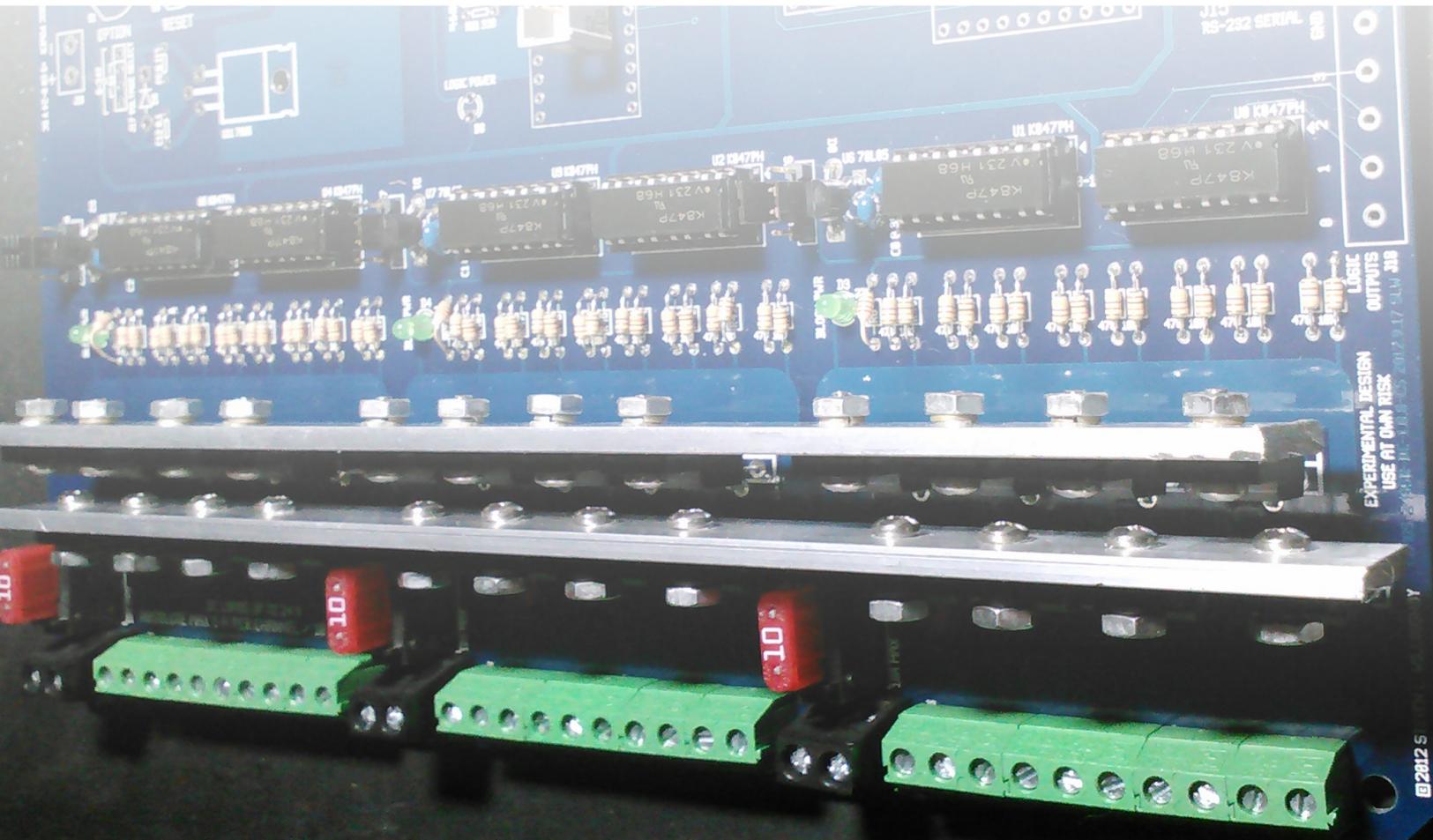


Using the Lumos™ SSR Controllers



EXPERIMENTAL DESIGN
USE AT OWN RISK



RISK OF FIRE, ELECTROCUTION, SERIOUS INJURY OR DEATH!

This circuit design, including but not limited to any associated plans, schematics, designs, board layouts, documentation, and/or components, is EXPERIMENTAL and for EDUCATIONAL purposes only. It is not a finished consumer-grade product. It is assumed that you have the necessary understanding and skill to assemble and/or use electronic circuits.

Proceed ONLY if you know exactly what you are doing, understand the proper procedures for working with the high voltage present on the components and PC boards, and understand that you do so ENTIRELY AT YOUR OWN RISK.

The author makes NO representation as to suitability or fitness for any purpose whatsoever, and disclaims any and all liability or warranty to the full extent permitted by applicable law.

Edition 2.0, for Lumos controllers with ROM version 3.0.

Copyright © 2013 by Steven L. Willoughby, Aloha, Oregon, USA. All Rights Reserved. This document is released under the terms and conditions of the Creative Commons “Attribution-NoDerivs 3.0 Unported” license. In summary, you are free to use, reproduce, and redistribute this document provided you give full attribution to its author and do not alter it or create derivative works from it. See <http://creativecommons.org/licenses/by-nd/3.0/> for the full set of licensing terms.



CONTENTS

Contents	iii
1 Introduction	1
1.1 Intended Audience	1
1.2 Limitation of Warranty	1
1.3 How to Use this Manual	2
1.4 The Name of the Game	2
1.5 Getting Additional Help	2
2 Safety Information	3
2.1 Small Part Danger	3
2.2 Hazardous Voltage	3
2.3 Electrostatic Discharge (ESD) Warning	4
2.4 Circuit Loading	4
3 Overview of the Lumos Controller Models	5
3.1 48-Channel Controller	5
3.2 48-Channel Front Panel	6
3.3 24-Channel AC Relay	7
3.4 24-Channel DC Controller/Relay	9
4 Operating the Board Controls	11
4.1 Resetting the Board	11
4.2 Entering Configuration Mode	11
4.3 Running a Test Pattern	12
4.4 Restoring to Factory Settings	12
5 Configuring the Board	15
5.1 Setting the Device Address	16
5.2 Setting the Device Speed	16
5.3 Sensors	17
5.4 Setting a Lumos Controller to Use DMX512	18
5.5 Canceling Configuration Mode	19
6 Creating Programmed Sequences	21
6.1 Basic Sequence Syntax and Structure	22

6.2 Expressions	22
6.3 Variable Assignment	23
6.4 Controlling Output Channels	23
6.5 Device State Changes	24
6.6 Flow Control	25
6.7 Subroutines and Sequence Chains	26
6.8 Temporary vs. Permanent Sequences	27
6.9 Storing Sequences into Device Memory	27
6.10 Bytecode Reference	28
6.11 Loading Binary Sequences Onto Lumos Boards	49
7 Communication Protocol Details	53
7.1 0x0–0x6: Common Commands	55
7.2 0x700–0x71f: Extended Commands	58
7.3 0x720–0x73f: Reserved	64
7.4 0x740–0x77f: Configuration-Mode Commands	64
8 DMX512 Command Structure	69
9 Theory of Operation	71
9.1 Output Relay Circuits	76
Diagnostic Codes	77
Decoding LED Patterns	77
Numeric Error Codes	77
Lumos CLI Command Manual Entries	81
Troubleshooting	97
Glossary	99
Acknowledgements	101
Index	103
Colophon	105

C H A P T E R

1

INTRODUCTION

CONGRATULATIONS ON JOINING the many computer-controlled Christmas light enthusiasts, theatrical lighting technicians, electronics hobbyists, and home automation innovators who are experimenting with new ways to have computers control lights and other electronic devices.

This manual details the software controls implemented by the Lumos controllers and the communication protocols they use with the host PC.

1.1 Intended Audience

This is an “advanced” level do-it-yourself electronic circuit project. It is not an off-the-shelf consumer-ready product. It is only designed for educational and experimental use by experienced hobbyists and professionals who possess the skill to construct electronic circuits, to understand how they function, troubleshoot problems with them, and to use them safely.

This manual provides basic usage and configuration instructions suitable for all users of Lumos controllers.

Some of the information in this manual gives a level of technical detail suited to advanced users and software developers who need to understand the workings of Lumos controllers to write applications which interface with them.



1.2 Limitation of Warranty

Since this is a do-it-yourself project, the quality of the final product, and whether it functions as intended, is largely a result of your own efforts in



building it. As such, we cannot offer to troubleshoot, repair, or replace a board we did not assemble for you. Accordingly, these instructions, and all accompanying plans, schematics, software, hardware, and other project materials are provided to you “AS-IS” at no cost, as a courtesy between DIY hobbyists with NO WARRANTY of any kind expressed or implied. If you proceed to build and/or use this unit, you do so ENTIRELY AT YOUR OWN RISK.

If you purchased hardware materials from us (such as a PC board or programmed controller chip), we will—at our sole discretion—replace, repair, or refund the cost of those materials if they were defective in manufacture as shipped to you, up to 90 days from the date they were shipped to you, but are not liable for damage caused by your handling or assembly of the unit. Otherwise, we make no representation of suitability or fitness for any particular purpose and disclaim all other warranty or liability of any kind to the full extent permitted by law.

1.3 How to Use this Manual

Start with the information in the first part of the manual to learn how to operate and configure your Lumos controller, and how to use the host PC to alter its configuration settings.

If you need to know the low-level details concerning how the board communicates with the PC, continue reading the more advanced material which comprises the remainder of this manual.

1.4 The Name of the Game

The name “Lumos” is a combination of *lumen*, the Latin word for “light,” and the initial letters of “Orchestration System.” Hence, “Light Orchestration System” which is the most common application for which the Lumos hardware and software are used—running computerized lighting displays.

1.5 Getting Additional Help

The product website at www.alchemy.com/lumos contains additional documentation, pointers, hints, and tips to assist you further. If that doesn’t answer all your questions, there is an online forum where you may submit questions for help.

C H A P T E R

2

SAFETY INFORMATION

BEFORE YOU USE your Lumos controller, please take the time to carefully read the following safety precautions. Failure to follow this advice could result in death or serious injury, damage to the Lumos controller unit, and/or damage to the other devices plugged into the controller.

2.1 Small Part Danger

This board contains small parts which could pose a choking hazard to small children. This product is not a toy and is not intended for use by children in any circumstance. The small parts on the product can be swallowed by children under 4 years of age. Keep out of reach of children.



2.2 Hazardous Voltage

Exercise care when working with any electrical system, including one such as the Lumos DC controllers (even though in theory they deal with low voltages). The power supplies of the loads plugged into the Lumos controller, and even the power loads being controlled, may present a shock hazard if not wired and handled using standard safety protocols. Never touch or work with live circuits. Always disconnect the power source before working on your Lumos controller.

When working with loads outdoors, be sure all supplies are plugged into GFCI-protected circuits.



2.3 Electrostatic Discharge (ESD) Warning



Many of the components used in this project are sensitive to static electricity. Always use a proper ESD-safe work environment when handling them, or these parts may be permanently damaged. If a part is damaged in this way, it is impossible to tell by looking at the part, and you won't necessarily feel the static discharge which caused the damage. Never take the risk of handling sensitive components without ESD protection in place.

These parts include all transistors (Q0–Q23), voltage regulators (U6–U8 and U11), diodes (D0–D11), and integrated circuits (U0–U5, U9–U10, and U12–U13).

2.4 Circuit Loading



Always respect the maximum voltage and current capacity of the board and your wiring. Overloading any of these may result serious injury, death, fire, and/or severe damage to any or all of the devices in use.

Each block of eight controlled loads may not exceed 10 A total for the block. Each single output channel may not exceed 5 A. These should be considered *absolute maximum* tolerances. The board was designed to operate at sustained levels below those limits.

C H A P T E R



OVERVIEW OF THE LUMOS CONTROLLER MODELS

HERE ARE CURRENTLY four different hardware projects which are part of the Lumos family of controllers, which are the subject of this manual:

3.1 48-Channel Controller

This board controls 48 channels of outputs but does not contain any actual relay circuits of its own. The TTL-level outputs are designed to be sent directly to a 24-channel relay board (AC, DC, or one of each), although of course they could be connected to another compatible circuit.

The controller supports only half-duplex RS-485 communications. It occupies a single address for the Lumos command protocol. When in DMX512 mode, it consumes 48 slots beginning at an arbitrarily-configured starting slot number.

This board contains its own power supply. It accepts a 120 V AC input (with provision for an optional power switch built-in). This power source provides the +5 V DC supply for the controller itself as well as the logic-side components of the relay boards to which it is connected. The controller also uses this AC supply to pick up the power waveform's zero-crossing point for AC relay board dimmer synchronization. This means that it is necessary that the AC power supplied to the controller be in phase with the power supply to the relays' loads (or exactly 180° out of phase). This should be the case in any standard residential or commercial office environment.

6 CHAPTER 3. OVERVIEW OF THE LUMOS CONTROLLER MODELS

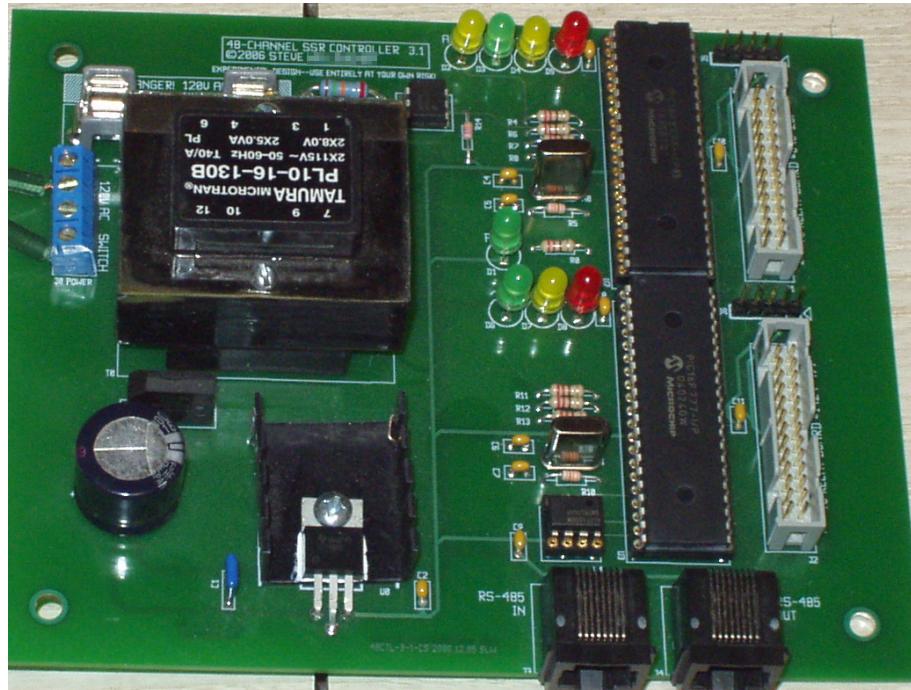


Figure 3.1: Lumos 48-Channel Controller

Although a few prototypes of this board have been built and used successfully, it has not yet been released as an open-hardware project. We may do so in the future, possibly with an enhanced circuit design.

3.2 48-Channel Front Panel

Designed to work with the Lumos 48-Channel controller, this panel is connected to the controller's outputs in parallel with the relay boards. It provides an LED for each output channel, allowing you to see at a glance the activity and state of all outputs. Additionally, it has LEDs which show that power is being sent to the relays, cable check LEDs to show that the cables are plugged in the entire length of the network, and serial I/O status indicators. It allows for the controller's status LEDs to be displayed on the front panel as well.

This board also contains a pair of RS-232-to-RS-485 converters, one full-duplex and one half-duplex. These may be switched between two modes: one where the host PC is always holding the bus to transmit constantly (or at least at will), and the other where the PC can turn the transmitter on or off by changing the state of the DTR line.

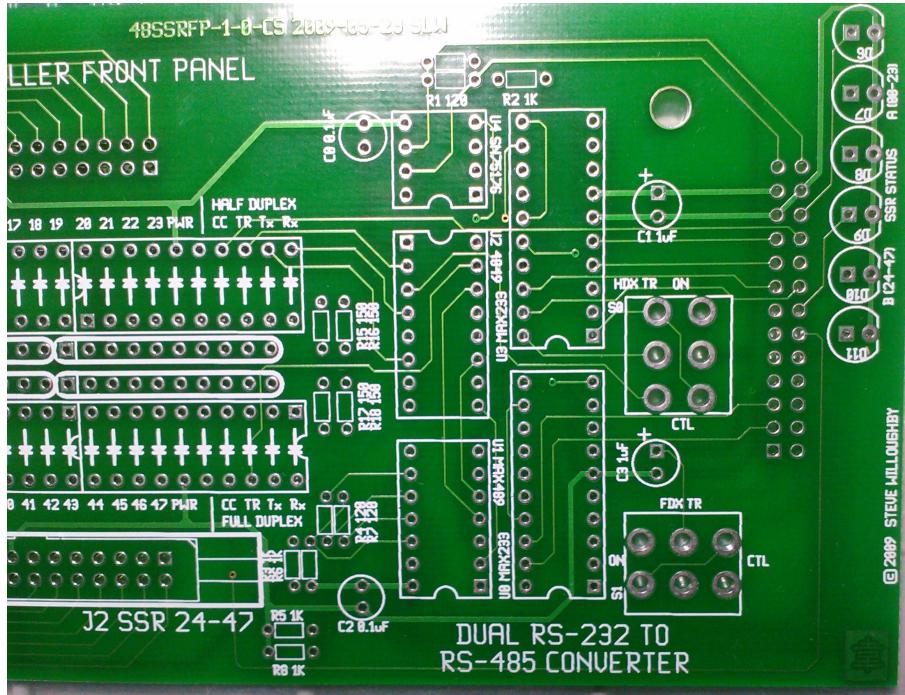


Figure 3.2: Lumos 48-Channel Front Panel (Bare PCB)

Although a few prototypes of this board have been built and used successfully, it has not yet been released as an open-hardware project. We may do so in the future, possibly with an enhanced circuit design.

3.3 24-Channel AC Relay

This board controls 24 output channels of 120 V AC, arranged in three separate blocks of 8 channels. Each block is separately powered and completely isolated from the other blocks and from the low-voltage logic side of the board (which connects to the controller). Each block is designed to supply up to 5 A per channel, with a maximum of 8 A total per block at any given time.

Although a few prototypes of this board have been built and used successfully, it has not yet been released as an open-hardware project. We may do so in the future, possibly with an enhanced circuit design.

8 CHAPTER 3. OVERVIEW OF THE LUMOS CONTROLLER MODELS

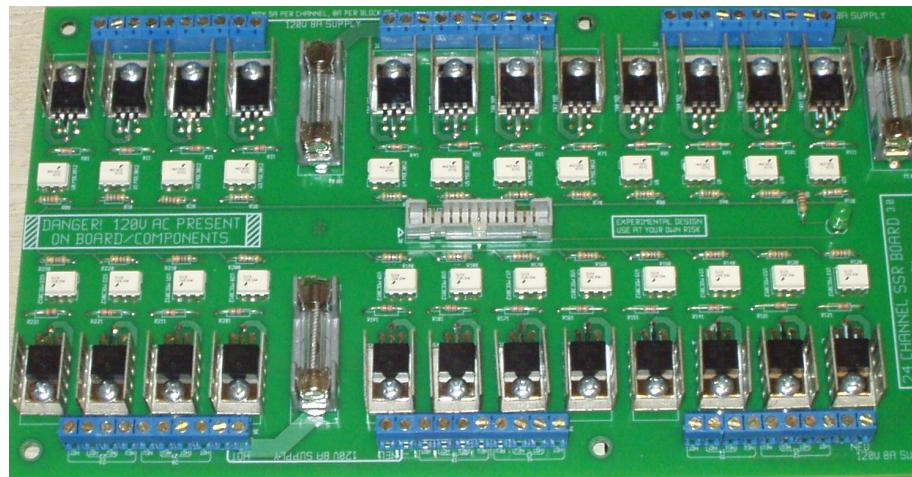


Figure 3.3: Lumos 24-Channel AC Relay

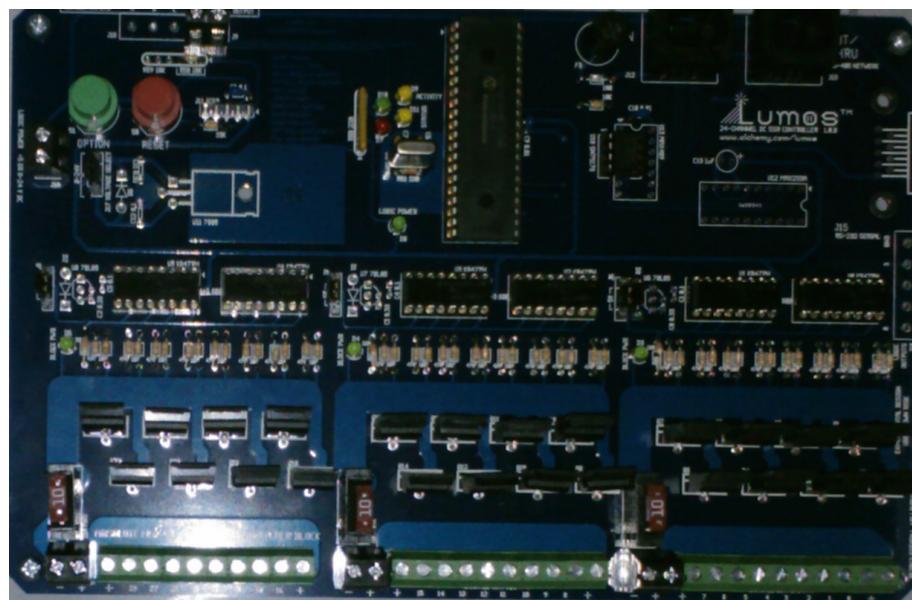


Figure 3.4: Lumos 24-Channel DC Controller/Relay

3.4 24-Channel DC Controller/Relay

This board controls 24 output channels for low-voltage DC loads, arranged in three separate blocks of 8 channels. Each block is separately powered and completely isolated from the other blocks and from the logic side of the board (i.e., the on-board controller or connection to an external controller). Each block may be powered with +5 V DC or a voltage from +8 to +24 V DC and is designed to supply up to 5 A per channel, with a maximum of 10 A total per block at any given time.

Unlike the AC relay board described above, this may be constructed as a relay-only board (and attached to something like the Lumos 48-Channel controller) or as a stand-alone controller itself. In the latter configuration, it uses one address for Lumos-protocol communication, or 24 DMX512 slots. Communications options (selected permanently at the time the controller is assembled) include RS-232, full-duplex RS-485, and half-duplex RS-485.

By sacrificing one or more of the on-board diagnostic LEDs, 1–4 input signals may be monitored by the controller, triggering user-programmed actions. This allows for things like a light sensor to trigger nighttime lighting which turns off in the daytime, or to announce the arrival of guests when a door sensor is opened.

C H A P T E R



OPERATING THE BOARD CONTROLS

THE LUMOS CONTROLLERS INCLUDE two front-panel buttons labeled “Option” and “Reset.” This chapter will describe how to use these buttons to control the board manually. We will assume the Reset button is red and the Option button is green. Depending on components chosen when building the board, these colors may vary.

4.1 Resetting the Board

The red Reset button is connected directly to the microcontroller’s $\overline{\text{MCLR}}$ input. As long as the button is pressed, the CPU will be halted. No operations of the controller will be active at this time.

When the button is released, the CPU will reboot as if powered up. This restores the ability to enter configuration mode, resets all output channels to be fully off, clears sequences stored in normal RAM memory, clears all faults and error conditions, and re-initialized all the hardware and software components.

4.2 Entering Configuration Mode

Some functions are only enabled when the controller is in a special “configuration” mode¹ to prevent potentially harmful effects such as accidentally changing the device’s address, baud rate, or other configuration parameters.

¹In some places in other documentation, including the controller’s firmware source code, this is also referred to as “privileged mode”.

To initiate configuration mode, press and hold the green Option button until the LEDs start flashing rapidly (approximately 2 seconds). Then release the Option button until the LEDs fade to a state where the green LED(s) are flashing rapidly.²

The board is now in configuration mode and can be instructed by the host PC to change critical settings.

To leave configuration mode, the host PC may issue a command to cancel the mode, or you may press the red Reset button (which also has the effect of rebooting the system).

4.3 Running a Test Pattern

When setting up a board in the field, it may be helpful to manually have the board turn on its output channels as a test that everything is connected and working properly.

To do this, first place the board in configuration mode as described in Section 4.2. After the green light is flashing rapidly and the others are off, press and hold the Option button again until the lights go off, then release.

The red LED will now be slowly fading up and down (instead of the green LED doing that as it does in normal run mode). The controller will turn off all output channels, and turn channel #0 on fully.

After a one-second delay, channel #0 will turn off and #1 will turn on. This will continue indefinitely, each channel turning on its turn each second. The panel LEDs will display the currently-active channel number in binary. Note that 24-channel boards only show the least significant 3 bits, so you will see which channel within the block (0–7) is active, but not which block. 48-channel controllers show the entire channel number. See Figure 4.1 for a reference chart of the output codes.

If the Option button is pressed briefly, the pulsing red LED freezes (steady on) and the cycle pauses on the current output channel. Pressing the Option button again resumes the cycle.

4.4 Restoring to Factory Settings

If the board is unresponsive and cannot be reconfigured via the host PC (for example, if you configured it to use a baud rate your host PC can't match), follow these steps to reset the device to its factory default settings:

1. Press and hold the Reset button.
2. While holding Reset, press and hold Option.
3. While continuing to hold Option, release Reset.
4. When the red LED lights, release the Option button.

²In normal run mode, the green LED(s) are slowly fading up and down.

48-ch 24-ch Channel	48-ch 24-ch Channel
	Channel #0
	Channel #1
	Channel #2
	Channel #3
	Channel #4
	Channel #5
	Channel #6
	Channel #7
	Channel #8
	Channel #9
	Channel #10
	Channel #11
	Channel #12
	Channel #13
	Channel #14
	Channel #15
	Channel #16
	Channel #17
	Channel #18
	Channel #19
	Channel #20
	Channel #21
	Channel #22
	Channel #23
	Channel #24
	Channel #25
	Channel #31
	Channel #32
	Channel #39
	Channel #40
	Channel #46
	Channel #47

Figure 4.1: Test-mode Channel Indicators

5. When the red LED turns off, press and hold the Option button again.
6. When the red LED turns on, release the Option button.

If those steps were carried out exactly as described, the LEDs will all flash rapidly and the controller will reboot with the factory settings in place. This means, among other things, it will be at address 0, set to use Lumos commands only (not DMX512), and will communicate at 19,200 baud.

Failure to perform each step at the right time will prevent the factory reset from occurring.

Note: If the red LED (the top one for 48-channel boards) is configured as a sensor input, one of the other LEDs will be used instead. *If all of the LEDs have been replaced with sensor inputs, then channel #0 will be turned on and off instead, to signal this sequence. Have a suitable load plugged into that output.*

C H A P T E R



CONFIGURING THE BOARD

HERE ARE A NUMBER OF SETTINGS which can be made on the Lumos controllers. Generally, this is accomplished by using the `lumosctl` command-line program on the host PC. The basic configuration options include such things as setting the device address, communication speed, etc. These can be set directly on the command line with `lumosctl`.

In the following examples, the command-line prompt is represented by a ‘\$’ character. Text that is typed literally as shown is printed in fixed-width type, while values which are to be replaced with a value appropriate for your usage are printed in *(Italics)* in angle brackets (the angle brackets are not actually typed, however).

For every command, you need to include three values:

- The serial port on the host PC to which the Lumos controller is attached. This is given to `lumosctl`’s `--port` option, as “`--port=<name>`”. On many systems, it is enough to say “`--port=0`” or “`--port=1`” to specify the first or second “standard” port. A specific port name may be given, such as “`--port=COM1`” or “`--port=/dev/ttys0`”.
- The speed (baud rate) at which to communicate with the Lumos controller. This is the speed the controller is *currently* using, not the one you want to change it to. This is given as “`--speed=<rate>`”, such as “`--speed=19200`”.
- The Lumos controller’s address, given as “`--address=<addr>`”. For example, “`--addr=0`”.

If you aren’t sure what device addresses exist on the serial line, you can have `lumosctl probe` to discover them all:

```
$ lumosctl --port=COM1 --speed=19200 --probe
Probe discovered 2 devices:
Address 00: lumos48ctl
Address 04: lumos24dc
$ _
```

If that doesn't discover everything you thought it should, you can add `--verbose` to the command to see more detail. Adding more and more `--verbose` options increases the amount of information printed. The default port is the "first" standard port, the default speed is 19,200 bits per second, and the default address is 0.

If you want to print a full report of the state of a device, include the `--report` option:

```
$ lumosctl --port=COM1 --speed=19200 --address=0 --report
```

The `--port`, `--speed`, and `--address` options assume a reasonable default value if they were not specified. For the sake of simplicity, we'll assume from here on that you either accept these defaults or are specifying them to have the values you need; we won't explicitly show them in the examples that follow.

5.1 Setting the Device Address

Set a new address by giving `--set-address=<newaddr>`. Don't forget to include the device's current address with `--address=<oldaddr>`. For example, to change a board from address 0 to 12:

```
$ lumosctl --address=0 --set-address=12
```

Once the address is set, you'll need to use that value for `--address` from that point forward.

5.2 Setting the Device Speed

Set a new speed by giving `--set-baud-rate=<newspeed>`. Don't forget to include the device's current speed with `--speed=<oldspeed>`, so it has any hope of seeing the command to change it. For example, to change a board from 19200 to 57600 baud:

```
$ lumosctl --speed=19200 --set-baud-rate=57600
```

Once the speed is set, you'll need to use that value for `--speed` from that point forward.

5.3 Sensors

If your board is built to accommodate sensor inputs, you need to set the EEPROM settings so the controller stops driving those lines as outputs and starts watching them as inputs.

To do this, use the `--dump-configuration=<file>` option. This dumps the device's configuration state into a text file on the host PC.

```
$ lumosctl --dump-configuration=lumos_board.cfg
```

Looking in the `lumos_board.cfg`, find a section beginning with “[lumos_device_settings]”. there is a field “`sensors=<list>`” which should contain a list of all the sensors configured as inputs.

So if we had lines A and C wired up to sensor inputs instead of LED outputs, we need to change this line in the `lumos_board.cfg` file to read:

```
[lumos_device_settings]
sensors=ac
```

Leave the other fields alone, just as they are.

For each sensor, we can arrange for an action to take place every time one of them activates. The actions taken are set up as “Programmed Sequences” (see Chapter 6). Assuming that the actions we want to carry out are already programmed and loaded as described in that chapter, we associate those sequences with sensor inputs by introducing a new section in the `lumos_board.cfg` file called `[lumos_device_sensor_<x>]`, where `<x>` is the sensor letter (or we edit that section, if it's already in the file).

For example, to set sensor A to play sequence 100 when it first activates, then continue to loop sequence 101 as long as the sensor remains active, then finally play sequence 102 as soon as the sensor stops being active, and assuming we want “active” to mean when the signal on that input is at a logic 1 level (active high), we would put this in the file:

```
[lumos_device_sensor_a]
enabled=yes
mode=while
setup=100
sequence=101
terminate=102
active_low=no
```

If we want sensor C to be active low, and trigger sequence 42 one time when it activates, our file needs this section added to it:

```
[lumos_device_sensor_c]
enabled=yes
mode=once
sequence=42
active_low=yes
```

Once the file is set up with all the configuration changes you wish to make, it may be loaded back to the board again with the `--load-configuration=<file>` option:

```
$ lumosctl --load-configuration=lumos_board.cfg
```

Assigning sequences to as sensor-triggered events may also be arranged on the fly via the `lumosctl` program. The same effects could be performed thus:

```
$ lumosctl --sensor=aw+:100:101:102 --sensor=co::42:
```

Note that configuring a sensor line as an input or output *must* be done from the configuration file.

See Chapter 6 for complete details on how to create sequences and store them into a Lumos controller. Full details on the operation of `lumosctl` in this and other areas begins on page 81 in the appendices.

5.4 Setting a Lumos Controller to Use DMX512

Setting up the controller to be a DMX512 device is another task performed via the configuration file (see Section 5.3 for instructions about how to dump and load a configuration file).

First, get the current configuration settings into a file:

```
$ lumosctl --dump-configuration=lumos_board.cfg
```

In the [lumos_device_settings] section, add a field “dmxchannel=<c>” where <c> is the starting slot number you wish the Lumos controller to use. This will be channel #0 on this board.

To cause a 24-channel Lumos board to occupy DMX slots 200–247, this would be:

```
[lumos_device_settings]
dmxchannel=200
```

Now download that configuration into the controller board:

```
$ lumosctl --load-configuration=lumos_board.cfg
```

If you want to change the board to be a Lumos-protocol board instead of DMX512, just follow the same process, but *delete* the dmxchannel=<c> line completely before loading the configuration onto the Lumos board.

5.5 Canceling Configuration Mode

When you’re finished configuring the controller board, you may use either of these options:

```
$ lumosctl --disable-privileged-mode
$ lumosctl --forbid-privileged-mode
```

Either of these returns the board to normal run mode. The second goes a step further, forbidding the board from going back into configuration mode again, until the next time the board is reset or power cycled.

C H A P T E R



CREATING PROGRAMMED SEQUENCES

THE NORMAL OPERATING MODE for a Lumos controller is to receive a stream of real-time commands from the host PC, acting on them as they arrive. However, it is also possible to program a number of complex actions into the controller's memory. The host PC can then send a much smaller command to tell the controller to execute a stored sequence of actions. If sensor inputs are enabled, the board may be set up to trigger the execution of a stored sequence when sensor activity is detected.

N.B.: The following information assumes a high-level (if somewhat simple) language to describe sequences, which gets compiled into a more compact bytecode which is loaded onto the device itself. This language compiler has not yet been written. Section 6.10 below describes the bytecode actually understood by the device. At the time of this writing, it is necessary to create this bytecode manually and load it into the device as described in that section.

These sequences are written using a small, simple programming language. It is intended to make it possible to describe some basic operations supported by the device, not to create a fully general-purpose programming system. This, plus the need to design the sequence data to be as compact as possible, drove the design of this language to be optimized toward efficiency and to enable a certain set of common operations. This optimization means, however, that some other functions are limited or omitted. We'll point out the major caveats along the way as we describe this system in this chapter.

6.1 Basic Sequence Syntax and Structure

Each sequence is numbered with an ID from 0–127. The sequence begins with a declaration of this ID in the form “SEQUENCE $\langle id \rangle$:” followed by all the lines of instructions, and ends with the word “END” on the final line.

When writing a sequence of instructions to be stored into the controller, each individual instruction is placed on a line by itself. For example, if we define sequence #1 to set channel #0 on, #1 to half brightness, and #2 off, the sequence code would be:

```
sequence 1:
    channel 0 on
    channel 1 50%
    channel 2 off
end
```

The indentation is simply a convention to make the code more readable; the Lumos system ignores it. Similarly, Lumos sequences are case-insensitive. The previous sequence could just as well have been written:

```
SEQUENCE 1:
    Channel 0 ON
    Channel 1 50%
    CHANNEL 2 Off
END
```

6.2 Expressions

Anywhere a number is expected, a complex math expression may be used. Standard algebraic order of operations is enforced on the expression, so parentheses force the enclosed sub-expression to be evaluated first, then multiplication and division, then addition and subtraction. All of the following instructions turn channel 10 to 1/3 brightness:

```
channel 10 33.3333333333%
channel 2*(2+3) 255/3
channel 10 255.0/3.0
```

Note: If the expression is able to be fully interpreted using known values at the time the source code is compiled on the host PC, it is evaluated with all the precision of floating-point math operations to the extent supported by the PC. The final result of the calculation, however, needs to be

Expression	Meaning
$x + y$	add $x + y$
$x - y$	subtract $x - y$
$x * y$	multiply $x \times y$
x / y	divide x/y
$x // y$	divide x/y using integer division
$x \% y$	the remainder when dividing x/y

Figure 6.1: Supported Math Operations

reduced to (in most cases) an unsigned 8-bit integer value (0–255) for storage on the Lumos controller. If the expression must be interpreted at runtime on the Lumos device itself, there's a much more limited expression evaluation system available at that point. Only addition, subtraction, and multiplication of unsigned 8-bit values is possible there. The compiler may reject your expression if it can't be handled within those constraints.

The math operators supported are summarized in Figure 6.1.

6.3 Variable Assignment

To a rather limited extent, variables may be defined values and then used in sequences in place of numeric values, like so:

```
x = 42
channel x on
channel x+1 on
channel x+2 on
```

6.4 Controlling Output Channels

We already saw the CHANNEL command which can be used to turn a designated channel ON, OFF, or to a given percentage. It may also be given a raw value from 0–255. All of the following set channel #4 to full brightness:

```
channel 4 100%
channel 4 255
cahnnel 4 on
```

We may also start a channel smoothly fading up or down from its current brightness to either fully on or fully off. The following instructions cause channel #12 to fade to black by reducing its output value by 2 every .2

seconds. Simultaneously, channel #13 will increase in brightness by 1 step every 1/120 second.

```
ramp down 12 by 2 per .2 seconds
ramp up 13
```

There is also a single command for turning all output channels off at once:

```
blackout
```

6.5 Device State Changes

There are commands SLEEP and WAKE to turn on and off sleep mode if desired. These control power-saving features but don't disrupt the interpretation or execution of incoming commands.

There are also SUSPEND and RESUME commands. SUSPEND causes the controller to ignore any further input, on the premise that the sequence wants to take priority over whatever the host PC wanted the controller to do. RESUME allows the board to resume normal operation (acting on received commands from the host PC).

If the command SUSPEND WITH UPDATE is given, then the device will track changes to channel outputs made by the host PC, but still will not act on them until a RESUME command (at which point, the output channels will simply change to what would be the final result of all the changes received—it does not "replay" all of the changes that were delayed at that point).

The RESUME WITH FADE is just like RESUME, except it will fade all channels to black first, then fade up the channels to their resume values.

For example, if you have a light sensor configured to run sequence 1 once when it senses daylight, and then run sequence 2 when it senses darkness, and you want to inhibit outputs during the daytime but still allow the host PC to set a pattern of channel values which will be displayed immediately upon dusk, the sequences could be coded:

```
sequence 1:
    suspend with update
    blackout
end

sequence 2:
    resume with fade
end
```

6.6 Flow Control

In addition to the linear sequence of instructions to change channel outputs, there are a number of instructions which alter the flow of execution itself. This includes the usual collection including conditionals, loops, and so forth.

If you want to have sequence 123 turn on channel #4 if sensor B is active at that moment, or turn it off otherwise, the following code will suffice:

```
sequence 123:
    if sensor b:
        channel 4 on
    else:
        channel 4 off
    end
end
```

We can turn on every even numbered channel from #0–#10, with a half-second delay up through #6 and then instantly for the rest:

```
sequence 47:
    for ch=0 to 10 by 2:
        channel ch on
        if channel <= 6:
            wait 0.5 seconds
        end
    end
end
```

We can flash channels #1 and #3 repeatedly until our sequence is forced to stop:

```
sequence 9:
repeat:
    channel 1 on
    channel 3 on
    wait 1 second
    channel 1 off
    channel 3 off
    wait 1 second
end
end
```

Loops can be interrupted using BREAK, CONTINUE, and EXIT. See the `lumosctl` documentation (page ??) for full details.

6.7 Subroutines and Sequence Chains

One sequence can invoke another sequence, either as a subroutine (so that the execution temporarily detours into the called sequence, and then the calling sequence resumes where it left off), or as a “chain” from one sequence to another (where the calling sequence stops as the called sequence starts, so there is no returning from that point).

Here we have sequence 12 which calls sequence 13 in a loop:

```
sequence 12:
repeat 10 times as i:
    call 13
    wait i seconds
end
execute 14
end

sequence 13:
channel 0 on
wait 2
channel 0 off
end

sequence 14:
blackout
end
```

Note that the EXECUTE instruction terminated sequence 12 and started sequence 14. When sequence 14 finishes, nothing further is done.

Sometimes it's valuable to be able to pass parameters to subroutines. Sequences support up to four parameters which must be declared in both the definition of the sequence and the EXECUTE or CALL which invokes it. Here's a program which turns on each channel from 0–24 repeatedly.

```
sequence 1:
repeat:
    for c=0 to 24:
        call 2(c)
    end
end
```

```

end

sequence 2(chan):
    channel chan on
    wait 1 second
    channel chan off
end

```

6.8 Temporary vs. Permanent Sequences

Sequences numbered from 0–63 are stored into non-volatile EEPROM memory and will remain in place until explicitly erased or replaced. Sequences numbered 64–127 are stored in temporary RAM and will be lost at the next device reset. This enables the host PC to download sequences for short-term temporary use.

6.9 Storing Sequences into Device Memory

The device files are stored into the Lumos controller using the `lumosctl` command:

```

$ lumosctl --clear-sequences
$ lumosctl --load-sequence=my_program_file

```

The initial `--clear-sequences` option is not always needed. It erases all sequences from memory. Then each subsequent `--load-sequence` option loads a file containing one or more sequences into the device.

If a sequence is loaded which already exists on the device, the old one is replaced with the new one. Note, however, that Lumos controllers have a very naive memory management capability. They are really designed to simply have a set of sequences loaded on them, later erased *en masse* and replaced by a new set.

If replacement sequences are written on top of old ones, it is quite likely that the memory will get fragmented quickly and you'll run out of available memory too early. Simply adding new sequences shouldn't be a problem, however.

You can see how much memory is available by running `lumosctl` with the `--report` option.

```

$ lumosctl --report --address=2
...

```

XXX
\$ _

6.10 Bytecode Reference

This is an advanced reference section most users will never have any reason to read. For the curious, however, we note that when you load a sequence source file into lumosctl, it compiles the source into bytecode which is what actually gets sent to the controller, and is what is stored in its memory. The following reference details the format of that bytecode.

Execution Model

The execution model is a small stack-based¹ machine environment, in which each sequence is an independent program which begins at address 0 in its own memory area.

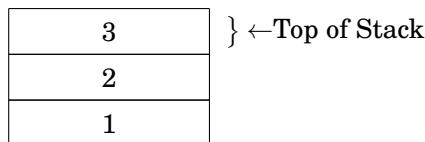
All memory locations, including program and stacks, are 8 bits wide. Unless otherwise indicated, data values are unsigned 8-bit integers.

Stacks

There are actually three independent stacks used: a call-return stack, data stack, and loop counter stack.

The call-return stack (not illustrated here since it's not directly accessible from your programs) receives three bytes every time a sequence calls another sequence: the return address, sequence number, and stack frame boundary pointer.

We can conceptually picture the stacks as “piles” of data values, with the most recently-placed value set on top of the previous ones. If we “push” three values 1, 2, and 3 onto a stack, it would look like this:



As we “pop” values off the stack, we pull them from the top of the stack, so they are retrieved in the reverse order in which they were pushed onto the stack.

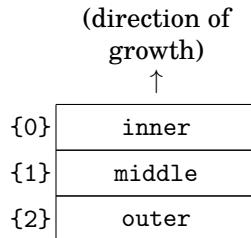
¹Several other models were experimented with, but this provided better flexibility with a minimal number of bytes required for the sequence instruction bytes themselves, which need to be stored in a limited space on the controller.

Loop Counter Stack

The loop counter stack holds all the loop counter values. If the code is executing a nested loop such as:

```
repeat as outer:
    for middle=1 to 10:
        repeat 10 as inner:
            ...
        end
    end
end
```

The loop stack would look like:



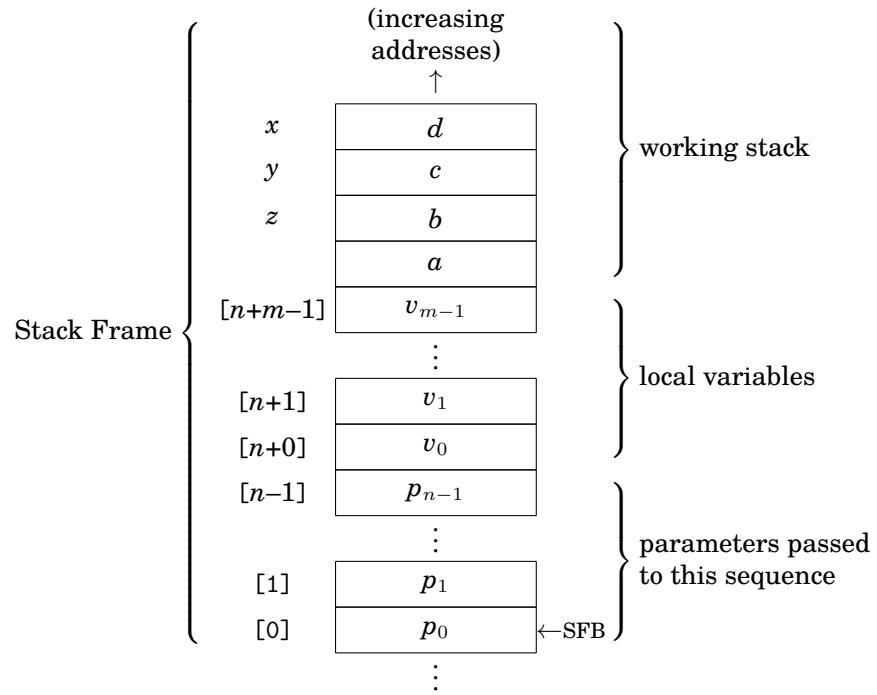
At any given time, {0} refers to the top of the loop stack, which is the innermost loop level currently running. Likewise, {1} is the counter for the next loop out from there, and so on.

Note that a counter variable is allocated for *every* loop entered, even if the source code didn't associate it with a variable name (e.g., if the innermost was "REPEAT 10:" without the "AS Z" part).

Data Stack

The data stack contains the parameters passed to the current sequence, if any, followed by local variables for the sequence. The remaining room is for working stack space for calculations and values consumed by the opcodes.

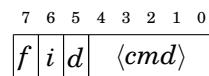
At each call, parameters are pushed onto the stack and then the stack frame boundary (SFB) pointer is moved to point to the first parameter. This way, parameters and local variables are referenced within the sequence as offsets from the frame pointer, while working values are obtained by pushing and popping values from the top of the stack.



For convenience in our description of the bytecode below, we'll refer to the value on the top of the data stack as x , the second value as y , and the third as z . In the assembly code, the notation $[n]$ directly addresses the value n bytes from the stack frame boundary (SFB), so the first parameter passed (if any) would be at $[0]$, and so on.

Instruction Format

Each instruction is at least one byte long. The first byte is the opcode which always has the general format:



Where:

f = Instruction-specific flag.

i = Data values are given as immediate bytes ($i=1$) instead of being popped from the stack ($i=0$).

d = Take default values for some or all of the arguments ($d=1$).

Note that these are generalizations; individual command opcode patterns vary.

Following the opcode byte are any direct addressing bytes (generally, addresses of locations on the stack for data values, or addresses to jump to) which must be constants at runtime.

After that are zero or more data bytes providing the values needed by the instruction, if the immediate addressing mode is used (bit $i=1$). Otherwise they will be popped from the data stack as needed.

Addressing Modes

The following addressing modes are supported:

- *Default addressing* means that some or all data values are omitted, (which may be omitted depends on the instruction), so the instruction should assume a useful default value instead. Denoted below as either the lack of argument, or an explicit mention of the default value (in which case the assembler would convert the instruction to the default form and avoid the argument).
- a *Direct addressing* specifies a memory location as a constant value following the opcode. Examples include jump and loop instructions which directly name an instruction address within the sequence, and operations which specify a local variable or parameter as a constant offset from the stack frame pointer. Note that some instructions mandate a direct argument even if others are pulled from the stack, memory, or take defaults. Denoted below as a number (or name defined as a number) without other punctuation.
- #v *Immediate addressing* mode means that the values needed by the instruction are provided as constant bytes following the opcode, rather than being taken from the stack. Denoted below as a value prefixed by a pound sign (#).
- [a] *Indirect addressing* is used when a data value needed by an operation is specified, but that value specifies an address in memory (specifically, the value is an unsigned integer added to the stack frame pointer). Denoted by square brackets around the argument value.
- [\$] *Indirect Stack* values are obtained by popping an address from the stack and then using it as the location for the value to use. Noted by square brackets and dollar sign together ([\\$]).
- *Inherent* means that the instruction simply needs no further values to perform its operation; the data is inherent in the instruction itself.
- \$ *Stack addressing* means that the data values are popped from the stack. These are popped in the order they would appear if given as immediate bytes, so they would need to be pushed in opposite order to that. Denoted by a dollar sign (\$) as the argument to the instruction.

{v} *Loop addressing* accesses loop counter values. {0} is the innermost loop counter (the top of the loop stack), {1} is the next loop out from there, etc. This can be combined with some other modes. For example, {\$} pops a value from the stack, uses that as the loop depth number, and fetches the loop counter at that level.

The addressing mode is selected by various bits in the opcode byte.

For example, the instruction to set the dimmer output level for a channel (mnemonic “dimmer”, base hex opcode 0x02) may have the following forms:

	byte 0	byte 1	byte 2	
immed.	0 1 0	0x02	0x01	0x10 dimmer #1,#16
stack	0 0 0	0x02		dimmer \$,\$

In the first case, the dimmer command takes the channel number (1) and level (16) as immediate values in the sequence program itself. In the second case, it pops two values off the stack, the first number popped being the channel number and the second one being the value.

Consider the push command as well:

	byte 0	byte 1	
inherent	0 0 0	0x0F	dup
default	0 0 1	0x0F	push #0
immediate	0 1 0	0x0F	push #9
loop default	0 1 1	0x0F	push {0}
indirect stack	1 0 0	0x0F	push [\$]
indirect default	1 0 1	0x0F	push [0]
indirect	1 1 0	0x0F	push [9]
loop stack	1 1 1	0x0F	push {\$}

Some of these combinations are special only for the PUSH and POP commands. The first mode is somewhat special as well. Since “PUSH \$” would not do anything particularly useful—pop a value from the stack and then put it right back again—this opcode was reassigned to a similar but more useful operation. Renamed DUP, it takes the value on the top of the stack and pushes another copy of it.

Using the Value Stack

The operation of the loop stack and call stack take place behind the scenes where you don't really need to be too concerned about them. The data stack, on the other hand, is important to understand if you'll be writing sequences at the low level of the bytecodes described here.

The stack has a maximum depth of 256 bytes. Values are pushed onto it (usually via the PUSH instruction) and those values are consumed by other instructions as needed. A \$ appearing in an instruction's argument list means to pop a value off the stack for that parameter.

Consider the following sequence of instructions:

```
base=2
for x=1 to 20:
    on x
    dimmer x+10+base 6*x+12
    wait 100
    base *= 3
end
```

This would be compiled into the following sequence bytecode:

ADDR	BYTES	LINE	SOURCE	CODE
0000	4F 02	0002	PUSH	#2
0002	AA	0003	LP1:	LOOP #1
0003	6F	0004	PUSH	{0}
0004	81	0005	ON	\$
0005	6F	0006	PUSH	{0}
0006	52 06	0007	MUL	\$,#6
0008	53 0C	0008	ADD	\$,#12
000A	6F	0009	PUSH	{0}
000B	53 0A	0010	ADD	\$,#10
000D	AF	0011	PUSH	[0]
000E	13	0012	ADD	\$,,
000F	02	0013	DIMMER	\$,,
0010	49 64	0014	WAIT	#100
0012	AF	0015	PUSH	[0]
0013	52 03	0016	MUL	\$,#3
0015	B0	0017	POP	[0]
0016	6B 02 14	0018	NEXT	LP1,#1,#20

When executed, this is the sequence of steps carried out, and the stack's contents at each step. Here's the initial state:

Instruction:	Value Stack:	Loop Stack:
[0] x	(empty)	$\leftarrow SFB \{0\}$ (empty)

Now we begin execution. First, we create a local variable “base” by pushing its initial value (2) onto the stack. This will be at address zero (relative to the base stack frame pointer), so “[0]” will directly reference this storage location.

PUSH #2	[0] x	2	$\leftarrow SFB \{0\}$	(empty)
---------	---------	---	------------------------	---------

A new loop context is created with an initial counter value of 1 (now the top of the loop stack).

LOOP #1	[0] x	2	$\leftarrow SFB \{0\}$	1
---------	---------	---	------------------------	---

Next we turn on the channel specified by the value of the loop counter, which is on the loop stack at {0}. We push that counter onto the data stack then use that as the argument to the ON instruction.

PUSH {0}	[1] x	1	
	[0] y	2	$\leftarrow SFB \{0\}$ 1

Since the value at the top of the stack (x) is 1, the instruction ON \$ pops that value and turns on channel 1.

ON \$	[0] x	2	$\leftarrow SFB \{0\}$	1
-------	---------	---	------------------------	---

Now we need to evaluate $6*x+12$.

PUSH {0}	[1] x	1	
	[0] y	2	$\leftarrow SFB \{0\}$ 1

The 1 on the top of the stack is popped off, multiplied by 6, and the product pushed back onto the stack:

MUL \$,#6	[1] x	6	
	[0] y	2	$\leftarrow SFB \{0\}$ 1

The product is popped, added to 12 and the sum pushed back:

ADD \$,#12	[1] x	18	
	[0] y	2	$\leftarrow SFB \{0\}$ 1

Now that we have the value to be set on the channel (18), we calculate the channel number as $x+10+base$:

PUSH {0}	[2] <i>x</i>	1		
	[1] <i>y</i>	18		
	[0] <i>z</i>	2	←SFB {0}	1
ADD \$,#10	[2] <i>x</i>	11		
	[1] <i>y</i>	18		
	[0] <i>z</i>	2	←SFB {0}	1

The value of the local variable base is at local address 0 (i.e., SFB+0, also known as “[0]”), so we push that onto the stack then add it to our sum.

PUSH [0]	[3] <i>x</i>	2		
	[2] <i>y</i>	11		
	[1] <i>z</i>	18		
	[0] <i>z</i>	2	←SFB {0}	1
ADD \$\$,\$	[2] <i>x</i>	13		
	[1] <i>y</i>	18		
	[0] <i>z</i>	2	←SFB {0}	1

Now we pop the channel number and level off the stack so channel 13 is set to dimmer level 18.

DIMMER \$\$,\$	[0] <i>x</i>	2	←SFB {0}	1
----------------	--------------	---	----------	---

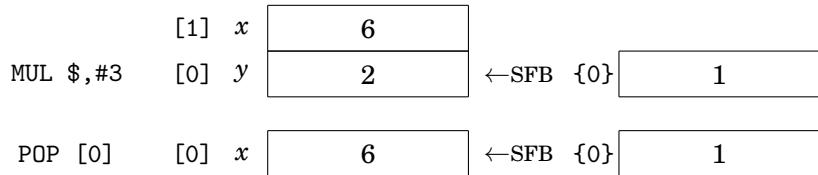
Now we wait, which doesn’t affect the stack.

WAIT #100	[0] <i>x</i>	2	←SFB {0}	1
-----------	--------------	---	----------	---

To update the value of the variable base, we push its current value from memory location [0], multiply it by 3 to get the new value, and then pop that result back into memory location [0] again.²

PUSH [0]	[1] <i>x</i>	2		
	[0] <i>y</i>	2	←SFB {0}	1

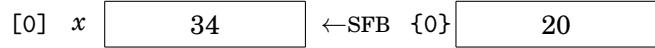
²“Wait a moment,” you may be asking yourself at this point, “The value of base is sitting *right there* on the stack, why not just execute the single instruction MUL #3 and accomplish the same thing in one step?” The answer is that you can, in this particular case. In some sequences, though, there may be other variables in the way, so the approach outlined here will work in the general case. If you’re hand-compiling your own bytecode, you can make optimizations like that.



We then start the next iteration.



... and so on, until we reach the end of the final iteration:



(Note that the value in [0]—the local variable “base”—is multiplied by $3 \times$ every iteration, so although the value in the 8-bit memory location is currently 34, it’s overflowed 13 times already. Remember that you are working with only 8-bit values on the Lumos device.)

When the NEXT instruction is executed, it determines that the loop exit condition has been met, and destroys the loop context, does *not* jump to the top of the loop, and execution will continue on past the loop now.



Subroutine Calls and Parameter Passing

The basic unit of execution is a sequence itself, so when we speak of calling a “subroutine” we really mean one sequence is calling another sequence (or recursively calling itself).

A number of parameters may be passed to the called sequence, which will appear as local variables to it. This is done by pushing their values onto the stack before making the call. Part of the call itself moves the stack frame boundary so that location [0] refers to the first parameter. The other parameters, if any, follow. If the called sequence sets up local variables, they’ll be pushed after the parameters (which means it is important that the sequences all agree about how many parameters are being passed and received).

To illustrate this, consider the following pair of sequences:

```
sequence 1:
delay=10
for i=0 to 255 by 4:
    for j=20 to 10 by -1:
```

```

call 5(i,j)
wait delay
end
delay += 10
end
end

sequence 5(val,ch):
foo=0x42
dimmer ch, val
on ch+20
off ch+20
end

```

This compiles into the following bytecode:

SEQUENCE #1 ASSEMBLY LISTING		
ADDR	BYTES	LINE SOURCE CODE
0000	4F 0A	0002 PUSH #10
0002	2A	0003 I_LOOP: LOOP #0
0003	4A 14	0004 J_LOOP: LOOP #20
0005	4F 01	0005 PUSH #1
0007	EF	0006 PUSH {\$}
0008	6F	0007 PUSH {0}
0009	48 05 02	0008 CALL #5,#2
000C	AF	0009 PUSH [0]
000D	09	0010 WAIT \$
000E	75 03 0A	0011 NEXTDEC J_LOOP,#1,#10
0011	AF	0012 PUSH [0]
0012	53 0A	0013 ADD \$,#10
0014	B0	0014 POP [0]
0015	4B 02 04 FF	0015 NEXT I_LOOP,#4,#255
0019	00	0016 EXIT

SEQUENCE #5 ASSEMBLY LISTING		
ADDR	BYTES	LINE SOURCE CODE
0000	40 02	0019 SSFB #2
0002	4F 42	0020 PUSH #0x42
0004	AF	0021 PUSH [0]
0005	CF 01	0022 PUSH [1]

0007 02	0023	DIMMER	\$,\$
0008 CF 01	0024	PUSH	[1]
000A 53 14	0025	ADD	\$,#20
000C 0F	0026	DUP	
000D 81	0027	ON	\$
000E 01	0028	OFF	\$
000F 00	0029	EXIT	

We'll step through the execution of these routines to show how the stack frames look between calls.

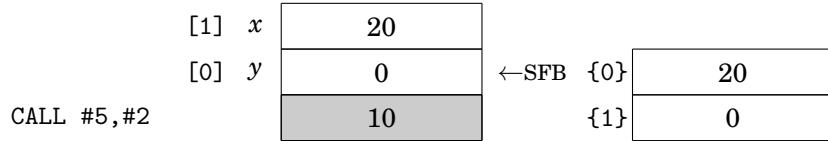
First, sequence 1 starts running, pushes its local variable to the stack, and enters its loops:

PUSH #10	[0] x	10	←SFB {0}	(empty)
LOOP #0	[0] x	10	←SFB {0}	0
LOOP #20	[0] x	10	←SFB {1}	{0} 20 0

We push the two parameters onto the stack in the order shown in the source code (i.e., the deepest value will be the first parameter). Since the call is "CALL 5(x,y)" we need to push the outer loop counter x (which is at position {1} on the loop stack), then the inner loop counter y (which is at position {0}).

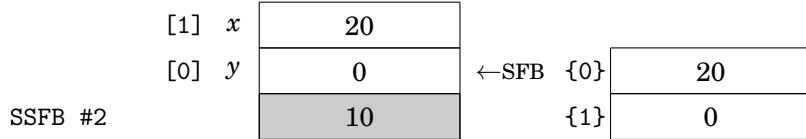
PUSH #1	[1] x	1	{0}	20
	[0] y	10	←SFB {1}	0
PUSH {\$}	[1] x	0	{0}	20
	[0] y	10	←SFB {1}	0
PUSH {0}	[2] x	20	{0}	20
	[1] y	0		
	[0] z	10	←SFB {1}	0

Now we make the call. The first parameter to CALL is the sequence number we're calling, and the second is the number of parameters being passed (2 in this case).

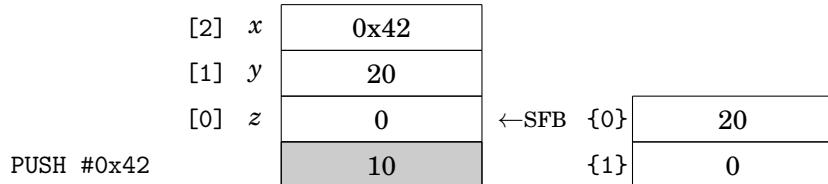


At this point, control transfers into sequence 5. Note that the stack frame boundary moved up until the frame includes only the top 2 items (the two parameters passed into this sequence). This means the parameters are directly accessible as [0] and [1], and the local variable sequence 5 is about to create will be [2]. The local variable from the calling sequence (the value 10 on the stack) is not directly accessible for now.³

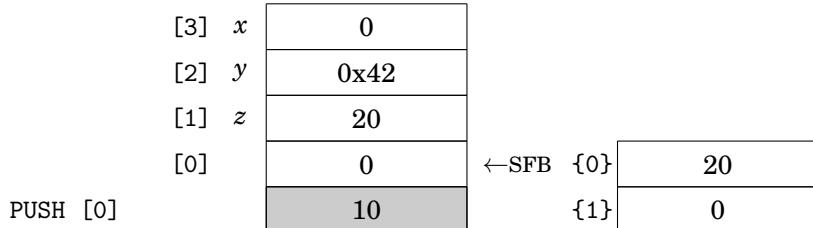
The first thing sequence 5 does is to ensure the stack frame is as expected.



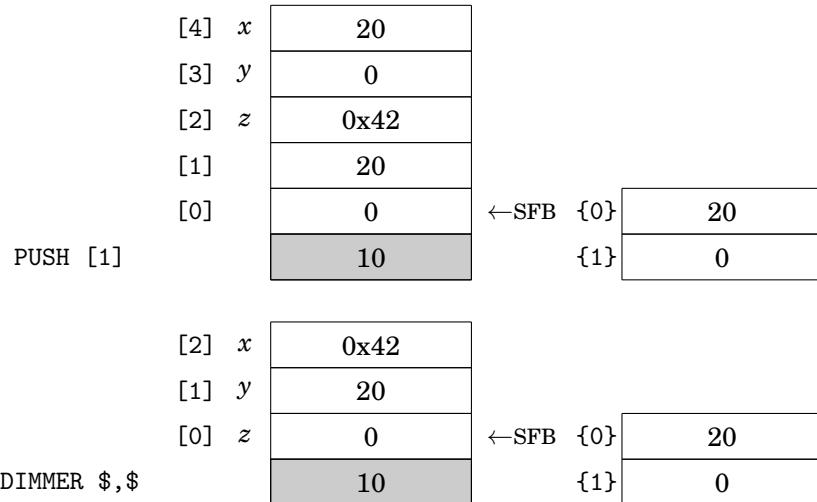
Next, it will push the initial value of the local variable `foo`, so we can get to it at location [2].



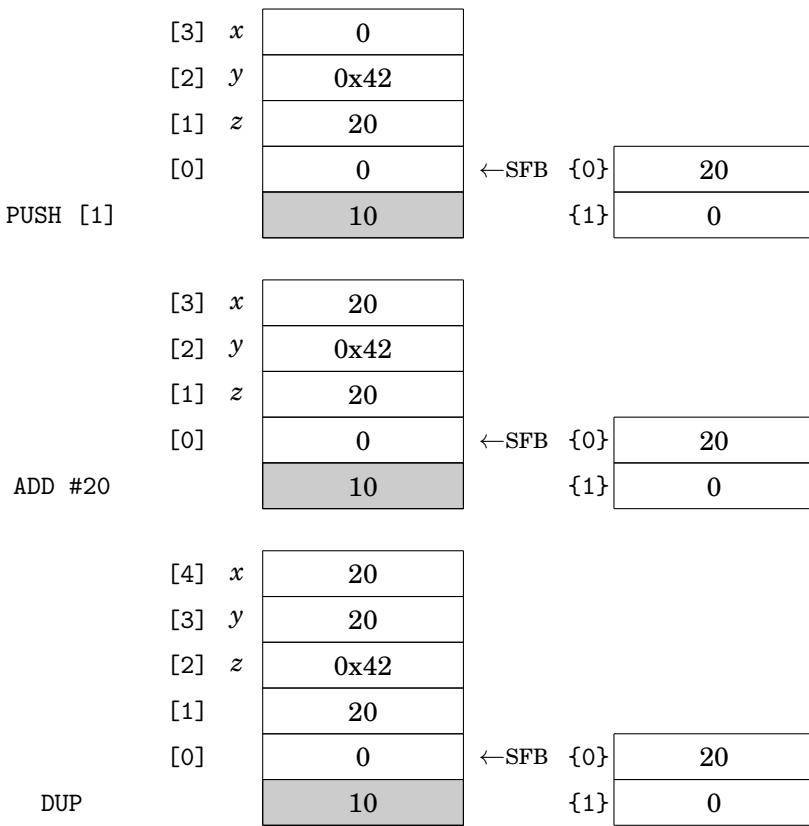
Now we set the dimmer value of channel [1] to level [0].

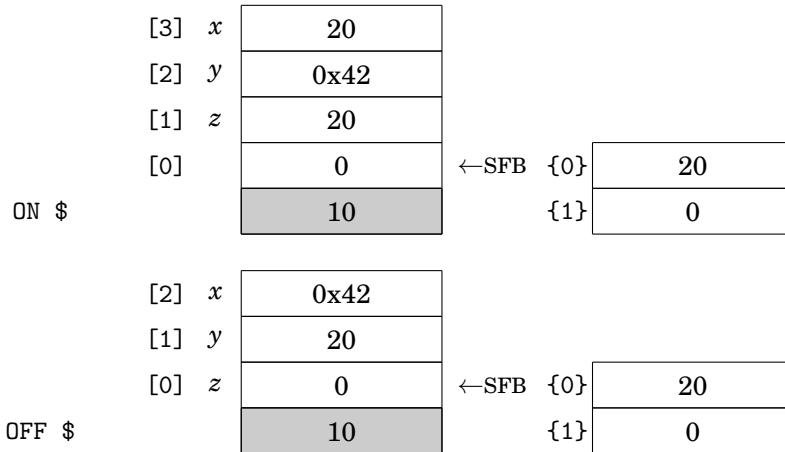


³Yes, if you're clever or careless enough, you can destroy the integrity of the stack(s) to retrieve or alter the values below SFB, but we're not recommending that.

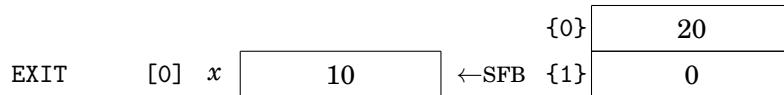


Now we calculate the value of $[1]+20$ which will be used for both the ON and OFF commands.

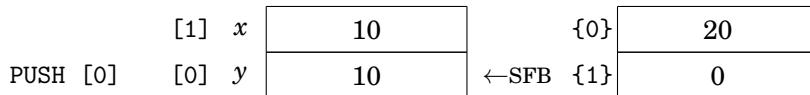




Now sequence 5 is done. As it exits, the calling sequence resumes where it left off, and the stack frame is destroyed, restoring the SFB to its previous location.



Execution of sequence 1 continues....



Instruction Set

ADD/INCR: Add Two Numbers

Adds the top two numbers on the stack (or the top number and another value), and pushes their sum back onto the stack. Note that both operands, and the sum, are unsigned 8-bit values (0–255). If the result exceeded this range, the result will be inaccurate (it will be the least significant 8 bits of the value) and the overflow flag will be raised which you can test using conditional branch instructions such as IFANY.

As a special case, the operation of adding 1 to a value is named INCR to reflect that it is simply incrementing that value by one.

Bytes	Instruction	Description
13	ADD \$,\$	$y + x \rightarrow x.$
53 $\langle v \rangle$	ADD \$,# $\langle v \rangle$	$x + \langle v \rangle \rightarrow x.$
33	INCR \$	$x + 1 \rightarrow x.$

BLACKOUT: Turn Off All Channels at Once

All channels are immediately turned off, or in the case of FADEOUT, all channels are smoothly faded down to off.

Bytes	Instruction	Description
0D	BLACKOUT	Turn off all channels.
8D	FADEOUT	Fade all channels to zero.

BREAK: Escape From Loop

This is a special form of JUMP which is designed to branch out of a loop body. In addition to jumping to the destination address, it destroys one or more loop contexts. You must ensure that the correct number of loops is specified when using BREAK to escape out of one or more loop bodies.

Bytes	Instruction	Description
83 <addr>	BREAK <addr>, \$	break out of <i>x</i> loops
A3 <addr>	BREAK <addr>	break out of innermost loop
C3 <addr> <n>	BREAK <addr>, #<n>	break out of <n> loops

CALL: Invoke Sequence as a Subroutine

Begins executing a new sequence, possibly with parameters passed into it. When that sequence completes, the current one will resume where it left off.

Bytes	Instruction	Description
08	CALL \$,\$	Call <i>x</i> with <i>y</i> params.
28	CALL \$	Call <i>x</i> with no params.
48 <seq> <np>	CALL #<seq>, #<np>	Call <seq> with <np> params.
68 <seq>	CALL #<seq>	Call <seq> with no params.

DIMMER: Set a Channel Dimmer To a Value

The specified channel <*ch*> is set to dimmer level <*lvl*> (0–255).

Bytes	Instruction	Description
02	DIMMER \$,\$	Set channel <i>x</i> to level <i>y</i> .
42 < <i>ch</i> > < <i>lvl</i> >	DIMMER #< <i>ch</i> >, #< <i>lvl</i> >	Set < <i>ch</i> > to < <i>lvl</i> >.

EXCH: Exchange Data Stack Values

Swaps the top two values on the data stack (*x* ↔ *y*).

Bytes	Instruction	Description
11	EXCH	Swap <i>x</i> and <i>y</i> .

EXEC: Start New Sequence

Begins executing a new sequence, possibly with parameters passed into it. This sequence replaces the current one. When the called sequence $\langle seq \rangle$ exits, execution returns to the caller of the current sequence (if any).

Bytes	Instruction	Description
88	EXEC \$,\$	Run x with y params.
A8	EXEC \$	Run x with no params.
C8 $\langle seq \rangle$ $\langle np \rangle$	EXEC # $\langle seq \rangle$,# $\langle np \rangle$	Run $\langle seq \rangle$ with $\langle np \rangle$ params.
E8 $\langle seq \rangle$	EXEC # $\langle seq \rangle$	Run $\langle seq \rangle$ with no params.

EXIT: Stop Execution of a Sequence

The current sequence is terminated. This will return to the previous sequence context (resuming the previous sequence if we arrived here from a CALL instruction). If this was the outermost executing sequence, then no sequence will be running anymore.

Bytes	Instruction	Description
00	EXIT	Terminate sequence.

FADE: Fade a Channel Down to Off

Incrementally decrease the dimmer level of channel $\langle ch \rangle$ by subtracting $\langle step \rangle$, with a delay of $\langle t \rangle / 120$ sec between each step, until it reaches the minimum level of 0.

Bytes	Instruction	Description
04	FADE \$,\$,\$	Fade x by y per z .
44 $\langle ch \rangle$ $\langle step \rangle$ $\langle t \rangle$	FADE # $\langle ch \rangle$,# $\langle step \rangle$,# $\langle t \rangle$	Fade $\langle ch \rangle$ by $\langle step \rangle$ per $\langle t \rangle$.

IF*: Conditional Jumps

The JUMP instruction always transfers control to the specified address. The following alternatives evaluate some condition, and if that condition is currently true, transfers the flow of the sequence execution to the specified address $\langle addr \rangle$.

Most of these compare two values for equality or inequality:

Bytes		Instruction	Description
0C00	$\langle addr \rangle$	IFEQ $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y = x$.
4C00	$\langle addr \rangle$ $\langle v \rangle$	IFEQ $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x = \langle v \rangle$.
0C40	$\langle addr \rangle$	IFLT $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y < x$.
4C40	$\langle addr \rangle$ $\langle v \rangle$	IFLT $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x < \langle v \rangle$.
0C80	$\langle addr \rangle$	IFGT $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y > x$.
4C80	$\langle addr \rangle$ $\langle v \rangle$	IFGT $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x > \langle v \rangle$.
8C00	$\langle addr \rangle$	IFNE $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y \neq x$.
CC00	$\langle addr \rangle$ $\langle v \rangle$	IFNE $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x \neq \langle v \rangle$.
8C40	$\langle addr \rangle$	IFGE $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y \geq x$.
CC40	$\langle addr \rangle$ $\langle v \rangle$	IFGE $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x \geq \langle v \rangle$.
8C80	$\langle addr \rangle$	IFLE $\langle addr \rangle, \$, \$$	Jump to $\langle addr \rangle$ if $y \leq x$.
CC80	$\langle addr \rangle$ $\langle v \rangle$	IFLE $\langle addr \rangle, \$, \# \langle v \rangle$	Jump to $\langle addr \rangle$ if $x \leq \langle v \rangle$.

The following forms test the status of a number of boolean flags in the system:

Bytes		Instruction	Description
0CCq	$\langle addr \rangle$	IFANY $\langle addr \rangle, \langle flags \rangle$	Jump to $\langle addr \rangle$ if any of $\langle flags \rangle$ true.
4CCq	$\langle addr \rangle$	IFALL $\langle addr \rangle, \langle flags \rangle$	Jump to $\langle addr \rangle$ if all of $\langle flags \rangle$ true.
8CCq	$\langle addr \rangle$	IFNONE $\langle addr \rangle, \langle flags \rangle$	Jump to $\langle addr \rangle$ if none of $\langle flags \rangle$ true.
CCCq	$\langle addr \rangle$	IFNALL $\langle addr \rangle, \langle flags \rangle$	Jump to $\langle addr \rangle$ if not all of $\langle flags \rangle$ true.

In other words, these correspond to the combination of the specified bits using logical OR, AND, NOR, and NAND operations, respectively.

The actual flag bits are specified by OR-ing together the following bits to get the final opcode value (second byte). Flags not included will not be part of the test.

Bits	Flag Name	Description
0020	Z	Last ADD/MUL/NEXT/SUB result was zero.
0010	V	Last ADD/MUL/NEXT/SUB result overflowed.
0008	A	Sensor A input is low (0 V).
0004	B	Sensor B input is low (0 V).
0002	C	Sensor C input is low (0 V).
0001	D	Sensor D input is low (0 V).

Watch for the fact that the sensors are “true” if they are at ground potential, regardless of whether any triggers configured for them are set to activate when the sensor reads high or low.

JUMP: Go to New Address

Jump to the specified $\langle address \rangle$.

Bytes		Instruction	Description
03	$\langle addr \rangle$	JUMP $\langle addr \rangle, \$$	Jump to $\langle addr \rangle + x$
23	$\langle addr \rangle$	JUMP $\langle addr \rangle$	Jump to $\langle addr \rangle$
43	$\langle addr \rangle$ $\langle offset \rangle$	JUMP $\langle addr \rangle, \# \langle offset \rangle$	Jump to $\langle addr \rangle + \langle offset \rangle$

LOOP: Start New Loop Context

Begins a new loop context, initializing the loop counter to a specified value. This pushes the value onto the loop stack. This will henceforth be accessible as “{0}” for instructions which can see loop counters, and the previous {0} is now {1} (i.e., {0} always refers to the innermost loop, {1} is the next level of loop context counting outward, etc.).

This loop context is destroyed (and the counter popped off the loop stack) by a corresponding BREAK, NEXT, or NEXTDEC.

Bytes	Instruction	Description
0A	LOOP \$	Start loop with initial value x .
2A	LOOP #0	Start loop with initial value 0.
4A $\langle n \rangle$	LOOP #(n)	Start loop with initial value $\langle n \rangle$.
AA	LOOP #1	Start loop with initial value 1.

MUL: Multiply Two Numbers

Multiplies the top two numbers on the stack (or the top number and another value), and pushes their product back onto the stack. Note that both operands, and the product, are unsigned 8-bit values (0–255). If the result exceeded this range, the result will be inaccurate (it will be the least significant 8 bits of the value) and the overflow flag will be raised which you can test using conditional branch instructions such as IFANY.

Bytes	Instruction	Description
12	MUL \$,\$	$y \times x \rightarrow x$.
52 $\langle v \rangle$	MUL \$,#(v)	$x \times \langle v \rangle \rightarrow x$.

NEXT: End Loop Body (Start Next Iteration)

This instruction (including a few variant forms as described below) marks the end of the loop body began with a LOOP instruction. It specifies an increment or decrement value to be applied to the innermost loop counter, and a maximum value (or minimum, if decrementing). If the loop counter is greater than the maximum or less than the minimum after updating it, the loop context is destroyed and execution continues with the following instruction. Otherwise, execution jumps to the address specified in the NEXT instruction, which should be the address of the first instruction after the matching LOOP (although it need not be, if you want some initialization code to take place at the start of the first iteration which is not repeated on subsequent iterations).

Bytes	Instruction	Description
0B <addr>	NEXT <addr>, \$, \$	Loop to <addr> if {0}+x>y.
2B <addr>	NEXT <addr>, #1, \$	Loop to <addr> if {0}+1>x.
4B <addr> <inc> <max>	NEXT <addr>, #<inc>, #<max>	Loop to <addr> if {0}+<inc>><max>.
6B <addr> <max>	NEXT <addr>, #1, #<max>	Loop to <addr> if {0}+1><max>.
15 <addr>	NEXTDEC <addr>, \$, \$	Loop to <addr> if {0}-x<y.
35 <addr>	NEXTDEC <addr>, #1, \$	Loop to <addr> if {0}-1<x.
55 <addr> <dec> <min>	NEXTDEC <addr>, #<dec>, #<min>	Loop to <addr> if {0}-<dec><<min>.
75 <addr> <min>	NEXTDEC <addr>, #1, #<min>	Loop to <addr> if {0}-1<<min>.
8B <addr>	NEXTINF <addr>, \$	∞ loop to <addr>, {0}+x.
AB <addr>	NEXTINF <addr>, #1	∞ loop to <addr>, {0}+1.
CB <addr> <inc>	NEXTINF <addr>, #<inc>	∞ loop to <addr>, {0}+<inc>.
95 <addr>	NXDCINF <addr>, \$	∞ loop to <addr>, {0}-x.
B5 <addr>	NXDCINF <addr>, #1	∞ loop to <addr>, {0}-1.
D5 <addr> <dec>	NXDCINF <addr>, #<dec>	∞ loop to <addr>, {0}-<dec>.

NOP: No Operation

When this instruction is executed, nothing happens.

Bytes	Instruction	Description
0E	NOP	No operation.

OFF: Turn a Channel Off

The specified channel is turned fully off.

Bytes	Instruction	Description
01	OFF \$	Turn off channel x.
41 <chan>	OFF #<chan>	Turn off <chan>.

ON: Turn a Channel On

The specified channel is turned fully on.

Bytes	Instruction	Description
81	ON \$	Turn on channel x.
C1 <chan>	ON #<chan>	Turn on <chan>.

DROP/POP: Remove a Value From the Stack

The DROP instructions remove values from the stack and discard them. The POP instructions remove values and store them elsewhere in memory (loop counters and direct memory locations).

For example, DROP \$ pops the top value off the data stack (x), and then pops x more values off and discards them.

POP [\$] removes the top value x, which is used as the address within memory (i.e., [x]) into which to store the next value y on the stack. This will remove both values from the stack.

Bytes	Instruction	Description
10	DROP \$	Drop x values, plus x itself.
30	DROP	Drop the top value (x).
50 $\langle n \rangle$	DROP # $\langle n \rangle$	Drop the top $\langle n \rangle$ values.
70	POP {0}	Store x into loop counter {0}.
90	POP [\$]	Store y into memory address in x .
B0	POP [0]	Store x into memory address 0.
D0 $\langle a \rangle$	POP [$\langle a \rangle$]	Store x into address $\langle a \rangle$.
F0	POP {\$}	Store y into loop counter for level x .

PUSH/DUP: Push a Value Onto the Stack

Adds a value to the data stack from various sources. As a special case, DUP duplicates the top value on the stack (i.e., pushes another copy of the top value).

The values pushed onto the data stack are consumed by most other commands, or removed from the stack by DROP and POP instructions.

Bytes	Instruction	Description
0F	DUP	Duplicate x .
2F	PUSH #0	Push a 0 constant value.
4F $\langle v \rangle$	PUSH # $\langle v \rangle$	Push value $\langle v \rangle$.
6F	PUSH {0}	Push loop counter {0}.
8F	PUSH [\$]	Push contents of address in x .
AF	PUSH [0]	Push contents of address 0.
CF $\langle a \rangle$	PUSH [$\langle a \rangle$]	Push contents of address $\langle a \rangle$.
EF	PUSH {\$}	Push loop counter for level x .

RAMP: Fade a Channel Up to Full Brightness

Incrementally increase the dimmer level of channel $\langle ch \rangle$ in increments of $\langle step \rangle$, with a delay of $\langle t \rangle / 120$ sec between each step, until it reaches the maximum level of 255.

Bytes	Instruction	Description
84	RAMP \$,\$,\$	Ramp x by y per z .
C4 $\langle ch \rangle \langle step \rangle \langle t \rangle$	RAMP # $\langle ch \rangle$,# $\langle step \rangle$,# $\langle t \rangle$	Ramp $\langle ch \rangle$ by $\langle step \rangle$ per $\langle t \rangle$.

RESUME: Resume From SUSPEND

Resume acting on incoming instructions from the host PC. If the RESFADE version is used, all channels are smoothly faded to black then faded up to their last-known intended values (either the levels they had when SUSPEND was invoked, or the result of the subsequent PC commands if SUSPUPD was used).

Bytes	Instruction	Description
06	RESUME	Resume command interpretation.
86	RESFADE	Fade through black and resume.

SLEEP: Shut Down Load Power

Tell the load power supply it can shut down. Requires a properly-configured compatible power supply.

Bytes	Instruction	Description
07	SLEEP	Turn off load power supply.

SSFB: Set Stack Frame Boundary

The first instruction of a sequence which expects parameters to be passed into it should be SSFB with the number of parameters expected ($\langle np \rangle$). This will reserve enough space on the stack and initialize it with zero values.

Ideally, if another sequence called this one, it will have already pushed parameters and moved the stack frame boundary (SFB) so that those parameters are the only items in the current stack frame. When the called sequence executes SSFB, if this is the case, and the number of values in the stack frame equals $\langle np \rangle$, nothing further is done. If the number of values is less than $\langle np \rangle$, additional zero bytes will be pushed onto the stack to make the stack have exactly $\langle np \rangle$ values from SFB upward. If the number of values in the stack frame exceeds $\langle np \rangle$, a fatal error is flagged and the sequence terminates immediately. (This latter condition means that too many values are there, which will cause the sequence to confuse them with where it expects its own local variables to be addressed within the stack frame).

Bytes	Instruction	Description
40 $\langle np \rangle$	SSFB # $\langle np \rangle$	Ensure stack frame has $\langle np \rangle$ values.

SUB/DECR: Subtract Two Numbers

Subtracts the top value on the stack from the next value (or another value from the top value), and pushes their difference back onto the stack. Note that both operands, and the result, are unsigned 8-bit values (0–255). If the result exceeded this range (including by dropping below zero), the result will be inaccurate (it will be the least significant 8 bits of the value) and the overflow flag will be raised which you can test using conditional branch instructions such as IFANY.

As a special case, the operation of subtracting 1 from a value is named DECR to reflect that it is simply decrementing that value by one.

Bytes	Instruction	Description
14	SUB \$,\$	$y - x \rightarrow x$.
54 $\langle v \rangle$	SUB \$,# $\langle v \rangle$	$x - \langle v \rangle \rightarrow x$.
34	DECR \$	$x - 1 \rightarrow x$.

SUSPEND: Stop Accepting Commands

Stop acting on incoming instructions from the host PC until a RESUME instruction is executed. If the SUSPEND UPDATE version is used, the effects of the commands are still remembered but channel outputs are not actually

affected.

Bytes	Instruction	Description
05	SUSPEND	Suspend command interpretation.
85	SUSPUPD	Suspend tracking updates.

WAIT: Delay

Pauses execution for $\langle time \rangle / 120$ seconds.

Bytes	Instruction	Description
09	WAIT \$	Wait $x/120$ s.
49 $\langle time \rangle$	WAIT # $\langle time \rangle$	Wait $\langle time \rangle / 120$ s.

WAKE: Turn On Load Power

Tell the load power supply it can start up. Requires a properly-configured compatible power supply.

Bytes	Instruction	Description
87	WAKE	Turn on load power supply.

6.11 Loading Binary Sequences Onto Lumos Boards

The binary sequences are stored in the Intel Hex Format, where memory addresses are effectively assumed to be 24 bits long, with the upper eight bits representing the sequence number and the lower 16 bits forming the address within that sequence, allowing for 65,536 bytes per instruction, which is far more than the current Lumos boards can actually accommodate.

Each sequence begins with a line of the form: :0200000400sskk where ss is the two-hex-digit sequence number and kk is the checksum obtained by adding all the bytes together and taking the 8-bit two's compliment of that sum.

Each line after that provides a number of data bytes in the sequence, and is of the form: :nnaaaa00<n data bytes (2n hex digits)...>kk where:

n is the number of data bytes on this line. This is typically 16 or 32 but may be any number from 0–255.

a is the address within the sequence where these data bytes reside.

$\langle data \rangle$ bytes are the instruction data being recorded at this address.

k is the two-hex-digit checksum calculated as described above.

Finally, the last line of the hex file is always: :00000001FF

For example, the sequence pair shown on page 37 would be encoded in the hex file as follows:

Opcode	Args	Mode	Instruction	Opcode	Args	Mode	Instruction
00		inh	exit	81		stk	on \$
01		stk	off \$	83	a	stk	break a,\$
02		stk	dimmer \$,\$	84		stk	ramp \$,\$,\$
03	a	stk	jump a,\$	85		inh	suspupd
04		stk	fade \$,\$,\$	86		inh	resfade
05		inh	suspend	87		inh	wake
06		inh	resume	88		stk	exec \$,\$
07		inh	sleep	8B	a	stk	nextinf a,\$
08		stk	call \$,\$	8D		inh	fadeout
09		stk	wait \$	8F		i/s	push [\$]
0A		stk	loop \$	90		i/s	pop [\$]
0B	a	stk	next a,\$,\$	95	a	stk	nxdclnf a,\$
0D		inh	blackout	A3	a	def	break a
0E		inh	nop	A8	i	d/s	exec \$
0F		inh	dup	AA		def	loop #1
10		inh	drop \$	AB	a	d/s	nextinf a,#1
11		stk	exch	AF		i/d	push [0]
12		stk	mul \$,\$	B0		i/d	pop [0]
13		stk	add \$,\$	B5	a	d/s	nxdclnf a,#1
14		stk	sub \$,\$	C1	c	imm	on #c
15	a	stk	nextdec a,\$,\$	C3	a n	imm	break a,#n
23	a	def	jump a	C4	c s t	imm	ramp #c,#s,#t
28		d/s	call \$	C8	i n	imm	exec #i,#n
2A		def	loop #0	CB	a i	imm	nextinf a,#i
2B	a	d/s	next a,#1,\$	CF	a	ind	push [a]
2F		def	push #0	D0	a	ind	pop [a]
30		def	drop	D5	a i	imm	nxdclnf a,#i
33		def	incr \$	E8	i	i/d	fexec #i
34		def	decr \$	6B	a x	i/d	next a,#1,\$x
35	a	d/s	nextdec a,#1,\$	68	i	d/l	push {0}
40	n	imm	ssfb #n	6F	a x	d/l	pop {0}
41	c	imm	off #c	70			push {\$}
42	c v	imm	dimmer #c,#v	75	a x	i/d	pop {\$}
43	a o	imm	jump a,#0				nextdec a,#1,\$x
44	c s t	imm	fade #c,#s,#t				
48	i n	imm	call #i,#n				
49	t	imm	wait #t				
4A	v	imm	loop #v				
4B	a i x	imm	next a,#i,#x				
4F	v	imm	push #v				
50	n	imm	drop #n				
52	x	imm	mul \$,#x				
53	x	imm	add \$,#x				
54	x	imm	sub \$,#x				
55	a i x	imm	nextdec a,#i,#x				
68	i	i/d	call #i				
6B	a x	i/d	next a,#1,#x				
6F		d/l	push {0}				
70		d/l	pop {0}				
75	a x	i/d	nextdec a,#1,#x				

Table 6.1: Summary of Sequence Instructions by Opcode Value

Opcode	Args	Mode	Instruction	Opcode	Args	Mode	Instruction
0C00	a	stk	ifeq a,\$	8C00	a	stk	ifne a,\$
0C40	a	stk	iflt a,\$	8C40	a	stk	ifge a,\$
0C80	a	stk	ifgt a,\$	8C80	a	stk	ifle a,\$
0CCx	a	inh	ifany a,f	8CCx	a	inh	ifnone a,f
4C00	a x	imm	ifeq a,#x	CC00	a x	imm	ifne a,#x
4C40	a x	imm	iflt a,#x	CC40	a x	imm	ifge a,#x
4C80	a x	imm	ifgt a,#x	CC80	a x	imm	ifle a,#x
4CCx	a	inh	ifall a,f	CCCx	a	inh	ifall a,f

Table 6.2: Summary of Conditional Jump Instructions by Opcode Value

Bits	Flag	Description
0020	Z	Last NEXT*/MUL/ADD/SUB result was zero
0010	V	Last NEXT*/MUL/ADD/SUB result overflowed byte
0008	A	Sensor A input is true (low)
0004	B	Sensor B input is true (low)
0002	C	Sensor C input is true (low)
0001	D	Sensor D input is true (low)

Table 6.3: Conditional Jump Condition Code Bits

```
:020000040001F9
:10000004F0A2A4A144F01EF6F480502AF097503E2
:0A0010000AAF530AB04B0204FF00D0
:020000040005F5
:1000000040024F42AFCF0102CF0153140F810100D4
:00000001FF
```

To assemble the sequence mnemonics into binary format, use the `lumosasm` CLI command (documented fully on page ??):

```
$ lumosasm --output=sequences.hex sequences.lasm
Lumos Sequence Assembler version 1.0
SEQUENCES: 002
ERRORS:    000
```

This binary file may then be downloaded into the Lumos controller using the `lumosctl` command:

```
$ lumosctl --port=COM1 --speed=19200 --address=2 \
--load-hex-sequences=sequences.hex
```

C H A P T E R

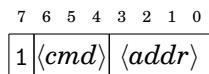


COMMUNICATION PROTOCOL DETAILS

THE PROTOCOL USED BY LUMOS BOARDS to receive commands is designed to be compact (so as to conserve the number of bytes transmitted to carry out common functions) while remaining simple and fast for the controller to interpret. It is also designed so that multiple devices can share the same RS-485 network connection. As long as they all implement this basic protocol, they can safely avoid misinterpreting each other's commands even if they do not know the details of each other's command structure. This includes, for example, Lumos boards at different revision levels. (There are also more devices designed by the author which have very different command sets but use a compatible protocol so they may peacefully coexist with Lumos controllers on the same network.)

The protocol is essentially a stream of 8-bit bytes transmitted over an asynchronous communication link such as RS-232 or RS-485. Commands may consist of as little as a single byte, or could be an arbitrarily large number of bytes long.

The first byte of a command always has this format:



The most significant bit is always set. The next significant three bits, $\langle cmd \rangle$, specify the command being given to the device. The least significant four bits, $\langle addr \rangle$, specify the address of the device which should act on this command.

Any following data bytes in a multi-byte command always have their most significant bit clear:

	7	6	5	4	3	2	1	0
0	$\langle data \rangle$							

This suggests the following algorithm for devices listening to the data stream, upon the receipt of each byte:

1. If bit 7 is set:
 - a) If I was in the middle of collecting data bytes for a command, clearly the host has abandoned it and is beginning a new command, so I should abandon it too and return to normal passive scanning mode.
 - b) If $\langle addr \rangle$ matches my address, interpret command code $\langle cmd \rangle$ and act upon it.
 - c) Otherwise, ignore this byte.
2. Otherwise:
 - a) If I was in the middle of collecting data bytes, collect this one too. Act on the command when the last expected byte is received.
 - b) Otherwise, ignore this byte.

If more than eight commands are needed, we reserve command 7 as an extended command, where the bits in the following byte are used:

	7	6	5	4	3	2	1	0
1	1	1	1	$\langle addr \rangle$				
0	$\langle cmd \rangle$							

Note that where a value for $\langle channel \rangle$ is given in the commands that follow, the protocol's bitfield may allow for a greater number of channels than the Lumos board actually has. For example, typically six bits are allocated for channel numbers, giving a range of values of 0–63, but Lumos boards are usually built to have 24 or 48 channels. If a command specifies a $\langle channel \rangle$ value the board does not support, it will flag the command as an error and ignore it.

If a value greater than 127 needs to be sent, there is an escape mechanism recognized by the Lumos controller. The following two byte values are special:

0x7E The following byte will have its MSB set upon receipt.

0x7F The following byte will be accepted as-is without further interpretation. This means a literal 0x7E byte needs to be sent as 0x7F 0x7E, and a literal 0x7F byte as 0x7F 0x7F.

In the command descriptions that follow, we will show the bytes sent or received graphically, using the following notation:

- Binary bits are shown in fixed-width type: 0, 1.
- Decimal numbers are shown in normal type: 123.
- Hexadecimal numbers are shown in fixed-width type with a leading 0x: 0x42.
- Single-bit flags are shown in Italics as single-letter names: *f*.
- Multi-bit fields are shown in Italics with angle brackets: $\langle speed \rangle$.

7.1 0x0–0x6: Common Commands

With the command code embedded in the packet's initial byte, these commands represent the most commonly used features of the Lumos boards, thereby using the minimum number of bytes to transmit over the wire, possibly even a single byte for the entire command.

0x0: Blackout

7	6	5	4	3	2	1	0	
1		0						$\langle addr \rangle$

Immediately turns all output channels completely off.

0x1: Channel On/Off

7	6	5	4	3	2	1	0	
1		1						$\langle addr \rangle$
0	<i>s</i>							$\langle channel \rangle$

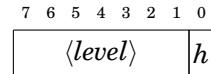
Turns output $\langle channel \rangle$ fully on (*s*=1) or off (*s*=0).

0x2: Set Channel Output Level

7	6	5	4	3	2	1	0	
1		2						$\langle addr \rangle$
0	<i>h</i>							$\langle channel \rangle$
0								$\langle level \rangle$

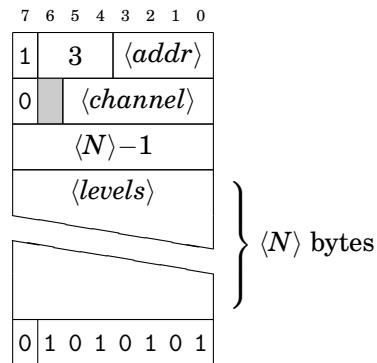
Sets the output level of the designated $\langle channel \rangle$ to a level in the range 0–255. The controller will simply switch the output to a steady off or on state if the level is 0 or 255, respectively. Otherwise, it will use pulse width modification to send a square wave pulse that will be on a fraction of the time approximately equivalent to the value on a 0–255 scale. See Chapter 9 for more information about how this pulsing works and how it affects the apparent brightness of lights controlled by that channel.

Note that the value of the output level is encoded as the combination of the $\langle level \rangle$ field and the h bit, with h forming the *least significant* bit of the resulting value:



This means is that for the sake of simplicity, you may choose to ignore the h bit,¹ and just use a 0–127 value for $\langle level \rangle$, and it will work (although the steps between each level will be twice as large).

0x3: Bulk Channel Update



If more than a small number of channel levels are being updated at once, it is more economical to send a single bulk update command with all their new values in one payload, rather than initiate individual commands for them.

This updates $\langle N \rangle$ channels starting with $\langle channel \rangle$ and continuing to $\langle channel \rangle + \langle N \rangle - 1$. Note that the value $\langle N \rangle - 1$ is passed, not $\langle N \rangle$. This allows for any number of channels from 1–256 to be updated (in theory; no Lumos controller currently implements that many channels).

Note that it may be necessary to escape some of these data bytes so their 8-bit values conform to the overall communications protocol as described at the beginning of this chapter (see p. 54 for details about this escape mechanism).

0x3: Bulk Channel Update—deprecated version

The following is obsolete:

¹For best results, if you choose to do this, set $h=0$ unless setting the channel at maximum brightness, in which case set $h=1$.

7 6 5 4 3 2 1 0							
1	3	$\langle addr \rangle$					
0	m	$\langle channel \rangle$					
0		$\langle N \rangle - 1$					
0		$\langle level_0 \rangle$					
:							
0		$\langle level_{N-1} \rangle$					
0	1 0 1 0 1 1 0						
0	$h_0 h_1 h_2 h_3 h_4 h_5 h_6$						
:							
0		$\dots h_{N-1}$					
0	1 0 1 0 1 0 1						

} if $m=1$

If more than a small number of channel levels are being updated at once, it is more economical to send a single bulk update command with all their new values in one payload, rather than initiate individual commands for them.

This updates $\langle N \rangle$ channels starting with $\langle channel \rangle$ and continuing to $\langle channel \rangle + \langle N \rangle - 1$. Note that the value $\langle N \rangle - 1$ is passed, not $\langle N \rangle$. This allows for any number of channels from 1–128 to be updated.

As with the “Set Output Channel Level” command above, the 8-bit level value for the i th channel in the list is the combination of the $\langle level_i \rangle$ field and the h_i bit:

7 6 5 4 3 2 1 0							
$\langle level_i \rangle$				h_i			

This command has two modes of operation as selected by the m bit. In low-resolution mode ($m=0$), all the h bits are omitted, and the level range is effectively 0–127, with 0 being fully off and 127 being fully on.² In high-resolution mode ($m=1$), then all bits are specified and the full range of values (0–255) is supported.

0x4: Ramp Channel Level

7 6 5 4 3 2 1 0											
1	4	$\langle addr \rangle$									
0	d	$\langle channel \rangle$									
$\langle steps \rangle - 1$											
$\langle time \rangle - 1$											

²The Lumos device intelligently handles these edge conditions rather than simply assuming the least significant bit is always set or clear.

Smoothly increases ($d=1$) or decreases ($d=0$) the output level of the specified $\langle channel \rangle$ by the specified number of $\langle steps \rangle$, with a delay of $\langle time \rangle / 120$ seconds between each change. Since the values passed are actually $\langle steps \rangle - 1$ and $\langle time \rangle - 1$, the effective ranges for each of those values is 1–256.

Note that it may be necessary to escape some of these data bytes so their 8-bit values conform to the overall communications protocol as described at the beginning of this chapter (see p. 54 for details about this escape mechanism).

Unimplemented Single-byte Codes

7	6	5	4	3	2	1	0
1		5		$\langle addr \rangle$			

7	6	5	4	3	2	1	0
1		6		$\langle addr \rangle$			

Codes 5 and 6 are not implemented and are reserved for future Lumos features.

7.2 0x700–0x71f: Extended Commands

Less frequently used commands use an extended code, where the initial byte's command code is 7 (all bits set), and the following byte gives the remaining bits of the command. The first 32 such codes (extended code values 0x00–0x1f) are set aside for normal-mode commands. Additionally, response packets sent back to the host PC from the Lumos boards are formatted the same as commands, with codes assigned starting at 0x71f, counting down, while the commands start at 0x700, counting up. There is a currently-unimplemented zone of codes in the middle which is reserved for future use.

0x700: Sleep

7	6	5	4	3	2	1	0
1		7		$\langle addr \rangle$			
0				0			
0	1	0	1	1	0	1	0
0	1	0	1	1	0	1	0

Put the controller into sleep mode. This signals the controlled load's power supply to shut down (whether the power supply is equipped or inclined to obey that signal is another matter). The Lumos board may automatically—even immediately—wake out of sleep mode if it is asked to supply an output >0 on any of its channels.

0x701: Wake

7 6 5 4 3 2 1 0							
1	7	<i>(addr)</i>					
0		1					
0	1	0	1	1	0	1	0
0	1	0	1	1	0	1	0

Wake up the controller out of sleep mode. This signals the controlled load's power supply to turn on (whether the power supply is equipped or inclined to obey that signal is another matter). The Lumos board may automatically go into sleep mode again if some period of time elapses during which it had no output channels with levels >0.

0x702: Shut Down

7 6 5 4 3 2 1 0							
1	7	<i>(addr)</i>					
0		2					
0	1	0	1	1	0	0	0
0	1	0	1	1	0	0	1

Shuts the controller completely down. After this command executes, the Lumos board will shut down as many of its functions as possible, reducing its power consumption to the bare minimum. It will no longer respond to any commands sent. The only way out of a shut down is to reset or power cycle the board.

0x703: Query

7 6 5 4 3 2 1 0							
1	7	<i>(addr)</i>					
0		3					
0	0	1	0	0	1	0	0
0	1	0	1	0	1	0	0

Reports the device status back to the host PC. On half-duplex networks, the host PC needs to switch to listening mode immediately after sending this command, although on full duplex networks, this is not necessary. The response packet is:

0x71f [reply] Query Response from Lumos Controller

7 6 5 4 3 2 1 0								
1	7	$\langle \text{addr} \rangle$						
0	31							
0	$\langle R_1 \rangle$		$\langle R_0 \rangle$					
0	$\langle S_c \rangle$		d	$\langle C_1 \rangle$				
0	$\langle C_0 \rangle$							
0	$\langle S_m \rangle$		q	s	f			
0	$\langle S_a \rangle$		X	$\langle P_1 \rangle$				
0	$\langle P_0 \rangle$							
0	$\langle E_1 \rangle$							
0	$\langle E_0 \rangle$							
0	$\langle M_1 \rangle$							
0	$\langle M_0 \rangle$							
0	Q	0	$\langle \text{model} \rangle$					
0	$\langle \text{exec} \rangle$							
0	o	w	e	$\langle \text{sens} \rangle$				
0	$\langle \text{pre} \rangle$							
0	$\langle \text{run} \rangle$							
0	$\langle \text{post} \rangle$							
0	:							
0	$\langle \text{fault}_1 \rangle$							
0	$\langle \text{fault}_0 \rangle$							
0					$\langle P'_1 \rangle$			
0					$\langle P'_0 \rangle$			
0	$\langle \text{serial}_1 \rangle$							
0	$\langle \text{serial}_0 \rangle$							
0	0	1	1	0	0	1	1	

} each sensor A–D

The information contained in this packet describes several facts about the Lumos unit:

Device identity: The reporting device's current address $\langle \text{addr} \rangle$ on the network; the ROM version (major revision $\langle R_1 \rangle$, minor $\langle R_0 \rangle$); the device's $\langle \text{serial} \rangle$ number, or 0 if no number was assigned (i.e., a ROM im-

age downloaded and loaded by a hobbyist working on their own); The device model code is in the $\langle model \rangle$ field. The following values are currently defined for $\langle model \rangle$: 0=48-channel controller, 1=24-channel DC controller.

General Status: The q flag indicates if the device is in configuration mode ($q=1$) or normal run mode ($q=0$); the s flag shows whether it is in sleep mode ($s=1$); The X flag is true ($X=1$) if configuration mode is locked out. If a sequence is currently running, the Q flag will be set and the sequence number is in field $\langle exec \rangle$. The fault code from the last failed command is contained in field $\langle fault \rangle$, which is a value in the range 0–16,383:

13	7 6	0
$\langle fault_1 \rangle$	$\langle fault_0 \rangle$	

A value of 0 means there was no fault detected since the last query command (i.e., the fault condition is cleared by the query command).

Sensor information: Three values describe the sensor configuration and state: $\langle S_c \rangle$ indicates the sensor lines configured as inputs (1) or as LED outputs (0); $\langle S_m \rangle$ shows which sensor inputs are masked out (1) or are being monitored (0); and $\langle S_a \rangle$ indicates which sensors are currently reading a logic 1 or 0. In each case, the sensor lines are represented by these bits in each field:

6 5 4 3	
A B C D	

Each sensor's trigger information is given by a block of four bytes. The first contains the field $\langle sens \rangle$ which identifies the sensor (0=A, 1=B, 2=C, 3=D). Flag o is set if the trigger is set to fire once only each time the sensor activates, w if it's set to run while the sensor is active, and e indicates if the sensor is active high ($e=1$) or active low ($e=0$). The following three bytes, $\langle pre \rangle$, $\langle run \rangle$, and $\langle post \rangle$, specify the sequence number to be executed initially, repeatedly while active, and at the end, respectively.

DMX512 configuration: The bit d indicates that the board is in normal Lu-mos mode ($d=0$) or DMX512 mode ($d=1$). If in DMX512 mode, its starting DMX channel is $\langle C \rangle+1$, which is a value in the range 1–512, sent in two fields:

8 7 6	0
$\langle C_1 \rangle$	$\langle C_0 \rangle$

Memory state: The f flag is true ($f=1$) if the sequence storage memory overflowed when being sent a new program; The number of bytes of EEPROM memory free for storage of permanent sequences is given by the $\langle E \rangle$ field, while the bytes of available RAM memory for temporary storage is in the $\langle M \rangle$ field. Each of these values is in the range 0–16,383, sent as two fields:

13	7 6	0
$\langle E_1 \rangle$	$\langle E_0 \rangle$	
$\langle M_1 \rangle$	$\langle M_0 \rangle$	

Operating parameters: The internal timing mechanism's phase offset value $\langle P \rangle$ is a number in the range 0–512, given by the combination of two fields. For 48-channel units, the phase offset of the secondary microcontroller is given in $\langle P' \rangle$.

8 7 6	0
$\langle P_1 \rangle$	$\langle P_0 \rangle$
$\langle P'_1 \rangle$	$\langle P'_0 \rangle$

0x704: Define Sequence

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0		4					
0		$\langle id \rangle$					
		$\langle N \rangle - 1$					
		$\langle data \rangle$					
0	1 0 0 0 1 0 0						
0	1 1 1 0 0 1 1						

$\} \langle N \rangle$ bytes

Downloads a new sequence $\langle data \rangle$ of $\langle N \rangle$ bytes in length into the Lumos controller. This sequence will be known with the given $\langle id \rangle$. If a sequence is stored as $\langle id \rangle=0$, it will be executed automatically whenever the Lumos board is reset. It cannot explicitly be invoked by a command from the host PC.

Sequences with $\langle id \rangle$ in the range 0–63 are stored in permanent EEPROM memory and will remain in the controller even after a reset or power cycle. Those with $\langle id \rangle$ in the range 64–127 are stored temporarily in RAM memory and will be lost when the device resets.

0x705: Execute Stored Sequence

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	5						
0	$\langle id \rangle$						

Starts executing the stored sequence with the given $\langle id \rangle$. If another sequence was in progress, it is stopped. If $\langle id \rangle=0$, the current sequence is stopped without starting a new one.

0x706: Define Sensor Action

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	6						
0	o	w	e	$\langle id \rangle$			
0	$\langle pre \rangle$						
0	$\langle exec \rangle$						
0	$\langle post \rangle$						
0	0	1	1	1	1	0	0

Defines the action to be taken when the sensor $\langle id \rangle$ triggers. ($\langle id \rangle=0$ is sensor A, $\langle id \rangle=1$ is B, $\langle id \rangle=2$ is C, and $\langle id \rangle=3$ is D.) The sensor is triggered on the rising edge if $e=1$, or the falling edge if $e=0$.

When that sensor is triggered, any currently executing sequence is stopped. Then the sequence number $\langle pre \rangle$ is executed once. The $\langle exec \rangle$ sequence will be executed one time if $o=1$, or repeatedly while the sensor continues to be active if $w=1$; if neither of those bits is set, the sequence loops forever until explicitly stopped.³ When the sensor is no longer triggering, $\langle exec \rangle$ plays on until it completes, then $\langle post \rangle$ is run once.

Note that if another sensor triggers or an “Execute Stored Sequence” command is run, it immediately stops the current sequence and all associated sequences. This may cause $\langle post \rangle$ to not execute.

0x707 Mask Sensors

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	7						
0	A B C D						

³The action if both bits are set is undefined.

Sets input sensor masks for the given sensors. If the mask value is 1, that sensor is ignored. If the mask is 0, it is responded to normally.

0x708 Erase All Stored Sequences

	7	6	5	4	3	2	1	0						
1	7	<addr>												
0	8													
0	1 0 0 0 0 1 1													
0	1 0 0 0 0 0 1													

All stored sequences are erased.

Reserved Command Codes

Codes 0x709–0x71d are reserved for future commands. 0x71e–0x71f are currently used for response packets from the Lumos controller.

0x71e [reply] Query NAK

	7	6	5	4	3	2	1	0
1	7	<addr>						
0	30							

When the host PC sends a “query” command to the Lumos controller, the ultimate response will be the reply packet documented on page 60. However, if there will be a delay before the Lumos controller is ready to provide that response, it may send this “NAK” packet to indicate that it’s not yet ready to reply. This keeps the host PC from timing out and abandoning the controller’s reply.

7.3 0x720–0x73f: Reserved

These codes are reserved for internal use by the Lumos system and may not be used externally.

7.4 0x740–0x77f: Configuration-Mode Commands

These commands make changes to the state of the device in some manner which requires the device to be in configuration mode. If these are encountered outside configuration mode, they will be rejected.

0x74x [config] Set Phase Offset

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	4	$\langle P_1 \rangle$					
0	$\langle P_0 \rangle$						
0	1 0 1 0 0 0 0						
0	1 0 0 1 1 1 1						

Sets the controller's phase offset to $\langle P \rangle$, which is a 9-bit value in the range 0–511:

8 7 6 0							
$\langle P_1 \rangle$				$\langle P_0 \rangle$			

This affects the internal timing mechanism within the Lumos controller used to synchronize dimmer pulses to the AC waveform (for AC controllers). It shouldn't need to be changed. **This command is only recognized if the controller is in configuration mode.**

0x76x [config] Set Device Address

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	6	$\langle addr' \rangle$					
0	1 0 0 1 0 0 1						
0	1 0 0 0 0 0 1						
0	1 0 0 0 1 0 0						

Changes the address of unit from $\langle addr \rangle$ to $\langle addr' \rangle$. This is effective immediately, so the very next command must be addressed to $\langle addr' \rangle$ for this unit to respond to it. **This command is only recognized if the controller is in configuration mode.**

0x770 [config] Cancel Configuration Mode

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0	112						

Cancel configuration mode and return to normal operating mode. From this point forward, commands marked “[config]” will no longer be recognized. **This command is only recognized if the controller is in configuration mode.**

0x771 [config] Configure Device

	7	6	5	4	3	2	1	0
1	7							$\langle \text{addr} \rangle$
0								113
0	A	B	C	D	d			$\langle C_1 \rangle$
0								$\langle C_0 \rangle$
0	0	1	1	1	0	1	0	
0	0	1	1	1	1	0	1	

Sets general configuration parameters not already covered elsewhere in this command set. Flags A–D control whether a given sensor input line is an input (1) or output (0). If configured as inputs, the Lumos controller does not drive a voltage on them but watches for a signal there to which the Lumos board will respond if programmed to do so. If configured as outputs, the Lumos controller assumes they are connected to diagnostic LEDs and will drive them with +5 V to turn on the corresponding LEDs and 0 V to turn them off.

If flag d is set ($d=1$), the Lumos controller will operate in DMX512 mode (however, note that anytime it is in configuration mode, DMX512 command reception is disabled and the Lumos commands described here are recognized. When $d=1$, the first DMX512 channel claimed by this controller is channel $\langle C \rangle+1$, which is a 9-bit value in the range 1–512:

8	7	6	0
$\langle C_1 \rangle$			$\langle C_0 \rangle$

This corresponds to Lumos controller channel 0. **This command is only recognized if the controller is in configuration mode.**

0x772 [config] Set Baud Rate

	7	6	5	4	3	2	1	0
1	7							$\langle \text{addr} \rangle$
0								114
0								$\langle \text{speed} \rangle$
0	0	0	1	0	0	1	1	0

Sets the Lumos controller's baud rate as indicated by $\langle \text{speed} \rangle$, according to the following table:

$\langle speed \rangle$	Bits per Second
0	300
1	600
2	1,200
3	2,400
4	4,800
5	9,600
6	19,200
7	38,400
8	57,600
9	115,200
10	250,000

This setting does not affect DMX512 mode, which always uses a fixed speed of 250,000 bps. **This command is only recognized if the controller is in configuration mode.**

Note that for 48-channel controllers, this also sets the intra-processor communication speed (which the two microcontrollers use to coordinate their actions). Setting this to a low speed is not recommended.

0x773 [config] Restore Factory Defaults

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0		115					
0	0	0 1 0 0 1 0 0					
0	1	1 1 1 0 0 1 0					

Restores the device to its original factory settings. Note that, among other things, this will change the device's address to 0 and its baud rate to 19,200 bps. **This command is only recognized if the controller is in configuration mode.**

0x774 [config] Forbid Configuration Mode

7 6 5 4 3 2 1 0							
1	7	$\langle addr \rangle$					
0		116					

Cancels configuration mode, returning to normal operating mode. Additionally, this command prevents the device from re-entering configuration mode from this point forward until it is reset or power-cycled. **This command is only recognized if the controller is in configuration mode.**

Reserved Command Codes

Codes 0x775–0x77f are reserved for future use as configuration-mode commands.



CHAPTER
8

DMX512 COMMAND STRUCTURE

When configured in DMX512 mode, all of the Lumos commands documented in Chapter 7 are not recognized. The Lumos controller will be only looking for DMX512 command packets. To use the Lumos commands for configuration of the board (e.g., to turn off DMX512 mode or change the starting channel number), activate configuration mode using the Option button. That will disable DMX512 mode for the time the device is in configuration mode.

The DMX protocol always uses a fixed speed of 250,000 baud. There is only one packet type recognized by a Lumos controller. All other packets are silently ignored.

A packet begins with a “break” condition on the line. Immediately following the break is a sequence of one or more bytes of data. The first byte must be a zero (other values for this initial byte are used to send other kinds of DMX512 packets, but we’re not interested in those so they are simply ignored).

After the zero byte there will be up to 512 channel value bytes. (It is permissible for the packet to end before all 512 channel values have been sent.) The Lumos controller is configured to have a starting address within the DMX512 “universe.” This corresponds to Channel 0 of the Lumos controller. For example, if a Lumos controller were configured to DMX512 channel 10, then it would ignore the first nine bytes of channel values in each packet. The tenth byte would set the output value of the controller’s Channel 0, the next byte would set the output level for Channel 1, and so on, up to the thirty-fourth byte which sets Channel 23 (for a 24-channel controller), or the fifty-eighth byte which sets Channel 47 (for a 48-channel controller).

If the packet ends before the Lumos controller has received enough bytes to set all of its channels, the remaining channels simply keep their previously-

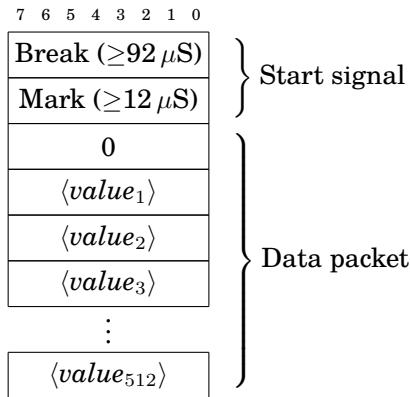


Figure 8.1: DMX Packet

known output values.

The bytes in these packets are 8-bit values, directly giving the channel output values of 0–255, with 0 being fully off and 255 being fully on.

A packet has the general form shown in Figure 8.1. Note that DMX512 channel numbers start with 1.

C H A P T E R



THEORY OF OPERATION

THE LUMOS CONTROLLER BOARDS provide 256 levels of dimmer control on their output channels by using pulse width modification (PWM). That is, each output is always either fully “on” (0 V output) or “off” (+5 V output) at any instant in time, but cycles between on and off states so that, for example, if a channel is set to 50%, it will be fully on half the time and fully off half the time. This is illustrated in Figure 9.1.

In more detail, each half-wave period (1/120 s for the 60 Hz power frequency standard used in many countries such as the USA)¹ is divided into 260 “slices” of approximately 0.000032051 s each. (See Figure 9.2.) An output channel may change state at one of those slice boundaries. This provides for 256 different output levels to be supported, plus a couple of idle slices at the beginning and end to make sure the TRIACs are fully off before the next zero-crossing point begins. (You’ll note in the timing diagrams that the outputs are always turned off a very tiny fraction of a second before the start of the next output cycle.)

For the main intended application of these controllers—incandescent lamps for the AC boards and LEDs for the DC boards—this produces a visual effect of the light being dimmed. Note that not all devices can tolerate being supplied power like this, so you need to make an informed decision about what to plug into a Lumos controller.²

¹The AC-powered Lumos boards are not designed to work in environments where the power frequency is not 60 Hz. DC-powered boards will work regardless, since they never see the AC power directly.

²You might decide it’s an acceptable risk to attach one of these loads, such as a “non-dimmable” CFL light, if you only tell the Lumos board to turn it on or off, and never use the dimmer settings. In theory, that may be ok, all other things being equal, but you’re trusting

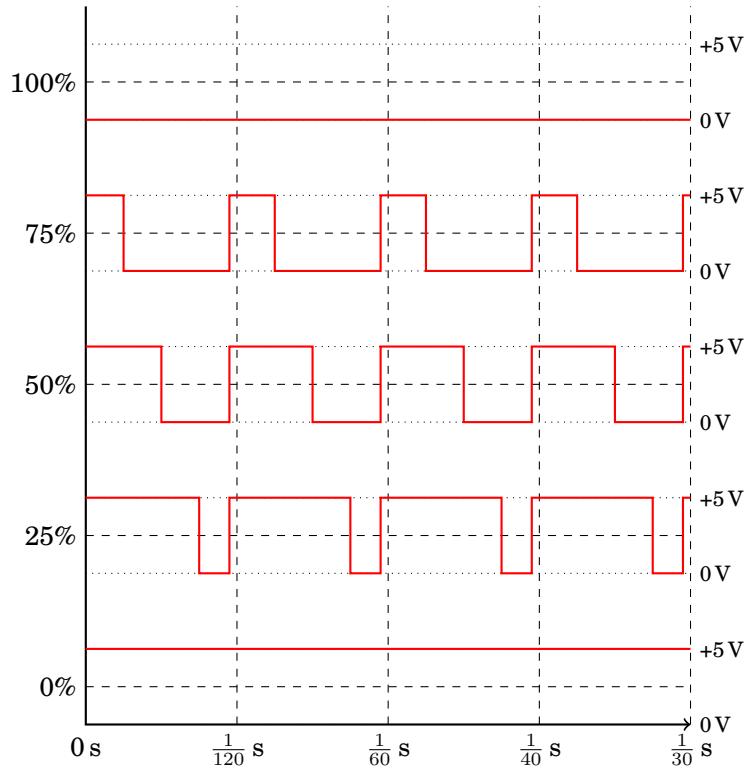


Figure 9.1: Duty Cycles of Channel Logic Drive Outputs

For DC loads, this PWM signal appears as-is on the output channel (although inverted). Note that the frequency of these pulses is $1/120$ s, which should be sufficient to avoid visible “flicker” for both incandescent and LED lights. This is shown in Figure 9.3.

Dimming AC loads works along similar lines, but the TRIAC outputs on those controllers require that these timing pulses be synchronized to the points where the AC power waveform crosses the 0 V line. The reason for this is that TRIACs, once turned on, stay on as long as power is applied to them, even if the controlling gate signal turns off. So the only opportunity to delay them from switching on is at a zero-crossing point. The 48-channel controller board uses its built-in power supply to sense the AC zero-crossing point and makes that available to the on-board logic, which will base its channel outputs on that timing signal.

that your host PC software won’t accidentally command the board to do something to fade that output, or a stored sequence won’t do that, or a glitch in communications won’t be misinterpreted as such, or that even a firmware bug on the controller won’t cause this. It’s an informed risk you can decide to take, but we don’t officially recommend it.

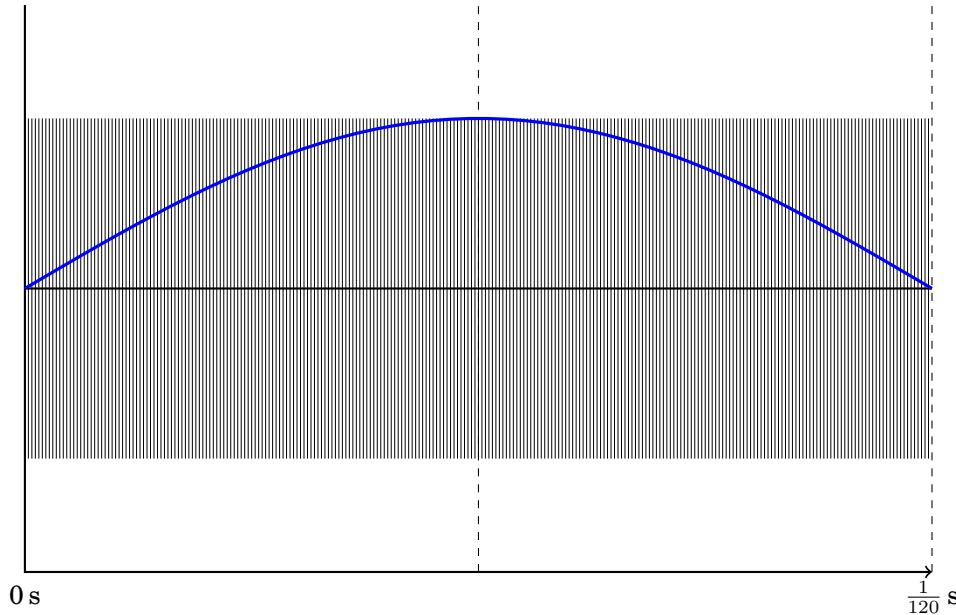


Figure 9.2: Half-AC Cycle Divided into 260 Slices

This means that the AC waveform which appears at the output of the relay board starts part-way into each half-cycle, as shown in Figure 9.4.

The DC controllers produce a 120 Hz timing pulse internally so they are consistent with the AC controllers, but this doesn't need to be synchronized with anything external.

There is an obscure device setting called “phase offset” on the Lumos controllers which controls the delay between the receipt of the zero-crossing signal and the time of the actual zero-crossing event. This compensates for the effect of the controller’s AC supply being out of phase with the load AC supply. (In the very first prototype Lumos design, the detector circuit needed this but that is no longer the case. In practice, there shouldn’t be any phase difference between the controller’s supply and the load supply which would require changing this offset. Note that being 180° out of phase—such as being supplied from separate “sides” of a residential AC breaker panel—doesn’t matter here, since they have the same zero-crossing point.)

We will now describe how the phase offset works in the internal timing chain implemented by the Lumos controller firmware, in case you should find yourself in some strange circumstance where you need to change this setting. Otherwise, the normal setting for this value is 2.

The timing diagram for the controller’s output update cycle is shown in Figure 9.5. Note the green AC waveform as perceived by the zero-crossing

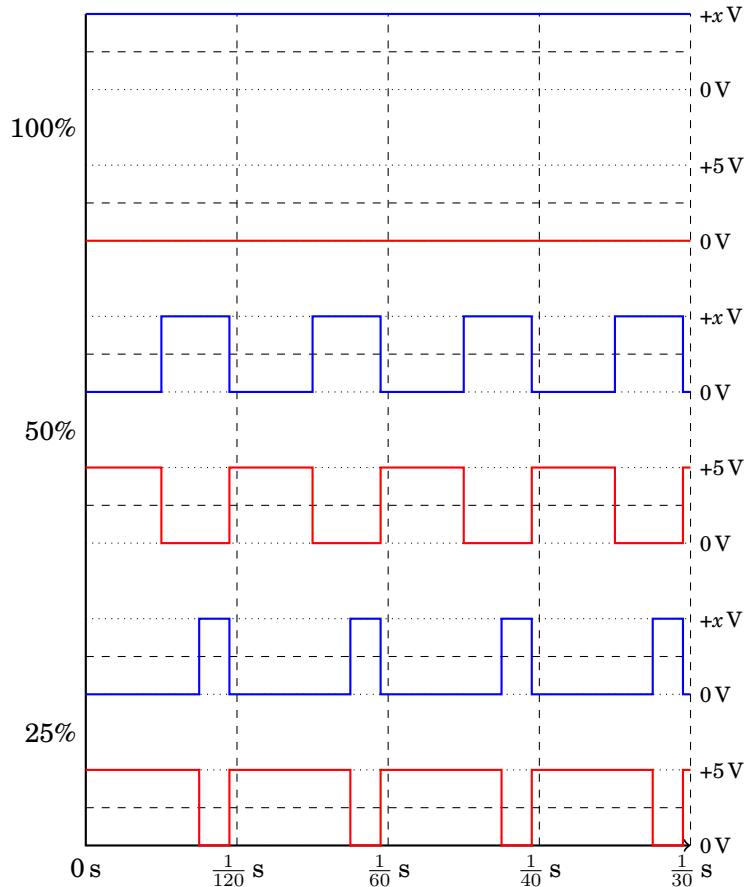


Figure 9.3: Duty Cycles of Logic (red) and DC SSR Outputs (blue)

detector vs. the blue actual waveform present at the loads. The phase offset is compensating for this difference.

When the zero-crossing detector senses 0 V on the incoming power line, it triggers an interrupt on the Lumos microcontroller (INT on the diagram). This starts the phase delay timer which counts down a number of slices equal to the “phase offset” setting (normally 2). Once that many slices have gone by, the “working slices” begin. During each of these slices, various output channels are turned on in order for each of them to be effectively generating the desired PWM duty cycle.

Assuming that the interrupt arrives at the actual zero-crossing event, as it ideally does, the default phase offset delay of 2 means that we get two idle slices before the 256 working slices begin, and two idle slices at the end. This compensates for any slight timing errors that may creep into the cycle

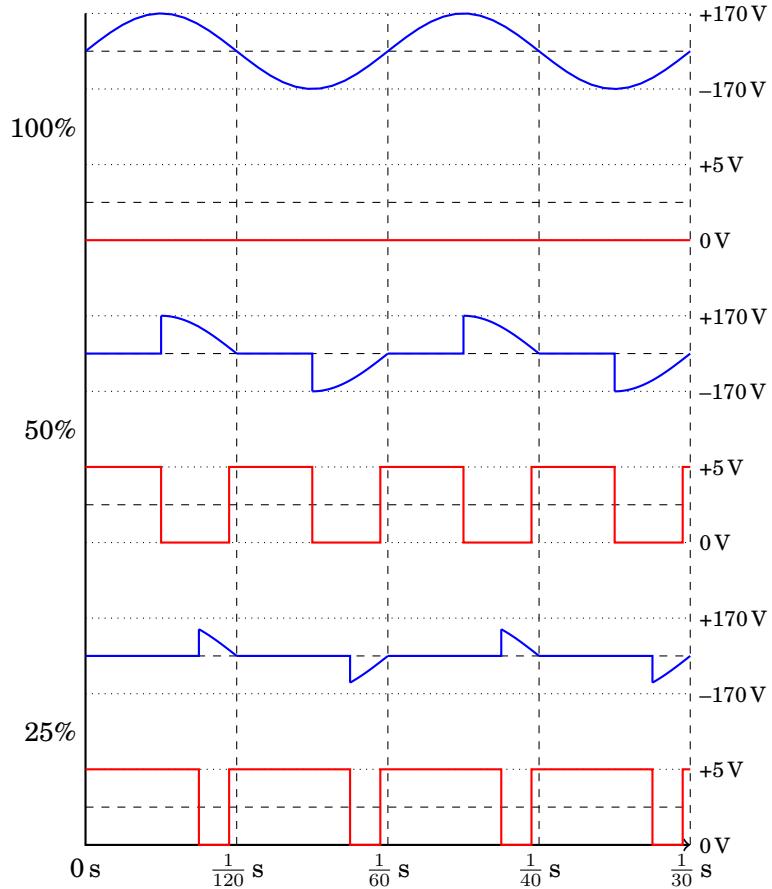


Figure 9.4: Duty Cycles of Logic (red) and AC SSR Outputs (blue)

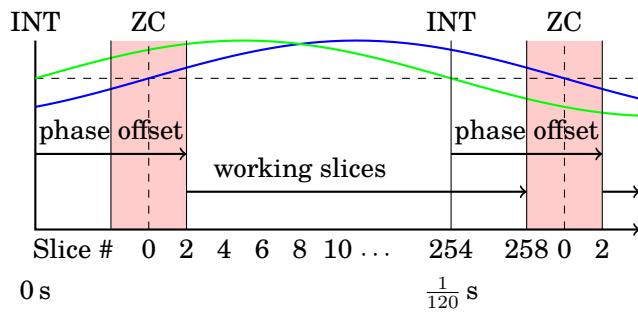


Figure 9.5: Cycle Timing with Phase Offset

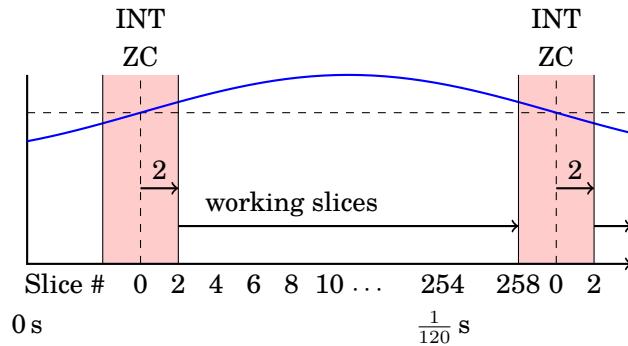


Figure 9.6: Cycle Timing with No Phase Difference

as well as ensuring the TRIACs settle as already described. This is shown in Figure 9.6.

9.1 Output Relay Circuits

The output solid-state relays (SSRs) used by the Lumos boards are variations of the standard SSR design used within the DIY animated lighting community over the years (see p. 102). For DC boards, this is a high-power MOSFET circuit, while for AC boards, this is a TRIAC. In both cases, the relay is intended to control a simple resistive load (typically incandescent and LED lights). They are not designed to control inductive loads (e.g., motors or fluorescent lights). If you intend to use those types of loads with these SSRs, you will need to add protective circuitry (such as an additional, higher-power SSR or a “snubber” circuit) to the Lumos output.

DIAGNOSTIC CODES

Decoding LED Patterns

The front panel LEDs provide an indication of the state of the Lumos controller. During boot, they rapidly change to indicate the phase of the initialization process being performed. If the device gets stuck during that process, the LED pattern will indicate where the problem occurred. During normal runtime operation, they inform the user of the mode and status of the system, errors encountered, etc.

The various codes are summarized in Figure 1.

Numeric Error Codes

48-ch 24-ch Description of Condition/Fault Indicated

	[boot] Boot process not yet started
	[boot] EEPROM setup stage
	[boot] EEPROM write operation
	[boot] EEPROM read operation / system initialization
	[boot] system initialization
	[boot] system initialized but main loop/timing system non-functional
	[run] factory defaults restored (will now reboot)
	[run] normal run mode
	[run] received command addressed to this unit
	[config] configuration mode
	[run] intra-processor communication activity
	[run] command rejected (invalid, bad arguments, disallowed, etc.)
	[run] communications error (framing error)
	[run] communications error (overrun error)
	[run] communications error (device buffer overflow)
	[run] internal fault detected
	[sleep] in sleep mode
	[halt] system halted (shutdown) normally
	[halt] system failure while trying to halt
	[halt] system failure (exact error on other LEDs)

Figure 1: Diagnostic LED Patterns

48-ch 24-ch Description of Condition/Fault Indicated

		Dispatch table overrun
		Input validator failure
		Reset failure
		Hardware fault
		Internal command error
		Other/unknown failure

Figure 2: Internal Fault Condition Codes

- steady on
- steady off
- slowly fading up/down
- quickly fading up/down
- slowly flashing
- quickly flashing
- blink then fade once
- super-slow fading up/down
- ✗ not involved/affected; may have any value

Figure 3: Key to LED Patterns

0x01	Command interpreter dispatch overrun
0x02	Pass-down command in non-master ROM (set level)
0x03	Pass-down command in non-master ROM (bulk update)
0x05	Command interpreter dispatch overrun (in state 6)
0x06	Pass-down command in non-master ROM (ramp level)
0x07	Command interpreter dispatch overrun (in state 9)
0x08	Command interpreter dispatch overrun (in state 10, non-slave)
0x0A	Bad sentinel byte in internal command
0x0B	Command interpreter dispatch overrun (internal commands)
0x0C	Command interpreter dispatch overrun (internal commands)
0x0D	Command interpreter dispatch overrun (state 13)
0x0E	Command interpreter dispatch overrun (state 17)
0x0F	Could not determine ROM type (query)
0x10	Operation on wrong ROM type (query)
0x11	Device does not support T/R operation
0x12	Code executed on wrong ROM (M/S communication)
0x20	Invalid command
0x21	Configuration-mode command outside configuration mode
0x22	Command not implemented
0x23	Command incomplete
0x70	Failed to reset following factory default restore

Figure 4: Error Condition Codes Reported Via Query Command

LUMOS CLI COMMAND MANUAL ENTRIES

This chapter provides the documentation for the `lumosasm` and `lumosctl` commands. This same information is also provided to the CLI user on Unix, Linux, or Macintosh systems via the `man` command.

In this documentation, the following typographical conventions are used:

- *<Variables>*, which indicate values to be replaced with suitable values when you invoke the program, are shown in Italic type inside angle brackets. (The angle brackets are not typed as part of the command syntax.)
- Literal text which should be typed as-is, as well as the names of commands, is set in fixed-width text.
- *File names* are set in Italics. Italics are also used for general points of emphasis.
- [Optional values] are enclosed in square brackets. These may be omitted if appropriate. (The brackets themselves are not typed as part of the command syntax.)

References to other program manual entries look like “`lumosctl(1)`” which indicates that the `lumosctl` command is documented in section 1 of the manual, which is the section for general user commands on Unix-like systems.

NAME

`lumosasm` – Lumos SSR controller sequence assembler

SYNOPSIS

`lumosasm [-hHv] [-l <file>] [-o <file>] <asm-files>...`

DESCRIPTION

Lumos stored sequences may be written in a high-level programming language (but see the note under “BUGS” below) and loaded into the controller board via `lumosctl(1)`. Alternatively, they may be written in the low-level assembly language which corresponds directly to the bytecode interpreted by the controller itself. The `lumosasm` command is used for the latter use case. It converts the sequences described in one or more `<asm-files>` to binary bytecode which is written to a single output file.

OPTIONS

Each of the following options may be specified by either a long option (like “`--verbose`”) or a shorter option letter (like “`-v`”). If an option takes a parameter, it may follow the option as “`-lfoo.txt`”, “`-l foo.txt`”, “`--listing foo.txt`”, or “`--listing=foo.txt`”.

Long option names may be abbreviated to any unambiguous initial substring.

`--help`

(`-h`) Prints a summary of these options and exits.

`--list-opcodes`

(`-H`) Prints out a list of all opcodes and the instruction signature they represent, then exits.

`--listing=<file>`

(`-l <file>`) Causes a listing to be written to the specified `<file>`. This will include the assembly code input and byte code output. By default, no listing is made.

`--output=<file>`

(`-o <file>`) Specifies the name of the output file into which the assembled binary bytecodes will be written. By default, this is the name of the first input file with its suffix replaced by “`.hex`”.

`--verbose`

(`-v`) Each `--verbose` option increases the level of information generated by the `lumosasm` command.

--version
 Prints the version of this program and exits.

AUTHOR

Software Alchemy / support@alchemy.com

COMPATIBILITY

This version of `lumosasm` is compatible with the following boards:

- Lumos 48-channel controller version 3.1 or 3.2 *providing it has been upgraded or installed with ROM firmware version 3.0 or later* (boards with ID markings beginning with “48CTL-3-1” or “LUMOS-48CTL-3.2”). (Whether this controller is driving AC or DC boards is irrelevant.)
- Lumos 24-channel DC controller version 1.0 (boards with ID markings beginning with “LUMOS-24SSR-DC-1.0”).

HISTORY

This program is new as of Lumos ROM version 3.0.

SEE ALSO

`lumosctl(1)`.

Using the Lumos SSR Controllers, which explains in detail the programming model and assembly language instructions used with this assembler.

BUGS

Currently, the high-level language compiler has yet to be written, so this assembly code is the only option which is available for writing sequences. The high-level compiler is a future task being planned.

Submit any other issues found to support@alchemy.com.

NAME

lumosctl – Manual control for Lumos SSR controller hardware

SYNOPSIS

```
lumosctl [-dhkPRSVwXz] [-a <addr>] [-A <addr>] [-b <speed>] [-B <speed>] [-c
<file>] [-C <file>] [-D <sens>] [-E <sens>] [-H <hexfile>] [-p <port>] [-P <phase>]
[-s <file>] [-t <s>[orw+]:<init>:<seq>:<term>] <channel-outputs>...
```

Where *<channel-outputs>* may be any combination of:

$$\begin{aligned} &\langle \text{channel} \rangle @ \langle \text{level} \rangle [, \dots] \\ &\langle \text{channel} \rangle d [(: \text{steps}) [(: \text{time})]] \\ &\langle \text{channel} \rangle u [(: \text{steps}) [(: \text{time})]] \\ &\quad x \langle \text{id} \rangle \\ &\quad p \langle \text{time} \rangle \end{aligned}$$

DESCRIPTION

This command allows you to directly manipulate the state of a supported Lumos SSR controller unit, including administration functions such as changing the unit’s address, phase offset, etc.

Other software such as lumos(1) or—providing appropriate drivers are installed—popular third-party programs such as Vixen are more appropriate for performing (“playing”) sequences of light patterns on these boards. By contrast, lumosctl is more suited to setting up and configuring the boards (although some basic real-time control of channel outputs is possible using lumosctl).

In the absence of any command-line options to the contrary, the normal operation of lumosctl is to make a number of channel output level changes as determined by the non-option arguments which are of the form:

(channel)

or

(channel) @ *(level)* [, ...]

or

(channel) {u | d} [(: *steps*) [(: *time*)]]

In the first case, a channel number by itself means to turn on that channel to full brightness. In the second case, by specifying a level value (a number from 0 to 255, inclusive), that channel’s output is dimmed to the given level. Level 255 is the same as turning on to full brightness; level 0 is the same as turning it fully off.

In the third case, the dimmer level is ramped up smoothly from its current value to full brightness (“u”), or down smoothly until fully off (“d”). Optionally you may specify the number of dimmer level increments to increase or decrease at each change (1–128, default is 1); additionally, you may specify the amount of time to wait between each step, in units of 1/120 second (1–128, default is 1). As a convenience, this may be expressed as a real number of seconds followed by the letter “s”. Thus, the argument 13@127 sets channel 13 to half brightness. If this were followed by the argument 13u then channel 13 would be smoothly increased in brightness from there to full brightness (which is another 128 levels to take it from 127 to 255), by incrementing it one level every 1/120th of a second, reaching a full brightness level 128/120 seconds later (1.0666 seconds). If the argument 13d:10:2 were given, then channel 13 would drop to being fully off, going in steps of 10 levels at a time, 1/60th of a second between each step. Finally, an argument 10u:5:0.25s fades channel 10 up from its current value to full brightness by incrementing its value by 5 every quarter-second.

Bulk updating of channels is also supported. If multiple values are listed for a channel, such as: 10@0,0,255,255,127,40,30,20,10

Then the channel named (10 in this example) is assigned the first value (0), and the subsequent values are assigned to the immediately following channels (so channel 11 is set to 0, 12 is set to 255, and so forth).

In addition to the channel-setting argument described above, an argument of the form $x\langle i \rangle$ causes stored sequence $\langle i \rangle$ to be executed. Note that this is run in the “background”—any subsequent channel-setting arguments will be acted upon *while* the sequence is running. If a sequence was already running, it is stopped first. As a special case, $x0$ stops the currently-running sequence but does not start a new one.

A pause in the execution of the arguments may be effected by adding an argument of the form $p\langle t \rangle[s[ec[ond[s]]]]$ which makes `lumosctl` pause for $\langle t \rangle$ seconds before continuing on to the next argument. The $\langle t \rangle$ value need not be an integer.

A number of options are provided as described below. These command the SSR controller to perform certain administrative functions or configuration changes.

When giving multiple types of commands in one invocation of this program, they will be carried out in the following order:

1. Address Change
2. Kill all channels
3. Other configuration changes
4. Disable privileges
5. Channel(s) off/on/dim/etc.
6. Shutdown

OPTIONS

Each of the following options may be specified by either a long option (like “`--verbose`”) or a shorter option letter (like “`-v`”). If an option takes a parameter, it may follow the option as “`-a12`”, “`-a 12`”, “`--address 12`”, or “`--address=12`”.

Long option names may be abbreviated to any unambiguous initial substring.

- address=<addr>**
 (-a *addr*) Specifies the address of the target controller unit. The *addr* value is an integer from 0 to 15, inclusive. It defaults to 0.
- clear-sequences**
 (-S) Delete all stored sequences from the device’s memory.
- disable-sensor=<s>**
 (-D *s*) Disable inputs from the sensor(s) specified as the *s* parameter (which are given as a set of one or more letters, e.g., `--disable-sensor=ab`). The Lumos board will act as though those sensors were inactive regardless of their actual inputs. The special character “*” appearing in *s* means to disable all sensors.
- drop-privileged-mode**
 (-d) If the Lumos device is in privileged command mode (for configuration of the device), this will cancel that mode. Further privileged commands will not be recognized on that device.
- dump-configuration=<file>**
 (-C *file*) Dump the device configuration into the named *file*. See below for a description of the configuration file format.
- enable-sensor=<s>**
 (-E *s*) Enable inputs from the sensor(s) specified as the *s* parameter. See `--disable-sensor`.
- help**
 (-h) Prints a summary of these options and exits.
- kill-all**
 (-k) Turn off all output channels at once.
- load-configuration=<file>**
 (-c *file*) Load the device configuration from the named *file* and program that into the device.
- load-hex-sequence=<file>**
 (-H *file*) Load one or more pre-compiled sequences from the specified hex *file*. This is expected to be the output from the `lumosasm(1)` command.

--load-sequence=⟨file⟩

(-s ⟨file⟩) Load one or more sequences from the specified source ⟨file⟩ (see below for sequence source code syntax) and program them into the device. If another sequence already exists with the same number, it replaces the old one; however, beware that the controller device does not optimize memory storage, so eventually stored sequences may become fragmented, resulting in running out of storage space for them. To avoid this, it is best to clear all sequences using the --clear-sequences option, then load all the sequences you want on the device at once.

--port=⟨port⟩

(-p ⟨port⟩) Specify the serial port to use when communicating with the controller unit. This may be a simple integer value (0 for the first serial port on the system, 1 for the next one, etc.) or the actual device name on your system (such as “COM1” or “/dev/ttys0”).

--probe

(-P) Search for, and report on, all Lumos controllers attached to the serial network. If the --report option is also specified, this provides that level of output for every attached device; otherwise, it only lists device models and addresses.

--report

(-R) Report on the current device status to standard output in human-readable form.

--sensor=⟨s⟩[orw+]:⟨init⟩:⟨seq⟩:⟨term⟩

(-t ⟨s⟩[orw+]:⟨init⟩:⟨seq⟩:⟨term⟩) Define an action to be taken when a sensor is triggered. When the sensor is activated, the sequence ⟨init⟩ is run, followed by the sequence ⟨seq⟩ and then finally the sequence ⟨term⟩ when the sensor event is over. The sensor assigned this action is given as the parameter ⟨s⟩ and is one of the letters A, B, C, or D. This may be followed by the following option letters as needed:

- o Trigger once: play sequence ⟨seq⟩ only one time. The action will not be taken again until the sensor input transitions to inactive and then asserts itself as active again. This is the default action.
- r Repeat mode: play sequence ⟨seq⟩ indefinitely until explicitly told to stop (by an overt stop command such as an x0 argument, or another sequence being triggered manually or by sensor action).
- w Trigger while active: play sequence ⟨seq⟩ repeatedly as long as the sensor remains active. When the sensor input transitions to inactive again, terminate the action.
- + The sensor is to be considered “active” when at a logic high output (active-high mode). Normally, sensors are active-low (active when the input is at ground).

If 0 is specified for any of the sequence numbers, that means no sequence is called for that part of the trigger action.

--set-address=<addr>
 (-A <addr>) Change the device address to <addr>. This must be an integer in the range 0–15.

--set-baud-rate=<speed>
 (-B <rate>) Set a new baud rate for the device to start using from now on.

--set-phase=<offset>
 (-P <offset>) Set the phase *offset* in the device to the specified value. This must be an integer in the range 0–511. *This is an advanced setting which affects the ability of the AC relay boards to function properly. Do not change this setting unless you know exactly what you are doing.*

--sleep
 (-z) Tell the unit to go to sleep (this instructs the board to turn off a power supply which it is controlling, if any, but has no other effect).

--shutdown
 (-X) Command the unit to shut down completely. It will be unresponsive until power cycled or the reset button is pressed to reboot the controller.

--speed=<rate>
 (-b <rate>) Set the serial port to the given baud <rate>. [Default is 19200 baud.]

--wake
 (-w) Tell the unit to start the attached power supply from sleep mode. command is given at a future time.

--verbose
 (-v) Output messages to the standard output. Additional --verbose options increases verbosity. High levels of verbosity include a dump of every bit sent or received on the serial network.

CONFIGURATION FILE FORMAT

The files read and written by the --dump-configuration and --load-configuration options use a fairly standard configuration file format similar to the “ini” files used by early versions of Microsoft Windows and other systems. For full details of this format see <http://docs.python.org/library/configparser.html>, but the highlights include:

1. One data value per line (long lines may be continued by indentation ala RFC 822 headers).

2. Each line consists of the name of a data value, either an equals sign or a colon, and the value itself.
3. A syntax `%(<name>)s` can be used to substitute values into other values. Literal percent signs in values are simply doubled (“`%%`”).

All configuration data are contained in a stanza called “[lumos_device_settings]”. The values are:

`baud=<n>`

The configured serial I/O speed of the device. Supported values include 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, and 250000. Speeds slower than 9600 baud are not recommended. [Default is 19200.]

`dmxchannel=<n>`

If this field exists, the Lumos board is to run in DMX512 mode, with its channel #0 appearing at DMX512 slot #`<n>`, where `<n>` is an integer in the range 1–512. If this field is not present, the Lumos board will not be configured to recognize DMX512 packets at all.

`phase=<offset>`

The AC waveform phase offset for the unit. This should only be changed if needed due to some anomaly with the zero-crossing detector which throws off the unit’s timing. This is an integer in the range 0–511. [Default is 2.]

`sensors=<list>`

The value is a list of single letters in the range A–D. Each letter appearing in this list indicates that the corresponding sensor input should be enabled in the hardware. You must ensure that the hardware is really configured that way.

Sensor Configuration

For each sensor listed in the `sensors` field, a corresponding stanza called “[lumos_device_sensor_<x>]” appears, where `<x>` is the name of the sensor (“A”, “B”, “C”, or “D”), with the following fields:

`enabled=<bool>`

If “yes”, the sensor input is set to be monitored. If “no”, it is ignored. [Default is “yes”.]

`mode={once | repeat | while}`

Define the operating mode of the sensor trigger: play once per trigger, repeat forever until another trigger (or explicit command to stop), or play as long as sensor remains active. [Default is once.]

`setup=<id>`

Sequence *<id>* number to be played initially when the sensor becomes active

`sequence=<id>`

Sequence *<id>* number to be played as the main (possibly repeated) action for the sensor.

`terminate=<id>`

Sequence *<id>* number to be played when the action stops. Note that the main sequence might not have played to completion.

SEQUENCE SOURCE SYNTAX

Each source file given to --load-sequence contains one or more sequence definitions as described here. The formal syntax definition for the sequence language is:

```

<sequence> ::= SEQUENCE <id> [‘(’ <arg-list> ‘)’] <block>
<statement-list> ::= <statement> <newline>
| <statement-list> <statement> <newline>
<statement> ::= BLACKOUT
| CHANNEL <chan> <state>
| RAMP <direction> <chan> [BY <steps> [PER <time>]]
| CALL <id> [‘(’ <value-list> ‘)’]
| EXECUTE <id> [‘(’ <value-list> ‘)’]
| WAIT <time>
| FOR <symbol> '=' <value> T0 <value> [BY <value>] <block>
| IF <condition> <block>
| UNLESS <condition> <block>
| REPEAT [<value> [TIME[S]]] [AS <symbol>] <block>
| BREAK <symbol>
| CONTINUE <symbol>
| <symbol> '=' <value>
| SLEEP
| WAKE
| SUSPEND [WITH UPDATE]
| RESUME [WITH FADE]
| EXIT
<block> ::= ‘:’ <newline> <statement-list> END
<id> ::= 0 | 1 | ... | 127
<chan> ::= <value> (allowed range 0...63)
<dimmer-value> ::= 0 | 1 | ... | 255
<percentage> ::= 0 | 1 | ... | 100
<state> ::= ON | OFF | <dimmer-value> | <percentage> '%'

```

```

<direction> ::= UP | DOWN
<steps> ::= 1 | 2 | ... | 128
<time> ::= <cycles> | <real> SEC[ONDS]
<cycles> ::= 1 | 2 | ... | 128
<real> ::= <digits> ['.'] [<digits>] | [<digits>] '.' <digits>
<condition> ::= SUSPENDED
    | SENSOR <sens-list>
    | <value> <comparison> <value>
<comparison> ::= '<' | '>' | '==' | '!='
    | '<=' | '>='
<arg-list> ::= <symbol> | <arg-list> ',' <symbol>
<value-list> ::= <value> | <value-list> ',' <value>
<value> ::= <int>
    | <symbol>
    | <value> '*' <value>
    | <value> '+' <value>
    | <value> '-' <value>
    | <value> '/' <value>
    | '(' <value> ')'
<int> ::= ['-'] <digits>
<symbol> ::= <letter> | <symbol> <digit> | <symbol> <letter>
<digits> ::= <digit> | <digits> <digit>
<digit> ::= '0' | '1' | ... | '9'
<letter> ::= '_' | 'A' | 'B' | ... | 'Z'
<sens-list> ::= <sens> | <sens-list> OR <sens>
<sens> ::= 'A' | 'B' | 'C' | 'D'
<newline> ::= '\n'

```

While the language keywords are shown here in upper-case letters, in fact all symbols are interpreted irrespective of case throughout the sequence source file.

Expressions

Simple math expressions are supported, including addition, subtraction, multiplication, and division, but the values supported by the hardware runtime system are exclusively 8 bit unsigned integers. Division is not supported at all in the hardware. Math involving compile-time values is not so limited, but if the expression evaluation remaining to be done at runtime it too complex, the compiler will issue a fatal error.

Flow Control Constructs

Each of these constructs operates on a *<block>* of code. A block is a sequence of one or more lines of code (each terminated by a newline), ending with an END statement.

IF SENSOR $\langle s1 \rangle$ [OR $\langle s2 \rangle$] [...]: ... END
 Execute the statements in the block if any of the sensors are currently active.

IF SUSPENDED: ... END
 Execute the block if the unit is currently in a suspended state.

IF $\langle condition \rangle$: ... END
 Execute the block if the condition holds. This is a simple comparison between two values, where the comparison may be equals, not equals, less than, greater than, less than or equal, or greater than or equal as represented by the operators ==, !=, <, >, <=, and >=, respectively.

UNLESS ...
 All of the forms of the IF construct may be used with the word UNLESS substituted for IF. In this case, the sense of the conditional test is reversed.

REPEAT [$\langle n \rangle$ [TIME[S]]] [AS $\langle var \rangle$]: ... END
 Repeat the statements in the block until forced to quit via EXIT or BREAK. If a repeat count $\langle n \rangle$ is given, then at most only repeat the statements that many times. The AS clause allows the current loop counter (which begins at 0 and counts up to, but never reaches, $\langle n \rangle$) to be visible within the block as the variable $\langle var \rangle$.

FOR $\langle var \rangle = \langle start \rangle$ TO $\langle end \rangle$ [BY $\langle step \rangle$]: ... END
 Repeat the statements in the block once for each value of $\langle var \rangle$ from $\langle start \rangle$ to $\langle end \rangle$, inclusive. If $\langle step \rangle$ is given, $\langle var \rangle$ is incremented by that amount between each iteration. The default $\langle step \rangle$ is 1. The behavior of this loop may be modified by BREAK or CONTINUE statements just like the REPEAT loops are.

BREAK [$\langle var \rangle$]
 Terminate the innermost loop immediately. If an outer loop is to be terminated, then specify its index $\langle var \rangle$ to identify which loop is the target of this statement.

CONTINUE [$\langle var \rangle$]
 Begin the next iteration of the innermost loop immediately. If an outer loop is to be continued, then specify its index $\langle var \rangle$ to identify which loop is the target of this statement.

SEQUENCE $\langle id \rangle[(\langle var \rangle, \dots)]$: ... END
 Define a stored sequence consisting of the statements to the END. The $\langle id \rangle$ may be a number from 0 to 127, with a few caveats:

#0: Sequence #0 may *never* be explicitly invoked by anything. This sequence is invoked automatically during device boot to initialize the unit.

#1–63: Sequences in this range are stored in EEPROM and will survive a power failure or device reboot. Note that there probably won't be enough available memory to actually store 63 sequences.

#64–127: Sequences in this range are stored in RAM and will *not* survive a power failure or device reboot. Note that there probably won't be enough available memory to actually store 64 sequences in RAM.

If input parameters are expected, they are given symbolic names inside parentheses between the sequence $\langle id \rangle$ and the colon. A maximum of four parameters are allowed.

Commands

Each statement is described briefly here. For more details, see the Lumos board user manual.

BLACKOUT

Immediately turn off all channel outputs.

CALL $\langle id \rangle[\langle value \rangle, \dots]$

Suspend the execution of the current sequence and begin executing sequence # $\langle id \rangle$. When that sequence has completed, execution of the current sequence will resume where it left off. Up to four parameters may be passed to the called sequence by placing them inside parentheses after the sequence $\langle id \rangle$. The $\langle id \rangle$ cannot be 0.

CHANNEL $\langle c \rangle$ ON|OFF| $\langle value \rangle$ [%]

Change the output level of channel $\langle c \rangle$ to fully on, fully off, or to a specific value. If the value is followed by a percent sign ("%"), then the value is assumed to be a percentage in the range 0–100.

EXECUTE $\langle id \rangle[\langle value \rangle, \dots]$

Abandon the current sequence and begin executing the specified sequence instead. When that sequence has completed, execution of the current sequence will not resume. Up to four parameters may be passed to the new sequence by placing them inside parentheses after the sequence $\langle id \rangle$. Note that the $\langle id \rangle$ cannot be 0.

EXIT

Terminate execution of the current sequence.

RAMP UP|DOWN $\langle c \rangle$ [BY $\langle steps \rangle$ [PER $\langle time \rangle$ [SEC[ONDS]]]]

Start fading channel $\langle c \rangle$ up or down from its current output level to the maximum or minimum value. This is done by adding or subtracting the value of $\langle steps \rangle$ (which is in the range 1–128) each time, pausing for $\langle time \rangle$ between each value change. If the $\langle time \rangle$ value is followed by SEC or SECONDS, it is assumed to be the number of seconds between

level changes (a real number in the range 0.0083–1.0667). Otherwise, it is assumed to be an integer in the range 1–128 specifying the number of 1/120ths of a second for the pause.

RESUME [WITH FADE]

Resume from SUSPEND mode. If the WITH FADE option is given, then all channels are faded down to fully off, and then up to their last-known state is supposed to be.

SLEEP

Enter sleep state. The load power supply is commanded to shut down (if such a supply is connected that way to the controller). Note that the unit may still wake on its own as needed, possibly even immediately.

SUSPEND [WITH UPDATE]

Suspend input of commands from outside. From this point forward, only saved sequences will affect channel outputs. If the WITH UPDATE option is given, then external commands are still accepted but have no effect on the outputs until after a RESUME is executed. Note that a unit may still remain in suspended state after the sequence putting it in that state exits. It only ends upon RESUME.

WAIT $\langle t \rangle$ [SEC[ONDS]]

Pause execution of the script for the designated time. The time $\langle t \rangle$ is interpreted identically to that described above for the RAMP command.

WAKE

Wake up the unit from SLEEP mode. The power supply is commanded to start (if a suitable one is appropriately configured).

AUTHOR

Software Alchemy / support@alchemy.com

COMPATIBILITY

This version of `lumosctl` is compatible with the following boards:

- Lumos 48-channel controller version 3.1 or 3.2 *providing it has been upgraded or installed with ROM firmware version 3.0 or later* (boards with ID markings beginning with “48CTL-3-1” or “LUMOS-48CTL-3.2”). (Whether this controller is driving AC or DC boards is irrelevant.)
- Lumos 24-channel DC controller version 1.0 (boards with ID markings beginning with “LUMOS-24SSR-DC-1.0”).

HISTORY

This program first appeared under the name 48ssrctl and was used only for the Lumos 48-channel AC controller boards, employing the older firmware (ROM versions less than 3.x).

This document describes version 2.0 of this utility, which is the first to carry this name and to include the expanded features for firmware version 3.0.

SEE ALSO

`lumosasm(1)`.

LIMITATIONS

This program does not send DMX512 commands to the device(s), only Lumos native commands.

BUGS

The sequence language is constrained by the limits of the hardware (such as 8-bit unsigned integer values and limited arithmetic expression evaluation), by the need to be compiled to fit in a very small memory space. As such, the optimization toward certain use cases and against others may seem odd at first, but it serves that purpose.

Submit any other issues found to support@alchemy.com.

TROUBLESHOOTING

factory reset

GLOSSARY

CLI

Power cycle

SSR (Solid-State Relay): A device which controls an external power load.

In contrast to a mechanical relay, an SSR has no moving parts, but does its switching electronically.

ACKNOWLEDGEMENTS

Kickstarter Project

We launched a Kickstarter project to build a test network of Lumos DC boards for final testing and debugging before releasing the final designs and firmware as an open source DIY project.

Thank you to all our Kickstarter backers who made the final testing of the Lumos DC controllers possible!

Fan Level

Amanda Allen

Supporter Level

Carey Adams
Sue Allen
Andrej Čibej
Betsy Fernley

Beth Gordon
Sara Jacobson
Tanya Spackman

Backer Level

DC

Silver Level

David Johnston

Melf

Gold Level

Rob Beasley

Phil Willoughby

Patron Level

Casey Annis
William H. Ayers
Jon and Rebecca Garver
Andy Kitzke
Joseph Moss

Robert A. Nesius
Patrick Quinn-Graham
Jama Scaggs
Doug Van Camp
Matthew Wentworth

We also wish to thank Darren Bliss who has been a great supporter of the Lumos project since the very first prototype was being experimented with, and the other Kickstarter backers and friends who offered moral support, other contributions, or who wished to contribute anonymously.

Technical Legacy

The do-it-yourself computerized Christmas light hobby thrives as a community of enthusiasts who contribute their ideas and designs for others to build, enjoy, and improve upon with new designs of their own. This journey began for me years ago with the discovery of Hill Robertson's Computer Christmas website (www.computerchristmas.com). It continues on many other sites including Chuck Smith's Planet Christmas (www.planetchristmas.com), doityourselfchristmas.com, and many others.

Over the years the users of these forums have produced some great designs which have become *de facto* standards as others adopt them and refine them in their own designs. The Lumos boards' TRIAC and MOSFET relay circuits (the final few components at the controlled outputs) are a continuation of the standard circuits used by those communities, inspired most by Robert Stark's TRIAC design and the DC MOSFET circuits by John Wilson (from Computer Christmas and Do It Yourself Christmas, respectively). I am pleased to contribute my own innovations on these common design themes back to the same community (the remainder of the Lumos circuits other than the TRIAC and MOSFET output sections is entirely my own original design).

INDEX

- AND, 44
- CFL, 71
 - choking hazard, 3
 - Christmas lights, 1
- CLI, 51, 81, 99
- CPU, 11
- danger warnings, ii, 1, 3, 4
- DIY, 2, 76, 101
- DMX, 19, 61, 69
 - DMX512, iii, 5, 9, 14, 18, 19, 61, 66, 67, 69, 70
- DTR, 6
- EDA, 105
- EEPROM, 17, 27, 62, 78
- ESD, 4
- GFIC, 3
- GUST, 105
- LED, iv, 6, 9, 12, 14, 17, 61, 66, 71, 72, 76–78
 - maximum limits, 4
- MOSFET, 76, 102
- MSB, 54
- NAK, 64
- NAND, 44
- NOR, 44
- online forum, 2
- OR, 44
- overloaded circuits, 4
- PCB, 7, 105
- PDF, 105
- phase offset, 73
- product website, 2
- PWM, 71, 72, 74
- RAM, 11, 27, 62
- ROM, ii, 60, 80
- safety precautions, 3
- SFB, 29, 30, 34–36, 38–41, 48
- shock hazard, 3
- SSR, 76, 99
- static discharge, 4
- theatrical lighting, 1
- TRIAC, 71, 72, 76, 102
- TTL, 5
- USA, 71
- warnings, ii, 1, 3, 4
- warranty, limitation of, ii, 1
- website, 2

COLOPHON

This manual was composed and typeset by the author using L^AT_EX with Memoir layout macros, augmented by wrapfig, lettrine, bytefield, wallpaper, TikZ, and a host of miscellaneous behind-the-scenes working packages.

It was set 10/12 pt using the T_EX Gyre Schola font family created by GUST, the Polish T_EX User Group. This typeface is based on URW Century Schoolbook L, originally designed by Morris Fuller Benton in 1919, for the American Type Founders.

Schematics were generated using the gEDA tool gschem. The PCB layout illustrations were created by pcb on Linux. All of the above are free and open-source tools.

Published electronically in PDF format for ease of viewing on any platform.