Unifi™ Software Development Kit (SDK)
Making Affordable Adaptive Optics Easy to Use

AgilOptics
1717 Louisiana NE
Suite 202
Albuquerque, NM 87110
(505) 268-4742
support@agiloptics.com

SDK Overview

The Unifi SDK is a software library that allows the user to quickly and easily create applications to manipulate the UNIFI™ mirrors from AgilOptics. The Unifi SDK can be used in either LabView or C.

The Unifi SDK can detect the first available UNIFI™ mirror on your USB, and write voltages to it's actuators, making controlling a UNIFI™ mirror a breeze. If you have need of multiple UNIFI™ mirrors, the Unifi SDK can connect to each mirror on the USB by utilizing the order in which they appear on the USB device list. Once connected, voltages can be written to any mirror simply by knowing it's device number. More on this is covered in page 7 of the **SDK Use Tutorial**, found later in this brochure.

The Unifi SDK can write single actuator voltages or whole voltage patterns to any particular device. Writing a whole voltage pattern allows for huge changes in the mirror's shape in a short amount of time, such as changing the mirror pattern from being flat to representing an astigmatism, or from representing coma in the X direction to representing coma in the Y direction, and back. Writing individual voltages allows for small adjustments to the voltages currently on the mirror.

With basic commands and LabView, building a working controller for a UNIFI™ mirror can be done quickly and simply. The commands in the SDK can be run through a simple user interface, like the examples provided with the SDK, or as part of a larger, more complex program. With the Unifi SDK, the possibilities are endless. Use a series of mirrors to gradually change a light beam into different patterns, or correct aberrations in multiple light beams at the same time. Add known aberrations to a beam with one mirror and remove them with another. Improve the image quality of a telescope, camera, or microscope, or just make interesting patterns on the mirror's surface.

Where the Unifi ™ SDK Fits Into Your System

Your Code

TextPad - [\\Server\users\DRH\example_code.cpp]

File Edit Search View Tools Macros Configure Window Help

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "unifi_controller.h"

void main()
  {
  int cur_actuator;
  int num_usb_devices;
  double cur_voltage;

  cur_actuator = 0;
  cur_voltage = 0.0;

  InitDLL();
  num_usb_devices = GetNumberOfDevices();

  if(num_usb_devices > 0)
    {
    // Now we're cooking with gas!
```

Unifi™ SDK

```
#ifndef _INCLUDE_UNIFI_CONTROLLER_H
#define _INCLUDE_UNIFI_CONTROLLER_H

// Exported functions:
// Note:  Actuators are numbered 1 to N, wher N is the number of actuators for
// a particular mirror.  The N + 1th actuator is the mirror surface itself.
//        Channels are numbered 1 to M, where M is the number of channels for
// a particular mirror.  There are 40 channels for every 32 actuators.  Not all
// the channels do anything useful, and actuators do NOT map directly to
// same-numbered channels.  Most users will wish to use the Actuator commands
// instead of the Channel commands.

// Find out how many devices are on the USB.
// Input:  None
// Output:  The number of devices it detected on the USB, regardless of
//          compatability with our driver requirements.
extern "C" __declspec(dllexport) int GetNumberOfDevices();

// Find out what devices are on the USB.
// Input:  char ***device_list  - A pointer to an array of char *'s.  This will
//                                contain the names of all devices found on the
//                                USB, regardless of compatability with our
//                                driver requirements.
//         int *num_devices     - A pointer to an integer.  This will contain
//                                the number of devices found on the USB.
// Output: None.  All output is stored on the input variables.
extern "C" __declspec(dllexport) void GetDevices(char ***device_list,
                                                 int *num_devices);

// Initialize the DLL.  This MUST be done before any commands below this can
// be sucessfully run.
// Input:  None
// Output:  Whether or not the initialization was successful.
extern "C" __declspec(dllexport) bool InitDLL();

// Uninitialize the DLL.  Good for making sure you clean up your memory.
extern "C" __declspec(dllexport) bool UninitDLL();
```

UNIFI™ Mirror

SDK Utilities

The Unifi SDK comes with two sample control packages. One is a Borland C++ program and the other is a LabView VI. Each controls one or more mirrors in a different way. One can control only a single mirror and writes to specific channels, while the other can control many mirrors and writes to actuators. More on the difference between actuators and channels can be found in the **SDK Use Tutorial** section, on page 8. The Borland C++ program comes with both the executable and the source code. No source code is necessary for the LabView VI.

With LabView, it is very easy for someone who doesn't know how to program to build a simple user interface to connect to the Unifi SDK. With C++, however, a similar interface will run about four times faster.

Using the C++ Unifi Controller:

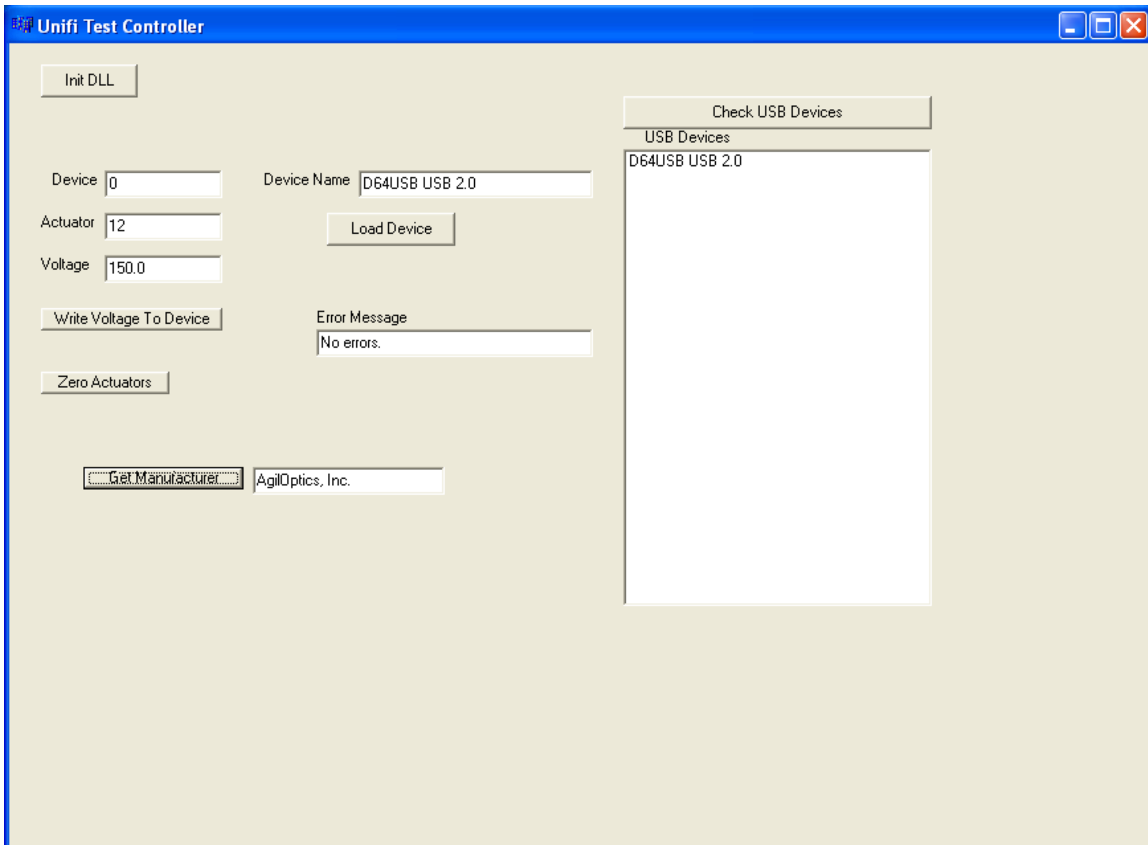


Figure 1. Unifi SDK Example in Borland C++.

The Borland C++ Unifi Controller makes use of the unifi_controller.lib file and the unifi_controller.h file. The C++ Unifi Controller uses the actuator commands in the unifi_controller.dll. The C++ Unifi Controller also allows multiple mirrors to be controlled. The user simply enters the device number of the mirror to be manipulated and chooses which actions are most appropriate for that device at the time.

Running the Unifi SDK C++ Example:

- **Step 1: Initialize the DLL.** Most of the functions in the SDK require the DLL to be initialized. The DLL could be initialized on start-up of the program but was left as a simple button to emphasis it's importance. All functions that rely on the DLL being initialized are disabled until "Init DLL" is successful.

- **Step 2: Check USB Devices.** Query the USB devices by clicking on the "Check USB Devices" button. A list of all detected devices connected to the USB will show up below the button, if there are any. USB devices include USB mice, printers, UNIFI™ mirrors, etc. If no devices are connected, an error message stating such will be displayed in the Error Message box.

- **Step 3: Load A Device.** If a UNIFI™ mirror was detected, it's device number is the position that it appears at in the USB Devices list. The first item in the list is device 0. The second item is device 1, etc. In the Device edit box, change the number from 0 to the UNIFI™ mirror's device number. Then select the "Load Device" button. If successful, the Device Name should change to the name of the UNIFI™ mirror. If unsuccessful, an error message will appear in the Error Message box.

- **Step 4: Write Voltages to D64-USB.** Once a UNIFI™ mirror is loaded, the user may choose actuators to write voltages to, one at a time. After an actuator is chosen and a voltage is set, the user simply selects "WriteVoltage To Device" to write the given voltage to the mirror.

- ***Note: If the user wishes to reset all the actuators to zero , here is a "Zero Actuators" button available.***

Using the LabView Unifi Controller:

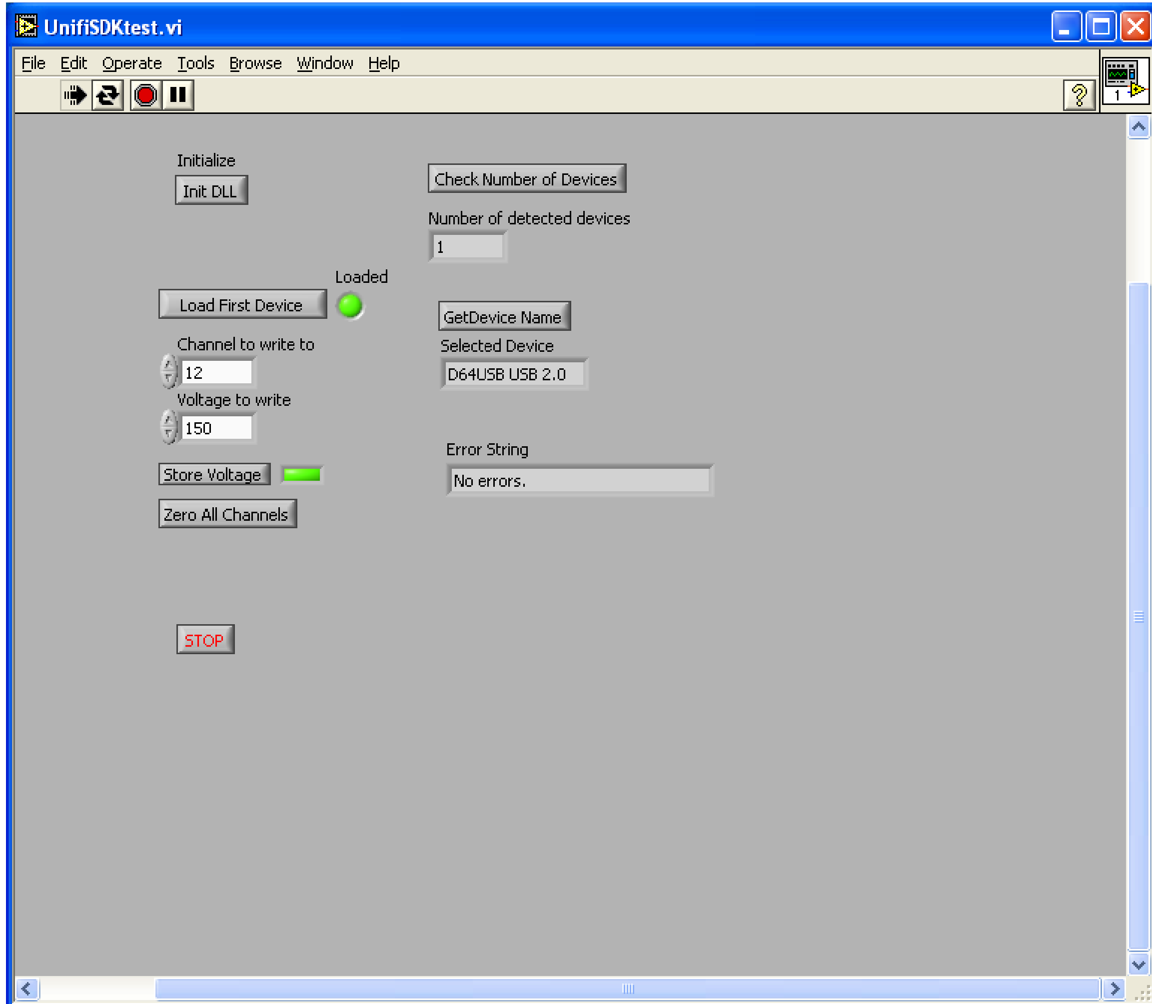


Figure 2. Unifi SDK Example in LabView.

The LabView Unifi Controller makes use of the unifi_controller.dll file. This .dll file can be used in any environment that allows importing C code .dll files. The LabView Unifi Controller uses the channel commands in the unifi_controller.dll. The Lab View Unifi Controller example only allows manipulation of a single UNIFI™ mirror, using the single UNIFI™ mirror commands. No device selection is necessary. To view the internals of the LabView Unifi Controller, select Window->Show Block Diagram, or press CTRL+E. The LabView example controller was built using LabView 7.0.

Running the Unifi SDK LabView Example:

- **Step 1: Choose "Run Continuously" to Start the Program.** Unless you want pausing for every step the program makes in LabView, let the program run continuously. If you prefer to watch how the program does it's job, feel free to just choose "Run" instead.

- <u>Step 2</u>: **Initialize the DLL.** Before most of the function calls in the unifi_controller.dll file can be called successfully, the .dll file must be initialized. This builds the necessary structures in the DLL for keeping track of the mirrors, as well as initialized important variables in the DLL.

- <u>Step 3</u>: **Choose "Check Number of Devices".** When executed, this command detects how many devices are attached to the USB ports on the computer. The number of detected devices will show up in the "Number of detected devices" edit box.

- <u>Step 4</u>: **Load the First Device.** Now that the DLL is initialized, and hopefully at least one UNIFI™ mirror was found on the USB, the user should select "Load First Device". If successful, the light next to "Load First Device" should turn on. If unsuccessful, an error message will display in the Error String edit box.

- <u>Step 5</u>: **Write Voltages to D64-USB.** Once the devices is loaded, the user may get the device name or set voltages to channels, using the "Store Voltage" button. Choose a channel and write a voltage. 150.0 is a good initial voltage. If the channel maps properly to an actuator, there should be a change in the mirror's surface.

- *Note: Any voltages on the* ***UNIFI™ mirror can be removed by using the "Zero All Channels" button. Not all channels map directly to actuators. See the notes on the difference between channels and actuators in the SDK Use Tutorial, on page 8.***

SDK Use Tutorial

The Unifi Controller SDK is simple to use, but has some basic requirements to keep in mind when you are preparing the program.

- **All function prototypes can be found in the file unifi_controller.h.** Above each function is a description of how the function works. This includes input values, output values, and any special case information about the function.

- **Always initialize the DLL before attempting any other commands.** This is done by the *InitDLL()* function. Only two commands do not require the DLL to be initialized. They are:
  int GetNumberOfDevices();
  *void GetDevices(char ***device_list, int *num_devices);*

  These functions do not rely on the DLL being ready for use. Note that the device list in *GetDevices* is a pointer to a list of strings, not a two-dimensional array of strings.

- **Device numbering always starts at 0.** The first device in the device list is 0, the second is 1, etc. The ordering on the list will not change unless USB devices are removed/added while the program is running. It is recommended that no devices are removed or added while the SDK is in use.

There are two types of commands in the SDK. Single-mirror commands and specific device commands. An example is:

bool InitDriver(int NumberOfChannels);
bool InitDriverDevice(int NumberOfChannels, int DeviceNumber);

All single-mirror commands have a specific device command who's name adds Device to the end, and has a last argument of *int DeviceNumber*. The single-mirror commands access the first available mirror detected on the USB. The specific device commands access the given device number. The specific device commands may fail if the device number given is not a valid Unifi device.

If device 1 was the first mirror in your device list, then the following commands would be equivalent:

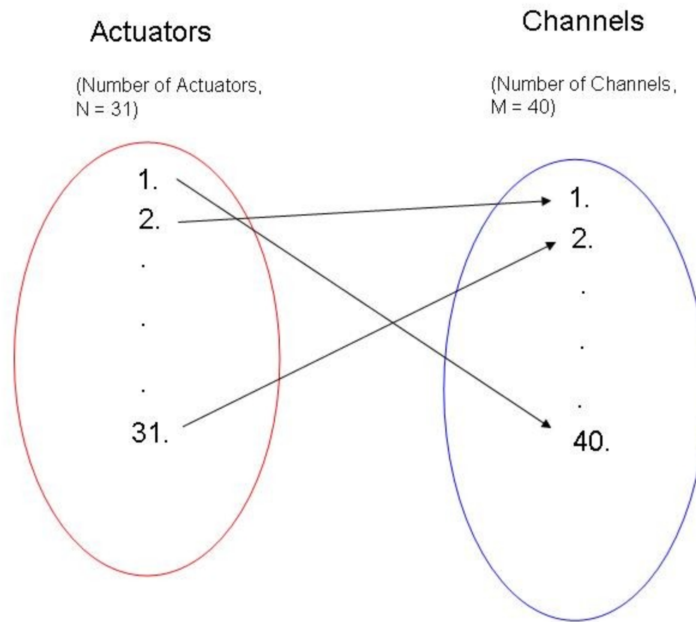
InitDriver(61);
InitDriverDevice(61, 1);

Each would initialize a mirror driver for 61 actuators.

- **There are two methods to write voltages to the mirror**. These are Set and Send. A Set command stores the voltage(s), but does not write them to the mirror immediately. Set commands can help save time when setting individual actuator voltages, since writing all actuator voltages to the mirror at once is faster than writing each actuator voltage to the mirror individually. The Set command must be followed by a *WriteDriver()* or a *WriteDriverDevice()* command for it's voltages to be successfully sent to the mirror.

Send commands set the voltage just like a Set command and write it immediately to the mirror. This is slower if multiple actuators need to be sent voltages at the same time, since each gets written to the mirror before the next one is set. Imagine you're trying to change all the voltages on a 36 actuator mirror to 150.0. With 36 Sends, and 8,000 writes per second (See **Actuator Manipulation Speeds** on page 9), you could set the whole mirror to 150.0 volts 222.2 times per second. With 36 Sets and one Write (or one SendAll), you could set the whole mirror to 150.0 volts 8,000 times per second. However, if you only need to change one actuator, a Send is equivalent to a Set followed by a Write.

- **The user may write to channels instead of actuators.** An actuator map maps actuator number (one through N) to channels (0 through M, where M is greater than N).



Actuators do not map directly to Channels.

Actuators do not map directly to channels of the same number. Actuator 1 may map to channel 53 while actuator 2 maps to channel 13. This is important because, currently, the Unifi SDK only supports one actuator mapping at a time. This means if you have two mirror, one with 61 actuators, and another with 36 actuators, only one will be supported by the Unifi SDK. To successfully write to the unsupported mirror, you can use the channel commands instead. This, however, requires you to build your own actuator mapping by testing each channel in the unsupported mirror for a change on the mirror's surface. An example of an actuator command versus a channel command is:

bool SetActuator(int Actuator, double Voltage);
bool SetChannel(int Channel, double Voltage);

If actuator 1 mapped to channel 13, the following code would be equivalent:

SetActuator(1, 150.0);
SetChannel(13, 150.0);

Note that, since there are more channels than actuators, some channels do not map to any actuators. Writing voltages to these channels will not alter the mirror's surface.

Actuator Manipulation Speeds:

- **8000 Full Mirror Changes can be made per second.** A full mirror change means changing every actuator's voltage to a different value than the previous mirror settings. An example would be going from zero volts on all actuators to setting each actuator to five volts above the previous actuator, starting at one volt on actuator one. If the interface program can send them quickly enough, 8000 patterns can be written to the UNIFI™ mirror per second, if the UNIFI™ mirror has no more than 128 actuators. For 129 to 256 actuators, the speed is reduced to 4000 full mirror changes per second. This limitation is imposed by USB data transfer speeds.

- **Partial Mirror Changes are no slower or faster than Full Mirror Changes.** If you don't wish to do full mirror changes, there is no speed increase between writing one actuator, twenty actuators, or all the actuators for UNIFI™ mirrors with up to 128 actuators.

- **Manipulating multiple mirrors divides the speed by the number of mirrors.** If you wish to make full mirror changes to multiple mirrors at the same time, the speed is reduced by the number of mirrors. Two mirrors can be fully changed 4000 times per second, three mirrors can be fully changed 2667 times per second, etc.

- **The unifi_controller.dll runs just as fast in LabView as in C++.** Since the .DLL is written in C++, it runs just as fast in LabView as in C++. LabView's interface code, however, is slower than C++'s, by about four times. This is because LabView uses generic code sections combined to build functions, some of which are not particular designed for the tasks they may be given. C++, however, is built strictly on the needs of the current project, allowing it to be streamlined to the situation.