EyeLink Programmer's Guide

Version 3.1

Contents

Chapter 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 EyeLink Software Development Kit (ELSDK) includes libraries that implement the link interface, and includes support code that makes programming simpler. It includes powerful general-purpose functions for data and file transfer, eye-image transfer, calibration, and control.

The ELSDK is implemented as a set of shared libraries, and pluggable components to a variety of graphic environments. It contains a standard set of functions for implementation of experiments using the EyeLink tracker. The toolkit can be used to produce experiments that use a standardized interface for setup, calibration, and data recording. The full set of EyeLink functions are also available for programming specially customized experiments.

The tool kit is made up of two high-level, shared libraries.

1. eyelink_core
2. eyelink_core_graphics

The eyelink_core library implements all core functions. That is, all functions in this library, directly talk to the tracker. The eyelink_core_graphics implements eyelink graphics, such as the display of camera image, calibration, validation, and drift correct. The eyelink_core_graphics is currently implemented using Simple Direct Media Layer (SDL: www.libsdl.org). However, this can easily be replaced by any other graphics library, eg., OpenGL,GDI.

1.1 Organization of This Document

We first introduce the standard form of an EyeLink experiment. This will help you 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.

Next, the organization of source files, libraries and functions in the EyeLink toolkit is described. This will help you to understand where files are located, which libraries are required, and the organization of the programs. The EyeLink programming conventions, messages and data types are also described.

The next section introduces the principles of Windows graphics programming using the support library for experiments.

A detailed analysis of the sample programs 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.

Analysis of data files using ASC files and the EDF2ASC utility is covered next. A sample program is given showing how to analyze a data file and produce data for a statistics program.

A description of the most useful EyeLink routines can be found in this documnent. You will rarely need other functions than those listed. In addition, important commands that can be sent to the EyeLink tracker are also listed in that manual. These commands can be used for on-line configuration and control.

Chapter 2

Getting Started

Please refer to the EyeLink Installation manual for instructions on how to set up the IP of the Display PC for use with the API. To install the sample experiments, double click on the installer downloaded from `https://www.sr-support.com/forums/showthread.php?t=6` This will copy in the source code, shared libraries, and utility programs. If you haven't changed any of the default settings of the installer program, the sample experiments can be found in the directory *C:\Program Files\SR Research\Eyelink\SampleExperiments*.

2.1 Upgrading to ELSDK with GDI graphics

If you already have a set of libraries or programs to use with EyeLink tracker, it may be easier for you to just continue use your programs or libraries with GDI graphics (the GDI version of the sample experiments can be found in the directory "gdi" subdirectory). It is very simple to upgrade to new version of ELSDK with GDI graphics. However, there are three ways to upgrade to the ELSDK. In all cases, please backup your existing code.

2.1.1 Upgrading without recompilation

This is the simplest way for you to upgrade your program to use the new libraries. The only thing you have to do is to make sure that , when you run your program the eyelink_exptkit20.dll provided with ELSDK is used and not the old ones. If you use the installer to install ELSDK, the installer takes care of replacing eyelink_exptkit20.dll in <windir>\system and <windir>\system32. However, if you have eyelink_exptkit20.dll in your application directory or any other place in your path you must delete them, otherwise the runtime system may load the old dll.

2.1.2 Upgrading with re-compilation without code modification

ELSDK is designed in such a way that it is 100% backwards compatible with the previous releases of eyelink_exptkit. You can continue using the existing eyelink_exptkit20.lib and simultaneously use the upgraded eyelink_exptkit20.dll for runtime. However, you will not be able to use the new functions that are added to the new eyelink_exptkit20.lib. If you want to use the new functions without linker changes then make sure your linker command is pointing to the new eyelink_exptkit20.lib.

2.1.3 Upgrading with re-compilation and with code modification

This is the recommended way of upgrading. This will make your code adapted to the new library environment.

1. In your program replace the code looks like:

   ```
   #include "eyelink.h"
   #include "w32_exptsppt2.h"    (or #include "w32_exptsppt.h")
   ```

   with

   ```
   #include "gdi_expt.h"
   ```

2. Remove the existing eyelink.h, eye_data.h, eyetypes.h, and w32_exptsppt2.h files from the project directory.

3. Once you made this change, remove the eyelink_exptkit.dll or eyelink_exptkit20.dll library from your project and add links to the eyelink_core.lib and eyelink_gdi_graphics.lib libraries. The new library files can be found in the C:\Program Files\SR Research\Eyelink\Libs by default.

For all of the above three approaches, please read the companion manual *Programming EyeLink®Experiments in Windows Version 2.1 (GDI)* for more information on programming with window GDI graphics.

2.2 Using ELSDK with SDL graphics

This document is intended for those users who want to program their experiments with SDL. Those users who want to use graphics libraries other than GDI and SDL may also read this document because the general EyeLink programming concepts would be the same across different graphics platforms and Chapter 9 of this document provides a brief introduction to implementing your own calibration graphics. The *ELSDK* DLLs and associated sample experiment templates have been updated and optimized to work under Windows 2000 and XP, but can also work (without sub-millisecond realtime performance) under Windows 95, 98 and Me. The code has been compiled under Visual C 6.0, but should be compatible with other Windows C compilers. The functions in the DLL may be called by other languages that support external DLL calls, but SR Research does not offer any support for these.

To use the SDL version of the EyeLink programming toolkit, you should have DirectX 7 or better installed in your Display PC. You should also have a video card that supports Hardware blits, color keying and double buffering (SR Research Limited recommends any high-end ATI or nVidia cards). Your subject PC monitor should support at least 16 bit depth colors and run at a screen resolution higher than 640 x 480, with refresh rate of at least 60 Hz.

Chapter 3

Overview of Experiments

The EyeLink system and this developer's kit are based on experience gained through actual research in diverse areas, including saccadic tasks, smooth pursuit, reading, and gaze contingent displays. This experience makes it possible for SR Research to write this manual aiming at providing the knowledge required for programmers to write the desired experiment program that produces valid experiment data. This toolkit contains samples for a variety of experiments, and the *ELSDK* library is designed to simplify implementation of almost any experimental task.

In addition, SR Research has studied each platform to which the EyeLink developer's kit has been ported, and has determined the best way to present graphics, control program flow, and to achieve reproducible timing. This manual and the included example source code will help you to write experiments without spending hundreds of hours in reading books and testing different methods. However, some knowledge of graphic programming and of the C language is required.

3.1 Outline of a Typical Windows Experiment

A typical experiment using the EyeLink eye tracker system will use some variation of the following 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 in the *ELSDK* library.
- 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 the following 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 types of experiments. A real-world 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 Display PC at the end of the experiment. Your experiment will also add messages to the EDF file which to identify trial conditions and to timestamp important events (such as subject responses and display changes) for data analysis. The EDF file may be processed directly using the EyeLink Data Viewer application, or converted to an ASC file and processed with your own software.

3.2 EyeLink Operation in Experiments

Several simple experiments are included in the EyeLink experiment kit, including all source code. These experiments will be discussed in detail throughout this manual. Each is an example of the basic operations needed to implement the most useful eye-tracking research: simple and complex static and dynamic displays, on-line data playback, real-time data transfer, gaze-contingent displays, and gaze-controlled computer interfaces.

For now, we will go through the simplest of the demonstration experiments included in the ELSDK kit, called simple. This includes four trials, each of which prints a single word at the center of the display. Start the EyeLink tracker, then go to the *C:\Program Files\SR Research\Eyelink\Sample-Experiments\sdl\simple* folder and execute *simple* on the Display PC. The program first asks for a file name for the EDF file that will be created on the EyeLink Host computer's hard disk. Enter 1 to 8 characters from the keyboard for the EDF file name. You may press "Esc" or click "Cancel" if you do not want to keep the EDF file(all recorded data will be discarded).

The Windows display then blanks, and the tracker displays the Setup menu screen. From this menu, the experimenter can perform camera setup, calibration, and validation. These may be practiced with the *track* application included with the *ELSDK* kit. Instructions for the camera setup, calibration and validation are included in the EyeLink User's Manual. All data collection and messaging operations are implemented by *eyelink_core.dll* and all other graphic operations, such as calibration, validation, and display of the eye image on the Display PC, are implemented by the *eyelink_core_graphics.dll*.

When the eye tracker has been set up and the subject calibrated, press the 'Esc' key on either the EyeLink

PC or the Display PC to exit the Setup menu. The experiment will immediately proceed to the first trial, and display a drift correction fixation target. Press the space bar on the Display PC or EyeLink tracker while fixating the target to perform the drift correction. Pressing the 'Esc' key during drift correction will switch the tracker to the Setup menu to re-calibrate or correct setup problems. Drift correction will resume after the Setup menu is exited with the 'Esc' key.

The screen now shows the stimulus for the first trial, a small text message. The tracker displays the subject's gaze position, overlaid on graphics indicating the stimulus position. Press a button on the EyeLink button box, or press the 'Esc' key on the Display PC keyboard to end the trial. The experiment will immediately proceed to the next trial. You may also hold down the 'Ctrl-C' key combination or press 'ALT-F4' on the Display PC to terminate trials and the experiment.

During recording, the EyeLink's Abort menu may be displayed by clicking on the "Abort" button (for EyeLink II or EyeLink1000) or pressing the 'Ctrl'-'Alt'-'A' keys on the tracker keyboard. You can now choose to enter the Camera Setup screen (the Setup menu on EyeLink I) for calibration or to restart or skip the current trial. The experiment may also be aborted. Special code in the *eyelink_core* DLL interprets your selections in this menu to control the sequencing of the experiment.

After the final trial, you will be prompted for a local name for the EDF file recorded during the experiment. Unless you press the 'Esc' key or select "Cancel", the file will be transferred from the EyeLink computer to the Subject PC. Files may also be copied using the *eyelink_getfile* application. The experiment then disconnects from the tracker and exits.

Remarks:
The simple.exe can be launched by simply clicking on the icon of the executable file. It can also be run as a command-line program. Open a command prompt and type "simple.exe", with one or several of the following optional command-line parameters:

- width <width of the screen>
- height <height of the screen>
- bpp <depth of the screen>
- tracker <tracker address >
- refresh <refresh rate>

The *width* and *height* parameters should be used together to override the default screen resolution (which uses the current display settings). The *tracker* parameter allows the user to use an alternative address for the tracker. The default address for the tracker is 100.1.1.1.

3.3 Features of the ELSDK Library

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 Display PC to aid subject setup, and record data with the experiment following the display of relevant 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 Display PC keyboard are transferred to the EyeLink PC and operate the tracker during setup. The buttons pressed on the tracker button box may be used to control the execution of the experiment on the Display PC.

All of these operations are implemented through the *eyelink_core* DLL library. 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.

Some of the features in the *ELSDK* library are:

- Simple connection to EyeLink system
- Execution of eye tracker commands with error detection, for tracker configuration, opening files, etc.
- Time stamped messages placed in tracker EDF file (1 millisecond time accuracy)
- Internal clocks with millisecond and microsecond resolution.
- Monitoring of tracker button box and keyboard
- One line of C code to perform all aspects of drift correction and Setup menu
- One line of C code to transfer data files
- Transfer of camera images and display on Display PC with the help of *eyelink_core_graphics* or any other plugged in graphics library.
- Real-time access to tracker data and eye movement events (such as fixations and saccades)
- Playback of eye-movement data between trials, which reduces the need for processing data during recording when data analysis within experiments is desired.
- Ability to find and communicate with other EyeLink applications running on other computers.
- Ability for multiple computers to share a single eye tracker, for example to perform real-time data analysis or control, or to enable the use of special display devices.
- Pluggable graphic component. Any graphic environment can be used to display tracker images and stimuli.
- No display mode support. This can be very useful for applications where multiple computers share single or multiple trackers.

A complete list of ELSDK routines are grouped into functional groups and given below.

- Initialize EyeLink Library
- Access Local Time
- Access Tracker Time
- Setup EyeLink tracker
- Keyboard Input Functions
- Data file utilities
- Application/Thread priority control
- Graphics display options
- Extract extended data from samples and events
- Time stamped messages to log file
- Online velocity and acceleration calculation
- Utility function to save bitmaps
- Record control and data collection
- Accessing and reporting error messages

- Playback and data acquisition.
- Message and Command Sending/Receiving
- Message Pump functions
- Tracker Mode functions
- Eyelink Button Functions

Programmers may also want to look at the *eyelink.h*, *core_expt.h*, *gdi_expt.h*, and *sdl_expt.h* C header files in the *SR Research\Eyelink\Includes\eyelink* directory.

3.4 Displays and Experiments

The eyelink_core DLL library handles the details of interfacing to the eye tracker, and of performing tasks such as calibration and setup. Its internal millisecond and microsecond clocks solve the problem of timekeeping for experiments. The major remaining problem for programmers is the proper display of stimuli.

The eyelink_core_graphics DLL library is built on top of SDL and associated libraries. This allows the user to get the best graphics performance on a given system and still it is simple to use. For example, in windows, SDL wraps DirectX. This allows the user to use hardware-accelerated blit and flip. The hardware assisted blit can be much faster than a software blit. For example, a full screen blit at 1600 x 1200 x 32 in a decent video card can be performed in less than 4 msec, whereas the same operation may take up to 100 msec in a software blit. SDL provides hardware flips if the configuration environment allows it. The hardware flip can be used to pinpoint the exact time when the display is made visible to the subject. In addition, SDL supports alpha blending and color keys.

3.4.1 Synchronizing to the Display

With SDL graphics, the exact time the display is modified can be recorded by sending a time stamped message to the eye tracker when drawing is completed. Some of the uncertainty caused by the delay between drawing and display on the monitor may be removed by waiting for the start of the monitor's refresh before drawing. However, we do not know how long to wait for the next refresh. So, Eyelink SDK modified the original SDL library to provide a new API function SDL_FlipEx(). This will not return till the real next retrace. That is, this function will return as soon as the image is displayed on the screen.

To accurately mark the time of onset, we need to place a display onset message in the EDF file. The method we will use in our examples is fast, accurate, and can be used with any drawing method that is sufficiently fast (including bitmap display, as we will see next). This technique requires the following steps:

- Draw the image to the back buffer
- Flip to the back buffer with SDL_FlipEx()
- Send SYNCTIME message to the EDF file.

Remarks:

Preceding the message with the delay of the message from retrace will allow new EyeLink analysis tools to automatically correct the time of the message. If a message begins with a number, this number will be subtracted from the time of the message, and the second item in the message is used to identify the message type.

3.4.2 Using SDL_Surfaces for Display

The best method in SDL for rapidly displaying complex graphics at the start of the trial is to use an SDL Surface that is initialized with SDL_CreateRGBSurface(). This is like an invisible display stored in memory, which can be used to draw images, and write texts. Most of the experiments (except the "Simple" template) use this mechanism to display images and texts on the screen.

Once the stimulus has been drawn to an SDL Surface, it can be rapidly blitted to the display, if the video driver supports hardware acceleration, you will be able to blit full screens to the display in less than 3 msec in a decent video card. For most video cards and display resolutions, this can be done in much less than one refresh period (1000ms/refresh rate).

Chapter 4

Programming Experiments

Programmers are strongly recommended to read this section carefully, and follow given instructions. Doing so will prevent future problems with upgrading to new development kit releases, and will make your experiments easier to modify in the future.

4.1 Important Programming Notes

The header files in this toolkit should never be modified, and only copies of the source files should be changed. Some files are designed as templates for creating your own experiments. These should be copied as new files in a new folder before being modified. If problems are found with any other files, contact SR Research Ltd. before making changes.

DO NOT under any circumstances edit or combine the header files. This will make future upgrades of the toolkit or EyeLink functions impossible. SR Research Ltd. will not support any code written in this way. It is important for researchers to supervise their programmers to prevent unsupportable software from being developed.

You should try to preserve the functionality of each source file from the examples while developing new experiments. **Make as few changes as possible, and do not reorganize the files in any way.** This code will be updated regularly, and you will be responsible for making any required changes to your modified files. If proper care is taken, file comparison utilities (such as the *Windiff* application included with Microsoft Visual C Studio) can be used to help find the differences. However, this will only work if the order of functions in the source code is not preserved, and if there is a match between functions in the old and new source files.

4.2 Programming Tools and Environment

The templates in this toolkit were programmed with Microsoft Visual C++ 6.0, under the Microsoft Developer's Studio. Most 32-bit C compilers are compatible with this compiler, when programming Windows applications. If you are using Visual C, you can simply double-click on a workspace (.dsw) file to open and build a sample experiment. For other compilers, you will have to create a project using the files as listed for each project.

4.3 Starting a New Project

The source code for each sample experiment template's project is stored in a folder in the *C:\Program Files\SR Research\Eyelink\SampleExperiments\sdl* folder. The easiest way to start a new experiment is to copy all files from the project and the library folder together: this will prevent inadvertently editing the header files influencing your project. To create a new working copy:

- Create a new folder for your project.
- Select all files in the directory containing the template you want to base your new project on. Copy (drag with the Ctrl key held down) the files to your new folder
- Rename the project (.dsp) file in the new folder to the new experiment's name.
- Rename the project header file (for example "simple.h" of the simple template) as your new experiment's link header file. This will contain declarations for the new functions and variables you may add to your experiment. Use the Visual Studio "Find in Files" tool to find all occurrences of the old header file in the files, and change these to the name of your experiment's header file.

Once an experiment has been developed, it will be almost certain that you will need to produce several variations of it. You should duplicate and rename the original project's folder for each new version of the project, to prevent older versions from being overwritten. This is also the safest way of backing up your work during development, as older versions can still be tested.

4.4 Planning the Experiment

Before beginning the implementation of a new experiment, be sure that the overall design of the experiment is clear. Be aware of how each trial is to be terminated, and what the important display elements of each trial are. If randomization is to be built into the experiment, understand how it is to be produced.

In many cases, the experiment will need to be developed in several stages, starting from a basic trial, which will be used to refine timing and graphics. After initial testing, the design can be refined to create the final experiment. However, the requirements for the basic trial will probably not change.

The most important points to determine before beginning a project are:

- How complex are the display graphics? This will determine if the display can be drawn directly at the start of each trial, or if you will need to draw to a bitmap and copy it to the display.
- What subject responses are required? Trials will usually end with a button press by the subject, or run for a fixed period of time. Code to detect these conditions is included in the sample code.
- Is the display static? If not, choose a template that uses real-time mode, and add drawing commands to the recording loop if the display is to change.
- Is real-time eye data required during recording? Usually only gaze-contingent displays really need to get data through the link during recording. Writing data to the Display PC hard disk during recording should be avoided, as this will severely reduce the accuracy of any timing your program needs to do. If you must analyze the data during the experiment, use data playback after the trial ends. Whenever possible, use data from the recorded EDF file instead of on-line analysis.

- What data is needed for analysis? Be sure the EDF file contains the correct data types and messages so you can analyze the experiment later. You may also want to include messages documenting parameters that do not change between trials so that experiment versions can be tracked. The EDF file should be designed as a complete archive of an experimental session.

- Do you need to change any of the default calibration settings or the display? In almost all cases, the only changes that need to be made are setting the target and background color, and changing the target size to match the display resolution.

4.5 Developing and Debugging New Experiments

The quickest way to develop a new experiment is to start with one of the experiment templates, and create a single trial. You may want to place your graphics code in a function, and call it directly instead of from within a trial, to simplify development.

Begin by implementing the graphics for a typical trial, as these are the most time-critical element. Read the time from `current_msec()` or `current_usec()` before and after drawing, in order to compute total drawing time. Note that many profiling tools (including the profiler in Microsoft Visual C) may not give the correct results when measuring the timing of graphics calls, due to context switches in the Windows kernel. In fact, running under a debugger will almost certainly disable real-time mode, so once the graphics seem to be working run the application using the "execute" command or outside the Visual Studio environment to measure delays.

At this stage of development, you do not need a subject or even the EyeLink tracker to develop and test your code. There are two ways to simplify development, while retaining the use of the EyeLink support functions and timing.

4.5.1 Simulated Link Mode

The link connection can be simulated for early development work, or when the EyeLink tracker is not available. By calling `open_eyelink_connection()` with an argument of 1 instead of 0, your program will not attempt to connect to the EyeLink tracker, but simply initializes the *eyelink_core* library. Try this with one of the sample programs, and note that 'Esc' exits the Setup menu mode as usual, while the spacebar can be used for drift correction and 'Esc' can be used to end trials.

In the simulated mode, all the millisecond and microsecond timing functions are available for benchmarking the graphics. Most EyeLink commands will return a plausible result in the simulation mode, except that real-time data is not available from the link. Your code can test if the link is simulated by calling `eyelink_is_connected()` as usual. This will return -1 instead of 1 indicating that the connection is simulated.

4.5.2 Mouse Simulation Mode

When real data is needed for development, or recording is needed to test the inclusion of data messages, you can still work without a subject. Change the argument to calling `open_eyelink_connection()` back to 0, and restart the EyeLink tracker in mouse-simulation mode (Toggle the "Mouse Simulation" button in the Set Options screen for EyeLink II or EyeLink 1000, or start the EyeLink I tracker using the '-m' command line option). This allows you to run the experiment with all eye tracking and button functions available, but without the need to setup and monitor a subject. All tracker functions (except camera image display) are available in mouse mode, including EDF file recording, calibration, drift correction and realtime data.

You can skip calibrations in this mode (press ’Esc’ as soon as the display blanks on the Display PC and the EyeLink display shows the calibration menu), or allow the tracker to automatically step through the calibration sequence. Use the tracker PC’s mouse to simulate subject gaze during recording, by moving the cursor into the gaze window of the EyeLink II or EyeLink1000 tracker, or placing the ’X’ cursor of EyeLink I to the position desired, and clicking the left mouse button to produce a "saccade" to this position. Hold the left mouse button down to continuously move the point of gaze. A blink is produced when the right mouse button is held down.

Once the experiment has been debugged using the mouse, test the program with yourself and at least with one other person as a subject. This will help to identify any problems such as flickers in the display, problems with graphics, or problems with the experimental design. Be sure to analyze the resulting data files at this point, to detect problems such as missing data or messages.

4.6 Converting Existing Experiments

Read this section if you are porting existing non-eye tracking experiments to EyeLink. The *ELSDK* library was designed to simplify the procedure of adding eye movement analysis to existing experiments. This involves adding calls to perform setup and drift correction, start and stop recording, and adding messages to record trial data. Modular code with separate functions for trials and blocks will make the conversion easier: you should split up functions that have more than 200 lines, and separate out randomization, graphics, and data recording code into separate functions.

In addition, any special timing code in your experiment should be replaced with calls to the EyeLink `current_msec()` function. You don’t need to use timer toolbox functions directly. Replace any special code for user input with calls to the `eyelink_last_button_press()` function, which detects the presses of the eye tracker buttons. These buttons are logged directly into the EDF file. You should never use the keyboard for subject response in reaction-time experiments, as the delays introduced by the operating system are highly variable.

Modify your trials to perform a drift correction at their start instead of displaying a fixation point. Ideally, you should separate the drawing code from the old experiment and include it in a new function. Then call it from within the recording loop, or draw into a bitmap before the trial begins. Finally display the drawing during the trial (as in the cases of most other example templates). You can make any display changes required for animation or masking from within the recording loop and draw directly to the display.

Most non-eye tracking experiments create an output file and write the results of each trial into this file. Instead, you should write this data into the EDF file by sending data messages using `eyemsg_printf()`. This will integrate eye data, experiment events, and subject responses in the same file. Place trial condition data in a "TRIALID" message before the start of the trial, and end each trial with a "TRIAL OK" message. Send messages to mark display onset ("SYNCTIME") or display changes, and to record subject responses. Compute the reaction time by analyzing the EDF file later. If you must analyze eye-movement data on-line (for example, to give feedback to the subject), play back the last trial (as in *playback_trial.c* of the *eyedata* template) rather than trying to analyze data during recording.

Chapter 5

Developer's Toolkit Files

The EyeLink eye-tracking system is designed to implement most of the required software base for data collection and processing. The developer's kit includes the *eyelink_core*, *eyelink_core_graphics* and associated libraries that implement the link interface and support functions that make programming simpler. These include functions to implement calibration and full-screen display, as well as synchronization with display refresh, real-time mode support, and simplified keyboard and mouse access.

5.1 Libraries and Files

This section documents the relationship between the files required for creating an experiment program, and identifies each file and folder.

5.1.1 Libraries

The *eyelink_core.dll* libraries implement the TCP/IP link, timing, and all functions documented in *eyelink.h*. It also contains the functions declared in *sdl_expt.h*, which simplify programming of file transfer and recording. The *eyelink_core_graphics.dll* implements the graphic support functions (such as camera image display, calibration, validation, and drift correction graphics) that may be used by *eyelink_core.dll*.

If you used the EyeLink display software installer program to install the sample experiments, you do not have to worry about the path of the .dll and .lib files and the header files, directory. Those files can be found at *C:\Program Files\SR Research\Eyelink\Includes* and *C:\Program Files\SR Research\Eyelink\Libs* folders. You should always include the import library for this DLL (*eyelink_core.lib* and other .lib files) in your projects. If you plan to use sound in your experiments, the Windows import library *winmm.lib* should also be linked.

If you want to copy the experiment programs between computers, makes sure that the whole directory of these library and header files are also copied to the new computer. To compile the examples, you will have to add the following entries to the environment variables of your computer.

```
INCLUDE = C:\\Program Files\\SR Research\\Eyelink\\Includes\\eyelink;
          C:\\Program Files\\SR Research\\Eyelink\\Includes\\sdl;
PATH = C:\\Program Files\\SR Research\\Eyelink\\LIBS;
```

(Note: The actual settings depend on the locations where you copied the "Includes" and "Libs" folders to. To avoid setting these parameters by yourself, you may simply run the EyeLink display software installer program in your target computer).

5.1.2 Required Source Files

Each source file of your experiment should include these header files:

```
#include "sdl_expt.h"    // references all you would need to develop your
                         // experiment based on SDL  and eyelink tracker.
```

In addition, your project's source files should use a header file that contains declarations for functions and variables that need to be shared between files (study the *simple.h* file of the *Simple* template as an example). Experienced programmers know that using a header file will save hours of debugging later, and helps to document your work.

The following table lists a summary of the minimum set of header files and libraries required to compile and run your experiment:

eyetypes.h	Declarations of basic data types.
eye_data.h	Declaration of complex EyeLink data types and link data structures.
eyelink.h	Declarations and constants for basic EyeLink functions, Ethernet link, and timing.
sdl_expt.h	Declarations of *eyelink_core_graphics* functions and types. This file will also reference the other EyeLink header files.
core_expt.h	Declarations of *eyelink_core* functions and types. This file will also reference the other EyeLink header files.

eyelink_core.dll	Implements basic EyeLink functions, Ethernet link, generic support functions and timing.
eyelink_core_graphics.dll	Implements the graphic support functions that may be used by *eyelink_core.dll*. This links to SDL libraries. The function `init_expt_graphics()` should be called to register with *eyelink_core.dll*.
Eyelink_w32_comp.dll	Implements some windows specific dialogs such as `edit_dialog()`.
eyelink_core.lib	Import library for *eyelink_core.dll*. Link with your code.
Eyelink_core_graphics.lib	Import library for *eyelink_core_graphics.dll*. Link with your code.
eyelink_w32_comp.lib	Import library for *eyelink_w32_comp.dll*. Link with your code.

These files may be useful for specific functionality:

sdl_image.h	This is also part of the SDL-associated library. If you need any image loading use this for creating SDL surfaces out of images. For more detail visit `http://www.libsdl.org/projects/SDL_-image/`.

The source and header files listed above should never be modified. If problems are found with any other files, contact SR Research Ltd. before making changes. Always make changes to a renamed copy of the source file. **Do not under any circumstances edit or combine the header files, or combine parts of the source files together, or copy parts of files to your own code.** This will make it difficult for you to use future upgrades of the toolkit.

Chapter 6

EyeLink Programming Conventions

The *ELSDK* library contains a set of functions used to program experiments on a variety of platforms, such as MS-DOS, Windows, Linux, MacOS 9 and MacOS X. 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 Data Viewer to process your EDF files.

6.1 Standard Messages

Experiments should place certain messages into the EDF file, to mark the start and end of trials. This 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 *ELSDK* 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 of 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, 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. See *trial.c* of the *Simple* template for an example of how to generate these messages.

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.

6.1.1 Trial Return Codes

The recording support functions in the *eyelink_core* library return several standard error codes. Your trials should return these codes as well, so that sequencing can be controlled by the return code. An example of this sequencing is given in *trials.c* of the *Simple* template.

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 or because link disconnected
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 like SKIP_TRIAL.

Chapter 7

EyeLink Data Types

The *eyelink_core* library defines special data types that allow the same programming calls to be used on different platforms such as MS-DOS, Windows, Linux, and Macintosh. You will need to know these types to read the examples and to write your own experiments. Using these types in your code will also help to prevent common program bugs, such as signed/unsigned conversions and integer-size dependencies.

7.1 Basic Portable Data Types

Several platform-portable data types are defined in *eyetypes.h*, which is automatically included in *eyelink.h* and *eye_data.h*. This creates the data types below:

Type Name	Data	Uses
Byte	8-bit unsigned byte	images, text and buffers
INT16	16-bit signed word	Return codes, signed integers
UINT16	16-bit unsigned word	Flags, unsigned integers
INT32	32-bit signed dword	Long data, signed time differences
UINT32	32-bit unsigned dword	Timestamps

7.2 Link Data Types

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 library defines a number of data types that are used for link data transfer, found in *eye_data.h*. These are based on the basic data types above. The useful parts of these structures are discussed in the following sections.

There are two basic types of data available through the link: samples and events.

7.2.1 Samples

The EyeLink tracker measures eye position 250 or 500 or 1000 times per second depending on the tracker and 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 `eyelink_get_float_data()` or `eyelink_newest_float_sample()`. These functions store the sample data as a structure of type `FSAMPLE`:

```
typedef struct {
                UINT32 time;            // time of sample
                INT16  type;            // always SAMPLE_TYPE

                UINT16 flags;           // flags to indicate contents
                // binocular data: indices are 0 (LEFT_EYE) or 1 (RIGHT_EYE)
                float  px[2], py[2];    // pupil xy
                float  hx[2], hy[2];    // headref xy
                float  pa[2];           // pupil size or area
                float gx[2], gy[2];     // screen gaze xy
                float rx, ry;           // screen pixels per degree (angular resolution)


                UINT16 status;          // tracker status flags
                UINT16 input;           // extra (input word)
                UINT16 buttons;         // button state & changes

                INT16  htype;           // head-tracker data type (0=noe)
                INT16  hdata[8];        // head-tracker data (not prescaled)
             } FSAMPLE;
```

Each field contains one type of data. If the data in a field was not sent for this sample, the value MISSING_DATA (or 0, depending on the field) will be stored in the field, 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 or EyeLink1000 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, but the flags will continue to indicate that this data is present.

The sample data fields are further described in the following table:

Field	Contents
Time	Timestamp when camera imaged eye (in milliseconds since EyeLink tracker was activated)
Type	Always `SAMPLE_TYPE`
Flags	Bits indicating what types of data are present, and for which eye(s)
px, py	Camera X, Y of pupil center
hx, hy	HEADREF angular gaze coordinates
Pa	Pupil size (arbitrary units, area or diameter as selected)
gx, gy	Display gaze position, in pixel coordinates set by the screen_pixel_coords command
rx, ry	Angular resolution at current gaze position, in screen pixels per visual degree
Status	Error and status flags (only useful for EyeLink II and EyeLink1000, report CR status and tracking error). See *eye_data.h* for useful bits.
Input	Data from input port(s)
Buttons	Button input data: high 8 bits indicate changes from last sample, low 8 bits indicate current state of buttons 8 (MSB) to 1 (LSB)
Htype	Type of head position data (0 if none) (RESERVED FOR FUTURE USE)
Hdata	8 words of head position data (RESERVED FOR FUTURE USE)

7.2.2 Event Data

The EyeLink tracker simplifies data analysis (both on-line and when processing data files) 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.

Events may be retrieved by the `eyelink_get_float_data()` function, and are stored as C structures. All events share the `time` and `type` fields in their structures. The `type` field uniquely identifies each event type:

```
       // EYE DATA EVENT: all use FEVENT structure
#define STARTBLINK 3    // pupil disappeared, time only
#define ENDBLINK   4    // pupil reappeared, duration data
#define STARTSACC  5        // start of saccade, time only
#define ENDSACC    6    // end of saccade, summary data
#define STARTFIX   7    // start of fixation, time only
#define ENDFIX     8    // end of fixation, summary data
#define FIXUPDATE  9    // update within fixation, summary data for interval

#define MESSAGEEVENT 24  // user-definable text: IMESSAGE structure

#define BUTTONEVENT  25  // button state change:  IOEVENT structure
#define INPUTEVENT   28  // change of input port: IOEVENT structure

#define LOST_DATA_EVENT 0x3F   // NEW: Event flags gap in data stream
```

Events are read into a buffer supplied by your program. Any event can be read into a buffer of type ALLF_EVENT, which is a union of all the event and sample buffer formats:

```
typedef union {

                FEVENT    fe;
                IMESSAGE  im;
                IOEVENT   io;
                FSAMPLE   fs;

              } ALLF_DATA ;
```

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 newly added event, introduced for 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.

7.2.3 Eye Data Events

The EyeLink tracker analyzes the eye-position samples during recording to detect saccades, and accumulates data on saccades and fixations. Events are produced to mark the start and end of saccades, fixations and blinks. When both eyes are being tracked, left and right eye events are produced, as indicated in the `eye` field of the `FEVENT` structure.

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. This includes start and end and average measures of position and pupil size, plus peak and average velocity in degrees per second.

```
typedef struct  {
                UINT32 time;       // effective time of event
                INT16  type;       // event type
                UINT16 read;       // flags which items were included
                INT16  eye;        // eye: 0=left,1=right
                UINT32 sttime, entime;   // start, end sample timestamps

                float  hstx, hsty;       // href position at start
                float  gstx, gsty;       // gaze or pupil position at start
                float  sta;                  // pupil size at start
                float  henx, heny;       // href position at end
                float  genx, geny;       // gaze or pupil position at end
                float  ena;                  // pupil size at start

                float  havx, havy;       // average href position
                float  gavx, gavy;       // average gaze or pupil position
                float  ava;                  // average pupil size
                float  avel;             // average velocity
                float  pvel;             // peak velocity
                float  svel, evel;       // start, end velocity

                float  supd_x, eupd_x;   // start, end angular resolution
                float  supd_y, eupd_y;   // (pixel units-per-degree)
                UINT16 status;           // error, warning flags
                } FEVENT;
```

The `sttime` and `entime` fields of an end event are the timestamps of the first and last samples in the event. To compute duration, subtract these and add 1 sample duration (i.e., 4 ms for a 250 hz recording, 2 ms for a 500 hz recording and 1 msec for a 1000 hz recording).

Each field of the FEVENT structure is further described in the following table:

Field	Contents
time	Timestamp of sample causing event (when camera imaged eye, in milliseconds since EyeLink tracker was activated)
type	The event code
eye	Which eye produced the event: 0 (LEFT_EYE) or 1 (RIGHT_EYE)
read	Bits indicating which data fields contain valid data. Empty fields will also contain the value MISSING_DATA
gstx, gsty genx, geny gavx, gavy	Display gaze position, in pixel coordinates set by the screen_pixel_coords command. Positions at start, end, and average during saccade, fixation or FIXUPDATE period are reported.
hstx, hsty henx, heny havx, havy	HEADREF gaze position at start, end, and average during saccade, fixation or FIXUPDATE period.
sta, ena, ava	Pupil size (arbitrary units, area or diameter as selected), at start, and average during fixation of FIXUPDATE interval.
svel, evel, avel, pvel	Gaze velocity in visual degrees per second. The velocity at the start and end of a saccade or fixation, and average and peak values of velocity magnitude (absolute value) are reported.
supd_x, supd_y eupd_x, eupd_y	Angular resolution at start and end of saccade or fixation, in screen pixels per visual degree. The average of start and end values may be used to compute magnitude of saccades.
Status	Collected error and status flags from all samples in the event (only useful for EyeLink II and EyeLink1000, report CR status and tracking error). See *eye_data.h* for useful bits.

Peak velocity for fixations is usually corrupted by terminal segments of the preceding and following saccades. Average velocity for saccades may be larger than the saccade magnitude divided by its duration, because of overshoots and returns.

The supd_x, supd_y, eupd_x, and eupd_y fields are the angular resolution (in pixel units per visual degree) 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 C code would compute the true magnitude of a saccade from an ENDSACC event stored in the buffer evt :

```
dx = (evt.fe.genx - evt.fe.gstx) /
      ((evt.fe.eupd_x + evt.fe.supd_x)/2.0);
dy = (evt.fe.geny - evt.fe.gsty) /
      ((evt.fe.eupd_y + evt.fe.supd_y)/2.0);
dist = sqrt(dx*dx + dy*dy);
```

When reading real-time data through the link, event data will be delayed by 12 to 24 milliseconds from the corresponding samples. This is caused by the velocity detector and event validation processing in the EyeLink tracker. The timestamps in the event reflect the true (sample) times.

7.2.4 Button and Input Events

BUTTONEVENT and INPUTEVENT types are the simplest events, reporting changes in button status or in the input port data. The time field records the timestamp of the eye-data sample where the change occurred, although the event itself is usually sent before that sample. The data field contains the data after the change, in the same format as in the FSAMPLE structure.

Button events from the link are rarely used; monitoring buttons with one of `eyelink_read_keybutton()`, `eyelink_last_button_press()`, or `eyelink_button_states()` is preferable, since these can report button states at any time, not just during recording.

```
typedef struct  {
                UINT32 time;       // time logged
                INT16  type;       // event type:

                UINT16 data;       // coded event data
                } IOEVENT;
```

7.2.5 Message Events

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. Although this method might be used to determine the tracker time (the `time` field of a message event will indicate when the message was received by the tracker), the use of `eyelink_tracker_time()` is more efficient for retrieving the current time from the eye tracker's timestamp clock. The eye tracker time is rarely needed in any case, and would only be useful to compute link transport delays.

```
typedef struct  {
                UINT32 time;       // time message logged
                INT16  type;       // event type: usually MESSAGEEVENT

                UINT16 length;  // length of message
                byte   text[260];   // message contents (max length 255)
                } IMESSAGE;
```

Chapter 8

Issues for Programming Experiments

Programming time-critical experiments (and this includes most eye-tracking experiments) requires an understanding of both the requirements for proper programming of experiments, as well as the quirks of the various versions of the operating systems. For programmers who have written experiments under DOS, writing applications for multi-tasking operating systems such as Windows in C environment can be confusing and full of pitfalls. Using a "simple" scripting languages, Delphi, or Visual Basic just hides the problems, resulting in experiments that have unpredictable timing and produce suspected or useless data - worst of all, the programmer and experimenter will not know that this is the case. In general, multi-tasking operating system programmers are not exposed to the concept of deterministic timing, and do not know how to achieve it under such operating systems. It is the goal of this section to educate the programmer in the art of deterministic experiment programming possible under Windows, and the goal of the code and libraries in this developer's kit is to ease the development of such programs. This is a large task, but we hope this helps.

8.1 Issues for DOS Programmers

For those programmers who come from DOS or other deterministic programming environments, Windows and other similar operating systems can be a confusing environment. A typical Windows programming book has hundreds of pages on windows, dialog boxes, message handlers and other items. It assumes the program will only act to respond to key presses or mouse clicks, or other system events. But what you want to do in a typical experiment are a few basic things that don't seem to match the typical Windows programming model: displaying graphics that cover the whole display, waiting for user input, and executing trials in a fixed order and with exact timing. You certainly don't want other applications or the operating system causing long, unpredictable delays during stimulus presentation.

8.2 Issues for Windows Programmers

A typical Windows programmer relies on a high-level programming language such as Visual Basic, Microsoft Foundation Classes, Delphi, or another language that provides a "wrapper" over the complexity of Windows interface code. The problem here is that such tools are designed to make your application coexist with other programs, giving up time whenever possible to Windows and other applications readily.

This is exactly the opposite of what needs to be done to create proper experiments - we need to be greedy, keeping all the computer time to ourselves, at least during trials or dynamic display sequences. Between these, it is safe to use dialog boxes and other Windows tools. These blocks of time-critical code should be written carefully, and Visual Basic should not be used in these sections. If you insist on using Visual Basic, it should be possible to write simple libraries in C to run trials, then to call these from Visual basic, which can then be used to control the experiment, load stimuli, and so on.

This manual and the sample code show how things should be done in order to write time-critical code. Just as importantly, hard-to-write sections of the code such as display of camera images and calibration have been encapsulated in the EyeLink tracker and the *eyelink_core* and *eyelink_core_graphics* DLLs.

8.3 Windows Timing Issues

The most important issue that needs to be covered is the timing of experiments (and other programs) running under Windows. This determines what it is possible to do in an experiment, and the strategy that needs to be used to obtain the desired results.

Under a single-tasking operating system such as DOS, your experiment ran alone on the PC: except for system operations such as writing to disk, you had control over what happened and when. For example, if graphics had to be presented at precise timings, you simply created a loop and waited until the proper time to change the display had arrived. But Windows is a *multitasking* operating systems. This means that other programs can steal time from you at any moment. If you don't have control over the computer at the instant you should be updating the display, then the timing of you experiment will be off. If you're running a gaze-contingent window display, then the window movement may be delayed(an undesired effect). In addition, a large number of Windows functions are used by Windows as an opportunity to steal time from your program, making the execution time of these functions appear slow and unpredictable.

Just how serious is the problem, and what can you do to minimize it? Under Windows 95/98/Me, there was nothing you could do to prevent Windows from stealing time - usually the delays were relatively short (less than 5 milliseconds), and occurred only a few times a second. Under Windows NT, these delays were much longer and unpredictable. Under Windows 2000 and XP, however, some changes to the Windows kernel and the addition of new real-time priority levels have allowed near-realtime programming while allowing graphics and Ethernet to work - just right for EyeLink applications.

8.3.1 Minimizing Windows Delays

To minimize windows delays:

- You should always ensure that no other time-critical programs, especially games, are running.
- Shut down all other applications (browser windows, chat clients, email programs, etc) prior to running an EyeLink experiment. These applications are listed in the taskbar at the bottom of the screen.
- Shut down any programs (Norton Antivirus, volume controller, Windows Messenger, etc) running in the notification area of the taskbar where you usually see the current time displayed (lower-right corner of the screen).
- Make sure no scheduled tasks (e.g., data backup, virus checking) are active.
- Remove unnecessary devices (e.g., DV converter, flash disk, external hard drive) connected through the USB or firewire ports.
- Shut down screen-saver management. Click the right mouse button at a blank space on the display PC desktop to open a dialog box for display properties settings. On the "Screen Saver" tab, set the screen saver to "None".
- Shut down power management. Select the "Screen Saver" tab of the "Display Properties" dialog box and click on the "Power ..." button. In the "Power Options Properties" dialog box, turn off all

features related to power management (hibernation, advanced power management support, turning off monitors or hard drives).

- For a computer with multiple Ethernet cards installed, use the Windows Control Panel to temporarily disable all network connections except for the one dedicated for EyeLink connection. The user should disable the firewall for the EyeLink Ethernet connection as well.

Even with no other programs running, the Windows kernel will try to steal some time about once a second for maintenance tasks: this cannot be shut off under Windows 95/98/Me, making these unsuitable for the kind of hard real-time graphics used in this version of the developer's kit. Because Windows 2000 and XP allow you to place your experimental application in a new level of real-time priority, graphics and the network continue to work while almost all other Windows tasks are disabled. This special mode is discussed next.

8.3.2 Windows 2000/XP Realtime Mode

Under Windows 2000 and Windows XP, it is possible place your application in a special level of real-time priority mode. This forces Windows to stop most other activity, including background disk access that might cause your experiment to have unpredictable delays. The sample experiment templates all use this mode in critical sections, including trials. Unfortunately, this mode does not work well under Windows 95, 98, and Windows Me, as these versions of Windows still allow the kernal and other applications to steal time even in real-time mode. This mode is also not useful in Windows NT 3.51 and 4.0, as it disables important parts of the system needed for graphics and the link.

The major problem with realtime mode is that certain system functions simply cease to work. You will not be able to play sounds, and the keyboard will not work properly. Under Windows XP, even the `escape_pressed()` and `break_pressed()` functions will not work (these are fine under Windows 2000, however, so it is possible a future release may fix this). DirectX functions works just fine under real-time mode, and ELSDK uses DirectX functions via SDL.

Using `getkey()` will probably return key presses in real-time mode, but it does this at the expense of allowing Windows to do background activity. This appears as unpredictable delays that can be as long as 20 milliseconds (but are usually on the order of 5-10 milliseconds). Therefore you should try to avoid using the Display PC keyboard for control or subject responses (keyboards are not very time accurate in any case). Instead, use `eyelink_last_button_press()` to detect tracker button responses, `break_pressed()` to detect program shutdown, `eyelink_is_connected()` to detect tracker disconnection, `check_recording()` to detect recording aborted by the eye tracker, and, if required, `eyelink_read_keybutton()` to monitor the EyeLink tracker keyboard.

8.4 Hardware I/O under Windows

Windows NT and its successors (Windows 2000 and XP) do not allow your programs to access I/O ports directly (for example to read button boxes or to send digital commands to external devices), and I/O access was slow under Windows 95/98/Me.

We have included a freeware hardware I/O port driver (the DriverLink PortIO package) with the EyeLink developer's kit to allow you to access I/O ports quickly and without any special programming techniques. This package must be installed on your computer before running any of the applications, and Windows restarted to complete installation of the driver. The driver DLL will be placed in the Windows directory by the installation, and the library is linked into the *ELSDK* DLLs. To use the I/O port functions, simply include the *nt_portio.h* header file in your source code - this includes the DriverLinx function definitions, and also redefines the usual C hardware I/O port access functions `_inp()`, `outp()`, `_inpw()`, `_outpw()`, `_inpd()` and `_outpd()` to use the DriverLinx functions instead.

8.5 Message Pumps and Loops

An experiment needs to be *deterministic:* it must produce events in a sequence determined by randomization, with accurate timing. This requires that the program always has control over the computer, executing code and loops without returning control to Windows. However, if you try to run programs like this under Windows, many problems can occur: windows aren't displayed, keys can't be read, and so on.

We can make Windows work with loops by calling a *message pump* often. This is similar to the standard core code found in regular C programs, which calls `GetMessage()`, `TranslateMessage()` and `DispatchMessage()`. This serves to pass messages from the Windows kernel on to other parts of your experiment. Even Microsoft Foundation Classes (MFC) and Visual Basic application have message pumps: they are just hidden inside the run-time libraries (and are therefore harder to avoid executing).

In general, it's not a good idea to call message pumps during time-critical parts of your code. Some messages require a lot of processing by other applications, or may even redraw parts of the display. Worst of all, Windows will take whatever time it needs during calls to this function, performing disk activity and so on. Under Windows 95/98/Me, Windows had its own message pump that it would use if you didn't, so these delays were unavoidable. Most debuggers (including the one in Visual Studio) also seem to run their own message loop, making real-time performance worse when debugging. The lack of this internal message loop allows much better real-time performance under Windows 2000 and XP - at the cost of much lower responsiveness if you don't call a message pump occasionally. Even so, you don't need to call a message pump except to read keys once the full-screen experiment window has been created.

The *eyelink_core* library supplies several message-pump functions. The simplest way to use a message pump is to call `getkey()` to check if any keys have been pressed: this will also call the message pump each time. You can call a message pump explicitly with `message_pump()`, which also allows you to process messages for a modeless dialog box (such as the progress display window in the EDF file transfer function).

To implement long delays, you should call `pump_delay()` rather than `msec_delay()`. This will allow the operating system to process messages while waiting.

It is possible to create loops that do not contain a message pump. In this case, you must call `break_-pressed()` in the loop to check if the program should terminate. You may also want to call `escape_-pressed()` to see if the ESC key is pressed, which can be used to interrupt trials. Note that these functions do not appear to work very well under Windows XP: they appear to be disabled in real-time mode, and may take several seconds to detect a key if you are drawing graphics often, even when not in real-time mode. These functions work very well in Windows 2000, however.

8.6 Windows Key Support

The preferred method to read keys in sections that are not time-critical is with `getkey()`, which also acts as a message pump and returns any keys that were recently pressed. This function returns `TERMINATE_-KEY` if CTRL-C is pressed, or if the experiment has been terminated by pressing ALT-F4: if this code is returned, exit from any loop immediately. Some non-character keys (such as the cursor keys) are translated into the key codes required by the EyeLink tracker: these are defined in *eyelink.h*.

In some cases, such as during subject setup, you will want to echo the Subject PC keyboard to the tracker for remote control. Use the function `echo_key()`, which is used like `getkey()` but also sends a copy of each key to the eye tracker through the link.

8.7 Terminating the Program

When a Windows program is terminated by ALT-F4, it receives WM_QUIT and WM_DESTROY system messages and its window is closed. This normally would exit the message pump in the `main()` function, but there is no simple message pump in our experiments. Instead, your code should detect program shutdown by one (or all) of the following: check `break_pressed()` for a nonzero return value, check if `getkey()` returns TERMINATE_KEY, or check if `eyelink_is_connected()` returns 0. If any of these happens, your functions must break out of any loops they are executing and return immediately. Don't use the `getkey()` test in time critical code, as this would cause delays. The following code is an example of pulling all of these tests into one loop:

```
while (1)  // our loop
{
        unsigned key = getkey();
        if( key == TERMINATE_KEY ) break; // check for program termination
        if( break_pressed() ) break;      // an alternative way to check termination
        if( escape_pressed() ) break;     // optional test for ESC key held down
        if( ! eyelink_is_connected() ) break;  // exit if EyeLink tracker stopped

        // YOUR LOOP CODE GOES HERE

}
```

The method outlined above should be used in EyeLink experiments for Windows instead of those used in DOS. You cannot use the standard C call `exit()` within a Windows program. You should not use `setjmp()` and `longjmp()` to transfer control back to the `main()` function, as this won't clean up Windows resources and close windows.

It is also possible to force your program to terminate, as if ALT-F4 was pressed. By calling `terminal_break(1)`, the `break_pressed()` and `getkey()` functions will behave as if the program was terminated. If the *eyelink_core* library is performing setup or drift correction, it will break out of these functions and return to your code immediately if `terminal_break()` with parameter 1 or `exit_calibration()` is called.

Chapter 9

Graphics Programming using SDL

The *eyelink_core_graphics* library uses SDL to display graphics on to the display PC. Simple Direct-Media Layer (SDL) is a cross-platform multimedia library designed to provide level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer. The SDL supports multiple platforms including Windows, Linux, MacOS 9, and MacOS X. It tries its best to get the best performance out of the given computer. For example, SDL uses page flipping and hardware accelerated blit, color keying, and alpha blending using DirectX on windows, using DGA on X graphics environments and so on (Visit `http://www.libsdl.org` for more details on SDL and for the SDL API documentations). The SDL library bundled with ELSDK is slightly modified from the one at `http://www.libsdl.org/download-1.2.php` . The changes involve the addition of following functions:

1. `SDL_SetVideoModeEx()`
2. `SDL_FlipEx()`
3. `Flip()` - macro provided in sdl_expt.h

Currently, these functions are only available on windows with DirectX and X environments with DGA only. The `SDL_SetVideoModeEx()` a slight modification of `SDL_SetVideoMode()`. With `SDL_SetVideoModeEx()` one can set the refresh rate that they want. In the `SDL_SetVideoMode()`, SDL chooses one of the available refresh rates. The function `SDL_Flip()`, if used with hardware flips, setup the flip and returns immediately. The real flip occurs on the next retrace. However, we need to know when the stimulus is really displayed to the subject. Thus, we need to wait till the real flip occurs. `SDL_FlipEx()` does exactly that. That is, it does not return until the real flip occurs. The Flip macro can also be used instead of `SDL_FlipEx()`. The macro loops through and checks for the `SDL_Flip()` return value. `SDL_Flip()` returns -1 if it cannot schedules a flip.

`SDL_SetVideoModeEx()` function need not be called for most experiments. The *eyelink_core_graphics* library provides `init_expt_graphics()`, which takes and SDL_Surface and DISPLAYINFO. If SDL_Surface parameter is NULL, `init_expt_graphics()` sets the video mode. If DISPLAYINFO parameter is NULL, `init_expt_graphics()` sets the video mode to the current display settings. One can override this by filling in the width, height, depth, refresh attributes of DISPLAYINFO and passing in to `init_expt_graphics()`. One can also set their video mode by calling `SDL_SetVideoMode()` or `SDL_SetVideoModeEx()` and pass in the SDL_Surface.

The modified SDL only works on DirectX enabled environments in windows. That is, it does not support GDI driver for SDL. So, make sure you have proper video driver and DirectX 7 or higher installed in your environment. Generic video drivers such as VGA, and SVGA are not supported. Make sure your setting is similar to the one given below.

9.1 Resolutions and Colors

Graphics modes vary in resolution, number of colors available, and refresh rates. The most useful resolutions and color modes are listed below:

640 by 480	Lowest common resolution, useful for pictures, simple graphics and single words printed in large text. Refresh rates over 160 Hz are usually only available in this mode.
800 by 600	Moderate resolution, useful for pictures and pages of large text Some monitors will support 160 Hz refresh rates or better in this mode.
1024 by 768	High resolution, useful for pages of smaller text. Refresh rates over 120 Hz not usually available.
256 colors	Indexed color modes. You need to use palettes to draw graphics in this mode, which the templates do not currently do. This mode is most useful for simple graphics and text. Loaded images do not display well in this mode.
Hi color (15 or 15 bits)	Displays real colors, but may show contour artifacts to images that have smooth transitions, as it only supports 32 levels of brightness.
Hi color (16 bits)	Same as 15 bit colors, except 1 bit (2 levels) alpha color blending is supported.
True color (24 bits)	Displays real colors, but may draw more slowly. Supports 256 levels of brightness (may actually be 64 levels on some video cards)
True color (32 bits)	Same as 24 bit color, except 8 bit (255 levels) alpha color blending is supported.

9.2 Drawing Speed

Drawing speed depends on the selected resolution, colors and the supported hardware. In general, more colors, higher resolution and faster refresh rates all result in slower graphics drawing. For experiments that follow the templates, the time to copy a bitmap to the display is the most critical factor in selecting a mode. This should be tested for your card, by measuring the time before and after a bitmap copy. Ideally, the copy should take less than one refresh period so that the drawing is not visible, and so the subject sees the stimulus all at once.

9.3 Display Mode Information

As part of initializing your experiment, you should check that the current display mode is appropriate to your experiment. For example, a fast refresh rate may be required, or a resolution that matches that of a picture may be needed. An experiment should always be run in the same display mode for all subjects, to prevent small differences in appearance or readability of text which could affect subject performance.

Information on the current display mode can be measured by calling `get_display_information()`. This fills a `DISPLAYINFO` structure with the display resolution, colors, and the refresh rate. If you use the global `DISPLAYINFO` structure `dispinfo`, then the macros SCRWIDTH and SCRHEIGHT can be used to compute the screen width and height in pixels.

This is the definition of the `DISPLAYINFO` structure:

```
typedef struct {
                 INT32 left;      // left of display
                 INT32 top;       // top of display
                 INT32 right;     // right of display
                 INT32 bottom;    // bottom of display
                 INT32 width;     // width of display
                 INT32 height;    // height of display
                 INT32 bits;      // bits per pixel
                 INT32 palsize;   // total entries in palette (0 if not indexed)
                 INT32 palrsvd;   // number of static entries in palette
                 INT32 pages;     // pages supported
                 float refresh;   // refresh rate in Hz (<40 if refresh sync not available)
                 INT32 winnt;     // Windows: 0=9x/Me, 1=NT, 2=2000, 3=XP
                } DISPLAYINFO;
```

Several fields in this structure require further explanation. The `palsize` field will be 0 if in 32, 24 or 16-bit color modes, which are required for image file display. If this is nonzero, you are in 256-color mode (or even 16-color) mode, which is only useful for drawing simple graphics. The `pages` field is always 1, as multiple pages are not supported. The refresh field is the measured display refresh rate, which may differ from that reported by the operating system.

Finally, the `winnt` field indicates under what version of Windows the application is running. This can distinguish between Windows 95/98/Me (where realtime mode is ineffective), Windows NT (for which realtime mode will stop the network from working), Windows 2000 (realtime mode works and does the keyboard), and Windows XP (realtime mode works but disables the keyboard).

9.4 Adapting to Display Resolutions

When the subject-to-display distance is twice the display width (the recommended distance for EyeLink operation), then the display will be 30° wide and 22.5° high. This allows you to compute pixel sizes of objects in degrees, as a fraction of display width and height. The templates also use this method to adapt to different display resolutions. Complete independence of display resolution is not possible, as fonts and small objects (i.e. the calibration targets) will look different at low resolutions.

9.5 Synchronization with Display Refresh

The use of `SDL_Flip()` and `SDL_FlipEx()` on the suggested environments will always make your drawing synchronized with display refresh.

9.6 Full-Screen Window

SDL only supports hardware assisted flips and blits on full screen window only. If you are setting your video mode, make sure you initialize your video mode with full screen window. If you are just calling `init_expt_graphics()`, it always creates full screen window. To get the handle to the window, just call `SDL_GetVideoSurface()` after calling `init_expt_graphics()`.

Chapter 10

Graphics Programming using GDI

This section is intended for programmers who have at least some understanding of Windows programming. If you need an introduction, read the on-line help files for C++, about the Windows GDI. An excellent book for an introduction is "Programming Windows", Fifth Edition: by Charles Petzold, published by Microsoft Press (1998): chapters 4, 5, 14, and 17 are the most useful for programmers using this development kit. Although old, this book is still a good introduction for GDI programming in C under any version of Windows. The examples given in Experiment Templates Overview are intented for SDL programming.

The EyeLink Software Development kit (ELSDK) supplies several functions that simplify GDI programming and creating stimulus displays. As well, there are several C files used extensively in the templates that provide easy-to-use support for fonts, bitmaps, and images in your experiments. Where appropriate, these functions will be discussed in this section. This is a list of the C files that directly support graphics: you may wish to read through the code in these, or copy sections for use in your own graphics routines (don't modify the original files-keep them as a reference, and create renamed versions to modify).

w32_demo_window.c	Typical window creation and support functions
w32_text_support.c	Font creation and printf()-like text output
w32_bitmap_sppt.c	Bitmap creation and display functions
w32_text_bitmap.c	Formatted text pages; create text page bitmap
w32_freeimage_bitmap.c	Load many types of image file to a bitmap

10.1 Displays and Experiments

For each experiment, the programmers need to decide how to properly display stimuli. It may seem straightforward to present stimuli by simply drawing graphics directly to the display, by using Windows GDI drawing functions. However, the finite time required to draw the display and the progressive transfer of the image to the monitor (refresh) can cause visible display changes. It is also important to accurately mark the moment the stimulus appears to the subject for reaction-time measurement. Careful attention to timing is required in order to produce reliable experimental results. Two methods of presenting stimuli (drawing directly to the display, and drawing to a bitmap which is then copied to the display) and the appropriateness of each are discussed below.

10.1.1 Drawing Directly to the Display

For small graphics such as a few words, small pictures, or pattern masks, drawing times may be only a few milliseconds. If drawing times are longer than this (for example loading large pictures from disk or drawing large screens of text) the progress of drawing will be clearly visible to the subject. This will also make measuring reaction times problematic.

It is up to the experimenter to determine what drawing speed is acceptable. Drawing time can be measured using the EyeLink millisecond clock, by reading the time before and after the drawing operation, or by sending messages to the EDF file before and after drawing. The sample experiments here use a method of marking stimulus onset that also reports the time required for drawing. While drawing times can be unpredictable under Windows for certain operations, acceptable drawing operations and techniques to obtain fast drawing times are discussed throughout this document.

10.1.2 Synchronizing to the Display

A more serious problem is determining exactly when the stimulus became visible to the subject, for reaction time measures. Even if drawing is fast, the image displayed on the display monitor is repainted gradually from top to bottom, taking about 80% of a refresh period (1000 msec/refresh rate). For example, it will take about 13 millisecond to completely repaint a display at 60 Hz refresh rate. This repainting occurs every 16.7 milliseconds for a 60 Hz display, which means that drawing changes will not become visible for an additional 1 to 16 milliseconds.

The exact time the display is modified can be recorded by sending a timestamped message to the eye tracker when drawing is completed. Some of the uncertainty caused by the delay between drawing and display on the monitor may be removed by waiting for the start of the monitorŠs refresh before drawing. The EyeLink Software Development kit (ELSDK) supplies the wait_for_video_refresh() function, which wait until the start of the vertical retrace (0.5 to 1/5 milliseconds before the top of the monitor begins to be updated). For drawing operations that take a few milliseconds (about 0.5 ms at the top of the display, about 300ms/(refresh rate) at the center of the display), the stimulus will become visible in the same refresh cycle it is being drawn in.

To accurately mark the time of onset, we need to place a display onset message in the EDF file. The method we will use in our examples is fast, accurate, and can be used with any drawing method that is sufficiently fast (including bitmap display, as we will see next). This technique requires the following steps:

- Wait for end of refresh, using the wait_for_video_refresh()
- Read the time using current_msec()
- Draw the stimulus (must be fast enough to appear in same refresh!)
- Read current_msec() again, subtract previous value. This computes the delay from the retrace, which is also the time it took to draw the stimulus
- Write a message with the delay from retrace followed by the keyword "DISPLAY ON". New EyeLink analysis tools will automatically subtract this number from the message timestamp to get the exact time of the retrace. The message "SYNCTIME" followed by the computed delay from retrace may also be included, for compatibility with the earlier analysis software.

Note that we did not send the message immediately after wait_for_video_refresh() returned. It may take a significant amount of time (0.2 milliseconds or more) to send a message, and delay between retrace and drawing could cause delayed presentation or flickering of moving stimuli at the top of the display.

Remarks:

Preceding the message with the delay of the message from retrace will allow new EyeLink analysis tools to automatically correct the time of the message. If a message begins with a number, this number will be subtracted from the time of the message, and the second item in the message is used to identify the message type.

This reporting method can also be used for non-synchronized displays, either by omitting the numbers (analysis tools will assume the stimulus became visible just before the message) or be using a best-guess value for the onset-to-message delay.

10.1.3 Using Bitmaps for Display

The best method in Windows for rapidly displaying complex graphics at the start of the trial is to use a device-dependent bitmap. This is like an invisible display stored in memory, which you can draw to using any of the Windows GDI drawing commands. Most of the GDI sample experiments included in the developerŠs kit (except for the "simple" experiment) use bitmaps for display. The mechanics of using bitmaps will be discussed in the section on Windows graphics programming.

Once the stimulus has been drawn to the bitmap, it can be rapidly copied to the display. For most video cards and display resolutions, this can be done in much less than one refresh period (1000ms/refresh rate). Even if the copy takes more than one refresh period, the results are better than drawing directly to the display. When the start of the bitmap drawing is synchronized with the vertical retrace, the drawing will out-race the progressive updating of the monitor display, and result in the bitmap appearing in the same refresh sweep it was drawn in. To achieve this, a full-screen bitmap copy needs to take less than half of a refresh period (actually about 70%, but a safety margin must be added). The fastest copy rates can be achieved by high-end AGP video card and fast (1 GHz or higher) CPUs and motherboards.

10.2 Graphics Modes

Windows supports many graphics modes, depending on your VGA card and driver. Using the "Display" item in the Windows Control Panel, you can explore and select the various modes available. Many video card drivers also supply a display-mode icon that is available at the right end of the desktop toolbar.

10.2.1 Resolutions and Colors

Graphics modes vary in resolution, number of colors available, and refresh rates. The most useful resolutions and color modes are listed below:

640x480	Lowest common resolution, useful for pictures, simple graphics and single words printed in large text. Refresh rates over 160 Hz are usually only available in this mode.
800x600	Moderate resolution, useful for pictures and pages of large text. Some monitors will support 160 Hz refresh rates in this mode.
1024x768	High resolution, useful for pages of smaller text. Refresh rates over 120 Hz are not usually available

256 colors	Indexed color modes. You need to use palettes to draw graphics in this mode, which the templates do not currently do. This mode is most useful for simple graphics and text. Loaded images do not display well in this mode.
Hi color (15 or 15 bits)	Displays real colors, but may show contour artifacts to images that have smooth transitions, as it only supports 32 levels of brightness.
True color (24 bits)	Displays real colors, but may draw more slowly. Supports 256 levels of brightness (may actually be 64 levels on some video cards)

10.2.2 Drawing Speed

Drawing speed depends on the selected resolution and colors. In general, more colors, higher resolution and faster refresh rates all result in slower graphics drawing. For experiments that follow the templates, the time to copy a bitmap to the display is the most critical factor in selecting a mode. This should be tested for your card, by measuring the time before and after a bitmap copy. Ideally, the copy should take less than one refresh period so that the drawing is not visible, and so the subject sees the stimulus all at once.

An important difference between Windows 2000/XP and DOS or Windows 95/98/Me is that graphics are not usually drawn immediately-instead, they are "batched" and drawn up to 16 milliseconds later. There are several ways to force Windows 2000 and XP to draw graphics immediately. The first is to call GdiFlush(), which forces any pending drawing operations to be performed. Batching can also be turned off by calling GdiSetBatchLimit(1), which turns off batching entirely (this has already been done for you in w32_demo_window.c). There are a few cases where batching might be useful, for example if a lot of small drawing operations need to be done together, or if a moving object needs to be erased and then redrawn at a different position as quickly as possible-batching could be enabled before drawing and disabled afterwards, as is done in w32_gcwindow.c for moving a gaze-contingent window.

Finally, many display drawing operations under Windows (especially copying bitmaps to the display) are done in hardware by the display card and the drawing function calls return immediately, long before the actual drawing is finished. While this can be an advantage if other non-drawing operations need to be done, this does make timing displays difficult. The function wait_for_drawing(HWND hwnd) is supplied for you, which both forces immediate drawing and does not return until all ongoing drawing is actually finished. The argument hwnd is a window handle, which can be either full_screen_window for the window created by w32_demo_window.c, or can be NULL to check for drawing operation anywhere on the display (including a second monitor if this is installed).

```
void CALLTYPE wait_for_drawing(HWND hwnd)
{
  HDC hdc;
  RECT rc;

  hdc = GetDC(validate_window(hwnd));
  if(IsWindow(hwnd))
    GetWindowRect(validate_window(hwnd), &rc);
  else
    {
      rc.top = dispinfo.top;
      rc.left = dispinfo.left;
    }

  GdiFlush();
  GetPixel(hdc, rc.top, rc.left);
  ReleaseDC(validate_window(hwnd), hdc);
}
```

The code for this function is very simple: it calls GdiFlush(), then calls GetPixel() to read a single pixel from the display. Because Windows cannot determine what color a pixel will be until drawing is finished, this read does not return until drawing is completed. As you can see, most of the code is involved with preparing for the call to GetPixel(), so if you are writing your own graphics code it may be more efficient to simply call GetPixel() following your own graphics code.

10.2.3 Display Mode Information

As part of initializing your experiment, you should check that the current display mode is appropriate to your experiment. For example, a fast refresh rate may be required, or a resolution that matches that of a picture may be needed. An experiment should always be run in the same display mode for all subjects, to prevent small differences in appearance or readability of text which could affect subject performance.

Information on the current display mode can be measured by calling get_display_information(). This fills a DISPLAYINFO structure with the display resolution, colors, and the refresh rate. If you use the global DISPLAYINFO structure dispinfo, then the macros SCRWIDTH and SCRHEIGHT can be used to compute the screen width and height in pixels.

This is the definition of the DISPLAYINFO structure:

```
  typedef struct {
     INT32 left;       // left of display
     INT32 top;        // top of display
     INT32 right;      // right of display
     INT32 bottom;     // bottom of display
     INT32 width;      // width of display
     INT32 height;     // height of display
     INT32 bits;       // bits per pixel
     INT32 palsize;    // total entries in palette (0 if not indexed)
     INT32 palrsvd;    // number of static entries in palette
     INT32 pages;      // pages supported
     float refresh;    // refresh rate in Hz (<40 if refresh sync not available)
     INT32 winnt;      // Windows: 0=9x/Me, 1=NT, 2=2000, 3=XP
  } DISPLAYINFO;
```

Several fields in this structure require further explanation. The palsize field will be 0 if in 32, 24 or 16-bit color modes, which are required for image file display. If this is nonzero, you are in 256-color mode (or even 16-color) mode, which is only useful for drawing simple graphics. The pages field is always 1, as multiple pages are not supported. The refresh field is the measured display refresh rate, which may differ from that reported by the operating system. If this is less than 40, the wait_for_video_refresh() function is not working (probably due to an incorrect implementation of VGA registers on the video card). In this case, the system's refresh rate can be retrieved with the following code:

```
      refresh = (float)GetDeviceCaps(NULL,VREFRESH);
```

Finally, the winnt field indicates what version of Windows the application is running under. This can distinguish between Windows 95/98/Me (where realtime mode is ineffective), Windows NT (for which realtime mode will stop the network from working), Windows 2000 (realtime mode works and does the keyboard), and Windows XP (realtime mode works but disables the keyboard).

10.2.4 Adapting to Display Resolutions

When the subject-to-display distance is twice the display width (the recommended distance for EyeLink operation), then the display will be 30° wide and 22.5° high. This allows you to compute pixel sizes of objects in degrees, as a fraction of display width and height. The templates also use this method to adapt to different display resolutions. Complete independence of display resolution is not possible, as fonts and small objects (i.e. the calibration targets) will look different at low resolutions.

10.2.5 Synchronization to Display Refresh

The image on the phosphor of the display monitor is redrawn from the video memory, proceeding from top to bottom in about (800/refresh rate) milliseconds. This causes changes in graphics to appear only at fixed intervals, when the display refresh process transfers that part of the screen to the monitor. Graphics at the top of the screen will appear about 0.5-2 millisecond after refresh begins, and those at the bottom will appear later.

The simplest way to determine just when graphics will be displayed to the subject is to pause before drawing until the refresh of the display begins, using this function wait_for_video_refresh(). This function returns only after video retrace begins; if it is called while video retrace is active, it will continue to wait for a full retrace period, until the retrace begins again. This is designed to provide the maximum time for drawing to the display. If you just want to check to see if retrace is active, call in_vertical_retrace() instead. However, be sure to allow enough time between calls to this function, as vertical retrace can be active for up to 4 milliseconds, and this could be interpreted as multiple video refresh intervals passing.

In summary, if graphics can be drawn quickly enough after the display retrace, they will become visible at a known delay that depends on their vertical position on the monitor. Placing a message in the EDF file after drawing which includes the delay from the vertical refresh allows applications to precisely calculate the onset time of a stimulus.

10.3 Full-Screen Window

Every Windows program must have a window. The module w32_demo_window.c implements the minimum functionality needed for supporting the full-screen window needed for experiments. It includes functions to create and destroy the window, and to clear it to any color. Also included is a function to process messages for the window.

10.3.1 Creating the Window

The function make_full_screen_window() creates the experiment window. It first registers a window class, which specifies the defaults for our window. It then creates a new window with no border and sized to fill the entire screen. It then makes the window visible, and waits for it to be drawn. Since Windows sometimes erases other windows before drawing our window for the first time, we can't draw to our window until it's seen its first WM_PAINT message, or our graphics could be erased by the redrawing of the window. The code in this function is very critical, and should not be changed. After creating the window, it pauses for 500 milliseconds while running the message pump, which allows Windows to remove the taskbar from the desktop.

This functions also calls GdiSetBatchLimit(1), which forces graphics to be drawn immediately by Windows, instead of being deferred until a later time. This does not guarantee that some drawing operations (especially bitmap copies) will be finished before a drawing function returns-use wait_for_drawing() after drawing to ensure this.

10.3.2 Clearing the Display

The window (and the whole display) can be cleared by calling clear_full_screen_window() and specifying a color. Windows colors are specified by the red, green, and blue components, using the RGB() macro to combine these into a COLORREF number.

10.3.3 Processing Windows Messages

All Windows messages for the window are handled by full_screen_window_proc(). This handles key press messages by passing them to the ELSDK library function process_key_messages(), which passes the keystrokes to any library functions that may be running, such as do_tracker_setup(), or saves them for later retrieval by getkey(). It also intercepts messages that Windows send to close the window, and calls terminal_break(1) to inform your experiment. It makes the mouse cursor invisible when only our window is visible, so it doesn't interfere with the experiment displays. Finally, it clears parts of the window to target_background_color which were covered by dialog boxes or other windows. Any display information in these areas is lost, so you may want to add redrawing calls here, or save the entire screen to a bitmap before calling up a dialog box, then restore it afterwards.

10.3.4 Changing the Window Template

Usually, there is little reason to change the code in w32_demo_window.c, unless you are adding functionality to the window. For example, you might add palette support when clearing the window. The handling of messages can be changed, for example to make the mouse cursor visible.

10.3.5 Registering the Window

Your full-screen window must be registered with the eyelink_gdi_graphics library in order to perform calibration, drift correction, and to display camera images. This is done by calling init_expt_graphics(). This allows ELSDK to be used with languages such as Visual Basic, that must create their own windows. During calls to do_tracker_setup() or do_drift_correct(), ELSDK will intercept some messages to your window, and will draw calibration targets and camera images into it. You should call close_expt_graphics() to unregister the window before closing it.

10.4 Windows Graphics Fundamentals

The great advantage to creating graphics in Windows is that there are powerful drawing tools such as TrueType fonts, patterns, pens and brushes available, and these work in any display mode without changes. However, this power comes at the cost of adding setup code before drawing, and cleanup code afterwards. Also, the Windows programming interface for some graphics operations can be rather complex, so you'll have to become familiar with the help files (or Visual Studio InfoViewer topics) for the Windows API.

The following discussion only covers programming issues that need to be considered for experiments. You should study the Windows API documentation that came with Visual Studio on the GDI for more general information, or consult a good book on the subject for more information.

10.4.1 Display Contexts

Drawing in Windows must always be performed using a display context (DC). This allows drawing to the full screen, a window, or a bitmap. Usually display contexts are allocated as needed and released immediately with GetDC(full_screen_window) and ReleaseDC(full_screen_window).

Each time you allocate a DC, you will have to select new objects such as brushes, pens, palettes and fonts into the DC, then reselect the original objects before releasing the DC. You could also select stock objects (such as white brushes, black pens, and the system font) instead of restoring the old objects.

When you allocate a DC, you need to supply a window handle (HWND). If this is NULL (0) you will get a DC that allows writing to the whole screen. It's better to use the full-screen window handle (usually full_screen_window) that your experiment created, however.

10.4.2 Specifying Colors

Windows colors are specified by the red, green, and blue components, using the RGB() macro to combine these into a COLORREF number. By combining this value with 0x02000000, (for example, RGB(0,255,0)|0x02000000) the color will be drawn as a solid shade. Otherwise, colors might be drawn using a dot pattern in 256-color modes.

10.4.3 Pens and Brushes

The drawing of lines is controlled by the currently selected pen. Filling in shapes is controlled by the currently selected brush. Both pens and brushes can be created in any color, or made invisible by selecting the stock NULL_PEN or NULL_BRUSH.

You must remember to remove pens and brushes from the DC before disposing of it. You can do this by reselecting the original pens or brush, or by selecting a stock pen or brush. Created pens or brushes must be deleted after being deselected. For an example, see the file w32_playback_trial.c from the eyedata sample experiment.

10.4.4 Fonts and Text

Windows has many options for selecting and using fonts. The most important text requirements for experiments are selecting standard fonts, printing of short text messages and words, and drawing of pages of text for reading trials and instruction screens. These are implemented in the template support modules w32_text_support.c and w32_text_bitmap.c.

Fonts are created by calling get_new_font(), supplying a font name, its size (height of a line in pixels), and whether the font should be boldface. The font created is stored in the global variable current_font, and is not deleted until a new font is selected. The most useful standard font names are:

Figure 10.1: A simple, proportionally spaced font.

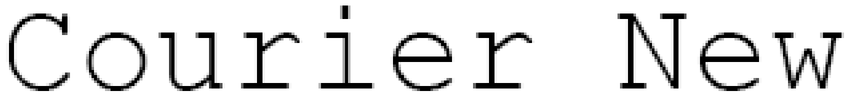

Figure 10.2: Monospaced font. Rather wide characters, so less will fit per line.

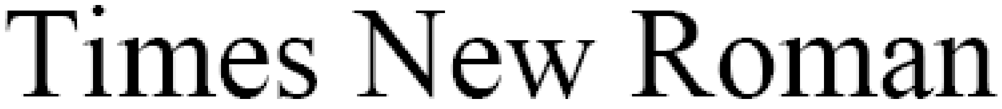

Figure 10.3: A proportionally spaced font, optimal for reading.

Fonts can be drawn with directly by selecting current_font into a display context, and released after drawing by selecting the system font, as in this example:

```
if(current_font) SelectObject(hdc, current_font);
SetTextColor(hdc, text_color | 0x02000000L);
TextOut(hdc, x, y, text, strlen(text));
SelectObject(hdc, GetStockObject(SYSTEM_FONT));
```

Drawing of fonts may be slower when a character is drawn for the first time in a new font. You can increase drawing speed by first drawing the text invisibly (in the background color), which will cause windows to create and cache a bitmap for each character. These cached characters will be used when you redraw the font in the foreground color.

When there is a need to print a few characters or a line of text, the function graphic_printf() can be used. You can specify a foreground and an optional background color, a position for the text, and whether to center it. The text is generated using a format string similar to the C printf() function.

Multi-line pages of text can be printed using draw_text_box(), supplying the margins and a line spacing in pixels. Optionally, boxes will be drawn at the position of each word on the EyeLink tracker's display, to serve as a reference for the gaze cursor during recording. You need to supply a display context to this function, which allows this function to draw to either the display or a bitmap. A bitmap containing a page of text can be created and drawn in one step with text_bitmap().

10.5 Drawing With Bitmaps

Drawing graphics takes time, usually more than one display refresh period unless the graphics are very simple. The progressive drawing of complex stimuli directly to the display will almost certainly be visible. This is distracting, and makes it difficult to determine reaction-time latencies. One method to make the display appear more rapidly is to draw the graphics to a bitmap (a memory buffer), then to copy the bitmap to the display. Bitmap copies in Windows are highly optimized, and for most video cards and modes the entire display can be updated in one refresh period.

The type of bitmaps that need to be used for drawing are device-dependent bitmaps (DDB). The Windows documentation is full of routines that support device-independent bitmaps (DIBs), but these are actually only used for loading and saving resources, and for copying images on the clipboard. Don't use any DIB functions with bitmaps: they will not work reliably, and are often slower.

10.5.1 Drawing to Bitmaps

The C source file w32_bitmap_sppt.c (used in the picture and text templates) contains functions to copy bitmaps to the display, and blank_bitmap(), a simple function to create and clear a bitmap that is the same size as the display. Its code is used in the discussion below.

Before accessing a bitmap, we have to create a memory device context (MDC) to draw in. We first get a DC to the display, then create a memory device context compatible with the display DC. We can then create a bitmap, in this case of the same dimensions as the display. We could also make a smaller bitmap, which would save memory when several bitmaps are required.

Then, the bitmap is selected into the MDC. We save the handle of whatever bitmap was originally selected into the DC, to use when deselecting our bitmap later.

```
hdc = GetDC(NULL);
mdc = CreateCompatibleDC(hdc);     // create display-compatible memory context
hbm = CreateCompatibleBitmap(hdc, SCRWIDTH, SCRHEIGHT);
obm = SelectObject(mdc, hbm);      // create DDB bitmap, select into context
```

Now, we can draw it to the bitmap using the same GDI functions as would be used to draw to the display. In this case, we simply clear the bitmap to a solid color:

```
oBrush = SelectObject(mdc, CreateSolidBrush(bgcolor | 0x02000000L));
PatBlt(mdc, 0, 0, SCRWIDTH, SCRHEIGHT, PATCOPY);
DeleteObject(SelectObject(mdc, oBrush));
```

Finally, we select the old bitmap to release our bitmap, and clean up by deleting the MDC and releasing the DC:

```
SelectBitmap(mdc, obm);
DeleteDC(mdc);
ReleaseDC(NULL, hdc);
```

We now have our bitmap, which we can draw in by creating an MDC and selecting it. Remember to delete the bitmap with DeleteObject() when it is no longer needed.

Other functions in the template modules that draw to bitmaps are text_bitmap() in w32_text_sppt.c, draw_-grid_to_bitmap() in w32_grid_bitmap.c, and bmp_file_bitmap() in w32_bmp_bitmap.c.

10.5.2 Loading Pictures

The function image_file_bitmap() defined in w32_bmp_bitmap.c loads an image file from disk, and creates a bitmap from it. It can also resize the image to fill the display if required. It uses the freeware "FreeImage" library for this purpose (www.6ixsoft.com). This loads many picture formats, including BMP, PCX and JPG (but not GIF). JPG files are smallest, but load more slowly and may have artifacts with sharp-edged images such as text. Creating a 256-color PCX file is much more efficient for text or simple line drawings. This library always loads images as an RGB bitmap, so any palette information in the original image is lost. This means that even 2-color images will be displayed in garish, saturated colors in 256-color modes, and programs that use loaded images should always run in 16, 24, or 32-bit color display modes.

The arguments to this function are image_file_bitmap(fname, keepsize, dx, dy, bgcolor). This function creates and returns a bitmap (NULL if an error occurred) that is either the size of the image (keepsize=1) or full-screen sized. If a full-screen bitmap is created, the bitmap is cleared to bgcolor and the image in copied to the center of the bitmap, resized to (dx, dy), or left its original size if dx is 0. Resizing is done using StretchBlt(), which can be rather slow, and the results will be poor if the image contains text or other sharp detail. It is best to create images of the same dimensions as the display mode you will use for our experiment.

10.5.3 Copying to the Display

Copying the bitmap to the display is similar to drawing: create a memory device context for the bitmap, get a DC for the display, and use BitBlt() to do the copying. It's a good idea to call GdiFlush() to ensure that drawing starts immediately. The Windows GDI passes information to the VGA card's driver, which usually does the copying in hardware. This means the call to BitBlt() may return long before the drawing is done-if this is not desirable, call wait_for_drawing() or call GetPixel() for a point on the display. Both functions will return only once the drawing is completed.

The w32_bitmap_sppt.c source code file is used in most of the sample experiment templates, and has two functions to copy bitmaps to the display. To copy an entire bitmap to the display, call display_bitmap(), specifying the position at which to place the top left corner of the bitmap. A rectangular section of a bitmap can be copied using the display_rect_bitmap() function.

All the source code examples create a full-screen bitmap to the display. It is obviously faster to create smaller bitmaps and copy these to the center of the display instead, but this should only be needed for slow video cards. For example, the margins of a page of text usually are 2° from the edges of the display. Copying only the area of the bitmaps within the margins will reduce the copying time by 40%.

Chapter 11

Eyelink Graphics Programming Using External Graphics Library

The plug-in design of new graphic component in the *eyelink_core* makes it possible for programmers to take advantage of any graphics environment that they have to use. Suppose we have a complex three-dimensional graphics to display to the subject. SDL may not be the greatest platform to perform such graphics. In this case, the user is not forced to use SDL as their development platform. ELSDK allows the user to create their own graphics development platform with their choice of graphics library. In addition, one can override an existing library's callback function to overcome a bug or to suit their needs.

If the user decides to implement his own graphics environment, the following needs to be done to set up the display environments for graphic operations, such as calibration, validation, and display of the eye image.

11.1 Roles of EyeLink Graphics library

There are three things that an EyeLink graphics library should do.

1. Display calibration, validation, drift correct, and camera image screens on the display pc.
2. Save display screens to disk for future analysis. This may be optional.
3. Provide feedback about any input given by the user.

To achieve the duties of eyelink core graphics library, we use a generic set of callback functions. The callback functions are set and get using the structure HOOKFCNS in combination with the functions `setup_graphic_hook_functions()` and get_all_hook_functions(). The following structures are also used to support various aspects of the core graphics library.

1. InputEvent
2. HOOKFCNS
3. EYECOLOR
4. EYEPALETTE
5. EYEPIXELFORMAT
6. EYEBITMAP

Except for the InputEvent and HOOKFCNS structures, all other structures are used to transfer image data to the core graphics library to save images to disk.

11.2 Writing Your Own Core Graphics.

ELSDK comes with a plug-in template that you can use as as starting point(C:\Program Files\SR Research\EyeLink\Docs\plugin). The plugin template also comes with project files that are intended for a dll. If you would like to use the template within your program, you may want to convert the project to an application.

11.2.1 SDL Core Graphics Sample

In this section we are going to go over a sample implementation of core graphics library. The source code and the project files can be found in C:\Program Files\SR Research\EyeLink\Docs\sdl_coregraphics.

11.2.1.1 File description of cal_graphics.c

This file implements all functions related to calibration presentation for SDL Core Graphics Example and the Source code for cal_graphics.c can be viewed here. In most cases this is by far the most easiest to implement. This file includes implementation of target presentation as well as optional audio presentation, which may improve the speed and stability of calibrations by cueing the subject and make the experimenter's task easier.

If you do not need audio presentation you can safely set the following functions to NULL.

```
HOOKFCNS fcns;
memset(&fcns,0,sizeof(fcns));

fcns.cal_target_beep_hook   = NULL;
fcns.cal_done_beep_hook     = NULL;
fcns.dc_done_beep_hook      = NULL;
fcns.dc_target_beep_hook    = NULL;
```

In our implementation, for function, cal_target_beep(), cal_done_beep(), dc_target_beep(), dc_done_-beep() just call the function cal_sound(). Both cal_target_beep() and cal_done_beep() will be called on calibration or validation procedure in the call to do_tracker_setup() while both the dc_target_beep() and dc_done_beep() will be called on drift correct procedure in the call to do_drift_correct().

The following functions implement the graphics presentation of calibration target.

```
fcns.clear_cal_display_hook = clear_cal_display;
fcns.erase_cal_target_hook  = erase_cal_target;
fcns.draw_cal_target_hook   = draw_cal_target;
fcns.setup_cal_display_hook = setup_cal_display;
fcns.exit_cal_display_hook  = exit_cal_display;
```

In our implementation, we did not need to do anything in setup_cal_display(). So, in turn we did not do anything in exit_cal_display(). Both of these functions can be used to initialize and release resources that are needed with in a calibration session. For example, if you need to use a customized target, that use an SDL Surface, then you would initialize it in the setup_cal_display() and release it in exit_cal_display().

In the function draw_cal_target(), we draw the target at the requested location. The function clear_cal_-display(), clears the entire screen. Since we did not use any fancy background, we implemented erase_-cal_target() so that it clears the entire screen.

11.2.1.2 File description of cam_graphics.c

This file implements all camera display related functions for SDL Core Graphics Example and the Source code for cam_graphics.c can be viewed here.

In this file all camera setup features are implemented. If no camera setup features are neeeded, you can safely set the following callback functions to NULL.

```
        HOOKFCNS fcns;
        memset(&fcns,0,sizeof(fcns));

        fcns.setup_image_display_hook = NULL;
        fcns.exit_image_display_hook  = NULL;
        fcns.image_title_hook         = NULL;
        fcns.draw_image_line_hook     = NULL;
        fcns.set_image_palette_hook   = NULL;
```

In our example, we implemented all the camera image functions except the image_title_hook.

```
  HOOKFCNS fcns;
  memset(&fcns,0,sizeof(fcns));

  fcns.setup_image_display_hook = setup_image_display;
  fcns.image_title_hook         = image_title;
  fcns.draw_image_line_hook     = draw_image_line;
  fcns.set_image_palette_hook   = set_image_palette;
  fcns.exit_image_display_hook  = exit_image_display;
```

For the implementation `setup_image_display()`, we create an RGB surface to hold the camera image. On the exit_image_display() we release the surface.

The draw_image_line() and set_image_palette() together make up the camera image. Once all the image lines are gathered, we draw the cross hair and display the image.

11.2.1.3 File description of graphics_main.c

All functions implemented in this file are miscellaneous functions except for get_input_key(). The Source code for graphics_main.c can be viewed here.

The get_input_key() provides eyelink_core.dll key events. The special keys and modifiers need to be set properly using eyelink's key value for the proper function of keyboard events. If this function is not implemented, you will not be able to control the tracker from Display PC.

The writeImage() writes the given images to disk. If you do not call el_bitmap_save_and_backdrop or el_bitmap_save, this can be ignored. Also, note this function is not set using the standard setup_graphic_-hook_functions() to set the callback.

```
  set_write_image_hook(writeImage,0);
```

All the other functions are completely optional, however, see the implementation of init_expt_graphics() where we make calls to setup_graphic_hook_functions() to set the callbacks.

11.3 Source code for cam_graphics.c

```
/*****************************************************************************
 * EYELINK PORTABLE EXPT SUPPORT      (c) 1996, 2003 by SR Research           *
 *     15 July 2003 by Suganthan Subramaniam       For non-commercial use only    *
 * Header file for standard functions                                         *
 * This module is for user applications   Use is granted for non-commercial       *
 * applications by Eyelink licencees only                                     *
 *                                                                            *
 *                                                                            *
 ******************************************* WARNING ****************************
 *                                                                            *
 * UNDER NO CIRCUMSTANCES SHOULD PARTS OF THESE FILES BE COPIED OR COMBINED.      *
 * This will make your code impossible to upgrade to new releases in the future,  *
 * and SR Research will not give tech support for reorganized code.           *
 *                                                                            *
 * This file should not be modified. If you must modify it, copy the entire file  *
 * with a new name, and change the the new file.                              *
 *                                                                            *
 ******************************************************************************/


/*!\file
\brief Example implementation of camera image graphics using SDL graphics envoronment

In this file, we perform all camera setup features.
@defgroup cam_example all camera display related functions for SDL Core Graphics Example

*/



#include "graphics_global.h"


//draws a line from (x1,y1) to (x2,y2) - required for all tracker versions.
void drawLine(CrossHairInfo *chi, int x1, int y1, int x2, int y2, int cindex);

//draws shap that has semi-circle on either side and connected by lines.
//Bounded by x,y,width,height. x,y may be negative.
//This only needed for EL1000.
void drawLozenge(CrossHairInfo *chi, int x, int y, int width, int height, int cindex);

//Returns the current mouse position and its state. only neede for EL1000.
void getMouseState(CrossHairInfo *chi, int *rx, int *ry, int *rstate);


static UINT32 image_palmap32[130+2];
static SDL_Surface *image = NULL;
/*!
        @ingroup cam_example
        This function is responsible for initializing any resources that are
        required for camera setup.

        @param width width of the source image to expect.
        @param height height of the source image to expect.
        @return -1 if failed,  0 otherwise.
 */
INT16 ELCALLBACK setup_image_display(INT16 width, INT16 height)
{
  image = SDL_CreateRGBSurface(SDL_SWSURFACE,width,height,32,0,0,0,0);
  return 0;
}


/*!
        @ingroup cam_example
        This is called to notify that all camera setup things are complete.  Any
        resources that are allocated in setup_image_display can be released in this
        function.
*/
void ELCALLBACK exit_image_display(void)
{
        if(image)  // release the image surface that we created in setup_image_display
        {
                SDL_FreeSurface(image);
                image = NULL;
        }
}


/*!
  @ingroup cam_example
  This function is called to update any image title change.
  @param threshold if -1 the entire tile is in the title string
                                   otherwise, the threshold of the current image.
  @param title     if threshold is -1, the title contains the whole title
                                   for the image. Otherwise only the camera name is given.
 */
void ELCALLBACK image_title(INT16 threshold, char *title)
{

}




/*!
        @ingroup cam_example
        This function is called after setup_image_display and before the first call to
        draw_image_line. This is responsible to setup the palettes to display the camera
        image.

    @param ncolors number of colors in the palette.
        @param r       red component of rgb.
        @param g       blue component of rgb.
        @param b       green component of rgb.



*/
void ELCALLBACK set_image_palette(INT16 ncolors, byte r[130], byte g[130], byte b[130])
{
        int i =0;
        for(i=0;i<ncolors;i++)
    {
      UINT32 rf = r[i];
      UINT32 gf = g[i];
      UINT32 bf = b[i];
      image_palmap32[i] = (rf<<16) | (gf<<8) | (bf); // we will have an rgba palette setup.
    }
}


/*!
        @ingroup cam_example
        This function is called to supply the image line by line from top to bottom.
        @param width  width of the picture. Essentially, number of bytes in \c pixels.
        @param line   current line of the image
        @param totlines total number of lines in the image. This will always equal
                                        the height of the image.
        @param pixels pixel data.

    Eg. Say we want to extract pixel at position (20,20) and print it out as rgb values.
```

```
        @code
    if(line == 20) // y = 20
        {
                byte pix = pixels[19];
                // Note the r,g,b arrays come from the call to set_image_palette
                printf("RGB %d %d %d\n",r[pix],g[pix],b[pix]);
        }
        @endcode

        @remark certain display draw the image up side down. eg. GDI.
*/
void ELCALLBACK draw_image_line(INT16 width, INT16 line, INT16 totlines, byte *pixels)
{

  short i;
  UINT32 *v0;
  byte *p = pixels;


  v0 = (UINT32 *)(((byte*)image->pixels) + ((line-1)*(image->pitch)));
  for(i=0; i<width; i++)
  {
     *v0++ = image_palmap32[*p++]; // copy the line to image
  }
  if(line == totlines)
  { // at this point we have a complete camera image. This may be very small.
        // we might want to enlarge it. For simplicity reasons, we will skip that.

        // center the camera image on the screen
        SDL_Rect r = {(mainWindow->w-image->w)/2,(mainWindow->h-image->h)/2,0,0};


        // now we need to draw the cursors.

        CrossHairInfo crossHairInfo;
        memset(&crossHairInfo,0,sizeof(crossHairInfo));

        crossHairInfo.w = image->w;
        crossHairInfo.h = image->h;
        crossHairInfo.drawLozenge = drawLozenge;
        crossHairInfo.drawLine = drawLine;
        crossHairInfo.getMouseState = getMouseState;
        crossHairInfo.userdata = image;

        eyelink_draw_cross_hair(&crossHairInfo);

        SDL_BlitSurface(image,NULL,mainWindow,&r);

  }
}


/*!
        @ingroup cam_example
        draws a line from (x1,y1) to (x2,y2) - required for all tracker versions.
*/
void drawLine(CrossHairInfo *chi, int x1, int y1, int x2, int y2, int cindex)
{
  //One can use SDL_gfx to implement this. For ELI and ELII this will always be
  //horizontal and vertical lines.
  //Note, because of rounding, you may get x,y values just off by 1 pixel.
   SDL_Rect r;
   UINT32 color =0;
   SDL_Surface *img = (SDL_Surface *)(chi->userdata);
   switch(cindex)
        {
                case CR_HAIR_COLOR:
```

```
                case PUPIL_HAIR_COLOR:
                        color = SDL_MapRGB(img->format,255,255,255); break;
                case PUPIL_BOX_COLOR:
                        color = SDL_MapRGB(img->format,0,255,0); break;
                case SEARCH_LIMIT_BOX_COLOR:
                case MOUSE_CURSOR_COLOR:
                        color = SDL_MapRGB(img->format,255,0,0); break;
        }
  if(x1 == x2) // vertical line
  {
          if(y1 < y2)
          {
                  r.x = x1;
                  r.y = y1;
                  r.w = 1;
                  r.h = y2-y1;
          }
          else
          {
                  r.x = x2;
                  r.y = y2;
                  r.w = 1;
                  r.h = y1-y2;
          }
          SDL_FillRect(img,&r,color);
  }
  else if(y1 == y2) // horizontal line.
  {
          if(x1 < x2)
          {
                  r.x = x1;
                  r.y = y1;
                  r.w = x2-x1;
                  r.h = 1;
          }
          else
          {
                  r.x = x2;
                  r.y = y2;
                  r.w = x1-x2;
                  r.h = 1;
          }
          SDL_FillRect(img,&r,color);
  }
  else
  {
        printf("non horizontal/vertical lines not implemented. \n");
  }
}

/*!
        @ingroup cam_example
        draws shap that has semi-circle on either side and connected by lines.
        Bounded by x,y,width,height. x,y may be negative.
        @remark This is only needed for EL1000.
*/
void drawLozenge(CrossHairInfo *chi, int x, int y, int width, int height, int cindex)
{
        // NOT IMPLEMENTED.
        printf("drawLozenge not implemented. \n");
}

/*!
        @ingroup cam_example
        Returns the current mouse position and its state.
        @remark This is only needed for EL1000.
*/
```

```
void getMouseState(CrossHairInfo *chi, int *rx, int *ry, int *rstate)
{
  int x =0;
  int y =0;
  Uint8 state =SDL_GetMouseState(&x,&y);
  x = x-(mainWindow->w - ((SDL_Surface*)chi->userdata)->w)/2;
  y = y-(mainWindow->h - ((SDL_Surface*)chi->userdata)->h)/2;
  if(x>=0 && y >=0 && x <=((SDL_Surface*)chi->userdata)->w && y <= ((SDL_Surface*)chi->userdata)->h)
  {
    *rx = x;
        *ry = y;
        *rstate = state;
  }
}
```

11.4 Source code for cal_graphics.c

```
/*****************************************************************************
 * EYELINK PORTABLE EXPT SUPPORT      (c) 1996- 2006 by SR Research           *
 *     15 July 2003 by Suganthan Subramaniam       For non-commercial use only    *
 * Header file for standard functions                                          *
 * This module is for user applications   Use is granted for non-commercial    *
 * applications by Eyelink licencees only                                      *
 *                                                                             *
 *                                                                             *
 ****************************************** WARNING ****************************
 *                                                                             *
 * UNDER NO CIRCUMSTANCES SHOULD PARTS OF THESE FILES BE COPIED OR COMBINED.   *
 * This will make your code impossible to upgrade to new releases in the future, *
 * and SR Research will not give tech support for reorganized code.            *
 *                                                                             *
 * This file should not be modified. If you must modify it, copy the entire file *
 * with a new name, and change the the new file.                               *
 *                                                                             *
 ******************************************************************************/

/*!
\file cal_graphics.c
\brief Example implementation of calibration graphics using SDL graphics envoronment

In this file, we perform all calibration displays.
@defgroup cal_example all functions related to calibration presentation for SDL Core Graphics Example

*/

#include "eyelink.h"
#include "graphics_global.h"


#define TARGET_SIZE 20 /*!< @ingroup cal_example Target size in pixels.*/
/*!
        @ingroup cal_example
        Setup the calibration display. This function called before any
        calibration routines are called.
*/
INT16  ELCALLBACK  setup_cal_display(void)
{
        /* If you would like to use custom targets, you might want to initialize here
           and release it in exit_cal_display. For our demonstration, we are going to use
           rectangle targets, that can easily be drawn on the display.

           So, nothing to do here.
        */
  return 0;
}


/*!
  @ingroup cal_example
  This is called to release any resources that are not required beyond calibration.
  Beyond this call, no calibration functions will be called.
 */
void ELCALLBACK exit_cal_display(void)
{
        /*
        Since we did nothing in setup_cal_display, we don't have much here either.
        */
}

/*!
  @ingroup cal_example
  This function is responsible for the drawing of the target for calibration,validation
  and drift correct at the given coordinate.
  @param x x coordinate of the target.
  @param y y coordinate of the target.
  @remark The x and y are relative to what is sent to the tracker for the
                  command screen_pixel_coords.
 */
void ELCALLBACK  draw_cal_target(INT16 x, INT16 y)
{
        SDL_Rect r = {x,y,TARGET_SIZE,TARGET_SIZE};
        SDL_FillRect(mainWindow,&r,0); // draw the rectangle
        SDL_UpdateRect(mainWindow,0,0,0,0); //update the entire window
}

/*!
        @ingroup cal_example
        This function is responsible for erasing the target that was drawn by the
        last call to draw_cal_target.
*/
void ELCALLBACK erase_cal_target(void)
{
        /*
        Technically, we should keep the last drawn x,y position and erace only the
        piece that we drawn in the last call to draw_cal_target. For simplicity reasons,
        we will clear the entire the screen
        */
        clear_cal_display();
}


/*!
        @ingroup cal_example
  Called to clear the calibration display.
 */
void ELCALLBACK clear_cal_display(void)
{
        // clear the window
        SDL_FillRect(mainWindow,NULL,SDL_MapRGB(mainWindow->format,192,192,192));
        SDL_UpdateRect(mainWindow,0,0,0,0); //update the entire window
}



#define CAL_TARG_BEEP   1 /*!< @ingroup cal_example Calibration target beep*/
#define CAL_GOOD_BEEP   0 /*!< @ingroup cal_example Calibration good beep*/
#define CAL_ERR_BEEP   -1 /*!< @ingroup cal_example Calibration error beep*/
#define DC_TARG_BEEP    3 /*!< @ingroup cal_example Drift correct target beep*/
#define DC_GOOD_BEEP    2 /*!< @ingroup cal_example Drift correct good beep*/
#define DC_ERR_BEEP    -2 /*!< @ingroup cal_example Drift correct error beep*/
/*!
        @ingroup cal_example
        In most cases on can implement all four (cal_target_beep,
        cal_done_beep,dc_target_beep,dc_done_beep) beep callbacks
        using just one function.

        This function is responsible for selecting and playing the audio clip.
        @param sound sound id to play.
*/
void ELCALLBACK  cal_sound(INT16 sound)
{
  char *wave =NULL;
  switch(sound) // select the appropriate sound to play
  {
    case CAL_TARG_BEEP: /* play cal target beep */
                wave ="type.wav";
        break;
    case CAL_GOOD_BEEP: /* play cal good beep */
                wave ="qbeep.wav";
        break;
    case CAL_ERR_BEEP:  /* play cal error beep */
                wave ="error.wav";
        break;
    case DC_TARG_BEEP:  /* play drift correct target beep */
                wave ="type.wav";
        break;
    case DC_GOOD_BEEP:  /* play drift correct good beep */
                wave ="qbeep.wav";
        break;
    case DC_ERR_BEEP:  /* play drift correct error beep */
                wave ="error.wav";
        break;
  }
  if(wave)
  {
          PlaySound(wave,NULL,SND_FILENAME);
  }
}

/*!
@ingroup cal_example
 This function is called to signal new target.
 */
void ELCALLBACK cal_target_beep(void)
{
        cal_sound(CAL_TARG_BEEP);
}

/*!
@ingroup cal_example
  This function is called to signal end of calibration.
  @param error if non zero, then the calibration has error.
 */
void ELCALLBACK cal_done_beep(INT16 error)
{
if(error)
    {
      cal_sound(CAL_ERR_BEEP);
    }
  else
    {
      cal_sound(CAL_GOOD_BEEP);
    }
}

/*!
@ingroup cal_example
  This function is called to signal a new drift correct target.
 */
void ELCALLBACK dc_target_beep(void)
{
        cal_sound(DC_TARG_BEEP);
}

/*!
@ingroup cal_example
  This function is called to singnal the end of drift correct.
  @param error if non zero, then the drift correction failed.
 */
void ELCALLBACK dc_done_beep(INT16 error)
{
 if(error)
    {
      cal_sound(DC_ERR_BEEP);
    }
  else
```

```
    {
      cal_sound(DC_GOOD_BEEP);
    }
}
```

11.5 Source code for graphics_main.c

```
/*****************************************************************************
 * EYELINK PORTABLE EXPT SUPPORT      (c) 1996- 2006 by SR Research          *
 *     15 July 2003 by Suganthan Subramaniam       For non-commercial use only    *
 * Header file for standard functions                                        *
 * This module is for user applications   Use is granted for non-commercial       *
 * applications by Eyelink licencees only                                    *
 *                                                                           *
 *                                                                           *
 ****************************************** WARNING ****************************
 *                                                                           *
 * UNDER NO CIRCUMSTANCES SHOULD PARTS OF THESE FILES BE COPIED OR COMBINED.      *
 * This will make your code impossible to upgrade to new releases in the future,  *
 * and SR Research will not give tech support for reorganized code.          *
 *                                                                           *
 * This file should not be modified. If you must modify it, copy the entire file  *
 * with a new name, and change the the new file.                             *
 *                                                                           *
 *****************************************************************************/




/*!
\file
\brief Example implementation of  graphics initialization using SDL grapphics
environment.

In this file, we detect display information, initialize graphics or close graphics,
write image and save image to disk.

@defgroup misc_example miscellaneous functions
*/


#include "graphics_global.h"
SDL_Surface *mainWindow = NULL; //!< full screen window for all our drawings.




/*!
@ingroup misc_example
  This is an optional function to get information
  on video driver and current mode use this to determine
  if in proper mode for experiment.

  @param[out] di  A valid pointer to DISPLAYINFO is passed in to return values.
  @remark The prototype of this function can be changed to match one's need or
  if it is not necessary, one can choose not to implement this function also.

 */
void ELCALLTYPE get_display_information(DISPLAYINFO *di)
{
  /*
  1. detect the current display mode
  2. fill in values into di
  */

  /* For the demonstration purposes, we assume windows. Other operating systems
     may provide other means of finding the same information.
  */

#ifdef _WIN32
  if(di)
  {
          HDC hdc = GetDC(NULL);
          memset(di,0, sizeof(DISPLAYINFO)); // clear everything to 0
          di->bits = GetDeviceCaps(hdc,BITSPIXEL); // depth
          di->width = GetDeviceCaps(hdc,HORZRES);  // width
          di->height = GetDeviceCaps(hdc,VERTRES); // height
          di->refresh = (float)GetDeviceCaps(hdc,VREFRESH); // refresh rate
  }
#endif
}



/*!
  @ingroup misc_example
  This is an optional function to initialze graphics and calibration system.
  Although, this is optional, one should do the innerds of this function
  elsewhere in a proper manner.

  @remark The prototype of this function can be modified to suit ones needs. Eg.
  The init_expt_graphics of eyelink_core_graphics.dll takes in 2 parameters.

*/
INT16 ELCALLTYPE sdl_init_expt_graphics()
{

  HOOKFCNS fcns;
  memset(&fcns,0,sizeof(fcns)); /* clear the memory */

  /* setup the values for HOOKFCNS */
  fcns.setup_cal_display_hook = setup_cal_display;
  fcns.exit_cal_display_hook  = exit_cal_display;
  fcns.setup_image_display_hook = setup_image_display;
  fcns.image_title_hook       = image_title;
  fcns.draw_image_line_hook   = draw_image_line;
  fcns.set_image_palette_hook = set_image_palette;
  fcns.exit_image_display_hook= exit_image_display;
  fcns.clear_cal_display_hook = clear_cal_display;
  fcns.erase_cal_target_hook  = erase_cal_target;
  fcns.draw_cal_target_hook   = draw_cal_target;
  fcns.cal_target_beep_hook   = cal_target_beep;
  fcns.cal_done_beep_hook     = cal_done_beep;
  fcns.dc_done_beep_hook      = dc_done_beep;
  fcns.dc_target_beep_hook    = dc_target_beep;
  fcns.get_input_key_hook     = get_input_key;


  /* register the call back functions with eyelink_core library */
  setup_graphic_hook_functions(&fcns);

  /* register the write image function */
  set_write_image_hook(writeImage,0);

  /*
        1. initalize graphics
        2. if graphics initalization suceeds, return 0 otherewise return 1.
  */

  if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) // Initialize SDL
  {
          printf( "Couldn't initialize SDL: %s\n",SDL_GetError());
          return -1;
  }
  else
  {
        DISPLAYINFO di;
        SDL_Surface *screen;
        get_display_information(&di);
        // set the display mode to our current display mode.
        screen= SDL_SetVideoMode(di.width,di.height,di.bits, SDL_FULLSCREEN);
        if(screen)
        {
                SDL_ShowCursor(SDL_DISABLE);
                mainWindow = screen;
        }
        else
        {
                printf("Failed to set video mode %dx%d@%d\n",di.width,di.height,di.bits);
                return -1;
        }
  }

  return 0;
}



/*!
 @ingroup misc_example
  This is an optional function to properly close and release any resources
  that are not required beyond calibration needs.
  @remark the prototype of this function can be modified to suit ones need.
 */
void ELCALLTYPE sdl_close_expt_graphics()
{
        mainWindow = NULL;
        SDL_Quit(); // quit sdl
}




/*!
    @ingroup misc_example
        This is called to check for keyboard input.
        In this function:
        \arg check if there are any input events
        \arg if there are input events, fill key_input and return 1.
                 otherwise return 0. If 1 is returned this will be called
             again to check for more events.
        @param[out] key_input  fill in the InputEvent structure to return
                 key,modifier values.
        @return if there is a key, return 1 otherwise return 0.

        @remark Special keys and modifiers should match the following code
        Special keys:
        @code

#define F1_KEY    0x3B00
#define F2_KEY    0x3C00
#define F3_KEY    0x3D00
#define F4_KEY    0x3E00
#define F5_KEY    0x3F00
#define F6_KEY    0x4000
#define F7_KEY    0x4100
#define F8_KEY    0x4200
#define F9_KEY    0x4300
#define F10_KEY   0x4400

#define PAGE_UP    0x4900
#define PAGE_DOWN  0x5100
#define CURS_UP    0x4800
#define CURS_DOWN  0x5000
#define CURS_LEFT  0x4B00
```

```
#define CURS_RIGHT 0x4D00

#define ESC_KEY   0x001B
#define ENTER_KEY 0x000D

        @endcode

        Modifier: If you are using SDL you do not need to modify the
        modifier value as they match the value.

        @code
#define ELKMOD_NONE   0x0000
#define ELKMOD_LSHIFT 0x0001
#define ELKMOD_RSHIFT 0x0002
#define ELKMOD_LCTRL  0x0040
#define ELKMOD_RCTRL  0x0080
#define ELKMOD_LALT   0x0100
#define ELKMOD_RALT   0x0200
#define ELKMOD_LMETA  0x0400
#define ELKMOD_RMETA  0x0800,
#define ELKMOD_NUM    0x1000
#define ELKMOD_CAPS   0x2000
#define ELKMOD_MODE   0x4000
        @endcode
*/

INT16 ELCALLBACK get_input_key(InputEvent *key_input)
{
        return 0;
}



/*!
        @ingroup misc_example
        This function provides support to writing images to disk. Upon calls
        to el_bitmap_save_and_backdrop or el_bitmap_save this function is
        requested to do the write operaiton in the preferred format.

        @param[in] outfilename Name of the file to be saved.
        @param[in] format  format to be saved as.
        @param[in] bitmap bitmap data to be saved.
        @return if successful, return 0.
*/
int ELCALLBACK writeImage(char *outfilename, IMAGETYPE format, EYEBITMAP *bitmap)
{
        // for our demonstration purposes, we will only use bmp format.


        // create an sdl surface from EYEBITMAP
        SDL_Surface * surf = SDL_CreateRGBSurfaceFrom(bitmap->pixels,
                bitmap->w, bitmap->h, bitmap->depth,bitmap->pitch,0,0,0,0);
        if(surf)
        {
                SDL_SaveBMP(surf,outfilename); // save the bitmap
                SDL_FreeSurface(surf); // release the surface
                return 0;
        }
        return -1;
}
```

Chapter 12

Controlling Calibration

The *ELSDK* library encapsulates much of the required functionality for calibration, drift correction, and handling subject setup after recording aborts. To do this, it requires you to supply a window that it can use for calibration, drift correction, and camera image display. You can customize some parameters of calibration and drift correction, including colors, target sizes, and sounds.

12.1 Calibration Colors

In the template, the `SDL_Color` variables `target_foreground_color` and `target_-background_color` record the colors to be used for calibration, drift correction, and camera image graphics respectively. The foreground and background colors must be set in the *eyelink_core_graphics* library by calling `set_calibration_colors()`.

The entire display is cleared to `target_background_color` before calibration and drift correction, and this is also the background for the camera images. The background color should match the average brightness of your experimental display, as this will prevent rapid changes in the subject's pupil size at the start of the trial. This will provide the best eye-tracking accuracy as well.

The `target_foreground_color` is used to draw calibration targets, and for the text on the camera image display. It should be chosen to supply adequate contrast to the background color.

A background color of black with white targets is used by many experimenters, especially for saccadic tasks. A full white (255,255,255) or a medium white (200,200,200) background with black targets is preferable for text. Using white or gray backgrounds rather than black helps reduce pupil size and increase eye- tracking range, and may reduce retinal afterimages.

12.2 Calibration Target Appearance

The standard calibration and drift correction target is a filled circle (for peripheral delectability) with a central "hole" target (for accurate fixation). The sizes of these features may be set with `set_target_-size(diameter, holesize)`. 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.

12.3 Calibration Sounds

The *ELSDK* library plays alerting sounds during calibration, validation and drift correction. These sounds have been found to improve the speed and stability of calibrations by cueing the subject, and make the experimenter's task easier. Three types of sounds are played: a target appearance alert sound, a success sound, and a failure/abort sound. Separate sounds can be selected for setup and drift correction.

Sounds are selected by strings passed to the *eyelink_core_graphics* functions `set_cal_sounds()` and `set_dcorr_sounds()`. If an empty string ("") or NULL is passed to these functions, the default sound is played. If the string is `"off"`, no sound is played. Otherwise, the string is assumed to be the name of a WAV file to be played.

If no sound card is installed, the sounds are produced as "beeps" or "ticks" from the PC speaker. One beep marks target movement, two fast beeps mark success, and three slow beeps mark errors. These can be turned off by passing an `"off"` string for the sound.

To select the sounds to be played for calibration, validation and drift correction during a call to `do_tracker_setup()`, call `set_cal_sounds(char *target, char *good, char *error)`. The sounds played mark the display or movement of the target, successful conclusion of calibration or good validation, and failure or interruption of calibration or validation.

During a call to `do_drift_correct()`, sounds are played as selected by `set_dcorr_sounds(char *target, char *good, char *setup)`. The events are the initial display of the target, successful conclusion of drift correction, and pressing the ESC key to start the Setup menu. Usually the `good` (drift correction completion sound) will be turned off, as this would occur at the start of the trial and might distract the subject.

12.4 Recording Abort Blanking

When a trial is aborted from the tracker (click on the "Abort" button in EyeLink II, or press 'Ctrl'-'Alt'-'A' keys on the tracker keyboard), the eye tracker displays the Abort menu. This is detected by the `check_recording()` function of the *eyelink_core* library, which should be called often by your code during recording. The record abort is handled by this function by clearing the Display PC display to the calibration background color. This prepares the subject for the drift correction or re-calibration that usually follows, and removes the stimuli from the display. After the experimenter selects "Skip Trial", "Repeat Trial", or "Terminate Experiment" from the Abort menu, `check_recording()` returns the appropriate error code.

Chapter 13

Connections and Multiple Computer Configurations

The EyeLink trackers and the *ELSDK* libraries (and similar EyeLink application libraries for other platforms) support connecting multiple computers to one eye tracker, and communication between EyeLink applications. Until version 2.1 of the Windows developer's kit (and tracker versions 2.1 of EyeLink I and v1.1 of EyeLink II), these were not fully functioning under Windows due some peculiarities of the Windows networking implementation. This release fixes these problems and further enhances networking support for multiple computer configurations. However, this version only supports IP addresses in the range of 100.1.1.x with sub net mask 255.255.255.0.

NOTE: EyeLink applications running on different platforms (such as Windows and MacOS) are designed to be able to share broadcast sessions and to communicate using the EyeLink library. However, MS-DOS applications using *SIMTSR* cannot participate in broadcast sessions or communicate with other platforms, as *SIMTSR* does not use TCPIP networking.

When reading the following, keep in mind the kind of configuration you will need. Some examples are:

- One tracker and one computer (the usual configuration).
- One tracker and one experiment computer, plus a "listener" that monitors the connection to generate displays (including calibration targets), perform real-time analysis, or other functions. This computer can use messages generated by the primary experiment computer as well as eye movement data. The *broadcast*, *comm_simple*, and *comm_listener* templates illustrate this.
- One tracker and an experiment computer, plus one or more computers that insert messages into the eye tracker data file.
- One eye tracker plus two experiment computers, both of which control the eye tracker at different times. The computer with the primary connection can do image displays and file transfer, while the broadcast-connected computer can also control most other functions. It is of course important that the two computers are in communication with each other in order to remain synchronized.
- Two eye trackers and two experiment computers, where two subjects can observe each other's gaze positions while performing some mutual task. Each experiment computer connects to one eye tracker (only one eye tracker can have a broadcast connection on a single network), and the two experiment computers must exchange data by another channel (such as the inter-remote EyeLink messaging system).

13.1 Connection Types

13.1.1 Broadcast Connections

In the usual EyeLink experiment, an eye tracker communicates only with the application that first opened a connection to it. If a second computer tries to connect, the connection to the first application was closed. For multiple connections, the EyeLink system uses *broadcast* connections, where data from the eye tracker is sent to all computers on the local network (that is, on the same network cable or connected to the same hub as the eye tracker).

One computer still has the primary connection, but up to 4 other computers can also have broadcast connections to the eye tracker. This primary connection has full control over the eye tracker - broadcast connections are closed when a primary connection is opened or closed, whereas a primary connection is unaffected by broadcast connections. This means that a primary connection should always be opened before a broadcast connection is made from another computer. Therefore it is critical that a "listening" application wait for a primary connection to be made before opening a broadcast connection - the templates *comm_simple* and *comm_listener* exchange messages for this purpose, while the *broadcast* template checks the connection status of the tracker directly. Closing a broadcast connection does not affect the primary connection or any other broadcast connections.

A broadcast connection allows other computers to listen in on real-time data sent from the eye tracker to the primary computer. In addition, broadcast connections allow computers to monitor the eye tracker's state, draw calibration targets, and even send commands and messages to the eye tracker. Some functions are not available with broadcast connections, including camera image display, data playback, and file transfers.

13.1.2 Unconnected Operation

Some functions are also available without opening any connection to the eye tracker. These include sending key presses and messages to the eye tracker, and requesting updates on tracker status and the current tracker time. This capability is designed to allow multiple computers to synchronize their clocks or place synchronizing messages in the tracker's data file. Another use (used in the *broadcast* template) is to allow computers to wait for a primary connection before attempting to open a broadcast connection to the tracker.

13.1.3 Connection Functions

Before a connection can be opened, the EyeLink DLL and networking system must be initialized. The function `open_eyelink_connection()` does this initialization and, if called with an argument of 0, immediately opens a connection. If the argument is -1, it simply initializes the DLL. This allows the application to perform unconnected operations or to open a broadcast session later. Finally, if called with an argument of 1, it "opens" a "dummy" connection, which can be used for debugging applications without the need for a tracker (or even a network).

The network (IP) address of the tracker is usually "100.1.1.1", set in the *eyenet.ini* file on the eye tracker PC. If the eye tracker is not at this address, or if multiple eye trackers are present at different addresses, the tracker address can be specified by calling `set_eyelink_address()` before calling `open_eyelink_connection(1)`. This address is also used for tracker communication if a connection has not been opened, to request tracker status or time or to send messages.

If the tracker address is set to "255.255.255.255", then the DLL will broadcast the connection request to all computers, expecting that the eye tracker will respond. This does not work properly with earlier versions of the eyelink_exptkit DLL, due to problems with broadcasting when multiple network adapters were installed under Windows. Newer tracker and DLL versions use subnet broadcasting, which works properly with all versions of Windows.

If the DLL was initialized with `open_eyelink_connection(-1)`, a primary connection to the tracker can be opened later with `eyelink_open()`, or a broadcast connection opened with `eyelink_broadcast_open()`. After these connections are opened, the connection status should be checked often with `eyelink_is_connected()`, which will return 0 if the connection has been closed. This may happen if the eye tracker software was closed, if another application opened a primary connection to the eye tracker or (in the case of a broadcast connection) if the computer with the primary connection closed its session. This function can also be used to determine the connection type, as it returns 1 for a primary connection, 2 for a broadcast connection, and -1 for a dummy "connection".

Finally, the connection should be closed when your application exits, by calling `close_eyelink_connection()`. This also releases other DLL resources, and can be used with any connection type. If you simply want to close the connection but still need to do unconnected communications, or need access to the high- resolution timing, then call `eyelink_close(1)` instead.

13.2 Finding Trackers and Remotes

The *eyelink_core* library can search for all trackers and computers running EyeLink applications ("remotes") that are connected to the link, and report these. This is performed by calling a polling function, then waiting a short time and checking for a list of responses. To search for a list of trackers, call `eyelink_poll_trackers()`, and call `eyelink_poll_remotes()` to look for applications. Wait 100-200 milliseconds, then call `eyelink_poll_responses()` to get the count of responses received (only the first 4 responses will be recorded).

Responses are retrieved by calling `eyelink_get_node()`, giving the index of the response. This index ranges from 1 (for the first tracker or remote to respond) up to the count returned by `eyelink_poll_responses()`. Several other indexes are defined: for example, index 0 always returns the ELINKNODE (name and address) of the computer the application is running on.

The responses are reported as an ELINKNODE data type, which contains the name and network address (as an ELINKADDR data type) of the tracker or remote. This allows an application or tracker to be selected by name, rather than having to know the IP address of the computer. For trackers, this name is set in the *eyenet.ini* file in the *eyelink\exe* directory on the eyetracker PC. For applications, this name will be the Windows computer network name. This name can also be set by the application by calling `eyelink_set_name()`, allowing other applications to find a specific application independent of the computer it is running on. See *comm_simple* and *comm_listener* for an example of this.

13.3 Inter-Remote Communication

13.3.1 EyeLink Messaging System

To communicate with another EyeLink application, you must have its address in the form of an ELINKADDR. This may be done by search for it by name (as discussed above) or by the IP address of the computer it is running on. The IP address can be converted into an ELINKADDR by calling `text_to_elinkaddr()`, supplying the "dotted" IP name and whether the target is a tracker or remote.

Messages can consist of any kind of text or data, of up to 400 bytes in length. The message can be sent by calling `eyelink_node_send()`, giving the ELINKADDR to send to, and the location and size of the data to be sent. The message will be received by the target application in less than a millisecond, and placed into a buffer. This buffer can then be read by calling `eyelink_node_receive()`, which returns 0 if no new data is present, or the size of the data if any has been received. The function can also supply the ELINKADDR of the sender for use in identification.

The inter-remote receive buffer can only hold one message, so if a new message arrives before the previous message has been read, the old message will be lost. This means that applications should transfer data carefully, with the sender waiting for the receiver to send a message acknowledging receipt before sending more data. There are some situations where this is not needed, for example, if only the most recent data is important (such as the latest gaze position data), or where data is synchronized with other events (such as a message echoing the TRIALID in an experiment).

13.3.2 Communication by Tracker Messages

When broadcast connections are used to let one computer listen in on real-time data, it is possible to eliminate almost all other types of communications between applications. This is done by enabling the inclusion of messages in the real-time link data stream, and having the listening application interpret these in the same way an analysis program would process the recorded EDF file. For example, messages are usually present in the EDF file (and therefore available to the listener) for trial conditions (`"TRIALID"`) and display resolution (`"DISPLAY_COORDS"`).

When messages are included in the real-time link data, each message sent to the tracker is sent back through the link as well. These messages tend to be sent in bursts every few milliseconds, and each burst may contain several messages. EyeLink tracker software is optimized specifically to handle large amounts of network traffic, and can usually handle messages as fast as Windows can send them over the link. However, Windows (especially Windows 98 and 2000) is not as good at receiving messages back, and if messages are sent too rapidly from the main application, the listener application may drop some data. From tests on a 1.5 GHz PC running Windows 2000, it appears to be safe to send a burst of up to 5 messages every 8 milliseconds or so. Dropped data can be detected by the presence of `LOST_DATA_EVENT` events when reading link data with `eyelink_get_next_data()`.

You should also be aware that messages and other events may arrive out of sequence through the link. Messages tend to arrive before any other data with the same time code. In addition, if commands are sent to the tracker just before messages, it is possible that the message may be sent back through the link before the command has finished executing (especially commands that change modes). For example, a TRIALID message may arrive before the listening application has finished processing data from the previous recording block. This can be prevented by adding a short delay before messages, or preceding messages sent after a mode switch command with a call to `eyelink_wait_for_mode_ready()`.

Examples of how to enable sending, receiving, and processing messages by a listening application are given in the *comm_simple* and *comm_listener* templates. One critical step is to enable sending of messages back through the link even when the tracker is not recording data - this is always enabled in EyeLink II and EyeLink1000, but a special version of EyeLink I (versions 2.1 and later) is required. To receive messages and all other data, the reception of link data is turned on by `eyelink_data_switch()`, and left on thereafter.

Chapter 14

Experiment Templates Overview

The fastest way to start developing EyeLink experiments is to use the supplied templates included with the EyeLink Software Development kit. Full source code is supplied for each of several experiments, each illustrating a typical experimental paradigm. You can copy the template's source code to a new directory and modify it to quickly create your own experiments. A detailed analysis of the operation of each template's operation and its source code is included in the following section.

14.1 Template Types

Each template is stored in a subdirectory of the *C:\Program Files\SR Research\Eyelink\Sample-Experiments\sdl* folder. Before compiling or modifying any of the templates, read the section "Programming Experiments" carefully. Be sure to follow the guidelines on copying files and creating new experiments. In particular, be sure to copy and rename files and folders, and do not modify the original template!

The templates are described below:

Simple	The basic experiment template, which is used to introduce the structure of all the templates. The graphics in this experiment are drawn directly to the display.
Text	Introduces the use of bitmaps. It displays 4 formatted pages of text.
Picture	Similar to *text*, but displays 3 pictures (JPG files).
Eyedata	Introduces the use of real-time link data. It records while displaying a gaze-position cursor, then plays back the data from the trial.
GCWindow	Implements a fast gaze-contingent window. This is demonstrated using both text and pictures. Full source code for the gaze contingent window is included.
Control	Uses gaze position data to select items from a grid of letters. This can be used to develop computer interfaces. The code for selection is longer than usual.
Dynamic	Uses refresh-locked drawing and moveable targets to implement sinusoidal smooth pursuit and gap-step-overlap saccadic trials. The code for selection is longer than usual.
Broadcast	Template for an application that eavesdrops on any application, reproducing calibration targets and displaying a gaze cursor (if real-time sample data is enabled).
Comm_simple and Comm_listener	Templates that illustrate a dual-computer experiment. The *comm_simple* template is a modified version of the *simple* template, which works with the *comm_listener* template. This illustrates how real-time data analysis might be performed, by reproducing the display (based on the TRIALID messages) and displaying a gaze cursor.

The discussion of the "simple" template must be read before working with any of the other templates, as it illustrates most of the shared code for all experiments. In general, you should read through all of the templates before beginning programming, as each section assumes you have read the all previous sections.

Chapter 15

"Simple" Template

The experiment first initializes the EyeLink library and connects to the EyeLink tracker. It then creates a full-screen window, and sends a series of commands to the tracker to configure its display resolution, eye movement parsing thresholds, and data types. Using a dialog box built into the *eyelink_w32_comp* library, it asks for a file name for an EDF data file, which it commands the EyeLink tracker to open on its hard disk.

The program then runs a block of trials. Each block begins by calling up the tracker's Camera Setup screen (Setup menu in EyeLink I), from which the experimenter can perform camera setup, calibration, and validation. Four trials are run, each of which displays a single word. After all blocks of trials are completed, the EDF file is closed and transferred via the link from the EyeLink hard disk to the Display PC. At the end of the experiment, the window is erased and the connection to the EyeLink tracker is closed.

Each trial begins by performing a drift correction, where the subject fixates a target to allow the eye tracker to correct for any drift errors. Recording is then started. Recording can be stopped by pressing the 'Esc' key on the Display PC keyboard, the EyeLink Abort menu ('Ctrl' 'Alt' 'A' on the EyeLink keyboard) or by pressing any button on the EyeLink button box.

15.1 Source Files for "Simple"

Before proceeding, you may want to print out copies of these files for reference:

simple.h	Declarations to link together the template experiment files.
main.c	`WinMain()` function for windows win32 compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file.
trials.c	Called to run a block of trials for the *simple* template. Performs system setup at the start of each block, then runs the trials. 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.
trial.c	Implements a trial with simple graphics that can be drawn in one screen refresh, and therefore doesn't need display blanking. You should be able to adapt this file to your experiments by updating the drawing code, messages sent to the EDF file, and changing the handling of subject responses as required.

These files are also required to compile all templates. **These must not be modified.**

eyetypes.h	Declarations of basic data types.
eye_data.h	Declaration of complex EyeLink data types and link data structures.
eyelink.h	Declarations and constants for basic EyeLink functions, Ethernet link, and timing.
sdl_expt.h, core_expt.h	Declarations of ELSDK DLL functions and types. This file will also reference the other EyeLink header files.
eyelink_core.lib	Import library for eyelink_core.dll.
eyelink_core_graphics.lib	Import library for eyelink_core_graphics.dll.
SDL_util.lib SDL_ttf.lib SDLmain.lib SDL.lib	SDL_util library that implements font loading and text printing.
eyelink_w32_comp.lib	Import library for implementing windows dialog boxes for SDL.

15.2 Analysis of "main.c"

This file has the initialization and cleanup code for the experiment, and is used by all the templates. 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.

15.2.1 WinMain()/main()

Execution of every Windows non-console program begins with `WinMain()`, others begin with `main()`. In this case it simply calls `app_main()`, which actually does all the work. The `main()` processes any given command line parameters. The *WinMain* does not take any command line parameters.

```
#if  defined(WIN32) && !defined(_CONSOLE)
```

```
// WinMain - Windows calls this to execute application
int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
        app_main(NULL, NULL);// call our real program
        return 0;
}
#else
// non windows application or win32 console application.
int main(int argc, char ** argv)
{
        DISPLAYINFO disp;
        char *trackerip = NULL;
        int rv = parseArgs(argc,argv, &trackerip, &disp);
        if(rv) return rv;

        if(disp.width)
                app_main(trackerip, &disp);// call our real program
        else
                // call our real program - no display parameters set
                app_main(trackerip, NULL);
        return 0;
}
#endif
```

The function `parseArgs()` in `main()` is used to set the experiment configuration (such as the display resolution, color bits, and tracker IP address) in the command-line mode. This function can be extended for other parameter settings, such as the EDF file name etc.

15.2.2 Initialization

The `app_main()` function begins by asking for an EDF file name, initializing the EyeLink library and opening a connection to the eye tracker. It sets up the `getkey()` system as well. Finally, it checks the EyeLink tracker type using `eyelink_get_tracker_version()`, to determine what enhanced features are available.

```
        #ifdef WIN32
                edit_dialog(NULL,"Create EDF File", "Enter Tracker EDF file name:", our_file_name,260);
        #endif

        if(trackerip)
                set_eyelink_address(trackerip);
        if(open_eyelink_connection(0))
                return -1;       // abort if we can't open link
        set_offline_mode();
        flush_getkey_queue();// initialize getkey() system
        is_eyelink2 = (2 == eyelink_get_tracker_version(NULL) );
```

Next, the video mode is set by calling `init_expt_graphics()`.

```
        if(init_expt_graphics(NULL, disp))
                return exit_eyelink();   // register window with EXPTSPPT

        window = SDL_GetVideoSurface();
```

Following this, information on the current display mode is retrieved. The display resolution and color depth is checked for suitability to the experiment (256-color modes are fine for text and simple graphics but work poorly with the samples that use pictures). Warnings or errors are reported using the `alert_printf()` function supplied by the eyelink_core library.

```
get_display_information(&dispinfo); // get window size, characteristics

// NOTE: Camera display does not support 16-color modes
// NOTE: Picture display examples don't work well with 256-color modes
// However, all other sample programs should work well.

if(dispinfo.palsize==16)      // 16-color modes not functional
{
        alert_printf("This program cannot use 16-color displays");
        return exit_eyelink();
}

// 256-color modes: palettes not supported by this example
if(dispinfo.palsize)
        alert_printf("This program is not optimized for 256-color displays");
```

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.

```
i = SCRWIDTH/60;        // select best size for calibration target
j = SCRWIDTH/300;       // and focal spot in target
if(j < 2) j = 2;
set_target_size(i, j);  // tell DLL the size of target features
// tell EXPTSPPT the colors
set_calibration_colors(&target_foreground_color, &target_background_color);

set_cal_sounds("", "", "");
set_dcorr_sounds("", "off", "off");
```

For this template, we print a title screen. The display is cleared by `clear_full_screen_window()`, a font is selected, and several lines of text are printed using `graphic_printf()`. The text is made visible by calling `SDL_Flip(window);`

```
clear_full_screen_window(target_background_color);
get_new_font("Times Roman", SCRHEIGHT/32, 1);

graphic_printf(window, target_foreground_color, CENTER, SCRWIDTH/2, 1*SCRHEIGHT/30,
        "EyeLink Demonstration Experiment: Sample Code");
graphic_printf(window, target_foreground_color, CENTER, SCRWIDTH/2, 2*SCRHEIGHT/30,
        "Included with the Experiment Programming Kit for Windows");
graphic_printf(window, target_foreground_color, CENTER, SCRWIDTH/2, 3*SCRHEIGHT/30,
        "All code is Copyright (c) 1997-2002 SR Research Ltd.");
graphic_printf(window, target_foreground_color, CENTER, SCRWIDTH/5, 4*SCRHEIGHT/30,
        "Source code may be used as template for your experiments.");
SDL_Flip(window);
```

15.2.3 Opening an EDF file

The *eyelink_w32_comp* library function `edit_dialog()` is called to ask for a file name, up to 8 characters in length. If no file name was entered, `our_file_name` is left blank and no file is created. A line of text is added to the start of the EDF file, to mark the application that created the file. This can later be viewed with a text editor. While in full screen mode of SDL calling `edit_dialog()` may be a problem since, the dialog will be covered by the full screen window. So, what you want to do is to call this function before calling `init_expt_graphics()`.

```
if(our_file_name[0])    // If file name set, open it
```

```
{
        // add extension
        if(!strstr(our_file_name, ".")) strcat(our_file_name, ".EDF");
        i = open_data_file(our_file_name); // open file
        if(i!=0)                           // check for error
        {
                alert_printf("Cannot create EDF file '%s'", our_file_name);
                return exit_eyelink();
        }                                // add title to preamble
        eyecmd_printf("add_file_preamble_text 'RECORDED BY %s' ", program_name);
}
```

15.2.4 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 viewers will work in display pixel coordinates. The calibration type is also set, with "HV9" setting the usual 9-point calibration. A calibration type of "H3" would calibrate and collect data for horizontal gaze position only.

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. The refresh rate is recorded in the "FRAMERATE" message, which can be used to correct the onset time of stimuli for monitor refresh delay. This message should not be included if refresh synchronization is not available, and is optional if refresh-locked presentation of stimuli is not used.

```
// Now configure tracker for display resolution
// Set display resolution
eyecmd_printf("screen_pixel_coords = %ld %ld %ld %ld",
        dispinfo.left, dispinfo.top, dispinfo.right, dispinfo.bottom);
// Setup calibration type
eyecmd_printf("calibration_type = HV9");

// Add resolution to EDF file
eyemsg_printf("DISPLAY_COORDS %ld %ld %ld %ld",
        dispinfo.left, dispinfo.top, dispinfo.right, dispinfo.bottom);
if(dispinfo.refresh>40)
        eyemsg_printf("FRAMERATE %1.2f Hz.", dispinfo.refresh);
```

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 or EyeLink 1000 eye tracker, we select a parser configuration rather than changing the saccade detector parameters:

```
// SET UP TRACKER CONFIGURATION
// set parser saccade thresholds (conservative settings)
if(is_eyelink2)
{
        // 0 = standard sensitivity
        eyecmd_printf("select_parser_configuration 0");
}
else
{
        eyecmd_printf("saccade_velocity_threshold = 35");
        eyecmd_printf("saccade_acceleration_threshold = 9500");
}
// set EDF file contents
eyecmd_printf("file_event_filter = LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON");
eyecmd_printf("file_sample_data  = LEFT,RIGHT,GAZE,AREA,GAZERES,STATUS");
// set link data (used for gaze cursor)
eyecmd_printf("link_event_filter = LEFT,RIGHT,FIXATION,SACCADE,BLINK,BUTTON");
eyecmd_printf("link_sample_data  = LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS");
```

With the SR Research button box, the large button (#5) is often used by the subject to start trials, by terminating the drift correction. This command to the EyeLink tracker causes the button to be programmed to act like the ENTER key or spacebar on the tracker.

```
        // Program button #5 for use in drift correction
        eyecmd_printf("button_function 5 'accept_target_fixation'");
```

15.2.5 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. This function is implemented in trials.c, and each of the templates implements this function differently. For a multiple-block experiment, you would call this function once for every block, and call this function in a loop. The function `run_trials()` returns an result code: if this is `ABORT_EXPT`, the program should exit the block loop and terminate the experiment.

```
        // make sure we're still alive
        if(!eyelink_is_connected() || break_pressed())
                return end_expt(our_file_name);

        // RUN THE EXPERIMENTAL TRIALS (code depends on type of experiment)
        // Calling run_trials() performs a calibration followed by trials
        // This is equivalent to one block of an experiment
        // It will return ABORT_EXPT if the program should exit

        i = run_trials();
        return end_expt(our_file_name);
```

15.2.6 Transferring the EDF file

Once all trials are run, the EDF file is closed and transferred via the link to the Display PC's hard disk. The tracker is set to Offline mode to speed the transfer with EyeLink I. The `receive_data_file()` function is called to copy the file. If the file name is not known (i.e. opened from the Output menu on the tracker) the first argument can be "" to receive the last recorded data file. If the second argument is "", a dialog box will be displayed to allow the file to be renamed, or the transfer cancelled. The second argument could also be used to supply a name for the file, in which case the dialog box will not be displayed. If the last argument is not 0, the second argument is considered to be a path to store the new file in.

```
int end_expt(char * our_file_name)
{
        //END: close, transfer EDF file
        set_offline_mode(); // set offline mode so we can transfer file
        pump_delay(500);    // delay so tracker is ready
        // close data file
        eyecmd_printf("close_data_file");

        if(break_pressed())
                return exit_eyelink(); // don't get file if we aborted experiment
        if(our_file_name[0])   // make sure we created a file
        {
                close_expt_graphics(); // tell EXPTSPPT to release window
                receive_data_file(our_file_name, "", 0);
        }
        // transfer the file, ask for a local name

        return exit_eyelink();
}
```

15.2.7 Cleaning Up

Finally, the program cleans up. It un-registers the window with *eyelink_core_graphics*, closes the connection to the eye tracker, and closes its window.

```
int exit_eyelink()
{
        // CLEANUP
        close_expt_graphics();           // tell EXPTSPPT to release window
        close_eyelink_connection();      // disconnect from tracker
        return 0;
}
```

15.2.8 Extending the Experiment Setup

The code in `app_main()` is very basic. In a real experiment, you would need to add some of these functions:

- Randomization, either computed in the program or by reading a control file containing trial data or lists of text or image files.
- Subject instructions and practice trials.
- Messages at the start of the EDF file describing the experiment type, subject ID, etc.

15.3 Analysis of "trials.c"

This module performs the actions needed to perform block of experiment trials. There are two functions: `run_trials()` is called to loop through a set of trials; and `do_simple_trial()`, which supplies stimuli and trial identifiers for each trial, indexed by the trial number.

15.3.1 Initial Setup and Calibration

The `run_trials()` function begins by setting the calibration background color (which should match the average brightness of the screen in the following trials), then 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.

```
int run_trials(void)
{
        int i;
        int trial;

        // This should match the display
        SETCOLOR(target_background_color ,255,255,255);
        set_calibration_colors(&target_foreground_color, &target_background_color);

        // TRIAL_VAR_LABELS message is recorded for EyeLink Data Viewer analysis
        // It specifies the list of trial variables for the trial
        // This should be written once only and put before the recording of individual
        // trials
        eyemsg_printf("TRIAL_VAR_LABELS TRIAL_WORD");
        // PERFORM CAMERA SETUP, CALIBRATION
        do_tracker_setup();
```

15.3.2 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_core* 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. The example uses a `switch()` statement to handle the return codes.

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:

```
        // loop through trials
        for(trial=1;trial<=NTRIALS;trial++)
        {
                // drop out if link closed
                if(eyelink_is_connected()==0 || break_pressed())
                {
                        return ABORT_EXPT;
                }

                // RUN THE TRIAL
                i = do_simple_trial(trial);
                // safety: make sure realtime mode stopped
                end_realtime_mode();

                switch(i)               // REPORT ANY ERRORS
                {
                        case ABORT_EXPT:        // handle experiment abort or disconnect
                                eyemsg_printf("EXPERIMENT ABORTED");
                                return ABORT_EXPT;
                        case REPEAT_TRIAL:              // trial restart requested
                                eyemsg_printf("TRIAL REPEATED");
                                trial--;
                                break;
                        case SKIP_TRIAL:                // skip trial
                                eyemsg_printf("TRIAL ABORTED");
                                break;
                        case TRIAL_OK:          // successful trial
                                eyemsg_printf("TRIAL OK");
                                break;
                        default:                // other error code
                                eyemsg_printf("TRIAL ERROR");
                                break;
                }
        }  // END OF TRIAL LOOP
        return 0;
}
```

15.3.3 Trial Setup Function

The actual trial is performed in two steps. The first is performed in the *trials.c* module, and does trial-specific setup. The second, in *trial.c*, presents the stimulus and performs the actual recording. This division of the code is ideal for most experiments, where every trial is recorded in the same way but only stimuli and trial identification differ. If trials do differ in function, the setup function can pass an argument to the trial execution function specifying trial type, or call the appropriate version of this function.

The trial setup is performed by `do_simple_trial()`. It first sets the trial identification title by sending the record_status_message command to the tracker. 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. This message is displayed at the bottom right of the EyeLink tracker display during recording. This will let the experimenter know how far the experiment has progressed, and will be essential in fixing problems that the experimenter notices during recording.

15.3.4 "TRIALID" Message

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. The first item in the TRIALID message should be a maximum of 12 characters, uniquely identifying the trial. This item will be included in data analysis files.

```
int do_simple_trial(int num)
{
        // This supplies the title at the bottom of the eyetracker display
        eyecmd_printf("record_status_message 'SIMPLE WORDS, TRIAL %d/%d' ", num, NTRIALS);

        // Always send a TRIALID message before starting to record.
        // It should contain trial condition data required for analysis.
        eyemsg_printf("TRIALID %s", trial_word[num-1]);

        // TRIAL_VAR_DATA message is recorded for EyeLink Data Viewer analysis
        // It specifies the list of trial variables value for the trial
        // This must be specified within the scope of an individual trial (i.e., after
        // "TRIALID" and before "TRIAL_RESULT")
        eyemsg_printf("!V TRIAL_VAR_DATA %s", trial_word[num-1]);
```

15.3.5 Tracker Feedback Graphics

Next, tracker background graphics are drawn on the EyeLink display, to allow the experimenter to evaluate subject performance and tracking errors using the real-time gaze cursor. Without these graphics, it is impossible to monitor the accuracy of data being produced by the EyeLink tracker, and to judge when a re-calibration is required. Very simple boxes around important details on the subject display are sufficient. For example, marking lines or words in text is sufficient for reading studies.

This is a list of the most useful drawing commands, which are further documented in the chapter "Useful EyeLink Commands".

Command	Drawing Operation
clear_screen	Clears EyeLink display to any color
draw_line	Draw a line in any color between two points
draw_box	Outlines a box in any color
draw_filled_box	Draws a filled rectangle in any color
draw_text	Prints text in any color at any location
draw_cross	Draws a small "+" target to indicate an important visual location

The background graphics can be very simple: in this case, a small box at the display center to mark the drift-correction target. The tracker must first be placed in Offline mode before drawing. Graphics drawn in Offline mode are saved for display in the next recording session.

The command "clear_screen" erases the tracker display to color 0 (black), and "draw_box" draws a small box in color 7 (medium gray). Drawing coordinates are the same as used for drawing on the local display: the EyeLink tracker converts these gaze coordinates before drawing.

```
        // Before recording, we place reference graphics on the EyeLink display
        set_offline_mode();  // Must be offline to draw to EyeLink screen
        // clear tracker display and draw box at center
        eyecmd_printf("clear_screen 0");
        eyecmd_printf("draw_box %d %d %d %d  7",
                SCRWIDTH/2-16, SCRHEIGHT/2-16, SCRWIDTH/2+16, SCRHEIGHT/2+16);
```

15.3.6 Executing the Trial

Finally, we can call the actual trial recording loop, supplying any data it needs. In this case, this is the text to display and the maximum recording time in milliseconds.

```
        // Actually run the trial: display a single word
        return simple_recording_trial(trial_word[num-1], 20000L);
}
```

15.4 Control by Scripts

Another method that can be used to sequence trials is to read a text file, interpreting each line as instructions for a single trial. These instructions might include the title of the trial (displayed at the bottom of the EyeLink tracker display during recording), the "TRIALID" message for the trial, location and identity of stimuli, and independent variable values.

The script file format could be of two types. The simplest contain one line per trial, containing all data needed to specify the trial conditions. A more flexible format defines one parameter with each line in the file, with the first word on the line specifying what the line contains. These lines might also be used to command tracker setup, or the display of an instruction display. Such scripted experiments do not require block or trial loops. Instead, a loop reads lines from the script file, interprets each line, and carries out its instructions. This loop must still interpret the return code from trials, and place proper messages into the EDF file. It may be helpful to record the script lines as messages in the EDF file as well, to help document the experiment.

Scripts can be generated by a separate C, Basic, or other program, which will do the randomization, convert independent variables into program data (i.e. BMP file names for graphical stimuli, or saccade target positions), and create the "TRIALID" message and title lines. These are then written to a large text file.

These files are also more easily analyzed for randomization errors than is the case for an experiment with a built-in randomizer.

15.5 Analysis of "trial.c"

The second part of the trial is the actual recording loop, implemented by `simple_recording_trial()`. This is the most basic recording loop, and is the basis for all the other template recording loops. 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.

15.5.1 Overview of Recording

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
- 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.

15.6 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 or EyeLink 1000 tracker's corneal reflection mode, a fixation target should be presented, and a drift correction allows the experimenter the opportunity to re-calibrate if needed.

The *eyelink_core* DLL function `do_drift_correct()` 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.

```
        // NOTE: TRIALID AND TITLE MUST HAVE BEEN SET BEFORE DRIFT CORRECTION!
        // FAILURE TO INCLUDE THESE MAY CAUSE INCOMPATIBILITIES WITH ANALYSIS SOFTWARE!
        // DO PRE-TRIAL DRIFT CORRECTION
        // We repeat if ESC key pressed to do setup.
```

```
while(1)
{
        // Check link often so we can exit if tracker stopped
        if(!eyelink_is_connected()) return ABORT_EXPT;
        // We let do_drift_correct() draw target in this example
        // 3rd argument would be 0 if we already drew the fixation target

        error = do_drift_correct((INT16)(SCRWIDTH/2), (INT16)(SCRHEIGHT/2), 1, 1);
        // repeat if ESC was pressed to access Setup menu
        if(error!=27) break;
}

// make sure display is blank
clear_full_screen_window(target_background_color);
```

In the template, we told `do_drift_correct()` 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. Usually, you will let *eyelink_core* clear the display and draw the drift correction target, as in the template. It's a good precaution to clear the display after drift correction completes - this should always cleared to `target_background_color`, to prevent abrupt display changes that could affect subject readiness.

Sometimes it is better to draw the target ourselves, for example if the drift correction is part of the initial fixation in a saccadic task and we want the target to stay on the screen after the drift correction. To do this, we pre-draw the target, then call `do_drift_correct()` with the third parameter set to 0.

If the 'Esc' key was pressed during drift correction, the EyeLink Camera Setup screen (the Setup menu in EyeLink I) is called up to allow calibration problems to be corrected, and `do_drift_correct()` will return 27 (`ESC_KEY`). In this case, the drift correction should be repeated. Any fixation target that was pre-drawn must also be redrawn, as this will have cleared the display. Access to the Setup menu can be disabled by setting the fourth argument to 0, which will cause the 'Esc' key press to simply abort the drift correction and return 27 (`ESC_KEY`).

15.7 Starting Recording

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

```
// Start data recording to EDF file, BEFORE DISPLAYING STIMULUS
// You should always start recording 50-100 msec before required
// otherwise you may lose a few msec of data

// record samples and events to EDF file only
error = start_recording(1,1,0,0);
if(error != 0)  return error;       // return error code if failed
```

The *eyelink_core* function `start_recording()` 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 `start_recording()` 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:

`INT16 start_recording(INT16 file_samples, INT16 file_events, INT16 link_samples, INT16 link_events);`

Argument	Controls
file_samples	Enables writing of samples to EDF file
file_events	Enables writing of events to EDF file
link_samples	Enables real-time samples through link
link_events	Enables real-time events through link

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 several megabytes for a 30-minute experiment) but allows the file to be viewed with Data Viewer software, and 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 begin_realtime_mode(), discussed below), 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 DataViewer.

```
        // record for 100 msec before displaying stimulus
        begin_realtime_mode(100);
        // Windows 2000/XP: no interruptions till display start marked
```

15.8 Starting Realtime Mode

Under Windows 2000 and Windows XP, it is possible to place your application in realtime priority mode - this forces Windows to stop most other activities, including background disk access, that might cause your experiment to have unpredictable delays. This will make the timing of stimulus presentation more predictable, and should be used whenever possible.

Realtime mode is entered by calling begin_realtime_mode(), which implements the correct procedure for whatever version of Windows is running the application. This function may take up to 100 milliseconds to return, and this delay may change in future versions of the *eyelink_core* DLL as new operating systems are introduced. The minimum delay of begin_realtime_mode() may be specified (100 milliseconds in this template, to allow recording of data before the display start) so that this delay can serve a useful purpose. We end realtime mode with end_realtime_mode(), which has no significant delay.

In the case of the *simple* template, the only critical temporal section is marking the onset of the display by a message in the EDF file. We begin realtime mode after recording starts, and will end it after the display-onset messages for the EDF file have been sent to the EyeLink tracker. In this experiment, we don't need refresh-locked accuracy in determining when to erase the stimulus word. If we needed accurate display timing (as we will in some other examples), we will not exit realtime mode until the end of the trial. The eyelink_core recording support functions check_recording() will end realtime mode if an error occurs, and calling check_record_exit() at the end of the trial function will also return to normal mode.

The major problem with realtime mode is that certain system functions simply cease to work - so try to exit realtime mode as soon as it is no longer needed. You will not be able to play sounds, and the keyboard will not work properly while in realtime mode. Under Windows XP, even the escape_pressed() and break_pressed() functions will not work (these are fine under Windows 2000, however). It is possible that the Windows DirectX library (DirectInput and DirectSound) may work in realtime mode, but this has not yet been tested.

Using getkey() will probably return key presses, but it does this at the expense of allowing Windows to do background activity. This appears as unpredictable delays that can be as long as 20 milliseconds (but are usually 5-10 milliseconds). Therefore you should try to avoid using the Display PC keyboard for control or subject responses (keyboards are not very time accurate in any case). Instead, use last_-button_press() to detect tracker button responses, break_pressed() to detect program shut-down, eyelink_is_connected() to detect tracker disconnection, check_recording() to detect recording aborted by the eye tracker, and, if required, eyelink_read_keybutton() to monitor the EyeLink tracker keyboard.

15.9 Drawing the Subject Display

The *simple* template uses the simplest possible graphics for clarity, drawing a single word of text to the display. We let *eyelink_core* to draw the drift correction target, so it also clears the display for us. The word is drawn rapidly, so we don't need to draw to a bitmap and copy it to the display. If a lot of text were being drawn, it might be useful to first draw the text in the background color - this allows Windows to generate and cache the font bitmaps internally (this is not demonstrated here).

```
// DISPLAY OUR IMAGE TO SUBJECT
// If graphics are very simple, you may be able to draw them
// in one refresh period.  Otherwise, draw to a bitmap first.

// select font for drawing
get_new_font("Times Roman", SCRWIDTH/25, 1);

// Because of faster drawing speeds and good refresh locking,
// we now place the stimulus onset message just after display refresh
// and before drawing the stimulus.  This is accurate and will allow
// drawing a new stimulus each display refresh.
// However, we do NOT send the message after the retrace--this may take too long
// instead, we add a number to the message that represents the delay
// from the event to the message in msec


// Draw the stimulus, centered
graphic_printf(window, target_foreground_color,
        CENTER, SCRWIDTH/2, (SCRHEIGHT-get_font_height())/2, "%s", text);
Flip(window);
// time of retrace
drawing_time = current_msec();
trial_start = drawing_time;
graphic_printf(window, target_foreground_color,
        CENTER, SCRWIDTH/2, (SCRHEIGHT-get_font_height())/2, "%s", text);

// delay from retrace (time to draw)
drawing_time = current_msec()-drawing_time;
// message for RT recording in analysis
eyemsg_printf("%d DISPLAY ON", drawing_time);
// message marks zero-plot time for EDFVIEW
eyemsg_printf("SYNCTIME %d", drawing_time);
```

The drawing procedure uses Flip(), which does not return until the next screen refresh really occurs. Immediately following this function, 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. (NOTE that the delay is placed first in the "DISPLAY ON" message - this allows new EyeLink analysis tools to automatically adjust the message time, and this method should be used in all messages where delay must be corrected for. The delay is placed last in the "SYNCTIME" message, for compatibility with the EDFVIEW viewer). Synchronizing with the display refresh before drawing and recording the delay from the retrace to the message writing will allow accurate calculation of the time of stimulus onset, as the refresh time can be recomputed by subtracting the delay from the message timestamp. The offset between the time that the stimulus appeared on the monitor and the message timestamp depends only on the stimulus vertical position.

15.10 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.

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

```
        // we would stay in realtime mode if timing is critical
        // for example, if a dynamic (changing) stimulus was used
        // or if display duration accuracy of 1 video refresh. was needed
        // we don't care as much about time now, allow keyboard to work

        end_realtime_mode();

        // Now get ready for trial loop
        eyelink_flush_keybuttons(0);   // reset keys and buttons from tracker

        // we don't use getkey() especially in a time-critical trial
        // as Windows may interrupt us and cause an unpredicatable delay
        // so we would use buttons or tracker keys only

        // Trial loop: till timeout or response
```

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:

```
// End recording: adds 100 msec of data to catch final events
static void end_trial(void)
{
        clear_full_screen_window(target_background_color);  // hide display
        end_realtime_mode();    // NEW: ensure we release realtime lock
        pump_delay(100);        // CHANGED: allow Windows to clean up
                // while we record additional 100 msec of data
        stop_recording();
}
```

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:

```
        // First, check if recording aborted
        if((error=check_recording())!=0) 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.

```
        // Check if trial time limit expired
        if(current_time() > trial_start+time_limit)
        {
                eyemsg_printf("TIMEOUT");    // message to log the timeout
                end_trial();                 // local function to stop recording
                button = 0;                      // trial result message is 0 if timeout
                break;                       // exit trial loop
        }
```

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 Display PC, as this function allows Windows multitasking and background disk activity to occur. It is preferable to use `escape_pressed()` and `break_pressed()` 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.

```
                // check for program termination or ALT-F4 or CTRL-C keys
                if(break_pressed())
                {
                        end_trial();          // local function to stop recording
                        return ABORT_EXPT;    // return this code to terminate experiment

                }

                // check for local ESC key to abort trial (useful in debugging)
                if(escape_pressed())
                {
                        end_trial();          // local function to stop recording
                        return SKIP_TRIAL;    // return this code if trial terminated
                }
```

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 `eyelink_last_button_press()` is the simplest way to check for new button presses since the last call to this function. It returns 0 if no button has been pressed since recording started, or the button number. 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.

```
        // BUTTON RESPONSE TEST
        // Check for eye-tracker buttons pressed
        // This is the preferred way to get response data or end trials

        button = eyelink_last_button_press(NULL);
        if(button!=0)        // button number, or 0 if none pressed
        {
                // message to log the button press
                eyemsg_printf("ENDBUTTON %d", button);
                // local function to stop recording
                end_trial();
                break;        // exit trial loop
        }
```

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.

15.11 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.

```
        }                          // END OF RECORDING LOOP
        end_realtime_mode();       // safety cleanup code
        while(getkey());           // dump any accumulated key presses
}
```

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 check_record_exit() function makes sure that no Abort menu operations remain to be performed, and generates the trial return code.

```
        // report response result: 0=timeout, else button number
        eyemsg_printf("TRIAL_RESULT %d", button);
        // Call this at the end of the trial, to handle special conditions
        return check_record_exit();
```

15.12 Extending "trial.c"

For many experiments, this code can be used with few modifications, except for changing the stimulus drawn. Another common enhancement would be to filter which buttons can be used for response.

A more sophisticated modification would be to animate the display. This might involve erasing and redrawing fixation targets for saccadic response experiments. For this experiment, the time each target is drawn and erased must be controlled and monitored carefully, which required leaving realtime mode on throughout the trial. Checking for recording errors or responses can be postponed until after the display sequence, unless it is very long (for example, a continuously moving smooth pursuit target).

Be sure to call eyemsg_printf() to place a message in the EDF file to record the time the display is changed. Don't send more than 10 messages every 20 milliseconds, or the EyeLink tracker may not be able to write them all to the EDF file.

This sample code illustrates the above concepts:

```
        SDL_FlipEx(window);// synchronized flip. This will return after the flip occur
        eyemsg_printf("TARGET MOVED %d %d", x, y);  // record time, new position
        eyemsg_printf("MARK 1");                    // time marker #1
```

Chapter 16

"TEXT" Template

For more complex display such as screens of text or pictures, drawing takes too long to complete in one (or even two) display refresh periods. This makes the drawing process visible, and there is no clear stimulus onset for reaction time measurement. The code in the *text* template draws to a bitmap, then copies it to the display, reducing the time to make the display visible. This also has the advantage of making the trial code more general: almost any stimulus can be displayed given its bitmap.

16.1 Source Files for "Text"

These are the files used to build *text*. Those that are the same as for *simple* are marked with an asterisk. (Standard *ELSDK* header files are the same as those used for *simple*, and are not listed here).

main.c	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file.
trials.c	Called to run a block of trials for the "text" template. Performs system setup at the start of each block, then runs the trials. 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.
trial.c	Implements a trial with simple graphics that can be draw in one screen refresh, and therefore doesn't need display blanking. You should be able to use this by replacing the drawing code, message code, and handling any subject responses.

16.2 Differences from "Simple"

The only difference in the *main.c* is the inclusion of "text.h" header file, instead of "simple.h" in the previous example. The block loop and trial setup code (*trials.c*) and the trial loop code (*trial.c*) are also changed. The latter two files will be discussed below, but here is an outline of the differences from the equivalent files for *simple:*

- Trial title and TRIALID messages are different.
- A bitmap of text is created as part of the trial setup.
- The display graphics are copied from a bitmap passed to the trial loop.

The discussion of *trial.c* is especially important, as it forms the basis of later templates that use bitmaps.

16.3 Analysis of "trials.c"

The function `do_text_trial()` is used to setup and run the text-reading trials. This first creates an SDL_Surface and calls `graphic_printf()` with the long text and alignment parameters. Margins are set at about 1.5° (1/20 of display height and width) from the edges of the display. Following this, the `bitmap_save_and_backdrop()` function saves the entire bitmap as a file in the specified path ("images"), and transfers the image to the tracker PC as backdrop for gaze cursors. Note that you can manipulate the file-saving option of the function so that the existing files could be either overwritten (set the sv_options as 0) or not to be overwritten (SV_NOREPLACE). The trial title and "TRIALID" message report the page number (and thus the progress of the experiment).

```
char *pages[4] = { text1, text2, text3, text4 };

// Given trial number, execute trials Returns trial result code
int do_text_trial(int num)
{
        SDL_Surface *bitmap = NULL;
        int i;
        char image_fn[100];     // image file name;

        // Must be offline to draw to EyeLink screen
        set_offline_mode();
        eyecmd_printf("clear_screen 0");

        // This supplies the title at the bottom of the eyetracker display
        eyecmd_printf("record_status_message 'TEXT, PAGE %d/%d' ", num, NTRIALS);

        // Always send a TRIALID message before starting to record.
        // It should contain trial condition data required for analysis.
        eyemsg_printf("TRIALID PAGE%d", num);

        // TRIAL_VAR_DATA message is recorded for EyeLink Data Viewer analysis
        // It specifies the list of trial variables value for the trial
        // This must be specified within the scope of an individual trial (i.e., after
        // "TRIALID" and before "TRIAL_RESULT")
        eyemsg_printf("!V TRIAL_VAR_DATA %d", num);

        sprintf(image_fn, "test%d.png", num);   // get the image file name
        // IMGLOAD command is recorded for EyeLink Data Viewer analysis
        // It displays a default image on the overlay mode of the trial viewer screen.
        // Writes the image filename + path info
        eyemsg_printf("!V IMGLOAD FILL Images/%s", image_fn);

        get_new_font("Times Roman", SCRHEIGHT/25, 0);

        bitmap = SDL_CreateRGBSurface(SDL_SWSURFACE,SCRWIDTH,
                SCRHEIGHT,dispinfo.bits,0,0,0,0);
        set_margin(SCRWIDTH/20,SCRHEIGHT/20,SCRWIDTH/20,SCRHEIGHT/20);
        set_line_spacing(((double)(SCRHEIGHT)/15.0)/(double) get_font_height());
        SDL_FillRect(bitmap,NULL,SDL_MapRGB(bitmap->format, target_background_color.r,
                target_background_color.g, target_background_color.b));
        graphic_printf(bitmap,target_foreground_color,WRAP,0,0,pages[num-1]);
        // Save bitmap and transfer to the tracker pc.
        // Since it takes a long to save the bitmap to the file, the
        // value of sv_options should be set as SV_NOREPLACE to save time
        bitmap_save_and_backdrop(bitmap, 0, 0, 0, 0, image_fn,"images",SV_NOREPLACE,
                0,0, BX_NODITHER|BX_GRAYSCALE);

        i = bitmap_recording_trial(bitmap, 20000L);
        SDL_FreeSurface(bitmap);

        return i;
}
```

Finally, the bitmap is passed to `bitmap_recording_trial()` for the actual recording. After recording is completed, the bitmap is deleted. That's all there is to it - the same trial function can be used to present almost any stimulus drawn to the bitmap.

16.4 Analysis of "trial.c"

The function `bitmap_recording_trial()` performs straightforward recording and presentation of bitmaps. The code is almost identical to `simple_recording_trial()`, except that drawing of text is replaced by copying the bitmap to the display.

```
        // COPY BITMAP to the back buffer
        SDL_BlitSurface(gbm,NULL,window, NULL);
        // asynchronously flip  flip does not return till the next retrace
        Flip(window);
        SDL_BlitSurface(gbm,NULL,window, NULL);
        trial_start = current_msec();                   // time of retrace
        // message marks zero-plot time for EDFVIEW
        display_time = current_msec()-trial_start;  // retrace-to-draw delay
        // message for RT recording in analysis
        eyemsg_printf("%d DISPLAY ON", display_time);
        eyemsg_printf("SYNCTIME %d", display_time);
```

Chapter 17

"PICTURE" Template

The template *picture* is almost identical to *text*, except that images are loaded from JPG files and displayed instead of text.

17.1 Source Files for "Picture"

These are the files used to build *picture*. Those that are the same as for *text* are marked with an asterisk.

picture.h	Declarations to link together the template experiment files. Most of the declarations in this file can be used in your experiments.
main.c ∗	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file.
trials.c	Called to run a block of trials for the "picture" template. Performs system setup at the start of each block, then runs the trials. 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.
trial.c	Implements a trial with simple graphics that can be drawn in one screen refresh, and therefore doesn't need display blanking. You should be able to use this by replacing the drawing code, message code, and handling any subject responses.

17.2 Differences from "Text"

The only difference is the block loop module (*trials.c*), and the use of *IMG_Load* to load an image file as a bitmap. The same trial recording file (*trial.c*) as in text template is used to display the stimuli and record data.

17.3 Analysis of "trials.c"

The first difference from the *trials.c* in text template is the color of the background. This is medium gray, matching the average luminance of the pictures. In a real experiment, each picture may have a different average luminance, and the calibration background color should be reset before each trial so that the drift correction background matches the image.

```
        // This should match the display
        SETCOLOR(target_background_color,128,128,128);
        set_calibration_colors(&target_foreground_color, &target_background_color);
```

The function `do_picture_trial()` is used to setup and run the picture- presentation trials. The trial bitmap is created by calling `create_image_bitmap()`. The `bitmap_to_backdrop()` function transfers the image bitmap over the link to the tracker PC as a backdrop for gaze cursors (Unlike the previous TEXT template, the image is not saved). The image bitmap is then passed to `bitmap_recording_-trial()`, where it is copied to the display at the proper time. After the trial, be sure to delete the bitmap.

```
char *images[3] = { "images/sacrmeto.jpg", "images/sac_blur.jpg", "images/composite.jpg" };


int do_picture_trial(int num)
{
        SDL_Surface * bitmap;
        int i;

        // This supplies the title at the bottom of the eyetracker display
        eyecmd_printf("record_status_message '%s, TRIAL %d/%d' ", imgname[num-1],num, NTRIALS);

        // Always send a TRIALID message before starting to record.
        // It should contain trial condition data required for analysis.
        eyemsg_printf("TRIALID PIX%d %s", num, imgname[num-1]);

        // TRIAL_VAR_DATA message is recorded for EyeLink Data Viewer analysis
        // It specifies the list of trial variables value for the trial
        // This must be specified within the scope of an individual trial (i.e., after
        // "TRIALID" and before "TRIAL_RESULT")
        eyemsg_printf("!V TRIAL_VAR_DATA %s", imgname[num-1]);

        set_offline_mode();// Must be offline to draw to EyeLink screen

        bitmap = create_image_bitmap(num-1);
        if(!bitmap)
        {
                alert_printf("ERROR: could not create image %d", num);
                return SKIP_TRIAL;
        }

        // NOTE:** THE FOLLOWING TEXT SHOULD NOT APPEAR IN A REAL EXPERIMENT!!!! **
        clear_full_screen_window(target_background_color);
        get_new_font("Arial", 24, 1);
        graphic_printf(window,target_foreground_color, CENTER,
                SCRWIDTH/2, SCRHEIGHT/2, "Sending image to EyeLink...");
        Flip(window);

        // Transfer bitmap to tracker as backdrop for gaze cursors
        bitmap_to_backdrop(bitmap, 0, 0, 0, 0,0, 0,
                BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));

        // record the trial
        i = bitmap_recording_trial(bitmap, 20000L);
        SDL_FreeSurface(bitmap);
        return i;
}
```

17.3.1 Loading pictures and creating composite images

The first two trials of the current template load a normal picture (sacrmeto.jpg) and a blurred one (sac_-blur.jpg) by calling `image_file_bitmap()`. The last trial creates a composite image from four smaller images (hearts.jpg, party.jpg, purses.jpg, and squares.jpg) by using `composite_image()`. This is achieved by first creating a blank bitmap (`blank_bitmap()`) and then loading individual images and adding them to the blank bitmap at a specific (x, y) position with the `add_bitmap()` function.

In this example, the bitmap is magnified or reduced to fill the display. This will degrade the image somewhat and slows loading, so it may be important to match the display mode to the original image file resolution. Alternatively, if the size is set to (0, 0), the picture is not resized but is simply clipped or centered on the display. Alternatively, the image may be loaded to a bitmap of identical size, and this bitmap copied to a blank bitmap to create a composite display with several pictures and text.

Because loading and resizing images can take significant time, this sample code displays a message while the picture is loading. This would not be desirable in a real experiment, of course. It is important to check that the picture loaded successfully before running the trial, in this example an error message is placed in the EDF file and an alert box is displayed (again, this may not be desirable in a real experiment).

```
// Adds individual pieces of source bitmap to create a composite picture
// Top-left placed at (x,y)
// If either the width or height of the display is 0, simply copy the bitmap
// to the display without chaning its size. Otherwise, stretches the bitmap to
// fit the dimensions of the destination display area

int add_bitmap(SDL_Surface * src, SDL_Surface * dst, int x, int y, int width, int height)
{
        SDL_Rect dstrect = {x,y,width,height};
        SDL_Rect srcrect = {0,0,width,height};
        if(src->w != srcrect.w || src->h != srcrect.h)
        {
                double zx = (double)srcrect.w/(double)src->w;
                double zy = (double)srcrect.h/(double)src->h;
                SDL_Surface *resized = zoomSurface(src,zx,zy,0);
                if(resized)
                        src = resized;
        }
        SDL_BlitSurface(src,&srcrect,dst,&dstrect);
        return 0;
}

// Creates a composite bitmap based on individual pieces

SDL_Surface * composite_image(void)
{
        int i = 0;
        // Handle to the background and forground bitmaps
        SDL_Surface* bkgr;
        SDL_Surface* img;

        // Filenames of the four small images
        char *small_images[4]={"images/hearts.jpg", "images/party.jpg",
                "images/squares.jpg", "images/purses.jpg"};

        // The x,y coordinates of the top-left corner of the region where
        // the individual small image is displayed
        SDL_Rect points[4]={{0, 0}, {SCRWIDTH/2, 0}, {0, SCRHEIGHT/2},
                {SCRWIDTH/2, SCRHEIGHT/2}};

        // Create a blank bitmap on which the smaller images are overlaid
        bkgr = SDL_CreateRGBSurface(SDL_SWSURFACE,SCRWIDTH,SCRHEIGHT, dispinfo.bits,0,0,0,0);
        if(!bkgr)
                return NULL;
        SDL_FillRect(bkgr,NULL,SDL_MapRGB(bkgr->format, 128,128,128));

        // loop through four small images
        for (i=0; i<4; i++)
        {
                // Load the small images, keep the original size
                // If the image can not be loaded, delete the created blank bitmap;

                img = image_file_bitmap(small_images[i], 1, 0, 0, 0);
                if(!img)
                {
                        SDL_FreeSurface(bkgr);
                        return NULL;
                }

                // Add the current bitmap to the blank bitmap at x, y position,
                // resizing the bitmap to the specified width and height
                // If the original size is to be kept, set the width and
                // height paremeters to 0

                add_bitmap(img, bkgr, points[i].x, points[i].y, SCRWIDTH/2, SCRHEIGHT/2);


                // IMGLOAD command is recorded for EyeLink Data Viewer analysis
                // It displays a default image on the overlay mode of the trial viewer
                // screen. Writes the image filename + path info
                // The IMGLOAD TOP_LEFT command specifies an image to use as a segment of
                // the spatial overlay view with specific top left x,y coordinates and image
                // width and height
                eyemsg_printf("!V IMGLOAD TOP_LEFT %s %d %d %d %d", small_images[i],
                        points[i].x, points[i].y, SCRWIDTH/2, SCRHEIGHT/2);

                // IAREA command is recorded for EyeLink Data Viewer analysis
                // Another way of handling segment information by recording the content
                // field in IAREA RECTANGLE command. The fields are: segment id, (x, y)
                // coordinate of top-left and bottom-right positions
                eyemsg_printf("!V IAREA RECTANGLE %d %d %d %d %d %s", i+1, points[i].x, points[i].y,
                        points[i].x + SCRWIDTH/2, points[i].y + SCRHEIGHT/2, small_images[i]);

                // Be sure to delete bitmap handle before re-using it.
                SDL_FreeSurface(img);
        }

        // If the operation is successful, the background image is now
        // overlaid with the smaller images
        return bkgr;
}
```

Chapter 18

"EYEDATA" Template

This template introduces the use of the link in transferring gaze-position data. This can be transferred in real time, or played back after recording has ended, which helps to separate recording from analysis.

The *eyedata* template uses real-time eye position data set through the link to display a real-time gaze cursor, using a trial implemented in *data_trial.c*. The data is then played back using the module *playback_trial.c*. The bitmap for the trial is a grid of letters, generated by *grid_bitmap.c*. This module is not discussed further, as it is similar to other bitmap-drawing functions discussed previously. You may wish to read through it as an example of drawing more complex displays using SDL.

18.1 Source Files for "Eyedata"

These are the files used to build *eyedata*. Those that were covered previously are marked with an asterisk.

main.c ∗	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file. This file is unchanged for all templates, and can be used with small changes in your experiments.
data_trials.c	Called to run a block of trials for the *picture* template. Performs system setup at the start of each block, then runs the trials. 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.
data_trial.c	Implements a trial with a real-time gaze cursor, and displays a bitmap.
playback_trial.c	Plays back data from the previous trial. The path of gaze is plotted by a line, and fixations are marked with "F".
grid_bitmap.c	Creates a bitmap, containing a 5 by 5 grid of letters.

18.2 Differences from "Text" and "Picture"

The block loop module (*data_trials.c*) creates a grid bitmap, and runs the real- time gaze trial (*data_trial.c*). This is followed immediately by playback of the trial's data (*playback_trial.c*).

18.3 Analysis of "data_trials.c"

Before the initial setup, the calibration background color is set to dark gray (60,60,60), matching the grid background.

```
        SETCOLOR(target_background_color, 60, 60, 60);
        set_calibration_colors(&target_foreground_color, &target_background_color);
```

Only a single trial is run by calling `do_data_trial()`. This marks the trial, creates the grid bitmap, and calls `realtime_data_trial()` to record data while displaying a gaze cursor. After this, `playback_trial()` plays back the data just recorded, plotting the path of gaze and fixations.

Note the command "`mark_playback_start`" that is sent to the eye tracker just before the "`TRIALID`" message. This command sets the point at which playback of data from the EDF file will start later, and is only available in the newest tracker software (EyeLink I version 2.1 and higher, and EyeLink II versions 1.1 and higher). If this command is not used (and for older tracker software versions), playback begins from the start of recording, and therefore will not include the TRIALID message. Similar to the TEXT template, the display PC bitmap is saved as a .png file in the "images" directory. The bitmap image is also transferred over the link to the tracker PC as a backdrop for gaze cursors.

```
// There is only one data trial. followed by playing back the data.
// Returns trial result code
int do_data_trial(int num)
{
        int i;
        SDL_Surface *bitmap = NULL;

        // This supplies the title at the bottom of the eyetracker display
        eyecmd_printf("record_status_message 'GAZE CURSOR TEST' ");

        // NEW command (EL I v2.1, EL II V1.1)
        // Marks where next playback will begin in file
        // If not used, playback begins from start of recording
        eyecmd_printf("mark_playback_start");

        // Always send a TRIALID message before starting to record.
        // It should contain trial condition data required for analysis.
        eyemsg_printf("TRIALID GRID");

        set_offline_mode();// Must be offline to draw to EyeLink screen
        // TRIAL_VAR_DATA message is recorded for EyeLink Data Viewer analysis
        // It specifies the list of trial variables value for the trial
        // This must be specified within the scope of an individual trial (i.e., after
        // "TRIALID" and before "TRIAL_RESULT")
        eyemsg_printf("!V TRIAL_VAR_DATA EYEDATA");

        // IMGLOAD command is recorded for EyeLink Data Viewer analysis
        // It displays a default image on the overlay mode of the trial viewer screen.
        // Writes the image filename + path info
        eyemsg_printf("!V IMGLOAD FILL images/grid.png");

        // IAREA command is recorded for EyeLink Data Viewer analysis
        // It creates a set of interest areas by reading the segment files
        // Writes segmentation filename + path info
        eyemsg_printf("!V IAREA FILE segments/grid.ias");

        bitmap = draw_grid_to_bitmap_segment("grid.ias", "segments", 1);
        if(!bitmap)
        {
                eyemsg_printf("ERROR: could not create bitmap");
                return SKIP_TRIAL;
        }

        // Save bitmap and transfer to the tracker pc.
        // Since it takes a long to save the bitmap to the file, the
        // value of sv_options should be set as SV_NOREPLACE to save time
        bitmap_save_and_backdrop(bitmap, 0, 0, 0, 0, "grid.png", "images",
                SV_NOREPLACE,0, 0, BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));

        //display gaze cursor during recording
        i = realtime_data_trial(bitmap, 60000L);

        playback_trial();   // Play back trial's data

        SDL_FreeSurface(bitmap);
        return i;
}
```

18.4 Analysis of "data_trial.c"

The *data_trial.c* module uses real-time gaze data to plot a cursor. This is a simple example of using transferred data, which will be expanded in a later template into a gaze-contingent window. Link data is only required during recording for real-time experiments, since playback is preferred for on-line analysis.

18.4.1 Newest Sample Data

As eye movement data arrives from the tracker, the samples and events are stored in a large queue, to prevent loss of data if an application cannot read the data immediately. The *eyelink_core* DLL provides two ways to access link data immediately or in sequence. As will be shown in the playback module, samples and events can be read from the queue in the same order they arrived. However, data read in this way will be substantially delayed if the queue contains significant amounts of data.

When eye position data is required with the lowest possible delay, the newest sample received from the link must be obtained. The EyeLink library keeps a copy of the latest data, which is read by `eyelink_newest_float_sample()`. This function returns 1 if a new sample is available, and 0 if no new samples have been received since the previous call to the function. It can be called with NULL as the buffer address in order to test if a new sample has arrived. The sample buffer can be a structure of type FSAMPLE or ALLF_DATA, defined in *eyetypes.h*.

18.4.2 Starting Recording

When starting recording, we tell the EyeLink tracker to send samples to us through the link, by setting the third argument to `start_recording()` to 1. Data will be available immediately, as `start_recording()` does not return until the first data has arrived. Realtime mode is also set, to minimize delays between reading data and updating the gaze cursor.

```
        initialize_cursor(window, 0);
        error = start_recording(1,1,1,1);       // record with link data enabled
        if(error != 0) return error;        // ERROR: couldn't start recording
        // record for 100 msec before displaying stimulus
```

```
// Windows 2000/XP: no interruptions from now on
begin_realtime_mode(100);
```

When using data from the link queue (which will be introduced in the playback module), some samples and/or events will build up in the data queue during the 100 millisecond delay before image display begins. These will be read later, causing an initial burst of data processing. This does not occur with `eyelink_newest_float_sample()`, as only the latest sample is read. The queue is not read in this example, so data simply builds up in the queue and the oldest data is eventually overwritten.

18.4.3 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 and EyeLink1000 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 function `eyelink_wait_for_block_start()` 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(!eyelink_wait_for_block_start(100, 1, 0)) //wait for link sample data
{
        end_trial();
        alert_printf("ERROR: No link samples received!");
        return TRIAL_ERROR;
}
```

Once the block start has been processed, data on the block is available. 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, `eyelink_eye_available()` 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.

```
// determine which eye(s) are available
eye_used = eyelink_eye_available();
switch(eye_used) // select eye, add annotation to EDF file
{
        case RIGHT_EYE:
                eyemsg_printf("EYE_USED 1 RIGHT");
                break;
        case BINOCULAR:  // both eye's data present: use left eye only
                eye_used = LEFT_EYE;
        case LEFT_EYE:
                eyemsg_printf("EYE_USED 0 LEFT");
                break;
}
```

Additional information on the block data can be accessed by a number of functions: see the reference sections of this manual and the *eyelink.h* file for more information. Sample rates and other data specific to the EyeLink II tracker is available with the new `eyelink2_mode_data()` function.

18.4.4 Reading Samples

Code is added to the recording loop to read and process samples. This calls `eyelink_newest_float_sample()` to read the latest sample. If new data is available, the gaze position for the monitored eye is extracted from the sample, along with the pupil size.

During a blink, pupil size will be zero, and both x and y gaze position components will be the value `MISSING_DATA`. The eye position is undefined during a blink, and this case must be treated carefully for gaze-contingent displays. In this example, the cursor is simply hidden. Otherwise, it is redrawn at the new location.

```
// NEW CODE FOR GAZE CURSOR
if(eyelink_newest_float_sample(NULL)>0)//check for new sample update
{
        eyelink_newest_float_sample(&evt);// get a copy of the sample
        x = evt.fs.gx[eye_used];// get gaze position from sample
        y = evt.fs.gy[eye_used];
        // make sure pupil is present
        if(x!=MISSING_DATA && y!=MISSING_DATA && evt.fs.pa[eye_used]>0)
                draw_gaze_cursor((int)x,(int)y);  // show and move cursor
        else
                erase_gaze_cursor();    // hide cursor if no pupil

}   // END OF RECORDING LOOP
```

18.5 Analysis of "playback_trial.c"

The EyeLink system can supply data on eye movements in real time during recording, via the Ethernet link. The high data rate (binocular, 250 or 500 samples per second for EyeLink II and binocular, 500 or 1000 samples per second for EyeLink 1000) makes data processing or display generation difficult while data is being transferred, and writing data to disk will cause significant delays that will impact stimulus presentation. Instead, data can be written to the EyeLink EDF file as a permanent record, then played back after recording of the trial is finished. This is the best method to implement on-line analysis when information is needed before the next trial, for example to implement convergent threshold paradigms or to detect if a subject fixated outside of a region on the display.

The *playback_trial.c* module also demonstrates processing of samples and events from the link queue. The data-processing code is similar to what would be used inside a recording loop for real-time data, except that we will not lose data if analysis takes a long time or the program is stopped by a debugger. In fact, playback data is sent "on demand" at a rate determined by how fast the data queue is read, so it is feasible to process thousands of samples per second during playback.

18.5.1 Starting Playback

Data from the last trial (or the last recording block if several were recorded) can be played back by calling the function `eyelink_playback_start()`, then waiting for data to arrive by calling `eyelink_wait_for_data()`. If no data arrives, this function will return an error code. The first data received during playback will include messages and button presses written to the file just after the end of the next-to-last recording block (or from the start of the file if the first data block is being played back). All data up to the start of block data can be skipped by calling `eyelink_wait_for_block_start()`, after which information on sample rate and eyes recorded will also be available. Playback will fail if no EDF file is open, or if no data has yet been recorded.

```
// Set up the display
clear_full_screen_window(grey);        // erase display
```

```
        get_new_font("Times Roman", 24, 0);      // title of screen
        graphic_printf(window, black, NONE, SCRWIDTH/2, 24, "Playing back last
                trial...");
        SDL_Flip(window);
        graphic_printf(window, black, NONE, SCRWIDTH/2, 24, "Playing back last
                trial..."); // drawing to the background

        set_offline_mode();           // set up eye tracker for playback
        eyelink_playback_start();     // start data playback

        // Wait for first data to arrive
        // Failure may mean no data or file not open

        // This function discards other data in file (buttons and messages)
        // until the start of recording.
        // If you need these events, then don't use this function.
        // Instead, wait for a sample or event before setting eye_data,
        // and have a timeout if no data is available in 2000 msec.

        if(!eyelink_wait_for_block_start(2000, 1, 1))
        {
                alert_printf("ERROR: playback data did not start!");
                return -1;
        }
```

Remarks:

The `eyelink_wait_for_block_start()` function reads and discards events until the start of the recording is encountered in the data stream. This means that any data before recording (such as the `"TRIALID"` message) will be lost. If you need to read this message or other data before recording begins, do not use this function. Instead, read and process events until the first sample is found, or until a message such as `"DISPLAY ON"` is found. If this data is not found within 100 milliseconds after playback is started, it is likely that the playback failed for some reason. Once a sample is found, `eyelink_eye_available()` and `eyelink2_mode_data()` may be called as described below.

In the same way as in *data_trial.c*, we call `eyelink_eye_available()` to determine which eye's data to use. (We could have used the messages we placed in the EDF file as well, which will be available during playback). We also determine the sample rate by calling `eyelink2_mode_data()`, which returns -1 if extended information is not available - in this case, the EyeLink I sample interval of 4 msec is used.

```
        eye_used = eyelink_eye_available();
        // use left eye if both available
        if(eye_used==BINOCULAR) eye_used = LEFT_EYE;

        // determine sample rate
        i = eyelink2_mode_data(&sample_rate, NULL, NULL, NULL);
        // EyeLink I: sample rate = 4 msec
        if(i==-1 || sample_rate<250) sample_rate = 250;
```

The data received will match that previously recorded in the EDF file. For example, if both samples and events were recorded, both will be sent through the link. Also, the types of events and data selected by EyeLink configuration commands for the file will apply to the playback data, not those selected for real-time link data.

Playback is halted when all data has been read, or when the `eyelink_playback_stop()` function is called. The usual tests for the ESC key and for program termination are performed in the loop.

```
        if(escape_pressed() || break_pressed() || eyelink_last_button_press(NULL))
        {
                eyelink_playback_stop();      // stop playback
                clear_full_screen_window(target_background_color);
                return 0;
        }
```

18.5.2 Processing Link Data

When the EyeLink library receives data from the tracker through the link, it places it in a large data buffer called the queue. This can hold 4 to 10 seconds of data, and delivers samples and events in the order they were received.

A data item can be read from the queue by calling `eyelink_get_next_data()`, which returns a code for the event type. The value `SAMPLE_TYPE` is returned if a sample was read from the queue. Otherwise, an event was read from the queue and a value is returned that identifies the event. The header file *eye_data.h* contains a list of constants to identify the event codes.

If 0 was returned, the data queue was empty. This could mean that all data has been played back, or simply that the link is busy transferring more data. We can use `eyelink_current_mode()` to test if playback is done.

```
// PROCESS PLAYBACK DATA FROM LINK
i = eyelink_get_next_data(NULL);   // check for new data item
if(i==0)                                 // 0: no new data
{                                 // Check if playback has completed
        if((eyelink_current_mode() & IN_PLAYBACK_MODE)==0) break;
}
```

If the item read by `eyelink_get_next_data()` was one we want to process, it can be copied into a buffer by `eyelink_get_float_data()`. This buffer should be a structure of type FSAMPLE for samples, and ALLF_DATA for either samples or events. These types are defined in *eye_data.h*.

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.

18.5.3 Processing Events

In *playback_trial.c*, fixations will be plotted by drawing an ’F’ at the average gaze position. The ENDFIX event is produced at the end of each fixation, and contains the summary data for gaze during the fixation. The event data is read from the `fe` (floating-point eye data) field of the `ALLF_DATA` type, and the average x and y gaze positions are in the `gavx` and `gavy` subfields respectively.

```
if(i == ENDFIX) // Was it a fixation event ?
{// PLOT FIXATIONS
        eyelink_get_float_data(&evt);   // get copy of fixation event
        if(evt.fe.eye == eye_used)        // is it the eye we are plotting?
        {// Print a black "F" at average position
                graphic_printf(window, black, NONE, (int)evt.fe.gavx, (int)evt.fe.gavy, "F");
                SDL_Flip(window);
                graphic_printf(window, black, NONE, (int)evt.fe.gavx, (int)evt.fe.gavy, "F");
        }
}
```

It is important to check which eye produced the ENDFIX event. When recording binocularly, both eyes produce separate ENDFIX events, so we must select those from the eye we are plotting gaze position for. The eye to monitor is determined during processing of samples.

18.5.4 Detecting 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). Versions 2.1 and higher of the library will mark data loss by causing a LOST_DATA_-EVENT to be returned by `eyelink_get_next_data()` at the point in the data stream where data is missing. This event is defined in the latest version of *eye_data.h*, and the `#ifdef` test in the code below ensures compatibility with older versions of this file.

```
#ifdef LOST_DATA_EVENT       // AVAILABLE IN V2.1 OR LATER DLL ONLY
else if(i == LOST_DATA_EVENT)
{
        alert_printf("Lost data in sequence");
}
#endif
```

18.5.5 Processing Samples

Samples are plotted by connecting them with lines to show the path of the subject's gaze on the display. The gaze position for the monitored eye is extracted from the sample, along with the pupil size. During a blink, pupil size will be zero, and both x and y gaze position components will be the value `MISSING_DATA`. Otherwise, we connect the current and last gaze position with a line.

```
else if(i==SAMPLE_TYPE)
{
        eyelink_get_float_data(&evt);  // get copy of sample
        if(eye_used != -1)             // do we know which eye yet?
        {
                msec_delay(1000/sample_rate);  // delay for real-time playback
                x = evt.fs.gx[eye_used];   // get gaze position from sample
                y = evt.fs.gy[eye_used];
                if(x!=MISSING_DATA && y!=MISSING_DATA && evt.fs.pa[eye_used]>0 )   // check if pup
                {
                        // everything ready: connect with line
                        if(last_sam_x != MISSING_DATA)
                        {
                                // draw to the background
                                lineRGBA(window,(Sint16)last_sam_x, (Sint16)last_sam_y, (Sint16)x,
                                SDL_Flip(window);
                                lineRGBA(window,(Sint16)last_sam_x, (Sint16)last_sam_y, (Sint16)x,
                        }
                        // record position for next sample
                        last_sam_x = (int)x;
                        last_sam_y = (int)y;
                }
                else    // no pupil present: must be in blink
                {
                        last_sam_x = MISSING;
                }
        }
        else
        {// if we don't know which eye yet, check which eye present
                eye_used = eyelink_eye_available();
                // use left eye if both available
                if(eye_used==BINOCULAR) eye_used = LEFT_EYE;
        }
}
```

Playback data arrives much more quickly than it was recorded. You can take as long as you want to process each data item during playback, because the EyeLink library controls data flow for you. To approximate the original timing of the data, we set a delay of 1, 2 or 4 milliseconds after each sample - this could be improved by using the timestamp available for each sample.

Chapter 19

"GCWINDOW" Template

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. SDL_util library can be used to draw the gaze cursors. Note that you will have to call initialize_dynamic_cursor to initialize.

19.1 Gaze-Contingent Window Issues

The gaze-contingent window in this template can be used to produce moving- window and masking paradigms. It is not intended for boundary paradigms, however.

19.1.1 Fast Updates

The window code in *gazecursor.c* 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 because the window is updated every 1, 2 or 4 milliseconds.

The first time the window is drawn, both the background and window are completely drawn. This takes much longer than the incremental updates during window motion, and is similar to the time it takes to copy a bitmap to the entire display. This depends on your CPU speed, VGA card, and display mode. If the window is hidden, or drawn after being hidden, it will also take more time to draw. This is also true if the window movement between updates is larger than the window size.

At high display resolutions (where pixels are as small as 0.02°), eye tracker noise and microsaccades can cause "jitter" of the window, which makes the edges of the window more visible. This constant drawing also makes Windows XP less responsive. The gaze contingent window code implements a "deadband" filter to remove the jitter while not adding any effective delay. This filter can be visualized as a washer on a tabletop being moved by a pencil through its hole - the washer only moves when the pencil reaches the sides of the hole, and small motions within the hole are ignored. The deadband filter is set to $0.1°$, which results in a negligible error in window position.

19.1.2 Windowing and Masking

The window can be used for two purposes: to show new information at the fovea, or to mask information from the fovea. When initializing the window, you must specify which operation the window is performing. The window code uses this to optimize its drawing to pass the *minimum foveal information.* This is done by erasing unwanted sections of the window first when showing new information, and by drawing new sections of the window (mask) first when masking foveal information. The effect is most important during large saccades.

19.1.3 Eye Tracker Data Delays

An important issue for gaze-contingent displays is the system delay from an eye movement to a display change. There are three factors affecting this delay: the eye tracker delay, the drawing delay, and the display delay.

The EyeLink tracker delay from an eye movement to data available by `eyelink_newest_sample()` is shown in the table below. This delay includes a time of $\frac{1}{2}$ sample, which is the average delay from an eye movement to the time the camera takes an image. Note that this delay only affects the latency of data available from the link - sample timestamps are still accurate to the time the camera imaged the eye.

It is not recommended to disable the heuristic filter for the EyeLink I tracker, as this will increase likelihood of false saccades in the output data. The EyeLink II and EyeLink1000 tracker has separate link and file filter settings, so the link filter can be disabled without affecting recorded data. The heuristic filter setting is controlled by the `heuristic_filter` command, sent to the tracker by the `eyecmd_printf()` function. (See the reference section of this manual for information on the EyeLink tracker configuration commands). The heuristic filter is automatically re-enabled at the end of each recording session, so needs to be explicitly changed for each trial.

Mode	Delay[1]	Notes
1000 Hz, heuristic filter off	< 2 ms	EyeLink 1000 only, high jitter
1000 Hz, level 1 (standard) heuristic filter	< 3 ms	EyeLink 1000 only
1000 Hz, level 2 (extra) heuristic filter	< 4 ms	EyeLink 1000 only, minimizes jitter
500 Hz, heuristic filter off	3 ms	EyeLink II and 1000, high jitter
500 Hz, level 1 (standard) heuristic filter	5 ms	EyeLink II and 1000
500 Hz, level 2 (extra) heuristic filter	7 ms	EyeLink II and 1000, minimizes jitter
250 Hz, heuristic filter off	6 ms	EyeLink I, heuristic filter off (high jitter) EyeLink II and 1000, heuristic filter off (high jitter)
250 Hz, level 1 (standard) heuristic filter	10 ms	EyeLink I, heuristic filter on EyeLink II and 1000, standard heuristic filter
250 Hz, level 2 (extra) heuristic filter	14 ms	EyeLink II and 1000, minimizes jitter

Remarks:

1. Reports average End to End latency, measured from an actual physical event to availability of first data sample that registered the event on the Display / Subject PC via Ethernet or Analog output in C code. Experiment Builder execution may add an extra 1 or 2 ms delay.

19.1.4 Display Delays

The second component of the delay is the time to draw the new gaze-contingent window, and for the changed image data to appear on the monitor. The drawing delay is the time from new data being read until drawing is completed. This depends on CPU speed, VGA card, and display resolution. This is usually less than 1 millisecond for typical saccades using a 10° gaze-contingent window, as only small sections of the window are erased and redrawn. This delay will be higher for large display changes, such as hiding or re-displaying a large window. In any case, the delay will be less that the full-screen bitmap copy time.

The final delay is caused by the time it takes to read out the image from the VGA card's memory to the monitor. This takes at most one refresh period, ranging from 17 milliseconds (for a 60 Hz display) to 6 milliseconds (for a 160 Hz display). The *gcwindow* sample experiment redraws the window whenever new eye position data are available and does not wait for a display refresh, which reduces the average delay to $\frac{1}{2}$ of a refresh period. This unsynchronized drawing does not cause any issues because only sections of the window that have changed are redrawn - only a few pixels are changed even during saccades.

The average delay for a gaze-contingent display therefore can be as low as 7 milliseconds (for 500 Hz sample rate, no filter, and a 160 Hz refresh rate). Worst-case delays will be higher by $\frac{1}{2}$ refresh period and 1 sample, or 15 milliseconds. This delay has been confirmed using an electronic artificial pupil and a light sensor attached to a monitor.

19.2 Source Files for "GCWindow"

These are the files used to build *gcwindow*. Those that were covered previously are marked with an asterisk.

main.c ∗	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file. This file is unchanged for all templates, and can be used with small changes in your experiments.
trials.c	Called to run a block of trials for the "gcwindow" template. Performs system setup at the start of each block, then runs the trials. Handles standard return codes from trials to allow trial skip, repeat, and experiment abort. This file creates multiple stimuli: text and pictures.
trial.c	Implements a trial with a real-time gaze-contingent window, displayed using two bitmaps.

19.3 Analysis of "trials.c"

There are 5 trials to demonstrate different types of gaze-contingent window conditions, two using text and three using pictures. This requires some care with background colors. The code for each type of trail setup is similar to the modules *trials.c* in text example and *trials.c* in *picture* example.

19.3.1 Setup and Block Loop

The block loop for *trials.c* for the *GCWindow* template is similar to *data_trial.c* from the *eyedata* template, except that the brightness of the calibration background is darker. This allows it to approximately match both the text and picture trials. In an actual experiment, mixing stimuli with extreme differences in background brightness is not optimal.

```
// INITIAL CALIBRATION: matches following trials
// color of calibration target
SETCOLOR(target_foreground_color ,0,0,0);
// background for drift correction
SETCOLOR(target_background_color,200,200,200);
// tell EXPTSPPT the colors
set_calibration_colors(&target_foreground_color, &target_background_color);

if(SCRWIDTH!=800 || SCRHEIGHT!=600)
        alert_printf("Display mode is not 800x600, resizing will slow loading.");
```

We also alert the user if the video mode resolution does not match that of the picture BMP files. This forces the images to be resized, which dramatically increases the time to load them.

19.3.2 Setting Up Trials

Unlink the previous templates, each of the 5 trials has different setup. The setup code for each trials is executed within a `switch()` statement.

For trials 1 and 2, 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 1, the normal text is in the foreground, and the "Xxx" text is outside the window.

```
switch(num)
{            // #1: gaze-contingent text, normal in window, "xx" outside
        case 1:
                // This supplies the title at the bottom of the eyetracker display
                eyecmd_printf("record_status_message 'GC TEXT (WINDOW)' ");
                // Always send a TRIALID message before starting to record.
                // It should contain trial condition data required for analysis.
                eyemsg_printf("TRIALID GCTXTW");
                // TRIAL_VAR_DATA message is recorded for EyeLink Data Viewer analysis
                // It specifies the list of trial variables value for the trial
                // This must be specified within the scope of an individual trial (i.e.,
                // after "TRIALID" and before "TRIAL_RESULT")
                eyemsg_printf("!V TRIAL_VAR_DATA TEXT TEXT MASK");
                // IMGLOAD command is recorded for EyeLink Data Viewer analysis
                // It displays a default image on the overlay mode of the trial viewer
                // screen.  Writes the image filename + path info

                if(create_text_bitmaps(1))
                {
                        eyemsg_printf("ERROR: could not create bitmap");
                        return SKIP_TRIAL;
                }

                eyemsg_printf("!V IMGLOAD FILL images/text.png");

                bitmap_save_and_backdrop(fgbm, 0, 0, 0, 0, "text.png", "images", SV_NOREPLACE,
                                0, 0, BX_NODITHER|BX_GRAYSCALE);

                // Gaze-contingent window, normal text
                i = gc_window_trial(fgbm, bgbm, SCRWIDTH/4, SCRHEIGHT/3, 0, 60000L);
                SDL_FreeSurface(fgbm); fgbm = NULL;
```

```
                        SDL_FreeSurface(bgbm); bgbm = NULL;
                        return i;
```

For trial 2, the bitmaps are reversed, and the window type is set to masking (fifth argument to `gc_window_trial()`).

```
                case 2:    // #2: gaze-contingent text, "xx" in window, normal outside
                        eyecmd_printf("record_status_message 'GC TEXT (MASK)' ");
                        eyemsg_printf("TRIALID GCTXTM");
                        eyemsg_printf("!V TRIAL_VAR_DATA TEXT MASK TEXT");

                        if(create_text_bitmaps(2))
                        {
                                eyemsg_printf("ERROR: could not create bitmap");
                                return SKIP_TRIAL;
                        }

                        eyemsg_printf("!V IMGLOAD FILL images/text.png");

                        bitmap_save_and_backdrop(fgbm, 0, 0, 0, 0, "text.png", "images", SV_NOREPLACE,
                                        0, 0, BX_NODITHER|BX_GRAYSCALE);

                        // Gaze-contingent window, masked text
                        i = gc_window_trial(bgbm, fgbm, SCRWIDTH/4, SCRHEIGHT/3, 1, 60000L);
                        SDL_FreeSurface(fgbm); fgbm = NULL;
                        SDL_FreeSurface(bgbm); bgbm = NULL;
                        return i;
```

For trials 3, 4, and 5, a mixture of pictures and blank bitmaps are used. For trial 3, the background bitmap is blank, and the foreground image is a picture (a peripheral mask).

```
                case 3:
                        eyecmd_printf("record_status_message 'GC IMAGE (WINDOW)' ");
                        eyemsg_printf("TRIALID GCTXTW");
                        eyemsg_printf("!V TRIAL_VAR_DATA IMAGE IMAGE MASK");

                        if(create_image_bitmaps(0))
                        {
                                eyemsg_printf("ERROR: could not create bitmap");
                                return SKIP_TRIAL;
                        }

                        bitmap_to_backdrop(fgbm, 0, 0, 0, 0, 0, 0,
                                BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));

                        eyemsg_printf("!V IMGLOAD FILL images/sacrmeto.jpg");

                        // Gaze-contingent window, normal image
                        i = gc_window_trial(fgbm, bgbm, SCRWIDTH/4, SCRHEIGHT/3, 0, 60000L);
                        SDL_FreeSurface(fgbm); fgbm = NULL;
                        SDL_FreeSurface(bgbm); bgbm = NULL;
                        return i;
```

For trial 4, the foreground bitmap is a picture, and the background bitmap is the picture (a foveal mask).

```
                case 4:    // #4: Image, blank in window, normal outside
                        eyecmd_printf("record_status_message 'GC IMAGE (MASK)' ");
                        eyemsg_printf("TRIALID GCTXTM");
                        eyemsg_printf("!V TRIAL_VAR_DATA IMAGE MASK IMAGE");

                        if(create_image_bitmaps(1))
                        {
                                eyemsg_printf("ERROR: could not create bitmap");
                                return SKIP_TRIAL;
                        }

                        eyemsg_printf("!V IMGLOAD FILL images/sacrmeto.jpg");

                        bitmap_to_backdrop(bgbm, 0, 0, 0, 0, 0, 0,
                                BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));

                        // Gaze-contingent window, masked image
                        i = gc_window_trial(fgbm, bgbm, SCRWIDTH/4, SCRHEIGHT/3, 1, 60000L);
                        SDL_FreeSurface(fgbm); fgbm = NULL;
                        SDL_FreeSurface(bgbm); bgbm = NULL;
                        return i;
```

In trial 5, the foreground bitmap is a clear image and the background is a blurred image of the same picture (a simple variable-resolution display).

```
                case 5:    // #5: Image, blurred outside window
                        eyecmd_printf("record_status_message 'GC IMAGE (BLURRED)' ");
                        eyemsg_printf("TRIALID GCTXTB");
                        eyemsg_printf("!V TRIAL_VAR_DATA IMAGE IMAGE BLURRED");

                        if(create_image_bitmaps(2))
                        {
                                eyemsg_printf("ERROR: could not create bitmap");
                                return SKIP_TRIAL;
                        }

                        eyemsg_printf("!V IMGLOAD FILL images/sacrmeto.jpg");

                        bitmap_to_backdrop(fgbm, 0, 0, 0, 0, 0, 0,
                                BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));

                        // Gaze-contingent window, masked image
                        i = gc_window_trial(fgbm, bgbm, SCRWIDTH/4, SCRHEIGHT/3, 0, 60000L);
                        SDL_FreeSurface(fgbm); fgbm = NULL;
                        SDL_FreeSurface(bgbm); bgbm = NULL;
                        return i;
```

The function `create_image_bitmaps()` creates the proper set of bitmaps for each trial, and checks that the bitmaps loaded properly. It also generates the EyeLink graphics, and sets the background color for the drift correction to match the images.

```
        switch(type)
        {
                case 0:     // blank background
                {
                        fgbm =image_file_bitmap("images/sacrmeto.jpg",0,SCRWIDTH,SCRHEIGHT,1);
                        bgbm =blank_bitmap(target_background_color, 0);

                        break;
                }
                case 1:     // blank fovea
                        fgbm = blank_bitmap(target_background_color,1);
                        bgbm = image_file_bitmap("images/sacrmeto.jpg", 0, SCRWIDTH,SCRHEIGHT,0);
                        break;
                case 2:     // normal and blurred bitmaps, stretched to fit display
                        fgbm = image_file_bitmap("images/sacrmeto.jpg", 0, SCRWIDTH,SCRHEIGHT,1);
                        bgbm = image_file_bitmap("images/sac_blur.jpg", 0, SCRWIDTH,SCRHEIGHT,0);
                        break;
        }
```

For trials 3 and 5, the window contains the maximum information, and the window masking type is set to 0 (erase before draw). In trial 4, the foreground bitmap is blank, and the window masking type is set to 1 (draw before erase). The masking type is set by the fifth argument to `gc_window_trial()`.

19.3.3 Drawing the Window

The code for moving the gaze-contingent window is similar to that used to move the gaze cursor in the *eyedata* template. However, we don't copy any bitmap to the display. The function `redraw_gc_window()` will draw the background the first time it is called. We place code here to accurately mark the time of the display onset, in case reaction time needs to be determined. There are two ways to handle a blink. The preferred way is to freeze the gaze-contingent window by not calling `redraw_gc_window()` during the blink. Another possibility is to erase the window, but this takes longer to draw and causes distracting flickering, and is not recommended.

```
// NEW CODE FOR GAZE CONTINGENT WINDOW
if(eyelink_newest_float_sample(NULL)>0)  // check for new sample update
{
        eyelink_newest_float_sample(&evt);   // get the sample
        x = evt.fs.gx[eye_used];    // yes: get gaze position from sample
        y = evt.fs.gy[eye_used];

        if(x!=MISSING_DATA && y!=MISSING_DATA && evt.fs.pa[eye_used]>0)
        // make sure pupil is present
        {

                // mark display start AFTER first drawing of window
                if(first_display)
                {
                        drawing_time = current_msec();  // time of retrace
                        trial_start = drawing_time;   // record the display onset time
                }

                draw_gaze_cursor((int)x, (int)y);  // move window if visible

                // mark display start AFTER first drawing of window
                if(first_display)
                {
                        first_display = 0;

                        // delay from retrace
                        drawing_time = current_msec() - drawing_time;
                        // message for RT recording in analysis
                        eyemsg_printf("%d DISPLAY ON", drawing_time);
                        // message marks zero-plot time for EDFVIEW
                        eyemsg_printf("SYNCTIME %d", drawing_time);
                }
        }
        else
        {
                // Don't move window during blink
        }
}
```

19.4 Other Gaze-contingent Paradigms

19.4.1 Saccade Contingent Displays

Although the EyeLink system makes start-of-saccade and end-of-saccade events available through the link, these will not usually be used for gaze-contingent displays, unless the display change is to occur at a significant delay after the saccade ends. The EyeLink on-line saccade detector uses a velocity-detection algorithm, which adds 4 to 6 samples (16 to 24 msec) delay to the production of the event, in addition to the regular (3 to 14 msec) sample delay from an eye movement. The timestamps in saccade events are modified to indicate the true time of the eye movement. As well, saccade size data is only available in the end-of-saccade event, which is available after the saccade ends.

For a gaze-contingent saccade detector, we recommend the use of a position- based algorithm computed from sample data. To detect the start of a saccade, compute the difference between the position of the eye in a sample and the average position of the last 6 samples. Do this separately for X and Y directions, then sum the absolute value of the differences and compare to a threshold of $0.3°$ to $0.6°$. This will detect saccade onset with a delay of 2 to 4 samples in addition to the sample data delay, and is most sensitive to large ($> 4°$) saccades. This delay is largely due to the time it takes the eye to move a significant distance at the beginning of a saccade, so the detector will usually trip before the eye has moved more than $1°$.

19.4.2 Boundary Paradigms

In a boundary paradigm, all or part of the display changes depending on where the locus of gaze is. For example, a line of text may change when the reader proceeds past a critical word in the sentence.

The best way to implement this is to modify the display when at least two samples occur within a region (in the reading case, the right side of the display). The display change would be made by copying a bitmap to the display. To keep the delay low, only part of the display should be copied from the bitmap, in this case the line of text. The area that can be changed to meet a given delay depends on your VGA card, CPU speed, and the display mode.

Chapter 20

"CONTROL" Template

This template implements a computer interface that is controlled by the subject's gaze. The subject can select one of a grid of letters by fixating on it. The template contains code to support many rectangular selection regions, but can be simplified if gaze in a single region is all that needs to be detected.

The "Control" template is implemented using the module *trial.c*. The bitmap for the trial is a grid of letters, generated by *grid_bitmap.c*, which is not covered in this manual, as it is similar to other bitmap-drawing functions discussed previously.

20.1 Source Files for "Control"

These are the files used to build "Control". Those that were covered previously are marked with an asterisk.

main.c ∗	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file.
trials.c	Called to run the trial: just calls the trial, and so is not analyzed in this manual.
trial.c	Creates and displays a grid of letters and does the recording.
regions.c	Implements a gaze-controlled interface: sets up an array of gaze-activated regions and performs gaze-activated selection.
grid_bitmap.c ∗	Creates a bitmap, containing a 5 by 5 grid of letters.

20.2 Analysis of "trial.c"

This module implements a computer interface controlled by the subject's gaze. A grid of letters is displayed to the subject, each of which can be selected by fixating it for at least 700 milliseconds. This demonstration also performs "dynamic recentering", where drift correction is performed concurrently with the selection process.

The type of gaze control in this example is intended for multiple gaze-selection regions, and long dwell threshold times. There is a substantial amount of extra code added to the module that supports this. If your task is to detect when gaze falls within one or two well-separated areas of the display, then there are simpler methods than that discussed here. For example, a saccade to a target can be detected by waiting for 5 to 10 samples where gaze position is within 2° of the target. This is not suitable for detecting if gaze leaves an area, however.

20.2.1 Fixation Update Events

A common use for real-time link data is to monitor eye position, to ensure the subject's gaze stays within a prescribed location throughout a trial. The obvious method of checking that each gaze-position sample falls within the prescribed region may not work, because blinks may cause gaze position to change rapidly, giving a false indication. Monitoring average gaze position during fixations is more reliable, as this excludes data from blinks and saccades, but this data is not available until the end of the fixation.

The EyeLink tracker implements fixation update (`FIXUPDATE`) events, which report the average gaze position during a period of a fixation. By configuring the tracker to produce a FIXUPDATE event every 50 to 100 milliseconds, fixation position is monitored in an efficient and timely manner. By looking at events instead of samples, the amount of processing required is also reduced. Blinks also are automatically excluded, and the sum of the time of fixation events in a region represent the true time of continuous gaze.

20.2.2 Enabling Fixation Updates

By default, fixation updates are disabled. Commands must be sent to enable fixation updates before recording is started, and to disable them afterwards. This code produces fixation updates every 50 milliseconds, and enables only FIXUPDATE events to be sent by the link (other events may be enabled if desired as well):

```
        // Configure EyeLink to send fixation updates every 50 msec
        eyecmd_printf("link_event_filter = LEFT,RIGHT,FIXUPDATE");
        eyecmd_printf("fixation_update_interval = 50");
        eyecmd_printf("fixation_update_accumulate = 50");

        init_regions(); // Initialize regions for this display

        // Start data recording to EDF file, BEFORE DISPLAYING STIMULUS
        // You should always start recording 50-100 msec before required
        // otherwise you may lose a few msec of data

        error = start_recording(1,1,0,1);   // send events only through link
```

Commands are added to the `end_trial()` function to disable fixation updates:

```
// End recording: adds 100 msec of data to catch final events
static void end_trial(void)
{
        clear_full_screen_window(target_background_color);    // hide display
        end_realtime_mode();   // NEW: ensure we release realtime lock
        pump_delay(100);       // CHANGED: allow Windows to clean up while
                // we record additional 100 msec of data
        stop_recording();
        eyecmd_printf("link_event_filter = LEFT,RIGHT,FIXATION");
        eyecmd_printf("fixation_update_interval = 0");
        eyecmd_printf("fixation_update_accumulate = 0");
}
```

20.2.3 Processing Fixation Updates

The code to read FIXUPDATE events for the link is similar to that used in *playback_trial.c*, except that only events are processed:

```
        // GET FIXUPDATE EVENTS, PROCESS

        i = eyelink_get_next_data(NULL);    // Check for data from link
        if(i == FIXUPDATE)        // only process FIXUPDATE events
        {
                // get a copy of the FIXUPDATE event
                eyelink_get_float_data(&evt);
                // only process if it's from the desired eye?
                if(evt.fe.eye == eye_used)
                {
                        // get average position and duration of the update, and
                        // Process event
                        process_fixupdate((int)(evt.fe.gavx), (int)(evt.fe.gavy),
                                evt.fe.entime-evt.fe.sttime);
                }
        }
```

The function `process_fixupdate()` is passed the average x and y gaze data and the total time accumulated during the fixation update, which may vary. Each event will be processed to determine which letter's region it falls within. The time of consecutive fixation updates in a region is accumulated into the region's `dwell` field. If a fixation update does not fall into the currently accumulating region, it is assumed that gaze has shifted and the total time in all regions is reset to zero. To prevent noise or drift from inadvertently interrupting a good fixation, the first event in a new region is discarded.

Once the `dwell` time in a region exceeds the threshold, the letter is selected by inverting its `rdraw` region. A drift correction is performed, based on the difference between the average fixation position during selection, and the center of the selection region. This assumes that the visual center of the stimulus in the region is at the location set by the `cx` and `cy` fields of the region's data. Each drift correction may cause a jump in eye-position data, which can produce a false saccade in the eye-movement record.

```
// Process a fixation-update event:
// Detect and handle a switch between regions
// Otherwise, accumulate time and position in region
// Trigger region when time exceeds threshold
void process_fixupdate(int x, int y, long dur)
{
        int avgx, avgy;
        int i = which_region(x, y);     // which region is gaze in

        if(i == -1)                 // NOT IN ANY REGION:
        {// allow one update outside of a region before resetting
                if(last_region == -1)   // 2 in a row: reset all regions
                {
                        reset_regions();
                }
        }
        else if(i == current_region)       // STILL IN CURRENT REGION
        {
                rgn[i].dwell += dur;           // accumulate time, position
                rgn[i].avgxsum += dur * x;
                rgn[i].avgysum += dur * y;

                // did this region trigger yet?
                if(rgn[i].dwell>dwell_threshold && !rgn[i].triggered)
                {
                        trigger_region(i);      // TRIGGERED:
                        // compute avg. gaze position
                        avgx = rgn[i].avgxsum / rgn[i].dwell;
                        avgy = rgn[i].avgysum / rgn[i].dwell;
                        // correct for drift (may cause false saccade in data)
                        eyecmd_printf("drift_correction %ld %ld %ld %ld",
                                (long)rgn[i].cx-avgx, (long)rgn[i].cy-avgy,
                                (long)rgn[i].cx, (long)rgn[i].cy);
                        // Log triggering to EDF file
                        eyemsg_printf("TRIGGER %d %ld %ld %ld %ld %ld",
                                i, avgx, avgy, rgn[i].cx, rgn[i].cy, rgn[i].dwell);
                }
        }
        // TWO UPDATES OUTSIDE OF CURRENT REGION: SWITCH TO NEW REGION
        else if(i == last_region)
        {
                reset_regions();              // clear and initialize accumulators
                rgn[i].dwell = dur;
                rgn[i].avgxsum = dur * x;
                rgn[i].avgysum = dur * y;
                current_region = i;           // now working with new region
        }
        last_region = i;
}
```

20.2.4 Multiple Selection Region Support

Each selection region is defined by a `REGION` structure. This defines the rectangular area where it senses gaze, a rectangular area to highlight when selected, and the expected gaze position during selection for use in drift correction. It also contains accumulators for dwell time and gaze position.

```
// CONTROL REGION DEFINITION
typedef struct {
                int triggered;  // is triggered
                long avgxsum;   // average position accumulators
                long avgysum;
                long dwell;     // total time in region
                SDL_Rect rsense;        // region for gaze
                SDL_Rect rdraw; // rectangle for invert
                int cx, cy;     // center for drift correction
        } REGION;
REGION rgn[NREGIONS];
```

The selection regions are created by `init_regions()`, which creates regions to match the display.

```
void SDL_SetRect(SDL_Rect *rect, int x, int y, int w, int h)
{
        rect->x = x-w;
        rect->y = y-h;
        rect->w = w+w;
        rect->h = h+h;
}

int SDL_PointInRect(SDL_Rect *rect, int x, int y)
{
        if(x >= rect->x  && x <=(rect->x +rect->w)
                && y>= rect->y &&  y <= (rect->y + rect->h) )
                return 1;
        return 0;
}

// Initial setup of region data
void init_regions(void)
{
        int i;
        int x,y;

        for(i=0;i<NREGIONS;i++)    // For all regions:
        {
                // compute center of region
                x = (i%5) * (SCRWIDTH / 5) + (SCRWIDTH / 10);
                y = (i/5) * (SCRHEIGHT / 5) + (SCRHEIGHT / 10);
                rgn[i].cx = x;   // record center for drift correction
                rgn[i].cy = y;
                SDL_SetRect(&(rgn[i].rdraw), x,y,SCRWIDTH/30,SCRHEIGHT/22);
                SDL_SetRect(&(rgn[i].rsense), x,y,SCRWIDTH/10,SCRHEIGHT/10);
        }
}
```

When a fixation update event arrives, its gaze position is checked against all selection regions:

```
// Determine which region gaze is in
// return 0-24 for a valid region, -1 if not in any region
int which_region(int x, int y)
{
        int i;
        for(i=0;i<NREGIONS;i++)      // scan all regions for gaze position match
                if(SDL_PointInRect(&(rgn[i].rsense), x,y)) return i;

        return -1;                   // not in any region
}
```

Finally, a mechanism is needed to update the highlighted region on the display. In this example, the highlight stays on the selected region. It may be more ergonomic to simply flash the region momentarily once selected.

```
void trigger_region(int region)
{
        int i;

        for(i=0;i<NREGIONS;i++)   // scan thru all regions
        {
                if(i==region)      // is this the new region?
                {
                        if(rgn[i].triggered==0)       // highlight new region
                                invert_rect(&(rgn[i].rdraw), 1, i);
                        rgn[i].triggered = 1;
                }
                else
                {
                        if(rgn[i].triggered)            // unhighlight old region
                                invert_rect(&(rgn[i].rdraw), 0, i);
                        rgn[i].triggered = 0;
                }
        }
}
```

Chapter 21

"DYNAMIC" Template

The *dynamic* sample experiment is moderately complex, but very powerful. It uses refresh-locked drawing to present dynamic, real-time displays, including sinusoidal smooth pursuit and a saccade performance task. The display timing achieved in both of these is extremely precise, and the source code was written so that the programmer does not need to modify any time-critical drawing code for most uses. In addition, this template demonstrates one-dimensional calibration which results in greater accuracy and faster setup, and shows how to seamlessly integrate drift correction into the task.

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 800 `x` 600 or even 640 `x` 480 to reach the highest refresh rates, depending on your monitor and display card and driver. Higher refresh rates mean smoother motion during smooth pursuit, and better temporal resolution for displaying stimuli.

The code consists of several new modules. The code that is specific to experiments is largely isolated in one file, *trials.c*, which contains the usual `do_trials()` function with the block-of-trials loop, and calls `do_dynamic_trial()` to set up trial identifier and call the proper execution functions. These are `do_sine_trial()` which executes a sinusoidal pursuit trial, and `do_saccadic_trial()` which executes a saccadic trial using the gap- step-overlap paradigm.

Each of these functions then sets up a number of variables to define the trial, and calls `run_dynamic_-trial()`, supplying a background bitmap and a pointer to a trial-specific "callback" function. This function is called by `run_dynamic_trial()` after each vertical retrace, to handle computing of target positions and visibility, update the target position and visibility, and send any required messages to the EDF file. These callback functions are fairly simple, because almost all the work is handled by target-drawing code in *targets.c* and the realtime trial loop in *trial.c*.

21.1 Source Files for "Dynamic"

These are the files used to build *dynamic*. Those that were covered previously are marked with an asterisk.

main.c *	`WinMain()` function for windows non console compilation and `main()` for other compilations, setup and shutdown link and graphics, open EDF file. This file is unchanged for all templates, and can be used with small changes in your experiments.
trials.c	Called to run the trials, and implements setup and callback functions for saccadic and smooth pursuit trials. Implements 1-D calibration.
trial.c	Implements a real-time, refresh locked trial loop. This includes monitoring of delayed or missed retrace to ensure the quality of data from the experiment.
targets.c targets.h	Creates a set of targets, which can have one of 3 shapes or can be hidden. Multiple targets may be visible at the same time.

21.2 Analysis of "targets.c"

This module implements a complete target graphics system, which can draw, erase, and move multiple targets. This code will not be examined in detail, as it should not have to be changed except to create new target sizes and styles. A set of target "shapes" are created by `create_shape()` and `initialize_targets()`, with shape 0 always being invisible. The set of available shapes may be expanded by redefining NSHAPES and adding code to the switch() function in `create_shape()` to draw the foreground of the new target image.

Target shapes consist of a small bitmap, which has the target image on a rectangular background. This entire bitmap, including the background, is copied to the display when drawing the target. Since the target background will be drawn it should match the background on which the targets will be displayed. The foreground color of the targets will usually be white or red (the red phosphor of most VGA monitors has very low persistence, which virtually eliminates the "trail" left by moving targets).

Targets are erased by copying sections of a background bitmap to the display, which should be assigned to the variable `target_background_bitmap`. This bitmap is also copied to the display by `run_dynamic_trial()` before drift correction. Usually the background bitmap will be blanked to `target_background_color`, but it may optionally contain cues or patterns.

These are the useful functions in targets.c, extracted from targets.h:

```
extern SDL_Surface target_background_bitmap;   // bitmap used to erase targets

// Create the target patterns
// set all targets as not drawn, pattern 0
// redefine as eeded for your targets
int initialize_targets(SDL_Color fgcolor, SDL_Color bgcolor)

// clean up targets after trial
void free_targets(void)

// draw target n, centered at x, y, using shape
// will only draw if target is erased
void draw_target(int n, int x, int y, int shape)

// erase target by copying back background bitmap
// will only erase if target is drawn
void erase_target(int n)

// call after screen erased so targets know they're not visible
// this will permit them to be redrawn
void target_reset(void)

// handles moving target, changing shape
// target will be hidden it if shape = 0
void move_target(int n, int x, int y, int shape)
```

21.2.1 Modifying Target Shapes

The only modifications you should do to *targets.c* are to add or change the size or shape of targets - any other changes could cause the code to stop working properly. These changes are made to the function `create_shape()`, and should be limited to the sections of code shown below.

The modifiable section of `create_shape()` consists of two `switch()` statements that set the size of the shapes and draw the foreground parts of the target shape bitmaps:

```
switch(n)          // set size of target
{
        case 0:        // invible target
        case 1:        // filled circle
        case 2:        // "\\" line
        case 3:        // "/" line
        default:
                width  = (SCRWIDTH/30.0)*0.5;   // all targets are 0.5 by 0.5 degrees
                height = (SCRHEIGHT/22.5)*0.5;
                break;
}
```

In the example, all shapes have a size of 0.5 by 0.5 degrees, as computed for a 30° display width (distance between display and subject is twice the display width). You can change the values assigned to `height` and `width` to change the bitmaps sizes, and these should be referenced when drawing the shape graphics to auto-size them to the bitmaps.

The second section of code draws the graphics for the target - the bitmap has already been created and cleared to the background color.

```
SDL_FillRect(hbm,NULL,SDL_MapRGB(hbm->format,bgcolor.r,  bgcolor.g, bgcolor.b));
switch(n)                       // draw the target bitmap
{
        case 0:        // invisible target
        default:
                break;
        case 1:        // filled circle
                // draw filled ellipse
                filledCircleRGBA(hbm, (Sint16) (width/2), (Sint16)(height/2),
                        (Sint16)(min(width/2, height/2)-1),
                        fgcolor.r,fgcolor.g,fgcolor.b,255);
                break;
        case 2:         // "\" line
                aalineRGBA(hbm,0,2,(Sint16)(width-2), (Sint16)height, fgcolor.r,
                        fgcolor.g, fgcolor.b,255);
                aalineRGBA(hbm,0,1,(Sint16)(width-1), (Sint16)height, fgcolor.r,
                        fgcolor.g, fgcolor.b,255);
                aalineRGBA(hbm,0,0,(Sint16)width,(Sint16)height, fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                aalineRGBA(hbm,1,0,(Sint16)width,(Sint16)(height-1), fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                aalineRGBA(hbm,2,0,(Sint16)width,(Sint16)(height-2), fgcolor.r,
                        fgcolor.g, fgcolor.b,255);
                break;
        case 3:         // "/" line
                aalineRGBA(hbm,0,(Sint16)(height-2),(Sint16)(width-2), 0, fgcolor.r,
                        fgcolor.g, fgcolor.b,255);
                aalineRGBA(hbm,0,(Sint16)(height-1),(Sint16)(width-1), 0, fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                aalineRGBA(hbm,0,(Sint16)height,(Sint16)width,0 ,fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                aalineRGBA(hbm,1,(Sint16)height,(Sint16)width,1,fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                aalineRGBA(hbm,2,(Sint16)height,(Sint16)width,2,fgcolor.r,
                        fgcolor.g,fgcolor.b,255);
                break;
}
```

Changes should be limited to adding new cases, and increasing the number of shapes by redefining the constant `NSHAPES`. Shape 0 will never be drawn, and should be left blank. The sample shapes draw a filled ellipse and two angles thick lines. Note that line width adapts to the display resolution to keep constant visibility at higher resolutions. Graphics should, if possible, be drawn in `fgcolor` so that your experiments can easily set the color to match experimental requirements.

21.3 Analysis of "trial.c"

This module implements a retrace-locked drawing trial in the function `run_dynamic_trial()`. The only reason to modify this code would be to remove the ability to terminate by a button press (which was included to allow skipping long pursuit trials), or to modify or remove drift correction (this is NOT recommended, as saccadic and pursuit tasks are typically run in non-CR modes which require drift correction before each trial).

The trial function requires three arguments:

```
// first argument is background bitmap for targets
// third argument ispointer to drawing function:
//      int drawfn(UINT32 t, int *x, int *y)
// where t = time from trial start in msec (-1 for initial target)
// x, y are pointers to integer to hold a reference position
// which is usually center or fixation target X,Y for drift correction
// this function is called immediately after refresh,
// and must erase and draw targets and write out any messages
// this function returns 0 to continue, 1 to end trial

int run_dynamic_trial(SDL_Surface* hbm, UINT32 time_limit,
        int (__cdecl * drawfn)(UINT32 t, UINT32 dt, int *x, int *y))
```

The first argument is a background bitmap, which will be copied to the display before drift correction, and over which the targets will be drawn (including the drift correction target). The second argument is the trial timeout value in milliseconds (this can be set to a very large number, and the drawing function can be used to determine timeout accurately). The third function is a pointer to a function that will handle retrace-locked drawing, sending messages, and ending the trial - this will be discussed later.

The code of this trial function has been modified in a number of ways. First, the drift correction loop now draws its own background and target, which causes the drift correction to be integrated with the initial fixation of the trial. The background is drawn by displaying the background bitmap - this should not contain stimuli, but might contain static graphics that are visible throughout the trial. After redrawing the whole display, `reset_targets()` must be called so that targets will know that they are erased and may be redrawn later. Finally, the drawing function is called with a time of 0 to display the initial fixation target. The coordinates of this target are placed in the variables `x` and `y`, and used to inform the eye tracker of the drift correction target.

Note that this redrawing is done within a loop, so it will be repeated if setup was done instead of drift correction, as this will clear the display.

```
// DO PRE-TRIAL DRIFT CORRECTION
// We repeat if ESC key pressed to do setup.
// we predraw target for drift correction in this example

while(1)
{       // Check link often so we can exit if tracker stopped
        if(!eyelink_is_connected()) return ABORT_EXPT;
                                // (re) draw display and target at starting point
        SDL_BlitSurface(hbm, NULL, window, NULL);
        Flip(window);
        SDL_BlitSurface(hbm, NULL, window, NULL);
        target_reset();
        drawfn(0, 0, &x, &y);

        // We drift correct at the current target location
        // 3rd argument is 0 because we already drew the display

        error = do_drift_correct((INT16)x, (INT16)y, 0, 1);
        // repeat if ESC was pressed to access Setup menu
        if(error!=27) break;
}
```

Unlike the previous examples, special code was added to mark the time of the first retrace in the trial as trial start.

```
if(trial_start == 0)     // is this the first time we draw?
{
        trial_start = t;     // record the display onset time
        drawfn(t, 0, &x, &y);
        // message for RT recording in analysis
        eyemsg_printf("%d DISPLAY ON", 0);
        // message marks zero-plot time for EDFVIEW
        eyemsg_printf("SYNCTIME %d", 0);
}
else   // not first: just do drawing
{
        if(drawfn(t, t-trial_start, &x, &y))
        {
                eyemsg_printf("TIMEOUT");// message to log the timeout
                end_trial();        // local function to stop recording
                button = 0;      // trial result message is 0 if timeout
                break;          // exit trial loop
        }
}
```

Next, the drawing function is called. This is supplied with the time of retrace (used to determine message delays), the time from the trial start (used to determine time in the drawing sequence), and pointers to variables to hold the primary fixation target position (these are only used for drift correction). For the first call after the start of the trial, the time of trial start is set to the time of the first retrace, and messages are placed in the EDF file. As before, messages are only written after drawing is finished, and the delay of the message from the retrace is appended.

If the drawing function returned with 1, the trial will immediately end with a timeout. This allows trial end to be precisely synchronized with stimulus sequences.

21.4 Analysis of "trials.c"

This is the module where almost all of the experiment is defined and implemented, and where you will make the changes required to adapt the template for your own experiments. This module contains the block-of-trials loop in `run_trials()`, and the trial setup and selector in `do_trial()`, which are similar to those in other experiments.

One important difference is that this template uses horizontal-only calibration, which requires only 4 fixations which is ideal for use with neurologically impaired subjects. The following code is added to `run_trials()` before tracker setup, which sets the vertical position for calibration, and sets the calibration type. After this, vertical position of the gaze data will be locked to this vertical position, which should match the vertical position of your stimuli:

```
target_background_color.r=target_background_color.g=target_background_color.b=0;
set_calibration_colors(&target_foreground_color, &target_background_color);

// TRIAL_VAR_LABELS message is recorded for EyeLink Data Viewer analysis
// It specifies the list of trial variables for the trial.  This should be
// written once only and put before the recording of individual trials
eyemsg_printf("TRIAL_VAR_LABELS TRIAL TYPE DIRECTION");

// SET UP FOR HORIZONTAL-ONLY CALIBRATION
eyecmd_printf("horizontal_target_y = %d", SCRHEIGHT/2); //vertical position

eyecmd_printf("calibration_type = H3");          // Setup calibration type
```

The only differences is that `run_trials()` reports any delayed retrace drawing events after the block is finished (if your experiment has multiple blocks, this should be moved to the end of the experiment). The trial dispatch function `do_trial()` also uses lookup tables for trial titles and TRIALID messages.

Each type of trial is implemented by a trial setup function and a drawing function. The setup function sets up variables used by the drawing function, creates targets and the background bitmap, and calls `run_dynamic_trial()`. The drawing function is then called every vertical retrace by `run_dynamic_trial()`, and performs drawing based on the time from the start of trial, places messages in the EDF file to record significant drawing events for analysis, and determines if the end of the trial has been reached.

21.4.1 Saccadic Trial Setup

The function `do_saccadic_trial()` sets up for each trial in a gap-step-overlap paradigm. The first step is to create a blank background bitmap and initialize the target shapes (if these do not change, this could be done once for the entire experiment, instead of for each trial). Next, the interval between the saccadic goal target appearing and the fixation target disappearing is computed - if negative, the fixation target disappears before the goal target onset. The position of the fixation and goal targets are next calculated, adapting to display resolution to cause a saccade of the desired amplitude to the left or right. The last step of the setup is to draw reference graphics to the eye tracker display - in this case, a box at the position of each target.

```
//********** SACCADIC GAP/STEP/OVERLAP **************

INT32 overlap_interval = 200;     // gap <0, overlap >0
UINT32 prestep_delay = 600;       // time from trial start to move
UINT32 trial_duration = 2000;     // total trial duration

int fixation_x;    // position of initial fixation  target
int fixation_y;
int goal_x;        // position of final saccade target
int goal_y;

int fixation_visible;  // used to detect changes
int goal_visible;

//********** SACCADIC TRIAL SETUP AND RUN *********

// run saccadic trial
// gso = -1 for gap, 0 for step, 1 for overlap
// dir = 0 for left, 1 for right
```

```
int do_saccadic_trial(int gso, int dir)
{
        int i;
        SDL_Color target_color = { 200,200,200};
                                 // blank background bitmap
        SDL_Surface* background;

        background = blank_bitmap(target_background_color);
        if(!background) return TRIAL_ERROR;
                                 // white targets (for visibility)
        initialize_targets(target_color, target_background_color);

        overlap_interval = 200 * gso;  // gap <0, overlap >0

        fixation_x = SCRWIDTH/2;    // position of initial fixation  target
        fixation_y = SCRHEIGHT/2;
                                                        // position of goal (10 deg. saccade)
        goal_x = fixation_x + (dir ? SCRWIDTH/3 : -SCRWIDTH/3);
        goal_y = fixation_y;

        set_offline_mode();     // Must be offline to draw to EyeLink screen
        eyecmd_printf("clear_screen 0");   // clear tracker display
                                                                       // add boxes at fixation goal tar
        eyecmd_printf("draw_filled_box %d %d %d %d  7", fixation_x-16, fixation_y-16,
                fixation_x+16, fixation_y+16);
        eyecmd_printf("draw_filled_box %d %d %d %d  7", goal_x-16, goal_y-16,
                goal_x+16, goal_y+16);

        // V_CRT command is recorded for EyeLink Data Viewer analysis
        // Adding a custom specific reaction time defintion, with the saccades as RT
        // end event and "SYNCTIME" as RT start event.  Saccade must be more than 2.0
        // degrees and falls within 50 pixels of the saccade goal.
        // Grammer:
        //      !V V_CRT SACCADE SYNCTIME amplitude x y diameter

        eyemsg_printf("!V V_CRT SACCADE SYNCTIME 2.0    %d %d 50", goal_x, goal_y);

        // run sequence and trial
        i = run_dynamic_trial(background, 200*1000L, saccadic_drawing);
                                               // clean up background bitmap
        SDL_FreeSurface(background);
        free_targets();

        return i;
}
```

The saccadic trial is then executed by calling `run_dynamic_trial()`, passing it the background bitmap and the drawing function for saccadic trials. Finally, we delete the targets and background bitmap, and return the result code from the trial.

21.4.2 Saccadic Trial Drawing Function

The drawing function `saccadic_drawing()` is called after each vertical retrace to create the display. For the saccadic trials, this involves deciding when the fixation and saccadic targets will become visible or be hidden, and reporting these events via messages in the EDF file.

The drawing code for saccadic trials is fairly straightforward, which was the goal of all the support code in this template. First, the visibility of the fixation target is determined by comparing the time after the start of the trial (dt) to the computed delay for fixation target offset. The fixation target is redrawn, and the process is repeated for the saccadic goal target. Note that `move_target()` may be called even if the target has not changed state, as this function will not do any drawing unless the shape, visibility, or position of the target has changed.

```
        fv = (dt <  prestep_delay+overlap_interval) ? 1 : 0;
                         // compute goal visibility
        gv = (dt >= prestep_delay) ? 1 : 0;

                        // draw or hide fixation target
        move_target(0, fixation_x, fixation_y, fv);
                         // draw or hide goal target
        move_target(1, goal_x, goal_y, gv);

        SDL_Flip(window);

                          // draw or hide fixation target
        move_target(0, fixation_x, fixation_y, fv);
                         // draw or hide goal target
        move_target(1, goal_x, goal_y, gv);
```

After drawing, we can output messages without causing delays in drawing the stimuli. We output messages only if the state of the target has changed from the values stored from the previous call. The text of the messages simply represents the expected change in target appearance. Note that the messages in this example are NOT standard messages that are automatically processed by analysis tools, and would require writing an ASC file analysis program to use them.

```
        if(dt > 0)  // no message for initial setup
        {
                if(fv != fixation_visible)  // mark fixation offset
                {
                        eyemsg_printf("%d HIDEFIX", current_msec()-t);
                        eyemsg_printf("!V FILLBOX 0 0 0 %d %d %d %d",
                          fixation_x-(outer_diameter+1)/2,
                          fixation_y-(outer_diameter+1)/2,
                          fixation_x+(outer_diameter+1)/2,
                          fixation_y+(outer_diameter+1)/2);
                }
                if(gv != goal_visible)      // mark target onset
                {
                        eyemsg_printf("%d SHOWGOAL", current_msec()-t);
                        eyemsg_printf("!V FIXPOINT 255 255 255 255 255 255 %d %d %d 0",
                          goal_x, goal_y, outer_diameter);
                }
        }
```

Finally, we send the fixation target position back to `run_dynamic_trial()` for use in drift correction, and check to see if the trial is completed.

```
        fixation_visible = fv;   // record state for change detection
        goal_visible = gv;

        if(xp) *xp = fixation_x;  //return fixation point location
        if(yp) *yp = fixation_y;

                                   // check if trial timed out
        if(dt > trial_duration)
                return 1;
        else
                return 0;
```

21.4.3 Pursuit Trial Setup

A second type of dynamic display trial implements sinusoidal smooth pursuit, where a target moves very smoothly and is tracked by the subject's gaze. High refresh rates and highly accurate positioning of the target are critical to produce the solid, subjectively smooth motion required. Drawing the target at the time of display refresh allows this to be achieved. This template implements sinusoidal pursuit by computing target position in the retrace drawing function. In addition it can change the target appearance at random intervals, which has been found to improve subject concentration and pursuit accuracy.

The code for sinusoidal pursuit is somewhat more complex than the previous example, mostly because of the code for target switching. The basics are very simple, and can be adapted for many pursuit patterns.

The function `do_sine_trial()` sets up for each trial in the sinusoidal pursuit paradigm, and in general follows the steps used in the saccadic trial setup. The first step is to create a blank background bitmap and initialize the target shapes (if these do not change, this could be done once for the entire experiment, instead of for each trial). The target is pure red, as we have found that most monitors have extremely low persistence for the red phosphor, which eliminate the "trail" left behind the moving target.

Next, the variables that set the frequency, phase (where the sinusoidal cycle starts), the amplitude of the motion and the target switching interval are set. (Phase of the sinusoid is in degrees, where 0° is at the center and moving right, 90° is at the right extreme of motion, 180° is at the center and moving left, and 270° is at the left extremum. Pursuit is usually begun at the left (270°) as the target accelerates smoothly from this position. The last step of the setup is to draw reference graphics to the eye tracker display - in this case, a box at the center and each end of the pursuit motion, and a line along the axis of target motion.

```
//******************* SINUSOIDAL PURSUIT DRAWING AND MOTION *****************
#define PI 3.14159265358979323846
#define FRACT2RAD(x)  (2*PI*(x))          // fraction of cycles-> radians
#define DEG2RAD(x)    ((2*PI/360*(x))     // degrees -> radians

#define FRACT2RAD(x)  (2*PI*(x))          // fraction of cycles-> radians
#define DEG2RAD(x)    ((2*PI/360*(x))     // degrees -> radians

int sine_amplitude;    // amplitude of sinusoid (pixels, center-to-left)
int sine_plot_x;       // center of sineusiod
int sine_plot_y;
float sine_frequency;    // sine cycles per second
float sine_start_phase;  // in degrees, 0=center moving right
int sine_cycle_count;

UINT32 min_target_duration = 2000;   // random target change interval
UINT32 max_target_duration = 4000;
UINT32 next_target_time;   // time of last target switch
int current_target;        // current target used

UINT32 target_report_interval = 50;  // report sine phase every 50 msec
UINT32 last_report_time;
int prev_x, prev_y, outer_diameter;

//********** SINUSOIDAL TRIAL SETUP AND RUN *********

// setup, run sinusiodal pursuit trial
// do_change controls target changes

int do_sine_trial(int do_change)
{
        int i;
        SDL_Color target_color = { 255,0,0} ;

        // blank background bitmap
        SDL_Surface * background = NULL;
        background = blank_bitmap(target_background_color);
        if(!background) return TRIAL_ERROR;

        // red targets (for minimal phosphor persistance)
        initialize_targets(target_color, target_background_color);

        sine_amplitude = SCRWIDTH/3;  // about 10 degrees for 20 deg sweep
        sine_plot_x = SCRWIDTH/2;     // center of display
        sine_plot_y = SCRHEIGHT/2;
        sine_frequency = 0.4F;        // 0,4 Hz, 2.5 sec/cycle
        sine_start_phase = 270;       // start at left
        sine_cycle_count = 10;        // 25 seconds

        if(do_change)               // do we do target flip?
        {
                current_target = 2;     // yes: set up for flipping
                next_target_time = 0;
                last_report_time = 0;
        }
        else
        {
                current_target = 1;             // round target
                next_target_time = 0xFFFFFFFF;   // disable target flipping
                last_report_time = 0xFFFFFFFF;
        }

        set_offline_mode();     // Must be offline to draw to EyeLink screen
        eyecmd_printf("clear_screen 0");   // clear tracker display
                                                              // add boxes at left, right extre
        eyecmd_printf("draw_filled_box %d %d %d %d  7", sine_plot_x-16, sine_plot_y-16,
                        sine_plot_x+16, sine_plot_y+16);
        eyecmd_printf("draw_filled_box %d %d %d %d  7", sine_plot_x+sine_amplitude-16,
                        sine_plot_y-16, sine_plot_x+sine_amplitude+16, sine_plot_y+16);
        eyecmd_printf("draw_filled_box %d %d %d %d  7", sine_plot_x-sine_amplitude-16,
                        sine_plot_y-16, sine_plot_x-sine_amplitude+16, sine_plot_y+16);
                                                              // add expected track line
        eyecmd_printf("draw_line %d %d %d %d  15", sine_plot_x+sine_amplitude-16,
                        sine_plot_y, sine_plot_x-sine_amplitude+16, sine_plot_y);

        prev_x = -1;
        prev_y = -1;

        i = run_dynamic_trial(background, 200*1000L, sinusoidal_drawing);

        SDL_FreeSurface(background);
        free_targets();

        return i;
}
```

The pursuit trial is then executed by calling `run_dynamic_trial()`, passing it the background bitmap and the drawing function for sinusoidal pursuit trials. Finally, we delete the targets and background bitmap, and return the result code from the trial.

21.4.4 Pursuit Trial Drawing Function

The drawing function `sinusoidal_drawing()` is called after each vertical retrace to create the display. For the sinusoidal pursuit trials, this involves computing the new target location, checking whether the target shape needs to be changed (and choosing a random duration for the next interval), redrawing the target, and reporting target position and appearance changes via messages in the EDF file. The drawing code is somewhat more complex than that for the saccadic trial, mostly due to the target shape changing code.

```
int __cdecl sinusoidal_drawing(UINT32 t, UINT32 dt, int *xp, int *yp)
{
        float phase;   // phase in fractions of cycle
        int tchange = 0;
        int x, y;
        // phase of sinusoid
        phase =(float) (sine_start_phase/360 + (dt/1000.0)*sine_frequency);
        x = (int)(sine_plot_x + sine_amplitude*sin(FRACT2RAD(phase)));
        y = sine_plot_y;
```

Computing the target position is relatively straightforward. First, the phase of the target is computed, as a fraction of a cycle. The sine of this phase multiplied by 2 π gives a number between 1 and -1, which is multiplied by the amplitude of the motion and added to the center position. The vertical position of the target is fixed for horizontal motion.

Next, we check to see if the target shape change interval has expired. If so, a new random interval is selected and a new shape is selected (this alternates between 2 and 3 which are left and right tilted lines). After this, the target is redrawn with the new position and appearance:

```
        // compute target position
        if(dt >= next_target_time)
        {
                current_target = (current_target==2) ? 3 : 2;
                next_target_time = dt +
                        (rand()%(max_target_duration-min_target_duration))+min_target_duration;
                tchange = 1;
        }

        move_target(0, x, y, current_target);
        Flip(window);
```

After drawing, it is safe to send messages to the EDF file. In this example, messages containing target position are sent every 50 milliseconds (the exact interval depends on the refresh rate, since these messages are only sent after retrace). The position message has the X and Y position, and the last digit is the delay of the message from the retrace. Similarly, target appearance changes are recorded.

Remarks:

These messages are NOT standard messages and will not be automatically supported by future analysis tools. An alternative method of reporting position would be to give the parameters of the saccadic motion (phase, amplitude, center position and frequency) at the start of the trial and to report the phase of the sinusoid at regular intervals - this would allow analysis programs to regenerate the perceived track of the sinusoid precisely for each sample.

```
        if(dt-last_report_time >= target_report_interval)
        {
                last_report_time = dt;
                eyemsg_printf("%d POSN %d %d", current_msec()-t, x, y);
        }

        // The following code is for Data Viewer Drawing
        // Erase the target, if shown
        if (prev_x!=-1 && prev_y!=-1)
                eyemsg_printf("!V FILLBOX 0 0 0 %d %d %d %d",
                        prev_x-(outer_diameter+1)/2,
                        prev_y-(outer_diameter+1)/2,
                        prev_x+(outer_diameter+1)/2,
                        prev_y+(outer_diameter+1)/2);

        // The following code is for Data Viewer Drawing
        // Draw the fixation point
        eyemsg_printf("!V FIXPOINT 255 0 0 0 0 0 %d %d %d 0", x, y, outer_diameter);

        if(tchange)
        {
                eyemsg_printf("%d TSET %d %d %d", current_msec()-t, current_target, x, y);
        }
```

Finally, the position of the target is reported for drift correction at the start of the trial, and the trial duration is checked. We have set the trial to end after a precise number of cycles of sinusoidal motion in this example.

```
if(xp) *xp = x;  // return fixation point location
if(yp) *yp = y;

prev_x = x ;
prev_y = y;
                            // check if proper number of cycles executed
if(floor(phase-sine_start_phase/360) >= sine_cycle_count)
        return 1;
else
        return 0;
```

21.5 Adapting the "Dynamic" template

The specific example trials (saccadic tasks and sinusoidal pursuit) used in this example can be easily modified to produce tasks that use similar stimuli, or to use slightly different target sizes and colors. However, there are many different types of paradigms that can benefit from refresh-locked drawing, and almost all of these can be implemented by the background bitmap and drawing function method implemented here. Some examples are given below:

- Other types of smooth pursuit motion can be implemented, such as linear motion, jumps, and so on.
- Real-time gaze position data can be combined with the target motion to implement open-loop pursuit paradigms (NOTE: this is not trivial to implement, but is technically possible in terms of data availability and display timing).
- Saccadic tasks using multiple targets, distractors, cues, and other methods can be implemented, using either target alone, the background bitmaps, or multiple bitmaps (see below).
- Instead of drawing targets, the drawing function can copy previously drawn bitmaps to the display, which can implement serial presentations, tachistoscopic displays, and even short animations. As long as only one bitmaps (or at most a few small bitmaps) are being copied, the new bitmap will appear in the same refresh cycle it was drawn in. This is not perfectly compatible with the use of the target support code, as drawing the bitmap will cover up any targets, which cannot be redrawn until the next refresh.

It is of course possible to move the drawing code into the trial function, rather than calling an external function. However, it is less easy to re-use such code, which was the goal of this example.

Chapter 22

COMM SIMPLE and COMM LISTENER Templates

The *comm_simple* and *comm_listener* projects are used together to show the required elements for an experiment where one application (*comm_simple*) is controlling the experiment, while a second application (*comm_listener*) opens a broadcast connection to monitor and analyze the real-time data. The programs are synchronized at startup by exchanging messages through the *eyelink_core* DLL, After this, *comm_listener* relies on the standard messages sent by *comm_simple* for synchronization and to identify the trial being executed. This data is simply used to reproduce the stimulus display, and to plot a gaze position cursor.

To run this experiment, you will need two computers connected by a network hub (a low-speed hub should be used, not a high-speed or multispeed hub) to the eye tracker. The eye tracker must be running EyeLink I version 2.1 or higher, or EyeLink II version 1.1 or higher. Start the *comm_listener* application first, then the *comm_simple* application. Run through the *comm_simple* application in the usual way, and note that the display of the *comm_listener* computer follows along, with a gaze cursor during recording. After the *comm_simple* application finishes, *comm_listener* will wait for another session to begin.

The *comm_simple* application first opens a connection to the eye tracker, then checks for the presence of the *comm_listener* application, and sends a message to inform *comm_listener* that the experiment has begun. The *comm_listener* then opens a broadcast connection to the tracker, and watches for messages and for the start and end of recording blocks. When *comm_simple* disconnects from the eye tracker, *comm_listener* is disconnected as well.

The source code for *comm_simple* is derived from the *simple* template, with only a few changes to enable messages and samples in real-time link data so these are available to *comm_listener*. A few minor rearrangements of commands and messages have also been made to ensure that messages are received by *comm_listener* at the proper time. The source code for *comm_listener* is mostly new, with plotting of the gaze cursor derived from the *trial.c* file used in the *eyedata* template.

22.1 Source Files for "Comm_simple"

These are the files used to build *comm_simple*. Those that were covered previously are marked with an asterisk.

comm_simple.h	Declarations to link together the template experiment files. Most of the declarations in this file can be used in your experiments.
simple_trials.c	(Same file as used in the *simple* template). Called to run a block of trials for the *simple* template. Performs system setup at the start of each block, then runs the trials. 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.
comm_simple_main.c	A modified version of *main.c* used in other templates, which changes the tracker configuration to allow messages in the real-time link data, and enables link data even when not recording. It also exchanges synchronizing messages with *comm_listener*.
comm_simple_trial.c	A modified version of *trial.c* from the simple template. The only changes are to enable samples and events over the link while recording, and to ensure that recording has ended before returning to the trial loop.

22.2 Analysis of "comm_simple_main.c"

This module is almost identical to the file *main.c* used in most previous templates, and only the differences will be discussed. These are in tracker setup to enable messages in the link data, and a new function that synchronizes startup with the *comm_listener* application.

22.2.1 Synchronizing with comm_listener

After being connected to the eye tracker, comm_simple must inform comm_listener that it may open a broadcast connection to the eye tracker. First, it polls the link for a remote named "comm_listener", calling eyelink_poll_remotes and checking for responses. If no remote named "comm_listener" is found, the application is probably not started and the program aborts. If the remote is found, its address is recorded and a message is sent to it. In this case, it is the name of the application, but in a real experiment it might be used to transfer other information.

```
// finds listener application
// sends it the experiment name and our display resolution
// returns 0 if OK, -1 if error
int check_for_listener(void)
{
        INT16 i, n;
        char message[100];
        ELINKNODE node;  //  this will hold application name and address

        eyelink_poll_remotes();  // poll network for any EyeLink applications
        pump_delay(500);         // give applications time to respond

        n = eyelink_poll_responses();  // how many responses?
        for(i=1;i<=n;i++)       // responses 1 to n are from other applications
        {
                if(eyelink_get_node(i, &node) < 0) return -1;  /// error: no such data
                if(!_stricmp(node.name, "comm_listener"))
                {           // Found COMM_LISTENER: now tell it we're ready
                        memcpy(listener_address, node.addr, sizeof(ELINKADDR));
                        eyelink_node_send(listener_address, "NAME comm_simple", 40);
                                                        // wait for "OK" reply
                        if(get_node_response(message, 1000) <= 0) return -1;
                        if(_stricmp(message, "OK")) return -1;   // wrong response?
                        return 0;   // all communication checks out.
                }
        }
        return -1;    // no listener node found
}
```

Next, *comm_simple* waits for a message confirming that the message was received. The process of sending data and waiting for an acknowledging message prevents data from being lost, and could be extended to transfer multiple messages by simply repeating the process. Because it might be necessary to wait for a response at several places in the program a "helper" function has been supplied that waits for reception of a message, and returns an error if no message is received within a set time limit. This function could also be extended to check that the message address matches that of the *comm_listener* application.

```
ELINKADDR listener_address;   // Network address of listerner application

// node reception "helper" function
// receives text message from another application
// checks for time out (max wait of 'time' msec)
int get_node_response(char *buf, UINT32 time)
{
        UINT32 t = current_msec();
        ELINKADDR msgaddr;   // address of message sender

        // wait with timeout
        while(current_time()-t < time)
        {
                int i = eyelink_node_receive(msgaddr, buf); // check for data
                if(i > 0) return i;
        }
        return -1;   // timeout failure
}
```

Finally, code is added to `app_main()` to call the `check_for_listener()` function. We set our network name to "comm_simple" first - this would allow an alternative means for a listening application to find or identify our application.

```
        eyelink_set_name("comm_simple");  // NEW: set our network name

        if(check_for_listener())   // check for COMM_LISTENER application
        {
                alert_printf("Could not communicate with COMM_LISTENER application.");
                goto shutdown;
        }
```

22.2.2 Enabling Messages in Link Data

The remaining changes are to the tracker setup code. This has been rearranged so that the "DISPLAY_-COORDS" message is sent after the link data configuration of the tracker is completed, to ensure that the *comm_listener* application will see this message. Alternatively, we could have re-sent this message later, or included the display resolution in the message sent directly to *comm_listener* earlier.

To enable messages in link data, the MESSAGE type is added to the list of event types for the `"link_-event_filter"` command. This is all that is required for EyeLink II and EyeLink1000 trackers. For EyeLink I trackers, messages must also be echoed as link data between recording blocks. This can be achieved using the "link_nonrecord_events" command, which supports messages for tracker versions 2.1 and later.

```
// SET UP TRACKER CONFIGURATION
// NOTE: set contents before sending messages!
// set EDF file contents
eyecmd_printf("file_event_filter = LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON");
eyecmd_printf("file_sample_data  = LEFT,RIGHT,GAZE,AREA,GAZERES,STATUS");

// set link data (used for gaze cursor)
// NEW: pass message events for listener to use
eyecmd_printf("link_event_filter = LEFT,RIGHT,FIXATION,SACCADE,BLINK,BUTTON,MESSAGE");
eyecmd_printf("link_sample_data  = LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS");

// NEW: Allow EyeLink I (v2.1+) to echo messages back to listener
eyecmd_printf("link_nonrecord_events = BUTTONS, MESSAGES");

// Program button #5 for use in drift correction
eyecmd_printf("button_function 5 'accept_target_fixation'");

// Now configure tracker for display resolution
// Set display resolution
eyecmd_printf("screen_pixel_coords = %ld %ld %ld %ld",
                         dispinfo.left, dispinfo.top, dispinfo.right, dispinfo.bottom);
eyecmd_printf("calibration_type = HV9");       // Setup calibration type
eyemsg_printf("DISPLAY_COORDS %ld %ld %ld %ld",// Add resolution to EDF file
                         dispinfo.left, dispinfo.top, dispinfo.right, dispinfo.bottom);
if(dispinfo.refresh>40)
        eyemsg_printf("FRAMERATE %1.2f Hz.", dispinfo.refresh);
```

22.3 Analysis of "comm_simple_trial.c"

This module is almost identical to the file *trial.c* used in the *simple* template, with only two differences. The first is that samples and events are enabled by start_recording(1,1,1,1), to make this data available to *comm_listener*. The second difference is to call the function eyelink_wait_for_mode_-ready() at the end of the trial. The reason for this is that we called stop_recording() to end the trial, which simply sends a command message to the eye tracker and does not wait for the tracker to actually stop recording data. This means that the TRIALID message for the next trial might actually be sent before recording ends, and comm_listener would see it arrive while processing eye data, and therefore not properly process it.

```
// Call this at the end of the trial, to handle special conditions
error = check_record_exit();

// ensure we are out of record mode before returning
// otherwise, TRIALID message could be send before
// comm_listener sees end of recording block data

eyelink_wait_for_mode_ready(500);
return error;
```

22.4 Source Files for "Comm_listener"

These are the files used to build *comm_listener*. Those that were covered previously are marked with an asterisk.

comm_listener.h	Declarations to link together the template experiment files. Most of the declarations in this file can be used in your experiments.
comm._listener_main.c	A modified version of *main.c* used in other templates, which does not do calibration setup or tracker setup. Instead, it waits for a message from *comm_simple*, then opens a broadcast connection and turns on link data reception.
comm_listener_loop.c	New code, which listens to link data and messages. It determines the display resolution of *comm_simple* from DISPLAY_COORD messages, and reproduces the trial stimulus from TRIALID messages. When a recording block start is found in the data stream, it transfers control to *comm_listener_record.c*.
comm_listener_record.c	A modified version of *data_trial.c*, from the *eyedata* template. This version does not start recording or draw trial stimulus (this was done previously in comm_listener_loop.c). It uses link data to plot a gaze cursor, and displays gaze position and time in the trial at the top of the display. Messages are read to determine trial start time from the SYNCTIME message. It exits when the end of the recording block is found in the data stream.

22.5 Analysis of "comm_listener_main.c"

This module is derived from *main.c* used in other templates, but it uses only some of the application initialization code from that file. The startup code for comm_listener does not need to configure calibration graphics, open a data file, or send configuration commands to the eye tracker - this is all done by *comm_simple*. Instead, it waits for a message from *comm_simple*, then opens a broadcast connection and turns on link data reception.

First, the DLL is initialized so we can send and receive messages with *comm_simple*. By calling `open_eyelink_connection(-1)`, this is done without opening a connection to the eye tracker. We also set our network name to `"comm_listener"`, so that *comm_simple* will be able to find us:

```
// open DLL to allow unconnected communications
if(open_eyelink_connection(-1))
        return -1;    // abort if we can't open link

eyelink_set_name("comm_listener");  // set our network name
```

After the usual display and application setup, we then wait for a message from *comm_simple*, by calling `wait_for_connection()` (described below). Once we have been contacted by *comm_simple*, a broadcast connection is opened and link data reception is enabled. Finally, we tell *comm_simple* that we are ready to proceed by sending an "OK" message, and call `listening_loop()` (defined in *comm_listener_loop.c*). When *comm_simple* closes its connection to the eye tracker, our broadcast connection is closed as well, and `listening_loop()` returns.

```
while(1)  // Loop through one or more sessions
{
        // wait for connection to listen to, or aborted
```

```
                if(wait_for_connection()) goto shutdown;

                // now we can start to listen in
                if(eyelink_broadcast_open())
                {
                        alert_printf("Cannot open broadcast connection to tracker");
                        goto shutdown;
                }

                //enable link data reception by EyeLink DLL
                eyelink_reset_data(1);
                //NOTE: this function can discard some link data
                eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS);

                pump_delay(500);       // tell COM_SIMPLE it's OK to proceed
                eyelink_node_send(connected_address, "OK", 10);

                clear_full_screen_window(target_background_color);
                get_new_font("Times Roman", SCRHEIGHT/32, 1);         // select a font
                i = 1;
                graphic_printf(window, target_foreground_color, NONE, SCRWIDTH/15,
                        i++*SCRHEIGHT/26, "Listening in on link data and tracker mode...");
                SDL_Flip(window);

                listening_loop();   // listen and process data and messages
                // returns when COMM_SIMPLE closes connection to tracker

                if(break_pressed())  // make sure we're still alive
                        goto shutdown;
        }
```

The function `wait_for_connection()` displays a startup message, and waits for a message from comm_simple. The contents of this message are ignored in this example, but the `ELINKADDR` of *comm_-simple* is saved for sending our reply.

```
ELINKADDR connected_address;   // address of comm_simple (from message)


//******** WAIT FOR A CONNECTION MESSAGE **********
// waits for a inter-application message
// checks message, responds to complete connection
// this is a very simple example of data exchange

int wait_for_connection(void)
{
        int i;
        int first_pass = 1;   // draw display only after first failure
        char message[100];

        while(1)          // loop till a message received
        {
                i = eyelink_node_receive(connected_address, message);
                if(i > 0)      // do we have a message?
                {              // is it the expected application?
                        if(!_stricmp(message, "NAME comm_simple"))
                        {              // yes: send "OK" and proceed
                                return 0;
                        }
                }

                if(first_pass)   // If not, draw title screen
                {
                        SDL_Color colr = { 0,0,0};
                        first_pass = 0;  // don't draw more than once

                        clear_full_screen_window(target_background_color);
                        get_new_font("Times Roman", SCRHEIGHT/32, 1); // select a font
                        i = 1;
                        graphic_printf(window, colr, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "EyeLink Data Listener and Communication Demonstration");
                        graphic_printf(window, colr, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Copyright 2002 SR Research Ltd.");
                        i++;
                        graphic_printf(window, colr, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Waiting for COMM_SIMPLE application to send startup message...");
                        graphic_printf(window, colr, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Press ESC to quit");

                        SDL_Flip(window);
                }

                i = getkey();          // check for exit
                if(i==ESC_KEY || i==TERMINATE_KEY) return 1;
        }
}
```

22.6 Analysis of "comm_listener_loop.c"

The core of this module is `listening_loop()`, which processes all link data broadcast from the tracker between recording blocks. It processes all messages (which are copies of those placed in the tracker data file by *comm_simple*) to determine display resolution (from DISPLAY_COORD messages) and to reproduce the trial stimulus (from TRIALIAD messages). In an actual data- listener application, the TRIALID message might be used to determine how to process recording data.

When the start of a recording block is encountered in the data stream, the function `eyelink_in_data_block(1, 1)` will return 1. We then call `listener_record_display()` to handle this data. Note that we look in the link data stream for the start of recording, rather than monitoring the eye tracker mode with `eyelink_current_mode()`, as this ensures that we get all data and messages between the start and end of recording. This would not be as critical if the code for reading eye data samples and events was included in the same loop as code to read data between trials, as messages would always be processed properly by keyword.

```
//********** LISTENING LOOP *************

void listening_loop(void)
{
        int i;
        int j = 6;

        char trial_word[40];  // Trial stimulus word (from TRIALID message)
        char first_word[40];  // first word in message (determines processing)

        tracker_pixel_left = (float)SCREEN_LEFT;    // set default display mapping
        tracker_pixel_top = (float)SCREEN_TOP;
        tracker_pixel_right = (float)SCREEN_RIGHT;
        tracker_pixel_bottom = (float)SCREEN_BOTTOM;

        // Now we loop through processing any link data and messages
        // The link will be closed when the COMM_SIMPLE application exits
        // This will also close our broadcast connection and exit this loop

        while(eyelink_is_connected())
        {
                ALLF_DATA data;   // link data or messages
                                                  // exit if ESC or ALT-F4 pressed
                if(escape_pressed() || break_pressed()) return;

                i = eyelink_get_next_data(NULL);   // check for new data item
                if(i == 0) continue;

                if(i == MESSAGEEVENT)   // message: check if we need the data
                {
                        eyelink_get_float_data(&data);
                        #ifdef PRINT_MESSAGES   // optionally, show messages for debugging
                        get_new_font("Times Roman", SCRHEIGHT/55, 1); // select a font
                        graphic_printf(window, target_foreground_color, NONE, SCRWIDTH/15,
                                        j++*SCRHEIGHT/55, "MESSAGE=%s", data.im.text);
                        #endif
                        sscanf(data.im.text, "%s", first_word);  // get first word
                        if(!_stricmp(first_word, "DISPLAY_COORDS"))
                        {       // get COMM_SIMPLE computer display size
                                sscanf(data.im.text, "%*s %f %f %f %f",
                                         &tracker_pixel_left, &tracker_pixel_top,
                                         &tracker_pixel_right, &tracker_pixel_bottom);
                        }
                        else if(!_stricmp(first_word, "TRIALID"))
                        {
                                // get TRIALID information
                                sscanf(data.im.text, "%*s %s", trial_word);
                                // Draw stimulus (exactly as was done in COMM_SIMPLE)
                                #ifndef PRINT_MESSAGES
                                clear_full_screen_window(target_background_color);
                                #endif

                                // We scale font size for difference in display resolutions
                                get_new_font("Times Roman", (int) (SCRWIDTH/25.0 *
                                                SCRWIDTH/(tracker_pixel_right-tracker_pixel_left+1)), 1);
                                graphic_printf(window, target_foreground_color,  NONE,
                                           (int) (SCRWIDTH/2), (int)(SCRHEIGHT/2), "%s", trial_word);
                                Flip(window); //
                                graphic_printf(window, target_foreground_color,  NONE,
                                           (int) (SCRWIDTH/2), (int)(SCRHEIGHT/2), "%s", trial_word);
                        }
                }

                // link data block opened for recording?
                if(eyelink_in_data_block(1, 1))
                {
                        listener_record_display(); // display gaze cursor on stimulus
                                                                // clear display at end of trial
                        #ifndef PRINT_MESSAGES
                        clear_full_screen_window(target_background_color);
                        #endif
                }
        }
}
```

It is very important to know the display resolution of the computer running comm_simple, as this sets the coordinate system that gaze data is reported in. Without this, differences in display settings between computers could cause gaze data to be plotted on the wrong position. This display information is read from the DISPLAY_COORDS message sent during tracker configuration. In addition, newer EyeLink trackers (EyeLink I version 2.1 and higher, and EyeLink II version 1.1 and higher) automatically insert a "GAZE_-COORDS" message just before each recording block. Two mapping functions are supplied to convert data in the coordinates of the comm_simple display to local display coordinates:

```
//****** MAP TRACKER TO LOCAL DISPLAY ***********

float tracker_pixel_left =   0;   // tracker gaze coord system
float tracker_pixel_top =    0;   // used to remap gaze data
float tracker_pixel_right =  0;   // to match our display resolution
float tracker_pixel_bottom = 0;

// remap X, Y gaze coordinates to local display
```

```
float track2local_x(float x)
{
        return SCREEN_LEFT +
                (x - tracker_pixel_left) * SCRWIDTH / (tracker_pixel_right - tracker_pixel_left + 1);
}

float track2local_y(float y)
{
        return SCREEN_TOP +
                (y - tracker_pixel_top) * SCRHEIGHT / (tracker_pixel_bottom - tracker_pixel_top + 1);
}
```

22.7 Analysis of "comm_listener_record.c"

This module processes link data during a recording block. It plots samples as a gaze cursor. It also prints the time (from the start of the trial as reported by the SYNCTIME message), and gaze position. It exits when `eyelink_in_data_block(1, 1)` return 0, indicating that the end of the recording block has been encountered in the data stream.

The first thing `listener_record_display()` does is to determine which eye's data to plot. This is available from `eyelink_eye_available()`, since we know we are in a recording block.

```
//******** PLOT GAZE DATA DURING RECORDING  ******
int listener_record_display(void)
{
        ALLF_DATA evt;
        UINT32 trial_start_time = 0;
        unsigned key;
        int eye_used;        // which eye to show gaze for
        float x, y;       // gaze position
        float ox=-1, oy=-1;  // old gaze positio (to determine change)
        int i,j=1;

        // create font for position display
        get_new_font( "Arial", SCRWIDTH/50, 0);

        initialize_cursor(window, SCRWIDTH/50);

        eye_used = eyelink_eye_available();
        // use left eye if both available
        if(eye_used==BINOCULAR) eye_used = LEFT_EYE;
```

Next, we loop and process events and samples until the application is terminated, the link is closed, or the recording block ends. The time of the trial start is computed by subtracting the time delay (following the "SYNCTIME" keyword) from the time of the message.

```
        while(eyelink_is_connected())  // loop while record data available mode
        {
                key = getkey();             // Local keys/abort test
                if(key==TERMINATE_KEY)      // test ALT-F4 or end of execution
                 break;

                if(!eyelink_in_data_block(1, 1))
                        break;  // stop if end of record data
                i = eyelink_get_next_data(NULL);   // check for new data item
                if(i == MESSAGEEVENT)   // message: check if we need the data
                {
                        eyelink_get_float_data(&evt);     // get message
                                           // get trial start time from "SYNCTIME" message
                        if(!_strnicmp(evt.im.text, "SYNCTIME", 8))
                        {
                                trial_start_time = 0;    // offset of 0, if none given
                                sscanf(evt.im.text, "%*s %d", &trial_start_time);
                                trial_start_time = evt.im.time - trial_start_time;
                        }
#ifdef PRINT_MESSAGES
                        graphic_printf(window, target_foreground_color, NONE, SCRWIDTH/100,
                                j++*SCRHEIGHT/50, "MESSAGE=%s", evt.im.text);
#endif
                }
```

It is very useful to be able to detect gaps in the link data, which might indicate link problems, lost data due to delays in processing, or too many messages arriving for the Windows networking kernal to handle. This can be done using the new LOST_DATA_EVENT event, which is inserted by the *eyelink_core* DLL in the data stream at the position of the gap:

```
#ifdef LOST_DATA_EVENT     // only available in V2.1 or later DLL
                if(i == LOST_DATA_EVENT)   // marks lost data in stream
                        alert_printf("Some link data was lost");
#endif
```

Samples are processed in much the same way as in the *data_trial.c* file in the *eyedata* template. Before plotting the gaze cursor, the gaze position data is first converted from the coordinates of the comm_simple display to our display coordinates. Gaze position data is printed in its original form. The time of the sample is also printed, if the time of the trial start has been determined from the SYNCTIME message.

```
                // CODE FOR PLOTTING GAZE CURSOR
                if(eyelink_newest_float_sample(NULL)>0)  // new sample?
                {
                        eyelink_newest_float_sample(&evt);  // get the sample data
                        x = evt.fs.gx[eye_used];   // get gaze position from sample
                        y = evt.fs.gy[eye_used];

                        if(x!=MISSING_DATA && y!=MISSING_DATA &&
                                evt.fs.pa[eye_used]>0)  // plot if not in blink
                        {// plot in local coords
                                draw_gaze_cursor((int)track2local_x(x), int)track2local_y(y));

                                // report gaze position (tracker coords)
                                if(ox!=x || oy!=y)   // only draw if changed
                                {
                                        SDL_Rect r = {(Sint16)(SCRWIDTH*0.87), 0,
                                                (Sint16)(window->w -SCRWIDTH*0.87), 50};
                                        SDL_FillRect(window,&r,SDL_MapRGB(window->format,target_backgrounc
                                                target_background_color.g, target_background_color.b));
                                        graphic_printf(window, target_foreground_color, NONE,
                                                (int)(SCRWIDTH*0.87), 0, " %4.0f  ", x);
                                        graphic_printf(window,target_foreground_color, NONE,
                                                (int)(SCRWIDTH*0.93), 0, " %4.0f  ", y);
                                }
                                ox = x;
                                oy = y;
                        }
                        else
                        {
                                erase_gaze_cursor();   // hide cursor during blink
                        }


                        // print time from start of trial
                        {
                                SDL_Rect r = {(Sint16)(SCRWIDTH*0.75), 0, (Sint16)(SCRWIDTH*0.87 - SCRWIDT
                                SDL_FillRect(window,&r,SDL_MapRGB(window->format,
                                        target_background_color.r, target_background_color.g, target_backg
                                        graphic_printf(window, target_foreground_color,NONE,
                                                  (int)(SCRWIDTH*0.75), 0, " % 8d ", evt.fs.time-trial_start_time);
                              }
                    }
          }
          erase_gaze_cursor();    // erase gaze cursor if visible
          return 0;
}
```

22.8 Extending the "comm_simple" and "comm_listener" Templates

These templates are designed as examples of how to write cooperating applications, where one computer listens in on an experiment in progress. The code here is designed to show the basic elements, such as startup synchronization, enabling and processing link data, mapping gaze coordinates for differences in display resolution, and exchanging messages between applications.

There are other ways to achieve these operations: for example, messages could be exchanged directly between applications to transfer display resolution or TRIALID data, and the connection state of the tracker could be monitored instead of exchanging messages at startup (this will be used in the broadcast template, discussed next).

Chapter 23

"BROADCAST" Template

The *broadcast* project is designed to show some alternative ways of performing multiple-computer experiments. These include:

- Determining when a primary connection has been made to the tracker by reading its state, without requiring a broadcast connection or messages from the other application. This allows it to be used with any application that outputs samples as realtime data.
- Reading display resolution information directly from the eye tracker, without reading messages.
- Reproducing calibration targets, by monitoring the tracker mode and calibration target updates. Calibration target positions are also remapped to our display coordinates.

The concept behind this example is to have this application listen in on any experiment, and to generate calibration and gaze-position displays, which would be combined with a video record of the experiment computer's display using a VGA-to-video overlay device. (In fact, this demonstration would need some enhancements to be used in this way, as the overlay it generates would probably not align precisely with the video of the experiment computer's display. This could be fixed by changing the display mapping functions to incorporate additional correction factors, but the new EyeLink video overlay generation features obviate the need for such a program).

To run this experiment, you will need two computers connected by a network hub (a low-speed hub should be used, not a high-speed or multispeed hub) to the eye tracker. The eye tracker must be running EyeLink I version 2.1 or higher, or EyeLink II version 1.1 or higher. Start the *broadcast* application first, then the *eyedata* or *gcwindow* application on the other computer (or any application that uses real-time sample data). Run through the *eyedata* application in the usual way, and note that the display of the *broadcast* computer follows along, displaying calibration targets and with a gaze cursor during recording. After the *eyedata* application finishes, *broadcast* will wait for another session to begin.

The *broadcast* application starts by requesting time and status updates from the tracker, and checks to see if a connection has been opened by any other application. It then opens a broadcast connection to the tracker, and watches the tracker mode. When in the proper modes, it reproduces calibration targets or plots a gaze cursor. Otherwise, it displays a black display that is transparent to the video overlay device. When the other application disconnects from the eye tracker, *broadcast* is disconnected as well, and begins to poll the eye tracker for the start of the next session.

The source code for *broadcast* is mostly new. Only those parts that illustrate new concepts will be discussed in detail.

23.1 Source Files for "broadcast"

These are the files used to build *broadcast*. Those that were covered previously are marked with an asterisk.

Broadcast.h ∗	Declarations to link together the template experiment files. Most of the declarations in this file can be used in your experiments.
Broadcast_main.c	A modified version of *main.c* in other templates, which does not do tracker setup. Instead, it polls the tracker state until another application opens a connection to it, then opens a broadcast connection. It then determines display resolution by reading tracker settings, and monitors tracker modes to determine when to display a gaze cursor or calibration targets.
broadcast_record.c	A modified version of *trial.c*, from the *eyedata* template. This version does not start recording but simply turns on link data reception. It uses link data to plot a gaze cursor, and displays gaze position and tracker time at the top of the display. It exits when the tracker leaves recording mode.

23.2 Analysis of "broadcast_main.c"

This module is derived from *main.c* used in most other templates, and does most of the usual setup, except for configuring the tracker and opening a data file. It begins by initializing the DLL so we can use the link to communicate with the tracker. By calling `open_eyelink_connection(-1)`, this is done without opening a connection to the eye tracker. We also set our network name to "broadcast", so that *comm_simple* will be able to find us:

```
if(trackerip)
        set_eyelink_address(trackerip);
else
        set_eyelink_address("100.1.1.1");


// open DLL to allow unconnected communications
if(open_eyelink_connection(-1))
        return -1;    // abort if we can't open link

eyelink_set_name("broadcast");  // set our network name
```

The usual display and calibration are done next. We also call `set_remap_hooks()`, which set up a "hook" function to remap the location of calibration targets to match our display resolution. We will discuss this code later.

Next, the code calls `wait_for_connection()` (described later) to determine if another application has connected to the eye tracker. Once this has occurred, a broadcast connection is opened to the eye tracker to allow reception of link data and monitoring of the tracker interactions with the application. Next, we read the display resolution (actually, the gaze position coordinate system) that the tracker has been configured for. We then call `track_mode_loop()` to monitor the tracker and determine when to display calibration targets or to plot the gaze cursor. When the other application closes its connection to the eye tracker, our broadcast connection is closed as well, and `track_mode_loop()` returns.

```
while(1)  // Loop through one or more sessions
```

```
        {
                // wait for connection to listen to, or aborted
                if(wait_for_connection()) goto shutdown;
                pump_delay(1000);   // give remote and tracker time for setup
                // now we can start to listen in
                if(eyelink_broadcast_open())
                {
                        alert_printf("Cannot open broadcast connection to tracker");
                        goto shutdown;
                }

                clear_full_screen_window(transparent_key_color);

                can_read_pixel_coords = 1;           // first try to read coords
                tracker_pixel_left = SCREEN_LEFT;    // set defaults in case fails
                tracker_pixel_top = SCREEN_TOP;
                tracker_pixel_right = SCREEN_RIGHT;
                tracker_pixel_bottom = SCREEN_BOTTOM;
                if(eyelink_is_connected())
                        if(read_tracker_pixel_coords()==-1)
                        {
                                alert_printf("Cannot determine tracker pixel coords:
                                        assuming %dx%d", SCRWIDTH, SCRHEIGHT);
                                can_read_pixel_coords = 0;
                        }

                track_mode_loop();   // listen and process by tracker mode
                if(break_pressed())  // make sure we're still alive
                        goto shutdown;
        }
```

23.2.1 Checking Tracker Connection Status

The function `wait_for_connection()` loops until the tracker is connected to another application. It waits 500 milliseconds between tests (otherwise the tracker would be overloaded with our request) using the `Sleep(500)` function, which also gives other Windows applications some time.

```
//******** WAIT FOR A CONNECTION TO TRACKER *********

int wait_for_connection(void)
{
        int i;
        int first_pass = 1;   // draw display only after first failure

        while(1)          // loop till a connection happens
        {               // check if tracker is connected
                i = preview_tracker_connection();
                if(i == -1)
                {
                        alert_printf("Cannot find tracker");
                        return -1;
                }
                else if(i > 0)
                        return 0;  // we have a connection!

                if(first_pass)   //If not, draw title screen
                {
                        SDL_Color bg = {192,192,192};
                        SDL_Color fg = {0,0,0};
                        first_pass = 0;  //don't draw more than once

                        clear_full_screen_window(bg);
                        get_new_font("Times Roman", SCRHEIGHT/32, 1);   //select a font
                        i = 1;
                        graphic_printf(window, fg, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "EyeLink Broadcast Listening Demonstration");
                        graphic_printf(window, fg, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Copyright 2002 SR Research Ltd.");
                        i++;
                        graphic_printf(window, fg, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Waiting for another computer to connect to tracker...");
                        graphic_printf(window, fg, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Press ESC to exit from this screen");
                        graphic_printf(window, fg, NONE, SCRWIDTH/15, i++*SCRHEIGHT/26,
                                "Press ALT-F4 to exit while connected");
                        SDL_Flip(window);
                }

                i = getkey();         //check for exit
                if(i==ESC_KEY || i==TERMINATE_KEY) return 1;

                Sleep(500);   //go to background, don't flood the tracker
        }
}
```

The tracker connection status is read by `preview_tracker_connection()`, which communicates with the tracker without requiring a connection to be opened. A status and time request is sent by calling `eyelink_request_time()`. When no connection has been opened by our application, this sends the request to the address set by `set_eyelink_address()` (or the default address of "100.1.1.1" if this function has not been used to change this).

Next, we wait for a response to be returned from the tracker, monitoring this with `eyelink_read_time()` which returns 0 until a response is received. This should take less than 1 millisecond, but we wait for 500 milliseconds before giving up (this means that no tracker is running at the specified address, or that the link is not functioning).

We then look at the link status flag data, which is part of the `ILINKDATA` structure kept by the *eyelink_core* DLL. A pointer to this structure is returned by `eyelink_data_status()`. Several flags indicate the connection status (for a complete list, see the *eye_data.h* header file).

```
//**************************** PREVIEW TRACKER STATE ************************
// checks link state of tracker
// DLL must have been started with open_eyelink_connection(-1)
// to allow unconnected time and message communication
// RETURNS: -1 if no reply
//          0 if tracker free
//          LINK_CONNECTED if connected to another computer
//          LINK_BROADCAST if already broadcasting

int preview_tracker_connection(void)
{
        UINT32 t, tt;
        ILINKDATA *idata = eyelink_data_status();  // access link status info
        eyelink_request_time();         // force tracker to send status and time
        t = current_msec();
        while(current_msec()-t < 500)   // wait for response
        {
                tt = eyelink_read_time();   // will be nonzero if reply
                if(tt != 0)
                {                         // extract connection state
                        if(idata->link_flags & LINK_BROADCAST) return LINK_BROADCAST;
                        if(idata->link_flags & LINK_CONNECTED) return LINK_CONNECTED;
                        else return 0;
                }
                message_pump();         // keep Windows happy
                if(break_pressed()) return 1;  // stop if program terminated
        }
        return -1;  // failed (timed out)
}
```

23.2.2 Reading and Mapping Display Resolution

In order to properly plot gaze position and display calibration targets at the proper location on our display, we need to know the gaze position coordinate system used by the tracker, which was set to match the display resolution of the application that connected to the tracker. In the template *comm_listener*, this was done by intercepting the DISPLAY_COORDS message. In this example, we will read the gaze coordinate settings directly from the eye tracker. (We will need to read this before calibration as well, in case the resolution was changed by the application).

To read the gaze position coordinate system setting, we need to read the tracker setting for "screen_pixel_-coords". We request this value by calling `eyelink_read_request("screen_pixel_coords")`, then wait for a response by calling `eyelink_read_reply()`. This will copy the tracker's response (as a series of numbers in a text string) into a buffer we supply. We then extract the desired numbers from this string using `sscanf()`.

Note the variable `can_read_pixel_coords`, which indicates if variables can be read from the tracker. Older EyeLink trackers my not allow reading of settings through a broadcast connection, and this variable will be set to 0 if the first read fails, preventing wasted time and error messages later.

```
//****** MAP TRACKER TO LOCAL DISPLAY ***********

int can_read_pixel_coords = 1;     // does tracker support read?
float tracker_pixel_left = 0;      // tracker gaze coord system
float tracker_pixel_top = 0;       // used to remap gaze data
float tracker_pixel_right = 0;     // to match our display resolution
float tracker_pixel_bottom = 0;

// Read setting of "screen_pixel_coords" from tracker
// This allows remapping of gaze data if our display
// has a different resolution than the connected computer
// The read may fail with older tracker software

int read_tracker_pixel_coords(void)
{
        char buf[100] = "";
        UINT32 t;

        if(!eyelink_is_connected() || break_pressed()) return 1;    // not connected

        eyelink_read_request("screen_pixel_coords");
        t = current_msec();
        while(current_msec()-t < 500)
        {
                if(eyelink_read_reply(buf) == OK_RESULT)
                {
                        sscanf(buf, "%f,%f,%f,%f", &tracker_pixel_left,
                                &tracker_pixel_top, &tracker_pixel_right, &tracker_pixel_bottom );
                        return 0;
                }
                message_pump();  // keep Windows happy
                if(!eyelink_is_connected) return 1;
                if(break_pressed()) return 1;
        }
        return -1;  // timed out
}
```

Once we have the gaze-position coordinate system of the tracker, we can map this to our display. These functions apply this mapping to X or Y position data:

```
// remap X, Y gaze coordinates to local display
float track2local_x(float x)
{
        return SCREEN_LEFT + (x - tracker_pixel_left) * SCRWIDTH /
                (tracker_pixel_right - tracker_pixel_left + 1);
}

float track2local_y(float y)
{
        return SCREEN_TOP + (y - tracker_pixel_top) * SCRHEIGHT /
                (tracker_pixel_bottom - tracker_pixel_top + 1);
}
```

Finally, we need to ensure that calibration targets are drawn at the proper position on our display. We do this by setting a "hook" to the *eyelink_core* DLL, causing it to call our function `setup_remap_hooks()` before drawing each calibration target.

```
// callback for calibration target drawing
// this moves target to match position on other displays
static HOOKFCNS hfcns;
void CALLBACK remap_cal_target(INT16 x, INT16 y)
{
        x = track2local_x(x);
        y = track2local_y(y);
        if(hfcns.draw_cal_target_hook)
                hfcns.draw_cal_target_hook(x,y);
}

// setup "hook" function to be called before calibration targets drawn
void setup_remap_hooks(void)
{
        HOOKFCNS *hooks = get_all_hook_functions();
        memcpy(&hfcns,hooks,sizeof(HOOKFCNS));

        hooks->draw_cal_target_hook = remap_cal_target;
        setup_graphic_hook_functions(hooks);
}
```

23.2.3 Tracker Mode Loop

While connected to the eye tracker, we can monitor what mode it is in by calling `eyelink_tracker_mode()`, and use this information to determine what we should be displaying. The function `track_mode_loop()` contains a mode- monitoring loop to do this. For each pass through the loop, it determines if the tracker mode has changed and executes the proper operations for the new mode. It also performs the usual checks for disconnection or program termination, and also sends any local keypresses to the tracker (this may not be desirable for some applications).

```
// Follow and process tracker modes
// Displays calibration and drift correction targets
// Also detects start of recording
// Black backgrounds would be transparent as video overlay
void track_mode_loop(void)
{
        int oldmode = -1;  // to force initial mode setup

        while(eyelink_is_connected())
        {
                int mode = eyelink_tracker_mode();
                unsigned key = getkey();

                if(key==27 || break_pressed() || !eyelink_is_connected()) return;
                else if(key)                    // echo to tracker
                        eyelink_send_keybutton(key,0,KB_PRESS);

                if(mode == oldmode) continue;
```

The core of `track_mode_loop()` is a switch statement that performs the proper operations for each tracker mode. For most modes, the display is cleared to black (`transparent_key_color`) to allow the video to be seen. During camera setup or calibration, a gray background is displayed, with white calibration targets (black targets would be transparent to the video).

To handle calibration, validation, and drift correction, we call the DLL function `target_mode_display()`, which handles display of calibration targets, calibration sounds, key presses, and so on. During recording, we call `record_target_display()`, discussed below. Note that we call `read_tracker_pixel_coords()` when entering the camera setup and calibration modes, to update this information.

```
                switch(mode)
                {
                        case EL_RECORD_MODE:        // Record mode: show gaze cursor
                                clear_full_screen_window(transparent_key_color);
                                record_mode_display();
                                clear_full_screen_window(transparent_key_color);
                                break;

                        case EL_IMAGE_MODE:     // IMAGE NOT AVAILABLE IN BROADCAST
                                break;

                        case EL_SETUP_MENU_MODE:  // setup menu: just blank display
                                clear_full_screen_window(target_background_color);
                                        // read gaze coords in case changed
                                if(eyelink_is_connected() && can_read_pixel_coords)
                                        read_tracker_pixel_coords();
                                break;

                        case EL_CALIBRATE_MODE:    // show calibration targets
                                if(eyelink_is_connected() && can_read_pixel_coords)
                                        read_tracker_pixel_coords();
                        case EL_VALIDATE_MODE:
                        case EL_DRIFT_CORR_MODE:
                                target_mode_display();
                                break;

                        case EL_OPTIONS_MENU_MODE:  // no change in visibility
                                break;

                        default:               // any other mode: transparent key (black)
                                clear_full_screen_window(transparent_key_color);
                                break;
                }
                oldmode = mode;
        }
}
```

23.3 Analysis of "broadcast_record.c"

This module processes link data during a recording block. It plots samples as a gaze cursor (using code from the trial.c file in the *eyedata* template). It also prints the time (as the tracker timestamps) and gaze position. It exits when `eyelink_tracker_mode()` indicates that the tracker has exited recording mode. This method is less precise than the monitoring of the data stream used in the comm_listener template, and can lose samples and messages at the start and end of the recording block, but is acceptable for simply plotting a visible gaze cursor.

Instead of starting recording, old link data is discarded by calling `eyelink_reset_data(1)`, and data reception is enabled by `eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS)`. This will probably discard the first few samples in the data stream as well.

```
//******** PERFORM AN EXPERIMENTAL TRIAL  ******

int record_mode_display(void)
{
        ALLF_DATA evt;
        unsigned key;
        int eye_used = -1;   // which eye to show gaze for
        float x, y;              // gaze position
        float ox=-1, oy=-1;  // old gaze position (to determine change)

        // create font for position display
        get_new_font( "Arial", SCRWIDTH/50, 0);
        while(getkey()); // dump any pending local keys

        //enable link data reception without changing tracker mode
        eyelink_reset_data(1);
        initialize_cursor(window, SCRWIDTH/50);
        eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS);
```

The code then loops until exit conditions are met: disconnection, application termination, or the tracker switching to a non-recording mode.

```
        while(1)    // loop while in record mode
        {
                if(eyelink_tracker_mode() != EL_RECORD_MODE) break;
                key = getkey();            // Local keys/abort test
                if(key==TERMINATE_KEY)     // test ALT-F4 or end of execution
                        break;
                else if(key)             // OTHER: echo to tracker for control
                        eyelink_send_keybutton(key,0,KB_PRESS);
```

Finally, samples are read from the link to plot the gaze cursor. The gaze cursor code is similar to that in the *trial.c* file in the *eyedata* template, except for its color and shape, and will not be discussed here. The gaze position data is first converted to the local display coordinates by `track2local_x()` and `track2local_y()`, which are implemented in *broadcast_main.c*.

The eye to be plotted is determined when the first sample is read from the link, using `eyelink_eye_available()`. In addition, the gaze position data and sample time are printed at the top left of the display.

```
                // CODE FOR PLOTTING GAZE CURSOR
                if(eyelink_newest_float_sample(NULL)>0)  // new sample?
                {
                        eyelink_newest_float_sample(&evt);  // get the sample data
                        if(eye_used == -1)   // set which eye to track by first sample
                        {
                                eye_used = eyelink_eye_available();
                                if(eye_used == BINOCULAR)  // use left eye if both tracked
                                        eye_used = LEFT_EYE;
                        }
                        else
                        {
                                x = evt.fs.gx[eye_used];   // get gaze position from sample
                                y = evt.fs.gy[eye_used];
                                if(x!=MISSING_DATA && y!=MISSING_DATA &&
                                        evt.fs.pa[eye_used]>0)  // plot if not in blink
                                {   // plot in local coords
                                        draw_gaze_cursor(track2local_x(x),  track2local_y(y));
                                        // report gaze position (tracker coords)
                                        if(ox!=x || oy!=y)   // only draw if changed
                                        {
                                                SDL_Rect r = {SCRWIDTH*0.87, 0,
                                                        window->w -SCRWIDTH*0.87, 50};
                                                SDL_FillRect(window,&r,
                                                        SDL_MapRGB(window->format,0, 0, 0));
                                                graphic_printf(window, target_foreground_color,
                                                        NONE, SCRWIDTH*0.87, 0, " %4.0f  ", x);
                                                graphic_printf(window, target_foreground_color,
                                                        NONE, SCRWIDTH*0.93, 0, " %4.0f  ", y);
                                        }
                                        ox = x;
                                        oy = y;
                                }
                                else
                                {
                                        erase_gaze_cursor();   // hide cursor during blink
                                }

                                // print tracker timestamp of sample
                                {
                                        SDL_Rect r = {SCRWIDTH*0.75, 0,
                                                SCRWIDTH*0.87 -SCRWIDTH*0.75, 50};
                                        SDL_FillRect(window,&r,
                                                SDL_MapRGB(window->format,0, 0, 0));
                                        graphic_printf(window,target_foreground_color,
                                                NONE, SCRWIDTH*0.75, 0, " % 8d ", evt.fs.time);
                                }
                        }
                }
        }
        erase_gaze_cursor();   // erase gaze cursor if visible
        return 0;
}
```

23.4 Extending "broadcast"

The *broadcast* example is designed mainly to illustrate some advanced concepts, including monitoring the tracker mode and connection status, duplicating the display of calibration targets, and reading tracker variables. These allow it to function with almost any application that sends real-time sample data over the link. Many of these methods could be added to the *comm_listener* template as well. However, to be useful for real-time analysis *broadcast* would need to handle data during recording in a similar way to *comm_listener*.

Chapter 24

ASC File Analysis

The EyeLink EDF files contain many types of data, including eye movement events, messages, button presses, and samples. The EDF file format is a highly compressed binary format, intended for use with SR Research EyeLink viewers and applications.

The preferred access method for programmers writing experiment-specific analyzers is the ASC file, created by the EDF2ASC translator program. This program converts selected events and samples into text, and sorts and formats the data into a form that is easier to work with. The EDF2ASC translator, EDF data types, and the ASC file format are covered in the "EDF Files" document.

24.1 Creating ASC files with EDF2ASC

The EDF2ASC translator must be run from the DOS command prompt. If you have EyeLink Data Viewer installed on your computer, you may run the GUI version of the EDF2ASC (edfconverterW.exe) from "C:\\Program Files\\SR Research\\edfconverter" This is the procedure required:

- In your experiments, transfer the EDF file to the Display PC. If the transfer fails, you may use the *eyelink_getfile* utility in the *utilities* folder to transfer the file. You may rename it during the transfer.
- When the experiment is completed, the EDF file will also be located in the directory that the EyeLink tracker software was run from, on the tracker PC. This serves as a backup, and may be deleted as disk space is required.
- Run EDF2ASC on the file, using instructions from the "EDF Files Documentation" document. The most useful option is '-ns', to remove samples from the output.

When the translation is completed, a file with the extension of "ASC" will be available for processing.

24.2 Analyzing ASC Files

The ASC files may be viewed with any text editor: the Windows accessory *wordpad.exe* or from the DOS prompt, *EDIT.EXE*. This editor can handle the MS- DOS text file format. Creating and viewing an ASC file should be the first step in creating an analyzer program, to see which messages need to be handled.

Viewing the ASC file is also important in validating an experimental application, to see if the messages, time, etc. match those expected. The DataViewer program can also be used to view messages in combination with eye-movement data.

Programs that process an ASC file must read the ASC file line by line, determine the type of data from the line from the first word in the line, and read words or numbers from the rest of the line as appropriate. A set of tools for reading ASC files is included in the *asc proc* folder. This is the C source file *read_asc.c*, and its header file *read_asc.h*.

A sample analyzer using this toolkit has also been included. This is the source file *sac_proc.c* which processes the sample data file *data.asc*. These can be used as a template for your own analyzers.

24.3 Functions defined by "read_asc.c"

As each line is of the ASC file is read, your analyzer program must determine which part of the experiment it is in (if in a trial, which trial, whether the display is visible, what response has been made, etc.) and compile data on each trial and the entire experimental session. This requires support functions to parse each line of the ASC file, reading keywords, numbers, and text.

The ASC-file processing support functions in *read_asc.c* perform these operations:

- Opening and closing the file
- Reading lines, words, and numerical values
- Matching words and messages to keywords
- Reading data items from the file, including recording start, button presses, eye events, and samples. These may span several lines of text.
- Re-reading from old locations in the file, for multi-pass analysis algorithms The implementation of these operations are described below.

24.3.1 File Reading Functions

When creating an ASC file, EDF2ASC adds the extension ".asc" to the end of the filename. A matching extension can be added to any file name using `add_extension()`, which will not add the extension if `force` is 0 and an extension is already present in the file name.

```
// copies file name from <in> to <out>
// if no extension given, adds <ext>
// <force> is nonzero, ALWAYS replaces extension
void add_extension(char *in, char *out, char *ext, int force);
```

An ASC file is opened by calling `asc_open_file()`, which returns 0 if it was successful. The file can be closed by `asc_close_file()`.

```
// opens ASC file (adds .ASC extension if none given)
// returns 0 if OK, -1 if error
int asc_open_file(char *fname);

// closes ASC file if open
void asc_close_file(void);
```

Each line in the ASC file is first read by `asc_read_line()`, which scans the file until it finds the first non-blank line. It separates out and returns a pointer to the first word in the line, which can be used to determine the data in the line.

```
// Starts new ASC file line, returns first word
// skips blank lines and comments
// returns "" if end of file
// NOTE: word string is volatile!
char *asc_read_line(void);
```

Sometimes a line being processed must be re-read by `asc_read_line()`. The file position can be restored by `asc_rewind_line()` so the current line can be read again.

```
// rewinds to start of line:
// asc_read_line() can again be used to read first word.
void asc_rewind_line(void);
```

24.3.2 Word Read and Compare

The first word of a line is returned by `asc_read_line()`. This word must be compared to expected line-type identifiers such as "ESACC", in order to determine how to process the line. The comparison is best done with the `match()` and `matchpart()` macros. These compare the target string argument to the variable `token`, where the return value of `asc_read_line()` should be stored. These macros perform a comparison without considering uppercase or lowercase characters. The `matchpart()` macro will only check the first characters in `token`, stopping when it reaches the end of the target string. For example, the target string "ES" would match "ESS" and "ES ET" but not "ET".

```
// this will check for full-word match
#define match(a)     (!_cmpnocase(token,a))

// this will check the first characters of the word
#define matchpart(a) (!_cmpnocasepart(token,a))
```

Numbers or words can be read from the current line using `asc_long()`, `asc_float()`, and `asc_-string()`. If the number was a missing value ("." in the ASC file) the value `MISSING_VALUE` is returned. Each number or word is read from the line from left to right. The entire line from the current read point can be fetched with `asc_rest_of_line()`.

```
// NOTE: when reading a float or long,
// if missing data ('.') or non-numerical values
// are encountered, MISSING_DATA is returned.
#define MISSING_DATA -32768

// reads integer or long value
// returns MISSING_VALUE if '.' or non-numeric
long asc_long(void);

// reads floating-point value
// returns MISSING_VALUE if '.' or non-numeric
double asc_float(void);

// returns pointer to next token (VOLATILE)
// returns "" if end of line
char *asc_string(void);

// returns pointer to rest of line text (VOLATILE)
// returns "" if end of line
char *asc_rest_of_line(void);
```

For lines of unknown length, the function `asc_at_eol()` can be called to test if there are any more words or numbers to read. When the line has been read, `asc_errors()` will return 0 if no errors were found, or the code of the last error found.

```
// returns 0 if more tokens available, 1 if at end of line
int asc_at_eol(void);

// returns 0 if no errors so far in line, else error code
int asc_errors(void);

#define NUMBER_EXPECTED -32000   // read string when number expected
#define LINE_TOO_SHORT  -32001   // missing token
#define SYNTAX_ERROR    -32002   // unexpected word
```

24.3.3 Reading Recording Configuration

The "START" line and several following lines in an ASC file contain information on the data that is available. These lines can be read with `asc_start_block()`, which should be called to finish reading any line with "START" as its first word.

```
// Before calling, the token "START" must have been read by asc_read_line()
// Scans the file for all block-start data
// Sets data flags, selects eye to use for event processing
// Returns: 0 if OK, else error encountered
int asc_start_block(void);
```

The `asc_start_block()` function processes the ASC file, setting these boolean variables to indicate what data is present in the trial:

```
// This reads all data associated with block starts
// It sets flags, and selects the eye to process

extern int block_has_left;        // nonzero if left eye data present
extern int block_has_right;     // nonzero if right eye data present

extern int block_has_samples;   // nonzero if samples present
extern int block_has_events;    // nonzero if events present

extern int samples_have_velocity;    // nonzero if samples have velocity data
extern int samples_have_resolution;  // nonzero if samples have resolution data
extern int events_have_resolution;   // nonzero if events have resolution data

extern int pupil_size_is_diameter;   // 0 if pupil units is area, 1 if diameter
```

For binocular recordings, one eye's data may need to be selected to be processed. If the variable `preferred_eye` is set to `LEFT_EYE` or `RIGHT_EYE`, then `asc_start_block()` will determine if that eye's data is available. If not, the other eye's data will be selected. The selected eye is stored in the variable `selected_eye`. You generally won't have to worry about eye selection, as the eye event-reading functions can do monocular eye data filtering.

```
// After opening the file, set prefferred_eye
// to the code for the eye you wish to process
// After starting a data block, selected_eye
// will contain the code for the eye that can be used
// (depends on preferred_eye and which eye(s) data is available)

#define LEFT_EYE  0     // codes for eyes (also index into sample data)
#define RIGHT_EYE 1

extern int preferred_eye;        // which eye's data to use if present
extern int selected_eye;        // eye to select events from
```

24.3.4 Reading Samples

If a line starts with a number instead of a word, it contains a sample. If your analyzer expects to read both samples and events, it should call `asc_read_sample()` to read each line. If this returns 1, it should then call `asc_read_line()` and process an event as usual.

The data from the sample is placed in the structure `a_sample`, of type `ASC_SAMPLE` defined below.

```
// Reads a file line, processes if sample
// Places data in the a_sample structure
// If not sample, rewinds line for event processing

// returns -1 if error, 0 if sample read,
// else 1 (not sample: use asc_read_line() as usual).
// x, y, p read to index of proper eye in a_sample
// For example, if right-eye data only,
// data is placed in a_sample.x[RIGHT_EYE],
// but not in a_sample.x[LEFT_EYE].
// Both are filled if binocular data.
int asc_read_sample(void);

typedef struct {
                UINT32 t;      // time of sample
                float x[2];    // X position (left and right eyes)
                float y[2];    // Y position (left and right eyes)
                float p[2];    // pupil size (left and right eyes)
                float resx;    // resolution (if samples_have_resolution==1)
                float resy;
                float velx[2]; // velocity (if samples_have_velocity==1)
                float vely[2]; // (left and right eyes)
               } ASC_SAMPLE;

extern ASC_SAMPLE a_sample;        // asc_read_sample() places data here
```

24.3.5 Reading Events

Data from ASC events may be read by special functions. Which function to call is determined by the first word read from the ASC file line with `asc_read_line()`. All event routines return 0 if no error occurred, or -1 if an error occurred. Eye data event readers can optionally filter out events from the non-processed eye in binocular recordings, and return 1 if the event is to be discarded.

A "BUTTON" line contains a button press or release event. The data from this line is stored in the variable `a_button`. This includes the time of the button press, the button number, and the state (0 if released and 1 if pressed).

```
// must have read "BUTTON" with asc_read_line() before calling
// returns -1 if error, 0 if read OK
int asc_read_button(void);

// "BUTTON" event data structure, filled by asc_read_button()
typedef struct {
                UINT32 t;    // time of button press
                int b;             // button number (1-8)
                int s;             // button change (1=pressed, 0=released)
               } ASC_BUTTON;

extern ASC_BUTTON a_button; // asc_read_button() places data here
```

Eye events are of two types: start events which contain only the time of a saccade, blink, or fixation; and end events which contain both start and end time (actually the time of the last sample fully within the saccade, blink or fixation), plus summary data. All eye event reading functions can filter the data by eye: If their argument is 1 then wrong-eye events will be discarded and 1 returned by the function.

All eye start events ("SSACC", "SFIX", and "SBLINK") are read by `asc_start_event()`, which fills in the variable `a_start`.

```
// must have read "SBLINK", "SSACC", or "SFIX"
// with asc_read_line() before calling
// returns -1 if error, 0 if skipped (wrong eye), 1 if read
// if <select_eye>==1, will skip unselected eye in binocular data
int asc_read_start(int select_eye);

// "SBLINK", "SSACC", "SFIX" events, read by asc_read_start()
typedef struct {
                int eye;     // eye
                UINT32 st;   // start time
               } ASC_START;

extern ASC_START a_start;   // asc_read_start() places data here
```

Fixation end events ("EFIX") are read by `asc_read_efix()` which fills the variable `a_efix` with the start and end times, and average gaze position, pupil size, and angular resolution for the fixation.

```
int asc_read_efix(int select_eye);

// "EFIX" event data structure, filled by asc_read_efix()
typedef struct {
                int eye;     // eye
                UINT32 st;   // start time
                UINT32 et;   // end time
                UINT32 d;    // duration
                float x;     // X position
                float y;     // Y position
                float p;     // pupil
                float resx;  // resolution (if events_have_resolution==1)
                float resy;
               } ASC_EFIX;
// Global event data, filled by asc_read functions

extern ASC_EFIX a_efix;     // asc_read_efix() places data here
```

Saccade end events ("ESACC") are read by `asc_read_esacc()` which fills the variable `a_esacc` with the start and end times, start and end gaze position, duration, amplitude, and peak velocity. Angular resolution may also be available.

```
// must have read "ESACC" with asc_read_line() before calling
// returns -1 if error, 0 if skipped (wrong eye), 1 if read
// if <select_eye>==1, will skip unselected eye in binocular data
int asc_read_esacc(int select_eye);


// "ESACC" event data structure, filled by asc_read_esacc()
typedef struct {
                int eye;     // eye
                UINT32 st;   // start time
                UINT32 et;   // end time
                UINT32 d;    // duration
                float sx;    // start X position
                float sy;    // start Y position
                float ex;    // end X position
                float ey;    // end Y position
                float ampl;  // amplitude in degrees
                float pvel;  // peak velocity, degr/sec
                float resx;  // resolution (if events_have_resolution==1)
                float resy;
               } ASC_ESACC;


extern ASC_ESACC a_esacc;   // asc_read_esacc() places data here
```

Blink end events ("EBLINK") mark the reappearance of the eye pupil. These are read by `asc_read_-eblink()` which fills the variable `a_eblink` with the start and end times, and duration. Blink events may be used to label the next "ESACC" event as being part of a blink and not a true saccade.

```
// must have read "EBLINK" with asc_read_line() before calling
// returns -1 if error, 0 if skipped (wrong eye), 1 if read
// if <select_eye>==1, will skip unselected eye in binocular data
int asc_read_eblink(int select_eye);

// "EBLINK" event data structure, filled by asc_read_eblink()
typedef struct {
                int eye;      // eye
                UINT32 st;    // start time
                UINT32 et;    // end time
                UINT32 d;     // duration
               } ASC_EBLINK;

extern ASC_EBLINK a_eblink; // asc_read_eblink() places data here
```

24.3.6 Rewinding and Bookmarks

It is common in experimental analysis to process a trial or an entire file more than once: for example, to determine statistical measures to reject outliers in the data. Several functions are supplied to allow rewinding of the ASC file processing to an earlier position.

The simplest is to rewind to the start of a trial (the "START" line, not the "TRIALID" message), which requires no setup.

```
int asc_rewind_trial(void);
```

You can also declare a variable of type BOOKMARK, then use it to record the current file position with `asc_set_bookmark()`. When `asc_goto_bookmark()` is called later, reading of the file will resume with the line where the bookmark was set. Bookmarks may be set anywhere inside or outside of a trial.

```
typedef struct {
                long fpos;
                long blkpos;

                int seleye;        // which eye's data to use if present
                int has_left;
                int has_right;
                int has_samples;
                int has_events;
                int vel;
                int sam_res;
                int evt_res;
                int pupil_dia;
               } BOOKMARK;

int asc_set_bookmark(BOOKMARK *bm);

int asc_goto_bookmark(BOOKMARK *bm);
```

24.4 A Sample ASC Analysis Application

The *sac_proc.c* source file implements a complete analyzer for an express saccade gap/overlap express saccade experiment. The experiment was run for one subject, then converted to *data.asc* using EDF2ASC. It should be built as either an MS-DOS program, or a Windows console application.

24.4.1 Planning the Analysis

The first step in writing an analyzer is to know how data is to be analyzed, what data is present, what measures are required, and under what circumstances the trial should be discarded.

In this experiment a target was displayed at the center of the display, and served as the drift correction target as well as part of the trial. After a short delay, a new target was drawn to the left or right of center. The center target was erased either before, at the same time as, or after the new target appeared. The subject ended the trial by pressing the left button (#2) or right button (#3).

For this very rudimentary analysis, the number of saccades made during the entire trial were counted, and the button response was scored as correct or incorrect. Trials were flagged if a blink had occurred at any time during recording. The most important measure is the time from the new target appearing to the start of the next saccade.

24.4.2 Typical Trial Data

This is the ASC file content for a typical trial:

```
MSG     2436129 TRIALID T1Rg200 0 0 220 200
START   2436164 LEFT RIGHT EVENTS
PRESCALER 1
VPRESCALER 1
EVENTS GAZE LEFT RIGHT
SFIX  L  2436164
SFIX  R  2436164
MSG      2436678 SYNCTIME
MSG      2436678 DRAWN NEW TARGET
EFIX  L  2436164 2436832        672       321.7   246.8    1422
EFIX  R  2436164 2436832        672       321.7   242.1    1683
SSACC L  2436836
SSACC R  2436836
ESACC R  2436836 2436872 40  323.6  247.4  496.5  250.2  6.75  276.4
SFIX  R  2436876
ESACC L  2436836 2436876 44  324.3  251.6  500.5  247.4  6.93  273.3
MSG      2436878 ERASED OLD TARGET
SFIX  L  2436880
EFIX  R  2436876 2437000 128 492.7  249.2  1682
SSACC R  2437004
EFIX  L  2436880 2437004 128 499.8  245.0  1323
SSACC L  2437008
ESACC L  2437008 2437028  24 506.6  242.2  565.4  251.1  2.35  151.4
ESACC R  2437004 2437028  28 493.9  248.5  551.7  258.4  2.29  147.2
SFIX  L  2437032
SFIX  R  2437032
EFIX  L  2437032 2437500        472       556.2   248.2    1281
EFIX  R  2437032 2437500        472       546.2   250.2    1653
BUTTON   2437512 2  1
MSG      2437521 ENDBUTTON 2
END      2437523 EVENTS RES  25.70  24.98
MSG      2437628 TRIAL_RESULT  2
MSG      2437628 TRIAL OK
```

These messages and events are placed in every trial:

- Each trial begins with a "TRIALID" message, with the first string being an ID for EDFVIEW, encoding the trial number, direction of motion, and gap or overlap time. This is followed by the block and trial number, the offset of the target (in pixels, negative for left and positive for right), and the delay between the target onset and fixation target offset.
- The "START" line and several following lines mark the start of recording, and encode the recording conditions for the trial.
- The "SYNCTIME" message marks the zero-time of the trial. This is the time the new target was drawn for this experiment.
- The "DRAWN NEW TARGET" message marks the time that the new, offset target was drawn, and "ERASED OLD TARGET" marks when the center fixation target was erased.
- Each saccade's end produced a "ESACC" line, which contains data on start and end time and gaze position, saccade amplitude in degrees, and peak velocity.
- The "END" line marks the end of recording.
- The "TRIAL_RESULT" reports the button press by the subject that ended the trial, or 0 if the subject allowed the trial to time out.
- The final message's first word "TRIAL" marks the end of data for the trial, and under what conditions the trial ended. The second word must be "OK" if the data is to be kept, indicating a successful trial.

24.4.3 Analyzing a File

The program begins by requesting the file to be analyzed. You can also supply it on the command line from the DOS prompt.

The function `process_file()` analyzes a singe ASC file. If the analyzer were designed to process multiple files, this function would be called once per file.

```
// call with file name to process
// returns error flag if can't open file
int process_file(char *fname)
{
        char ascname[120];
        char *token;
        long etime;

        add_extension(fname, ascname, "asc", 0);        // make file name
        if(asc_open_file(ascname)) return -1;               // can't open file!!!

        while(1)
        {
                token = asc_read_line();          // get first word on line
                if(token[0]==0) break;
                else if (match("MSG"))                // message
                {
                        etime = asc_long();                   // message time
                        token = asc_string();         // first word of message
                        if(match("TRIALID"))
                        {
                                process_trial();          // process trial data
                        }
                }           // IGNORE EVERYTHING ELSE
        }
        asc_close_file();
        return 0;
}
```

The extension ".asc" is added to the file name (unless an extension is already specified). The file is then opened with `asc_open_file()`.

The program now loops until it finds a message line. It does this by calling `asc_read_line()`, and checking if the first word in the line is "MSG". The time filed is read with `asc_long()`, and the first word of the message read with `asc_string()`. Your messages should all be designed so the first word identifies the message uniquely. In this case, we are looking for the "TRIALID" message that identifies a trial and marks its start. We could also look for other messages that were written at the start of the experiment or between trials by using `match()` for each case in this loop.

Once we have located a trial by finding the "TRIALID" message, we call `process_trial()` to read and analyze the trial.

24.4.4 Analyzing Trials

The first thing `process_trial()` does is to continue reading the trial information from the TRIALID line. After reading the numbers, a call to `asc_errors()` tests to see if any problems were encountered in reading the trial information.

```
  asc_string();                        // skip ID string
  block_number = asc_long();           // read data
  trial_number = asc_long();
  target_posn = asc_long();
  target_delay = asc_long();
  is_left = (target_posn < 320);       // left or right?
  if(asc_errors()) return -1;          // any errors?
```

In this example, analysis of the trial is handled by a loop that reads a line from the file, determines its type, and processes it. We have a number of data variables that are preset to a value that will not occur in the experiment, such as zero for the time of an event. These allow us to track the point in the trial we are in. In this analysis, the important variables are:

```
  long target_on_time = 0;             // set if new target has appeared
  long first_sacc_time = 0;     // set if a saccade occurred to the target
  int response_button = 0;             // set if a button was pressed
  long button_time = 0;         // when the button was pressed

  int num_saccades = 0;         // conts the number of saccades
  int have_blink = 0;                  // set if a blink occurred

  int is_left;                  // set if left target offset
  int correct_response = 0;     // set if correct button was pressed
              // THESE WERE READ FROM TRIALID MESSAGE
  int target_posn;                     // offset of new target
  int target_delay;                    // delay from new target drawn to old erased
  int trial_number;                    // trial and block number
  int block_number;

  char *token;  // variables for reading line elements
  long etime;
```

The "START" line marks the point where recording began. We process this line and any other recording data line by calling `asc_start_block()`. This will determine whether samples and events are available, and which eye(s) were recorded from.

Each "MSG" line contains a message. The time of the message and first word are read with `asc_long()` and `asc_string()`, and the first word checked with `match()` to determine how to process it. If the first word is "DRAWN", it marks the offset target being drawn and the time is recorded. The message "TRIAL_RESULT" records the response: a button press or timeout if zero.

If the first word in the message was "TRIAL", the second word determines the trial status and is read with `asc_string()`. If this is "OK" the trial was recorded successfully and we can process the trial data: in this case, we simply print it out. Otherwise, the trial was aborted and we return without further processing.

Each button press or release event line starts with the word "BUTTON", followed by the time, button number, and state. All this can be read by calling `asc_read_button()`, with the result placed in the `ASC_BUTTON` structure `a_button`. The button number is used to determine if the subject responded correctly.

Lines with "ESACC" as the first word contain end-of-saccade event data, including summary data on the saccade. Call `asc_read_esacc()` to read the event: this will return 0 checks if this saccade was produced by the correct eye (determined by the call to the `asc_read_start()` function), and reads the line's contents to the `ASC_ESACC` structure `a_esacc`. This will contain the start and end times and positions of the saccade, the amplitude, and the peak velocity. Saccades with amplitudes of less than 1° are ignored in this analysis. The time of the first large saccade that occurs after the offset target appeared is recorded in the variable `first_saccade`.

Finally, the "EBLINK" lines mark the end of a blink. In this example, we simply use this to flag that a blink occurred. In a more complex analysis, we would use this event to mark the next "ESACC" and belonging to a blink, not a saccade.

This is the entire analysis loop:

```
while(1)
{
        token = asc_read_line();          // get first word on line
        if(token[0]==0) return -1;        // end of file

        else if (match("START"))                // START: select eye to process
        {
                asc_start_block();
        }

        else if (match("MSG"))                // message
        {
                etime = asc_long();                    // message time
                token = asc_string();         // first word of message

                if(match("DRAWN"))                // new target drawn
                {
                        target_on_time = etime;
                }
                else if(match("TRIAL_RESULT")) // trial result
                {
                        response_button = asc_long();
                }
                else if(match("TRIAL"))        // trial is OK?
                {
                        token = asc_string();
                        if(match("OK") && response_button!=0) // report data, only if OK
                        {
                                // A VERY SIMPLE DATA REPORT: FORMAT AS REQUIRED.

                                printf("trial:%d delay:%d  sac_rt:%ld but_rt:%ld corr:%d  nsac:%d
                                        trial_number, target_delay, first_sacc_time-target_on_time
                                        button_time-target_on_time,correct_response,num_saccades,h
                        }
                        return 0;                // done trial!
                }
                }

                else if (match("BUTTON"))                // button
                {
                        asc_read_button();
                        if(a_button.s==1 && (a_button.b==3 || a_button.b==2))
                        {
                                button_time = a_button.t;
                                response_button = a_button.b;
                                correct_response = ((is_left==1 && a_button.b==2) || (is_left==0 && a_butt
                        }
                }

                else if (match("ESACC"))        // end saccade
                {
                        if(asc_read_esacc(1)) continue;  // skip if wrong eye

                        // ignore if smaller than 1 degree
                        // or if we haven't displayed target yet
                        if(a_esacc.ampl>1.0 && target_on_time>0)
                        {
                                num_saccades++;
                                if(num_saccades==1) first_sacc_time = a_esacc.st;
                        }
                }
                else if (match("EBLINK"))        // blink
                {
                        have_blink++;
                }
                // IGNORE EVERYTHING ELSE
        }
}
```

Chapter 25

Useful EyeLink Commands

25.1 Useful EyeLink Commands

These commands may be sent by the eyecmd_printf() function to the EyeLink tracker. You will need to use only a few of these: rarely more than those used in the template source.

For current information on the EyeLink tracker configuration, examine the ∗.INI files in the EYELINK\EXE\ directory of the eye tracker computer.

- Calibration Setup
- Configuring Key and Buttons
- Display Setup
- File Open and Close
- Tracker Configuration
- Drawing Commands
- File Data Control
- Link Data Control
- Parser Configuration

25.2 Calibration Setup

25.2.1 calibration_type

```
calibration_type = <type>
```

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

Arguments:

- `<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
  - HV13: 13-point calibration for large calibration region (EyeLink II version 2.0 or later; EyeLink 1000)

25.2.2 gaze_constraint

```
x_gaze_constraint = <position>
y_gaze_constraint = <position>
```

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

Arguments:

- `<position>`: a gaze coordinates, or AUTO

25.2.3 horizontal_target_y

```
horizontal_target_y = <position>
```

Sets the Y position (vertical display coordinate) for computing targets positions when the H3 calibration mode is set. Set before issuing the "calibration_type = H3" command.

Arguments:

- `<position>`: Y display coordinate for targets in H3 calibration mode

25.2.4 enable_automatic_calibration

```
enable_automatic_calibration = <YES or NO>
```

"YES" enables auto-calibration sequencing, "NO" forces manual calibration sequencing.

Arguments:

- `YES` or `NO`

25.2.5 automatic_calibration_pacing

```
automatic_calibration_pacing = <time>
```

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

Arguments:

- `<time>`: shortest delay

25.3 Configuring Key and Buttons

See the tracker file "BUTTONS.INI" and "KEYS.INI" for examples.

25.3.1 key_function

```
key_function <keyspec> <command>
```

Look at the tracker "KEYS.INI" file for examples.

Arguments:

- `<keyspec>`: key name and modifiers
- `<command>`: command string to execute when key pressed

25.3.2 create_button

```
create_button <button> <ioport> <bitmask> <inverted>
```

Defines a button to a bit in a hardware port.

Arguments:

- `<button>`: button number, 1 to 8
- `<ioport>`: address of hardware port
- `<bitmask>`: 8-bit mask ANDed with port to test button line
- `<inverted>`: 1 if active-low, 0 if active-high

25.3.3 button_function

```
button_function <button> <presscmd> <relcmd>
```

Assigns a command to a button. This can be used to control recording with a digital input, or to let a button be used instead of the spacebar during calibration.

Arguments:

- `<button>`: hardware button 1 to 8, keybutton 8 to 31
- `<presscmd>`: command to execute when button pressed
- `<relcmd>`: command to execute when button released

25.3.4 button_debounce_time

```
button_debounce_time = <delay>
```

Sets button debounce time. Button responds immediately to first change, then is ignored until it is stable for this time.

Arguments:

- `<delay>`: debounce in milliseconds.

25.3.5 write_ioport

```
write_ioport <ioport> <data>
```

Writes data to I/O port. Useful to configure I/O cards.

Arguments:

- `<ioport>`: byte hardware I/O port address
- `<data>`: data to write

25.3.6 read_ioport

```
read_ioport <ioport>
```

Performs a dummy read of I/O port. Useful to configure I/O cards.

Arguments:

- `<ioport>`: byte hardware I/O port address

25.4 Display Setup

25.4.1 screen_pixel_coords

```
screen_pixel_coords = <left> <top> <right> <bottom>
```

Sets the gaze-position coordinate system, which is used for all calibration target locations and drawing commands. Usually set to correspond to the pixel mapping of the subject display. Issue the calibration_-type command after changing this to recompute fixation target positions.

You should also write a DISPLAY_COORDS message to the start of the EDF file to record the display resolution.

Arguments:

- `<left>`: X coordinate of left of display area
- `<top>`: Y coordinate of top of display area
- `<right>`: X coordinate of right of display area
- `<bottom>`: Y coordinate of bottom of display area

25.4.2 screen_write_prescale

```
screen_write_prescale = <multiplier>
```

Sets the value by which gaze position data is multiplied before writing to EDF file or link as integer. This is nominally 10, but should be reduced for displays with greater than 1500x1500 pixels, or increased for gaze coordinate systems with less than 150x150 units.

Arguments:

- `<multiplier>`: integer value, 1 to 1000

25.5 File Open and Close

25.5.1 open_data_file

```
open_data_file <name>
```

Opens an EDF file, closes any existing file

Arguments:

- `<name>`: name of data file

25.5.2 add_file_preamble_text

```
add_file_preamble_text <text>
```

Must be used immediately after open_data_file, to add a note on file history.

Arguments:

- `<text>`: text to add, in quotes

25.5.3 close_data_file

```
close_data_file
```

Closes any open EDF file

Arguments:

- None

25.5.4 data_file_path

```
data_file_path = <path>
```

Can be used to set where EDF files are to be written. Any directory information in file name overrides

Arguments:

- `<path>`: directory or drive path in quotes, "." for current directory

25.6 Tracker Configuration

These commands are used to set tracker configuration. This should be done at the start of each experiment, in case the default setting were modified by a previous experiment.

25.6.1 Configuration: Eyes Tracked

```
active_eye = <LEFT or RIGHT>
```

Controls which eye is recorded from in monocular mode

Arguments:

- `LEFT` or `RIGHT` monocular eye selection

```
binocular_enabled = <YES or NO>
```

Controls whether in monocular or binocular tracking mode .

Arguments:

- `YES` for binocular tracking
- `NO` for monocular tracking

25.6.2 Anti-Reflection Control

Remarks:
EyeLink I only!!!

```
head_subsample_rate = <0, -1, or 4>
```

Can be used to disable monitor marker LEDS, or to control antireflection option.

Arguments:

- `0` for normal operation
- `4` for antireflection on
- `-1` to turn off markers

25.6.3 heuristic_filter (EyeLink I)

```
heuristic_filter = <ON or OFF>
```

Can be used to disable filtering, reduces system delay by 4 msec. **NEVER** TURN OFF THE FILTER WHEN ANTIREFLECTION IS TURNED ON.

Arguments:

- `ON` enables filter (usual)
- `OFF` disables filter

25.6.4 heuristic_filter (EyeLink II, EyeLink1000)

```
heuristic_filter = <ON or OFF>
heuristic_filter = <linkfilter>
heuristic_filter = <linkfilter> <filefilter>
```

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.

Arguments:

- `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).

25.6.5 pupil_size_diameter

```
pupil_size_diameter = <YES or NO>
```

Arguments:

- `YES` to convert pupil area to diameter
- `NO` to output pupil area data

25.6.6 simulate_head_camera

simulate_head_camera = <YES or NO> Can be used to turn off head tracking if not used. Do this before calibration

Arguments:

- `YES` to disable head tracking
- `NO` to enable head tracking

25.6.7 simulation_screen_distance

```
simulation_screen_distance = <mm>
```

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

Arguments:

- <`mm`>: simulated distance from display to subject in millimeters.

25.7 Drawing Commands

25.7.1 echo

```
echo <text>
```

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

Arguments:

- `<text>`: text to print in quotes

25.7.2 print_position

```
print_position <column><line>
```

Coordinates are text row and column, similar to C gotoxy() function.

Remarks:
Row cannot be set higher than 25.

Use "draw_text" command to print anywhere on the tracker display. **Arguments:**

- `<col>`: text column, 1 to 80
- `<row>`: text line, 1 to 25

25.7.3 clear_screen

```
clear_screen <color>
```

Clear tracker screen for drawing background graphics or messages.

Arguments:

- `<color>`: 0 to 15

25.7.4 draw_line

```
draw_line <x1> <y1> <x2> <y2> <color>
```

Draws line, coordinates are gaze-position display coordinates.

Arguments:

- `<x1>`,`<y1>`: start point of line
- `<x2>`,`<y2>`: end point of line
- `<color>`: 0 to 15

25.7.5 draw_box

```
draw_box <x1> <y1> <x2> <y2> <color>
```

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

Arguments:

- `<x1>`,`<y1>`: corner of box
- `<x2>`,`<y2>`: opposite corner of box
- `<color>`: 0 to 15

25.7.6 draw_filled_box

```
draw_filled_box <x1> <y1> <x2> <y2> <color>
```

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

Arguments:

- `<x1>`,`<y1>`: corner of box
- `<x2>`,`<y2>`: opposite corner of box
- `<color>`: 0 to 15

25.7.7 draw_line

```
draw_line <x1> <y1> <x2> <y2> <color>
```

Draws line, coordinates are gaze-position display coordinates.

Arguments:

- `<x1>`,`<y1>`: start point of line
- `<x2>`,`<y2>`: end point of line
- `<color>`: 0 to 15

25.7.8 draw_text

```
draw_text <x1> <y1> <color> <text>
```

Draws text, coordinates are gaze-position display coordinates.

Arguments:

- `<x1>`,`<y1>`: center point of text
- `<color>`: 0 to 15
- `<text>`: text of line, in quotes

25.7.9 draw_cross

```
draw_cross <x> <y>
```

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

Arguments:

- `<x1>`,`<y1>`: center point of cross
- `<color>`: 0 to 15

25.8 File Data Control

25.8.1 file_sample_data

```
file_sample_data = <list>
```

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

Arguments:

- `<list>`: list of data types
  - GAZE screen xy (gaze) position
  - GAZERES units-per-degree screen resolution
  - HREF head-referenced eye position data
  - 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
  - HTARGET Head target data. Reports target distance and X/Y position for EyeLink Remote eye tracker.

25.8.2 file_event_data

```
file_event_data = <list>
```

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

Arguments:

- `<list>`: list of data types
  - 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

25.8.3 file_event_filter

```
file_event_filter = <list>
```

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

Arguments:

- `<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

25.8.4 mark_playback_start

```
mark_playback_start
```

NEW for EyeLink I v2.1, EyeLink II v1.1 Marks the location in the data file from which playback will begin at the next call to eyelink_playback_start(). 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 software is shut down.

Arguments:

- NONE

25.9 Link Data Control

25.9.1 link_sample_data

```
link_sample_data = <list>
```

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

Arguments:

- `<list>`: list of data types
  - `GAZE` screen xy (gaze) position
  - `GAZERES` units-per-degree screen resolution
  - `HREF` head-referenced eye position data
  - `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
  - `HTARGET` Head target data. Reports target distance and X/Y position for EyeLink Remote eye tracker.

25.9.2 link_event_data

```
link_event_data = <list>
```

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

Arguments:

- `<list>`: list of data types
  - `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

25.9.3 link_event_filter

```
link_event_filter = <list>
```

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

Arguments:

- `<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

25.9.4 recording_parse_type

```
recording_parse_type = <type>
```

Sets how velocity information for saccade detection is to be computed. Almost always left to GAZE.

Arguments:

- `<type>`: GAZE or HREF

25.9.5 link_nonrecord_events

```
link_nonrecord_events = <list>
```

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 EyeLink1000, and messages cannot be enabled for versions of EyeLink I before v2.1..

Arguments:

- `<list>`: event types seperated by spaces or commas:
  - `MESSAGE` messages (user notes in file)
  - `BUTTON` button 1..8 press or release
  - `INPUT` changes in input port lines

25.10 Parser Configuration

Remarks:

For EyeLink II and EyeLink1000, you should use select_parser_configuration rather than the other commands listed below.

25.10.1 select_parser_configuration

```
select_parser_configuration = <set>
```

EyeLink II and EyeLink1000 ONLY! Selects the preset standard parser setup (0) or more sensitive saccade detector (1). These are equivalent to the cognitive and psychophysical configurations listed below.

Arguments:

- `<set>`: 0 for standard, 1 for high sensitivity saccade detector configuration

The remaining commands are documented for use with EyeLink I only.

25.10.2 saccade_velocity_threshold

```
saccade_velocity_threshold = <vel>
```

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

Arguments:

- `<vel>`: minimum velocity ($^\circ/sec$) for saccade

25.10.3 saccade_acceleration_threshold

```
saccade_acceleration_threshold = <accel>
```

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

Arguments:

- `<accel>`: minimum acceleration ($^\circ/sec/sec$) for saccades

25.10.4 saccade_motion_threshold

```
saccade_motion_threshold = <deg>
```

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

Arguments:

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

25.10.5 saccade_pursuit_fixup

```
saccade_pursuit_fixup = <maxvel>
```

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

Arguments:

- `<maxvel>`: maximum pursuit velocity fixup ($^\circ/sec$)

25.10.6 fixation_update_interval

```
fixation_update_interval = <time>
```

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

Arguments:

- `<time>`: milliseconds between fixation updates, 0 turns off

25.10.7 fixation_update_accumulate

```
fixation_update_accumulate = <time>
```

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

Arguments:

- `<time>`: milliseconds to collect data before fixation update for average gaze position

25.10.8 Typical Parser Configurations

These are suggested settings for the EyeLink I parser. For EyeLink II, you should use select_parser_-configuration rather than the other commands listed below. The Cognitive configuration is optimal for visual search and reading, and ignores most saccades smaller than 0.5°. It is less sensitive to set up problems. The Pursuit configuration is designed to detect small ($<0.25^\circ$) saccades, but may produce false saccades if subject setup is poor.

Cognitive Configuration:

```
recording_parse_type = GAZE
saccade_velocity_threshold = 30
saccade_acceleration_threshold = 9500
saccade_motion_threshold = 0.15
saccade_pursuit_fixup = 60
fixation_update_interval = 0
```

Pursuit and Neurological Configuration:

```
recording_parse_type = GAZE
saccade_velocity_threshold = 22
saccade_acceleration_threshold = 5000
saccade_motion_threshold = 0.0
saccade_pursuit_fixup = 60
fixation_update_interval = 0
```

Chapter 26

Function List

26.1 Initialize EyeLink Library

26.1.1 Detailed Description

These methods are used to initialize the EyeLink Library.

Defines

- #define LINK_INITIALIZE_FAILED -200
- #define CONNECT_TIMEOUT_FAILED -201
- #define WRONG_LINK_VERSION -202
- #define TRACKER_BUSY -203

Functions

- UINT16 open_eyelink_system (UINT16 bufsize, char FARTYPE ∗options)
- void eyelink_set_name (char FARTYPE ∗name)
- void close_eyelink_system (void)
- INT16 eyelink_open_node (ELINKADDR node, INT16 busytest)
- INT16 eyelink_open (void)
- INT16 eyelink_broadcast_open (void)
- INT16 eyelink_dummy_open (void)
- INT16 eyelink_close (INT16 send_msg)
- INT16 eyelink_is_connected (void)
- INT16 eyelink_quiet_mode (INT16 mode)
- INT16 eyelink_poll_trackers (void)
- INT16 eyelink_poll_remotes (void)
- INT16 eyelink_poll_responses (void)
- INT16 eyelink_get_node (INT16 resp, void FARTYPE ∗data)
- INT16 eyelink_get_tracker_version (char FARTYPE ∗c)
- void eyelink_set_tracker_node (ELINKADDR node)
- INT16 open_eyelink_connection (INT16 mode)
- void close_eyelink_connection (void)
- INT16 set_eyelink_address (char ∗addr)

Variables

- ELINKADDR eye_broadcast_address
- ELINKADDR rem_broadcast_address
- ELINKADDR our_address

26.1.2 Define Documentation

26.1.2.1 #define CONNECT_TIMEOUT_FAILED -201

CONNECTION ERROR CODES: timed out waiting for reply

26.1.2.2 #define LINK_INITIALIZE_FAILED -200

CONNECTION ERROR CODES: can't use link

26.1.2.3 #define TRACKER_BUSY -203

CONNECTION ERROR CODES: tracker already connected

26.1.2.4 #define WRONG_LINK_VERSION -202

CONNECTION ERROR CODES: wrong TSR or source version

26.1.3 Function Documentation

26.1.3.1 void close_eyelink_connection (void)

Closes any connection to the eye tracker, and closes the link.

Remarks:
NEW (v2.1): Broadcast connections can be closed, but not to affect the eye tracker. If a non-broadcast (primary) connection is closed, all broadcast connections to the tracker are also closed.

Example: See `open_eyelink_connection()`

See also:
`open_eyelink_connection()` and `eyelink_close()`

26.1.3.2 void close_eyelink_system (void)

Resets the EyeLink library, releases the system resources used by the millisecond clock.

Remarks:
MUST DO BEFORE EXITING.

Example: See `open_eyelink_system()`

See also:
`open_eyelink_system()`, `eyelink_close()` and `set_offline_mode()`

26.1.3.3 INT16 eyelink_broadcast_open (void)

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.

Remarks:

May not function properly, if there are more than one Ethernet cards installed.

Returns:

`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.

Example:

```
// This program illustrates the use of eyelink_broadcast_open(); see the COMM_SIMPLE and
// BROADCAST templates for more examples

#include <eyelink.h>
#include <stdio.h>

// Initialize the EyeLink DLL
if(open_eyelink_connection(-1))
    return -1;

// set our network name
eyelink_set_name("Broadcast");

...
// Extra code here to check for the tracker status or
// wait for the go-signal from the other application

// Starts the broadcast connection to the tracker
if(eyelink_broadcast_open())
{
    printf("Cannot open broadcast connection to tracker");
    return -1;
}

// enable link data reception by EyeLink DLL
eyelink_reset_data(1);
// NOTE: this function can discard some link data
eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS);
```

See also:

`eyelink_close()`, `eyelink_dummy_open()`, `eyelink_is_connected()`, `eyelink_open()` and `eyelink_open_node()`

26.1.3.4 INT16 eyelink_close (INT16 *send_msg*)

Sends a disconnect message to the EyeLink tracker, resets the link data system.

Parameters:

send_msg Usually `<send_msg>` is `1`. However, `0` can be used to reset the eyelink system if it is listening on a broadcast session.

Returns:

0 if successful, otherwise link error.

Example:

```
// This program illustrates the use of eyelink_close

#include <eyelink.h>

// Checks whether the tracker is still connected
if(eyelink_is_connected())
{
   set_offline_mode();          // off-line mode
   eyecmd_printf("close_data_file");
   eyelink_close(1);         // disconnect from tracker
}
```

See also:

eyelink_broadcast_open(), eyelink_open(), eyelink_dummy_open() and eyelink_open_node()

26.1.3.5 INT16 eyelink_dummy_open (void)

Sets the EyeLink library to simulate an eyetracker connection. Functions will return plausible values, but no data.

Remarks:

The function eyelink_is_connected() will return −1 to indicate a simulated connection.

Returns:

Always returns 0.

Example:

```
// This program illustrates the use of eyelink_dummy_open

#include <eyelink.h>

// Initialize the EyeLink DLL; otherwise the application will crash
// when using the eyelink_dummy_open()function!
if(open_eyelink_connection(-1))
    return -1;

// Opens a "dummy" connection for debugging the application
eyelink_dummy_open();
```

See also:

eyelink_broadcast_open(), eyelink_open() and eyelink_open_node()

26.1.3.6 INT16 eyelink_get_node (INT16 *resp*, void FARTYPE ∗ *data*)

Reads the responses returned by other trackers or remotes in response to eyelink_poll_trackers() or eyelink_poll_remotes(). 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.

data Pointer to buffer of ELINKNODE type, to hold name and ELINKADDR of the respondent.

Returns:

0 if OK, −1 if node response number is too high.

Example: See eyelink_poll_remotes()

See also:

eyelink_node_receive(), eyelink_node_send() and eyelink_poll_remotes()

26.1.3.7 INT16 eyelink_get_tracker_version (char FARTYPE ∗ *c*)

After connection, determines if the connected tracker is an EyeLink I or II.

Remarks:

For the EyeLink II tracker, it can optionally retrieve the tracker software version.

Parameters:

c NULL, or pointer to a string (at least 40 characters) to hold the version string. This will be "EYELINK I" or "EYELINK II x.xx", where "x.xx" is the software version.

Returns:

0 if not connected, 1 for EyeLink I, 2 for EyeLink II.

Example:

```
// This program illustrates the use of eyelink_get_tracker_version

#include <eyelink.h>

int i;
char version_info[256];

i = eyelink_get_tracker_version(version_info);
eyemsg_printf("EyeLink %d version %s", i, version_info);
```

Output:

```
MSG     500850 EyeLink 2 version EYELINK II 1.10
```

26.1.3.8 INT16 eyelink_is_connected (void)

Checks whether the connection to the tracker is alive.

Remarks:

Call this routine during loops and wherever the experiment might lock up if the tracker is shut down. Exit the experiment (by terminating loops and returning from all calls) if this returns 0.

Returns:

0 if link closed.
-1 if simulating connection.
1 for normal connection.
2 for broadcast connection (NEW for v2.1 and later).

Example:

```
// This program illustrates the use of eyelink_is_connected( when performing a pre-trial drift cor

#include <eyelink.h>

while(1)
{
    // Checks link often so we can exit if tracker stopped
    if(!eyelink_is_connected())
        return ABORT_EXPT;

    // Performs drift correction with target drawn in the center
    error = do_drift_correct(SCRWIDTH/2, SCRHEIGHT/2, 1, 1);

    // repeat if ESC was pressed to access Setup menu
    if(error!=27) break;
}
```

See also:

`eyelink_close()` and `eyelink_open()`

26.1.3.9 INT16 eyelink_open (void)

Attempts to open a link connection to the EyeLink tracker. Simple connect to single Eyelink tracker. Equivalent to `eyelink_open_node(broadcast_address, 0)`.

Remarks:

The tracker address can be set by calling the function `set_eyelink_address()`. If this address was "255.255.255.255" the call to `eyelink_open()` will "broadcast" a request to any tracker, however this may fail if multiple Ethernet cards are installed. For the broadcast option to work use EyeLink I v2.1 or higher, EyeLink II v1.1 or higher. Before using this command, call either `open_eyelink_connection`(-1) or `open_eyelink_system` () to prepare the link for use.

Returns:

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.

Example:

```
// This program illustrates the use of eyelink_open()

#include <eyelink.h>

// Initialize the EyeLink DLL
if(open_eyelink_connection(-1))
    return -1;
```

```
// Set the address of the tracker
set_eyelink_address("100.1.1.1");

// connect to single Eyelink tracker
if (eyelink_open())
    return -1;
```

See also:

eyelink_broadcast_open(), eyelink_close(), eyelink_dummy_open(), eyelink_open_node(), open_eyelink_connection(), open_eyelink_system() and set_eyelink_address()

26.1.3.10 INT16 eyelink_open_node (ELINKADDR *node*, INT16 *busytest*)

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

Parameters:

node Must be an address returned by eyelink_poll_trackers() and eyelink_poll_-responses() to connect to any tracker.

busytest If non-zero the call to eyelink_open_node() will not disconnect an existing connection.

Returns:

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.
TRACKER_BUSY if <busytest> is set, and tracker is connected to another computer.

Example:

```
// This program illustrates the case of making connection using an ELINKADDR tracker node address

ELINKADDR node;                          // EyeLink address node

// Initialize the EyeLink DLL
if(open_eyelink_connection(-1))
    return -1;

// Converts tracker IP address to an EyeLink node
text_to_elinkaddr("100.1.1.1", node, 0);

// Makes connection using an ELINKADDR tracker node address; checks
// whether the tracker is already connected with another application
if (eyelink_open_node(node, 1))
    return -1;
```

See also:

eyelink_node_receive(), eyelink_node_send(), eyelink_node_send_-message() and eyelink_open()

26.1.3.11 INT16 eyelink_poll_remotes (void)

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

Returns:

0 if successful, otherwise link error.

Example:

```
// This program illustrates the use of eyelink_poll_remotes(). See COMM_SIMPLE/COMM_LISTERNER
// templates for the complete example

#include <eyelink.h>

ELINKADDR listener_address;  // Address of listener application

// Finds the listener application
int check_for_listener(void)
{
    int i, n;
    ELINKNODE node;  // This will hold application name and address

    eyelink_poll_remotes(); // Polls network for EyeLink applications
    pump_delay(500);        // Gives applications time to respond

    n = eyelink_poll_responses();  // How many responses?
    //  Responses 1 to n are from other applications
    for(i = 1; i<= n; i++)
    {
        if(eyelink_get_node(i, &node) < 0)
            return -1;  // error: no such data

        // Found the "Comm_listener" application
        if(!_stricmp(node.name, "comm_listener"))
        {
            // Stores the listener's address
            memcpy(listener_address, node.addr, sizeof(ELINKADDR));

            // Send a message to the listener
            eyelink_node_send(listener_address, "NAME comm_simple", 40);

            ...
            // Other code here to check responses from the listener
            ...

            return 0;   // all communication checks out
        }
    }
    return -1;    // no listener node found
}
```

See also:

eyelink_poll_responses() and eyelink_poll_trackers()

26.1.3.12 INT16 eyelink_poll_responses (void)

Returns the count of node addresses received so far following the call of eyelink_poll_remotes() or eyelink_poll_trackers().

Remarks:

You should allow about 100 milliseconds for all nodes to respond. Up to 4 node responses are saved.

Returns:

Number of nodes responded. 0 if no responses.

Example: See `eyelink_poll_remotes()`

See also:

`eyelink_poll_remotes()` and `eyelink_poll_trackers()`

26.1.3.13 INT16 eyelink_poll_trackers (void)

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

Returns:

`0` if successful, otherwise link error.

Example: See `eyelink_poll_remotes()`

See also:

`eyelink_poll_remotes()` and `eyelink_poll_responses()`

26.1.3.14 INT16 eyelink_quiet_mode (INT16 *mode*)

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

Remarks:

This is used in combination with broadcast mode (multiple applications connected to one tracker) to ensure that "listener" applications do not inadvertently send commands, key presses, or messages to the tracker. This is mostly useful when quickly converting an existing application into a listener.

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.

Returns:

The previous settings.

Example:

```
// This program illustrates the use of eyelink_quiet_mode() to control message sending from a Broa
// application to the tracker

#include <eyelink.h>
#include <stdio.h>

// Starts the broadcast connection to the tracker
if(eyelink_broadcast_open())
{
    printf("Cannot open broadcast connection to tracker");
    return -1;
}

// enable link data reception by EyeLink DLL
eyelink_reset_data(1);
```

```
// NOTE: this function can discard some link data
eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS);

// The following codes test the message sending from the BROADCAST
// application to tracker with eyelink_quiet_mode()
eyelink_quiet_mode(0);  // Allows for message sending
eyelink_send_message("This should be recorded in the EDF file");
eyelink_quiet_mode(2);   // Disables message sending
eyelink_send_message("This should not appear in the EDF file");
eyelink_quiet_mode(0);   // Allows for message sending again
eyelink_send_message("This should appear in the EDF file");
```

Output:

```
MSG     3304947 This message should be recorded in the EDF file
MSG     3304947 This message should appear in the EDF file
```

See also:
`eyelink_broadcast_open()` and `eyelink_open()`

26.1.3.15 void eyelink_set_name (char FARTYPE ∗ *name*)

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

Parameters:
name String to become new name.

Example:

```
// This program illustrates the use of eyelink_set_name()
#include <eyelink.h>

ELINKNODE node;  // This will hold application name and address

// Sets the EyeLink host address, if tracker IP address is different
// from the default "100.1.1.1"
if (set_eyelink_address("100.1.1.7"))
    return -1;

// Initializes EyeLink library, and opens connection to the tracker
if(open_eyelink_connection(0))
    return -1;

// set our EyeLink node name
eyelink_set_name("Newapplication");

// Verify the name set by getting copy of node data
// Since we are checking our own data, set resp = 0
if(eyelink_get_node(0, &node) == OK_RESULT)
    eyemsg_printf("%s", node.name);
```

Output:

```
MSG     2248248 Newapplication
```

See also:
`eyelink_get_node()`, `open_eyelink_connection()` and `set_eyelink_address()`

26.1.3.16 void eyelink_set_tracker_node (ELINKADDR *node*)

Address used for non-connected time requests and message sends. the "proper" way to do this is with the "node" type of functions but we allow a "back door" to simplify higher level support functions. This is also the address used under Windows for looking for tracker (an IP broadcast is used on all other platforms). There is a bug in the Windows networking, causing broadcasts sent on all cards to have the IP source addres of only the first card. This means the tracker sends its connection reply to the wrong address. So the exact address or a subnet broadcast address (i.e. 100.1.1.255 for a subnet mask of 255.255.255.0) needs to be set to that of the tracker.

26.1.3.17 INT16 open_eyelink_connection (INT16 *mode*)

Initializes the EyeLink library, and opens a connection to the EyeLink tracker.

Remarks:

By setting `<mode>` to be `1`, the connection can be simulated for debugging purposes. Only timing operations and simple tests should be done in simulation mode, and the Windows TCP/IP system must be installed. This function is intended for networks where a single tracker is connected to the network.

Parameters:

mode Mode of connection:

- `0`, opens a connection with the eye tracker;
- `1`, will create a dummy connection for simulation;
- `-1`, initializes the DLL but does not open a connection.

Returns:

`0` if success, else error code

Example:

```
// This program illustrates the use of open_eyelink_connection()

#include <eyelink.h>

// Sets the EyeLink host address, if tracker IP address is different
// from the default "100.1.1.1"
if (set_eyelink_address("100.1.1.7"))
    return -1;

//Initializes EyeLink library, and opens connection to the tracker
if(open_eyelink_connection(0))
    return -1;

...
// Code for the setup, recording, and cleanups
close_eyelink_connection();      // disconnect from tracker
```

See also:

`close_eyelink_connection()`, `close_eyelink_system()`, `eyelink_broadcast_open()`, `eyelink_dummy_open()`, `eyelink_open()` and `eyelink_open_node()`

26.1.3.18 UINT16 open_eyelink_system (UINT16 *bufsize*, char FARTYPE ∗ *options*)

Use this function to initialize the EyeLink library. This will also start the millisecond clock. No connection is attempted to the eyetracker yet. It is preferable to call open_eyelink_connection(-1) instead, as this prepares other parts of the DLL for use.

Remarks:
MUST BE FIRST CALL TO LINK INTERFACE.

Parameters:
bufsize Size of sample buffer. `60000` is the maximum allowed. (`0` for default)

options Text specifying initialization options. Currently no options are supported. (`NULL` or "" for defaults)

Returns:
`0` if failed, nonzero (`-1` or TSR SWI number) if success.

Example:

```
// This program shows the use of open_eyelink_system() and close_eyelink_system ()

#include <eyelink.h>
int dummy = 0;  // Sets the connection type

// Initialize the EyeLink library
if(!open_eyelink_system(0, ""))
{
    printf("\nERROR: Cannot initialize eyelink library\n");
    return -1;
}

if(dummy)
    i = eyelink_dummy_open(); // Starts a dummy connection
else
    i = eyelink_open();  // Connects to the tracker

...
// Code for the setup, recording, and cleanups

// Now, the following code closes the eyelink connection
// Checks whether we still have the connection to the tracker
if(eyelink_is_connected())
{
    // Places EyeLink tracker in off-line (idle) mode
    set_offline_mode();
    eyecmd_printf("close_data_file");    // close data file
    eyelink_close(1);         // disconnect from tracker
}

// shut down system (MUST do before exiting)
close_eyelink_system();
```

See also:
`close_eyelink_connection()`, `close_eyelink_system()`, `eyelink_close()` and `eyelink_open()`

26.1.3.19 INT16 set_eyelink_address (char ∗ *addr*)

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.

Remarks:

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:

addr Pointer to a string containing a "dotted" 4-digit IP address.

Returns:

`0` if success, `-1` if could not parse address string.

Example:

```
// This program illustrates the use of set_eyelink_address()

#include <eyelink.h>

// Initialize the EyeLink DLL
if(open_eyelink_connection(-1))
    return -1;

// Set the address of the tracker
set_eyelink_address("100.1.1.1");

// connect to single Eyelink tracker
if (eyelink_open())
    return -1;
```

See also:

`eyelink_open()`, `eyelink_open_node()`, `text_to_elinkaddr()`

26.1.4 Variable Documentation

26.1.4.1 ELINKADDR eye_broadcast_address

The broadcast address for the eye trackers.

26.1.4.2 ELINKADDR our_address

This EyeLink node's address for the link implementation.

26.1.4.3 ELINKADDR rem_broadcast_address

The broadcast address for the remotes.

26.2 Access Local Time

26.2.1 Detailed Description

These functions are used to access display time

Data Structures

- struct MICRO

Defines

- #define current_msec() current_time()

Functions

- UINT32 current_time (void)
- UINT32 current_micro (MICRO FARTYPE ∗m)
- UINT32 current_usec (void)
- void msec_delay (UINT32 n)
- double current_double_usec (void)
- INT16 eyelink_reset_clock (INT16 enable)

26.2.2 Define Documentation

26.2.2.1 #define current_msec() current_time()

See `current_time()`

26.2.3 Function Documentation

26.2.3.1 double current_double_usec (void)

Returns the current microsecond as double (56 bits) since the initialization. Equivalent to `current_micro()` and `current_usec()`.

Returns:
The current microsecond as a double value since the initialization of the library, modulo 2^32.

Example: See `current_usec()`

See also:
`current_micro()`, `current_usec()`, `current_msec()`, `current_time()` and `msec_delay()`

26.2.3.2 UINT32 current_micro (MICRO FARTYPE ∗ *m*)

Returns the current microsecond since the initialization. Equivalent to `current_usec()` and `current_double_usec()`.

Parameters:

m Pointer to MICRO structure.

Returns:

The current microsecond since the initialization of the library, modulo 2^32. It can also fill in the `MICRO` structure if the pointer is not `NULL`.

Example: See `current_usec()`

See also:

`current_usec()`, `current_double_usec()`, `current_msec()`, `current_time()` and `msec_delay()`

26.2.3.3 UINT32 current_time (void)

Returns the current millisecond since the initialization.

Remarks:

If the eyelink_exptkit library is not initialized, or initialized multiple times, the return value is invalid and the return value is unpredictable. The call to `current_msec()` is always equivalent to `current_time()`.

Returns:

The current millisecond since the initialization of the library.

Example:

```
// This program illustrates the use of current_msec
#include <eyelink.h>
#include <stdio.h>

eyemsg_printf("Delay test starts: %ld", current_msec());
msec_delay(100);
eyemsg_printf("Delay test ends: %ld", current_time());
```

Output:

```
MSG     4532575 Delay test starts: 5236
MSG     4532671 Delay test ends: 5336
```

See also:

`current_msec()`, `current_micro()`, `current_usec()`, `eyelink_tracker_time()` and `msec_delay()`

26.2.3.4 UINT32 current_usec (void)

Returns the current microsecond since the initialization. Equivalent to `current_micro()` and `current_double_usec()`.

Remarks:

If the eyelink_exptkit library is not initialized, or initialized multiple times, the return value is invalid and unpredictable. The call to `current_usec()` is equivalent to `current_micro(NULL)`. The function is very platform dependent. Platforms such as windows 95, 98, ME may not return usec properly. Ie. It may return `current_time() * 1000`.

Returns:

The current microsecond since the initialization of the library, modulo 2^32.

Example:

```
// This program illustrates the use of current_micro and current_usec

#include <eyelink.h>
#include <stdio.h>

MICRO m1, m2;  // Special high-resolution time structure

// Get the current subject PC time in microseconds
current_micro(&m1);
eyemsg_printf("Delay test starts: %ld", current_usec());

// Delay for 100 msec
msec_delay(100);

// Get the current subject PC time again
current_micro(&m2);
eyemsg_printf("Delay test ends: %ld", current_usec());

// Calculate the actual amount of delay
eyemsg_printf("Total Delay: %6.3f",
    m2.msec + m2.usec/1000.0 - m1.msec + m1.usec/1000.0);
```

Output:

```
MSG     5441107 Delay test starts: 4610094
MSG     5441206 Delay test ends: 4710005
MSG     5441206 Total Delay: 100.003
```

See also:

`current_micro()`, `current_double_usec()`, `current_msec()`, `current_time()` and `msec_delay()`

26.2.3.5 INT16 eyelink_reset_clock (INT16 *enable*)

Initializes the high frequency clock.

With TSR interface under msdos, Start/stop timing resources.

Parameters:

enable Usually `<enable>` is `1`. However, in MS-DOS passing `0` will cleanup the timing resources. In other platforms passing `0` has no effect.

26.2.3.6 void msec_delay (UINT32 *n*)

Does a unblocked delay using `current_time()`.

Parameters:

n n milliseconds to delay.

Example:

```
// This program illustrates the use of msec_delay
#include <eyelink.h>
#include <stdio.h>

eyemsg_printf("Delay test starts: %ld", current_msec());
// delay for 100 msec
msec_delay(100);
eyemsg_printf("Delay test ends: %ld", current_time());
```

Output:

```
MSG     4346690 Delay test starts: 12768
MSG     4346791 Delay test ends: 12868
```

See also:

`current_msec()` and `pump_delay()`

26.3 Access Tracker Time

26.3.1 Detailed Description

These function produce a local estimate of the eye tracker clock. There are several time bases in the system: the eye tracker time, the local eyelink DLL time, and the OS system time. All of these may not run at the same speed, differing by up to 200 ppm (2 milliseconds every 10 seconds).

These functions give a reasonable estimate of the EyeLink tracker time that is used in EDF files and for sample and event timestamps over the link. The time estimate is perturbed by link delays, but will never decrease, but may jump forwards sometimes or "freeze" for a few milliseconds. Two resolution of time are given: milliseconds as returned by current_time(), and microseconds as returned by current_double_usec().

Defines

- #define eyelink_tracker_time() eyelink_tracker_msec()
- #define eyelink_time_offset() eyelink_msec_offset()

Functions

- UINT32 eyelink_request_time (void)
- UINT32 eyelink_node_request_time (ELINKADDR node)
- UINT32 eyelink_read_time (void)
- double eyelink_tracker_double_usec (void)
- UINT32 eyelink_tracker_msec (void)
- double eyelink_double_usec_offset (void)
- UINT32 eyelink_msec_offset (void)

26.3.2 Define Documentation

26.3.2.1 #define eyelink_time_offset() eyelink_msec_offset()

See `eyelink_time_offset()`

26.3.2.2 #define eyelink_tracker_time() eyelink_tracker_msec()

See `eyelink_tracker_msec()`

26.3.3 Function Documentation

26.3.3.1 double eyelink_double_usec_offset (void)

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

Returns:
Returns the time difference between the tracker time and display pc time in microseconds.

See also:
`current_time()` and `eyelink_tracker_msec()`

26.3.3.2 UINT32 eyelink_msec_offset (void)

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

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

See also:
`current_time()`, `eyelink_tracker_msec()` and `eyelink_tracker_double_-usec()`

26.3.3.3 UINT32 eyelink_node_request_time (ELINKADDR *node*)

Sends a request the connected eye tracker to return its current time.

Remarks:
The time reply can be read with `eyelink_read_time()`.

Parameters:
node Address of a specific tracker.

Returns:
`0` if no error, else link error code.

Example: See `eyelink_request_time()`

See also:
`eyelink_request_time()` and `eyelink_read_time()`

26.3.3.4 UINT32 eyelink_read_time (void)

Returns the tracker time requested by `eyelink_request_time()` or `eyelink_node_-request_time()`.

Returns:
`0` if no response yet, else timestamp in millisecond.

Example: See `eyelink_request_time()`

See also:
`eyelink_node_request_time()` and `eyelink_request_time()`

26.3.3.5 UINT32 eyelink_request_time (void)

Sends a request the connected eye tracker to return its current time.

Remarks:
The time reply can be read with `eyelink_read_time()`.

Returns:

0 if no error, else link error code.

Example:

```
// This program illustrates the use of eyelink_request_time to retrieve the current tracker time

#include <eyelink.h>

// This routine is used to retrieve the concurrent tracker PC time.
// If successful, this function returns the tracker PC time, else 0;
// Note, the waiting time is set as 50 msec, although normally the function should return within 1
UINT32 current_tracker_time()
{
    UINT32  time_subject=0, time_tracker=0;

    eyelink_request_time();         // request the tracker PC time
    time_subject = current_msec();
    // wait for a maximum of 50 msec
    while(!time_tracker && current_msec() -time_subject < 50)
        time_tracker = eyelink_read_time();  // read tracker PC time

    return time_tracker;
}
```

See also:

eyelink_node_request_time() and eyelink_read_time()

26.3.3.6 double eyelink_tracker_double_usec (void)

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

Returns:

Each of these functions returns the current tracker time (in microseconds) since tracker initialization.

Example:

```
// This program illustrates the use of eyelink_tracker_double_usec()
#include <eyelink.h>
#include <stdio.h>

int i;
ALLF_DATA evt;

i = eyelink_get_next_data(NULL);   // Checks for new data item
switch(i)
{
    case SAMPLE_TYPE:
    {
        // Gets the new data
        eyelink_get_float_data(&evt);
        // Checks for the difference of the current tracker time and the
        // time returned from the latest sample
        if(eyelink_tracker_double_usec()/1000000 >(evt.time +2) )
        printf("Oops it took longer than 2 milliseconds for the data to get
                here.\n");
    }
    break;
}
```

See also:
eyelink_tracker_msec(), current_time(), eyelink_msec_offset() and eyelink_double_usec_offset()

26.3.3.7 UINT32 eyelink_tracker_msec (void)

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

Returns:
Each of these functions returns the current tracker time (in microseconds) since tracker initialization.

Example:

```
// This program illustrates the use of eyelink_tracker_time()
#include <eyelink.h>
#include <stdio.h>

int i;
ALLF_DATA evt;

i = eyelink_get_next_data(NULL);   // Checks for new data item
switch(i)
{
    case SAMPLE_TYPE:
    {
        // Gets the new data
        eyelink_get_float_data(&evt);
        // Checks for the difference of the current tracker time and the
        // time returned from the latest sample
        if(eyelink_tracker_msec()/1000 >(evt.time +2) )
        printf("Oops it took longer than 2 milliseconds for the data to get
                here.\n");
    }
    break;
}
```

See also:
current_time(), eyelink_msec_offset(), eyelink_double_usec_offset() and eyelink_tracker_double_usec()

26.4 Setup EyeLink tracker

26.4.1 Detailed Description

These methods are used to setup the EyeLink tracker such as Camera Setup, Calibration, Drift Correction, Validation, etc.

Functions

- INT16 eyelink_abort (void)
- INT16 eyelink_start_setup (void)
- INT16 eyelink_in_setup (void)
- INT16 eyelink_target_check (INT16 FARTYPE ∗x, INT16 FARTYPE ∗y)
- INT16 eyelink_target_checkf (float FARTYPE ∗x, float FARTYPE ∗y)
- INT16 eyelink_accept_trigger (void)
- INT16 eyelink_driftcorr_start (INT16 x, INT16 y)
- INT16 eyelink_driftcorr_startf (float x, float y)
- INT16 eyelink_cal_result (void)
- INT16 eyelink_apply_driftcorr (void)
- INT16 eyelink_cal_message (char FARTYPE ∗msg)
- void exit_calibration (void)
- INT16 do_tracker_setup (void)
- INT16 do_drift_correct (INT16 x, INT16 y, INT16 draw, INT16 allow_setup)
- INT16 do_drift_correctf (float x, float y, INT16 draw, INT16 allow_setup)

26.4.2 Function Documentation

26.4.2.1 INT16 do_drift_correct (INT16 *x*, INT16 *y*, INT16 *draw*, INT16 *allow_setup*)

Performs a drift correction before a trial.

Remarks:

When the 'ESC' key is pressed during drift correction, `<allow_setup>` determines the result. If `1`, the EyeLink Setup menu is accessed. This will always clear the display, so redrawing of hidden stimuli may be required. Otherwise, the drift correction is aborted. Calling `exit_calibration()` from an event handler will cause any call to `do_drift_correct()` in progress to return immediately. In all cases, the return code will be `27` (`ESC_KEY`).

Parameters:

x X Position of drift correction target.

y Y Position 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.

Returns:

`0` if successful, `27` if 'ESC' key was pressed to enter Setup menu or abort.

Example:

```
// This program illustrates the use of do_drift_correction with drift correction target drawn by t

#include <eyelink.h>

int target_shown = 0;
int draw_own_target = 1;

while(1)
{
    // Checks link often so we can exit if tracker stopped
    if(!eyelink_is_connected())
        return ABORT_EXPT;

    // If drift correct target is not drawn, we have to draw it here
    if (draw_own_target && !target_shown)
    {
        // Code for drawing own drift correction target
        target_shown = 1;
    }

    // Performs drift correction with target drawn in the center
    error = do_drift_correct(SCRWIDTH/2, SCRHEIGHT/2,
        draw_own_target, 1);

    // repeat if ESC was pressed to access Setup menu
    // Redawing the target may be necessary
    if(error!=27)
        break;
    else
        target_shown = 0;
}
```

See also:
`do_tracker_setup()` and `set_dcorr_sounds()`

26.4.2.2 INT16 do_drift_correctf (float *x*, float *y*, INT16 *draw*, INT16 *allow_setup*)

Performs a drift correction before a trial. Same as do_drift_correct() except, this takes the x,y values as float.

Remarks:
When the 'ESC' key is pressed during drift correction, `<allow_setup>` determines the result. If `1`, the EyeLink Setup menu is accessed. This will always clear the display, so redrawing of hidden stimuli may be required. Otherwise, the drift correction is aborted. Calling `exit_calibration()` from an event handler will cause any call to `do_drift_correct()` in progress to return immediately. In all cases, the return code will be `27` (`ESC_KEY`).

Parameters:

x X Position of drift correction target.

y Y Position 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.

Returns:
`0` if successful, `27` if 'ESC' key was pressed to enter Setup menu or abort.

Example:

```
// This program illustrates the use of do_drift_correction with drift correction target drawn by t

#include <eyelink.h>

int target_shown = 0;
int draw_own_target = 1;

while(1)
{
    // Checks link often so we can exit if tracker stopped
    if(!eyelink_is_connected())
        return ABORT_EXPT;

    // If drift correct target is not drawn, we have to draw it here
    if (draw_own_target && !target_shown)
    {
        // Code for drawing own drift correction target
        target_shown = 1;
    }

    // Performs drift correction with target drawn in the center
    error = do_drift_correctf(SCRWIDTH/2, SCRHEIGHT/2,
        draw_own_target, 1);

    // repeat if ESC was pressed to access Setup menu
    // Redawing the target may be necessary
    if(error!=27)
        break;
    else
        target_shown = 0;
}
```

See also:
`do_tracker_setup()` and `set_dcorr_sounds()`

26.4.2.3 INT16 do_tracker_setup (void)

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 `exit_calibration()` from an event handler will cause any call to `do_tracker_setup()` in progress to return immediately.

Returns:
Always `0`.

Example:

```
// This program shows an example of using the do_tracker_setup()
#include <eyelink.h>

// Colors used for drawing calibration target and background
COLORREF target_foreground_color = RGB(0,0,0);
COLORREF target_background_color = RGB(255,255,255);
int i = SCRWIDTH/60;     // Selects best size for calibration target
int j = SCRWIDTH/300;    // Selects size for focal spot in target
if(j < 2) j = 2;

// Sets diameter of target and of hole in middle of target
```

```
set_target_size(i, j);

// Sets target color and display background color
set_calibration_colors(target_foreground_color, target_background_color);

// Sets sounds for Setup menu (calibration, validation)
set_cal_sounds("", "", "");
// Sets sounds for drift correction
set_dcorr_sounds("", "off", "off");

// Performs camera setup, calibration, validation, etc.
do_tracker_setup();

...
// Code for running the trials
```

See also:

do_drift_correct(), set_cal_sounds(), set_calibration_colors() and set_target_size()

26.4.2.4 void exit_calibration (void)

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

Example:

```
// The following code illustrates the use of exit_calibration().  This would usually be
// called from a message or event handler (see the w32_demo_window.c module) for an example.
#include <eyelink.h>
switch (message)
{
case WM_KEYDOWN:
case WM_CHAR:
    {
        // Process messages, translates key messages and queue
        UINT16 key = process_key_messages(hwnd, message, wparam, lparam);
        eyemsg_printf("key %d", key);

        // Checks the current tracker state.  If it is in setup mode, pressing the PAGE_DOWN
        // key would abort the setup process (i.e., drift-calibration, validation, coreection)
        if (key == 0x5100 && (eyelink_current_mode() & IN_SETUP_MODE))
            exit_calibration();
        break;
    }
    ...
    // Other windows messages and events
}
```

See also:

do_tracker_setup(), do_drift_correct() and eyelink_current_mode()

26.4.2.5 INT16 eyelink_abort (void)

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

Remarks:

Use before attempting to draw graphics on the tracker display, transferring files, or closing link. Always call eyelink_wait_for_mode_ready() afterwards to ensure tracker has finished the mode transition. This function pair is implemented by the eyelink_exptkit library function set_-offline_mode().

Returns:
0 if mode switch begun, else link error.

Example:

```
// The following is the functional implementation of the stop_recording(), using the eyelink_abort

#include <eyelink.h>

eyecmd_printf("heuristic_filter = ON");
eyelink_abort();    // stop data flow
eyelink_wait_for_mode_ready(500); // wait till finished mode switch
```

See also:
eyelink_wait_for_mode_ready() and set_offline_mode()

26.4.2.6 INT16 eyelink_accept_trigger (void)

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

Returns:
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.

Example: See eyelink_driftcorr_start()

See also:
eyelink_apply_driftcorr(), eyelink_current_mode(), eyelink_driftcorr_-start() and eyelink_target_check()

26.4.2.7 INT16 eyelink_apply_driftcorr (void)

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

Returns:
0 if command sent OK, else link error.

Example: See eyelink_driftcorr_start()

See also:
eyelink_accept_trigger(), eyelink_cal_result(), eyelink_current_-mode(), eyelink_driftcorr_start() and eyelink_target_check()

26.4.2.8 INT16 eyelink_cal_message (char FARTYPE ∗ *msg*)

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

Parameters:

msg Buffer to return back the message.

Returns:

0 if no message since last command sent, else length of string.

Example:

```
// This programs illustrates the use of eyelink_cal_message()
#include <eyelink.h>
char message[256];

// Performs a drift correction
while(1)
{   // Check link often so we can exit if tracker stopped
    if(!eyelink_is_connected()) return ABORT_EXPT;

    // Performs drift correction with target pre-drawn
    error = do_drift_correct(SCRWIDTH/2, SCRHEIGHT/2, 1, 1);

    // repeat if ESC was pressed to access Setup menu
    if(error!=27) break;
}
// Retrieves and writes out the calibration result message
eyelink_cal_message(message);
eyemsg_printf(message);
```

Output:

```
MSG     1896559 DRIFTCORRECT R RIGHT at 320,40  OFFSET 0.11 deg.  -1.0,-4.0 pix.
MSG     1896560 drift_correction: 0.11 -1.00 -4.00
```

See also:

eyelink_accept_trigger(), eyelink_apply_driftcorr() and eyelink_cal_-result()

26.4.2.9 INT16 eyelink_cal_result (void)

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

Returns:

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.

Example: See eyelink_driftcorr_start()

See also:

eyelink_accept_trigger(), eyelink_apply_driftcorr(), eyelink_cal_-message() and eyelink_driftcorr_start()

26.4.2.10 INT16 eyelink_driftcorr_start (INT16 *x*, INT16 *y*)

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

Parameters:

- ***x*** X position of the target.
- ***y*** Y position of the target.

Returns:

`0` if command sent OK, else link error.

Example:

```
// This program illustrates the use of eyelink_driftcorr_start() for the
// implementation of a drift correction mechanism

#include <eyelink.h>

unsigned key;
int result = 0;
int x, y; // position of the drift correction target

if(eyelink_is_connected())
{
  eyecmd_printf("heuristic_filter = ON");
  ...
  // Code to draw the target here
  while(getkey()) {};  // dump the keys

  eyelink_driftcorr_start(x, y); // start the drift correction
  do {
       result = eyelink_cal_result();

       key = getkey();
       switch(key)
         {
           case TERMINATE_KEY:  // breakout code
             return TERMINATE_KEY;
           case 0:              // no key
           case JUNK_KEY:       // no key
             break;
           case ESC_KEY:       // ESC key: we flag abort from our end
             result = 27;
             break;
           case 32:             // Spacebar: we trigger ourselves
             eyelink_accept_trigger();
             break;
           default:
             eyelink_send_keybutton(key,0,KB_PRESS);
             break;
         }
    } while(result == NO_REPLY);

     // Applies the drift correction result
     if (result != 27 && result != -1)
        eyelink_apply_driftcorr();
    else
       ; // Other code for handling

    return result;
}
```

See also:

`eyelink_accept_trigger()` and `eyelink_send_keybutton()`

26.4.2.11 INT16 eyelink_driftcorr_startf (float *x*, float *y*)

Sets the position of the drift correction target, and switches the tracker to drift-correction mode. Should be followed by a call to `eyelink_wait_for_mode_ready()`. Same as eyelink_driftcorr_start() except the x,y parameters take floating point values.

Parameters:

x X position of the target.

y Y position of the target.

Returns:

`0` if command sent OK, else link error.

Example:

```
// This program illustrates the use of eyelink_driftcorr_start() for the
// implementation of a drift correction mechanism

#include <eyelink.h>

unsigned key;
int result = 0;
int x, y; // position of the drift correction target

if(eyelink_is_connected())
{
  eyecmd_printf("heuristic_filter = ON");
  ...
  // Code to draw the target here
  while(getkey()) {};  // dump the keys

  eyelink_driftcorr_start(x, y); // start the drift correction
  do {
       result = eyelink_cal_result();

       key = getkey();
       switch(key)
         {
           case TERMINATE_KEY:  // breakout code
             return TERMINATE_KEY;
           case 0:              // no key
           case JUNK_KEY:       // no key
             break;
           case ESC_KEY:       // ESC key: we flag abort from our end
             result = 27;
             break;
           case 32:             // Spacebar: we trigger ourselves
             eyelink_accept_trigger();
             break;
           default:
             eyelink_send_keybutton(key,0,KB_PRESS);
             break;
         }
    } while(result == NO_REPLY);

     // Applies the drift correction result
     if (result != 27 && result != -1)
```

```
            eyelink_apply_driftcorr();
        else
           ; // Other code for handling

        return result;
    }
```

See also:
`eyelink_accept_trigger()` and `eyelink_send_keybutton()`

26.4.2.12 INT16 eyelink_in_setup (void)

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

Returns:
`0` if no longer in setup mode.

Example:

```
// This program illustrates the use of eyelink_in_setup

#include <eyelink.h>

int current_mode;
int prev_mode =0;
UINT start_time  = current_msec();

// Checks for 10 seconds
while(current_msec() < start_time + 10000)
{
    if(!eyelink_is_connected())
        return -1;

    current_mode =eyelink_in_setup();
    if (current_mode!=prev_mode)
        eyemsg_printf("%s", current_mode?"In setup":"Not in setup");

    prev_mode = current_mode;
}
```

Output:

```
MSG     905992 In setup
MSG     909596 Not in setup
```

See also:
`eyelink_current_mode()`

26.4.2.13 INT16 eyelink_start_setup (void)

Enters setup mode

26.4.2.14 INT16 eyelink_target_check (INT16 FARTYPE ∗ *x*, INT16 FARTYPE ∗ *y*)

Returns the current target position and state.

Parameters:

x Pointer to variable to hold target X position.

y Pointer to variable to hold target Y position.

Returns:

`1` if target is visible, 0 if not.

Example:

```
INT16 target_mode_display(void)
{
  int target_visible = 0;       // target currently drawn
  INT16 tx;             // new target position
  INT16 ty;

  INT16 otx=MISSING;    // current target position
  INT16 oty=MISSING;

  unsigned key;         // local key pressed
  int i;
  int result = NO_REPLY;

  // LOOP WHILE WE ARE DISPLAYING TARGETS
  while(eyelink_current_mode() & IN_TARGET_MODE)
    {
      if(!eyelink_is_connected()) return -1;
      key = getkey();
          // HANDLE LOCAL KEY PRESS
      if(key)
                {
                  switch(key)
                        {
                          case TERMINATE_KEY:       // breakout key code
                        clear_cal_display();
                        return TERMINATE_KEY;
                          case 32:                      // Spacebar: accept fixation
                        eyelink_accept_trigger();
                        break;
                          case 0:               // No key
                          case JUNK_KEY:        // No key
                        break;
                          case ESC_KEY: if(eyelink_is_connected()==-1) goto exit;
                          default:          // Echo to tracker for remote control
                        if(allow_local_control)
                                eyelink_send_keybutton(key,0,KB_PRESS);
                        break;
                        }
                }

      result = eyelink_cal_result();
      if(result != NO_REPLY) break;

          // HANDLE TARGET CHANGES
      i = eyelink_target_check(&tx, &ty);
                              // erased or moved: erase target
      if( (target_visible && i==0) || tx!=otx || ty!=oty)
        {
          erase_cal_target();
          target_visible = 0;
```

```
            }
                                    // redraw if visible
          if(!target_visible && i)
            {
              draw_cal_target(tx, ty);

              target_visible = 1;
              otx = tx;             // record position for future tests
              oty = ty;
            }
        }

    exit:

      if(target_visible)
              erase_cal_target();   // erase target on exit

      clear_cal_display();
      return result;
    }
```

See also:
eyelink_accept_trigger(), eyelink_apply_driftcorr(), eyelink_current_-mode() and eyelink_driftcorr_start()

26.4.2.15 INT16 eyelink_target_checkf (float FARTYPE ∗ *x*, float FARTYPE ∗ *y*)

Returns the current target position and state. Same as eyelink_target_check() except this function returns data in floating point values.

Parameters:
x Pointer to variable to hold target X position.
y Pointer to variable to hold target Y position.

Returns:
1 if target is visible, 0 if not.

26.5 Keyboard Input Functions

Defines

- #define CURS_UP 0x4800
- #define CURS_DOWN 0x5000
- #define CURS_LEFT 0x4B00
- #define CURS_RIGHT 0x4D00
- #define ESC_KEY 0x001B
- #define ENTER_KEY 0x000D
- #define PAGE_UP 0x4900
- #define PAGE_DOWN 0x5100
- #define JUNK_KEY 1
- #define TERMINATE_KEY 0x7FFF

Functions

- void flush_getkey_queue (void)
- UINT16 read_getkey_queue (void)
- UINT16 echo_key (void)
- UINT16 getkey (void)
- UINT32 getkey_with_mod (UINT16 ∗unicode)
- INT16 escape_pressed (void)
- INT16 break_pressed (void)
- void terminal_break (INT16 assert)

26.5.1 Define Documentation

26.5.1.1 #define CURS_DOWN 0x5000

Cursor down key.

26.5.1.2 #define CURS_LEFT 0x4B00

Cursor left key.

26.5.1.3 #define CURS_RIGHT 0x4D00

Cursor right key.

26.5.1.4 #define CURS_UP 0x4800

Cursor up key.

26.5.1.5 #define ENTER_KEY 0x000D

Return key.

26.5.1.6 #define ESC_KEY 0x001B

Escape key.

26.5.1.7 #define JUNK_KEY 1

Junk key to indicate untranslatable key.

26.5.1.8 #define PAGE_DOWN 0x5100

Page down key.

26.5.1.9 #define PAGE_UP 0x4900

Page up key.

26.5.1.10 #define TERMINATE_KEY 0x7FFF

Returned by getkey if program should exit.

26.5.2 Function Documentation

26.5.2.1 INT16 break_pressed (void)

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 `eyelink_is_connected()` returns `0`.

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.

Returns:

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

Example:

```
// This program illustrates the use of break_pressed() and escape_pressed()

#include <eyelink.h>

// reset keys and buttons from tracker
eyelink_flush_keybuttons(0);

// Trial loop: till timeout or response
while(1)
{
    // Checks if recording aborted
    if((error=check_recording())!=0) return error;

    // check for program termination or ALT-F4 or CTRL-C keys
```

```
            if(break_pressed())
                   return ABORT_EXPT;

            // check for local ESC key to abort trial (useful in debugging)
            if(escape_pressed())
                   return SKIP_TRIAL;

            ...
            // Other code for display update, trial terminating, etc.
        }
```

See also:

echo_key(), escape_pressed() and getkey()

26.5.2.2 UINT16 echo_key (void)

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

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.

Returns:

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

Example:

```
// This program illustrates the use of echo_key(), if you want to write your own code for do_track
#include <eyelink.h>

// Resets keys and buttons from tracker
eyelink_flush_keybutton();
// dump any accumulated key presses
while(getkey());

// If we make sure that we are in set-up mode
while(eyelink_current_mode() & IN_SETUP_MODE)
{
    int i = eyelink_current_mode();

    // calibrate, validate, etc: show targets
    if(i & IN_TARGET_MODE)
    {
        ...
        // Code here for displaying and updating calibration target
        // If using Windows library, call target_mode_display();
    }
    else if(i & IN_IMAGE_MODE)
    {
        ...
        // Code here for showing the camera image
        // If using Windows library, call target_mode_display();
    }

    // If we allow local tracker control, echo to tracker for remote control
    echo_key();
}
```

See also:
`eyelink_read_keybutton()`, `eyelink_send_keybutton()` and `getkey()`

26.5.2.3 INT16 escape_pressed (void)

This function tests if the 'ESC' key is held down, and is usually used to break out of nested loops. This does not allow processing of Windows messages, unlike `getkey()`.

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.

Returns:
`1` if 'ESC' key held down; `0` if not.

Example: See `break_pressed()`

See also:
`break_pressed()`, `getkey()` and `echo_key()`

26.5.2.4 void flush_getkey_queue (void)

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

Example:

```
// This program illustrates the use of flush_getkey_queue()

#include <eyelink.h>

// Sets the EyeLink host address, if tracker IP address is different
// from the default "100.1.1.1"
if (set_eyelink_address("100.1.1.7"))
    return -1;

// Initializes EyeLink library, and opens connection to the tracker
if(open_eyelink_connection(0))
    return -1;

flush_getkey_queue();   // initialize getkey() system

...
// Code for the setup, recording, and cleanups
close_eyelink_connection();      // disconnect from tracker
```

See also:
`read_getkey_queue()`

26.5.2.5 UINT16 getkey (void)

A routine for checking for Windows keystroke events, and dispatching Windows messages. If no key is pressed or waiting, it returns `0`. For a standard ASCII key, a value from `31` to `127` is returned. For extended keys, a special key value is returned. If the program has been terminated by ALT-F4 or a call to `terminal_break()`, or the "Ctrl" and "C" keys are held down, the value `TERMINATE_KEY` is returned. The value `JUNK_KEY (1)` is returned if a non-translatable key is pressed.

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 `escape_pressed()` or `break_pressed()` 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 defined in core_expt.h, as:

- `CURS_UP 0x4800`
- `CURS_DOWN 0x5000`
- `CURS_LEFT 0x4B00`
- `CURS_RIGHT 0x4D00`
- `ESC_KEY 0x001B`
- `ENTER_KEY 0x000D`
- `TERMINATE_KEY 0x7FFF`
- `JUNK_KEY 0x0001`

Returns:

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

Example:

```
// This program illustrates the use of getkey()

#include <eyelink.h>
UINT32  delay_time = 5000L;  // Set the maximum wait time

// flushes any waiting keys or buttons
eyelink_flush_keybuttons(0);

delay_time += current_msec();
while(1)
{
    // Waitkey times out
    if(current_time() > delay_time)
    {
        eyemsg_printf("WAITKEY TIMEOUT");
        break;
    }
    key = getkey();
    if(key) // If key press occurs
    {
        // Is this printable key?
        if(key < 256 && isprint(key))
            eyemsg_printf("WAITKEY '%c'", key);
        else
            eyemsg_printf("WAITKEY 0x%04X", key);
        break;
    }
}
```

See also:

break_pressed(), echo_key(), escape_pressed(), eyelink_flush_-keybuttons() and eyelink_send_keybutton()

26.5.2.6 UINT32 getkey_with_mod (UINT16 ∗ *unicode*)

Same as getkey except it returns the modifier and the key pressed. It returns a 32 bit unsigned integer. The first 16 bits are reserved for the modifier and the last 16 bits are reserved for the key values. If there are no modifiers present, the return value of this is the same as getkey(). If non null pointer passed in for unicode, the translated key value will be set if a key is preent.

26.5.2.7 UINT16 read_getkey_queue (void)

Reads 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.

Remarks:

These functions are intended to support languages other than C or C++.

Returns:

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.
else, code of key if any key pressed.

See also:

flush_getkey_queue()

26.5.2.8 void terminal_break (INT16 *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 break_pressed() to return 1, and getkey() to return TERMINATE_KEY. These functions can be re-enabled by calling terminal_break() with an argument of 0.

Parameters:

assert 1 to signal a program break, 0 to reset break.

Example:

```
// The following code illustrates the use of terminal_break().  This would usually be called
// from a message or event handler (see the w32_demo_window.c module) for a complete example.

#include <eyelink.h>
```

```
switch (message)
{
case WM_CLOSE:        // If ALT-F4 pressed, force program to close
    PostQuitMessage(0);
    terminal_break(1);// break out of loops
    break;

case WM_DESTROY:      // Window being closed by ALT-F4
    PostQuitMessage( 0 );
    ...
    // Code here for graphics cleaning up
    terminal_break(1);// break out of loops
    break;

case WM_QUIT:         // Needs to break out of any loops
    terminal_break(1);
    break;
    ...
    // Other windows messages and events
}
```

See also:

`break_pressed()` and `getkey()`

26.6 Data file utilities

Defines

- #define BAD_FILENAME -2222
- #define BAD_ARGUMENT -2223
- #define receive_data_file receive_data_file_dialog

Functions

- INT16 eyelink_request_file_read (char FARTYPE ∗src)
- INT16 eyelink_get_file_block (void FARTYPE ∗buf, INT32 FARTYPE ∗offset)
- INT16 eyelink_request_file_block (UINT32 offset)
- INT16 eyelink_end_file_transfer (void)
- INT32 receive_data_file (char ∗src, char ∗dest, INT16 dest_is_path)
- INT32 receive_data_file_feedback (char ∗src, char ∗dest, INT16 dest_is_path, void(∗progress)(unsigned int size, unsigned int received))
- INT16 open_data_file (char ∗name)
- INT16 close_data_file (void)
- void splice_fname (char ∗fname, char ∗path, char ∗ffname)
- int check_filename_characters (char ∗name)
- int file_exists (char ∗path)
- int create_path (char ∗path, INT16 create, INT16 is_dir)
- INT32 receive_data_file_dialog (char ∗src, char ∗dest, INT16 dest_is_path)

26.6.1 Define Documentation

26.6.1.1 #define BAD_ARGUMENT -2223

Invalid argument

26.6.1.2 #define BAD_FILENAME -2222

Bad file name

26.6.1.3 #define receive_data_file receive_data_file_dialog

This macro is introduced so that the old eyelink projects compiled against new libraries, the file transfer behaves the same as the old receive_data_file in win32 platform See `receive_data_file_dialog()`

26.6.2 Function Documentation

26.6.2.1 int check_filename_characters (char ∗ *name*)

Checks file name for legality. Attempts to ensure cross-platform for viewer. No spaces allowed as this interferes with messages. Assume viewer will translate forward/backward slash. Windows: don't allow <>:"/\| Also, device names, such as aux, con, lpt1, and prn are not allowed in windows. forward slashes is missed.

26.6.2.2 INT16 close_data_file (void)

Closes any open EDF file.

Returns:

Returns `0` if success, else error code.

26.6.2.3 int create_path (char * *path*, INT16 *create*, INT16 *is_dir*)

Checks if path exists. Will create directory if 'create'. Creates directory from last name in 'path', unless ends with '\' or 'is_dir' nonzero. Otherwise, last item is assumed to be filename and is dropped.

Returns:

`0` if exists, `1` if created, `-1` if failed.

26.6.2.4 INT16 eyelink_end_file_transfer (void)

Aborts send of file.

Returns:

`0` if OK, else send error.

Remarks:

Internal function. use receive_data_file()

26.6.2.5 INT16 eyelink_get_file_block (void FARTYPE * *buf*, INT32 FARTYPE * *offset*)

Get next block of file. If `<offset>` is not `NULL`, will be filled with block-start offset in file.

Returns:

negative if error, `NO_REPLY` if waiting for packet, else block size (0..512). size is $<$ 512 (can be 0) if at EOF.

Remarks:

Internal function. use receive_data_file()

26.6.2.6 INT16 eyelink_request_file_block (UINT32 *offset*)

Ask for next block of file, reads from `<offset>`.

Remarks:

Internal function. use receive_data_file()

26.6.2.7 INT16 eyelink_request_file_read (char FARTYPE ∗ *src*)

Request send of file "src". If "", gets last data file.

Returns:
`0` if OK, else send error.

Remarks:
Internal function. use receive_data_file()

26.6.2.8 int file_exists (char ∗ *path*)

Checks if file and/or path exists.

Returns:
`0` if does not exist, `1` if exists, `-1` if cannot overwrite.

26.6.2.9 INT16 open_data_file (char ∗ *name*)

Opens an EDF file, closes any existing file.

Parameters:
name Name of data file.

Returns:
Returns `0` if success, else error code.

26.6.2.10 INT32 receive_data_file (char ∗ *src*, char ∗ *dest*, INT16 *dest_is_path*)

This receives a data file from the EyeLink tracker PC.

Parameters:

← ***src*** Name of eye tracker file (including extension). If "" (empty string), asks tracker for name of last opened data file.

← ***dest*** Name of local file to write to (including extension). This must be a valid file name or directory name.

← ***dest_is_path*** If nonzero, appends file name to <dest> as a directory path.

Returns:
`0` if file transfer was cancelled.
Size of file if successful.
`FILE_CANT_OPEN` if no such file.
`FILE_XFER_ABORTED` if data error.

Example: See `open_data_file()`

See also:
`close_data_file()` and `open_data_file()`

Remarks:
If linked against eyelink_exptkit20.lib or w32_dialogs.h is included, the behaviour of this function is receive_data_file_dialog().

26.6.2.11 INT32 receive_data_file_dialog (char ∗ *src*, char ∗ *dest*, INT16 *dest_is_path*)

This receives a data file from the EyeLink tracker PC with graphical progressbar. This function only available in Win32 platform.

Parameters:

← ***src*** Name of eye tracker file (including extension). If "" (empty string), asks tracker for name of last opened data file.

← ***dest*** Name of local file to write to (including extension). If "" (empty string), prompts for file name.

← ***dest_is_path*** If nonzero, appends file name to <dest> as a directory path.

Returns:

`0` if file transfer was cancelled.
Size of file if successful.
`FILE_CANT_OPEN` if no such file.
`FILE_XFER_ABORTED` if data error.

See also:

`receive_data_file()`

26.6.2.12 INT32 receive_data_file_feedback (char ∗ *src*, char ∗ *dest*, INT16 *dest_is_path*, void(∗)(unsigned int size, unsigned int received) *progress*)

This receives a data file from the EyeLink tracker PC. Exact same as `receive_data_file()`. except the feedback parameters can be used for showing what is the full size of the edf file and how much is received so far. This function is currently used internally only.

Parameters:

← ***src*** Name of eye tracker file (including extension). If "" (empty string), asks tracker for name of last opened data file.

← ***dest*** Name of local file to write to (including extension). This must be a valid file name or directory name.

← ***dest_is_path*** If nonzero, appends file name to <dest> as a directory path.

← ***progress*** A function pointer, that takes size and received size integers. This allows, one to display progress bar on edf file transfer.

Returns:

`0` if file transfer was cancelled.
Size of file if successful.
`FILE_CANT_OPEN` if no such file.
`FILE_XFER_ABORTED` if data error.

See also:

`receive_data_file`, `close_data_file()` and `open_data_file()`

26.6.2.13 void splice_fname (char ∗ *fname*, char ∗ *path*, char ∗ *ffname*)

Splice 'path' to 'fname', store in 'ffname'. Tries to create valid concatenation. If 'fname' starts with '\', just adds drive from 'path'. If 'fname' contains drive specifier, it is not changed.

26.7 Application/Thread priority control

Functions

- INT32 set_application_priority (INT32 priority)
- void begin_realtime_mode (UINT32 delay)
- void end_realtime_mode (void)
- void set_high_priority (void)
- void set_normal_priority (void)
- INT32 in_realtime_mode (void)

26.7.1 Function Documentation

26.7.1.1 void begin_realtime_mode (UINT32 *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. Use the <`delay`> value to set the minimum time this function takes, so that this function can act as a useful delay.

Remarks:

Warning: Under Windows XP, this call will lock out all keyboard input. The Task Manager will take about 30 seconds to respond to CTRL-ALT-DEL, so press this once and be patient. The keyboard functions well in realtime mode under Windows 2000. This function has little or no effect under Windows 95/98/Me. Keyboard, mouse, and sound may be disabled in some OS.

Parameters:

delay Minimum delay in milliseconds (should be about 100).

Example:

```
// This program shows the use of begin_realtime_mode() and end_realtime_mode()
#include <eyelink.h>

int error;

// Start data recording to EDF file, BEFORE DISPLAYING STIMULUS
error = start_recording(1,1,1,1);
if(error != 0)  return error;     // return error code if failed

// Records for 100 msec before displaying stimulus
// Sets up for realtime execution (minimum delays)
begin_realtime_mode(100);

...
// Code for drawing, updating display and process trial loop
// including timing, key or button response handling

// Exits realtime execution mode
end_realtime_mode();

...
// Code for trial clean up and exiting
```

See also:

`end_realtime_mode()`

26.7.1.2 void end_realtime_mode (void)

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.

Remarks:
Warning: This function has little or no effect under Windows 95/98/Me.

Example: See `begin_realtime_mode()`

See also:
`begin_realtime_mode()`

26.7.1.3 INT32 in_realtime_mode (void)

returns whether the current mode is real-time.

Returns:
`1` if in realtime mode, else `0`.

26.7.1.4 INT32 set_application_priority (INT32 *priority*)

Changes the multitasking proirity of current application Using THREAD_PRIORITY_ABOVE_-NORMAL may reduce latency Reset priority with THREAD_PRIORITY_NORMAL Too high priority will stop the link from functioning!

26.7.1.5 void set_high_priority (void)

Raise application priority. May interfere with other applications.

26.7.1.6 void set_normal_priority (void)

Sets application priority to system normal

26.8 Graphics display options

Data Structures

- struct EYECOLOR

  Represents an RGB color.

- struct EYEPALETTE

  Represents a palette index.

- struct EYEPIXELFORMAT

  Represents pixel format of an image or surface.

- struct EYEBITMAP

  Represents a bitmap image.

- struct KeyInput

  Keyboard input event structure.

- struct MouseMotionEvent

  Mouse motion event structure (For future).

- struct MouseButtonEvent

  Mouse button event structure (For future).

- union InputEvent

  Union of all input types.

- struct HOOKFCNS

  Structure used set and get callback functions.

- struct HOOKFCNS2

  Structure used set and get callback functions.

- struct _CrossHairInfo

  Structure to hold cross hair information.

Defines

- #define KEYINPUT_EVENT 0x1
- #define MOUSE_INPUT_EVENT 0x4
- #define MOUSE_MOTION_INPUT_EVENT 0x5
- #define MOUSE_BUTTON_INPUT_EVENT 0x6
- #define ELKMOD_NONE 0x0000
- #define ELKMOD_LSHIFT 0x0001
- #define ELKMOD_RSHIFT 0x0002
- #define ELKMOD_LCTRL 0x0040
- #define ELKMOD_RCTRL 0x0080

- #define ELKMOD_LALT 0x0100
- #define ELKMOD_RALT 0x0200
- #define ELKMOD_LMETA 0x0400
- #define ELKMOD_RMETA 0x0800
- #define ELKMOD_NUM 0x1000
- #define ELKMOD_CAPS 0x2000
- #define ELKMOD_MODE 0x4000
- #define **CR_HAIR_COLOR** 1
- #define **PUPIL_HAIR_COLOR** 2
- #define **PUPIL_BOX_COLOR** 3
- #define **SEARCH_LIMIT_BOX_COLOR** 4
- #define **MOUSE_CURSOR_COLOR** 5

Typedefs

- typedef enum _EL_CAL_BEEP EL_CAL_BEEP

  Enum used for calibration beeps.

Enumerations

- enum **IMAGETYPE** {

  JPEG, **PNG**, **GIF**, **BMP**,

  XPM }
- enum _EL_CAL_BEEP {

  EL_DC_DONE_ERR_BEEP = -2, EL_CAL_DONE_ERR_BEEP = -1, EL_CAL_DONE_GOOD_-BEEP = 0, EL_CAL_TARG_BEEP = 1,

  EL_DC_DONE_GOOD_BEEP = 2, EL_DC_TARG_BEEP = 3 }

  Enum used for calibration beeps.

Functions

- INT16 target_mode_display (void)
- INT16 image_mode_display (void)
- void setup_graphic_hook_functions (HOOKFCNS ∗hooks)
- HOOKFCNS ∗ get_all_hook_functions ()
- INT16 setup_graphic_hook_functions_V2 (HOOKFCNS2 ∗hooks)
- HOOKFCNS2 ∗ get_all_hook_functions_V2 ()
- int get_image_xhair_data (INT16 x[4], INT16 y[4], INT16 ∗xhairs_on)
- INT32 eyelink_draw_cross_hair (CrossHairInfo ∗chi)

26.8.1 Define Documentation

26.8.1.1 #define ELKMOD_CAPS 0x2000

Modifier for KeyInput.modifier; Cap lock key

26.8.1.2 #define ELKMOD_LALT 0x0100

Modifier for KeyInput.modifier; Left Alt

26.8.1.3 #define ELKMOD_LCTRL 0x0040

Modifier for KeyInput.modifier; left conrol

26.8.1.4 #define ELKMOD_LMETA 0x0400

Modifier for KeyInput.modifier; Left Meta

26.8.1.5 #define ELKMOD_LSHIFT 0x0001

Modifier for KeyInput.modifier; Left shift

26.8.1.6 #define ELKMOD_MODE 0x4000

Modifier for KeyInput.modifier; Mode key

26.8.1.7 #define ELKMOD_NONE 0x0000

Modifier for KeyInput.modifier; No modifier present

26.8.1.8 #define ELKMOD_NUM 0x1000

Modifier for KeyInput.modifier; Number lock key

26.8.1.9 #define ELKMOD_RALT 0x0200

Modifier for KeyInput.modifier; Right alt

26.8.1.10 #define ELKMOD_RCTRL 0x0080

Modifier for KeyInput.modifier; Right control

26.8.1.11 #define ELKMOD_RMETA 0x0800

Modifier for KeyInput.modifier; Right Meta

26.8.1.12 #define ELKMOD_RSHIFT 0x0002

Modifier for KeyInput.modifier; Right shift

26.8.1.13 #define KEYINPUT_EVENT 0x1

set to InputEvent.type to notify keyboard input event

26.8.1.14 #define MOUSE_BUTTON_INPUT_EVENT 0x6

set InputEvent.type to notify mouse button input event.

Remarks:

For future use.

26.8.1.15 #define MOUSE_INPUT_EVENT 0x4

set InputEvent.type to notify mouse input event

Remarks:

For future use.

26.8.1.16 #define MOUSE_MOTION_INPUT_EVENT 0x5

set InputEvent.type to notify mouse motion input event

Remarks:

For future use.

26.8.2 Enumeration Type Documentation

26.8.2.1 enum _EL_CAL_BEEP

Enum used for calibration beeps.

Enumerator:

EL_DC_DONE_ERR_BEEP	Drift Correct Done error beep
EL_CAL_DONE_ERR_BEEP	Calibration Done error beep
EL_CAL_DONE_GOOD_BEEP	Calibration Done correct beep
EL_CAL_TARG_BEEP	Calibration target present beep
EL_DC_DONE_GOOD_BEEP	Drift Correct Done correct beep
EL_DC_TARG_BEEP	Drift Correct target present beep

26.8.3 Function Documentation

26.8.3.1 INT32 eyelink_draw_cross_hair (CrossHairInfo ∗ *chi*)

Convenient function to draw cross hair on camera image. It is very tedious to draw and maintain cross hair drawing. This is due to evolving eyelink camera image protocol and the introduction of newer trackers and still single library handle all eyelink trackers. eyelink_draw_cross_hair fixes this issue by allowing the drawing function to draw to the device contexts and does all magic of extracting cursor info from the tracker.

eyelink_draw_cross_hair calls drawLine(), drawEllipse() and getMouseState() to draw the cross hair. eyelink_draw_cross_hair expects both width(w) and height(h) are correct and the function pointers to drawLine, drawEllipse and getMouseState are not NULL.

Parameters:

← ***chi*** initialized CrossHairInfo structure.

26.8.3.2 HOOKFCNS∗ get_all_hook_functions ()

Returns a pointer to HOOKFCNS, with values that are set by setup_graphic_hook_functions().

This function with combination of setup_graphic_hook_functions can be used to over-ride an existing hook function.

26.8.3.3 HOOKFCNS2∗ get_all_hook_functions_V2 ()

Returns a pointer to HOOKFCNS2, with values that are set by setup_graphic_hook_functions_V2().

This function with combination of setup_graphic_hook_functions can be used to over-ride an existing hook function.

26.8.3.4 int get_image_xhair_data (INT16 *x*[4], INT16 *y*[4], INT16 ∗ *xhairs_on*)

Value is image coord scaled so l,t=0, r,b=8192 Values may be negative or beyond image limits Value is 0x8000 in X (or L) field if crosshair is not to be drawn Head camera: indexes 0..3 are markers Eye camera: Index 0 is pupil center Index 1 is CR center Index 2 is pupil-limit box left, top Index 3 is pupil-limit box right, bottom

Parameters:

xhairs_on Set to indicate if xhairs enabled on display (may be `NULL`).

x array of 4 to be filled to return x data

y array of 4 to be filled to return y data

Returns:

Channel number (`1` = left, `2` = head, `3` = right).

26.8.3.5 INT16 image_mode_display (void)

This handles display of the EyeLink camera images. While in imaging mode, it contiuously requests and displays the current camera image. It also displays the camera name and threshold setting. Keys on the subject PC keyboard are sent to the tracker, so the experimenter can use it during setup. It will exit when the tracker leaves imaging mode or disconnects.

Returns:

`0` if OK, `TERMINATE_KEY` if pressed, `-1` if disconnect.

Remarks:

This function not normally used externally. If you need camera setup use do_tracker_setup() or if you need drift correction use do_drift_correct()

26.8.3.6 void setup_graphic_hook_functions (HOOKFCNS ∗ *hooks*)

Primary function to setup display graphics hook functions.

Example:

```
INT16 ELCALLTYPE init_expt_graphics(HWND hwnd, DISPLAYINFO *info)
{
  HOOKFCNS fcns;
  memset(&fcns,0,sizeof(fcns));
  fcns.setup_cal_display_hook = setup_cal_display;
  fcns.exit_cal_display_hook  = exit_cal_display;
  fcns.record_abort_hide_hook = record_abort_hide;
  fcns.setup_image_display_hook = setup_image_display;
  fcns.image_title_hook       = image_title;
  fcns.draw_image_line_hook   = draw_image_line;
  fcns.set_image_palette_hook = set_image_palette;
  fcns.exit_image_display_hook= exit_image_display;
  fcns.clear_cal_display_hook = clear_cal_display;
  fcns.erase_cal_target_hook  = erase_cal_target;
  fcns.draw_cal_target_hook   = draw_cal_target;
  fcns.cal_target_beep_hook   = cal_target_beep;
  fcns.cal_done_beep_hook     = cal_done_beep;
  fcns.dc_done_beep_hook      = dc_done_beep;
  fcns.dc_target_beep_hook    = dc_target_beep;
  fcns.get_input_key_hook     = get_input_key;
  fcns.alert_printf_hook      = alert_printf_hook;

  setup_graphic_hook_functions(&fcns);

  return 0;
}
```

26.8.3.7 INT16 setup_graphic_hook_functions_V2 (HOOKFCNS2 ∗ *hooks*)

Primary function to setup display graphics hook functions of the second revision of the interface. One of the major difference between this and setup_graphic_hook_functions is, this has fewer functions to implement also, takes extra parameters like the major and minor versions for future enhancements.

Example:

```
INT16 ELCALLTYPE init_expt_graphics(HWND hwnd, DISPLAYINFO *info)
{
        HOOKFCNS2 fcns;
        memset(&fcns,0,sizeof(fcns));
        fcns.major = 1;
        fcns.minor = 0;
        fcns.userData = ts;

        // cam image
        fcns.draw_image    = draw_image;
        fcns.exit_image_display_hook= exit_image_display;
        fcns.setup_image_display_hook = setup_image_display;



        fcns.setup_cal_display_hook = setup_cal_display;
        fcns.clear_cal_display_hook = clear_display;
        fcns.erase_cal_target_hook  = clear_display;
        fcns.draw_cal_target_hook   = draw_cal_target;
        fcns.image_title_hook       = image_title;

        fcns.get_input_key_hook     = get_input_key;
```

```
            fcns.alert_printf_hook      = alert_printf_hook;
            return setup_graphic_hook_functions_V2(&fcns);
    }
```

26.8.3.8 INT16 target_mode_display (void)

This function needs some "helper" graphics to clear the scren and draw the fixation targets. Since C graphics are compiler-dependent, these are found in other C source files.

While tracker is in any mode with fixation targets. Reproduce targets tracker needs. (if local_trigger) Local Spacebar acts as trigger. (if local_control) Local keys echoes to tracker.

Returns:
`0` if OK, `27` if aborted, `TERMINATE_KEY` if pressed.

26.9 Extract extended data from samples and events

Functions

- INT32 eyelink_initialize_mapping (float left, float top, float right, float bottom)
- INT32 eyelink_href_to_gaze (float ∗xp, float ∗yp, FSAMPLE ∗sample)
- INT32 eyelink_gaze_to_href (float ∗xp, float ∗yp, FSAMPLE ∗sample)
- float eyelink_href_angle (float x1, float y1, float x2, float y2)
- void eyelink_href_resolution (float x, float y, float ∗xres, float ∗yres)

26.9.1 Function Documentation

26.9.1.1 INT32 eyelink_gaze_to_href (float ∗ *xp*, float ∗ *yp*, FSAMPLE ∗ *sample*)

Apply gaze->href to point (xp, yp). This function requires HREFPT data in FSAMPLE. The function `eyelink_initialize_mapping()` should be called before calling this function.

Parameters:

xp X point to apply gaze->href.

yp Y point to apply gaze->href.

sample Sample.

Returns:

`0` if OK, `-1` if there is a math error, `-2` if the tracker does not support this operation.

26.9.1.2 float eyelink_href_angle (float *x1*, float *y1*, float *x2*, float *y2*)

Convenient function to calculate the href angle.

Parameters:

x1 Point 1 x.

y1 Point 1 y.

x2 Point 2 x.

y2 Point 2 y.

26.9.1.3 void eyelink_href_resolution (float *x*, float *y*, float ∗ *xres*, float ∗ *yres*)

Convenient function to calculate the href resolution.

Parameters:

x X value of point.

y Y value of point.

xres Pointer to return the x resolution.

yres Pointer to return the y resolution.

26.9.1.4 INT32 eyelink_href_to_gaze (float ∗ *xp*, float ∗ *yp*, FSAMPLE ∗ *sample*)

Apply href->gaze to point (xp, yp). This function requires HREFPT data in FSAMPLE. The function `eyelink_initialize_mapping()` should be called before calling this function.

Parameters:

xp X point to apply href->gaze.

yp Y point to apply href->gaze.

sample Sample.

Returns:

`0` if OK, `-1` if there is a math error, `-2` if the tracker does not support this operation.

26.9.1.5 INT32 eyelink_initialize_mapping (float *left*, float *top*, float *right*, float *bottom*)

Function to initialize the gaze to href and href to gaze mapping. This function should be called before calling `eyelink_href_to_gaze()` or `eyelink_gaze_to_href()`.

Parameters:

left Left pixel value (normally 0).

top Top pixel value (normally 0).

right Right pixel value (width of the display).

bottom Bottom pixel value (height of the display).

Returns:

`-1` if the tracker does not support the commands, href_point_eye set_href_point

26.10 Time stamped messages to log file

Functions

- INT16 timemsg_printf (UINT32 t, char ∗fmt,...)
- int open_message_file (char ∗fname)
- void close_message_file (void)

26.10.1 Function Documentation

26.10.1.1 void close_message_file (void)

Flush and close message file, opened by open_message_file.

26.10.1.2 int open_message_file (char ∗ *fname*)

Creates message file, once open call to timemsg_printf(), will not send messages to tracker. Messages are kept in a queue if the application is in realtime mode, and written to disk on non real-time mode except when close_message_file() is called while in real-time mode.

Parameters:

← ***fname*** Message file name

26.10.1.3 INT16 timemsg_printf (UINT32 *t*, char ∗ *fmt*, ...)

Very similar to eyemsg_printf, with the following features:

- Optionally write the timestamped message to file instead of sending over the link to tracker.
- Optional time of event.

Parameters:

t optional time of event (0 = NOW)

fmt format messages

26.11 Online velocity and acceleration calculation

Defines

- #define **FIVE_SAMPLE_MODEL** 1
- #define **NINE_SAMPLE_MODEL** 2
- #define **SEVENTEEN_SAMPLE_MODEL** 3
- #define **EL1000_TRACKER_MODEL** 4

Functions

- int eyelink_calculate_velocity_x_y (int slen, float xvel[2], float yvel[2], FSAMPLE ∗vel_sample)
- int eyelink_calculate_velocity (int slen, float vel[2], FSAMPLE ∗vel_sample)
- int eyelink_calculate_overallvelocity_and_acceleration (int slen, float vel[2], float acc[2], FSAMPLE ∗vel_sample)

26.11.1 Function Documentation

26.11.1.1 int eyelink_calculate_overallvelocity_and_acceleration (int *slen*, float *vel*[2], float *acc*[2], FSAMPLE ∗ *vel_sample*)

Calculates overall velocity and acceleration for left and right eyes separately.

Parameters:

← ***slen*** Sample model to use for velocity calculation. Acceptable models are `FIVE_SAMPLE_MODEL`, `NINE_SAMPLE_MODEL`, `SEVENTEEN_SAMPLE_MODEL` and `EL1000_TRACKER_MODEL`.

→ ***vel*** A float array of 2 to fill in the calculated velocity results. Upon return of this function, vel[0] will contain overall velocity for left eye and vel[1] will contain overall velocity for right eye. If velocity cannot be calculated for any reason(eg. insufficient samples, no data) MISSING_DATA is filled for the given velocity.

→ ***acc*** A float array of 2 to fill in the calculated acceleration results. Upon return of this function, acc[0] will contain overall acceleration for left eye and acc[1] will contain overall acceleration for right eye. If acceleration cannot be calculated for any reason(eg. insufficient samples, no data) MISSING_DATA is filled for the given acceleration.

→ ***vel_sample*** Velocity for sample. Expects a FSAMPLE structure to fill in the sample, the velocity is calculated for.

26.11.1.2 int eyelink_calculate_velocity (int *slen*, float *vel*[2], FSAMPLE ∗ *vel_sample*)

Calculates overall velocity for left and right eyes separately.

Parameters:

← ***slen*** Sample model to use for velocity calculation. Acceptable models are `FIVE_SAMPLE_MODEL`, `NINE_SAMPLE_MODEL`, `SEVENTEEN_SAMPLE_MODEL` and `EL1000_TRACKER_MODEL`.

→ ***vel*** A float array of 2 to fill in the calculated results. Upon return of this function, vel[0] will contain overall velocity for left eye and vel[1] will contain overall velocity for right eye. If velocity cannot be calculated for any reason(eg. insufficient samples, no data) MISSING_DATA is filled for the given velocity.

→ ***vel_sample*** Velocity for sample. Expects a FSAMPLE structure to fill in the sample, the velocity is calculated for.

26.11.1.3 int eyelink_calculate_velocity_x_y (int *slen*, float *xvel*[2], float *yvel*[2], FSAMPLE ∗ *vel_sample*)

Calculates left x velocity, left y velocity, right x velocity and right y velocity from queue of samples.

Parameters:

← ***slen*** Sample model to use for velocity calculation. Acceptable models are `FIVE_SAMPLE_MODEL`, `NINE_SAMPLE_MODEL`, `SEVENTEEN_SAMPLE_MODEL` and `EL1000_TRACKER_MODEL`.

→ ***xvel*** Left and right x velocity. Expects an array of 2 elements of floats. The array is filled with left and right velocity values. Upon return of this function xvel[0] contains the left x velocity data and xvel[1] contains right x velocity data. If velocity cannot be calculated for any reason(eg. insufficient samples, no data) MISSING_DATA is filled for the given velocity.

→ ***yvel*** Left and right y velocity. Expects an array of 2 elements of floats. The array is filled with left and right velocity values. Upon return of this function yvel[0] contains the left y velocity data and xvel[1] contains right y velocity data. If velocity cannot be calculated for any reason(eg. insufficient samples, no data) MISSING_DATA is filled for the given velocity.

→ ***vel_sample*** Velocity for sample. Expects a FSAMPLE structure to fill in the sample, the velocity is calculated for.

```
#include <stdio.h>
#include <core_expt.h>

int main(int argc, char ** argv)
{
        if(open_eyelink_connection(0)) // connect to tracker
        {
                return -1;
        }

        eyecmd_printf("link_sample_data  = LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS"); // tell the tracker to s
        if(start_recording(0,0,1,0)) // start recording failed.
        {
                close_eyelink_connection();
                return -1;
        }

        if(!eyelink_wait_for_block_start(100, 1, 0)) // wait for block start
        {
                stop_recording();
                close_eyelink_connection();
                return -1;
        }
        else
        {
                UINT32 st = current_time();
                while(current_time()-st<10000) // record for 10 seconds
                {
                        FSAMPLE fs;
                        float xvel[2];
                        float yvel[2];
                        if(check_recording()!=0)
                        {
                                close_eyelink_connection();
                                return -4; // recording aborted.
                        }
                        eyelink_calculate_velocity_x_y(FIVE_SAMPLE_MODEL,xvel,yvel,&fs);
                        printf("%lu %f %f %f %f\n",fs.time,xvel[0],yvel[0], xvel[1], yvel[1]);
                        pump_delay(100); // we check the velocity every 100 ms.
                }
                stop_recording();
                close_eyelink_connection();
                return 0;
        }


}
```

26.12 Utility function to save bitmaps

Defines

- #define BX_AVERAGE 0
- #define BX_DARKEN 1
- #define BX_LIGHTEN 2
- #define BX_MAXCONTRAST 4
- #define BX_NODITHER 8
- #define BX_GRAYSCALE 16
- #define BX_DOTRANSFER 256
- #define SV_NOREPLACE 1
- #define SV_MAKEPATH 2

Functions

- int el_bitmap_save_and_backdrop (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 xferoptions)
- int el_bitmap_to_backdrop (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 xferoptions)
- int el_bitmap_save (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- int set_write_image_hook (int(∗hookfn)(char ∗outfilename, int format, EYEBITMAP ∗bitmap), int options)

26.12.1 Define Documentation

26.12.1.1 #define BX_AVERAGE 0

Average combined pixels

26.12.1.2 #define BX_DARKEN 1

Choose darkest (keep thin dark lines)

26.12.1.3 #define BX_DOTRANSFER 256

Send bitmap to host

26.12.1.4 #define BX_GRAYSCALE 16

Gray scale

26.12.1.5 #define BX_LIGHTEN 2

Choose darkest (keep thin white lines)

26.12.1.6 #define BX_MAXCONTRAST 4

Stretch contrast to black->white

26.12.1.7 #define BX_NODITHER 8

No dither, just quantize

26.12.1.8 #define SV_MAKEPATH 2

make destination path if does not already exists

26.12.1.9 #define SV_NOREPLACE 1

do not replace if the file already exists

26.12.2 Function Documentation

26.12.2.1 int el_bitmap_save (EYEBITMAP ∗ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*)

This function saves the entire bitmap or selected part of a bitmap in an image file (with an extension of .png, .bmp, .jpg, or .tif). It creates the specified file if this file does not exist.

Parameters:

hbm Bitmap to save or transfer or both.

xs X position.

ys Y position.

width Width.

height Height.

fname File name to save as. The extension decides the format of the file.

path The directory to which the file will be written.

sv_options If the file exists, it replaces the file unless SV_NOREPLACE is specified.

Remarks:

This function relies on the writeImageHook set by call to set_write_image_hook() to write the images in different formats. By default, if eyelink_core_graphics library is used, gd library is used to write the images and if eyelink_gdi_graphics is used FreeImage library is used to write the images. If neither one of them is used, call to this function does not write the images unless, set_write_image_hook() is used to set the writeImageHook.
This function should not be called when timing is critical, as this might take very long to return.

26.12.2.2 int el_bitmap_save_and_backdrop (EYEBITMAP ∗ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*, INT16 *xd*, INT16 *yd*, UINT16 *xferoptions*)

This function saves the entire bitmap as a .BMP, .JPG, .PNG, or .TIF file, and transfers the image to tracker as backdrop for gaze cursors.

Parameters:

hbm Bitmap to save or transfer or both.

xs X position.

ys Y position.

width Width.

height Height.

fname File name to save as. The extension decides the format of the file.

path The directory to which the file will be written.

sv_options If the file exists, it replaces the file unless SV_NOREPLACE is specified.

xd X positon.

yd Y positon.

xferoptions Transfer options set with bitwise OR of the following constants, determines how bitmap is processed:

- `BX_AVERAGE` Averaging combined pixels
- `BX_DARKEN` Choosing darkest and keep thin dark lines.
- `BX_LIGHTEN` Choosing darkest and keep thin white lines and control how bitmap size is reduced to fit tracker display.
- `BX_MAXCONTRAST` Maximizes contrast for clearest image.
- `BX_NODITHER` Disables the dithering of the image.
- `BX_GREYSCALE` Converts the image to grayscale (grayscale works best for EyeLink I, text, etc.).

See also:

el_bitmap_to_backdrop(), el_bitmap_save(), sdl_bitmap_to_backdrop(), sdl_bitmap_save(),sdl_bitmap_save_and_backdrop, gdi_bitmap_to_backdrop(), gdi_bitmap_save(),gdi_bitmap_save_and_backdrop, bitmap_save(), and bitmap_to_backdrop() for more information.

Remarks:

This function relies on the writeImageHook set by call to set_write_image_hook() to write the images in different formats. By default, if eyelink_core_graphics library is used, gd library is used to write the images and if eyelink_gdi_graphics is used FreeImage library is used to write the images. If neither one of them is used, call to this function does not write the images unless, set_write_image_hook() is used to set the writeImageHook.
This function should not be called when timing is critical, as this might take very long to return.

26.12.2.3 int el_bitmap_to_backdrop (EYEBITMAP $*$ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, INT16 *xd*, INT16 *yd*, UINT16 *xferoptions*)

This function transfers the bitmap to the tracker PC as backdrop for gaze cursors.

Parameters:

hbm Bitmap to save or transfer or both.

xs X position.

ys Y position.

width Width.

height Height.

xd X positon.

yd Y positon.

xferoptions Transfer options set with bitwise OR of the following constants, determines how bitmap is processed:

- `BX_AVERAGE` Averaging combined pixels
- `BX_DARKEN` Choosing darkest and keep thin dark lines.
- `BX_LIGHTEN` Choosing darkest and keep thin white lines and control how bitmap size is reduced to fit tracker display.
- `BX_MAXCONTRAST` Maximizes contrast for clearest image.
- `BX_NODITHER` Disables the dithering of the image.
- `BX_GREYSCALE` Converts the image to grayscale (grayscale works best for EyeLink I, text, etc.).

See also:

sdl_bitmap_to_backdrop(), el_bitmap_save_and_backdrop(), sdl_bitmap_save_and_backdrop(), gdi_-bitmap_to_backdrop(), gdi_bitmap_save_and_backdrop(), and bitmap_to_backdrop() for more information.

Remarks:

This function should not be called when timing is critical, as this might take very long to return.

26.12.2.4 int set_write_image_hook (int(∗)(char ∗outfilename, int format, EYEBITMAP ∗bitmap) *hookfn*, int *options*)

Use this function to set function pointer, so that the call to el_bitmap_save and el_bitmap_save_and_-backdrop will use the passed in function to write the image to disk.

26.13 Record control and data collection

26.13.1 Detailed Description

Functions and constants to control recording and data collection.

Defines

- #define RECORD_FILE_SAMPLES 1
- #define RECORD_FILE_EVENTS 2
- #define RECORD_LINK_SAMPLES 4
- #define RECORD_LINK_EVENTS 8
- #define DONE_TRIAL 0
- #define TRIAL_OK 0
- #define REPEAT_TRIAL 1
- #define SKIP_TRIAL 2
- #define ABORT_EXPT 3
- #define TRIAL_ERROR -1

Functions

- INT16 eyelink_in_data_block (INT16 samples, INT16 events)
- INT16 eyelink_wait_for_block_start (UINT32 maxwait, INT16 samples, INT16 events)
- INT16 eyelink2_mode_data (INT16 ∗sample_rate, INT16 ∗crmode, INT16 ∗file_filter, INT16 ∗link_filter)
- INT16 eyelink_mode_data (INT16 ∗sample_rate, INT16 ∗crmode, INT16 ∗file_filter, INT16 ∗link_-filter)
- INT16 start_recording (INT16 file_samples, INT16 file_events, INT16 link_samples, INT16 link_-events)
- INT16 check_recording (void)
- void stop_recording (void)
- INT16 check_record_exit (void)

26.13.2 Define Documentation

26.13.2.1 #define ABORT_EXPT 3

return codes for trial result

26.13.2.2 #define DONE_TRIAL 0

return codes for trial result

26.13.2.3 #define RECORD_FILE_EVENTS 2

only active if file open

26.13.2.4 #define RECORD_FILE_SAMPLES 1

only active if file open

26.13.2.5 #define RECORD_LINK_EVENTS 8

accept events from link

26.13.2.6 #define RECORD_LINK_SAMPLES 4

accept samples from link

26.13.2.7 #define REPEAT_TRIAL 1

return codes for trial result

26.13.2.8 #define SKIP_TRIAL 2

return codes for trial result

26.13.2.9 #define TRIAL_ERROR -1

Bad trial: no data, etc.

26.13.2.10 #define TRIAL_OK 0

return codes for trial result

26.13.3 Function Documentation

26.13.3.1 INT16 check_record_exit (void)

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.

Returns:
`TRIAL_OK` if no error.
`REPEAT_TRIAL, SKIP_TRIAL, ABORT_EXPT` if Abort menu activated.

Example: See `start_recording()`

See also:
`check_recording()`, `eyelink_abort()`, `start_recording()` and `stop_-recording()`

26.13.3.2 INT16 check_recording (void)

Check if we are recording: if not, report an error. Call this function while recording. It will return `0` if recording is still in progress, or an error code if not. It will also handle the EyeLink Abort menu by calling `record_abort_handler()`. Any errors returned by this function should be returned by the trial function. On error, this will disable realtime mode and restore the heuristic.

Returns:

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

Example: See `start_recording()`

See also:

`check_record_exit()`, `eyelink_abort()`, `start_recording()` and `stop_recording()`

26.13.3.3 INT16 eyelink2_mode_data (INT16 ∗ *sample_rate*, INT16 ∗ *crmode*, INT16 ∗ *file_filter*, INT16 ∗ *link_filter*)

26.13.3.4 INT16 eyelink_in_data_block (INT16 *samples*, INT16 *events*)

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

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 SLL before v2.0 (did not detect end of blocks).

Parameters:

samples If non-zero, check if in a block with samples.

events If non-zero, check if in a block with events.

Returns:

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

Example:

```
// This program illustrates the use of eyelink_in_data_block in a broadcast connection.
// First a broadcast connection is opened and data reception from the tracker is reset.
// Following that, checks whether the block contains samples or events and make use of this inform

#include <eyelink.h>
#include <stdio.h>

// Initializes the link
if(open_eyelink_connection(-1))
    return -1;

...
// Extra code here to check for the tracker status or
// wait for the go-signal from the other application
// Starts the broadcast connection to the tracker
if(eyelink_broadcast_open())
{
    printf("Cannot open broadcast connection to tracker");
    return -1;
}
// Enables link data reception by EyeLink DLL
eyelink_reset_data(1);

// NOTE: this function can discard some link data
eyelink_data_switch(RECORD_LINK_SAMPLES | RECORD_LINK_EVENTS);

// Makes use of the link data
while(eyelink_is_connected())
{
    if(escape_pressed() || break_pressed()) return;

    //  check for new data item
    i = eyelink_get_next_data(NULL);
    if(i == 0) continue;

    // link data block available?
    if(eyelink_in_data_block(1, 1))
    {
        ...
        // Code to read the link data, etc.
    }
}
```

See also:
`eyelink_data_status()` and `eyelink_wait_for_block_start()`

26.13.3.5 INT16 eyelink_mode_data (INT16 ∗ *sample_rate*, INT16 ∗ *crmode*, INT16 ∗ *file_filter*, INT16 ∗ *link_filter*)

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

Parameters:

sample_rate `NULL`, or pointer to variable to be filled with samples per second.

crmode `NULL`, or pointer to variable to be filled with CR mode flag (`0` if pupil-only mode, else pupil-CR mode).

file_filter `NULL`, or pointer to variable to be filled with filter level to be applied to file samples (`0` = off, `1` = std, `2` = double filter).

link_filter `NULL`, or pointer to variable to be filled with filter level to be applied to link and analog output samples (`0` = off, `1` = std, `2` = double filter).

Returns:

If no data available -1 else 0.

Example:

```
// This program illustrates the use of eyelink2_mode_data to check the sample rate and tracking
// mode of the application

#inlcude <eyelink.h>
#include <stdio.h>
```

```
int is_eyelink2;

// waits till a block of samples, events, or both is begun
if(!eyelink_wait_for_block_start(2000, 1, 1))
{
    printf("ERROR: No sample or event been detected!");
    return -1;
}

// Gets tracker version
is_eyelink2 = (2 == eyelink_get_tracker_version(NULL));

// For EyeLink II, determine sample rate and tracking mode
if(is_eyelink2 && !eyelink2_mode_data(&sample_rate, &crmode, NULL, NULL))
{
    eyemsg_printf("Sample rate: %d", sample_rate);
    eyemsg_printf("Tracking mode: %s", crmode?"CR":"Pupil only");
}
```

Output:

```
MSG     1151024 Sample rate: 250
MSG     1151024 Tracking mode: CR
```

26.13.3.6 INT16 eyelink_wait_for_block_start (UINT32 *maxwait*, INT16 *samples*, INT16 *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 `start_recording()`.

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

Parameters:

maxwait 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.

Returns:
`0` if time expired without any data of masked types available.

Example:

```
// This program illustrates the use of eyelink_wait_for_block_start

#include <eyelink.h>
#include <stdio.h>

// Starts recording with both sample and events to the file and link
if(start_recording(1,1,1,1)!= 0)
    return -1;          // ERROR: couldn't start recording

// record for 100 msec before displaying stimulus
begin_realtime_mode(100);
```

```
// wait for link sample data
if(!eyelink_wait_for_block_start(100, 1, 0))
{
      printf("ERROR: No link samples received!");
      return TRIAL_ERROR;
}

// determine which eye(s) are available
eye_used = eyelink_eye_available();
switch(eye_used) // select eye, add annotation to EDF file
{
      case RIGHT_EYE:
        eyemsg_printf("EYE_USED 1 RIGHT");
        break;
      case BINOCULAR:  // both eye's data present: use left eye only
        eye_used = LEFT_EYE;
      case LEFT_EYE:
        eyemsg_printf("EYE_USED 0 LEFT");
        break;
}
```

26.13.3.7 INT16 start_recording (INT16 *file_samples*, INT16 *file_events*, INT16 *link_samples*, INT16 *link_events*)

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

Remarks:

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.

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.

Returns:

`0` if successful, else trial return code.

Example:

```
// This program illustrates the use of start_recording(), stop_recording(), checking_recording(),
// and check_record_exit()

#include <eyelink.h>

// Starts data recording to EDF file
// Records samples and events to EDF file only in this example
// Returns error code if failed
error = start_recording(1,1,0,0);
if(error != 0)  return error;

// Sets up for realtime execution
begin_realtime_mode(100);

...
// Display drawing code here
// Trial loop
while(1)
{
    // Checks if recording aborted
    if((error=check_recording())!=0) return error;
    ...
    // Other code for display updates, timing, key or button
    // response handling
}

// Ensures we release realtime lock
end_realtime_mode();
// Records additional 100 msec of data
pump_delay(100);
// halt recording, return when tracker finished mode switch
stop_recording();

while(getkey());          // dump any accumulated key presses

// Call this at the end of the trial, to handle special conditions
return check_record_exit();
```

See also:
`check_record_exit()`, `check_recording()`, `eyelink_data_start()` and `stop_recording()`

26.13.3.8 void stop_recording (void)

Stops recording, resets EyeLink data mode.

Remarks:
Call 50 to 100 msec after an event occurs that ends the trial. This function waits for mode switch before returning.

Example: See `start_recording()`

See also:
`eyelink_data_stop()`, `set_offline_mode()` and `start_recording()`

26.14 Accessing and reporting error messages

Functions

- void alert_printf (char ∗fmt,...)
- char ∗ eyelink_get_error (int id, char ∗function_name)

26.14.1 Function Documentation

26.14.1.1 void alert_printf (char ∗ *fmt*, ...)

When an error occurs, a notification must be given to the user. If no alert_printf_hook is set, this function uses the Windows MessageBox() function in windows. On other platforms printf is called.

Parameters:

fmt A printf() formatting string<...>: any arguments required.

Remarks:

The error message may no showup in certain display environment eg. SDL with SDL_-FULLSCREEN|SDL_HWSURFACE | SDL_DOUBLEBUF

26.14.1.2 char∗ eyelink_get_error (int *id*, char ∗ *function_name*)

Returns error description for given function with error id. Example:

```
int rv = open_eyelink_connection(0);
if(rv)
{
        char *errmsg = eyelink_get_error(rv,"open_eyelink_connection");
        printf("Error: %s \n", errmsg); // report the error
        return -1;
}
```

Parameters:

← *id* Error id

← *function_name* Name of the function that generated the error id.

26.15 Playback and data acquisition.

Functions

- INT16 eyelink_playback_start (void)
- INT16 eyelink_playback_stop (void)

26.15.1 Function Documentation

26.15.1.1 INT16 eyelink_playback_start (void)

Flushes data from queue and starts data playback. An EDF file must be open and have at least one recorded trial. Use `eyelink_wait_for_data()` 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.

Returns:
`0` if command sent OK, else link error.

Example:

```
// This program illustrates the use of eyelink_playback_start() and eyelink_playback_stop()
// functions for trial data playback.  See EYEDATA template for a complete example

#include <eyelink.h>
#include <stdio.h>

set_offline_mode();           // set up eye tracker for playback
eyelink_playback_start();     // start data playback

// wait for first data to arrive
if(!eyelink_wait_for_block_start(2000, 1, 1))
{
    printf("ERROR: playback data did not start!");
    return -1;
}

while(1)   // Loop while data available
{
    // Exit if ESC, ALT-F4, CTRL-C, or tracker button pressed
    if(escape_pressed() || break_pressed() ||
        eyelink_last_button_press(NULL))
    {
        eyelink_playback_stop(); // stop playback
        return 0;
    }

    // Process playback data from the link
    i = eyelink_get_next_data(NULL);   // check for new data item
    if(i==0)                           // 0: no new data
    {   // Checks if playback has completed
        if((eyelink_current_mode() & IN_PLAYBACK_MODE)==0) break;
    }
    ...
    // Code for processing data and delay handling here
    ...
}

// End of playback loop
eyelink_playback_stop();
```

See also:
`eyelink_playback_stop()` and `eyelink_wait_for_block_start()`

26.15.1.2 INT16 eyelink_playback_stop (void)

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

Returns:
`0` if mode switched, else link error.

Example: See `eyelink_playback_start()`

See also:
`eyelink_playback_start()` and `eyelink_wait_for_block_start()`

26.16 Message and Command Sending/Receiving

Modules

- General Data Constants
- Sample Data Flags
- Eyelink Sample and Event Type Identifiers
- Read Data Flags
- Data Type Flags
- Event Data Flags
- Eyelink II/Eyelink I Marker flags
- Eyelink II specific flags
- Eyelink 1000 Remote specific target status flags

Data Structures

- struct ISAMPLE

  Integer sample data.

- struct FSAMPLE

  Floating-point sample.

- struct DSAMPLE

  Floating-point sample with floating point time.

- struct IEVENT

  Integer eye-movement events.

- struct FEVENT

  Floating-point eye event.

- struct DEVENT

  Floating-point eye event with floating point time.

- struct IMESSAGE

  Message events: usually text but may contain binary data.

- struct DMESSAGE

  Message events: usually text but may contain binary data with floating point time.

- struct IOEVENT

  Button, input, other simple events.

- struct DIOEVENT

  Button, input, other simple events with floating point time.

- union ALL_DATA

  Union of message, io event and integer sample and integer event.

- union ALLF_DATA

  Union of message, io event and float sample and float event.

- union ALLD_DATA

  Union of message, io event and double sample and double event.

Functions

- INT16 eyelink_node_send (ELINKADDR node, void FARTYPE ∗data, UINT16 dsize)
- INT16 eyelink_node_receive (ELINKADDR node, void FARTYPE ∗data)
- INT16 eyelink_send_command (char FARTYPE ∗text)
- INT16 eyelink_command_result (void)
- INT16 eyelink_timed_command (UINT32 msec, char FARTYPE ∗text)
- INT16 eyelink_last_message (char FARTYPE ∗buf)
- INT16 eyelink_send_message (char FARTYPE ∗msg)
- INT16 eyelink_node_send_message (ELINKADDR node, char FARTYPE ∗msg)
- INT16 eyelink_send_message_ex (UINT32 exectime, char FARTYPE ∗msg)
- INT16 eyelink_node_send_message_ex (UINT32 exectime, ELINKADDR node, char FARTYPE ∗msg)
- INT16 eyelink_read_request (char FARTYPE ∗text)
- INT16 eyelink_read_reply (char FARTYPE ∗buf)
- INT16 eyelink_position_prescaler (void)
- INT16 eyelink_reset_data (INT16 clear)
- void FARTYPE ∗ eyelink_data_status (void)
- INT16 eyelink_get_next_data (void FARTYPE ∗buf)
- INT16 eyelink_get_last_data (void FARTYPE ∗buf)
- INT16 eyelink_newest_sample (void FARTYPE ∗buf)
- INT16 eyelink_get_float_data (void FARTYPE ∗buf)
- INT16 eyelink_get_double_data (void FARTYPE ∗buf)
- INT16 eyelink_newest_float_sample (void FARTYPE ∗buf)
- INT16 eyelink_newest_double_sample (void FARTYPE ∗buf)
- INT16 eyelink_eye_available (void)
- UINT16 eyelink_sample_data_flags (void)
- UINT16 eyelink_event_data_flags (void)
- UINT16 eyelink_event_type_flags (void)
- INT16 eyelink_data_count (INT16 samples, INT16 events)
- INT16 eyelink_wait_for_data (UINT32 maxwait, INT16 samples, INT16 events)
- INT16 eyelink_get_sample (void FARTYPE ∗sample)
- INT16 eyelink_data_switch (UINT16 flags)
- INT16 eyelink_data_start (UINT16 flags, INT16 lock)
- INT16 eyelink_data_stop (void)
- INT16 eyecmd_printf (const char ∗fmt,...)
- INT16 eyemsg_printf (const char ∗fmt,...)
- INT16 eyemsg_printf_ex (UINT32 exectime, const char ∗fmt,...)

26.16.1 Function Documentation

26.16.1.1 INT16 eyecmd_printf (const char ∗ *fmt*, ...)

The EyeLink tracker accepts text commands through the link. These commands may be used to configure the system, open data files, and so on.

Remarks:

The function waits up to 500 msec. for a success or failure code to be returned from the tracker, then returns the error code `NO_REPLY`. If you need more time, use `eyelink_timed_command()` instead.

Parameters:

fmt Similar to printf(), format string plus arguments.

Returns:

`0` if successfully executed, else error code.

Example:

```
// This program illustrates the use of eyecmd_printf()

#include <eyelink.h>

UINT32 t = current_msec();
int i, j;

// Draws a box on the tracker screen
eyecmd_printf("draw_box %d %d %d %d  7", 100, 100, 300, 400);

// Waits for a maximum of 1000 msec
while(current_msec()-t < 1000)
{
    // Checks for result from command execution
    i = eyelink_command_result();
    // Used to get more information on tracker result
    j = eyelink_last_message(buf);

    if (i == OK_RESULT)
    {
        eyemsg_printf("Command executed successfully: %s", j?buf:"");
        break;
    }
    else if (i!=NO_REPLY)
    {
        eyemsg_printf("Error in executing command: %s", j?buf:"");
        break;
    }
}
```

Output:

```
MSG     5312110 Command executed successfully: OK
```

See also:

`eyemsg_printf()`, `eyelink_send_command()` and `eyelink_timed_command()`

26.16.1.2 INT16 eyelink_command_result (void)

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

Returns:

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

Example: See `eyelink_send_command()`

See also:

`eyelink_last_message()`, `eyelink_send_command()` and `eyelink_timed_-command()`

26.16.1.3 INT16 eyelink_data_count (INT16 *samples*, INT16 *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.

Returns:

Total number of samples and events is in the queue.

Example:

```
// This program uses eyelink_data_count() to check whether the desired sample/event data
// have arrived after the start_recording()

// Check whether the desired sample/event data have arrived.  This function should be
// called after start_recording()
// link_samples: Whether samples should be sent over the link
// link_events: Whether events should be sent over the link

int check_sample_event_status(int link_samples, int link_events)
{
    UINT32 t = current_time();
    int i;

    // Checks the sample and event availability in 300 msec
    while(current_time() < t + 300)
    {
        int arrived = 1;
        // check that recording still OK
        if((i=check_recording())!=0) return i;

        // Checks whether the samples, if set, have arrived
        if(link_samples)
            if(eyelink_data_count(1,0)==0)
                arrived = 0;

        // Checks whether the samples, if set, have arrived
        if(link_events)
            if(eyelink_data_count(0,1)==0)
                arrived = 0;
        // Exit if desired data type(s) have arrived
        if(arrived) return 0;
    }

    // If the desired data type(s) didn't arrive within 300 msec
    return -1;
}
```

26.16.1.4 INT16 eyelink_data_start (UINT16 *flags*, INT16 *lock*)

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

- `RECORD_FILE_SAMPLES (1)` - only active if file open.
- `RECORD_FILE_EVENTS (2)` - only active if file open.
- `RECORD_LINK_SAMPLES (4)` - accept samples from link.
- `RECORD_LINK_EVENTS (8)` - accept events from link.

Remarks:

If `<lock>` is nonzero, the recording may only be terminated through `stop_recording()` or `eyelink_data_stop()`, or by the Abort menu (ŚCtrlŠ ŠAltŠ ŠAŠ keys on the eye tracker). If zero, the tracker 'ESC' key may be used to halt recording.

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.

Returns:

`0` if command sent OK, else link error.

Example:

```
// This program illustrates the use of eyelink_data_start() and eyelink_data_stop()
// functions for trial recording

#include <eyelink.h>

// data types requested for the EDF file and link data
INT16 file_samples,  file_events, link_samples, link_events;

...
// Functions to set the file and link data type
// f(file_samples,  file_events, link_samples, link_events);
...

// Checks whether we still have the connection
if(eyelink_is_connected())      return ABORT_EXPT;

i = eyelink_data_start((file_samples?RECORD_FILE_SAMPLES:0) |
                       (file_events?RECORD_FILE_EVENTS:0)   |
                       (link_samples?RECORD_LINK_SAMPLES:0) |
                       (link_events?RECORD_LINK_EVENTS:0) , 1);

if(i) return i;
```

```
// wait for mode change completion
i = eyelink_wait_for_mode_ready(500);
if(i==0) return TRIAL_ERROR;

// Checks that recording started OK
if((i = check_recording())!=0)
    return i;
...
 // Recording code here
...

// Stops data flow and ends recording
eyelink_data_stop();
// wait for mode change completion
eyelink_wait_for_mode_ready(500);
```

See also:
`eyelink_data_stop()`, `start_recording()` and `stop_recording()`

26.16.1.5 void FARTYPE∗ eyelink_data_status (void)

Updates buffer status (data count, etc), returns pointer to internal `ILINKDATA` structure.

Returns:
Pointer to `ILINKDATA` structure.

Example:

```
Example 1:
// This program illustrates the use of eyelink_data_status to retrieve the name of the current app

ILINKDATA* current_data;  // Stores the link state data

// Sets up the EyeLink system and connect to tracker
if(open_eyelink_connection(0))  return -1;

// Updates the link data
current_data = eyelink_data_status();
eyemsg_printf("the subject PC name %s ", current_data -> our_name);


Example 2:

// Checks link state of tracker
int preview_tracker_connection(void)
{
    UINT32 t, tt;
    ILINKDATA *idata = eyelink_data_status();
    // Accesses link status info

    // Forces tracker to send status and time
    eyelink_request_time();
    t = current_msec();
    while(current_msec()- t < 500)   // Waits for response
    {
        tt = eyelink_read_time();   // Will be nonzero if reply

        if(tt != 0)
        {   // Retrieves the current connection state
            if(idata->link_flags & LINK_BROADCAST)
                return LINK_BROADCAST;
            if(idata->link_flags & LINK_CONNECTED)
                return LINK_CONNECTED;
            else
                return 0;
        }

        message_pump(NULL);          // Keeps Windows happy
        if(break_pressed())
            return 1;    // Stops if program terminated
    }
    return -1;  // Failed (timed out)
}
```

Output:

```
    MSG     15252571 the subject PC name GOLDBERG
```

See also:
eyelink_in_data_block() and eyelink_reset_data()

26.16.1.6 INT16 eyelink_data_stop (void)

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

Remarks:
Should be followed by a call to eyelink_wait_for_mode_ready().

Returns:
0 if command sent OK, else link error.

Example: See eyelink_data_start()

See also:
eyelink_data_start() and eyelink_wait_for_mode_ready()

26.16.1.7 INT16 eyelink_data_switch (UINT16 *flags*)

Sets what data from tracker will be accepted and placed in queue.

Remarks:
This does not start the tracker recording, and so can be used with eyelink_broadcast_open(). It also does not clear old data from the queue. The data is set with a bitwise OR of these flags: RECORD_LINK_SAMPLES - send samples on link. RECORD_LINK_EVENTS - send events on link.

Parameters:
flags Bitwise OR flags.

Returns:
0 if OK else link error.

Example: See eyelink_in_data_block()

See also:
eyelink_in_data_block()

26.16.1.8 UINT16 eyelink_event_data_flags (void)

Returns the event data content flags.

Remarks:

This will be 0 if the data being read from queue is not in a block with events.

Returns:

Event data content flags: EVENT_VELOCITY if has velocity data. EVENT_PUPILSIZE if has pupil size data. EVENT_GAZERES if has gaze resolution. EVENT_STATUS if has status flags. EVENT_-GAZEXY if has gaze xy position. EVENT_HREFXY if has head-ref xy position. EVENT_PUPILXY if has pupil xy position. FIX_AVG_ONLY if only avg. data to fixation events. START_TIME_ONLY if only start-time in start events. PARSEDBY_GAZE if how events were generated. PARSEDBY_HREF. PARSEDBY_PUPIL.

Example:

```
// This program illustrates the use of eyelink_get_sample()

#include <eyelink.h>
#include <stdio.h>
int error;

// Recording with link data enabled
error = start_recording(1,1,1,0);
if(error != 0) return error;   // ERROR: couldn't start recording

// Wait for link sample data
if(!eyelink_wait_for_data(100, 1, 0))
{
    printf("ERROR: No link samples received!");
    return -1;
}

// gets event data/type content flag
emsg_printf("Event data%d Event type %d",
    eyelink_event_data_flags(), eyelink_event_type_flags());
```

Output:

```
MSG     2689937 Event data 26316 Event type 29760
```

See also:

eyelink_event_type_flags()

26.16.1.9 UINT16 eyelink_event_type_flags (void)

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

Returns:

A set of bit flags: LEFTEYE_EVENTS if has left eye events. RIGHTEYE_EVENTS if has right eye events. BLINK_EVENTS if has blink events. FIXATION_EVENTS if has fixation events. FIXUPDATE_EVENTS if has fixation updates. SACCADE_EVENTS if has saccade events. MESSAGE_EVENTS if has message events. BUTTON_EVENTS if has button events. INPUT_-EVENTS if has input port events.

Example: See `eyelink_event_data_flags()`

See also:
`eyelink_event_data_flags()`

26.16.1.10 INT16 eyelink_eye_available (void)

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

Returns:
One of these constants, defined in EYE_DATA.H: `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.

Example:

```
// This program illustrates the use of eyelink_eye_available()

#include <eyelink.h>

int eye_used = 0;       // indicates which eye's data to display

// Determines which eye(s) are available
eye_used = eyelink_eye_available();

// Selects eye, add annotation to EDF file
switch(eye_used)
{
    case RIGHT_EYE:
        eyemsg_printf("EYE_USED 1 RIGHT");
        break;
    case BINOCULAR:   // both eye's data present: use left eye only
        eye_used = LEFT_EYE;
    case LEFT_EYE:
        eyemsg_printf("EYE_USED 0 LEFT");
        break;
}
```

Output:

```
MSG     22157314 EYE_USED 0 LEFT
```

See also:
`eyelink_wait_for_block_start()`

26.16.1.11 INT16 eyelink_get_double_data (void FARTYPE * *buf*)

Reads the last item fetched by `eyelink_get_next_data()` into a buffer. The event is converted to a floating-point format with floating point time (`DSAMPLE` or `DEVENT`). This can handle both samples and events. The buffer type can be `ALLD_DATA` for both samples and events, `DSAMPLE` for a sample, or a specific event buffer.

Parameters:
buf Pointer to buffer for floating-point data: type is `ALLD_DATA` or `DSAMPLE`.

Returns:
0 if no data, SAMPLE_TYPE if sample, else event type code.

Example: See eyelink_get_next_data()

See also:
eyelink_get_last_data(), eyelink_get_next_data(), eyelink_newest_float_sample() eyelink_newest_double_sample() eyelink_get_float_data() and eyelink_newest_sample()

26.16.1.12 INT16 eyelink_get_float_data (void FARTYPE ∗ *buf*)

Reads the last item fetched by eyelink_get_next_data() into a buffer. The event is converted to a floating-point format (FSAMPLE or FEVENT). This can handle both samples and events. The buffer type can be ALLF_DATA for both samples and events, FSAMPLE for a sample, or a specific event buffer.

Parameters:
buf Pointer to buffer for floating-point data: type is ALLF_DATA or FSAMPLE.

Returns:
0 if no data, SAMPLE_TYPE if sample, else event type code.

Example: See eyelink_get_next_data()

See also:
eyelink_get_last_data(), eyelink_get_next_data(), eyelink_newest_float_sample() eyelink_newest_double_sample() eyelink_get_double_data() and eyelink_newest_sample()

26.16.1.13 INT16 eyelink_get_last_data (void FARTYPE ∗ *buf*)

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

Parameters:
buf Pointer to buffer (ISAMPLE, IEVENT, or ALL_DATA type).

Returns:
0 if no data, SAMPLE_TYPE if sample, else event type code.

Example:

```
ALLF_DATA evt;         // buffer to hold sample and event data
int eye_used = -1;     // indicates which eye's data to display
int i;
UINT32 prev_event_time = -1;

i = eyelink_get_next_data(NULL);  // Checks for data from link
if(i == ENDFIX)   // only process ENDFIX events
{
    eyelink_get_float_data(&evt);  // get a copy of the ENDFIX event
    if(evt.fe.eye == eye_used)  // only process data from desired eye?
```

```
                eyemsg_printf("NEXT Event: %ld %ld", evt.fe.sttime,
                        evt.fe.entime);
        }

        // makes copy of last item from eyelink_get_next_data
        i = eyelink_get_last_data(&evt);
        if (i == ENDFIX && evt.fe.eye == eye_used
            && evt.fe.entime != prev_event_time)
        {
            eyemsg_printf("LAST Event: %ld %ld", evt.fe.sttime, evt.fe.entime);
            // Makes sure that we do not write out redundant information
            prev_event_time = evt.fe.entime;
        }
```

Output:

```
        SFIX L   7812616
        SFIX R   7812616
        EFIX L   7812616        7813232 620       124.7   274.0      50
        EFIX R   7812616        7813232 620       118.2   262.4      50
        SSACC L  7813236
        SSACC R  7813236
        MSG     7813254 NEXT Event: 7812616 7813232
        MSG     7813254 LAST Event: 7812616 7813232
```

See also:

eyelink_get_float_data(), eyelink_get_next_data() and eyelink_newest_-float_sample()

26.16.1.14 INT16 eyelink_get_next_data (void FARTYPE ∗ *buf*)

Fetches next data item from link buffer. Usually called with <buf> = NULL, and returns the data item type. If the item is not wanted, simply ignore it. Otherwise, call eyelink_get_float_data() to read it into a buffer.

Parameters:

buf If NULL, saves data, else copies integer data into buffer.

Returns:

0 if no data, SAMPLE_TYPE if sample, else event type.

Example:

```
        // This program illustrates the use of eyelink_get_next_data() and eyelink_get_float_data()

        #include <eyelink.h>

        ALLF_DATA evt;          // buffer to hold sample and event data
        int eye_used = -1;      // indicates which eye's data to display
        int i;

        // Determines which eye(s) are available
        eye_used = eyelink_eye_available();

        // Selects eye, add annotation to EDF file
        switch(eye_used)
        {
            case RIGHT_EYE:
                eyemsg_printf("EYE_USED 1 RIGHT");
                break;
            case BINOCULAR:   // both eye's data present: use left eye only
                eye_used = LEFT_EYE;
            case LEFT_EYE:
                eyemsg_printf("EYE_USED 0 LEFT");
                break;
        }
        while(1)
        {
            // Check for data from link and process fixation update events
            i = eyelink_get_next_data(NULL);
            if(i == FIXUPDATE)
            {
                // get a copy of the FIXUPDATE event
                eyelink_get_float_data(&evt);
                // only process if it's from the desired eye?
                if(evt.fe.eye == eye_used)
                {
                    // Records the average position and duration of update
                    eyemsg_printf("Fixupdate: avg_x %8.2f, y %8.2f, dur %d",
                        evt.fe.gavx, evt.fe.gavy, evt.fe.entime-evt.fe.sttime);
                }
            }
            ...
            // Other code for drawing and exiting
        }
```

Output:

```
MSG     30244970 Fixupdate: avg_x   863.10, y   244.10, dur 48
MSG     30245018 Fixupdate: avg_x   863.10, y   245.60, dur 48
MSG     30245070 Fixupdate: avg_x   863.10, y   247.30, dur 48
```

See also:
`eyelink_get_float_data()`, `eyelink_get_last_data()`, `eyelink_newest_float_sample()` and `eyelink_newest_sample()`

26.16.1.15 INT16 eyelink_get_sample (void FARTYPE ∗ *sample*)

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

Parameters:
sample Pointer to buffer (`ISAMPLE` or `ALL_DATA` type).

Returns:
`0` if no data. `1` if data retrieved.

Example:

```
// This program illustrates the use of eyelink_get_sample()

#include <eyelink.h>

ISAMPLE   isample; // INTEGER SAMPLE DATA
int eye_used = 0;  // indicates which eye's data to display
float x, y;                 // gaze position
int i;

// wait for link sample data
```

```
if(!eyelink_wait_for_block_start(100, 1, 0))
{
    end_trial();
    return TRIAL_ERROR;
}

// determine which eye(s) are available
eye_used = eyelink_eye_available();
switch(eye_used) // select eye, add annotation to EDF file
{
case RIGHT_EYE:
    eyemsg_printf("EYE_USED 1 RIGHT");
    break;
case BINOCULAR: // both eye's data present: use left eye only
    eye_used = LEFT_EYE;
case LEFT_EYE:
    eyemsg_printf("EYE_USED 0 LEFT");
    break;
}

while(1)
{
    // get an integer copy of sample: skips any events
    i = eyelink_get_sample(&isample);
    if(i)
    {
        // convert the integrer eye data to float data
        // eyelink_position_prescaler() is used as a divisor
        if (x!=MISSING_DATA)
            x = ((float) isample.gx[eye_used])/
                ((float) eyelink_position_prescaler());
        else
            x = (float) MISSING_DATA;

        if (y!=MISSING_DATA)
            y = ((float) isample.gy[eye_used])/
                ((float) eyelink_position_prescaler());
        else
            y = (float) MISSING_DATA;

        eyemsg_printf("Sample: %ld %6.2f %6.2f", isample.time, x, y);
    }

    ...
    // Code for exiting, display drawing etc.
}
```

Output:

```
MSG     14839670 Sample: 14839666 539.20 372.60
MSG     14839670 Sample: 14839668 539.20 372.60
MSG     14839674 Sample: 14839670 539.20 372.60
MSG     14839674 Sample: 14839672 539.20 372.60
MSG     14839678 Sample: 14839674 547.90 367.60
MSG     14839678 Sample: 14839676 556.60 362.50
MSG     14839682 Sample: 14839678 565.30 357.40
MSG     14839682 Sample: 14839680 574.10 352.30
MSG     14839686 Sample: 14839682 582.80 347.20
MSG     14839686 Sample: 14839684 591.50 342.00
MSG     14839690 Sample: 14839686 600.30 336.80
MSG     14839690 Sample: 14839688 609.00 331.60
MSG     14839694 Sample: 14839690 617.80 326.40
MSG     14839694 Sample: 14839692 626.60 321.20
MSG     14839698 Sample: 14839694 635.30 315.90
MSG     14839698 Sample: 14839696 644.10 310.70
MSG     14839702 Sample: 14839698 652.90 305.40
MSG     14839702 Sample: 14839700 661.70 300.00
MSG     14839706 Sample: 14839702 670.50 294.70
MSG     14839706 Sample: 14839704 679.30 289.40
MSG     14839710 Sample: 14839706 688.10 284.00
MSG     14839710 Sample: 14839708 696.90 278.60
MSG     14839714 Sample: 14839710 705.80 273.20
MSG     14839714 Sample: 14839712 714.60 267.70
MSG     14839718 Sample: 14839714 723.40 262.30
MSG     14839718 Sample: 14839716 732.30 256.80
MSG     14839722 Sample: 14839718 741.20 251.30
MSG     14839722 Sample: 14839720 750.00 245.80
```

See also:
`eyelink_get_float_data()`, `eyelink_get_last_data()`, `eyelink_get_next_data()`, `eyelink_newest_float_sample()` and `eyelink_newest_sample()`

26.16.1.16 INT16 eyelink_last_message (char FARTYPE ∗ *buf*)

Checks for and gets the last packet received, stores the data and the node address sent from.

Parameters:
buf String buffer to return text message.

Returns:
`0` if no message since last command sent, otherwise length of string.

Example: See `eyelink_timed_command()`

See also:
`eyelink_send_command()` and `eyelink_timed_command()`

26.16.1.17 INT16 eyelink_newest_double_sample (void FARTYPE ∗ *buf*)

Check if a new sample has arrived from the link. This is the latest sample, not the oldest sample that is read by `eyelink_get_next_data()`, and is intended to drive gaze cursors and gaze-contingent displays. Typically the function is called with a `NULL` buffer pointer, to test if new data has arrived. If a value of `1` is returned, the function is called with a `DSAMPLE` buffer to get the new sample.

Parameters:
buf Pointer to sample buffer type `DSAMPLE`. If `NULL`, just checks new-sample status.

Returns:
`-1` if no samples, `0` if no new data, `1` if new sample.

See also:
`eyelink_get_float_data()`, `eyelink_get_last_data()`, `eyelink_get_next_data()`, `eyelink_get_sample()` `eyelink_newest_float_sample()` and `eyelink_newest_sample()`

26.16.1.18 INT16 eyelink_newest_float_sample (void FARTYPE ∗ *buf*)

Check if a new sample has arrived from the link. This is the latest sample, not the oldest sample that is read by `eyelink_get_next_data()`, and is intended to drive gaze cursors and gaze-contingent displays. Typically the function is called with a `NULL` buffer pointer, to test if new data has arrived. If a value of `1` is returned, the function is called with a `FSAMPLE` buffer to get the new sample.

Parameters:

buf Pointer to sample buffer type `FSAMPLE`. If `NULL`, just checks new-sample status.

Returns:

`-1` if no samples, `0` if no new data, `1` if new sample.

Example:

```
// This program illustrates the use of eyelink_newest_float_sample
#include <eyelink.h>

ALLF_DATA evt;          // buffer to hold sample and event data
int eye_used = 0;       // indicates which eye's data to display
float x, y;              // gaze position

// Recording with link data enabled
error = start_recording(1,1,1,0);
if(error != 0) return error;   // ERROR: couldn't start recording

// Determines which eye(s) are available
eye_used = eyelink_eye_available();

// Selects eye, add annotation to EDF file
switch(eye_used)
{
    case RIGHT_EYE:
        eyemsg_printf("EYE_USED 1 RIGHT");
        break;
    case BINOCULAR:   // both eye's data present: use left eye only
        eye_used = LEFT_EYE;
    case LEFT_EYE:
        eyemsg_printf("EYE_USED 0 LEFT");
        break;
}
while (1)
{
    // check for new sample update
    if(eyelink_newest_float_sample(NULL)>0)
    {
        // get the sample
        eyelink_newest_float_sample(&evt);

        x = evt.fs.gx[eye_used];
        y = evt.fs.gy[eye_used];

        // make sure pupil is present
        if(x!=MISSING_DATA && y!=MISSING_DATA && evt.fs.pa[eye_used]>0)
            eyemsg_printf("Sample: %ld %8.2f %8.2f",
                evt.fs.time, x, y);
    }
    ...
    // Code for exiting, display drawing etc.
}
```

Output:

```
MSG     23701980 EYE_USED 0 LEFT
MSG     23703426 Sample: 23703424   412.90   217.90
MSG     23703430 Sample: 23703426   433.20   216.20
MSG     23703430 Sample: 23703428   453.40   214.40
MSG     23703434 Sample: 23703430   473.60   212.60
MSG     23703434 Sample: 23703432   493.80   210.80
MSG     23703438 Sample: 23703434   514.00   209.00
MSG     23703438 Sample: 23703436   534.20   207.10
MSG     23703442 Sample: 23703438   554.30   205.20
MSG     23703442 Sample: 23703440   574.40   203.30
MSG     23703446 Sample: 23703442   594.50   201.30
MSG     23703446 Sample: 23703444   614.60   199.30
MSG     23703450 Sample: 23703446   634.70   197.20
MSG     23703450 Sample: 23703448   634.70   197.20
```

See also:

`eyelink_get_float_data()`, `eyelink_get_last_data()`, `eyelink_get_next_data()`, `eyelink_get_sample()` `eyelink_newest_double_sample()` and `eyelink_newest_sample()`

26.16.1.19 INT16 eyelink_newest_sample (void FARTYPE * *buf*)

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

Parameters:

buf Pointer to buffer (`ISAMPLE`, `ALL_DATA` type).

Returns:

`0` if no data, else `SAMPLE_TYPE`.

Example:

```
ISAMPLE isample;
float x, y;              // gaze position

if(eyelink_newest_sample(NULL)>0)  // check for new sample update
{
    eyelink_newest_sample(&isample);

    if (x!=MISSING_DATA)
        x = ((float) isample.gx[eye_used])/((float) eyelink_position_prescaler());    // yes: get
    else
        x = (float) MISSING_DATA;

    if (y!=MISSING_DATA)
        y = ((float) isample.gy[eye_used])/((float) eyelink_position_prescaler());
    else
        y = (float) MISSING_DATA;

    ...
    //  code for processing the sample
    ...
}
```

See also:

`eyelink_get_float_data()`, `eyelink_get_last_data()`, `eyelink_get_next_data()`, `eyelink_get_sample()` and `eyelink_newest_float_sample()`

26.16.1.20 INT16 eyelink_node_receive (ELINKADDR *node*, void FARTYPE ∗ *data*)

Checks for and gets the last packet received, stores the data and the node address sent from.

Remarks:

Data can only be read once, and is overwritten if a new packet arrives before the last packet has been read.

Parameters:

node Address of the sender.

data Pointer to a buffer to hold the data.

Returns:

`0` if successful, otherwise link error.

Example: See `eyelink_node_send()`

See also:

`eyelink_get_float_data()`, `eyelink_open_node()` and `eyelink_node_send()`

26.16.1.21 INT16 eyelink_node_send (ELINKADDR *node*, void FARTYPE ∗ *data*, UINT16 *dsize*)

Sends a given data to the given node.

Parameters:

node `ELINKADDR` node address.

data Pointer to buffer containing data to send.

dsize Number of bytes of data. Maximum `ELREMBUFSIZE` bytes.

Returns:

`0` if successful, otherwise link error.

Example:

```
// The following two code pieces show the exchanges of messaging between two
// remote applications (SENDER and LISTENER).  In the first program, the SENDER sends
// a "START_RECORD" message to the LISTENER application and wait for an "echo" message.
// The second program shows the LISTENER application receives the "START_RECORD" message
// and sends an "echo" message

#include <eyelink.h>

ELINKADDR listener_address; // Address of the listener application
char text_message[256], echo_message[256];

...
// Code for retrieving the listener's address; See COMM_SIMPLE
...

sprintf(text_message, "START_RECORD");
eyelink_node_send(listener_address, text_message, 40);

// Now checks for the echo response from the listener
while (1)
{
    // get the message from the listener application
    k = eyelink_node_receive(listener_address, echo_message);
    if (k > 0 && strstr(echo_message, text_message))
                        break;
    ...
    // Other code for error handling
    ...
}


// This program shows the LISTENER application receives the "START_RECORD" message and sends an "e

#include <eyelink.h>

ELINKADDR sender_address; // Address of the sender application
char text_message[256], echo_message[256];

// get the message from the sender application
k = eyelink_node_receive(sender_address, text_message);

if(k > 0 && !_strnicmp(text_message, "START_RECORD", 12))
{
    eyemsg_printf(text_message);
    error = start_recording(1,1,1,1); // Starts recording
    if(error != 0)
        return error;        // Return error code if failed

    sprintf(echo_message, "ECHO %s", text_message);
    // Sends the echo message to the sender application
    eyelink_node_send(sender_address, echo_message, 60);
}
```

See also:
`eyelink_get_node()`, `eyelink_node_receive()` and `eyelink_open_node()`

26.16.1.22 INT16 eyelink_node_send_message (ELINKADDR *node*, char FARTYPE ∗ *msg*)

Sends a text message the connected eye tracker. The text will be added to the EDF file.

Remarks:
NEW (v2.1): If the link is initialized but not connected to a tracker, the message will be sent to the tracker set by `set_eyelink_address()`.

Parameters:
msg Text to send to the tracker.

node Address of the tracker.

Returns:
`0` if no error, else link error code.

Example: See `eyelink_quiet_mode()`

See also:
`eyelink_send_message()`, `eyelink_quiet_mode()` and `set_eyelink_address()`

26.16.1.23 INT16 eyelink_node_send_message_ex (UINT32 *exectime*, ELINKADDR *node*, char FARTYPE ∗ *msg*)

Send a data file message to any or all trackers with time offset.

Parameters:

exectime time offset. If the offset is 0, this function is the same as eyelink_node_send_message()

node Node address

msg Message to be sent

Returns:

`OK_RESULT` or `LINK_TERMINATED_RESULT`.

See also:

eyelink_node_send_message()

26.16.1.24 INT16 eyelink_position_prescaler (void)

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

Returns:

Divisor (usually 10).

Example:

```
// This program uses the eyecmd_printf() and eyelink_position_prescaler() to set
// and retrieve the screen_write_prescale value.  See eyelink_get_sample() for
// another example of using eyelink_position_prescaler()

#include <eyelink.h>
UINT32 start_time;

// Sets the value by which gaze position data is multiplied before writing to EDF file or link as
eyecmd_printf("screen_write_prescale = 10");

start_time = current_msec();
while(current_msec() < start + 1000)
    if (eyelink_command_result() == OK_RESULT)
    {
        // Checks out the value set
        eyemsg_printf("Position scalar %d",
            eyelink_position_prescaler());
        break;
    }
```

26.16.1.25 INT16 eyelink_read_reply (char FARTYPE ∗ *buf*)

Returns text with reply to last read request.

Parameters:

buf String to contain text.

Returns:

`OK_RESULT (0)` if response received.
`NO_REPLY` if no response yet.

Example: See eyelink_read_request()

See also:

eyelink_read_request()

26.16.1.26 INT16 eyelink_read_request (char FARTYPE ∗ *text*)

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

Remarks:

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

Parameters:

text String with message to send.

Returns:

0 if success, otherwise link error code.

Example:

```
// This code illustrates the use of eyelink_read_request() to get the coordinate information of th

int read_tracker_pixel_coords(void)
{
    char buf[100] = "";
    UINT32 t;

    // Checks whether we are still connected
    if(!eyelink_is_connected() || break_pressed())
        return 1;

    // Tries to retrieve the info about screen_pixel_coords
    eyelink_read_request("screen_pixel_coords");

    t = current_msec();
    // Waits for a maximum of 500 msec
    while(current_msec()-t < 500)
    {
        if(eyelink_read_reply(buf) == OK_RESULT)
        {
            eyemsg_printf("%s", buf);
            return 0;
        }
        message_pump(NULL);  // Keeps Windows happy
        if(!eyelink_is_connected) return 1;
        if(break_pressed()) return 1;
    }
    return -1;  // Timed out
}
```

Output:

```
MSG     374986 0.000000,0.000000,1023.000000,767.000000
```

See also:

eyelink_read_reply()

26.16.1.27 INT16 eyelink_reset_data (INT16 *clear*)

Prepares link buffers to receive new data. If `<clear>` is nonzero, removes old data from buffer.

Parameters:

clear If clear is non-zero, any buffer data is discarded.

Returns:

Always returns `0`.

Example: See `eyelink_in_data_block()`

See also:

`eyelink_data_status()` and `eyelink_in_data_block()`

26.16.1.28 UINT16 eyelink_sample_data_flags (void)

Gets sample data content flag (`0` if not in sample block).

26.16.1.29 INT16 eyelink_send_command (char FARTYPE ∗ *text*)

Sends a command to the connected eye tracker.

Remarks:

If `eyelink_send_commnd()` is used, the text command will be executed, and a result code returned that can be read with `eyelink_command_result()`.

Parameters:

text String command to send.

Returns:

`0` if successful, otherwise link error.

Example:

```
// This program illustrates the use of eyelink_send_command

#include <eyelink.h>

UINT32 t;
int i, j;
char buf[256];

// Change the assignment of the button functions so that pressing
// button 1 will accept target fixation
eyelink_send_command("button_function 1 'accept_target_fixation'");

t = current_msec();
// Waits for a maximum of 1000 msec
while(current_msec()-t < 1000)
{
    // Checks for result from command execution
    i = eyelink_command_result();
    // Used to get more information on tracker result
```

```
            j = eyelink_last_message(buf);

            if (i == OK_RESULT)
            {
                eyemsg_printf("Command executed successfully: %s", j?buf:"");
                break;
            }
            else if (i!=NO_REPLY)
            {
                eyemsg_printf("Error in executing command: %s", j?buf:"");
                break;
            }
        }
```

Output:

```
MSG     4608038 Command executed successfully: OK
```

See also:
`eyelink_command_result()` and `eyelink_timed_command()`

26.16.1.30 INT16 eyelink_send_message (char FARTYPE ∗ *msg*)

Sends a text message the connected eye tracker. The text will be added to the EDF file.

Parameters:
msg Text to send to the tracker.

Returns:
`0` if no error, else link error code.

Example: See `eyelink_quiet_mode()`

See also:
`eyelink_node_send_message()`, `eyelink_quiet_mode()` and `set_eyelink_-address()`

26.16.1.31 INT16 eyelink_send_message_ex (UINT32 *exectime*, char FARTYPE ∗ *msg*)

Send a data file message string to connected tracker with time offset.

Parameters:
exectime time offset. If the offset is 0, this function is the same as eyelink_send_message()
msg Message to be sent

Returns:
`OK_RESULT` or `LINK_TERMINATED_RESULT`.

See also:
eyemsg_printf_ex(), eyelink_send_message()

26.16.1.32 INT16 eyelink_timed_command (UINT32 *msec*, char FARTYPE ∗ *text*)

Sends a command to the connected eye tracker, wait for reply.

Remarks:

If `eyelink_send_command()` is used, the text command will be executed, and a result code returned that can be read with `eyelink_command_result()`.

Parameters:

text String command to send.

msec Maximum milliseconds to wait for reply.

Returns:

`OK_RESULT (0)` if OK.
`NO_REPLY` if timed out.
`LINK_TERMINATED_RESULT` if cannot send.
other error codes represent tracker execution error.

Example:

```
// This program illustrates the use of eyelink_timed_command

#include <eyelink.h>

INT16 error;
char buf[256];

// send command string to tracker, wait for reply
error = eyelink_timed_command(1000, "button_function 5
     'accept_target_fixation'");

// Checks for the command result
if (error)
{
    eyelink_last_message(buf);
    eyemsg_printf("Error in excuting the command: %s", buf);
}
```

See also:

`eyelink_command_result()` and `eyelink_send_command()`

26.16.1.33 INT16 eyelink_wait_for_data (UINT32 *maxwait*, INT16 *samples*, INT16 *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.

Returns:

`1` if data is available. `0` if timed out.

Example:

```
// This program illustrates the use of eyelink_wait_for_data()

#include <eyelink.h>
#include <stdio.h>
int error;

// Recording with link data enabled
error = start_recording(1,1,1,0);
if(error != 0) return error;   // ERROR: couldn't start recording

// Wait for link sample data
if(!eyelink_wait_for_data(100, 1, 0))
{
    printf("ERROR: No link samples received!");
    return -1;
}
```

See also:

`eyelink_wait_for_block_start()`

26.16.1.34 INT16 eyemsg_printf (const char ∗ *fmt*, ...)

This sends a text message to the EyeLink tracker, which timestamps it and writes it to the EDF data file. Messages are useful for recording trial conditions, subject responses, or the time of important events. This function is used with the same formatting methods as printf(), allowing numbers to be included. Avoid end-of-line characters ("\n") at end of messages.

Parameters:

fmt Similar to printf(), format string plus arguments.

Returns:

`0` if successfully sent to tracker, else error code.

Example:

```
// This program illustrates the use of eyemsg_printf()

#include <eyelink.h>
char program_name[100] = "Windows Sample Experiment 2.0";

// Add a message to the EDF file
eyemsg_printf("RECORDED BY %s", program_name);
```

Output:

```
MSG     2248248 RECORDED BY Windows Sample Experiment 2.0
```

See also:

`eyecmd_printf()`

26.16.1.35 INT16 eyemsg_printf_ex (UINT32 *exectime*, const char ∗ *fmt*, ...)

This allows us to send messages to the Eyelink tracker to be logged into the data file with a time offset. Use it just like printf() to format the message text.

Parameters:

exectime time offset that reflects in the message's time stamp

fmt printf format string

Returns:

`0` if OK, else error code.

example:

```
if(open_eyelink_connection(0) !=0)
        return 0;
open_data_file("msgtest.edf");

eyemsg_printf("hello");
msec_delay(100);
eyemsg_printf_ex(-100,"hello1");
msec_delay(100);

eyemsg_printf_ex(100,"hello2");
msec_delay(100);
eyemsg_printf("hello3");
msec_delay(100);
close_data_file();
receive_data_file("",".",1);
```

As you can see in the edf file data generated by the above code, both Message1 and Message2 has the same time stamp and message3 and message4 has the same time stamp.

```
MSG     8004932 Message1
MSG     8004932 Message2
MSG     8005232 Message3
MSG     8005232 Message4
```

26.17 Message Pump functions

Functions

- INT16 message_pump ()
- INT16 key_message_pump (void)
- void pump_delay (UINT32 delay)

26.17.1 Function Documentation

26.17.1.1 INT16 key_message_pump (void)

Similar to `message_pump()`, but only processes keypresses. This may help reduce latency.

26.17.1.2 INT16 message_pump ()

Almost all experiments must run in a deterministic fashion, executing sequentially and in loops instead of the traditional Windows event-processing model. However, Windows messages must still be dispatched for keyboard input and other events. Calling `getkey()` will dispatch messages and return keys. The `message_pump()` function also dispatches messages, but does not read the keys. It can also handle messages for a modeless dialog box.

Returns:

`0` normally, `1` if ALT-F4 or CTRL-C pressed, or if `terminal_break()` called. Any loops should exit in this case.

Example: The following two programs works together to show the use of `message_pump()` function. In this case, writes a message to EDF file when the left mouse button is pressed.

```
// Program 1.  The typical trial loop plus the message_pump()
#include <eyelink.h>

// Trial loop: till timeout or response
while(1)
{
    ...
    // Other code for display update, trial terminating, etc.

    // Allows messages to operate in loops
    message_pump(NULL);
}


// Program 2.  Revised code in the full_screen_window_proc() function of the w32_demo_window.c
// module.  In this case, a WM_LBUTTONDOWN message is recorded in the EDF file.

#include <eyelink.h>
switch (message)
{
case WM_KEYDOWN:
case WM_CHAR:
    // Processes key messages: these can be accessed by getkey()
    process_key_messages(hwnd, message, wparam, lparam);
    break;

case WM_LBUTTONDOWN:
    eyemsg_printf("Left button is down");
```

```
            break;
            ...
            // Other windows messages and events
    }
```

Output:

```
    MSG     11661891 SYNCTIME 1
    MSG     11662745 left button is down
    MSG     11663048 left button is down
    BUTTON  11665520        4       1
    MSG     11665521 ENDBUTTON 4
```

See also:
`pump_delay()`

26.17.1.3 void pump_delay (UINT32 *delay*)

During calls to `msec_delay()`, 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 `message_pump()` 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 Number of milliseconds to delay.

Example:

```
    // This program illustrates the use of pump_delay() at the end of trial
    #include <eyelink.h>

    // End recording: adds 100 msec of data to catch final events
    static void end_trial(void)
    {
        ...
        // Add code here to clean the display

        // Ensure we release realtime lock
        end_realtime_mode();
        // Delay for 100 msec, allow Windows to clean up
        pump_delay(100);

        // halt recording, return when tracker finished mode switch
        stop_recording();
    }
```

See also:
`msec_delay()` and `message_pump()`

26.18 Tracker Mode functions

Defines

- #define OK_RESULT 0
- #define NO_REPLY 1000
- #define LINK_TERMINATED_RESULT -100
- #define ABORT_RESULT 27
- #define UNEXPECTED_EOL_RESULT -1
- #define SYNTAX_ERROR_RESULT -2
- #define BAD_VALUE_RESULT -3
- #define EXTRA_CHARACTERS_RESULT -4
- #define IN_DISCONNECT_MODE 16384
- #define IN_UNKNOWN_MODE 0
- #define IN_IDLE_MODE 1
- #define IN_SETUP_MODE 2
- #define IN_RECORD_MODE 4
- #define IN_TARGET_MODE 8
- #define IN_DRIFTCORR_MODE 16
- #define IN_IMAGE_MODE 32
- #define IN_USER_MENU 64
- #define IN_PLAYBACK_MODE 256

Functions

- INT16 eyelink_current_mode (void)
- INT16 eyelink_tracker_mode (void)
- INT16 eyelink_wait_for_mode_ready (UINT32 maxwait)
- INT16 eyelink_user_menu_selection (void)
- void set_offline_mode (void)

26.18.1 Define Documentation

26.18.1.1 #define ABORT_RESULT 27

EyeLink TRACKER RETURN CODES: operation aborted (calibration)

26.18.1.2 #define BAD_VALUE_RESULT -3

COMMAND PARSE ERRORS: value is not right for command or tracker state

26.18.1.3 #define EXTRA_CHARACTERS_RESULT -4

COMMAND PARSE ERRORS: bad format or too many values

26.18.1.4 #define IN_DISCONNECT_MODE 16384

Tracker state bit: disconnected.

26.18.1.5 #define IN_DRIFTCORR_MODE 16

Tracker state bit: drift correction

26.18.1.6 #define IN_IDLE_MODE 1

Tracker state bit: off-line

26.18.1.7 #define IN_IMAGE_MODE 32

Tracker state bit: image-display mode

26.18.1.8 #define IN_PLAYBACK_MODE 256

Tracker state bit: tracker sending playback data

26.18.1.9 #define IN_RECORD_MODE 4

Tracker state bit: data flowing

26.18.1.10 #define IN_SETUP_MODE 2

Tracker state bit: setup or cal/val/dcorr

26.18.1.11 #define IN_TARGET_MODE 8

Tracker state bit: some mode that needs fixation targets

26.18.1.12 #define IN_UNKNOWN_MODE 0

Tracker state bit: mode fits no class (i.e setup menu)

26.18.1.13 #define IN_USER_MENU 64

Tracker state bit: user menu

26.18.1.14 #define LINK_TERMINATED_RESULT -100

LINK RETURN CODES: can't send or link closed

26.18.1.15 #define NO_REPLY 1000

LINK RETURN CODES: no reply yet (for polling test)

26.18.1.16 #define OK_RESULT 0

LINK RETURN CODES: OK

26.18.1.17 #define SYNTAX_ERROR_RESULT -2

COMMAND PARSE ERRORS: unknown command, unknown variable etc.

26.18.1.18 #define UNEXPECTED_EOL_RESULT -1

COMMAND PARSE ERRORS: not enough data

26.18.2 Function Documentation

26.18.2.1 INT16 eyelink_current_mode (void)

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 EXPTSPPT toolkit is `IN_USER_MENU` to test if the EyeLink Abort menu has been activated.

Returns:
Set of bitflags 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.

Example: See `echo_key()`

See also:
`echo_key()`, `eyelink_tracker_mode()` and `eyelink_user_menu_selection()`

26.18.2.2 INT16 eyelink_tracker_mode (void)

Returns raw EyeLink mode numbers, defined in eyelink.h as EL_xxxx definitions.

Returns:
Raw EyeLink mode, `-1` if link disconnected

Example:

```
// This programs illustrates the use of track_mode_loop() function. See BROADCAST for the complete

#include <eyelink.h>

// Follow and process tracker modes
// Displays calibration and drift correction targets
// Also detects start of recording
// Black backgrounds would be transparent as video overlay
void track_mode_loop(void)
```

```
{
    int oldmode = -1;

    while(eyelink_is_connected())
    {
        int mode = eyelink_tracker_mode();
        unsigned key = getkey();

        if(key==27 || break_pressed() || !eyelink_is_connected())
            return;
        else if(key) // Echo to tracker
            eyelink_send_keybutton(key,0,KB_PRESS);

        if(mode == oldmode)
            continue;
        switch(mode)
        {
        case EL_RECORD_MODE:     // Recording mode
            // Code for processing recording mode ;
            break;
        case EL_IMAGE_MODE:      // Image mode
            // Code for processing image mode ;
            break;
        case EL_SETUP_MENU_MODE: // Setup menu mode
            // Code for processing setup menu mode ;
            break;
        case EL_CALIBRATE_MODE:  // Calibration, validation, DC mode
        case EL_VALIDATE_MODE:
        case EL_DRIFT_CORR_MODE:
            // Code for processing calibration, validation, dc mode ;
            break;
        case EL_OPTIONS_MENU_MODE: // Option menu mode
            // Code for processing calibration mode ;
            break;
        default:
            // Code for processing default case ;
            break;
        }
        oldmode = mode;
    }
}
```

See also:
`eyelink_current_mode()` and `eyelink_is_connected()`

26.18.2.3 INT16 eyelink_user_menu_selection (void)

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

Returns:
0 if no selection made since last call, else code of selection.

Example:

```
// This programs illustrates the use of eyelink_user_menu_selection

#include <eyelink.h>
int i;

i = eyelink_current_mode();
if(i & IN_USER_MENU)     // handle user menu selections
{
    switch(eyelink_user_menu_selection()
    {
    case 1:     // SETUP selected
        break;
    case 2:     // REPEAT trial
        return REPEAT_TRIAL;
    case 3:     // SKIP trial
        return SKIP_TRIAL;
    case 4:     // Abort experiment
        eyelink_abort();// this cleans up by erasing menu
        return ABORT_EXPT;
    default:    // no selection: continue
        break;
    }
}
```

See also:
 eyelink_current_mode()

26.18.2.4 INT16 eyelink_wait_for_mode_ready (UINT32 *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.

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

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

Returns:
 0 if mode switching is done, else still waiting.

Example: See eyelink_data_start()

See also:
 eyelink_data_start() and set_offline_mode()

26.18.2.5 void set_offline_mode (void)

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

```
// This program illustrates the use of set_offline_mode() function when doing cleaning up
// at the end of data recoridng

// Checks whether we still have the connection to the tracker
if(eyelink_is_connected())
{
    // Places EyeLink tracker in off-line (idle) mode
    set_offline_mode();
    eyecmd_printf("close_data_file");    // close data file
    eyelink_close(1);         // disconnect from tracker
}

// shut down system (MUST do before exiting)
close_eyelink_system();
```

See also:

`eyelink_abort()`

26.19 Eyelink Button Functions

Functions

- UINT16 eyelink_read_keybutton (INT16 FARTYPE ∗mods, INT16 FARTYPE ∗state, UINT16 ∗kcode, UINT32 FARTYPE ∗time)
- INT16 eyelink_send_keybutton (UINT16 code, UINT16 mods, INT16 state)
- UINT16 eyelink_button_states (void)
- UINT16 eyelink_last_button_states (UINT32 FARTYPE ∗time)
- UINT16 eyelink_last_button_press (UINT32 FARTYPE ∗time)
- INT16 eyelink_flush_keybuttons (INT16 enable_buttons)

26.19.1 Function Documentation

26.19.1.1 UINT16 eyelink_button_states (void)

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.

Remarks:
Buttons above 8 are not realized on the EyeLink tracker.

Returns:
Flag bits for buttons currently pressed.

Example:

```
// This program illustrates the use of eyelink_button_states

#include <eyelink.h>
int   state =0;
int     prev_state = 0;
UINT32 start_time = current_time();

// Exits when the tracker is not connected or times out
while(eyelink_is_connected()
      && current_time() > start_time + 5000)
{
    // reads the currently-known state of all buttons
    state = eyelink_button_states();
    if (state != prev_state)
    {
        eyemsg_printf("Button 1:%s 2:%s 3:%s 4:%s 5:%s",
            state & 0x01 ? "Pressed" : "Released",
            (state & 0x02) >> 1 ? "Pressed" : "Released",
            (state & 0x04) >> 2 ? "Pressed" : "Released",
            (state & 0x08) >> 3 ? "Pressed" : "Released",
            (state & 0x10) >> 4 ? "Pressed" : "Released");

       prev_state = state;
    }
}
```

Output:

```
BUTTON  4144034 1       1
MSG     4144035 Button 1:Pressed 2:Released 3:Released 4:Released 5:Released
BUTTON  4144266 1       0
MSG     4144267 Button 1:Released 2:Released 3:Released 4:Released 5:Released
BUTTON  4144650 2       1
MSG     4144651 Button 1:Released 2:Pressed 3:Released 4:Released 5:Released
BUTTON  4144898 2       0
MSG     4144899 Button 1:Released 2:Released 3:Released 4:Released 5:Released
BUTTON  4145260 3       1
MSG     4145261 Button 1:Released 2:Released 3:Pressed 4:Released 5:Released
BUTTON  4145492 3       0
MSG     4145493 Button 1:Released 2:Released 3:Released 4:Released 5:Released
BUTTON  4145834 4       1
MSG     4145835 Button 1:Released 2:Released 3:Released 4:Pressed 5:Released
BUTTON  4146106 4       0
MSG     4146107 Button 1:Released 2:Released 3:Released 4:Released 5:Released
BUTTON  4146498 5       1
MSG     4146499 Button 1:Released 2:Released 3:Released 4:Released 5:Pressed
BUTTON  4146778 5       0
MSG     4146779 Button 1:Released 2:Released 3:Released 4:Released 5:Released
```

See also:
`eyelink_last_button_press()`

26.19.1.2 INT16 eyelink_flush_keybuttons (INT16 *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:

enable_buttons Set to `0` to monitor last button press only, `1` to queue button events.

Returns:

Always `0`.

Example:

```
// This program illustrates the use of eyelink_flush_keybuttons.

UINT32 wait_time = 5000;
int i;

// Flushes any waiting keys or buttons
eyelink_flush_keybuttons(0);

// Sets the time-out duration
wait_time += current_msec();

// Makes sure that the tracker connection is still alive
while(eyelink_is_connected())
{
    // handle the error or abort situations
    if(getkey()==27 || !eyelink_is_connected())
        break;

    // Checks the button response
    i = eyelink_last_button_press(NULL);
    if(i)
```

```
            {
                eyemsg_printf("WAITBUTTON %d", i);
                break;
            }

            // Times out if no button is pressed
            if(current_time() > wait_time)
            {
                eyemsg_printf("WAITBUTTON TIMEOUT");
                break;
            }
        }
```

Output:

```
BUTTON  19585661        5       1
MSG     19585662 WAITBUTTON 5
BUTTON  19586005        5       0
```

See also:

`eyelink_button_states()`, `eyelink_last_button_press()`, `eyelink_read_keybutton()` and `eyelink_send_keybutton()`

26.19.1.3 UINT16 eyelink_last_button_press (UINT32 FARTYPE * *time*)

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:

time Far pointer to a variable to hold tracker time of last button press. Usually left as `NULL` to ignore time.

Returns:

Button last pressed, `0` if no button pressed since last read, or call to `eyelink_flush_keybuttons()`.

Example:

```
// This program illustrates the use of eyelink_flush_keybuttons

#include <eyelink.h>
int   button;

// reset keys and buttons from tracker
eyelink_flush_keybuttons(0);

while(1)
{
     // Check for eye-tracker buttons pressed
     button = eyelink_last_button_press(NULL);

     // Disables button 6 and 7 and process all other button events
     if(button != 0 && button != 6 && button != 7)
     {
         // Presses button 5 to break the loop
         if (button == 5)
```

```
                {
                    eyemsg_printf("ENDBUTTON %d", button);
                    break;
                }
                // Records all other button press messages
                else
                    eyemsg_printf("BUTTON PRESSED %d", button);
            }
        }
```

Output:

```
BUTTON  19753748        2       1
MSG     19753749 BUTTON PRESSED 2
BUTTON  19754018        2       0
BUTTON  19755595        5       1
MSG     19755595 ENDBUTTON 5
BUTTON  19755808        5       0
```

See also:
`eyelink_flush_keybuttons()`, `eyelink_button_states()`, `eyelink_read_keybutton()` and `eyelink_send_keybutton()`

26.19.1.4 UINT16 eyelink_last_button_states (UINT32 FARTYPE ∗ *time*)

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. Same as eyelink_button_states() except, optionally time of the button states can be acquired.

Parameters:
→ ***time*** pointer to return time of the button states.

Returns:
Flag bits for buttons currently pressed.

See also:
`eyelink_send_keybutton()`

26.19.1.5 UINT16 eyelink_read_keybutton (INT16 FARTYPE ∗ *mods*, INT16 FARTYPE ∗ *state*, UINT16 ∗ *kcode*, UINT32 FARTYPE ∗ *time*)

Reads any queued key or button events from tracker.

Remarks:
Any of the parameters(mods/state/kcode/time) can be null to ignore.

Parameters:
mods Pointer to variable to hold button number or key modifier (Shift, Alt and Ctrl key states).

state Pointer to variable to hold key or button change (`KB_PRESS`, `KB_RELEASE`, or `KB_REPEAT`).

kcode Pointer to variable to hold key scan code.

time Pointer to a variable to hold tracker time of the key or button change.

Returns:

Key character is key press/release/repeat, `KB_BUTTON (0xFF00)` if button press or release. `0` if none.

Example:

```
// This program illustrates the use of eyelink_read_keybutton to read key press events from the tr

#include <eyelink.h>

UINT16  key;
INT16   state;

// Reads any queued key or button events from the tracker keyboard
key = eyelink_read_keybutton(NULL, &state, NULL, NULL);

// Makes sure that we checks only the key press
if (key && state == KB_PRESS && key != KB_BUTTON)
{
    // Writes out the pressed key id
    if(key < 256 && isprint(key))
        eyemsg_printf("KEY '%c'", key);
    else
        eyemsg_printf("WAITKEY 0x%04X", key);
}
```

See also:

`eyelink_send_keybutton()`

26.19.1.6 INT16 eyelink_send_keybutton (UINT16 *code*, UINT16 *mods*, INT16 *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`).

Returns:

`0` if OK, else send link error.

Example:

```
// This program illustrates the implementation of echo_key() function using the eyelink_send_keybu

// ECHO_KEY() function is similar to getkey() but also echoes key to tracker
UINT16 echo_key(void)
{
    UINT16 k = getkey();

    if(k!=0 && k!=1)
        eyelink_send_keybutton(k, 0, KB_PRESS);
    return k;
}
```

See also:

`eyelink_read_keybutton()`

26.20 all camera display related functions for SDL Core Graphics Example

Functions

- INT16 setup_image_display (INT16 width, INT16 height)
- void exit_image_display (void)
- void image_title (INT16 threshold, char ∗title)
- void set_image_palette (INT16 ncolors, byte r[130], byte g[130], byte b[130])
- void draw_image_line (INT16 width, INT16 line, INT16 totlines, byte ∗pixels)
- void drawLine (CrossHairInfo ∗chi, int x1, int y1, int x2, int y2, int cindex)
- void drawLozenge (CrossHairInfo ∗chi, int x, int y, int width, int height, int cindex)
- void getMouseState (CrossHairInfo ∗chi, int ∗rx, int ∗ry, int ∗rstate)

26.20.1 Function Documentation

26.20.1.1 void draw_image_line (INT16 *width*, INT16 *line*, INT16 *totlines*, byte ∗ *pixels*)

This function is called to supply the image line by line from top to bottom.

Parameters:

width width of the picture. Essentially, number of bytes in `pixels`.

line current line of the image

totlines total number of lines in the image. This will always equal the height of the image.

pixels pixel data.

Eg. Say we want to extract pixel at position (20,20) and print it out as rgb values.

```
if(line == 20) // y = 20
    {
            byte pix = pixels[19];
            // Note the r,g,b arrays come from the call to set_image_palette
            printf("RGB %d %d %d\n",r[pix],g[pix],b[pix]);
    }
```

Remarks:

certain display draw the image up side down. eg. GDI.

26.20.1.2 void drawLine (CrossHairInfo ∗ *chi*, int *x1*, int *y1*, int *x2*, int *y2*, int *cindex*)

draws a line from (x1,y1) to (x2,y2) - required for all tracker versions.

26.20.1.3 void drawLozenge (CrossHairInfo ∗ *chi*, int *x*, int *y*, int *width*, int *height*, int *cindex*)

draws shap that has semi-circle on either side and connected by lines. Bounded by x,y,width,height. x,y may be negative.

Remarks:

This is only needed for EL1000.

26.20.1.4 void exit_image_display (void)

This is called to notify that all camera setup things are complete. Any resources that are allocated in setup_image_display can be released in this function.

26.20.1.5 void getMouseState (CrossHairInfo ∗ *chi*, int ∗ *rx*, int ∗ *ry*, int ∗ *rstate*)

Returns the current mouse position and its state.

Remarks:
This is only needed for EL1000.

26.20.1.6 void image_title (INT16 *threshold*, char ∗ *title*)

This function is called to update any image title change.

Parameters:
threshold if -1 the entire tile is in the title string otherwise, the threshold of the current image.

title if threshold is -1, the title contains the whole title for the image. Otherwise only the camera name is given.

26.20.1.7 void set_image_palette (INT16 *ncolors*, byte *r*[130], byte *g*[130], byte *b*[130])

This function is called after setup_image_display and before the first call to draw_image_line. This is responsible to setup the palettes to display the camera image.

Parameters:
ncolors number of colors in the palette.

r red component of rgb.

g blue component of rgb.

b green component of rgb.

26.20.1.8 INT16 setup_image_display (INT16 *width*, INT16 *height*)

This function is responsible for initializing any resources that are required for camera setup.

Parameters:
width width of the source image to expect.

height height of the source image to expect.

Returns:
-1 if failed, 0 otherwise.

26.21 miscellaneous functions

Functions

- void get_display_information (DISPLAYINFO ∗di)
- INT16 sdl_init_expt_graphics ()
- void sdl_close_expt_graphics ()
- INT16 get_input_key (InputEvent ∗key_input)
- int writeImage (char ∗outfilename, IMAGETYPE format, EYEBITMAP ∗bitmap)

26.21.1 Function Documentation

26.21.1.1 void get_display_information (DISPLAYINFO ∗ *di*)

This is an optional function to get information on video driver and current mode use this to determine if in proper mode for experiment.

Parameters:

→ ***di*** A valid pointer to DISPLAYINFO is passed in to return values.

Remarks:

The prototype of this function can be changed to match one's need or if it is not necessary, one can choose not to implement this function also.

26.21.1.2 INT16 get_input_key (InputEvent ∗ *key_input*)

This is called to check for keyboard input. In this function:

- check if there are any input events
- if there are input events, fill key_input and return 1. otherwise return 0. If 1 is returned this will be called again to check for more events.

Parameters:

→ ***key_input*** fill in the InputEvent structure to return key,modifier values.

Returns:

if there is a key, return 1 otherwise return 0.

Remarks:

Special keys and modifiers should match the following code Special keys:

```
#define F1_KEY    0x3B00
#define F2_KEY    0x3C00
#define F3_KEY    0x3D00
#define F4_KEY    0x3E00
#define F5_KEY    0x3F00
#define F6_KEY    0x4000
#define F7_KEY    0x4100
#define F8_KEY    0x4200
#define F9_KEY    0x4300
#define F10_KEY   0x4400

#define PAGE_UP    0x4900
#define PAGE_DOWN  0x5100
```

```
        #define CURS_UP    0x4800
        #define CURS_DOWN  0x5000
        #define CURS_LEFT  0x4B00
        #define CURS_RIGHT 0x4D00

        #define ESC_KEY   0x001B
        #define ENTER_KEY 0x000D
```

Modifier: If you are using SDL you do not need to modify the modifier value as they match the value.

```
#define ELKMOD_NONE   0x0000
#define ELKMOD_LSHIFT 0x0001
#define ELKMOD_RSHIFT 0x0002
#define ELKMOD_LCTRL  0x0040
#define ELKMOD_RCTRL  0x0080
#define ELKMOD_LALT   0x0100
#define ELKMOD_RALT   0x0200
#define ELKMOD_LMETA  0x0400
#define ELKMOD_RMETA  0x0800,
#define ELKMOD_NUM    0x1000
#define ELKMOD_CAPS   0x2000
#define ELKMOD_MODE   0x4000
```

26.21.1.3 void sdl_close_expt_graphics ()

This is an optional function to properly close and release any resources that are not required beyond calibration needs.

Remarks:
the prototype of this function can be modified to suit ones need.

26.21.1.4 INT16 sdl_init_expt_graphics ()

This is an optional function to initialze graphics and calibration system. Although, this is optional, one should do the innerds of this function elsewhere in a proper manner.

Remarks:
The prototype of this function can be modified to suit ones needs. Eg. The init_expt_graphics of eyelink_core_graphics.dll takes in 2 parameters.

26.21.1.5 int writeImage (char ∗ *outfilename*, IMAGETYPE *format*, EYEBITMAP ∗ *bitmap*)

This function provides support to writing images to disk. Upon calls to el_bitmap_save_and_backdrop or el_bitmap_save this function is requested to do the write operaiton in the preferred format.

Parameters:
← ***outfilename*** Name of the file to be saved.

← ***format*** format to be saved as.

← ***bitmap*** bitmap data to be saved.

Returns:
if successful, return 0.

26.22 all functions related to calibration presentation for SDL Core Graphics Example

Defines

- #define TARGET_SIZE 20
- #define **CAL_TARG_BEEP** 1
- #define **CAL_GOOD_BEEP** 0
- #define **CAL_ERR_BEEP** -1
- #define **DC_TARG_BEEP** 3
- #define **DC_GOOD_BEEP** 2
- #define **DC_ERR_BEEP** -2

Functions

- INT16 setup_cal_display (void)
- void exit_cal_display (void)
- void draw_cal_target (INT16 x, INT16 y)
- void erase_cal_target (void)
- void clear_cal_display (void)
- void cal_sound (INT16 sound)
- void cal_target_beep (void)
- void cal_done_beep (INT16 error)
- void dc_target_beep (void)
- void dc_done_beep (INT16 error)

26.22.1 Define Documentation

26.22.1.1 #define TARGET_SIZE 20

26.22.2 Function Documentation

26.22.2.1 void cal_done_beep (INT16 *error*)

This function is called to signal end of calibration.

Parameters:
error if non zero, then the calibration has error.

26.22.2.2 void cal_sound (INT16 *sound*)

In most cases on can implement all four (cal_target_beep, cal_done_beep,dc_target_beep,dc_done_beep) beep callbacks using just one function.

This function is responsible for selecting and playing the audio clip.

Parameters:
sound sound id to play.

26.22.2.3 void cal_target_beep (void)

This function is called to signal new target.

26.22.2.4 void clear_cal_display (void)

Called to clear the calibration display.

26.22.2.5 void dc_done_beep (INT16 *error*)

This function is called to singnal the end of drift correct.

Parameters:

error if non zero, then the drift correction failed.

26.22.2.6 void dc_target_beep (void)

This function is called to signal a new drift correct target.

26.22.2.7 void draw_cal_target (INT16 *x*, INT16 *y*)

This function is responsible for the drawing of the target for calibration,validation and drift correct at the given coordinate.

Parameters:

x x coordinate of the target.

y y coordinate of the target.

Remarks:

The x and y are relative to what is sent to the tracker for the command screen_pixel_coords.

26.22.2.8 void erase_cal_target (void)

This function is responsible for erasing the target that was drawn by the last call to draw_cal_target.

26.22.2.9 void exit_cal_display (void)

This is called to release any resources that are not required beyond calibration. Beyond this call, no calibration functions will be called.

26.22.2.10 INT16 setup_cal_display (void)

Setup the calibration display. This function called before any calibration routines are called.

26.23 SDL Graphics Functions

Data Structures

- struct **_CCDBS**

Defines

- #define **SDLRGB**(x, y) SDL_MapRGB(x → format,(y).r,(y).g,(y).b)
- #define **SCREEN_LEFT** dispinfo.left
- #define **SCREEN_TOP** dispinfo.top
- #define **SCREEN_RIGHT** dispinfo.right
- #define **SCREEN_BOTTOM** dispinfo.bottom
- #define **SCRHEIGHT** dispinfo.height
- #define **SCRWIDTH** dispinfo.width
- #define bitmap_save_and_backdrop sdl_bitmap_save_and_backdrop
- #define bitmap_to_backdrop sdl_bitmap_to_backdrop
- #define bitmap_save sdl_bitmap_save
- #define **Flip**(x) while(SDL_Flip(x)<0)
- #define **EXTERNAL_DEV_NONE** ((getExButtonStates)0)
- #define **EXTERNAL_DEV_CEDRUS** ((getExButtonStates)1)
- #define **EXTERNAL_DEV_SYS_KEYBOARD** ((getExButtonStates)2)

Typedefs

- typedef _CCDBS **CCDBS**
- typedef int(∗ **getExButtonStates**)(CCDBS ∗)

Functions

- void set_calibration_colors (SDL_Color ∗fg, SDL_Color ∗bg)
- void set_target_size (UINT16 diameter, UINT16 holesize)
- void set_cal_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- void set_dcorr_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- INT16 set_camera_image_position (INT16 left, INT16 top, INT16 right, INT16 bottom)
- void get_display_information (DISPLAYINFO ∗di)
- INT16 init_expt_graphics (SDL_Surface ∗hwnd, DISPLAYINFO ∗info)
- void close_expt_graphics (void)
- int sdl_bitmap_save_and_backdrop (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 bx_options)
- int sdl_bitmap_to_backdrop (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 bx_options)
- int sdl_bitmap_save (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- void set_cal_target_surface (SDL_Surface ∗surface)
- void set_cal_background_surface (SDL_Surface ∗surface)
- void reset_background_surface ()
- void **disable_custombackground_on_imagemode** ()
- int set_cal_animation_target (const char ∗aviName, int playCount, int options)
- int enable_external_calibration_device (getExButtonStates buttonStatesfcn, const char ∗config, void ∗userData)

26.23.1 Define Documentation

26.23.1.1 #define bitmap_save sdl_bitmap_save

See `sdl_bitmap_save()`

26.23.1.2 #define bitmap_save_and_backdrop sdl_bitmap_save_and_backdrop

See `sdl_bitmap_save_and_backdrop()`

26.23.1.3 #define bitmap_to_backdrop sdl_bitmap_to_backdrop

See `sdl_bitmap_to_backdrop()`

26.23.2 Function Documentation

26.23.2.1 void close_expt_graphics (void)

Call this function at the end of the experiment or before destroying the window registered with `init_expt_graphics()`. This call will disable calibration and drift correction until a new window is registered.

26.23.2.2 int enable_external_calibration_device (getExButtonStates *buttonStatesfcn*, const char ∗ *config*, void ∗ *userData*)

Enables non keyboard devices to be used for calibration control.

Parameters:

← ***buttonStatesfcn*** callback function reads the device and returns appropriate data.

Remarks:

: Use EXTERNAL_DEV_NONE to disable the device. Use EXTERNAL_DEV_CEDRUS for built-in cedrus device support.

Parameters:

← ***config*** - A character string of the config file content or config file name. Whether the config is the content or a file name is determined by looking for a new line character. If there is a new line character the content is assumed. To use the default config, set the parameter to NULL. The default config will be:

```
# User mode definition
# [MODE n]
# Defines a 'user' mode, where n is the mode ID
# Valid Commands for user mode:
#       TEXT_LINE "This is a line of text"
#       DISPLAY_IMAGE
#       BUTTON id button_command [command args]
#               Valid BUTTON button_commands:
#                       NO_ACTION
#
#                       AUTO_THRESH
#                       PUPIL_THRESH_UP
#                       PUPIL_THRESH_DOWN
#                       CR_THRESH_UP
#                       CR_THRESH_DOWN
#
#                       START_CALIBRATION
#                       START_VALIDATION
#                       START_DRIFT_CORRECT
#
#                       GOTO_MODE
#                       EXIT
#
#                       NEXT_IMAGE (only makes sense when DISPLAY_IMAGE is set for mode)
#                       PREV_IMAGE (only makes sense when DISPLAY_IMAGE is set for mode)
#
# Predefined modes
# [MODE C]
# Calibration mode
# Valid Commands
#       REDO_LAST_TARGET
#       ACCEPT_TARGET
#
# [MODE V]
# Validation mode
# Valid Commands
#       REDO_LAST_TARGET
#       ACCEPT_TARGET
#
# [MODE D]
# Drift Correction mode
# Valid Commands
#       ACCEPT_TARGET

[MODE 1]
TEXT_LINE "EyeLink Setup:"
TEXT_LINE "1 -> View Camera Images"
TEXT_LINE "2 -> Start Calibration"
TEXT_LINE "3 -> Start Validation"
TEXT_LINE "4 -> Exit EyeLink Setup"

BUTTON 1 GOTO_MODE 2
BUTTON 2 GOTO_MODE C
BUTTON 3 GOTO_MODE V
BUTTON 4 EXIT




[MODE 2]
TEXT_LINE "Camera Views:"
TEXT_LINE "1 -> Next Camera View"
TEXT_LINE "2 -> Previous Camera View"
TEXT_LINE "3 -> Go To Pupil Threshold Adjustment Mode"
TEXT_LINE "4 -> Exit EyeLink Setup"


BUTTON 1 NEXT_IMAGE
BUTTON 2 PREV_IMAGE
BUTTON 3 GOTO_MODE 3
BUTTON 4 EXIT
DISPLAY_IMAGE TRUE


[MODE 3]
TEXT_LINE "Pupil Threshold Adjustment:"
TEXT_LINE "1 -> Increase Threshold"
TEXT_LINE "2 -> Decrease Threshold"
TEXT_LINE "3 -> Auto Threshold"
TEXT_LINE "4 -> Go To CR Threshold Adjustment Mode"
```

```
BUTTON 1 PUPIL_THRESH_UP
BUTTON 2 PUPIL_THRESH_DOWN
BUTTON 3 AUTO_THRESH
BUTTON 4 GOTO_MODE 4
DISPLAY_IMAGE TRUE


[MODE 4]
TEXT_LINE "CR Threshold Adjustment:"
TEXT_LINE "1 -> Increase Threshold"
TEXT_LINE "2 -> Decrease Threshold"
TEXT_LINE "3 -> Auto Threshold"
TEXT_LINE "4 -> Go To EyeLink Setup Mode"


BUTTON 1 CR_THRESH_UP
BUTTON 2 CR_THRESH_DOWN
BUTTON 3 AUTO_THRESH
BUTTON 4 GOTO_MODE 1
DISPLAY_IMAGE TRUE


[MODE C]
BUTTON 1 ACCEPT_TARGET
BUTTON 2 REDO_LAST_TARGE
BUTTON 4 EXIT

[MODE V]
BUTTON 1 ACCEPT_TARGET
BUTTON 2 REDO_LAST_TARGE
BUTTON 4 EXIT

[MODE D]
BUTTON 1 ACCEPT_TARGET
BUTTON 4 EXIT
```

← ***userData*** user data to pass back in the callback.

Returns:

1 upon success, 0 otherwise

26.23.2.3 void get_display_information (DISPLAYINFO ∗ *di*)

This is an optional function to get information on video driver and current mode use this to determine if in proper mode for experiment.

Parameters:

→ ***di*** A valid pointer to DISPLAYINFO is passed in to return values.

Remarks:

The prototype of this function can be changed to match one's need or if it is not necessary, one can choose not to implement this function also.

26.23.2.4 INT16 init_expt_graphics (SDL_Surface ∗ *hwnd*, DISPLAYINFO ∗ *info*)

You must always create a borderless, full-screen window for your experiment. This function registers the window with EXPTSPPT so it may be used for calibration and drift correction. The window should not be destroyed until it is released with `close_expt_graphics()`. This window will be subclassed (some messages intercepted) during calibration and drift correction.

Parameters:

hwnd Handle of window that is to be used for calibration and drift correction. If `NULL` is passed in, SDL initialized and requested display mode is set.

info `NULL` or pointer to a `DISPLAYINFO` structure to fill with display mode data. If `NULL` is passed in, current display mode is used.

Returns:

`0` if success, `-1` if error occurred internally.

Default initialization of eyelinkn_core_library:

```
int defaultGraphicsSetup()
{
        DISPLAYINFO disp;
        memset(&disp,0,sizeof(DISPLAYINFO));

        disp.width =640;
        disp.height = 480;
        disp.bits =32;
        disp.refresh = 60;
        if(init_expt_graphics(NULL, &disp))
        {
                printf("init_expt_graphics failed \n");
                return -1;
        }
        return 0;
}
```

Custom initialization of SDL can be done in the following manner.

```
int customGraphicsSetup()
{
        SDL_Surface *mainwindow = NULL;
        if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) // initialize SDL
        {
                printf("Couldn't initialize SDL: %s!",SDL_GetError());
                return -1;
        }

        mainwindow = SDL_SetVideoMode(800,600,32,SDL_SWSURFACE|SDL_FULLSCREEN); // set video mode
        if(!mainwindow)
        {
                printf("Failed to set video mode: %s! ",SDL_GetError());
                return -1;
        }
        if(init_expt_graphics(mainwindow, NULL)) // tell core graphics to use the set video mode.
                return -1;
        return 0;
}
@remark eyelink_core_graphics library does not support OPENGL.
```

26.23.2.5 void reset_background_surface ()

Removes the custom background. equivalent of calling set_cal_background_surface(NULL);

26.23.2.6 int sdl_bitmap_save (SDL_Surface ∗ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*)

This function saves the entire bitmap or selected part of a bitmap in an image file (with an extension of .png, .bmp, .jpg, or .tif). It creates the specified file if this file does not exist. If the file exists, it replaces the file unless `SV_NOREPLACE` is specified in the field of "sv_options". The directory to which the file will be written is specified in the path field.

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

fname Name of the image file to be saved. Currently, only .PNG, .BMP, .JPG, and .TIF files are saved.

path Directory or drive path in quotes ("." for current directory).

sv_options Use `SV_NOREPLACE` if not to replace an existing file; use `SV_MAKEPATH` to create a new path.

Returns:

`0` if successful, else `-1`.

26.23.2.7 int sdl_bitmap_save_and_backdrop (SDL_Surface ∗ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*, INT16 *xd*, INT16 *yd*, UINT16 *bx_options*)

This function saves the entire bitmap as a .BMP, .JPG, .PNG, or .TIF file, and transfers the image to tracker as backdrop for gaze cursors (See `bitmap_save()` and `bitmap_to_backdrop()` for more information).

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

fname Name of the image file to be saved. Currently, only .PNG, .BMP, .JPG, and .TIF files are saved.

path Directory or drive path in quotes ("." for current directory).

sv_options Use `SV_NOREPLACE` if not to replace an existing file; use `SV_MAKEPATH` to create a new path.

xd Specifies the x-coordinate of the upper-left corner of the tracker screen.

yd Specifies the y-coordinate of the upper-left corner of the tracker screen.

bx_options Set with a bitwise OR of the following constants: `BX_MAXCONTRAST:` Maximizes contrast for clearest image; `BX_AVERAGE:` averages combined pixels; `BX_DARKEN:` chooses darkest (keep thin dark lines); `BX_LIGHTEN:` chooses darkest (keep thin white lines); `BX_NODITHER:` disables dithering to get clearest text; `BX_GREYSCALE:` converts to grayscale.

Returns:

`0` if successful, `-1` if couldn't save, `-2` if couldn't transfer

26.23.2.8 int sdl_bitmap_to_backdrop (SDL_Surface ∗ *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, INT16 *xd*, INT16 *yd*, UINT16 *bx_options*)

This function transfers the bitmap to the tracker PC as backdrop for gaze cursors. The field "bx_options", set with bitwise OR of the following constants, determines how bitmap is processed: `BX_AVERAGE` (averaging combined pixels), `BX_DARKEN` (choosing darkest and keep thin dark lines), and `BX_LIGHTEN` (choosing darkest and keep thin white lines) control how bitmap size is reduced to fit tracker display; `BX_MAXCONTRAST` maximizes contrast for clearest image; `BX_NODITHER` disables the dithering of the image; `BX_GREYSCALE` converts the image to grayscale (grayscale works best for EyeLink I, text, etc.)

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

xd Specifies the x-coordinate of the upper-left corner of the tracker screen.

yd Specifies the y-coordinate of the upper-left corner of the tracker screen.

bx_options Set with a bitwise OR of the following constants: `BX_MAXCONTRAST:` Maximizes contrast for clearest image; `BX_AVERAGE:` averages combined pixels; `BX_DARKEN:` chooses darkest (keep thin dark lines); `BX_LIGHTEN:` chooses darkest (keep thin white lines); `BX_NODITHER:` disables dithering to get clearest text; `BX_GREYSCALE:` converts to grayscale.

Returns:

`0` if successful, else `-1` or `-2`;

26.23.2.9 int set_cal_animation_target (const char ∗ *aviName*, int *playCount*, int *options*)

Allow one to set target with animation. The expected video can be loadable using VFW(type 1 avi) also, both audio and video streams must be present. The audio stream must be of pcm type.

Parameters:

aviname Name of the avi to use

playCount How many time to loop through the video. Specify -1 to loop indefinitely. for future use.

26.23.2.10 void set_cal_background_surface (SDL_Surface ∗ *surface*)

Allow one to set arbituary background in place of flat background

Parameters:

surface

26.23.2.11 void set_cal_sounds (char ∗ *ontarget*, char ∗ *ongood*, char ∗ *onbad*)

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.

Remarks:

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:

ontarget Sets sound to play when target moves.

ongood Sets sound to play on successful operation.

onbad Sets sound to play on failure or interruption.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()` and `set_dcorr_sounds()`

26.23.2.12 void set_cal_target_surface (SDL_Surface ∗ *surface*)

Allow one to set arbituary target in place of circle target. Eg. a custom cursor.

Parameters:

surface

26.23.2.13 void set_calibration_colors (SDL_Color ∗ *fg*, SDL_Color ∗ *bg*)

Passes the colors of the display background and fixation target to the EXPTSPPT 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 EXPTSPPT library. This also prevents flickering of the display at the beginning and end of drift correction.

Parameters:

fg Color used for drawing calibration target.

bg Color used for drawing calibration background.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()`

26.23.2.14 INT16 set_camera_image_position (INT16 *left*, INT16 *top*, INT16 *right*, INT16 *bottom*)

To adjust camera image position. By default the camera is placed at the centre of the screen.

Parameters:

left Left position.

top Top position.

right Right position.

bottom Bottom position.

26.23.2.15 void set_dcorr_sounds (char * *ontarget*, char * *ongood*, char * *onbad*)

Selects the sounds to be played during `do_drift_correct()`. 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.

Remarks:

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:

ontarget Sets sound to play when target moves.

ongood Sets sound to play on successful operation.

onbad Sets sound to play on failure or interruption.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()` and `set_cal_sounds()`

26.23.2.16 void set_target_size (UINT16 *diameter*, UINT16 *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. If `<holesize>` is `0`, no central feature will be drawn.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()`

26.24 GDI Graphics Functions

Defines

- #define **SCREEN_LEFT** dispinfo.left
- #define **SCREEN_TOP** dispinfo.top
- #define **SCREEN_RIGHT** dispinfo.right
- #define **SCREEN_BOTTOM** dispinfo.bottom
- #define **SCRHEIGHT** dispinfo.height
- #define **SCRWIDTH** dispinfo.width
- #define process_key_messages gdi_process_key_messages
- #define bitmap_save_and_backdrop gdi_bitmap_save_and_backdrop
- #define bitmap_to_backdrop gdi_bitmap_to_backdrop
- #define bitmap_save gdi_bitmap_save
- #define message_pump(x) message_pump()
- #define **CALLTYPE** ELCALLTYPE
- #define **HOOK_ERROR** -1
- #define **HOOK_CONTINUE** 0
- #define **HOOK_NODRAW** 1
- #define **CAL_TARG_BEEP** 1
- #define **CAL_GOOD_BEEP** 0
- #define **CAL_ERR_BEEP** -1
- #define **DC_TARG_BEEP** 3
- #define **DC_GOOD_BEEP** 2
- #define **DC_ERR_BEEP** -2

Functions

- void set_calibration_colors (COLORREF fg, COLORREF bg)
- void set_target_size (UINT16 diameter, UINT16 holesize)
- void set_cal_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- void set_dcorr_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- INT16 set_camera_image_position (INT16 left, INT16 top, INT16 right, INT16 bottom)
- void get_display_information (DISPLAYINFO ∗di)
- INT16 init_expt_graphics (HWND hwnd, DISPLAYINFO ∗info)
- void close_expt_graphics (void)
- void wait_for_video_refresh (void)
- UINT16 gdi_process_key_messages (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
- void wait_for_drawing (HWND hwnd)
- int gdi_bitmap_save_and_backdrop (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 bx_options)
- int gdi_bitmap_to_backdrop (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 bx_options)
- int gdi_bitmap_save (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- INT16 set_setup_cal_display_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 set_clear_cal_display_hook (INT16(∗hookfn)(HDC hdc), INT16 options)
- INT16 CALLTYPE set_erase_cal_target_hook (INT16(∗hookfn)(HDC hdc), INT16 options)

- INT16 CALLTYPE set_draw_cal_target_hook (INT16(∗hookfn)(HDC hdc, INT16 ∗x, INT16 ∗y), INT16 options)
- INT16 set_exit_cal_display_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 CALLTYPE set_cal_sound_hook (INT16(∗hookfn)(INT16 ∗error), INT16 options)
- INT16 set_record_abort_hide_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 set_setup_image_display_hook (INT16(∗hookfn)(INT16 width, INT16 height), INT16 options)
- INT16 set_image_title_hook (INT16(∗hookfn)(INT16 threshold, char ∗cam_name), INT16 options)
- INT16 set_draw_image_line_hook (INT16(∗hookfn)(INT16 width, INT16 line, INT16 totlines, byte ∗pixels), INT16 options)
- INT16 set_set_image_palette_hook (INT16(∗hookfn)(INT16 ncolors, byte r[ ], byte g[ ], byte b[ ]), INT16 options)
- INT16 set_exit_image_display_hook (INT16(∗hookfn)(void), INT16 options)
- void initialize_gc_window (int wwidth, int wheight, HBITMAP window_bitmap, HBITMAP background_bitmap, HWND window, RECT display_rect, int is_mask, int deadband)
- void redraw_gc_window (int x, int y)
- HDC get_window_dc (void)
- INT16 release_window_dc (HDC hdc)
- INT16 edit_dialog (HWND hwnd, LPSTR title, LPSTR msg, LPSTR txt, INT16 maxsize)
- INT16 ask_session (HWND hw, LPSTR title, LPSTR msg, LPSTR path, INT16 pathmax, LPSTR txt, INT16 maxsize)

Variables

- DISPLAYINFO **dispinfo**

26.24.1 Define Documentation

26.24.1.1 #define bitmap_save gdi_bitmap_save

See `gdi_bitmap_save()`

26.24.1.2 #define bitmap_save_and_backdrop gdi_bitmap_save_and_backdrop

See `gdi_bitmap_save_and_backdrop()`

26.24.1.3 #define bitmap_to_backdrop gdi_bitmap_to_backdrop

See `gdi_bitmap_to_backdrop()`

26.24.1.4 #define message_pump(x) message_pump()

See `message_pump()`

26.24.1.5 #define process_key_messages gdi_process_key_messages

See `gdi_process_key_messages()`

26.24.2 Function Documentation

26.24.2.1 INT16 ask_session (HWND *hw*, LPSTR *title*, LPSTR *msg*, LPSTR *path*, INT16 *pathmax*, LPSTR *txt*, INT16 *maxsize*)

This function accepts path and session name from the user.

Parameters:

hwnd The "parent" window, usually the experiment window or `NULL` if no window exists.

title Text to be displayed in the frame of the dialog box.

msg Instructions to be displayed in the dialog box.

txt The buffer into which text will be entered. Any text in this buffer will be displayed as the initial contents of the edit box.

maxsize The maximum buffer length of txt.

inout] path The default path and selected path upon return. If "" used as default path the default documents directory will be used as the default path.

pathmax size of the path parameter.

Returns:

`0` if the ENTER key was pressed or "OK" was clicked. `1` if 'ESC' pressed or "Cancel" clicked. `-1` if ALT-F4 pressed to destroy the dialog box.

26.24.2.2 void close_expt_graphics (void)

Call this function at the end of the experiment or before destroying the window registered with `init_expt_graphics()`. This call will disable calibration and drift correction until a new window is registered.

26.24.2.3 INT16 edit_dialog (HWND *hwnd*, LPSTR *title*, LPSTR *msg*, LPSTR *txt*, INT16 *maxsize*)

All experiments require the input of information: EDF file name, randomization file, and so on. This function implements a simple text-entry dialog box for this purpose. A title in the window frame and a message can be set, and the initial text set. The length of the text to be entered can also be limited.

Parameters:

hwnd The "parent" window, usually the experiment window or `NULL` if no window exists.

title Text to be displayed in the frame of the dialog box.

msg Instructions to be displayed in the dialog box.

txt The buffer into which text will be entered. Any text in this buffer will be displayed as the initial contents of the edit box.

maxsize The maximum buffer length of txt.

Returns:

`0` if the ENTER key was pressed or "OK" was clicked. `1` if 'ESC' pressed or "Cancel" clicked. `-1` if ALT-F4 pressed to destroy the dialog box.

26.24.2.4 int gdi_bitmap_save (HBITMAP *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*)

This function saves the entire bitmap or selected part of a bitmap in an image file (with an extension of .png, .bmp, .jpg, or .tif). It creates the specified file if this file does not exist. If the file exists, it replaces the file unless `SV_NOREPLACE` is specified in the field of "sv_options". The directory to which the file will be written is specified in the path field.

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

fname Name of the image file to be saved. Currently, only .PNG, .BMP, .JPG, and .TIF files are saved.

path Directory or drive path in quotes ("." for current directory).

sv_options Use `SV_NOREPLACE` if not to replace an existing file; use `SV_MAKEPATH` to create a new path.

Returns:

`0` if successful, else `-1`.

26.24.2.5 int gdi_bitmap_save_and_backdrop (HBITMAP *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, char ∗ *fname*, char ∗ *path*, INT16 *sv_options*, INT16 *xd*, INT16 *yd*, UINT16 *bx_options*)

This function saves the entire bitmap as a .BMP, .JPG, .PNG, or .TIF file, and transfers the image to tracker as backdrop for gaze cursors (See `bitmap_save()` and `bitmap_to_backdrop()` for more information).

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

fname Name of the image file to be saved. Currently, only .PNG, .BMP, .JPG, and .TIF files are saved.

path Directory or drive path in quotes ("." for current directory).

sv_options Use `SV_NOREPLACE` if not to replace an existing file; use `SV_MAKEPATH` to create a new path.

xd Specifies the x-coordinate of the upper-left corner of the tracker screen.

yd Specifies the y-coordinate of the upper-left corner of the tracker screen.

bx_options Set with a bitwise OR of the following constants:

- `BX_MAXCONTRAST`: Maximizes contrast for clearest image.
- `BX_AVERAGE`: averages combined pixels.
- `BX_DARKEN`: chooses darkest (keep thin dark lines).

- `BX_LIGHTEN`: chooses darkest (keep thin white lines).
- `BX_NODITHER`: disables dithering to get clearest text.
- `BX_GREYSCALE`: converts to grayscale.

Returns:

`0` if successful, `-1` if couldn't save, `-2` if couldn't transfer.

26.24.2.6 int gdi_bitmap_to_backdrop (HBITMAP *hbm*, INT16 *xs*, INT16 *ys*, INT16 *width*, INT16 *height*, INT16 *xd*, INT16 *yd*, UINT16 *bx_options*)

This function transfers the bitmap to the tracker PC as backdrop for gaze cursors. The field "bx_options", set with bitwise OR of the following constants, determines how bitmap is processed: `BX_AVERAGE` (averaging combined pixels), `BX_DARKEN` (choosing darkest and keep thin dark lines), and `BX_LIGHTEN` (choosing darkest and keep thin white lines) control how bitmap size is reduced to fit tracker display; `BX_MAXCONTRAST` maximizes contrast for clearest image; `BX_NODITHER` disables the dithering of the image; `BX_GREYSCALE` converts the image to grayscale (grayscale works best for EyeLink I, text, etc.)

Parameters:

hbm Handle to the bitmap image.

xs Specifies the x-coordinate of the upper-left corner of the source bitmap.

ys Specifies the y-coordinate of the upper-left corner of the source bitmap.

width Specify the width of the source image to be copied (set to `0` to use all).

height Specify the height of the source image to be copied (set to `0` to use all).

xd Specifies the x-coordinate of the upper-left corner of the tracker screen.

yd Specifies the y-coordinate of the upper-left corner of the tracker screen.

bx_options Set with a bitwise OR of the following constants:

- `BX_MAXCONTRAST`: Maximizes contrast for clearest image.
- `BX_AVERAGE`: averages combined pixels.
- `BX_DARKEN`: chooses darkest (keep thin dark lines).
- `BX_LIGHTEN`: chooses darkest (keep thin white lines).
- `BX_NODITHER`: disables dithering to get clearest text.
- `BX_GREYSCALE`: converts to grayscale.

Returns:

`0` if successful, else `-1` or `-2`.

26.24.2.7 UINT16 gdi_process_key_messages (HWND *hWnd*, UINT *message*, WPARAM *wParam*, LPARAM *lParam*)

Call this function in your window message processing function to handle `WM_CHAR` and `WM_KEYDOWN` messages. These will be translated to an EyeLink key code and saved for `getkey()`.

Parameters:

hWnd Handle to window. This message may be `NULL`.

message Windows message.

wParam First message parameter.

lParam Second windows parameter.

Returns:

`0` or `JUNK_KEY` if no key generated.

Example:

```
// The following code illustrates the use of process_key_messages().
// This would usually be called from a message or event handler (see the
// w32_demo_window.c module) for a complete example

#include <eyelink.h>
switch (message)
{
case WM_KEYDOWN:
case WM_CHAR:
    // Processes key messages: these can be accessed by getkey()
    process_key_messages(hwnd, message, wparam, lparam);
    break;

    ...
    // Other windows messages and events
}
```

See also:

`getkey()` and `translate_key_messages()`

26.24.2.8 void get_display_information (DISPLAYINFO ∗ *di*)

This is an optional function to get information on video driver and current mode use this to determine if in proper mode for experiment.

Parameters:

→ ***di*** A valid pointer to DISPLAYINFO is passed in to return values.

Remarks:

The prototype of this function can be changed to match one's need or if it is not necessary, one can choose not to implement this function also.

26.24.2.9 HDC get_window_dc (void)

Gets a display context to draw on the page. This is a static DC, so all settings from your last use are preserved.

26.24.2.10 INT16 init_expt_graphics (HWND *hwnd*, DISPLAYINFO ∗ *info*)

You must always create a borderless, full-screen window for your experiment. This function registers the window with EXPTSPPT so it may be used for calibration and drift correction. The window should not be destroyed until it is released with `close_expt_graphics()`. This window will be subclassed (some messages intercepted) during calibration and drift correction.

Parameters:

hwnd Handle of window that is to be used for calibration and drift correction. This should be a borderless, full-screen window. If your language can't give you a window handle, use `NULL` and the topmost window will be detected.

info NULL or pointer to a DISPLAYINFO structure to fill with display mode data.

Returns:
0 if success, -1 if error occurred internally.

26.24.2.11 void initialize_gc_window (int *wwidth*, int *wheight*, HBITMAP *window_bitmap*, HBITMAP *background_bitmap*, HWND *window*, RECT *display_rect*, int *is_mask*, int *deadband*)

- To initialize a gaze contengent window.
- Initial setup of gaze-contingent window before drawing it.
- Sets size of window, and whether it is a foveal mask.
- If height or width is -1, the window will be a bar covering the display.

Parameters:

wwidth

wheight

window_bitmap

background_bitmap

window The window to display in.

display_rect Area of display in window.

is_mask

deadband Sets number of pixels of anti-jitter applied.

26.24.2.12 void redraw_gc_window (int *x*, int *y*)

Set the location of the gaze window to this new position. The first time window is drawn, the background outside the window will be filled in too. If x or y is MISSING_DATA (defined in eyelink.h), window is hidden.

Parameters:

x X location.

y Y location.

26.24.2.13 INT16 release_window_dc (HDC *hdc*)

Release the page DC

26.24.2.14 INT16 CALLTYPE set_cal_sound_hook (INT16(∗)(INT16 ∗error) *hookfn*, INT16 *options*)

To modify the behaviour of cal_sound() function.

Parameters:

hookfn Function to replace cal_sound().

options For future use.

26.24.2.15 void set_cal_sounds (char ∗ *ontarget*, char ∗ *ongood*, char ∗ *onbad*)

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.

Remarks:

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:

ontarget Sets sound to play when target moves.

ongood Sets sound to play on successful operation.

onbad Sets sound to play on failure or interruption.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()` and `set_dcorr_sounds()`

26.24.2.16 void set_calibration_colors (COLORREF *fg*, COLORREF *bg*)

Passes the colors of the display background and fixation target to the EXPTSPPT 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 EXPTSPPT library. This also prevents flickering of the display at the beginning and end of drift correction.

Parameters:

fg Color used for drawing calibration target.

bg Color used for drawing calibration background.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()`

26.24.2.17 INT16 set_camera_image_position (INT16 *left*, INT16 *top*, INT16 *right*, INT16 *bottom*)

To adjust camera image position. By default the camera is placed at the centre of the screen.

Parameters:

left Left position.

top Top position.

right Right position.

bottom Bottom position.

26.24.2.18 INT16 set_clear_cal_display_hook (INT16(∗)(HDC hdc) *hookfn*, INT16 *options*)

To modify the behaviour of `clear_cal_display()` function.

Parameters:

hookfn Function to replace `clear_cal_display()`.

options For future use.

26.24.2.19 void set_dcorr_sounds (char ∗ *ontarget*, char ∗ *ongood*, char ∗ *onbad*)

Selects the sounds to be played during `do_drift_correct()`. 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.

Remarks:

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:

ontarget Sets sound to play when target moves.

ongood Sets sound to play on successful operation.

onbad Sets sound to play on failure or interruption.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()` and `set_cal_sounds()`

26.24.2.20 INT16 CALLTYPE set_draw_cal_target_hook (INT16(∗)(HDC hdc, INT16 ∗x, INT16 ∗y) *hookfn*, INT16 *options*)

To modify the behaviour of `draw_cal_target()` function.

Parameters:

hookfn Function to replace `draw_cal_target()`.

options For future use.

26.24.2.21 INT16 set_draw_image_line_hook (INT16(∗)(INT16 width, INT16 line, INT16 totlines, byte ∗pixels) *hookfn*, INT16 *options*)

To modify the behaviour of `draw_image_line()` function.

Parameters:

hookfn Function to replace `draw_image_line()`.

options For future use.

26.24.2.22 INT16 CALLTYPE set_erase_cal_target_hook (INT16(∗)(HDC hdc) *hookfn*, INT16 *options*)

To modify the behaviour of `erase_cal_target()` function.

Parameters:

hookfn Function to replace `erase_cal_target()`.

options For future use.

26.24.2.23 INT16 set_exit_cal_display_hook (INT16(∗)(void) *hookfn*, INT16 *options*)

To modify the behaviour of `exit_cal_display()` function.

Parameters:

hookfn Function to replace `exit_cal_display()`.

options For future use.

26.24.2.24 INT16 set_exit_image_display_hook (INT16(∗)(void) *hookfn*, INT16 *options*)

To modify the behaviour of `exit_image_display()` function.

Parameters:

hookfn Function to replace `exit_image_display()`.

options For future use.

26.24.2.25 INT16 set_image_title_hook (INT16(∗)(INT16 threshold, char ∗cam_name) *hookfn*, INT16 *options*)

To modify the behaviour of `image_title()` function.

Parameters:

hookfn Function to replace `image_title()`.

options For future use.

26.24.2.26 INT16 set_record_abort_hide_hook (INT16(∗)(void) *hookfn*, INT16 *options*)

To modify the behaviour of `record_abort_hide()` function.

Parameters:

hookfn Function to replace `record_abort_hide()`.

options For future use.

26.24.2.27 INT16 set_set_image_palette_hook (INT16(∗)(INT16 ncolors, byte r[ ], byte g[ ], byte b[ ]) *hookfn*, INT16 *options*)

To modify the behaviour of `set_image_palette()` function.

Parameters:

- ***hookfn*** Function to replace `set_image_palette()`.
- ***options*** For future use.

26.24.2.28 INT16 set_setup_cal_display_hook (INT16(∗)(void) *hookfn*, INT16 *options*)

To modify the behaviour of `setup_cal_display()` function.

Parameters:

- ***hookfn*** Function to replace `setup_cal_display()`.
- ***options*** For future use.

26.24.2.29 INT16 set_setup_image_display_hook (INT16(∗)(INT16 width, INT16 height) *hookfn*, INT16 *options*)

To modify the behaviour of `setup_image_display()` function.

Parameters:

- ***hookfn*** Function to replace `setup_image_display()`.
- ***options*** For future use.

26.24.2.30 void set_target_size (UINT16 *diameter*, UINT16 *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. If <`holesize`> is `0`, no central feature will be drawn.

Example: See `do_tracker_setup()`

See also:

`do_tracker_setup()`

26.24.2.31 void wait_for_drawing (HWND *hwnd*)

Forces start of drawing, waits for drawing to finish hwnd may be NULL for all displays together

Parameters:

- ***hwnd*** Handle to the window.

26.24.2.32 void wait_for_video_refresh (void)

This function will not return until the current refresh of the monitor has completed (at the start of vertical retrace). This can be used to synchronize drawing to the scanning out of the display, and to determine when a stimulus was first seen by the subject. The DriverLinx PortIO driver must be installed for this function to work.

Remarks:
This is function is only applicable for the GDI version.

Chapter 27

Data Structure

27.1 _CrossHairInfo Struct Reference

27.1.1 Detailed Description

Structure to hold cross hair information.

Structure to hold cross hair information that are needed to draw the cross hair on camera images.

Data Fields

- short majorVersion
- short minorVersion
- int w
- int h
- void ∗ privatedata
- void ∗ userdata
- void(∗ drawLine)(CrossHairInfo ∗dt, int x1, int y1, int x2, int y2, int colorindex)
- void(∗ drawLozenge)(CrossHairInfo ∗dt, int x, int y, int w, int h, int colorindex)
- void(∗ getMouseState)(CrossHairInfo ∗dt, int ∗x, int ∗y, int ∗state)
- int reserved1
- int reserved2
- int reserved3
- int reserved4

27.1.2 Field Documentation

27.1.2.1 void(∗ drawLine)(CrossHairInfo ∗dt, int x1, int y1, int x2, int y2, int colorindex)

drawLine shoud draw a line from (x1,y1) -> (x2,y2) with the given color

Parameters:

← ***x1***

27.1.2.2 void(* drawLozenge)(CrossHairInfo *dt, int x, int y, int w, int h, int colorindex)

drawLozenge shoud draw circle if the width and height are equal. otherwise find the smallest width and draw semi-circles on the longest side at both ends and connect the semi circles with lines.

27.1.2.3 void(* getMouseState)(CrossHairInfo *dt, int *x, int *y, int *state)

getMouseState shoud fill values for (x,y) with current mouse position and the state = 1 if pressed. the x and y values are respective to top left hand corner of the image

27.1.2.4 int h

Display height of the camera image. This need not to match the size given at setup_image_display_hook

27.1.2.5 short majorVersion

for the moment this should be set to 1.

27.1.2.6 short minorVersion

for the moment this should be set to 0.

27.1.2.7 void* privatedata

Private data used by internals of eyelink_draw_cross_hair. This pointer should not be touched externally

27.1.2.8 int reserved1

for future use

27.1.2.9 int reserved2

for future use

27.1.2.10 int reserved3

for future use

27.1.2.11 int reserved4

for future use

27.1.2.12 void* userdata

Attribute to hold any user data. Any data set here untouched by eyelink_draw_cross_hair.

27.1.2.13 int w

Display width of the camera image. This need not to match the size given at setup_image_display_hook

27.2 DISPLAYINFO Struct Reference

27.2.1 Detailed Description

This structure holds information on the display.

This structure holds information on the display Call `get_display_infomation()` to fill this with data Check mode before running experiment!

Data Fields

- INT32 left
- INT32 top
- INT32 right
- INT32 bottom
- INT32 width
- INT32 height
- INT32 bits
- INT32 palsize
- INT32 palrsvd
- INT32 pages
- float refresh
- INT32 winnt

27.2.2 Field Documentation

27.2.2.1 INT32 bits

bits per pixel

27.2.2.2 INT32 bottom

bottom of display

27.2.2.3 INT32 height

height of display

27.2.2.4 INT32 left

left of display

27.2.2.5 INT32 pages

pages supported

27.2.2.6 INT32 palrsvd

number of static entries in palette (`0` if not indexed display mode)

27.2.2.7 INT32 palsize

total entries in palette (`0` if not indexed display mode)

27.2.2.8 float refresh

refresh rate in Hz

27.2.2.9 INT32 right

right of display

27.2.2.10 INT32 top

top of display

27.2.2.11 INT32 width

width of display

27.2.2.12 INT32 winnt

`0` if Windows 95, `1` if Windows NT

27.3 ELINKNODE Struct Reference

27.3.1 Detailed Description

Name and address for connection.

Name and address for connection or ping

Data Fields

- ELINKADDR addr
- char name [ELNAMESIZE]

27.3.2 Field Documentation

27.3.2.1 ELINKADDR addr

address of the remote or local tracker

27.3.2.2 char name[ELNAMESIZE]

name of the remote or local tracker

27.4 EYEBITMAP Struct Reference

27.4.1 Detailed Description

Represents a bitmap image.

Data Fields

- INT32 w
- INT32 h
- INT32 pitch
- INT32 depth
- void * pixels
- EYEPIXELFORMAT * format

27.4.2 Field Documentation

27.4.2.1 INT32 depth

Depth of the image. possible values are 8,15,16,24,32

27.4.2.2 EYEPIXELFORMAT* format

pixel format of the image.

27.4.2.3 INT32 h

height of the image

27.4.2.4 INT32 pitch

pitch of image lines. This can be 0. if this is 0, then ((depth+7)/8)*width is used

27.4.2.5 void* pixels

uncompressed pixel data

27.4.2.6 INT32 w

width of the image

27.5 EYECOLOR Struct Reference

27.5.1 Detailed Description

Represents an RGB color.

Data Fields

- byte r
- byte g
- byte b
- byte **unused**

27.5.2 Field Documentation

27.5.2.1 byte b

Blue

27.5.2.2 byte g

Green

27.5.2.3 byte r

Red

27.6 EYEPALETTE Struct Reference

27.6.1 Detailed Description

Represents a palette index.

Data Fields

- int ncolors
- EYECOLOR ∗ colors

27.6.2 Field Documentation

27.6.2.1 EYECOLOR∗ colors

Actual colors

27.6.2.2 int ncolors

Number of colors in the palette

27.7 EYEPIXELFORMAT Struct Reference

27.7.1 Detailed Description

Represents pixel format of an image or surface.

Data Fields

- byte **colorkey**
- INT32 **Rmask**
- INT32 **Gmask**
- INT32 **Bmask**
- INT32 **Amask**
- EYEPALETTE $*$ **palette**

27.8 FEVENT Struct Reference

27.8.1 Detailed Description

Floating-point eye event.

The EyeLink tracker analyzes the eye-position samples during recording to detect saccades, and accumulates data on saccades and fixations. Events are produced to mark the start and end of saccades, fixations and blinks. When both eyes are being tracked, left and right eye events are produced, as indicated in the eye field of the FEVENT structure.

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. This includes start and end and average measures of position and pupil size, plus peak and average velocity in degrees per second.

Data Fields

- UINT32 time
- INT16 type
- UINT16 read
- INT16 eye
- UINT32 sttime
- UINT32 entime
- float hstx
- float hsty
- float gstx
- float gsty
- float sta
- float henx
- float heny
- float genx
- float geny
- float ena
- float havx
- float havy
- float gavx
- float gavy
- float ava
- float avel
- float pvel
- float svel
- float evel
- float supd_x
- float eupd_x
- float supd_y
- float eupd_y
- UINT16 status

27.8.2 Field Documentation

27.8.2.1 float ava

average area

27.8.2.2 float avel

avg velocity accum

27.8.2.3 float ena

ending area

27.8.2.4 UINT32 entime

end times

27.8.2.5 float eupd_x

end units-per-degree x

27.8.2.6 float eupd_y

end units-per-degree y

27.8.2.7 float evel

end velocity

27.8.2.8 INT16 eye

eye: 0=left,1=right

27.8.2.9 float gavx

average x

27.8.2.10 float gavy

average y

27.8.2.11 float genx

ending point x

27.8.2.12 float geny

ending point y

27.8.2.13 float gstx

starting point x

27.8.2.14 float gsty

starting point y

27.8.2.15 float havx

average x

27.8.2.16 float havy

average y

27.8.2.17 float henx

ending point x

27.8.2.18 float heny

ending point y

27.8.2.19 float hstx

starting point x

27.8.2.20 float hsty

starting point y

27.8.2.21 float pvel

peak velocity accum

27.8.2.22 UINT16 read

flags which items were included

27.8.2.23 float sta

starting area

27.8.2.24 UINT16 status

error, warning flags

27.8.2.25 UINT32 sttime

start times

27.8.2.26 float supd_x

start units-per-degree x

27.8.2.27 float supd_y

start units-per-degree y

27.8.2.28 float svel

start velocity

27.8.2.29 UINT32 time

effective time of event

27.8.2.30 INT16 type

event type

27.9 FSAMPLE Struct Reference

27.9.1 Detailed Description

Floating-point sample.

The EyeLink tracker measures eye position 250, 500, 1000 or 2000 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 eyelink_get_float_data() or eyelink_-newest_float_sample().

If sample rate is 2000hz, two samples with same time stamp possible. If SAMPLE_ADD_OFFSET is set on the flags, add .5 ms to get the real time. Convenient FLOAT_TIME can also be used.

Data Fields

- UINT32 time
- INT16 type
- UINT16 flags
- float px [2]
- float py [2]
- float hx [2]
- float hy [2]
- float pa [2]
- float gx [2]
- float gy [2]
- float rx
- float ry
- UINT16 status
- UINT16 input
- UINT16 buttons
- INT16 htype
- INT16 hdata [8]

27.9.2 Field Documentation

27.9.2.1 UINT16 buttons

button state & changes

27.9.2.2 UINT16 flags

flags to indicate contents

27.9.2.3 float gx[2]

screen gaze x

27.9.2.4 float gy[2]

screen gaze y

27.9.2.5 INT16 hdata[8]

head-tracker data (not prescaled)

27.9.2.6 INT16 htype

head-tracker data type (0=none)

27.9.2.7 float hx[2]

headref x

27.9.2.8 float hy[2]

headref y

27.9.2.9 UINT16 input

extra (input word)

27.9.2.10 float pa[2]

pupil size or area

27.9.2.11 float px[2]

pupil x

27.9.2.12 float py[2]

pupil y

27.9.2.13 float rx

screen pixels per degree

27.9.2.14 float ry

screen pixels per degree

27.9.2.15 UINT16 status

tracker status flags

27.9.2.16 UINT32 time

time of sample

27.9.2.17 INT16 type

always SAMPLE_TYPE

27.10 HOOKFCNS Struct Reference

27.10.1 Detailed Description

Structure used set and get callback functions.

Structure used set and get callback functions so that the calibration graphics can be drawn.

Data Fields

- INT16(∗ setup_cal_display_hook)(void)
- void(∗ exit_cal_display_hook)(void)
- void(∗ record_abort_hide_hook)(void)
- INT16(∗ setup_image_display_hook)(INT16 width, INT16 height)
- void(∗ image_title_hook)(INT16 threshold, char ∗cam_name)
- void(∗ draw_image_line_hook)(INT16 width, INT16 line, INT16 totlines, byte ∗pixels)
- void(∗ set_image_palette_hook)(INT16 ncolors, byte r[ ], byte g[ ], byte b[ ])
- void(∗ exit_image_display_hook)(void)
- void(∗ clear_cal_display_hook)()
- void(∗ erase_cal_target_hook)()
- void(∗ draw_cal_target_hook)(INT16 x, INT16 y)
- void(∗ cal_target_beep_hook)(void)
- void(∗ cal_done_beep_hook)(INT16 error)
- void(∗ dc_done_beep_hook)(INT16 error)
- void(∗ dc_target_beep_hook)(void)
- short(∗ get_input_key_hook)(InputEvent ∗event)
- void(∗ alert_printf_hook)(const char ∗)

27.10.2 Field Documentation

27.10.2.1 void(∗ alert_printf_hook)(const char ∗)

This function is called whenever alert_printf is called. In windows, if no callback is set calls MessageBox function. In other platforms, if no call back is set prints out to console.

27.10.2.2 void(∗ cal_done_beep_hook)(INT16 error)

This function is called to signal end of calibration.

Parameters:

error if non zero, then the calibration has error.

27.10.2.3 void(∗ cal_target_beep_hook)(void)

This function is called to signal new target.

27.10.2.4 void(∗ clear_cal_display_hook)()

Called to clear the calibration display.

27.10.2.5 void(* dc_done_beep_hook)(INT16 error)

This function is called to singnal the end of drift correct.

Parameters:

error if non zero, then the drift correction failed.

27.10.2.6 void(* dc_target_beep_hook)(void)

This function is called to signal a new drift correct target.

27.10.2.7 void(* draw_cal_target_hook)(INT16 x, INT16 y)

This function is responsible for the drawing of the target for calibration,validation and drift correct at the given coordinate.

Parameters:

x x coordinate of the target.

y y coordinate of the target.

Remarks:

The x and y are relative to what is sent to the tracker for the command screen_pixel_coords.

27.10.2.8 void(* draw_image_line_hook)(INT16 width, INT16 line, INT16 totlines, byte *pixels)

This function is called to supply the image line by line from top to bottom.

Parameters:

width width of the picture. Essentially, number of bytes in `pixels`.

line current line of the image

totlines total number of lines in the image. This will always equal the height of the image.

pixels pixel data.

Eg. Say we want to extract pixel at position (20,20) and print it out as rgb values.

```
if(line == 20) // y = 20
    {
            byte pix = pixels[19];
            // Note the r,g,b arrays come from the call to set_image_palette
            printf("RGB %d %d %d\n",r[pix],g[pix],b[pix]);
    }
```

Remarks:

certain display draw the image up side down. eg. GDI.

27.10.2.9 void(* erase_cal_target_hook)()

This function is responsible for erasing the target that was drawn by the last call to draw_cal_target.

27.10.2.10 void(* exit_cal_display_hook)(void)

This is called to release any resources that are not required beyond calibration. Beyond this call, no calibration functions will be called.

27.10.2.11 void(* exit_image_display_hook)(void)

This is called to notify that all camera setup things are complete. Any resources that are allocated in setup_image_display can be released in this function.

27.10.2.12 short(* get_input_key_hook)(InputEvent *event)

This is called to check for keyboard input. In this function:

- check if there are any input events
- if there are input events, fill key_input and return 1. otherwise return 0. If 1 is returned this will be called again to check for more events.

Parameters:

→ ***key_input*** fill in the InputEvent structure to return key,modifier values.

Returns:

if there is a key, return 1 otherwise return 0.

Remarks:

Special keys and modifiers should match the following code Special keys:

```
#define F1_KEY     0x3B00
#define F2_KEY     0x3C00
#define F3_KEY     0x3D00
#define F4_KEY     0x3E00
#define F5_KEY     0x3F00
#define F6_KEY     0x4000
#define F7_KEY     0x4100
#define F8_KEY     0x4200
#define F9_KEY     0x4300
#define F10_KEY    0x4400

#define PAGE_UP     0x4900
#define PAGE_DOWN   0x5100
#define CURS_UP     0x4800
#define CURS_DOWN   0x5000
#define CURS_LEFT   0x4B00
#define CURS_RIGHT  0x4D00

#define ESC_KEY    0x001B
#define ENTER_KEY  0x000D
```

Modifier: If you are using SDL you do not need to modify the modifier value as they match the value.

```
#define ELKMOD_NONE   0x0000
#define ELKMOD_LSHIFT 0x0001
#define ELKMOD_RSHIFT 0x0002
#define ELKMOD_LCTRL  0x0040
#define ELKMOD_RCTRL  0x0080
#define ELKMOD_LALT   0x0100
#define ELKMOD_RALT   0x0200
#define ELKMOD_LMETA  0x0400
```

```
#define ELKMOD_RMETA  0x0800,
#define ELKMOD_NUM    0x1000
#define ELKMOD_CAPS   0x2000
#define ELKMOD_MODE   0x4000
```

27.10.2.13 void(∗ image_title_hook)(INT16 threshold, char ∗cam_name)

This function is called to update any image title change.

Parameters:

threshold if -1 the entire tile is in the title string otherwise, the threshold of the current image.

title if threshold is -1, the title contains the whole title for the image. Otherwise only the camera name is given.

27.10.2.14 void(∗ record_abort_hide_hook)(void)

This function is called if abort of record. It is used to hide display from subject.

27.10.2.15 void(∗ set_image_palette_hook)(INT16 ncolors, byte r[ ], byte g[ ], byte b[ ])

This function is called after setup_image_display and before the first call to draw_image_line. This is responsible to setup the palettes to display the camera image.

Parameters:

ncolors number of colors in the palette.

r red component of rgb.

g blue component of rgb.

b green component of rgb.

27.10.2.16 INT16(∗ setup_cal_display_hook)(void)

This function is called to setup calibration/validation display. This function called before any calibration routines are called.

27.10.2.17 INT16(∗ setup_image_display_hook)(INT16 width, INT16 height)

This function is responsible for initializing any resources that are required for camera setup.

Parameters:

width width of the source image to expect.

height height of the source image to expect.

Returns:

-1 if failed, 0 otherwise.

27.11 IEVENT Struct Reference

27.11.1 Detailed Description

Integer eye-movement events.

Data Fields

- UINT32 time
- INT16 type
- UINT16 read
- INT16 eye
- UINT32 sttime
- UINT32 entime
- INT16 hstx
- INT16 hsty
- INT16 gstx
- INT16 gsty
- UINT16 **sta**
- INT16 henx
- INT16 heny
- INT16 genx
- INT16 geny
- UINT16 **ena**
- INT16 havx
- INT16 havy
- INT16 gavx
- INT16 gavy
- UINT16 ava
- INT16 avel
- INT16 pvel
- INT16 svel
- INT16 evel
- INT16 supd_x
- INT16 eupd_x
- INT16 supd_y
- INT16 eupd_y
- UINT16 status

27.11.2 Field Documentation

27.11.2.1 UINT16 ava

also used as accumulator

27.11.2.2 INT16 avel

avg velocity accum

27.11.2.3 UINT32 entime

end times

27.11.2.4 INT16 eupd_x

end units-per-degree y

27.11.2.5 INT16 eupd_y

end units-per-degree y

27.11.2.6 INT16 evel

end velocity

27.11.2.7 INT16 eye

eye: 0=left,1=right

27.11.2.8 INT16 gavx

average x

27.11.2.9 INT16 gavy

average y

27.11.2.10 INT16 genx

ending point x

27.11.2.11 INT16 geny

ending point y

27.11.2.12 INT16 gstx

starting point x

27.11.2.13 INT16 gsty

starting point y

27.11.2.14 INT16 havx

average x

27.11.2.15 INT16 havy

average y

27.11.2.16 INT16 henx

ending point x

27.11.2.17 INT16 heny

ending point y

27.11.2.18 INT16 hstx

starting point x

27.11.2.19 INT16 hsty

starting point y

27.11.2.20 INT16 pvel

peak velocity accum

27.11.2.21 UINT16 read

flags which items were included

27.11.2.22 UINT16 status

error, warning flags

27.11.2.23 UINT32 sttime

start times

27.11.2.24 INT16 supd_x

start units-per-degree x

27.11.2.25 INT16 supd_y

start units-per-degree y

27.11.2.26 INT16 svel

start velocity

27.11.2.27 UINT32 time

effective time of event

27.11.2.28 INT16 type

event type

27.12 ILINKDATA Struct Reference

27.12.1 Detailed Description

Class to represent tracker status.

Class to represent tracker status information such as time stamps, flags, tracker addresses and so on.

Data Fields

- UINT32 time
- UINT32 version
- UINT16 samrate
- UINT16 samdiv
- UINT16 prescaler
- UINT16 vprescaler
- UINT16 pprescaler
- UINT16 hprescaler
- UINT16 sample_data
- UINT16 event_data
- UINT16 event_types
- byte in_sample_block
- byte in_event_block
- byte have_left_eye
- byte have_right_eye
- UINT16 last_data_gap_types
- UINT16 last_data_buffer_type
- UINT16 last_data_buffer_size
- UINT16 control_read
- UINT16 first_in_block
- UINT32 last_data_item_time
- UINT16 last_data_item_type
- UINT16 last_data_item_contents
- ALL_DATA last_data_item
- UINT32 block_number
- UINT32 block_sample
- UINT32 block_event
- UINT16 last_resx
- UINT16 last_resy
- UINT16 last_pupil [2]
- UINT16 last_status
- UINT16 queued_samples
- UINT16 queued_events
- UINT16 queue_size
- UINT16 queue_free
- UINT32 last_rcve_time
- byte samples_on
- byte **events_on**
- UINT16 packet_flags
- UINT16 link_flags

- UINT16 state_flags
- byte link_dstatus
- byte link_pendcmd
- UINT16 reserved
- char our_name [40]
- ELINKADDR **our_address**
- char eye_name [40]
- ELINKADDR **eye_address**
- ELINKADDR ebroadcast_address
- ELINKADDR rbroadcast_address
- UINT16 polling_remotes
- UINT16 poll_responses
- ELINKNODE nodes [4]

27.12.2 Field Documentation

27.12.2.1 UINT32 block_event

events (excl. control read in block

27.12.2.2 UINT32 block_number

block in file

27.12.2.3 UINT32 block_sample

samples read in block so far

27.12.2.4 UINT16 control_read

set if control event read with last data

27.12.2.5 ELINKADDR ebroadcast_address

Broadcast address for eye trackers

27.12.2.6 UINT16 event_data

0 if off, else all flags

27.12.2.7 UINT16 event_types

0 if off, else event-type flags

27.12.2.8 char eye_name[40]

name of tracker connected to

27.12.2.9 UINT16 first_in_block

set if control event started new block

27.12.2.10 byte have_left_eye

set if any left-eye data expected

27.12.2.11 byte have_right_eye

set if any right-eye data expected

27.12.2.12 UINT16 hprescaler

head-distance prescale (to mm)

27.12.2.13 byte in_event_block

set if in block with events

27.12.2.14 byte in_sample_block

set if in block with samples

27.12.2.15 UINT16 last_data_buffer_size

buffer size of last item

27.12.2.16 UINT16 last_data_buffer_type

buffer-type code

27.12.2.17 UINT16 last_data_gap_types

flags what we lost before last item

27.12.2.18 ALL_DATA last_data_item

buffer containing last item

27.12.2.19 UINT16 last_data_item_contents

content: <read> (IEVENT), <flags> (ISAMPLE)

27.12.2.20 UINT32 last_data_item_time

time field of item

27.12.2.21 UINT16 last_data_item_type

type: 100=sample, 0=none, else event type

27.12.2.22 UINT16 last_pupil[2]

updated by samples only

27.12.2.23 UINT32 last_rcve_time

time tracker last sent packet

27.12.2.24 UINT16 last_resx

updated by samples only

27.12.2.25 UINT16 last_resy

updated by samples only

27.12.2.26 UINT16 last_status

updated by samples, events

27.12.2.27 byte link_dstatus

tracker data output state

27.12.2.28 UINT16 link_flags

status flags from link packet header

27.12.2.29 byte link_pendcmd

tracker commands pending

27.12.2.30 ELINKNODE nodes[4]

data on nodes

27.12.2.31 char our_name[40]

a name for our machine

27.12.2.32 UINT16 packet_flags

status flags from data packet

27.12.2.33 UINT16 poll_responses

total nodes responding to polling

27.12.2.34 UINT16 polling_remotes

1 if polling remotes, else polling trackers

27.12.2.35 UINT16 pprescaler

pupil prescale (1 if area, greater if diameter)

27.12.2.36 UINT16 prescaler

amount to divide gaze x,y,res by

27.12.2.37 UINT16 queue_free

unused bytes in queue

27.12.2.38 UINT16 queue_size

total queue buffer size

27.12.2.39 UINT16 queued_events

includes control events

27.12.2.40 UINT16 queued_samples

number of items in queue

27.12.2.41 ELINKADDR rbroadcast_address

Broadcast address for remotes

27.12.2.42 UINT16 reserved

0 for EyeLink I or original EyeLink API DLL. EYELINK II ONLY: MSB set if read crmode<<8 + file_-filter<<4 + link_filter crmode = 0 if pupil, else pupil-CR file_filter, link_filter: 0, 1, or 2 for level of heuristic filter applied

27.12.2.43 UINT16 samdiv

sample "divisor" (min msec between samples)

27.12.2.44 UINT16 sample_data

0 if off, else all flags

27.12.2.45 byte samples_on

data type rcve enable (switch)

27.12.2.46 UINT16 samrate

10∗sample rate (0 if no samples, 1 if nonconstant)

27.12.2.47 UINT16 state_flags

tracker error state flags

27.12.2.48 UINT32 time

time of last control event

27.12.2.49 UINT32 version

structure version

27.12.2.50 UINT16 vprescaler

amount to divide velocity by

27.13 IMESSAGE Struct Reference

27.13.1 Detailed Description

Message events: usually text but may contain binary data.

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. Although this method might be used to determine the tracker time (the time field of a message event will indicate when the message was received by the tracker), the use of eyelink_request_time() and eyelink_read_time() is more efficient for retrieving the current time from the eye tracker's timestamp clock. The eye tracker time is rarely needed in any case, and would only be useful to compute link transport delays.

Data Fields

- UINT32 time
- INT16 type
- UINT16 length
- byte text [260]

27.13.2 Field Documentation

27.13.2.1 UINT16 length

length of message

27.13.2.2 byte text[260]

message contents (max length 255)

27.13.2.3 UINT32 time

time message logged

27.13.2.4 INT16 type

event type: usually MESSAGEEVENT

27.14 InputEvent Union Reference

27.14.1 Detailed Description

Union of all input types.

A union that is used by the callback function get_input_key_hook, to collect input data. At the moment, it is meant to collect only keyboard inputs. In the future this may be used to support mouse inputs as well.

Data Fields

- byte **type**
- KeyInput **key**
- MouseMotionEvent **motion**
- MouseButtonEvent **button**

27.15 IOEVENT Struct Reference

27.15.1 Detailed Description

Button, input, other simple events.

BUTTONEVENT and INPUTEVENT types are the simplest events, reporting changes in button status or in the input port data. The time field records the timestamp of the eye-data sample where the change occurred, although the event itself is usually sent before that sample. The data field contains the data after the change, in the same format as in the FSAMPLE structure.

Button events from the link are rarely used; monitoring buttons with one of eyelink_read_keybutton(), eyelink_last_button_press(), or eyelink_button_states() is preferable, since these can report button states at any time, not just during recording.

Data Fields

- UINT32 time
- INT16 type
- UINT16 data

27.15.2 Field Documentation

27.15.2.1 UINT16 data

coded event data

27.15.2.2 UINT32 time

time logged

27.15.2.3 INT16 type

event type:

27.16 ISAMPLE Struct Reference

27.16.1 Detailed Description

Integer sample data.

Data Fields

- UINT32 time
- INT16 type
- UINT16 flags
- INT16 px [2]
- INT16 py [2]
- INT16 hx [2]
- INT16 hy [2]
- UINT16 pa [2]
- INT16 gx [2]
- INT16 gy [2]
- INT16 rx
- INT16 ry
- UINT16 status
- UINT16 input
- UINT16 buttons
- INT16 htype
- INT16 hdata [8]

27.16.2 Field Documentation

27.16.2.1 UINT16 buttons

button state & changes

27.16.2.2 UINT16 flags

flags to indicate contents

27.16.2.3 INT16 gx[2]

screen gaze x

27.16.2.4 INT16 gy[2]

screen gaze y

27.16.2.5 INT16 hdata[8]

head-tracker data

27.16.2.6 INT16 htype

head-tracker data type (0=none)

27.16.2.7 INT16 hx[2]

headref x

27.16.2.8 INT16 hy[2]

headref y

27.16.2.9 UINT16 input

extra (input word)

27.16.2.10 UINT16 pa[2]

pupil size or area

27.16.2.11 INT16 px[2]

pupil x

27.16.2.12 INT16 py[2]

pupil y

27.16.2.13 INT16 rx

screen pixels per degree

27.16.2.14 INT16 ry

screen pixels per degree

27.16.2.15 UINT16 status

tracker status flags

27.16.2.16 UINT32 time

time of sample

27.16.2.17 INT16 type

always SAMPLE_TYPE

27.17 KeyInput Struct Reference

27.17.1 Detailed Description

Keyboard input event structure.

Data Fields

- byte type
- byte state
- UINT16 key
- UINT16 modifier
- UINT16 unicode

27.17.2 Field Documentation

27.17.2.1 UINT16 key

keys

27.17.2.2 UINT16 modifier

modifier

27.17.2.3 byte state

KEYDOWN = 1 or KEYUP = 0

27.17.2.4 byte type

The value of type should have value of KEYINPUT_EVENT

27.17.2.5 UINT16 unicode

unicode character value of the key

27.18 MouseButtonEvent Struct Reference

27.18.1 Detailed Description

Mouse button event structure (For future).

Data Fields

- byte type
- byte which
- byte button
- byte state
- UINT16 **x**
- UINT16 y

27.18.2 Field Documentation

27.18.2.1 byte button

The mouse button index

27.18.2.2 byte state

BUTTONDOWN = 0 or BUTTONUP = 1

27.18.2.3 byte type

MOUSE_BUTTON_INPUT_EVENT

27.18.2.4 byte which

The mouse device index

27.18.2.5 UINT16 y

The X/Y coordinates of the mouse at press time

27.19 MouseMotionEvent Struct Reference

27.19.1 Detailed Description

Mouse motion event structure (For future).

Data Fields

- byte type
- byte which
- byte state
- UINT16 **x**
- UINT16 y
- UINT16 xrel
- UINT16 yrel

27.19.2 Field Documentation

27.19.2.1 byte state

The current button state

27.19.2.2 byte type

MOUSE_MOTION_INPUT_EVENT

27.19.2.3 byte which

The mouse device index

27.19.2.4 UINT16 xrel

The relative motion in the X direction

27.19.2.5 UINT16 y

The X/Y coordinates of the mouse

27.19.2.6 UINT16 yrel

The relative motion in the Y direction

Chapter 28

File List

28.1 core_expt.h File Reference

28.1.1 Detailed Description

Declarations of eyelink_core functions and types.

This file will also reference the other EyeLink header files.

Data Structures

- struct DISPLAYINFO

  This structure holds information on the display.

- struct EYECOLOR

  Represents an RGB color.

- struct EYEPALETTE

  Represents a palette index.

- struct EYEPIXELFORMAT

  Represents pixel format of an image or surface.

- struct EYEBITMAP

  Represents a bitmap image.

- struct KeyInput

  Keyboard input event structure.

- struct MouseMotionEvent

  Mouse motion event structure (For future).

- struct MouseButtonEvent

  Mouse button event structure (For future).

- union InputEvent

  Union of all input types.

- struct HOOKFCNS

  Structure used set and get callback functions.

- struct HOOKFCNS2

  Structure used set and get callback functions.

- struct _CrossHairInfo

  Structure to hold cross hair information.

Defines

- #define CURS_UP 0x4800
- #define CURS_DOWN 0x5000
- #define CURS_LEFT 0x4B00
- #define CURS_RIGHT 0x4D00
- #define ESC_KEY 0x001B
- #define ENTER_KEY 0x000D
- #define PAGE_UP 0x4900
- #define PAGE_DOWN 0x5100
- #define JUNK_KEY 1
- #define TERMINATE_KEY 0x7FFF
- #define DONE_TRIAL 0
- #define TRIAL_OK 0
- #define REPEAT_TRIAL 1
- #define SKIP_TRIAL 2
- #define ABORT_EXPT 3
- #define TRIAL_ERROR -1
- #define BX_AVERAGE 0
- #define BX_DARKEN 1
- #define BX_LIGHTEN 2
- #define BX_MAXCONTRAST 4
- #define BX_NODITHER 8
- #define BX_GRAYSCALE 16
- #define BX_DOTRANSFER 256
- #define SV_NOREPLACE 1
- #define SV_MAKEPATH 2
- #define BAD_FILENAME -2222
- #define BAD_ARGUMENT -2223
- #define KEYINPUT_EVENT 0x1
- #define MOUSE_INPUT_EVENT 0x4
- #define MOUSE_MOTION_INPUT_EVENT 0x5
- #define MOUSE_BUTTON_INPUT_EVENT 0x6
- #define ELKMOD_NONE 0x0000
- #define ELKMOD_LSHIFT 0x0001
- #define ELKMOD_RSHIFT 0x0002
- #define ELKMOD_LCTRL 0x0040

- #define ELKMOD_RCTRL 0x0080
- #define ELKMOD_LALT 0x0100
- #define ELKMOD_RALT 0x0200
- #define ELKMOD_LMETA 0x0400
- #define ELKMOD_RMETA 0x0800
- #define ELKMOD_NUM 0x1000
- #define ELKMOD_CAPS 0x2000
- #define ELKMOD_MODE 0x4000
- #define **ELKEY_DOWN** 1
- #define **ELKEY_UP** 0
- #define **FIVE_SAMPLE_MODEL** 1
- #define **NINE_SAMPLE_MODEL** 2
- #define **SEVENTEEN_SAMPLE_MODEL** 3
- #define **EL1000_TRACKER_MODEL** 4
- #define **CR_HAIR_COLOR** 1
- #define **PUPIL_HAIR_COLOR** 2
- #define **PUPIL_BOX_COLOR** 3
- #define **SEARCH_LIMIT_BOX_COLOR** 4
- #define **MOUSE_CURSOR_COLOR** 5

Typedefs

- typedef enum _EL_CAL_BEEP EL_CAL_BEEP

  Enum used for calibration beeps.

- typedef _CrossHairInfo **CrossHairInfo**

Enumerations

- enum **IMAGETYPE** {

  JPEG, **PNG**, **GIF**, **BMP**,

  XPM }
- enum _EL_CAL_BEEP {

  EL_DC_DONE_ERR_BEEP = -2, EL_CAL_DONE_ERR_BEEP = -1, EL_CAL_DONE_GOOD_-BEEP = 0, EL_CAL_TARG_BEEP = 1,

  EL_DC_DONE_GOOD_BEEP = 2, EL_DC_TARG_BEEP = 3 }

  Enum used for calibration beeps.

Functions

- INT16 open_eyelink_connection (INT16 mode)
- void close_eyelink_connection (void)
- INT16 set_eyelink_address (char ∗addr)
- INT32 set_application_priority (INT32 priority)
- INT16 message_pump ()
- INT16 key_message_pump (void)
- void pump_delay (UINT32 delay)

- void flush_getkey_queue (void)
- UINT16 read_getkey_queue (void)
- UINT16 echo_key (void)
- UINT16 getkey (void)
- UINT32 getkey_with_mod (UINT16 ∗unicode)
- INT16 eyecmd_printf (const char ∗fmt,...)
- INT16 eyemsg_printf (const char ∗fmt,...)
- INT16 eyemsg_printf_ex (UINT32 exectime, const char ∗fmt,...)
- INT16 start_recording (INT16 file_samples, INT16 file_events, INT16 link_samples, INT16 link_-events)
- INT16 check_recording (void)
- void stop_recording (void)
- void set_offline_mode (void)
- INT16 check_record_exit (void)
- void exit_calibration (void)
- INT16 do_tracker_setup (void)
- INT16 do_drift_correct (INT16 x, INT16 y, INT16 draw, INT16 allow_setup)
- INT16 do_drift_correctf (float x, float y, INT16 draw, INT16 allow_setup)
- INT16 target_mode_display (void)
- INT16 image_mode_display (void)
- void alert_printf (char ∗fmt,...)
- INT32 receive_data_file (char ∗src, char ∗dest, INT16 dest_is_path)
- INT32 receive_data_file_feedback (char ∗src, char ∗dest, INT16 dest_is_path, void(∗progress)(unsigned int size, unsigned int received))
- INT16 open_data_file (char ∗name)
- INT16 close_data_file (void)
- INT16 escape_pressed (void)
- INT16 break_pressed (void)
- void terminal_break (INT16 assert)
- INT16 **application_terminated** (void)
- void begin_realtime_mode (UINT32 delay)
- void end_realtime_mode (void)
- void set_high_priority (void)
- void set_normal_priority (void)
- INT32 in_realtime_mode (void)
- void **eyelink_enable_extended_realtime** (void)
- char ∗ eyelink_get_error (int id, char ∗function_name)
- void splice_fname (char ∗fname, char ∗path, char ∗ffname)
- int check_filename_characters (char ∗name)
- int file_exists (char ∗path)
- int create_path (char ∗path, INT16 create, INT16 is_dir)
- int el_bitmap_save_and_backdrop (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 xferoptions)
- int el_bitmap_to_backdrop (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 xferoptions)
- int el_bitmap_save (EYEBITMAP ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- void setup_graphic_hook_functions (HOOKFCNS ∗hooks)
- HOOKFCNS ∗ get_all_hook_functions ()
- INT16 setup_graphic_hook_functions_V2 (HOOKFCNS2 ∗hooks)

- HOOKFCNS2 ∗ get_all_hook_functions_V2 ()
- int set_write_image_hook (int(∗hookfn)(char ∗outfilename, int format, EYEBITMAP ∗bitmap), int options)
- int **eyelink_peep_input_event** (InputEvent ∗event, int mask)
- int **eyelink_get_input_event** (InputEvent ∗event, int mask)
- int **eyelink_peep_last_input_event** (InputEvent ∗event, int mask)
- void **eyelink_flush_input_event** ()
- INT32 eyelink_initialize_mapping (float left, float top, float right, float bottom)
- INT32 eyelink_href_to_gaze (float ∗xp, float ∗yp, FSAMPLE ∗sample)
- INT32 eyelink_gaze_to_href (float ∗xp, float ∗yp, FSAMPLE ∗sample)
- float eyelink_href_angle (float x1, float y1, float x2, float y2)
- void eyelink_href_resolution (float x, float y, float ∗xres, float ∗yres)
- int get_image_xhair_data (INT16 x[4], INT16 y[4], INT16 ∗xhairs_on)
- int **eyelink_get_extra_raw_values** (FSAMPLE ∗s, FSAMPLE_RAW ∗rv)
- int **eyelink_get_extra_raw_values_v2** (FSAMPLE ∗s, int eye, FSAMPLE_RAW ∗rv)
- int eyelink_calculate_velocity_x_y (int slen, float xvel[2], float yvel[2], FSAMPLE ∗vel_sample)
- int eyelink_calculate_velocity (int slen, float vel[2], FSAMPLE ∗vel_sample)
- int eyelink_calculate_overallvelocity_and_acceleration (int slen, float vel[2], float acc[2], FSAMPLE ∗vel_sample)
- INT16 timemsg_printf (UINT32 t, char ∗fmt,...)
- int open_message_file (char ∗fname)
- void close_message_file (void)
- INT32 eyelink_draw_cross_hair (CrossHairInfo ∗chi)
- void eyelink_dll_version (char FARTYPE ∗c)

28.1.2 Function Documentation

28.1.2.1 void eyelink_dll_version (char FARTYPE ∗ *c*)

Returns the eyelink_core library version number.

Parameters:

→ ***c*** fills in the version number of th eyelink_core library.

28.2 eye_data.h File Reference

28.2.1 Detailed Description

Declaration of complex EyeLink data types and link data structures.

Data Structures

- struct ISAMPLE

  Integer sample data.

- struct FSAMPLE

  Floating-point sample.

- struct DSAMPLE

  Floating-point sample with floating point time.

- struct **FSAMPLE_RAW**
- struct IEVENT

  Integer eye-movement events.

- struct FEVENT

  Floating-point eye event.

- struct DEVENT

  Floating-point eye event with floating point time.

- struct IMESSAGE

  Message events: usually text but may contain binary data.

- struct DMESSAGE

  Message events: usually text but may contain binary data with floating point time.

- struct IOEVENT

  Button, input, other simple events.

- struct DIOEVENT

  Button, input, other simple events with floating point time.

- union ALL_DATA

  Union of message, io event and integer sample and integer event.

- union ALLF_DATA

  Union of message, io event and float sample and float event.

- union ALLD_DATA

  Union of message, io event and double sample and double event.

- struct ELINKNODE

Name and address for connection.

- struct ILINKDATA

  Class to represent tracker status.

Defines

- #define MISSING_DATA -32768
- #define MISSING -32768
- #define INaN -32768
- #define LEFT_EYE 0
- #define RIGHT_EYE 1
- #define LEFTEYEI 0
- #define RIGHTEYEI 1
- #define LEFT 0
- #define RIGHT 1
- #define BINOCULAR 2
- #define FLOAT_TIME(x) (((double)((x) $\rightarrow$ time)) + (((x) $\rightarrow$ type==SAMPLE_TYPE && (x) $\rightarrow$ flags & SAMPLE_ADD_OFFSET)?0.5:0.0))
- #define SAMPLE_LEFT 0x8000
- #define SAMPLE_RIGHT 0x4000
- #define SAMPLE_TIMESTAMP 0x2000
- #define SAMPLE_PUPILXY 0x1000
- #define SAMPLE_HREFXY 0x0800
- #define SAMPLE_GAZEXY 0x0400
- #define SAMPLE_GAZERES 0x0200
- #define SAMPLE_PUPILSIZE 0x0100
- #define SAMPLE_STATUS 0x0080
- #define SAMPLE_INPUTS 0x0040
- #define SAMPLE_BUTTONS 0x0020
- #define SAMPLE_HEADPOS 0x0010
- #define SAMPLE_TAGGED 0x0008
- #define SAMPLE_UTAGGED 0x0004
- #define SAMPLE_ADD_OFFSET 0x0002
- #define **FSAMPLEDEF** 1
- #define **DSAMPLEDEF** 1
- #define **FSAMPLERAWDEF** 1
- #define **FEVENTDEF** 1
- #define **DEVENTDEF** 1
- #define SAMPLE_TYPE 200
- #define **STARTPARSE** 1
- #define **ENDPARSE** 2
- #define **BREAKPARSE** 10
- #define STARTBLINK 3
- #define ENDBLINK 4
- #define STARTSACC 5
- #define ENDSACC 6
- #define STARTFIX 7
- #define ENDFIX 8

- #define FIXUPDATE 9
- #define STARTSAMPLES 15
- #define ENDSAMPLES 16
- #define STARTEVENTS 17
- #define ENDEVENTS 18
- #define MESSAGEEVENT 24
- #define BUTTONEVENT 25
- #define INPUTEVENT 28
- #define LOST_DATA_EVENT 0x3F
- #define **ISAMPLE_BUFFER** SAMPLE_TYPE
- #define **IEVENT_BUFFER** 66
- #define **IOEVENT_BUFFER** 8
- #define **IMESSAGE_BUFFER** 250
- #define **CONTROL_BUFFER** 36
- #define **ILINKDATA_BUFFER** CONTROL_BUFFER
- #define READ_ENDTIME 0x0040
- #define READ_GRES 0x0200
- #define READ_SIZE 0x0080
- #define READ_VEL 0x0100
- #define READ_STATUS 0x2000
- #define READ_BEG 0x0001
- #define READ_END 0x0002
- #define READ_AVG 0x0004
- #define READ_PUPILXY 0x0400
- #define **READ_HREFXY** 0x0800
- #define **READ_GAZEXY** 0x1000
- #define READ_BEGPOS 0x0008
- #define **READ_ENDPOS** 0x0010
- #define **READ_AVGPOS** 0x0020
- #define FRIGHTEYE_EVENTS 0x8000
- #define FLEFTEYE_EVENTS 0x4000
- #define LEFTEYE_EVENTS 0x8000
- #define RIGHTEYE_EVENTS 0x4000
- #define BLINK_EVENTS 0x2000
- #define FIXATION_EVENTS 0x1000
- #define FIXUPDATE_EVENTS 0x0800
- #define SACCADE_EVENTS 0x0400
- #define MESSAGE_EVENTS 0x0200
- #define BUTTON_EVENTS 0x0040
- #define INPUT_EVENTS 0x0020
- #define EVENT_VELOCITY 0x8000
- #define EVENT_PUPILSIZE 0x4000
- #define EVENT_GAZERES 0x2000
- #define EVENT_STATUS 0x1000
- #define EVENT_GAZEXY 0x0400
- #define EVENT_HREFXY 0x0200
- #define EVENT_PUPILXY 0x0100
- #define FIX_AVG_ONLY 0x0008
- #define START_TIME_ONLY 0x0004
- #define PARSEDBY_GAZE 0x00C0

- #define PARSEDBY_HREF 0x0080
- #define PARSEDBY_PUPIL 0x0040
- #define **ILINKDATAVERSION** 2
- #define ELNAMESIZE 40
- #define ELREMBUFSIZE 420
- #define ELINKADDRSIZE 16
- #define PUPIL_DIA_FLAG 0x0001
- #define HAVE_SAMPLES_FLAG 0x0002
- #define HAVE_EVENTS_FLAG 0x0004
- #define HAVE_LEFT_FLAG 0x8000
- #define HAVE_RIGHT_FLAG 0x4000
- #define **DROPPED_SAMPLE** 0x8000
- #define **DROPPED_EVENT** 0x4000
- #define **DROPPED_CONTROL** 0x2000
- #define DFILE_IS_OPEN 0x80
- #define DFILE_EVENTS_ON 0x40
- #define DFILE_SAMPLES_ON 0x20
- #define DLINK_EVENTS_ON 0x08
- #define DLINK_SAMPLES_ON 0x04
- #define DRECORD_ACTIVE 0x01
- #define COMMAND_FULL_WARN 0x01
- #define MESSAGE_FULL_WARN 0x02
- #define LINK_FULL_WARN 0x04
- #define FULL_WARN 0x0F
- #define LINK_CONNECTED 0x10
- #define LINK_BROADCAST 0x20
- #define LINK_IS_TCPIP 0x40
- #define LED_TOP_WARNING 0x0080
- #define LED_BOT_WARNING 0x0040
- #define LED_LEFT_WARNING 0x0020
- #define LED_RIGHT_WARNING 0x0010
- #define HEAD_POSITION_WARNING 0x00F0
- #define LED_EXTRA_WARNING 0x0008
- #define LED_MISSING_WARNING 0x0004
- #define HEAD_VELOCITY_WARNING 0x0001
- #define CALIBRATION_AREA_WARNING 0x0002
- #define MATH_ERROR_WARNING 0x2000
- #define INTERP_SAMPLE_WARNING 0x1000
- #define INTERP_PUPIL_WARNING 0x8000
- #define CR_WARNING 0x0F00
- #define CR_LEFT_WARNING 0x0500
- #define CR_RIGHT_WARNING 0x0A00
- #define CR_LOST_WARNING 0x0300
- #define CR_LOST_LEFT_WARNING 0x0100
- #define CR_LOST_RIGHT_WARNING 0x0200
- #define CR_RECOV_WARNING 0x0C00
- #define CR_RECOV_LEFT_WARNING 0x0400
- #define CR_RECOV_RIGHT_WARNING 0x0800
- #define HPOS_TOP_WARNING 0x0080
- #define HPOS_BOT_WARNING 0x0040

- #define HPOS_LEFT_WARNING 0x0020
- #define HPOS_RIGHT_WARNING 0x0010
- #define HPOS_WARNING 0x00F0
- #define HPOS_ANGLE_WARNING 0x0008
- #define HPOS_MISSING_WARNING 0x0004
- #define HPOS_DISTANCE_WARNING 0x0001
- #define TFLAG_MISSING 0x4000
- #define TFLAG_ANGLE 0x2000
- #define TFLAG_NEAREYE 0x1000
- #define TFLAG_CLOSE 0x0800
- #define **TFLAG_FAR** 0x0400
- #define TFLAG_T_TSIDE 0x0080
- #define TFLAG_T_BSIDE 0x0040
- #define TFLAG_T_LSIDE 0x0020
- #define TFLAG_T_RSIDE 0x0010
- #define TFLAG_E_TSIDE 0x0008
- #define TFLAG_E_BSIDE 0x0004
- #define TFLAG_E_LSIDE 0x0002
- #define TFLAG_E_RSIDE 0x0001

Typedefs

- typedef byte **ELINKADDR** [ELINKADDRSIZE]

28.2.2 Define Documentation

28.2.2.1 #define COMMAND_FULL_WARN 0x01

too many commands: pause

28.2.2.2 #define DFILE_EVENTS_ON 0x40

disk file writing events

28.2.2.3 #define DFILE_IS_OPEN 0x80

disk file active

28.2.2.4 #define DFILE_SAMPLES_ON 0x20

disk file writing samples

28.2.2.5 #define DLINK_EVENTS_ON 0x08

link sending events

28.2.2.6 #define DLINK_SAMPLES_ON 0x04

link sending samples

28.2.2.7 #define DRECORD_ACTIVE 0x01

in active recording mode

28.2.2.8 #define ELINKADDRSIZE 16

Node address (format varies)

28.2.2.9 #define ELNAMESIZE 40

max. tracker or remote name size

28.2.2.10 #define ELREMBUFSIZE 420

max. remote-to-remote message size

28.2.2.11 #define FULL_WARN 0x0F

test mask for any warning

28.2.2.12 #define HAVE_EVENTS_FLAG 0x0004

set if we have events

28.2.2.13 #define HAVE_LEFT_FLAG 0x8000

set if we have left-eye data

28.2.2.14 #define HAVE_RIGHT_FLAG 0x4000

set if we have right-eye data

28.2.2.15 #define HAVE_SAMPLES_FLAG 0x0002

set if we have samples

28.2.2.16 #define LINK_BROADCAST 0x20

link is broadcasting

28.2.2.17 #define LINK_CONNECTED 0x10

link is connected

28.2.2.18 #define LINK_FULL_WARN 0x04

link, command, or message load

28.2.2.19 #define LINK_IS_TCPIP 0x40

link is TCP/IP (else packet)

28.2.2.20 #define MESSAGE_FULL_WARN 0x02

too many messages: pause

28.2.2.21 #define PUPIL_DIA_FLAG 0x0001

set if pupil is diameter (else area)

28.3 eyelink.h File Reference

28.3.1 Detailed Description

Declarations and constants for basic EyeLink functions, Ethernet link, and timing.

Data Structures

- struct **IMAGE_PALDATA**

Defines

- #define OK_RESULT 0
- #define NO_REPLY 1000
- #define LINK_TERMINATED_RESULT -100
- #define ABORT_RESULT 27
- #define UNEXPECTED_EOL_RESULT -1
- #define SYNTAX_ERROR_RESULT -2
- #define BAD_VALUE_RESULT -3
- #define EXTRA_CHARACTERS_RESULT -4
- #define current_msec() current_time()
- #define LINK_INITIALIZE_FAILED -200
- #define CONNECT_TIMEOUT_FAILED -201
- #define WRONG_LINK_VERSION -202
- #define TRACKER_BUSY -203
- #define IN_DISCONNECT_MODE 16384
- #define IN_UNKNOWN_MODE 0
- #define IN_IDLE_MODE 1
- #define IN_SETUP_MODE 2
- #define IN_RECORD_MODE 4
- #define IN_TARGET_MODE 8
- #define IN_DRIFTCORR_MODE 16
- #define IN_IMAGE_MODE 32
- #define IN_USER_MENU 64
- #define IN_PLAYBACK_MODE 256
- #define **EL_IDLE_MODE** 1
- #define **EL_IMAGE_MODE** 2
- #define **EL_SETUP_MENU_MODE** 3
- #define **EL_USER_MENU_1** 5
- #define **EL_USER_MENU_2** 6
- #define **EL_USER_MENU_3** 7
- #define **EL_OPTIONS_MENU_MODE** 8
- #define **EL_OUTPUT_MENU_MODE** 9
- #define **EL_DEMO_MENU_MODE** 10
- #define **EL_CALIBRATE_MODE** 11
- #define **EL_VALIDATE_MODE** 12
- #define **EL_DRIFT_CORR_MODE** 13
- #define **EL_RECORD_MODE** 14
- #define **SCENECAM_ALIGN_MODE** 15

- #define **SCENECAM_DEPTH_MODE** 16
- #define **USER_MENU_NUMBER**(mode) ((mode)-4)
- #define **SAMPLE_TYPE** 200
- #define RECORD_FILE_SAMPLES 1
- #define RECORD_FILE_EVENTS 2
- #define RECORD_LINK_SAMPLES 4
- #define RECORD_LINK_EVENTS 8
- #define **ELIMAGE_2** 0
- #define **ELIMAGE_16** 1
- #define **ELIMAGE_16P** 2
- #define **ELIMAGE_256** 3
- #define **ELIMAGE_128HV** 4
- #define **ELIMAGE_128HVX** 5
- #define **KB_PRESS** 10
- #define **KB_RELEASE** -1
- #define **KB_REPEAT** 1
- #define **NUM_LOCK_ON** 0x20
- #define **CAPS_LOCK_ON** 0x40
- #define **ALT_KEY_DOWN** 0x08
- #define **CTRL_KEY_DOWN** 0x04
- #define **SHIFT_KEY_DOWN** 0x03
- #define **KB_BUTTON** 0xFF00U
- #define **F1_KEY** 0x3B00
- #define **F2_KEY** 0x3C00
- #define **F3_KEY** 0x3D00
- #define **F4_KEY** 0x3E00
- #define **F5_KEY** 0x3F00
- #define **F6_KEY** 0x4000
- #define **F7_KEY** 0x4100
- #define **F8_KEY** 0x4200
- #define **F9_KEY** 0x4300
- #define **F10_KEY** 0x4400
- #define **PAGE_UP** 0x4900
- #define **PAGE_DOWN** 0x5100
- #define **CURS_UP** 0x4800
- #define **CURS_DOWN** 0x5000
- #define **CURS_LEFT** 0x4B00
- #define **CURS_RIGHT** 0x4D00
- #define **ESC_KEY** 0x001B
- #define **ENTER_KEY** 0x000D
- #define **FILE_XFER_ABORTED** -110
- #define **FILE_CANT_OPEN** -111
- #define **FILE_NO_REPLY** -112
- #define **FILE_BAD_DATA** -113
- #define **FILEDATA_SIZE_FLAG** 999
- #define **FILE_BLOCK_SIZE** 512
- #define **ABORT_BX** -32000
- #define **PAUSE_BX** -32001
- #define **DONE_BX** -32002
- #define eyelink_tracker_time() eyelink_tracker_msec()
- #define eyelink_time_offset() eyelink_msec_offset()

Functions

- UINT16 open_eyelink_system (UINT16 bufsize, char FARTYPE ∗options)
- void eyelink_set_name (char FARTYPE ∗name)
- void close_eyelink_system (void)
- UINT32 current_time (void)
- UINT32 current_micro (MICRO FARTYPE ∗m)
- UINT32 current_usec (void)
- void msec_delay (UINT32 n)
- double current_double_usec (void)
- INT16 eyelink_open_node (ELINKADDR node, INT16 busytest)
- INT16 eyelink_open (void)
- INT16 eyelink_broadcast_open (void)
- INT16 eyelink_dummy_open (void)
- INT16 eyelink_close (INT16 send_msg)
- INT16 eyelink_reset_clock (INT16 enable)
- INT16 eyelink_is_connected (void)
- INT16 eyelink_quiet_mode (INT16 mode)
- INT16 eyelink_poll_trackers (void)
- INT16 eyelink_poll_remotes (void)
- INT16 eyelink_poll_responses (void)
- INT16 eyelink_get_node (INT16 resp, void FARTYPE ∗data)
- INT16 eyelink_node_send (ELINKADDR node, void FARTYPE ∗data, UINT16 dsize)
- INT16 eyelink_node_receive (ELINKADDR node, void FARTYPE ∗data)
- INT16 eyelink_send_command (char FARTYPE ∗text)
- INT16 eyelink_command_result (void)
- INT16 eyelink_timed_command (UINT32 msec, char FARTYPE ∗text)
- INT16 eyelink_last_message (char FARTYPE ∗buf)
- INT16 eyelink_send_message (char FARTYPE ∗msg)
- INT16 eyelink_node_send_message (ELINKADDR node, char FARTYPE ∗msg)
- INT16 eyelink_send_message_ex (UINT32 exectime, char FARTYPE ∗msg)
- INT16 eyelink_node_send_message_ex (UINT32 exectime, ELINKADDR node, char FARTYPE ∗msg)
- INT16 eyelink_read_request (char FARTYPE ∗text)
- INT16 eyelink_read_reply (char FARTYPE ∗buf)
- UINT32 eyelink_request_time (void)
- UINT32 eyelink_node_request_time (ELINKADDR node)
- UINT32 eyelink_read_time (void)
- INT16 eyelink_abort (void)
- INT16 eyelink_start_setup (void)
- INT16 eyelink_in_setup (void)
- INT16 eyelink_target_check (INT16 FARTYPE ∗x, INT16 FARTYPE ∗y)
- INT16 eyelink_target_checkf (float FARTYPE ∗x, float FARTYPE ∗y)
- INT16 eyelink_accept_trigger (void)
- INT16 eyelink_driftcorr_start (INT16 x, INT16 y)
- INT16 eyelink_driftcorr_startf (float x, float y)
- INT16 eyelink_cal_result (void)
- INT16 eyelink_apply_driftcorr (void)
- INT16 eyelink_cal_message (char FARTYPE ∗msg)
- INT16 eyelink_current_mode (void)

- INT16 eyelink_tracker_mode (void)
- INT16 eyelink_wait_for_mode_ready (UINT32 maxwait)
- INT16 eyelink_user_menu_selection (void)
- INT16 eyelink_position_prescaler (void)
- INT16 eyelink_reset_data (INT16 clear)
- void FARTYPE ∗ eyelink_data_status (void)
- INT16 eyelink_in_data_block (INT16 samples, INT16 events)
- INT16 eyelink_wait_for_block_start (UINT32 maxwait, INT16 samples, INT16 events)
- INT16 eyelink_get_next_data (void FARTYPE ∗buf)
- INT16 eyelink_get_last_data (void FARTYPE ∗buf)
- INT16 eyelink_newest_sample (void FARTYPE ∗buf)
- INT16 eyelink_get_float_data (void FARTYPE ∗buf)
- INT16 eyelink_get_double_data (void FARTYPE ∗buf)
- INT16 eyelink_newest_float_sample (void FARTYPE ∗buf)
- INT16 eyelink_newest_double_sample (void FARTYPE ∗buf)
- INT16 eyelink_eye_available (void)
- UINT16 eyelink_sample_data_flags (void)
- UINT16 eyelink_event_data_flags (void)
- UINT16 eyelink_event_type_flags (void)
- INT16 eyelink_data_count (INT16 samples, INT16 events)
- INT16 eyelink_wait_for_data (UINT32 maxwait, INT16 samples, INT16 events)
- INT16 eyelink_get_sample (void FARTYPE ∗sample)
- INT16 eyelink_data_switch (UINT16 flags)
- INT16 eyelink_data_start (UINT16 flags, INT16 lock)
- INT16 eyelink_data_stop (void)
- INT16 eyelink_playback_start (void)
- INT16 eyelink_playback_stop (void)
- INT16 **eyelink_request_image** (INT16 type, INT16 xsize, INT16 ysize)
- INT16 **eyelink_image_status** (void)
- void **eyelink_abort_image** (void)
- INT16 **eyelink_image_data** (INT16 FARTYPE ∗xsize, INT16 FARTYPE ∗ysize, INT16 FARTYPE ∗type)
- INT16 **eyelink_get_line** (void FARTYPE ∗buf)
- INT16 **eyelink_get_palette** (void FARTYPE ∗pal)
- UINT16 eyelink_read_keybutton (INT16 FARTYPE ∗mods, INT16 FARTYPE ∗state, UINT16 ∗kcode, UINT32 FARTYPE ∗time)
- INT16 eyelink_send_keybutton (UINT16 code, UINT16 mods, INT16 state)
- UINT16 eyelink_button_states (void)
- UINT16 eyelink_last_button_states (UINT32 FARTYPE ∗time)
- UINT16 eyelink_last_button_press (UINT32 FARTYPE ∗time)
- INT16 eyelink_flush_keybuttons (INT16 enable_buttons)
- INT16 eyelink_request_file_read (char FARTYPE ∗src)
- INT16 eyelink_get_file_block (void FARTYPE ∗buf, INT32 FARTYPE ∗offset)
- INT16 eyelink_request_file_block (UINT32 offset)
- INT16 eyelink_end_file_transfer (void)
- INT16 eyelink_get_tracker_version (char FARTYPE ∗c)
- INT16 eyelink2_mode_data (INT16 ∗sample_rate, INT16 ∗crmode, INT16 ∗file_filter, INT16 ∗link_filter)
- INT16 eyelink_mode_data (INT16 ∗sample_rate, INT16 ∗crmode, INT16 ∗file_filter, INT16 ∗link_-filter)

- INT16 **eyelink_bitmap_packet** (void ∗data, UINT16 size, UINT16 seq)
- INT16 **eyelink_bitmap_ack_count** (void)
- void eyelink_set_tracker_node (ELINKADDR node)
- double eyelink_tracker_double_usec (void)
- UINT32 eyelink_tracker_msec (void)
- double eyelink_double_usec_offset (void)
- UINT32 eyelink_msec_offset (void)

Variables

- ELINKADDR eye_broadcast_address
- ELINKADDR rem_broadcast_address
- ELINKADDR our_address

28.4 eyetypes.h File Reference

28.4.1 Detailed Description

Declarations of basic data types.

Data Structures

- struct MICRO

Defines

- #define **FARTYPE**
- #define **BYTEDEF** 1
- #define **MICRODEF** 1

Typedefs

- typedef unsigned char **byte**
- typedef signed short **INT16**
- typedef unsigned short **UINT16**
- typedef signed int **INT32**
- typedef unsigned int **UINT32**

28.5 gdi_expt.h File Reference

28.5.1 Detailed Description

Declarations of eyelink_exptkit functions and types.

This file will also reference the other EyeLink header files.

Defines

- #define **SCREEN_LEFT** dispinfo.left
- #define **SCREEN_TOP** dispinfo.top
- #define **SCREEN_RIGHT** dispinfo.right
- #define **SCREEN_BOTTOM** dispinfo.bottom
- #define **SCRHEIGHT** dispinfo.height
- #define **SCRWIDTH** dispinfo.width
- #define process_key_messages gdi_process_key_messages
- #define bitmap_save_and_backdrop gdi_bitmap_save_and_backdrop
- #define bitmap_to_backdrop gdi_bitmap_to_backdrop
- #define bitmap_save gdi_bitmap_save
- #define message_pump(x) message_pump()
- #define **CALLTYPE** ELCALLTYPE
- #define **HOOK_ERROR** -1
- #define **HOOK_CONTINUE** 0
- #define **HOOK_NODRAW** 1
- #define **CAL_TARG_BEEP** 1
- #define **CAL_GOOD_BEEP** 0
- #define **CAL_ERR_BEEP** -1
- #define **DC_TARG_BEEP** 3
- #define **DC_GOOD_BEEP** 2
- #define **DC_ERR_BEEP** -2

Functions

- void set_calibration_colors (COLORREF fg, COLORREF bg)
- void set_target_size (UINT16 diameter, UINT16 holesize)
- void set_cal_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- void set_dcorr_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- INT16 set_camera_image_position (INT16 left, INT16 top, INT16 right, INT16 bottom)
- void get_display_information (DISPLAYINFO ∗di)
- INT16 init_expt_graphics (HWND hwnd, DISPLAYINFO ∗info)
- void close_expt_graphics (void)
- void wait_for_video_refresh (void)
- UINT16 gdi_process_key_messages (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
- void wait_for_drawing (HWND hwnd)
- int gdi_bitmap_save_and_backdrop (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 bx_options)
- int gdi_bitmap_to_backdrop (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 bx_options)

- int gdi_bitmap_save (HBITMAP hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- INT16 set_setup_cal_display_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 set_clear_cal_display_hook (INT16(∗hookfn)(HDC hdc), INT16 options)
- INT16 CALLTYPE set_erase_cal_target_hook (INT16(∗hookfn)(HDC hdc), INT16 options)
- INT16 CALLTYPE set_draw_cal_target_hook (INT16(∗hookfn)(HDC hdc, INT16 ∗x, INT16 ∗y), INT16 options)
- INT16 set_exit_cal_display_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 CALLTYPE set_cal_sound_hook (INT16(∗hookfn)(INT16 ∗error), INT16 options)
- INT16 set_record_abort_hide_hook (INT16(∗hookfn)(void), INT16 options)
- INT16 set_setup_image_display_hook (INT16(∗hookfn)(INT16 width, INT16 height), INT16 options)
- INT16 set_image_title_hook (INT16(∗hookfn)(INT16 threshold, char ∗cam_name), INT16 options)
- INT16 set_draw_image_line_hook (INT16(∗hookfn)(INT16 width, INT16 line, INT16 totlines, byte ∗pixels), INT16 options)
- INT16 set_set_image_palette_hook (INT16(∗hookfn)(INT16 ncolors, byte r[ ], byte g[ ], byte b[ ]), INT16 options)
- INT16 set_exit_image_display_hook (INT16(∗hookfn)(void), INT16 options)
- void initialize_gc_window (int wwidth, int wheight, HBITMAP window_bitmap, HBITMAP background_bitmap, HWND window, RECT display_rect, int is_mask, int deadband)
- void redraw_gc_window (int x, int y)
- HDC get_window_dc (void)
- INT16 release_window_dc (HDC hdc)

Variables

- DISPLAYINFO **dispinfo**

28.6 sdl_expt.h File Reference

28.6.1 Detailed Description

Declarations of eyelink_core_graphics functions and types.

This file will also reference the other EyeLink header files.

Data Structures

- struct **_CCDBS**

Defines

- #define **SDLRGB**(x, y) SDL_MapRGB(x → format,(y).r,(y).g,(y).b)
- #define **SCREEN_LEFT** dispinfo.left
- #define **SCREEN_TOP** dispinfo.top
- #define **SCREEN_RIGHT** dispinfo.right
- #define **SCREEN_BOTTOM** dispinfo.bottom
- #define **SCRHEIGHT** dispinfo.height
- #define **SCRWIDTH** dispinfo.width
- #define bitmap_save_and_backdrop sdl_bitmap_save_and_backdrop
- #define bitmap_to_backdrop sdl_bitmap_to_backdrop
- #define bitmap_save sdl_bitmap_save
- #define **Flip**(x) while(SDL_Flip(x)<0)
- #define **EXTERNAL_DEV_NONE** ((getExButtonStates)0)
- #define **EXTERNAL_DEV_CEDRUS** ((getExButtonStates)1)
- #define **EXTERNAL_DEV_SYS_KEYBOARD** ((getExButtonStates)2)

Typedefs

- typedef _CCDBS **CCDBS**
- typedef int(∗ **getExButtonStates**)(CCDBS ∗)

Functions

- void set_calibration_colors (SDL_Color ∗fg, SDL_Color ∗bg)
- void set_target_size (UINT16 diameter, UINT16 holesize)
- void set_cal_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- void set_dcorr_sounds (char ∗ontarget, char ∗ongood, char ∗onbad)
- INT16 set_camera_image_position (INT16 left, INT16 top, INT16 right, INT16 bottom)
- void get_display_information (DISPLAYINFO ∗di)
- INT16 init_expt_graphics (SDL_Surface ∗hwnd, DISPLAYINFO ∗info)
- void close_expt_graphics (void)
- int sdl_bitmap_save_and_backdrop (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options, INT16 xd, INT16 yd, UINT16 bx_options)
- int sdl_bitmap_to_backdrop (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, INT16 xd, INT16 yd, UINT16 bx_options)

- int sdl_bitmap_save (SDL_Surface ∗hbm, INT16 xs, INT16 ys, INT16 width, INT16 height, char ∗fname, char ∗path, INT16 sv_options)
- void set_cal_target_surface (SDL_Surface ∗surface)
- void set_cal_background_surface (SDL_Surface ∗surface)
- void reset_background_surface ()
- void **disable_custombackground_on_imagemode** ()
- int set_cal_animation_target (const char ∗aviName, int playCount, int options)
- int enable_external_calibration_device (getExButtonStates buttonStatesfcn, const char ∗config, void ∗userData)
- void **set_cal_font** (const char ∗fontPath, int size)

28.7 w32_dialogs.h File Reference

28.7.1 Detailed Description

Declaration of Win32 support dialogs.

Defines

- #define receive_data_file receive_data_file_dialog

Functions

- INT32 receive_data_file_dialog (char ∗src, char ∗dest, INT16 dest_is_path)
- INT16 edit_dialog (HWND hwnd, LPSTR title, LPSTR msg, LPSTR txt, INT16 maxsize)
- INT16 ask_session (HWND hw, LPSTR title, LPSTR msg, LPSTR path, INT16 pathmax, LPSTR txt, INT16 maxsize)

Index