINT16 eyelink_last_button_press(UINT32 *time);**

3.3.28 getLastData

getLastData()

Gets an integer (unconverted) copy of the last/newest link data (sample or event) seen by **getNextData()**.

Parameters
None
Return Value
Object of type Sample or Event.

This function is equivalent to the C API **INT16 eyelink_get_last_data(void *buf);**

3.3.29 getLastMessage

getLastMessage()

Returns text associated with last command response: may have error message.

Parameters:
None.

Return Value:
Text associated with last command response or None.

This is equivalent to the C API **INT16 eyelink_last_message(char *buf);**

3.3.30 getModeData

getModeData()

After calling waitForBlockStart(), or after at least one sample or eye event has been read, returns EyeLink II extended mode data.

Parameters
None.

Return Value
A five-item tuple holding (in the following order):
- eye information (LEFT_EYE if left eye data, RIGHT_EYE if right eye data, BINOCULAR if both left and right eye data, -1 if no eye data is available),
- sampling rate (samples per second),
- CR mode flag (0 if pupil-only mode, else pupil-CR mode),
- filter level applied to file samples (0=off, 1=std, 2=extra),
- filter level applied to link and analog output samples (0=off, 1=std, 2=extra).

This function is equivalent to the C API **INT16 eyelink2_mode_data(INT16 *sample_rate, INT16 *crmode, INT16 *file_filter, INT16 *link_filter);**

3.3.31 getNewestSample

getNewestSample()

Check if a new sample has arrived from the link. This is the latest sample, not the oldest sample that is read by **getNextData()**, and is intended to drive gaze cursors and gaze-contingent displays.
Parameters
None

Return Value
None if there is no sample, instance of Sample type otherwise.

This function is equivalent to the C API **INT16 CALLTYPE eyelink_newest_sample(void FARTYPE *buf);**

3.3.32 getNextData

getNextData()

Fetches next data item from link buffer.

Parameters
None
Return Value
0 if no data, SAMPLE_TYPE (200) if sample, else event type. Possible return values are,

Constant Name	Value	Description
STARTBLINK	3	Pupil disappeared, time only
ENDBLINK	4	Pupil reappeared (duration data)
STARTSACC	5	Start of saccade (with time only)
ENDSACC	6	End of saccade (with summary data)
STARTFIX	7	Start of fixation (with time only)
ENDFIX	8	End of fixation (with summary data)
FIXUPDATE	9	Update within fixation, summary data for interval
MESSAGEEVENT	24	User-definable text (IMESSAGE structure)
BUTTONEVENT	25	Button state change (IOEVENT structure)
INPUTEVENT	28	Change of input port (IOEVENT structure)
SAMPLE_TYPE	200	Event flags gap in data stream

This function is equivalent to the C API **INT16 eyelink_get_next_data(void *buf);**

3.3.33 getNode

getNode(*response*)

Reads the responses returned by other trackers or remotes in response to pollTrackers() or pollRemotes(). It can also read the tracker broadcast address and remote broadcast addresses.

Parameters

<resp>
Nmber of responses to read: 0 gets our data, 1 get first response, 2 gets the second response, etc. -1 to read the tracker broadcast address. -2 to read remote broadcast addresses.

Return Value
If successful, an instance of EyelinkMessage class returned.

This function is equivalent to the C API **INT16 eyelink_get_node(INT16 resp, void *data);**

3.3.34 getPositionScalar

getPositionScalar()

Returns the divisor used to convert integer eye data to floating point data.

Parameters
None

Return Value
Integer for the divisor (usually 10)

This function is equivalent to the C API **INT16 eyelink_position_prescaler(void);**

3.3.35 getRecordingStatus

getRecordingStatus()

Checks if we are in Abort menu after recording stopped and returns trial exit code. Call this function on leaving a trial. It checks if the EyeLink tracker is displaying the Abort menu, and handles it if required. The return value from this function should be returned as the trial result code.

Parameters
None.

Return Value
TRIAL_OK if no error
REPEAT_TRIAL, SKIP_TRIAL, ABORT_EXPT if Abort menu activated.

This function is equivalent to the C API **INT16 check_record_exit(void);**

3.3.36 getSample

getSample()

Gets an integer (unconverted) sample from end of queue, discards any events encountered.

Parameters
None
Return Value
Object of type Sample

This is equivalent to the C API **INT16 eyelink_get_sample(void *sample);**

3.3.37 getSampleDataFlags

getSampleDataFlag()

After calling waitForBlockStart(), or after at least one sample or eye event has been read, returns sample data content flag (0 if not in sample block).

Parameters
None.

Return Value

Possible return values are a set of the following bit flags:

Constant Name	Value	Description
SAMPLE_LEFT	0x8000	Data for left eye
SAMPLE_RIGHT	0x4000	Data for right eye
SAMPLE_TIMESTAMP	0x2000	always for link, used to compress files
SAMPLE_PUPILXY	0x1000	pupil x,y pair
SAMPLE_HREFXY	0x0800	head-referenced x,y pair
SAMPLE_GAZEXY	0x0400	gaze x,y pair
SAMPLE_GAZERES	0x0200	gaze res (x,y pixels per degree) pair
SAMPLE_PUPILSIZE	0x0100	pupil size
SAMPLE_STATUS	0x0080	error flags
SAMPLE_INPUTS	0x0040	input data port
SAMPLE_BUTTONS	0x0020	button state: LSBy state, MSBy changes
SAMPLE_HEADPOS	0x0010	head-position: byte tells # words
SAMPLE_TAGGED	0x0008	reserved variable-length tagged
SAMPLE_UTAGGED	0x0004	user-defineabe variable-length tagged

This function is equivalent to the C API **INT16 eyelink_sample_data_flags(void);**

3.3.38 getTargetPositionAndState

getTargetPositionAndState()

Returns the current target position and state.

Parameters
None.

Return Value
A three-item tuple holding (in the following order):
- the target visibility (0 if visible),
- x position of the target,
- and y position of the target.

This function is equivalent to the C API **INT16 eyelink_target_check(INT16 *x, INT16 *y);**

3.3.39 getTrackerInfo

getTrackerInfo()

Returns the current tracker information.

Parameters
None.

Return Value
An instance of the ILinkData class (see section 3.12).

3.3.40 getTrackerMode

getTrackerMode()

Returns raw EyeLink mode numbers.

Parameters
None.

Return Value
Raw EyeLink mode, -1 if link disconnected.

Constant Name	Value
EL_IDLE_MODE	1
EL_IMAGE_MODE	2

EL_SETUP_MENU_MODE	3
EL_USER_MENU_1	5
EL_USER_MENU_2	6
EL_USER_MENU_3	7
EL_OPTIONS_MENU_MODE	8
EL_OUTPUT_MENU_MODE	9
EL_DEMO_MENU_MODE	10
EL_CALIBRATE_MODE	11
EL_VALIDATE_MODE	12
EL_DRIFT_CORR_MODE	13
EL_RECORD_MODE	14

USER_MENU_NUMBER(mode) ((mode)-4)

This function is equivalent to the C API **INT16 eyelink_tracker_mode(void);**

3.3.41 getTrackerVersion

getTrackerVersion()

After connection, determines if the connected tracker is an EyeLink I or II. Use getTrackerVersionString to get the string value.

Parameters:
None

Return Values:
The returned value is a number (0 if not connected, 1 for EyeLink I, 2 for EyeLink II).

This is equivalent to the C API **INT16 eyelink_get_tracker_version(char *c);**

3.3.42 getTrackerVersionString

getTrackerVersionString()

After connection, determines if the connected tracker is an EyeLink I or II (use getTrackerVersion) to get number value.

Parameters:
None

Return Value:
A string indicating EyeLink tracker version.

This is equivalent to the C API **INT16 eyelink_get_tracker_version(char *c);**

3.3.43 inSetup

inSetup()

Checks if tracker is still in a Setup menu activity (includes camera image view, calibration, and validation). Used to terminate the subject setup loop.

Parameters
None

Return Value
0 if no longer in setup mode.

This function is equivalent to the C API **INT16 eyelink_in_setup(void);**

3.3.44 isConnected

isConnected()

Checks whether the connection to the tracker is alive.

Parameters:
None

Return Values:
Returns 0 if link closed, -1 if simulating connection, 1 for normal connection, 2 for broadcast connection.

This is equivalent to the C API **INT16 eyelink_is_connected(void);**

3.3.45 isInDataBlock

isInDataBlock(*samples, events*)

Checks to see if framing events read from queue indicate that the data is in a block containing samples, events, or both.

Parameters
<samples>: if non-zero, check if in a block with samples.
<events>: if non-zero, check if in a block with events.

Return Value
0 if no data of either masked type is being sent.

Remarks
The first item in queue may not be a block start even, so this should be used in a loop while discarding items using eyelink_get_next_data(NULL). NOTE: this function did not work reliably in versions of the DLL before v2.0 (did not detect end of blocks).

This function is equivalent to the C API **INT16 eyelink_in_data_block(INT16 samples, INT16 events);**

3.3.46 isRecording

isRecording()

Check if we are recording: if not, report an error. Call this function while recording. It will return true if recording is still in progress, otherwise it will throw an exception. It will also handle the EyeLink Abort menu. Any errors returned by this function should be returned by the trial function. On error, this will disable realtime mode and restore the heuristic.

Parameters
None.

Return Value
TRIAL_OK (0) if no error
REPEAT_TRIAL, SKIP_TRIAL, ABORT_EXPT, TRIAL_ERROR if recording aborted.

This function is equivalent to the C API **int check_recording(void);**

3.3.47 nodeReceive

nodeReceive()

Checks for and gets the last packet received, stores the data and the node address sent from. Note: Data can only be read once, and is overwritten if a new packet arrives before the last packet has been read.

Parameters
None

Return Value
An instance of EyelinkMessage class is returned, if successful.

This function is equivalent to the C API **INT16 eyelink_node_receive(ELINKADDR node, void *data);**

3.3.48 nodeRequestTime

nodeRequestTime(*address*)

Sends a request the connected eye tracker to return its current time. Note: The time reply can be read with getTrackerTime().

Parameters
<address>: text IP address (for example, "100.1.1.1") for a specific tracker.

Return Value
0 if no error, else link error code

This function is equivalent to the C API **UINT32 eyelink_node_request_time(ELINKADDR node);**

3.3.49 nodeSend

nodeSend(address, data, length)

Sends a given data to the given node.

Parameters
<addr>: the address of the node;
<data>: Pointer to buffer containing data to send*;*
<length>: Number of bytes of data

Return Value
0 if successful, otherwise link error.

This function is equivalent to the C API **INT16 eyelink_node_send(ELINKADDR node, void *data, UINT16 dsize);**

3.3.50 nodeSendMessage

nodeSendMessage(*address, message*)

Sends a text message the connected eye tracker. The text will be added to the EDF file. Note: If the link is initialized but not connected to a tracker, the message will be sent to the tracker set by *setAddress()* of the pylink module.

Parameters
<address>: Address of a specific tracker.
<message>: Text to send to the tracker.

Return Value
0 if no error, else link error code

This function is equivalent to the C API **INT16 eyelink_node_send_message(ELINKADDR node, char *msg);**

3.3.51 open

open(*eyelink_address = "100.1.1.1", busytest=0*)

Opens connection to single tracker. If no parameters are given, it tries to open connection to the default host (100.1.1.1).

Parameters
<eyelink_address>: text IP address of the host PC (the default value is, "100.1.1.1");
<busytest>: if non-zero the call to eyelink_open_node will not disconnect an existing connection.

Return Value
Returns *None.* Throws Runtime error exception if it cannot open the connection.

This is equivalent to the C API **INT16 eyelink_open_node(ELINKADDR node, INT16 busytest);**

3.3.52 openDataFile

openDataFile(*name*)

Opens a new EDF file on the EyeLink tracker computer's hard disk. By calling this function will close any currently opened file. This may take several seconds to complete. The file name should be formatted for MS-DOS, usually 8 or less characters with only 0-9, A-Z, and '_' allowed.

Parameters
<name>: Name of eye tracker file, 8 characters or less.

Return Value
0 if file was opened successfully else error code.

This function is equivalent to the C API **int open_data_file(char *name);**

3.3.53 openNode

openNode(*eyelink_address , busytest*)

Allows the computer to connect to tracker, where the tracker is on the same network.

Parameters:
<eyelink_address>: text IP address of the host PC (the default value is, "100.1.1.1");
<busytest>: if non-zero the call to openNode will not disconnect an existing connection.

Return Values:
None. Throws Runtime Exception if it connects to the remote host.

This is equivalent to the C API **INT16 eyelink_open_node(ELINKADDR node, INT16 busytest);** with node parameter converted from text to ELINKADDR.

3.3.54 pollRemotes

pollRemotes()

Asks all non-tracker computers (with EyeLink software running) on the network to send their names and node address.

Parameters
None

Return Value
0 if successful, otherwise link error.

This function is equivalent to the C API **INT16 eyelink_poll_ remotes(void);**

3.3.55 pollResponses

pollResponses()

Returns the count of node addresses received so far following the call of pollRemotes() or pollTrackers(). Note: You should allow about 100 milliseconds for all nodes to respond. Up to 4 node responses are saved.

Parameters
None

Return Value

Number of nodes responded. 0 if no responses.

This function is equivalent to the C API **INT16 eyelink_poll_responses(void);**

3.3.56 pollTrackers

pollTrackers()

Asks all trackers (with EyeLink software running) on the network to send their names and node address.

Parameters
None

Return Value
0 if successful, otherwise link error.

This function is equivalent to the C API **INT16 eyelink_poll_trackers(void);**

3.3.57 pumpMessages

pumpMessages()

Forces the graphical environment to process any pending key or mouse events.

Parameter:

None

Return Value:

None

This function is equivalent to the C API **INT16 message_pump(HWND dialog_hook).**

3.3.58 quietMode

quietMode(*mode*)

Controls the level of control an application has over the tracker.

Parameters
<mode>: 0 to allow all communication; 1 to block commands (allows only key presses, messages, and time or variable read requests); 2 to disable all commands, requests and messages; -1 to just return current setting

Return Value
Returns the previous mode settings.

This function is equivalent to the C API **INT16 eyelink_quiet_mode(INT16 mode);**

3.3.59 readKeyButton

readKeyButton()

Reads any queued key or button events from tracker.

Parameters
None.

Return Value
A five-item tuple, recording (in the following order):
- Key character if key press/release/repeat, KB_BUTTON (0xFF00) if button press or release,
- Button number or key modifier (Shift, Alt and Ctrl key states),
- Key or button change (KB_PRESS, KB_RELEASE, or KB_REPEAT),
- Key scan code,

- Tracker time of the key or button change.

This function is equivalent to the C API **UINT16 eyelink_read_keybutton(INT16 *mods,INT16 *state, UINT16 *kcode, UINT32 *time);**

3.3.60 readKeyQueue

readKeyQueue()

Read keys from the key queue. It is similar to getkey(), but does not process Windows messages. This can be used to build key-message handlers in languages other than C.

Parameters
None.

Return Value
0 if no key pressed
JUNK_KEY (1) if untranslatable key
TERMINATE_KEY (0x7FFF) if CTRL-C is pressed, terminal_break() was called, or the program has been terminated with ALT-F4.
or code of key if any key pressed.

This function is equivalent to the C API **UINT16 read_getkey_queue(void);**

3.3.61 readReply

readReply*()*

Returns text with reply to last read request.

Parameters:
None.

Return Value:
String to contain text or None.

This is equivalent to the C API **INT16 eyelink_read_reply(char *buf);**

3.3.62 readRequest

readRequest(*text*)

Sends a text variable name whose value is read and returned by the tracker as a text string.

Parameters:
<text>: String with message to send.

Return Value:
0 if success, otherwise link error code.

Remarks
If the link is initialized but not connected to a tracker, the message will be sent to the tracker set by setAddress(). However, these requests will be ignored by tracker versions older than EyeLink I v2.1 and EyeLink II v1.1.

This is equivalent to the C API **INT16 eyelink_read_request(char *text);**

3.3.63 receiveDataFile

receiveDataFile(*src, dest*)

This receives a data file from the EyeLink tracker PC. Source file name and destination file name should be given.

Parameters
<Src>: Name of eye tracker file (including extension).
<Dest>: Name of local file to write to (including extension).

Return Value
Size of file if successful,
Otherwise Runtime Exception is raised.

This function is equivalent to the C API **int receive_data_file(char *src, char *dest, int is_path);**

3.3.64 reset

reset()

Resets the link data system if it is listening in on a broadcast data session. This does not shut down the link, which remains available for other operations.

Parameters
None.

Return Value
0 if successful, otherwise linkerror.

This function is equivalent to the C API **INT16 eyelink_close(int send_msg);** with send_msg parameter 0.

3.3.65 resetData

resetData()

Prepares link buffers to receive new data.

Parameters
<clear>: If nonzero, removes old data from buffer.

Return Value
Always 0.

This function is equivalent to the C API **INT16 eyelink_reset_data(INT16 clear);**

3.3.66 sendCommand

sendCommand(*command_text*)

Sends the given command to connected eyelink tracker and returns the command result.

Parameters:
<command_text>: text command to be sent. It does not support printf() kind of formatting.

Return Values:
Command result. If there is any problem sending the command, a runtime exception is raised.

This is equivalent to the C API **int eyecmd_printf(char *fmt, ...);** without any formatting.

3.3.67 sendKeybutton

sendKeybutton(*code, mods, state*)

Sends a key or button event to tracker. Only key events are handled for remote control.

Parameters
<code>: key character, or KB_BUTTON (0xFF00) if sending button event
<mods>: button number, or key modifier (Shift, Alt and Ctrl key states).
<state>: key or button change (KB_PRESS or KB_RELEASE)

Return Value
0 if ok, else send link error.

This function is equivalent to the C API **INT16 eyelink_send_keybutton(UINT16 code, UINT16 mods, INT16 state);**

3.3.68 sendMessage

sendMessage(*message_text*)

Sends the given message to the connected eyelink tracker. The message will be written to the eyelink tracker.

Parameters:
<message_text>: text message to be sent. It does not support printf() kind of formatting.

Return Values:
If there is any problem sending the message, a runtime exception is raised.

This is equivalent to the C API **int eyecmd_printf(char *fmt, ...);**

3.3.69 setAddress

setAddress(*text_IP_address*)

Sets the IP address used for connection to the EyeLink tracker. This is set to "100.1.1.1" in the DLL, but may need to be changed for some network configurations. This must be set before attempting to open a connection to the tracker.

A "broadcast" address ("255.255.255.255") may be used if the tracker address is not known—this will work only if a single Ethernet card is installed, or if DLL version 2.1 or higher, and the latest tracker software versions (EyeLink I v2.1 or higher, and EyeLink II v1.1 or higher) are installed.

Parameters:
<text_IP_address>: Pointer to a string containing a "dotted" 4-digit IP address;

Return Value:
0 if success, -1 if could not parse address string

This is equivalent to the C API **INT16 set_eyelink_address(char *addr);**

3.3.70 setName

SetName(*name*)

Sets the node name of this computer (up to 35 characters).

Parameters
<name>: String to become new name.

Return Value
None.

This function is equivalent to the C API **INT16 eyelink_set_name(char *name);**

3.3.71 setOfflineMode

setOfflineMode()

Places EyeLink tracker in off-line (idle) mode. Wait till the tracker has finished the mode transition.

Parameters:
None

Return Values:
None

This is equivalent to the C API **INT16 set_offline_mode(void);**

3.3.72 startData

startData(*flags, lock*)

Switches tracker to Record mode, enables data types for recording to EDF file or sending to link. These types are set with a bit wise OR of these flags:

Constant Name	Value	Description
RECORD_FILE_SAMPLES	1	Enables sample recording to EDF file
RECORD_FILE_EVENTS	2	Enables event recording to EDF file
RECORD_LINK_SAMPLES	4	Enables sending samples to the link
RECORD_LINK_EVENTS	8	Enables sending events to the link

Parameters
<flags>: Bitwise OR of flags to control what data is recorded. If 0, recording will be stopped.
<lock>: If nonzero, prevents 'Esc' key from ending recording.

Return Value
0 if command sent ok, else link error.

Remarks
If <lock> is nonzero, the recording may only be terminated through stopRecording() or stopData() method of the EyeLinkListener class, or by the Abort menu ('Ctrl''Alt''A' keys on the eye tracker). If zero, the tracker 'Esc' key may be used to halt recording.

This function is equivalent to the C API **INT16 eyelink_data_start(UINT16 flags, INT16 lock);**

3.3.73 startDriftCorrect

startDriftCorrect(*x, y*)

Sets the position of the drift correction target, and switches the tracker to drift-correction mode. Should be followed by a call to **waitForModeReady()** method.

Parameters
<x>: x position of the target.
<y>: y position of the target.

Return Value
0 if command sent ok, else link error.

This function is equivalent to the C API **INT16 eyelink_driftcorr_start(INT16 x, INT16 y);**

3.3.74 startPlayBack

startPlayBack()

Flushes data from queue and starts data playback. An EDF file must be open and have at least one recorded trial. Use waitForData() method to wait for data: this will time out if the playback failed. Playback begins from start of file or from just after the end of the next-but-last recording block. Link data is determined by file contents, not by link sample and event settings.

Parameters
None.

Return Value
0 if command sent ok, else link error.

This function is equivalent to the C API **INT16 eyelink_playback_start(void);**

3.3.75 startRecording

startRecording(File_samples, File_events, Link_samples, Link_events)

Starts the EyeLink tracker recording, sets up link for data reception if enabled.

Parameters:
<File_samples>: If 1, writes samples to EDF file. If 0, disables sample recording.
<File_events>: If 1, writes events to EDF file. If 0, disables event recording.
<Link_samples>: If 1, sends samples through link. If 0, disables link sample access.
<Link_events>: If 1, sends events through link. If 0, disables link event access.

Return Values:
0 if successful, else trial return code

This is equivalent to the C API **INT16 start_recording(INT16 file_samples, INT16 file_events, INT16 link_samples, INT16 link_events);**

Note:
Recording may take 10 to 30 milliseconds to begin from this command. The function also waits until at least one of all requested link data types have been received. If the return value is not zero, return the result as the trial result code.

3.3.76 startSetup

startSetup()

Switches the EyeLink tracker to the setup menu, for calibration, validation, and camera setup. Should be followed by a call to waitForModeReady().

Parameters:
None

Return Value:
0 if command send OK.

This is equivalent to the C API **INT16 eyelink_start_setup(void);**

3.3.77 stopData

StopData()

Places tracker in idle (off-line) mode, does not flush data from queue.

Parameters
None.

Return Value
0 if command sent ok, else link error.

Remark:
Should be followed by a call to **waitForModeReady()** method.

This function is equivalent to the C API **INT16 eyelink_data_stop(void);**

3.3.78 stopPlayBack

StopPlayBack()

Stops playback if in progress. Flushes any data in queue.

Parameters
None.

Return Value
None.

This function is equivalent to the C API **INT16 eyelink_playback_stop(void);**

3.3.79 stopRecording

stopRecording()

Stops recording, resets EyeLink data mode. Call 50 to 100 msec after an event occurs that ends the trial. This function waits for mode switch before returning.

Parameters
None

Return Value
None

This is equivalent to the C API **void stop_recording(void);**

3.3.80 terminalBreak

terminalBreak(assert)

This function can be called in an event handler to signal that the program is terminating. Calling this function with an argument of 1 will cause breakPressed() to return 1, and getkey() to return TERMINATE_KEY. These functions can be re-enabled by calling terminalBreak() with an argument of 0.

Parameters
<Assert>: 1 to signal a program break, 0 to reset break.

Return Value
None.

This function is equivalent to the C API **void terminal_break(INT16 assert);**

3.3.81 trackerTime

trackerTime()

Returns the current tracker time (in milliseconds) since the tracker application started.

Parameters
None.

Return Value
An integer data for the current tracker time (in milliseconds) since tracker initialization.

This is equivalent to the C API **UINT32 eyelink_tracker_time();**

3.3.82 trackerTimeOffset

trackerTimeOffset()

Returns the time difference between the tracker time and display pc time.

Parameters
None.

Return Value

An integer data for the time difference (in milliseconds) between the tracker time and display pc time.

This is equivalent to the C API **UINT32 eyelink_time_offset();**

3.3.83 trackerTimeUsec

trackerTimeUsec()

Returns the current tracker time (in microseconds) since the tracker application started.

Parameters
None.

Return Value
A double precision data for the current tracker time (in microseconds) since tracker initialization.

This is equivalent to the C API **UINT32 eyelink_tracker_time();**

3.3.84 trackerTimeUsecOffset

trackerTimeUsecOffset()

Returns the time difference between the tracker time and display pc time.

Parameters
None.

Return Value
A double precision data for the time difference (in microseconds) between the tracker time and display pc time.

This is equivalent to the C API **double eyelink_time_usec_offset();**

3.3.85 userMenuSelection

userMenuSelection()

Checks for a user-menu selection, clears response for next call.

Parameters
None

Return Value
0 if no selection made since last call, else code of selection

This function is equivalent to the C API **INT16 eyelink_user_menu_selection(void);**

3.3.86 waitForBlockStart

waitForBlockStart(*tiemout, samples, events*)

Reads and discards events in data queue until in a recording block. Waits for up to <timeout> milliseconds for a block containing samples, events, or both to be opened. Items in the queue are discarded until the block start events are found and processed. This function will fail if both samples and events are selected but only one of link samples and events were enabled by startRecording().

Parameters
<timeout>: time in milliseconds to wait.
<samples>: if non-zero, check if in a block with samples.
<events>: if non-zero, check if in a block with events..

Return Value
0 if time expired without any data of masked types available.

Remarks
This function did not work in versions previous to 2.0.

This function is equivalent to the C API **INT16 eyelink_wait_for_block_start(UINT32 maxwait,INT16 samples, INT16 events);**

3.3.87 waitForData

waitForData(*maxwait, samples, events*)

Waits for data to be received from the eye tracker. Can wait for an event, a sample, or either. Typically used after record start to check if data is being sent.

Parameters
<maxwait>: time in milliseconds to wait for data.
<samples>: if 1, return when first sample available.
<events>: if 1, return when first event available.

Return Value
1 if data is available; 0 if timed out.

This function is equivalent to the C API **INT16 eyelink_wait_for_data (UINT32 maxwait, INT16 samples, INT16 events);**

3.3.88 waitForModeReady

waitForModeReady(*maxwait*)

After a mode-change command is given to the EyeLink tracker, an additional 5 to 30 milliseconds may be needed to complete mode setup. Call this function after mode change functions.

Parameters
<maxwait>: Maximum milliseconds to wait for the mode to change.

Return Value
0 if mode switching is done, else still waiting.

Remarks
If it does not return 0, assume a tracker error has occurred.

This function is equivalent to the C API **INT16 eyelink_wait_for_mode_ready(UINT32 maxwait);**

3.4 *Eyelink Class*

The Eyelink class is an extension of the EyeLinkListener class with additional utility functions. Most of these functions are used to perform tracker setups (For current information on the EyeLink tracker configuration, examine the *.INI files in the EYELINK\EXE\ directory of the eye tracker computer). An instance of the Eyelink class can be created by using the class constructor function. For example,

```
try:
    EYELINK = EyeLink()
except:
    EYELINK = None
```

An instance of EyeLink class can directly use all of the EyeLinkListener methods listed in the previous sections. In addition, it has its own methods as listed in the following sections. All of the methods should be called in the format of: EYELINK.functionName(parameters), where EYELINK is an instance of EyeLink class.

3.4.1 *Calibration Setup*

3.4.1.1 doTrackerSetup

doTrackerSetup(width = None, height = None)

Switches the EyeLink tracker to the Setup menu, from which camera setup, calibration, validation, drift correction, and configuration may be performed. Pressing the 'Esc' key on the tracker keyboard will exit the Setup menu and return from this function. Calling exitCalibration() from an event handler will cause any call to do_tracker_setup() in progress to return immediately.

Parameters
<width>: width of the screen
<height>: height of he screen

Return Value
None**;**

3.4.1.2 setAcceptTargetFixationButton

setAcceptTargetFixationButton(button)

This programs a specific button for use in drift correction.

Parameters
<button>: ID of the button that is used to accept target fixation;

Return Value
None**;**

Remarks:
This function is equivalent to
EYELINK.sendCommand("button_function %d 'accept_target_fixation'"%button);

3.4.1.3 setCalibrationType

setCalibrationType(caltype)

This command sets the calibration type, and recomputed the calibration targets after a display resolution change.

Parameters
<type>: one of these calibration type codes:
H3: horizontal 3-point calibration
HV3: 3-point calibration, poor linearization
HV5: 5-point calibration, poor at corners
HV9: 9-point grid calibration, best overall

Return Value
None**;**

Remark:
This function is equivalent to
EYELINK.sendCommand("calibration_type=%s"%caltype);

3.4.1.4 setXGazeConstraint

setXGazeConstraint(x_position)

Locks the X part of gaze position data. Usually set to AUTO: this will use the last drift-correction target position when in H3 mode.

Parameters
<x_posittion>: x gaze coordinate, or AUTO

Return Value
None**;**

Remark:
This function is equivalent to
EYELINK.sendCommand("x_gaze_constraint=%s"%(str(value)));

3.4.1.5 setYGazeConstraint

setYGazeConstraint(y_position)

Locks the Y part of gaze position data. Usually set to AUTO: this will use the last drift-correction target position when in H3 mode.

Parameters
<y_posittion>: y gaze coordinate, or AUTO

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("y_gaze_constraint=%s"%(str(value)));

3.4.1.6 enableAutoCalibration

enableAutoCalibration()

Enables the auto calibration mechanism.

Parameters
None.

Return Value
None;

Remark:
This function is equivalent to
if(EYELINK.isConnected()):
EYELINK.sendCommand("enable_automatic_calibration=YES")

3.4.1.7 disableAutoCalibration

disableAutoCalibration()

Disables the auto calibration mechanism.

Parameters
None.

Return Value
None;

Remark:
This function is equivalent to
if(EYELINK.isConnected()):
EYELINK.sendCommand("enable_automatic_calibration=NO")

3.4.1.8 setAutoCalibrationPacing

setAutoCalibrationPacing(time)

Sets automatic calibration pacing. 1000 is a good value for most subjects, 1500 for slow subjects and when interocular data is required.

Parameters
<time>: shortest delay.

Return Value

None;

Remark:
This function is equivalent to
EYELINK.sendCommand("automatic_calibration_pacing=%d"%(time))

3.4.2 Configuring Key and Buttons

3.4.2.1 setKeyFunction

setKeyFunction(keyspec, command)

Used to configures and assigns special functions to tracker keys (see keys.ini for examples).

Parameters
<keyspec>: key name and modifiers;
<command>: command string to execute when key pressed.

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("key_function %s %s"%(str(key_specifier), str(cmd)));

3.4.3 Drawing Commands

3.4.3.1 echo

echo(text)

Prints text at current print position to tracker screen, gray on black only.

Parameters
<text>: text to print in quotes;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("echo %s"%text)

3.4.3.2 drawText

drawText(text, pos = (-1, -1))

Draws text, coordinates are gaze-position display coordinates.

Parameters
<text>: text to print in quotes;
<pos>: Center point of text; Default position is (-1, -1).

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("print_position= %d %d"%pos)
EYELINK.sendCommand("echo %s"%(text))

3.4.3.3 drawCross

drawCross(x, y, color)

Draws a small "+" to mark a target point.

Parameters
<x>, <y>: x, y coordinates for the center point of cross.
<color>: 0 to 15 (0 for black; 1 for blue; 2 for green; 3 for cyan; 4 for red; 5 for magenta; 6for brown; 7 for light gray; 8 for dark gray; 9 for light blue; 10 for light green; 11 for light cyan; 12 for light red; 13 for bright magenta; 14 for yellow; 15 for bright white);

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("clear_cross %d"%(x,y, color));

3.4.3.4 drawBox

drawBox(x, y, width, height, color)

Draws an empty box, coordinates are gaze-position display coordinates.

Parameters
<x>, <y>: x, y coordinates for the top-left corner of the rectangle.
<width>, <height>: width, height of the filled rectangle.
<color>: 0 to 15;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("draw_box %d"%(x,y,x+width,y+height,color));

3.4.3.5 drawFilledBox

drawFilledBox(x, y, width, height, color)

Draws a solid block of color, coordinates are gaze-position display coordinates.

Parameters
<x>, <y>: x, y coordinates for the top-left corner of the rectangle.
<width>, <height>: width, height of the filled rectangle.
<color>: 0 to 15;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("draw_filled_box %d"%(x,y,x+width,y+height,color));

3.4.3.6 drawLine

drawLine(firstPoint, secondPoint, color)

Draws line, coordinates are gaze-position display coordinates.

Parameters
<firstPoint>: a two-item tuple, containing the x, y coordinates of the start point.
<secondPoint>: a two-item tuple, containing the x, y coordinates of the end point.
<color>: 0 to 15;

Return Value
None;

Remark:
This function is equivalent to
EYELINK. sendCommand("draw_line %d %d %d %d %d"%
(firstPoint[0],firstPoint[1],secondPoint[0],secondPoint[1], color));

3.4.3.7 clearScreen

clearScreen(color)

Clear tracker screen for drawing background graphics or messages.

Parameters
<color>: 0 to 15;

Return Value
None;

Remark:
This function is equivalent to
EYELINK. sendCommand("clear_screen %d"%(color));

3.4.4 File/Link Data Control

3.4.4.1 setRecordingParseType

setRecordingParseType(rtype)

Sets how velocity information for saccade detection is computed.

Parameters
<rtype>: GAZE or HREF; Almost always left to GAZE;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("recording_parse_type %s"%(rtype));

3.4.4.2 setLinkEventFilter

setLinkEventFilter(list)

Sets which types of events will be sent through link. See tracker file "DATA.INI" for types.

Parameters
<list>: list of event types

LEFT, RIGHT	events for one or both eyes
FIXATION	fixation start and end events
FIXUPDATE	fixation (pursuit) state updates
SACCADE	saccade start and end
BLINK	blink start an end
MESSAGE	messages (user notes in file)
BUTTON	button 1..8 press or release
INPUT	changes in input port lines;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("link_event_filter = %s"%list);

3.4.4.3 setLinkEventData

setLinkEventData(list)

Sets data in events sent through link. See tracker file "DATA.INI" for types.

Parameters
<list>: list of data types, separated by spaces or commas.

GAZE	screen xy (gaze) position
GAZERES	units-per-degree angular resolution
HREF	HREF gaze position
AREA	pupil area or diameter
VELOCITY	velocity of eye motion (avg, peak)
STATUS	warning and error flags for event
FIXAVG	include ONLY average data in ENDFIX events
NOSTART	start events have no data, just time stamp

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("link_event_data = %s"%list);

3.4.4.4 setLinkSampleFilter

setLinkSampleFilter(list)

Sets data in samples sent through link. See tracker file "DATA.INI" for types.

Parameters
<list>: list of data types, separated by spaces or commas.

GAZE	screen xy (gaze) position
GAZERES	units-per-degree screen resolution
HREF	head-referenced gaze
PUPIL	raw eye camera pupil coordinates
AREA	pupil area
STATUS	warning and error flags
BUTTON	button state and change flags
INPUT	input port data lines

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("link_sample_data = %s"%list)

3.4.4.5 setFileEventFilter

setFileEventFilter(list)

Sets which types of events will be written to EDF file. See tracker file "DATA.INI" for types.

Parameters
<list>: list of the following event types, separated by spaces or commas

LEFT, RIGHT	events for one or both eyes
FIXATION	fixation start and end events
FIXUPDATE	fixation (pursuit) state updates
SACCADE	saccade start and end
BLINK	blink start an end
MESSAGE	messages (user notes in file)
BUTTON	button 1..8 press or release
INPUT	changes in input port lines;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("file_event_filter = %s"%list);

3.4.4.6 setFileEventData

setFileEventData(list)

Sets data in events written to EDF file. See tracker file "DATA.INI" for types.

Parameters
<list>: list of the following event data types, separated by spaces or commas

GAZE	screen xy (gaze) position
GAZERES	units-per-degree angular resolution
HREF	HREF gaze position
AREA	pupil area or diameter
VELOCITY	velocity of eye motion (avg, peak)
STATUS	warning and error flags for event
FIXAVG	include ONLY average data in ENDFIX events
NOSTART	start events have no data, just time stamp

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("file_event_data = %s"%list);

3.4.4.7 setFileSampleFilter

setFileSampleFilter(list)

Sets data in samples written to EDF file. See tracker file "DATA.INI" for types.

Parameters
<list>: list of the following data types, separated by spaces or commas

GAZE	screen x/y (gaze) position
GAZERES	units-per-degree screen resolution
HREF	head-referenced gaze
PUPIL	raw eye camera pupil coordinates
AREA	pupil area
STATUS	warning and error flags
BUTTON	button state and change flags
INPUT	input port data lines

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("file_sample_data = %s"%list)

3.4.4.8 setNoRecordEvents

setNoRecordEvents(message=False, button = False, inputevent =False)

Selects what types of events can be sent over the link while not recording (e.g between trials).
This command has no effect for EyeLink II, and messages cannot be enabled for versions of EyeLink I before v2.1.

Parameters
<message>: 1 to enable the recording of EyeLink messages
<button>: 1 to enable recording of buttons (1..8 press or release)
<inputevent>: 1 to enable recording of changes in input port lines

Return Value
None;

Remark:
This function is equivalent to

```
re = []
if(message):
        re.append("MESSAGE ")
if(button):
        re.append("BUTTON ")
if(inputevent):
        re.append("INPUT ")
EYELINK.sendCommand("link_nonrecord_events = %s"%"".join(re));
```

3.4.4.9 markPlayBackStart

markPlayBackStart()

Marks the location in the data file from which playback will begin at the next call to EYEYLINK.startPlayBack (). When this command is not used (or on older tracker versions), playback starts from the beginning of the previous recording block. This default behavior is suppressed after this command is used, until the tracker application is shut down.

Parameters
None.

Return Value
None**;**

Remark:
This function is equivalent to

```
EYELINK.sendCommand("mark_playback_start");
```

3.4.5 Tracker Configuration

3.4.5.1 setHeuristicFilterOn

setHeuristicFilterOn()

EyeLink 1 Only: Can be used to enable filtering, increases system delay by 4 msec if the filter was originally off. For Eyelink II, you should use the setHuresticFileAndLinkFilter() method instead.

Parameters
None.

Return Value
None**;**

Remark:
This function is equivalent to

```
EYELINK.sendCommand("heuristic_filter=ON");
```

3.4.5.2 setHeuristicFilterOff

setHeuristicFilterOn()

EyeLink 1 Only: Can be used to disable filtering, reduces system delay by 4 msec. NEVER TURN OFF THE FILTER WHEN ANTIREFLECTION IS TURNED ON. For Eyelink II, you should use the following setHuresticFileAndLinkFilter() method instead.

Parameters
None.

Return Value
None**;**

Remark:

This function is equivalent to
EYELINK.sendCommand("heuristic_filter = OFF");

3.4.5.3 setHeuresticLinkAndFileFilter

setHeuresticLinkAndFileFilter(linkfilter, filefilter=-1)

EyeLink II only:
Can be used to set level of filtering on the link and analog output, and on file data. An additional delay of 1 sample is added to link or analog data for each filter level. If an argument of <on> is used, link filter level is set to 1 to match EyeLink I delays. The file filter level is not changed unless two arguments are supplied. The default file filter level is 2.

Parameters
<Linkfilter>: Filter level of the link data.
0 or OFF disables link filter
1 or ON sets filter to 1 (moderate filtering, 1 sample delay)
2 applies an extra level of filtering (2 sample delay).
<filefilter>: Filter level of the data written to EDF file.
0 or OFF disables link filter
1 or ON sets filter to 1 (moderate filtering, 1 sample delay)
2 applies an extra level of filtering (2 sample delay).

Return Value
None;

Remark:
This function is equivalent to
```
if(EYELINK.getTrackerVersion() >=2):
    if(filefilter == -1):
        EYELINK.sendCommand("heuristic_filter %d"%(linkfilter))
    else:
        EYELINK.sendCommand(" %d %d"%(linkfilter, filefilter));
```

3.4.5.4 setPupilSizeDiameter

setPupilSizeDiameter(value)

Can be used to determine pupil size information to be recorded.

Parameters
<value>: YES to convert pupil area to diameter; NO to output pupil area data.

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("pupil_size_diameter = %s"%(value));

3.4.5.5 setSimulationMode

setSimulationMode(value)

Can be used to turn off head tracking if not used. Do this before calibration.

Parameters
<value>: YES to disable head tracking; NO to enable head tracking.

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("simulate_head_camera = %s"%(value));

3.4.5.6 setScreenSimulationDistance

setScreenSimulationDistance(distance)

Used to compute correct visual angles and velocities when head tracking not used.

Parameters
<distance>: simulated distance from display to subject in millimeters.

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("simulation_screen_distance = %s"%(distance));

3.4.6 Parser Configuration

3.4.6.1 setSaccadeVelocityThreshold

setSaccadeVelocityThreshold(vel)

Sets velocity threshold of saccade detector: usually 30 for cognitive research, 22 for pursuit and neurological work.

Parameters
<vel>: minimum velocity (°/sec) for saccade;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("saccade_velocity_threshold =%d"%(vel));

3.4.6.2 setAccelerationThreshold

setAccelerationThreshold (accel)

Sets acceleration threshold of saccade detector: usually 9500 for cognitive research, 5000 for pursuit and neurological work.

Parameters
<accel>: minimum acceleration (°/sec/sec) for saccades;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("saccade_acceleration_threshold =%d"%(accl));

3.4.6.3 setMotionThreshold

setMotionThreshold(deg)

Sets a spatial threshold to shorten saccades. Usually 0.15 for cognitive research, 0 for pursuit and neurological work.

Parameters
<deg>: minimum motion (degrees) out of fixation before saccade onset allowed;

Return Value
None;

Remark:

This function is equivalent to
EYELINK. sendCommand("saccade_motion_threshold =%d"%(deg));

3.4.6.4 setPursuitFixup

setPursuitFixup(maxvel)

Sets the maximum pursuit velocity accommodation by the saccade detector. Usually 60.

Parameters
<maxvel>: maximum pursuit velocity fixup (°/sec);

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("saccade_pursuit_fixup = %d"%(v));

3.4.6.5 setUpdateInterval

setUpdateInterval(time)

Normally set to 0 to disable fixation update events. Set to 50 or 100 milliseconds to produce updates for gaze-controlled interface applications.

Parameters
<time>: milliseconds between fixation updates, 0 turns off;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("fixation_update_interval = %d"%(time));

3.4.6.6 setFixationUpdateAccumulate

setFixationUpdateAccumulate(time)

Normally set to 0 to disable fixation update events. Set to 50 or 100 milliseconds to produce updates for gaze-controlled interface applications. Set to 4 to collect single sample rather than average position.

Parameters
<time>: milliseconds to collect data before fixation update for average gaze position;

Return Value
None;

Remark:
This function is equivalent to
EYELINK.sendCommand("fixation_update_accumulate = %d"%(time));

3.5 DisplayInfo Class

The DisplayInfo class is used to contain information on display configurations, including width, height, color bits, and refresh rate. The current display configuration can be retrieved by the getDisplayInformation() function of the pylink module. The DisplayInfo has the following four attributes.

width:	integer	Display width in screen pixels.
height:	integer	Display height in screen pixels.
bits:	integer	Color resolution, in bits per pixel, of the display device (for example: 4 bits for 16 colors, 8 bits for 256 colors, or 16 bits for 65,536 colors).
refresh	float	Refresh rate

For example:

```
from pylink import *

/*Code to open the display */
currentDisplay = getDisplayInformation();

print "Current display settings: ", currentDisplay.width, currentDisplay.height, \
            currentDisplay.bits, currentDisplay.refresh
```

3.6 EyeLinkAddress Class

The EyeLinkAddress class is used to hold addresses to EyeLink nodes. An instance of EyelinkAddress class can be initialized with the class constructor:

```
EyeLinkAddress(ip = (100,1,1,1), port = 4000),
```

where *ip* is a four-item tuple containing the IP address of the EyeLink node and *port* is the port number of the connection.

For example,

```
myAddress = EyeLinkAddress((100, 1, 1, 1), 4000).
```

The EyeLinkAddress class has the following two methods.

3.6.1 getIP

getIP()

Returns the IP address of the EyeLink node.

Parameters
None.

Return Value
A four-item tuple (integer) containing the IP address of the EyeLink node.

3.6.2 getPort

getPort()

Returns the port number of the EyeLink node.

Parameters
None.

Return Value
An integer for the port number of the connection.

3.7 EyelinkMessage

EyelinkMessage class, derived from EyelinkAddress class, is used to send and receive messages between EyeLink nodes. Instances of this class are commonly used as the return values of the nodeReceive() and getNode() methods of the EyeLinkListener or Eyelink class. An instance of EyelinkMessage class can be initialized with the class constructor:

```
EyelinkMessage(ip = (100,1,1,1), port = 4000, msg = ""),
```

where *ip* is a four-item tuple containing the IP address of the EyeLink node, *port* is the port number of the connection, *msg* is the message to be sent to or received from the node.

For example,

```
myMessage = EyelinkMessage((100, 1, 1, 1), 4000, "test").
```

In addition to the getIP() and getPort() methods of the EyeLinkAddress class, the EyelinkMessage class also has the getText() method.

3.7.1 getText

getText()

Returns the message to be sent to or received from the node.

Parameters
None.

Return Value
Text message.

3.8 Sample

The EyeLink toolkit library defines special data classes that allow the same programming calls to be used on different platforms such as MS-DOS, Windows, and Macintosh. You will need to know these classes to read the examples and to write your own experiments. In this documentation, the common data classes are: Sample class, Eye Event Classes, MessageEvent Class, and IOEvent Class. You only need to read this section if you are planning to use real-time link data for gaze-contingent displays or gaze-controlled interfaces, or to use data playback.

The EyeLink tracker measures eye position 250 or 500 times per second depending on the tracking mode you are working with, and computes true gaze position on the display using the head camera data. This data is stored in the EDF file, and made available through the link in as little as 3 milliseconds after a physical eye movement.

Samples can be read from the link by getFloatData() or NewestFloatSample() method of the EyeLink/EyeLinkLisenter class. These functions can return instances of Sample class. For example,

```
newSample = EYELINK.getFloatData().
```

The following methods can be used to retrieve properties of a Sample class instance. For example, the time of the sample can be retrieved as newSample.getTime(). Please note that all methods for the Sample class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp when camera imaged eye (in milliseconds since EyeLink tracker was activated)
getType()	integer	Always SAMPLE_TYPE
getFlags()	integer	Bits indicating what types of data are present, and for which eye(s). See eye_data.h for useful bits.
getPPD()	Two-item tuple in the format of (float, float)	Angular resolution at current gaze position in screen pixels per visual degree. The first item of the tuple stores the x-coordinate resolution and the second item of the tuple stores the y-coordinate resolution.
getStatus()	integer	Error and status flags (only useful for EyeLink II, report CR status and tracking error). See eye_data.h for useful bits.
getInput()	integer	Data from input port(s)
getButtons()	interger	Button input data: high 8 bits indicate changes from last sample, low 8 bits indicate current state of buttons 8 (MSB) to 1 (LSB)
isLeftSample()	integer	1 if the sample contains the left eye data; 0 if not.

isRightSample()	integer	1 if the sample contains the right eye data; 0 if not.
isBinocular()	integer	1 if the sample contains data from both eyes; 0 if not.
getRightEye () getLeftEye()	Instance of sample data class (see below)	Returns the sample data information from the desired eye.

The getRightEye () or getLeftEye() functions returns an instance of SampleData class, which contains the current sample position (raw, HREF, or gaze) and pupil size information of the desired eye. The following methods can be used to retrieve the attributes of an instance of the SampleData class. For example, the x gaze position of the left eye for a given sample can be retrieved as:

```
newSample = EYELINK.getFloatData()
gaze = newSample. getLeftEye().getGaze()
left_eye_gaze_x = gaze[0]
```

Method	Return Value	Contents
getRawPupil()	Two-item tuple in the format of (float, float)	Camera x, Y of pupil center. The first and second items of the tuple store pupil center in the x- and y- coordinate respectively.
getHREF()	Two-item tuple in the format of (float, float)	HREF angular coordinates. The first and second items of the tuple are for the x and y coordinates, respectively.
getGaze()	Two-item tuple in the format of (float, float)	Display gaze position (in pixel coordinates set by the **screen_pixel_coords** command). The first and second item of the tuple store the X- and Y- coordinate gaze position respectively.
getPupilSize()	Float	Pupil size (in arbitrary units, area or diameter as selected)

If certain property information not sent for this sample, the value MISSING_DATA (or 0, depending on the field) will be returned, and the corresponding bit in the flags field will be zero (see eye_data.h for a list of bits). Data may be missing because of the tracker configuration (set by commands sent at the start of the experiment, from the Set Options screen of the EyeLink II tracker, or from the default configuration set by the DATA.INI file for the EyeLink I tracker). Eye position data may also be set to MISSING_VALUE during a blink.

3.9 Eye Event

The EyeLink tracker simplifies data analysis by detecting important changes in the sample data and placing corresponding events into the data stream. These include eye-data events (blinks, saccades, and fixations), button events, input-port events, and messages. Several classes have been created to holds eye event data (start/end of fixation, start/end of saccade, start/end of blink, fixation update) information. Start events contain only the start time, and optionally the start eye or gaze position. End events contain the start and end time, plus summary data on saccades and fixations.

It is important to remember that data sent over the link does not arrive in strict time sequence. Typically, eye events (such as STARTSACC and ENDFIX) arrive up to 32 milliseconds after the corresponding samples, and messages and buttons may arrive before a sample with the same time code. This differs from the order seen in an ASC file, where the events and samples have been sorted into a consistent order by their timestamps.

The LOST_DATA_EVENT is a new event, introduced for EyeLink tracker version 2.1 and later, and produced within the DLL to mark the location of lost data. It is possible that data may be lost, either during recording with real-time data enabled, or during playback. This might happen because of a lost link packet or because data was not read fast enough (data is stored in a large queue that can hold 2 to 10 seconds of data, and once it is full the oldest data is discarded to make room for new data). This event has no data or time associated with it.

Event data returned by the getFloatData() method the EyeLink class.

For example,

newEvent = EYELINK.getFloatData().

Right now, the developer kit implements the following eye events:

Constant Name	Value	Description
STARTBLINK	3	Pupil disappeared (time only)
ENDBLINK	4	Pupil reappeared (duration data)
STARTSACC	5	Start of saccade (with time only)
ENDSACC	6	End of saccade (with summary data)
STARTFIX	7	Start of fixation (with time only)
ENDFIX	8	End of fixation (with summary data)
FIXUPDATE	9	Update within fixation (summary data for interval)
MESSAGEEVENT	24	User-definable text (IMESSAGE structure)
BUTTONEVENT	25	Button state change (IOEVENT structure)
INPUTEVENT	28	Change of input port (IOEVENT structure)
SAMPLE_TYPE	200	Event flags gap in data stream

The following section lists all of the methods to retrieve the property of a particular eye event. Please note that due to the tracker configuration, some of the property information returned may be a missing value MISSING_DATA (or 0, depending on the field). So make sure you check for the validity of the data before trying to use them. To do the tracker configuration, the user can use the setLinkEventFilter() and setLinkEventData() methods of the EyeLink class to send commands at the start of the experiment or modify the DATA.INI file on the tracker PC.

3.9.1 StartFixationEvent

The following methods can be used to retrieve properties of a fixation start event. Please note that all methods of the StartFixationEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be STARTFIX (i.e., 7)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of the fixation.
getStartGaze()	two-item tuple in the format of (float, float)	Gaze position at the start of a fixation (in pixel coordinates set by the **screen_pixel_coords** command). The first and second items of the tuple store the X- and Y- gaze position respectively.
getStartHREF()	two-item tuple in the format of (float, float)	HEADREF position at the start of a fixation. The first and second items of the tuple store the X- and Y- HREF data respectively.
getStartPupilSize()	float	Pupil size (in arbitrary units, area or diameter as selected) at start of fixation interval.
getStartVelocity()	float	Gaze velocity at the start of a fixation (in visual degrees per second).
getStartPPD()	two-item tuple in the format of (float, float)	Angular resolution at the start of fixation (in screen pixels per visual degree, PPD). The first item of the tuple stores the x-coordinate PPD resolution and the second item of the tuple stores the y-coordinate PPD resolution.

When both eyes are being tracked, left and right eye events are produced. The eye from which data was produced can be retrieved by the getEye() method.

3.9.2 EndFixationEvent

The following methods can be used to retrieve properties of a fixation end event. Please note that all methods of the EndFixationEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be ENDFIX (i.e., 8)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of a fixation.
getEndTime()	long integer	Timestamp of the last sample of a fixation.
getStartGaze()	Two-item tuple in the format of (float, float)	Gaze position at the start of a fixation (in pixel coordinates set by the **screen_pixel_coords** command).
getEndGaze()	Two-item tuple in the format of (float, float)	Gaze position at the end of a fixation (in pixel coordinates set by the **screen_pixel_coords** command).
getAverageGaze()	Two-item tuple in the format of (float, float)	The average gaze position during the fixation period (in pixel coordinates set by the **screen_pixel_coords** command).
getStartHREF()	Two-item tuple in the format of (float, float)	HEADREF position at the start of a fixation.
getEndHREF()	Two-item tuple in the format of (float, float)	HEADREF position at the end of a fixation.
getAverageHREF()	Two-item tuple in the format of (float, float)	Average HEADREF position during the fixation period.
getStartPupilSize()	Float	Pupil size (in arbitrary units, area or diameter as selected) at the start of a fixation interval.
getEndPupilSize()	Float	Pupil size (in arbitrary units, area or diameter as selected) at the end of a fixation interval.
getAveragePupilSize()	Float	Average pupil size (in arbitrary units, area or diameter as selected) during a fixation.
getStartVelocity()	Float	Gaze velocity at the start of a fixation (in visual degrees per second).
getEndVelocity()	Float	Gaze velocity at the end of a fixation (in visual degrees per second).
getAverageVelocity()	Float	Average gaze velocity during a fixation (in visual degrees per second).
getPeakVelocity()	Float	Peak gaze velocity during a fixation (in visual degrees per second).
getStartPPD()	Two-item tuple in the format of (float, float)	Angular resolution at the start of fixation (in screen pixels per visual degree).
getEndPPD()	Two-item tuple in the format	Angular resolution at the end of fixation (in screen pixels per visual degree).

	of (float, float)	

Please note that the getStartTime() and getEndTime() methods of the event class are the timestamps of the first and last samples in the event. To compute duration, subtract these two values and add 4 msec (even in 500 Hz tracking modes, the internal parser of EyeLink II quantizes event times to 4 milliseconds).

Peak velocity returned by getPeakVelocity() for fixations is usually corrupted by terminal segments of the preceding and following saccades.

3.9.3 FixUpdateEvent

The methods can be used to retrieve properties of a fixation update event are exactly the same as those for fixation end event. The event code returned by getType() method should always be FIXUPDATE (i.e., 9).

3.9.4 StartSaccadeEvent

The following methods can be used to retrieve properties of a saccade start event. Please note that all methods of the StartSaccadeEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be STARTSACC (i.e., 5)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of the saccade.
getStartGaze()	Two-item tuple in the format of (float, float)	Gaze position at the start of a saccade (in pixel coordinates set by the **screen_pixel_coords** command). The first and second items of the returned tuple store the X and Y gaze position respectively.
getStartHREF()	Two-item tuple in the format of (float, float)	HEADREF position at the start of a saccade. The first and second items of the returned tuple store the X- and Y-coordinate HREF data respectively.
getStartVelocity()	Float	Gaze velocity in visual degrees per second at the start of a saccade.
getStartPPD()	Two-item tuple in the format of (float, float)	Angular resolution at the start of saccade (in screen pixels per visual degree). The first and second items of the returned tuple store the x- and y- coordinate PPD resolution respectively.

3.9.5 EndSaccadeEvent

The following methods can be used to retrieve properties of a saccade end event. Please note that all methods for the EndSaccadeEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be ENDFIX (i.e., 8)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1

		(RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of a saccade.
getEndTime()	long integer	Timestamp of the last sample of a saccade.
getStartGaze()	Two-item tuple in the format of (float, float)	Gaze position at the start of a saccade (in pixel coordinates set by the **screen_pixel_coords** command). The first and second items of the returned tuple store the X and Y gaze position respectively.
getEndGaze()	Two-item tuple in the format of (float, float)	Gaze position at the end of a saccade (in pixel coordinates set by the **screen_pixel_coords** command). The first and second items of the returned tuple store the X and Y gaze position respectively.
getStartHREF()	Two-item tuple in the format of (float, float)	HEADREF position at the start of a saccade. The first and second items of the returned tuple store the X- and Y-coordinate HREF data respectively.
getEndHREF()	Two-item tuple in the format of (float, float)	HEADREF position at the end of a saccade. The first and second items of the returned tuple store the X- and Y-coordinate HREF data respectively.
getStartVelocity()	Float	Gaze velocity (in degrees per second) at the start of a saccade.
getEndVelocity()	Float	Gaze velocity (in degrees per second) at the end of a saccade.
getAverageVelocity()	Float	Average gaze velocity (in degrees per second) during a saccade.
getPeakVelocity()	Float	Peak gaze velocity (in degrees per second) during a saccade.
getStartPPD()	Two-item tuple in the format of (float, float)	Angular resolution at the start of saccade (in screen pixels per visual degree). The first and second items of the returned tuple store the x- and y- coordinate PPD resolution respectively.
getEndPPD()	Two-item tuple in the format of (float, float)	Angular resolution at the end of saccade (in screen pixels per degree). The first and second items of the returned tuple store the x- and y- coordinate PPD resolution respectively.

The getStartPPD() and getEndPPD() methods contain the angular resolution at the start and end of the saccade or fixation. The average of the start and end angular resolution can be used to compute the size of saccades in degrees. This Python code would compute the true magnitude of a saccade from an ENDSACC event in the following way:

```
newEvent = EYELINK.getFloatData()

if isinstance(newEvent, EndFixationEvent):
      Start_Gaze = newEvent.getStartGaze()
      Start_PPD = newEvent.getStartPPD()
      End_Gaze = newEvent.getEndGaze()
      End_PPD = newEvent. getEndPPD()

dx = (End_Gaze[0] – Start_Gaze[0]) / ((End_PPD[0] + Start_PPD[0])/2.0);
dy = (End_Gaze[1] – Start_Gaze[1]) / ((End_PPD[1] + Start_PPD[1])/2.0);
dist = math.sqrt(dx*dx + dy*dy);
```

Please note that the average velocity for saccades may be larger than the saccade magnitude divided by its duration because of overshoots and returns.

3.9.6 StartBlinkEvent

The following methods can be used to retrieve properties of a blink start event. Please note that all methods for the StartBlinkEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be STARTBLINK (i.e., 3)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of the blink.

3.9.7 EndBlinkEvent

The following methods can be used to retrieve properties of a blink end event. Please note that all methods for the EndBlinkEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be ENDBLINK (i.e., 4)
getEye()	integer	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
getRead()	integer	Bits indicating which data fields contain valid data (see eye_data.h file for details of the bits information).
getStartTime()	long integer	Timestamp of the first sample of the blink.
getEndTime()	long integer	Timestamp of the last sample of the blink.

3.10 IOEvent

IOEvent class is used to handle BUTTONEVENT and INPUTEVENT types, which report changes in button status or in the input port data. The getTime() method records the timestamp of the eye-data sample where the change occurred, although the event itself is usually sent before that sample. Button events from the link are rarely used; monitoring buttons with one of readKeybutton(), lastButtonPress(), or buttonStates() of the EyeLink class methods is preferable, since these can report button states at any time, not just during recording.

The following methods can be used to retrieve properties of an IOEvent class. Please note that all methods for the IOEvent class do not take a parameter whereas the return values are listed in the following table.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be BUTTONEVENT (i.e., 25) or INPUTEVENT (i.e., 28).
getData()	long integer	Coded event data

A ButtonEvent class, derived from the IOEvent class, is created to handle specifics of button events. For this class, in addition to the above generic IOEvent class methods, two additional methods can be used for instances of the ButtonEvent class.

buttons	getButtons()	list of integers	A list of buttons pressed or released.
states	getStates()	list of integers	The button state (1 for pressed or 0 for released).

3.11 MessageEvent

A message event is created by your experiment program and placed in the EDF file. It is possible to enable the sending of these messages back through the link, although there is rarely a reason to do this. The MessageEvent class is designed to hold information on EyeLink message events retrieved from the link. The following methods can be used with a MessageEvent class. Please note that all methods for the MessageEvent class do not take a parameter.

Method	Return Value	Contents
getTime()	long integer	Timestamp of the sample causing event (when camera imaged eye, in milliseconds since EyeLink tracker was activated)
getType()	integer	The event code. This should be MESSAGEEVENT (i.e., 24).
getText()	string	Message contents (max length 255 characters)

3.12 ILinkData

ILinkData consists of tracker status information such as time stamps, flags, tracker addresses and so on. A valid reference to this object can be obtained by calling the function EYELINK.getTrackerInfo().

Method	Equivalent field in ILINKDATA "C"	Description
getTime	Time	Time of last control event
getSampleRate	samrate	10*sample rate (0 if no samples, 1 if nonconstant)
getSampleDivisor	samdiv	Sample "divisor" (min msec between samples)
getPrescaler	prescaler	Amount to divide gaze x,y,res by
getVelocityPrescaler	vprescaler	Amount to divide velocity by
getPupilPrescaler	pprescaler	Pupil prescale (1 if area, greater if diameter)
getHeadDistencePrescaler	hprescaler	Head-distance prescale (to mm)
getSampleDataFlags	sample_data	0 if off, else all flags
getEventDataFlags	event_data	0 if off, else all flags
getEventTypeFlags	event_types	if off, else event-type flags
isInBlockWithSamples	in_sample_block	Set if in block with samples
isInBlockWithEvents	in_event_block	Set if in block with events
haveLeftEye	have_left_eye	Set if any left-eye data expected
haveRightEye	have_right_eye	Set if any right-eye data expected
getLostDataTypes	last_data_gap_types	Flags what we lost before last item
getLastBufferType	last_data_buffer_type	Buffer-type code
getLastBufferSize	last_data_buffer_size	Buffer size of last item
isControlEvent	control_read	Set if control event read with last data
isNewBlock	first_in_block	Set if control event started new block
getLastItemTimeStamp	last_data_item_time	Time field of item
getLastItemType	last_data_item_type	Type: 100=sample, 0=none, else event type
getLastItemContent	last_data_item_contents	Content: <read> (IEVENT), <flags> (ISAMPLE)
getBlockNumber	block_number	Block in file
getSamplesInBlock	block_sample	Samples read in block so far

getEventsInBlock	block_event	Events (excl. control read in block
getLastResX	last_resx	Updated by samples only
getLastResY	last_resy	Updated by samples only
getLastPupil	last_pupil	Updated by samples only
getLastItemStatus	last_status	Updated by samples, events
getSampleQueueLength	queued_samples	Number of items in queue
getEventQueueLength	queued_events	Includes control events
getQueueSize	queue_size	Total queue buffer size
getFreeQueueLength	queue_free	Unused bytes in queue
getLastReceiveTime	last_rcve_time	Time tracker last sent packet
isSamplesEnabled	samples_on	Data type rcve enable (switch)
isEventsEnabled	events_on	Data type rcve enable (switch)
getPacketFlags	packet_flags	Status flags from data packet
getLinkFlags	link_flags	Status flags from link packet header
getStateFlags	state_flags	Tracker error state flags
getTrackerDataOutputState	link_dstatus	Tracker data output state
getPendingCommands	link_pendcmd	Tracker commands pending
isPoolingRemote	polling_remotes	1 if polling remotes, else polling trackers
getPoolResponse	poll_responses	Total nodes responding to polling
getReserved	reserved	0 for EyeLink I or original EyeLink API DLL.
getName	our_name	a name for our machine
getTrackerName	eye_name	Name of tracker connected to
getNodes	nodes	Data on nodes
getLastItem	last_data_item	Buffer containing last item
getAddress	our_address	Address of our machine
getTrackerAddress	eye_address	Address of the connected tracker
getTrackerBroadcastAddress	ebroadcast_address	TSR exports
getRemoteBroadcastAddress	rbroadcast_address	

3.13 Overriding Calibration Target

Overriding calibration target is required for controlling calibration targets on broadcast applications. In C this is handled by using function pointers. However, in Python this is easily handled by over-riding specific methods namely drawCalTarget in EyeLinkListener class.

```
class BroadcastEyelink(EyeLinkListener):
        def __init__(self):
                EyeLinkListener.__init__(self)

        def drawCalTarget(self, pos):
                x = track2local_x(pos[0])
                y = track2local_y(pos[1])
                EyeLinkListener.drawCalTarget(self,(x,y))
```

4. Example: GCWindow

This section gives one specific example of programming an EyeLink experiment using the pylink module. The most useful real-time experiment is a gaze-contingent display, where the part of the display the subject is looking at is changed, or where the entire display is modified depending on the location of gaze. These manipulations require high sampling rates and low delay, which the EyeLink tracker can deliver through the link.

You should run this experiment at as high a display refresh rate as possible (at least 120 Hz). You may have to decrease the display resolution to 800x600 or even 640x480 to reach the highest refresh rates, depending on your monitor and display card and driver. Higher refresh rates mean lower delays between eye movements and the motion of the window.

This template demonstrates how to use the link's real-time gaze-position data to display a gaze-contingent window. This is an area centered on the point of gaze that shows a foreground image, while areas outside the window show the background image. You supply full-screen sized bitmaps for these, and update the display based on the current gaze position. The template demonstrates this window with text and pictures.

The graphics used in the current example is pygame (http://www.pygame.org/), please make sure that this program has been installed in your computer.

4.1 Files Required for the Example

To run the GCWindow demonstration, type "gcwindow_main.py" from the dos prompt. Make sure that you have the gcwindow_main.py and gcwindow_trial.py file in the experiment directory. The former is used for experiment setup, such as connecting or shutting down the link and experiment graphics, opening EDF file, etc. The latter is used to run the actual trials. This file handles standard return codes from trials to allow trial skip, repeat, and experiment abort. This file can be modified for your experiments, by replacing the trial instance code. In addition, other support files (sacrmeto.jpg, scr_blur.jpg, and cour.ttf) should also be included in the same directory.

4.2 Analyzing "gcwindow_main.py"

This file has the initialization and cleanup code for the experiment. You should use as much of the code and operation sequence from this file as possible in your experiments. Also, try to keep your source files separated by function in the same way as the source files in this template are: this will make the maintenance of the code easier.

4.2.1 Importing Python Source Code

At the beginning of the experiment code, all of the required modules should be imported by using the *import* statement. In this example, we need to import all of the classes and functions from the *pylink* and *pygame* modules. In addition, *gc*, *sys* and *time* modules are also required so that we can handle file operations, check for timing, and perform garbage collection. Last but not least, the *gcwindow_trial* module should also be imported.

```
from pylink import *
from pygame import *
import time
import gc
import sys

import gcwindow_trial
```

4.2.2 Initializes Experiment Graphics

The first thing the user should do is to manually initialize SDL's video subsystem and opens the graphics based on the display mode set. To make the display changes faster, a low screen resolution of 800 x 600 is used.

```
#Initializes the graphics
display.init()
display.set_mode((800, 600), FULLSCREEN |RLEACCEL |DOUBLEBUF ,16)
pylink.openGraphics()
```

4.2.3 Opening an EDF File

Following this, an EDF file ("Test.EDF") can be opened for data recording. Note that, we use the built-in constant of the pylink module EYELINK, which is an instance of the EyeLink/EyelinkListener class. The file name can also be read from a command line by the following code:

```
edfFileName = sys.argv[1];
```

```
#Opens the EDF file.
edfFileName = "TEST.EDF";
EYELINK.openDataFile(edfFileName)
```

4.2.4 EyeLink Tracker Configuration

Before recording, the EyeLink tracker must be set up. The first step is to record the display resolution, so that the EyeLink tracker and Data Viewer will work in right display pixel coordinates. We also write information on the display to the EDF file, to document the experiment and for use during analysis. The "DISPLAY_COORDS" message records the size of the display, which may be used to control data display tools.

```
pylink.flushGetkeyQueue();
EYELINK.setOfflineMode();

#Gets the display surface and sends a mesage to EDF file;
surf = display.get_surface()
EYELINK.sendCommand("screen_pixel_coords =  0 0 %d %d" %(surf.get_rect().w, surf.get_rect().h))
EYELINK.sendMessage("DISPLAY_COORDS  0 0 %d %d" %(surf.get_rect().w, surf.get_rect().h))
```

The saccade detection thresholds and the EDF file data contents are set next. Setting these at the start of the experiment prevents changes to default settings made by other experiments from affecting your experiment. The saccadic detection thresholds determine if small saccades are detected and how sensitive to noise the tracker will be. If we are connected to an EyeLink II tracker, we select a parser configuration rather than changing the saccade detector parameters:

```
if (EYELINK.getTrackerVersion()== 2):
        EYELINK.sendCommand("select_parser_configuration 0");
else:
        EYELINK.sendCommand("saccade_velocity_threshold = 35");
        EYELINK.sendCommand("saccade_acceleration_threshold = 9500");

EYELINK.setFileEventFilter("LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON");
EYELINK.setFileSampleFilter("LEFT,RIGHT,GAZE,AREA,GAZERES,STATUS");
EYELINK.setLinkEventFilter("LEFT,RIGHT,FIXATION,SACCADE,BLINK,BUTTON");
EYELINK.setLinkSampleFilter("LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS");
EYELINK.sendCommand("button_function 5 'accept_target_fixation'");
```

Calibration and drift correction are customized by setting the target size, the background and target colors, and the sounds to be used as feedback. Target size is set as a fraction of display width, so that the templates will be relatively display-mode independent. A gray background and black target is initially set for the window: this will be changed before calibration to match trial stimuli brightness. The default sounds are used for most, but the sound after drift correction is turned off.

```
pylink.setCalibrationColors((255, 255, 255), (0, 0, 0));  #Sets the calibration target and background color
pylink.setTargetSize(int(surf.get_rect().w/70), int(surf.get_rect().w/300));
                                                    #select best size for calibration target
pylink.setCalibrationSounds("", "", "");
```

```
pylink.setDriftCorrectSounds("", "off", "off");
```

4.2.5 Running the Experiment

Everything is now set up. We check to make sure the program has not been terminated, then call run_trials() to perform a block of trials. The implementation of this function, documented below, differs across experiment. For a multiple-block experiment, you would call this function once for every block, and call this function in a loop.

```
if(EYELINK.isConnected() and not EYELINK.breakPressed()):
        gcwindow_trial.run_trials(surf, tracker)
```

4.2.6 Transferring the EDF File

Once all trials are run, the EDF file is closed and transferred via the link to the subject PC's hard disk. The tracker is set to Offline mode to speed the transfer with EyeLink I. The receiveDataFile() function is called to copy the file.

```
if tracker != None:
        # File transfer and cleanup!
        EYELINK.setOfflineMode();
        msecDelay(500);

        #Close the file and transfer it to Display PC
        EYELINK.closeDataFile()
        EYELINK.receiveDataFile(edfFileName, edfFileName)
```

4.2.7 Cleaning Up

Finally, the program cleans up. It closes the connection to the eye tracker, and closes its experimental graphics.

```
EYELINK.close();
#Close the experiment graphics
pylink.closeGraphics()
display.quit()
```

4.3 Analyzing "gcwindow_trial.py"

The gcwindow_trial.py module performs the actions needed to perform block of experiment trials. The run_trials() is called to loop through a set of trial and the do_trial() function supplies the bitmap for each trial and executes the trial.

4.3.1 Initial Setup and Calibration

The run_trials() function begins by calling up the Camera Setup screen (Setup menu on the EyeLink I tracker), for the experimenter to perform camera setup, calibration, and validation. Scheduling this at the start of each block gives the experimenter a chance to fix any setup problems, and calibration can be skipped by simply pressing the 'Esc' key immediately. This also allows the subject an opportunity for a break, as the headband can be removed and the entire setup repeated when the subject is reseated.

```
NTRIALS = 4
def run_trials(surface, tracker):
        ''' This function is used to run individual trials and handles the trial return values. '''

        ''' Returns a successful trial with 0, aborting experiment with ABORT_EXPT (3); It also handles
        the case of re-running a trial. '''
        #Do the tracker setup at the beginning of the experiment.

        EYELINK.doTrackerSetup()
```

4.3.2 Trial Loop and Result Code Processing

Each block loops through a number of trials, which calls a function to execute the trial itself. The trial should return one of a set of trial return codes, which match those returned by the EyeLink trial support functions. The trial return code should be interpreted by the block-of-trials loop to determine which trial to execute next, and to report errors via messages in the EDF file. The trial loop function should return the code ABORT_EXPT if a fatal error occurred, otherwise 0 or your own return code.

The standard trial return codes are listed below, and the appropriate message to be placed in the EDF file is also given.

Return Code	Message	Caused by
TRIAL_OK	"TRIAL OK"	Trial recorded successfully
TRIAL_ERROR	"TRIAL ERROR"	Error: could not record trial
ABORT_EXPT	"EXPERIMENT ABORTED"	Experiment aborted from EyeLink Abort menu, link disconnect or ALT-F4 key
SKIP_TRIAL	"TRIAL SKIPPED"	Trial terminated from EyeLink Abort menu
REPEAT_TRIAL	"TRIAL REPEATED"	Trial terminated from EyeLink Abort menu: repeat requested

The REPEAT_TRIAL function cannot always be implemented, because of randomization requirements or the experimental design. In this case, it should be treated as SKIP_TRIAL.

This is the code that executes the trials and processes the result code. This code can be used in your experiments, simply by changing the function called to execute each trial:

```
for trial in range(NTRIALS):
        if(EYELINK.isConnected() ==0 or EYELINK.breakPressed()): break;

        while 1:
                ret_value = do_trial(trial, surface, tracker)
                endRealTimeMode()

                if (ret_value == TRIAL_OK):
                        EYELINK.sendMessage("TRIAL OK");
                        break;
                elif (ret_value == SKIP_TRIAL):
                        EYELINK.sendMessage("TRIAL ABORTED");
                        break;
                elif (ret_value == ABORT_EXPT):
                        EYELINK.sendMessage("EXPERIMENT ABORTED")
                        return ABORT_EXPT;
                elif (ret_value == REPEAT_TRIAL):
                        EYELINK.sendMessage("TRIAL REPEATED");
                else:
                        EYELINK.sendMessage("TRIAL ERROR")
                        break;

return 0;
```

4.3.3 Trial Setup

The do_trial() function first sets the trial identification title by sending a "record_status_message" command to the tracker. This is displayed at the bottom right of the EyeLink tracker display during recording. This message should be less than 80 characters long (35 characters for the EyeLink I tracker), and should contain the trial and block number and trial conditions. Such information will let the experimenter know how far the experiment has progressed, and will be essential in fixing problems that the experimenter notices during recording.

The "TRIALID" message is sent to the EDF file next. This message must be placed in the EDF file before the drift correction and before recording begins, and is critical for data analysis. EyeLink analysis software

treats this message in special ways. It should contain information that uniquely identifies the trial for analysis, including a number or code for each independent variable. Each item in the message should be separated by spaces.

```
trial_condition = ["Image-Window", "Image-Mask", "Text-Window", "Text-Mask"];

def dotrial(trial, surf, tracker):
        '''Does the simple trial'''

        message ="record_status_message 'Trial %d %s'"%(trial+1, trial_condition[trial])
        EYELINK.sendCommand(message);
        msg = "TRIALID %s"%trial_condition[trial];
        EYELINK.sendMessage(msg);
```

4.3.4 Creating Trial Bitmaps

Four trials are run to demonstrate different aspects of the gaze-contingent window manipulation. For trials 1 and 2, a clear image and a blurred image of the same picture are used. In trial 1, the foreground bitmap is a clear image and the background is a blurred one. For trial 2, the bitmaps are reversed, and the window type is set to masking. For trials 3 and 4, two bitmaps of text are created, one with characters and the other with "Xxx" words. A monospaced font (Courier) is used so that the two displays overlap. For trial 3, the normal text is in the foreground, and the "Xxx" text is outside the window. For trial 4, the bitmaps are reversed, and the window type is set to masking.

```
        #Creates the bitmap images for the foreground and background images;
        if(trial == 0):
                fgbm = getImageBitmap(1)
                bgbm = getImageBitmap(2)
        elif(trial == 1):
                fgbm = getImageBitmap(2)
                bgbm = getImageBitmap(1)
        elif(trial == 2):
                fgbm = getTxtBitmap(fgtext, (surf.get_rect().w, surf.get_rect().h))
                bgbm = getTxtBitmap(bgtext, (surf.get_rect().w, surf.get_rect().h))
        elif(trial == 3):
                bgbm = getTxtBitmap(fgtext, (surf.get_rect().w, surf.get_rect().h))
                fgbm = getTxtBitmap(bgtext, (surf.get_rect().w, surf.get_rect().h))
        else:
                return SKIP_TRIAL ;

        if (fgbm == None or bgbm == None):
                print "Skipping trial ", trial+1, "because images cannot be loaded"
                return SKIP_TRIAL ;
```

The image bitmaps are created by the getImageBitmap() function.

```
def getImageBitmap(pic):
        ''' This function is used to load an image into a new surface. '''

        ''' return image object if successful; otherwise None '''

        if(pic == 1):
                try:
                        bmp = image.load("sacrmeto.jpg", "jpg")
                        return bmp
                except:
                        print "Cannot load image sacrmeto.jpg";
                        return None;
        else:
                try:
                        bmp = image.load("sac_blur.jpg", "jpg")
                        return bmp
                except:
                        print "Cannot load image sac_blur.jpg";
                        return None;
```

The text bitmaps are created by the getTxtBitmap() function.

```
def getTxtBitmap(text, dim):
        ''' This function is used to create a page of text. '''

        ''' return image object if successful; otherwise None '''

        if(not font.get_init()):
                font.init()
        fnt = font.Font("cour.ttf",15)
        fnt.set_bold(1)
        sz = fnt.size(text[0])
        bmp = Surface(dim)

        bmp.fill((255,255,255,255))
        for i in range(len(text)):
                txt = fnt.render(text[i],1,(0,0,0,255), (255,255,255,255))
                bmp.blit(txt, (0,sz[1]*i))

        return bmp
```

4.3.5 Overview of Recording

The second part of the trial is the actual recording loop. It performs a drift correction, displays the graphics, records the data, and exits when the appropriate conditions are met. These include a response button press, the maximum recording time expired, the 'Esc' key pressed, or the program being terminated.

The sequence of operations for implementing the trial is:

- Perform a drift correction, which also serves as the pre-trial fixation target.
- Start recording, allowing 100 milliseconds of data to accumulate before the trial display starts
- Draw the subject display, recording the time that the display appeared by placing a message in the EDF file
- Loop until one of these events occurs
- Recording halts, due to the tracker Abort menu or an error
- The maximum trial duration expires
- 'Esc' is pressed, or the program is interrupted.
- A button on the EyeLink button box is pressed
- Add special code to handle gaze-contingent display updates
- Blank the display, stop recording after an additional 100 milliseconds of data has been collected
- Report the trial result, and return an appropriate error code

Each of these sections of the code is described below.

4.3.6 Drift Correction

At the start of each trial, a fixation point should be displayed, so that the subject's gaze is in a known position. The EyeLink system is able to use this fixation point to correct for small drifts in the calculation of gaze position that can build up over time. Even when using the EyeLink II tracker's corneal reflection mode, a fixation target should be presented, and a drift correction allows the experimenter the opportunity to recalibrate if needed.

The doDriftCorrect() method of the EyeLinkListener class implements this operation. The display coordinates where the target is to be displayed must be supplied. Usually this is at the center of the display, but could be anywhere that gaze should be located at the trial start. For example, it could be located over the first word in a page of text.

```
        #The following does drift correction at the begin of each trial
        while 1:
                # Checks whether we are still connected to the tracker
                if not EYELINK.isConnected():
                        return ABORT_EXPT;

                # Does drift correction and handles the re-do camera setup situations
                try:
```

```
                        error = EYELINK.doDriftCorrect(surf.get_rect().w/2,surf.get_rect().h/2,1,1)
                        if error != 27:
                                break;
                        else:
                                EYELINK.doTrackerSetup();
                except:
                        EYELINK.doTrackerSetup()
```

In the template, we told doDriftCorrect() to draw the drift correction display for us, by setting the third argument to 1. It cleared the screen to the calibration background color, drew the target, and cleared the screen again when finished.

4.3.7 Starting Recording

After drift correction, recording is initiated. Recording should begin about 100 milliseconds before the trial graphics are displayed to the subject, to ensure that no data is lost:

```
        error = EYELINK.startRecording(1,1,1,1)
        if error:  return error;
        #gc.disable();
        #begin the realtime mode
        #pylink.beginRealTimeMode(100)
```

The startRecording() method of the EyeLinkListener class starts the EyeLink tracker recording, and does not return until recording has actually begun. If link data has been requested, it will wait for data to become available. If an error occurs or data is not available within 2 seconds, it returns an error code. This code should be returned as the trial result if recording fails.

Four arguments to startRecording() set what data will be recorded to the EDF file and sent via the link. If an argument is 0, recording of the corresponding data is disabled. At least one of the data selectors must be enabled. This is the prototype and arguments to the function:

startRecording(File_samples, File_events, Link_samples, Link_events)
- *<File_samples>:* If 1, writes samples to EDF file. If 0, disables sample recording.
- *<File_events>:* If 1, writes events to EDF file. If 0, disables event recording.
- *<Link_samples>:* If 1, sends samples through link. If 0, disables link sample access.
- *<Link_events>:* If 1, sends events through link. If 0, disables link event access.

The type of data recorded to the EDF file affects the file size and what processing can be done with the file later. If only events are recorded, the file will be fairly small (less than 300 kilobytes for a 30-minute experiment) and can be used for analysis of cognitive tasks such as reading. Adding samples to the file increases its size (about 4 megabytes for a 30-minute experiment) but allows the file to reprocessed to remove artifacts if required. The data stored in the EDF file also sets the data types available for post-trial playback.

We also introduce a 100 millisecond delay after recording begins (using pylink.beginRealTimeMode()), to ensure that no data is missed before the important part of the trial starts. The EyeLink tracker requires 10 to 30 milliseconds after the recording command to begin writing data. This extra data also allows the detection of blinks or saccades just before the trial start, allowing bad trials to be discarded in saccadic RT analysis. A "SYNCTIME" message later in the trial marks the actual zero-time in the trial's data record for analysis.

4.3.8 Confirming Data Availability

After recording starts, a block of sample and/or event data will be opened by sending a special event over the link. This event contains information on what data will be available sent in samples and events during recording, including which eyes are being tracked and (for the EyeLink II tracker only) sample rates, filtering levels, and whether corneal reflections are being used.

The data in this event is only available once it has been read from the link data queue, and the waitForBlockStart() method of the EyeLinkListener class is used to scan the queue data for the block start event. The arguments to this function specify how long to wait for the block start, and whether samples, events, or both types of data are expected. If no data is found, the trial should end with an error.

```
        if not EYELINK.waitForBlockStart(100, 1, 0):
                end_trial();
                print "ERROR: No link samples received!";
```

```
        return TRIAL_ERROR;
```
Once the block start has been processed, data on the block is available.

4.3.9 Checking Recording Eye

Because the EyeLink system is a binocular eye tracker, we don't know which eye's data will be present, as this was selected during camera setup by the experimenter. After the block start, eyeAvailable() method of the EyeLinkListener class returns one of the constants LEFT_EYE, RIGHT_EYE, or BINOCULAR to indicate which eye(s) data is available from the link. LEFT_EYE (0) and RIGHT_EYE (1) can be used to index eye data in the sample; if the value is BINOCULAR (2) we use the LEFT_EYE. A message should be placed in the EDF file to record this, so that we know which eye's data to use during analysis.

```
eye_used = EYELINK.eyeAvailable(); #determine which eye(s) are available
if eye_used == RIGHT_EYE:
        EYELINK.sendMessage("EYE_USED 1 RIGHT");
elif eye_used == LEFT_EYE or eye_used == BINOCULAR:
        EYELINK.sendMessage("EYE_USED 0 LEFT");
        eye_used = LEFT_EYE;
else:
        print "Error in getting the eye information!";
        return TRIAL_ERROR;
```

4.3.10 Drawing the Subject Display

For the display drawing, we first blit the image onto the background surface and then flip it to screen to make it visible. Immediately following display flipping, the time of the retrace is read. This allows us to determine the time of stimulus onset (actually the stimulus is painted on the monitor phosphors at a delay dependent on its vertical position on the display, but this can be corrected for during analysis). If additional drawing is made, the time is read again, and used to compute the delay from the retrace. This delay is included in the messages "DISPLAY ON" and "SYNCTIME", which are then placed in the EDF file to mark the time the stimulus appeared.

```
surf.fill((255,255,255,255))
surf.blit(bgbm,((surf.get_rect().w-bgbm.get_rect().w)/2,(surf.get_rect().h-bgbm.get_rect().h)/2))
display.flip()
startTime = currentTime()
surf.fill((255,255,255,255))
surf.blit(bgbm,((surf.get_rect().w-bgbm.get_rect().w)/2,(surf.get_rect().h-bgbm.get_rect().h)/2))
        # write to the back buffer

EYELINK.sendMessage("SYNCTIME %d"%(currentTime()-startTime));
```
Please note that if you use a different graphics library other than pygame, the drawing may be done differently.

4.3.11 Recording Loop

With recording started and the stimulus visible, we wait for an event to occur that ends the trial. In the template, we look for the maximum trial duration to be exceeded, the 'Esc' key on the local keyboard to be pressed, a button press from the EyeLink button box, or the program being terminated. For the present example, the recording loop is implemented in the drawgc() function.

Any tracker buttons pressed before the trial, and any pending events from the local keyboard are discarded:

```
EYELINK.flushKeybuttons(0)
buttons =(0, 0);
```

The main part of the trial is a loop that tests for error conditions and response events. Any such event will stop recording and exit the loop, or return an error code. The process of halting recording and blanking the display is implemented as a small local function, because it is done from several places in the loop:

```
def end_trial():
        '''Ends recording: adds 100 msec of data to catch final events'''

        pylink.endRealTimeMode();
```

```
pumpDelay(100);
EYELINK.stopRecording();
while EYELINK.getkey() :
        pass;
```

The trial recording loop tests for recording errors, trial timeout, the local 'Esc' key, program termination, or tracker button presses. The first test in the loop detects recording errors or aborts, and handles EyeLink Abort menu selections. It also stops recording, and returns the correct result code for the trial:

```
error = EYELINK.isRecording()  # First check if recording is aborted
if error!=0:
        end_trial();
        return error
```

Next, the trial duration is tested. This could be used to implement fixed-duration trials, or to limit the time a subject is allowed to respond. When the trial times out, a "TIMEOUT" message is placed in the EDF file and the trial is stopped. The local end_trial() function will properly clear the display, and stop recording after an additional 100 milliseconds.

```
#Writres out a time out message if no response is made
if (currentTime() -startTime) > DURATION:
        EYELINK.sendMessage("TIMEOUT");
        end_trial();
        buttons =(0, 0);
        break;
```

Next, the local keyboard is checked to see if we should abort the trial. This is most useful for testing and debugging an experiment. There is some time penalty for using getkey() in the recording loop on the Subject PC, as this function allows Windows multitasking and background disk activity to occur. It is preferable to use escapePressed() and breakPressed() to test for termination without processing system messages, as these do not allow interruption. However, these functions may respond slowly or not at all under Windows XP if a lot of drawing is being done within the loop, or in realtime mode.

```
if(EYELINK.breakPressed()):         # Checks for program termination or ALT-F4 or CTRL-C keys
        end_trial();
        return ABORT_EXPT
elif(EYELINK.escapePressed()): # Checks for local ESC key to abort trial (useful in debugging)
        end_trial();
        return SKIP_TRIAL
```

Finally, we check for a subject response to end the trial. The tracker button box is the preferred way to collect a response, as it accurately records the time of the response. Using the getLastButtonPress() method is the simplest way to check for new button presses since the last call to this function. It returns (0, 0) if no button has been pressed since recording started, or a two-item tuple of the button number and tracker time for button repsonse. This function reports only the latest button press, and so could miss multiple button presses if not called often. This is not usually a problem, as all button presses will be recorded in the EDF file.

```
buttons = EYELINK.getLastButtonPress() # Checks for eye-tracker buttons pressed
if(buttons[0] != 0):
        EYELINK.sendMessage("ENDBUTTON %d"%(buttons[0]));
        end_trial();
        break;
```

The message "ENDBUTTON" is placed in the EDF data file to record the button press. The timestamp of this message is not as accurate as the button press event that is also recorded in the EDF file, but serves to record the reason for ending the trial. The "ENDBUTTON" message can also be used if the local keyboard as well as buttons can be used to respond, with keys translated into a button number.

4.3.12 Reading Samples

In the current example, the display is updated constantly based on the observer's current gaze position. To that purpose, code is added to the recording loop to read and process samples. This calls getNewestSample() method to read the latest sample. If new data is available, the gaze position for the monitored eye is extracted from the sample, using the getGaze() method of the SampleData class.

```
dt = EYELINK.getNewestSample() # check for new sample update
```

```
        if(dt != None):
                # Gets the gaze position of the latest sample,
                if eye_used == RIGHT_EYE and dt.isRightSample():
                        gaze_position = dt.getRightEye().getGaze()
                elif eye_used == LEFT_EYE and dt.isLeftSample():
                        gaze_position = dt.getLeftEye().getGaze()
```

4.3.13 Drawing the Window

The window code for gaze-contingent display update operates by copying areas from a foreground bitmap to draw within the window, and from a background bitmap to draw outside the window. Once the window and background are initially drawn, the code needs only to redraw those parts of the window or background that changed due to window motion. Even during saccades, only a small fraction of the window area will need to be updated.

The code here first checks whether redrawing the display is necessarily or not for a given sample.

```
        # Determines the top-left corner of the update region and determines
        # whether an update is necessarily or not
        region_topleft = (gaze_position[0]-cursorsize[0]/2, gaze_position[1]-cursorsize[1]/2)
        if(oldv != None and oldv == region_topleft):
                continue
        oldv = region_topleft
```

Otherwise, we will update the content of the cursor and shows the cursor content.

```
        if(backcursor != None): #copy the current backcursor to the surface and get a new backup
                if(prevrct != None):
                        surf.blit(pbackcursor, (prevrct.x, prevrct.y))

                pbackcursor.blit(backcursor,(0,0))
                pbackcursor.blit(backcursor,(0,0))
                prevrct = srcrct.move(0,0) #make a copy
                srcrct.x = region_topleft[0]
                srcrct.y = region_topleft[1]
                backcursor.blit(surf, (0,0),srcrct)

        else: # create a new backcursor and copy the new back cursor
                backcursor = Surface(cursorsize)
                pbackcursor = Surface(cursorsize)
                backcursor.fill((0,255,0,255))
                srcrct = cursor.get_rect().move(0,0)
                srcrct.x = region_topleft[0]
                srcrct.y = region_topleft[1]
                backcursor.blit(surf, (0,0), srcrct)
                backcursor.blit(surf, (0,0), srcrct)

        updateCursor(cursor,region_topleft, fgbm)  #Updates the content of the cursor
        surf.blit(cursor, region_topleft)          #Draws and shows the cursor content;
        display.flip()
```

Again, the code for display updating may be different if you are using a different graphics library.

4.3.14 Cleaning Up and Reporting Trial Results

After exiting the recording loop, it's good programming practice to prevent anything that happened during the trial from affecting later code, in this case by making sure we are out of realtime mode and by discarding any accumulated key presses. Remember, it's better to have some extra code than to waste time debugging unexpected problems.

Finally, the trial is completed by reporting the trial result and returning the result code. The standard message "TRIAL_RESULT" records the button pressed, or 0 if the trial timed out. This message is used during analysis to determine why the recording stopped. Any post-recording subject responses (i.e. questionnaires, etc.) should be performed before returning from the trial, to place them before the "TRIAL OK" message. Finally, the call to the getRecordingStatus() method of the EyeLinkListener class makes sure that no Abort menu operations remain to be performed, and generates the trial return code.

```
    EYELINK.sendMessage("TRIAL_RESULT %d"%(buttons[0]));
    return EYELINK.getRecordingStatus()
```

If the trial is run under the realtime mode, it is the place to turn off the mode. Also, garbage collection should be re-enabled if it has been disabled previously.

```
    ret_value = drawgc(surf, tracker, fgbm, startTime);
    pylink.endRealTimeMode();
    gc.enable();
    return ret_value;
```

4.3.15 Extending the Current Example

For many experiments, this code can be used with few modifications, except for changing the stimulus drawn. For experiments that show a single static display for each recording trial, the user simply needs to remove the code of 4.3.12 (reading samples) and 4.3.13 (performing gaze-contingent display updates).