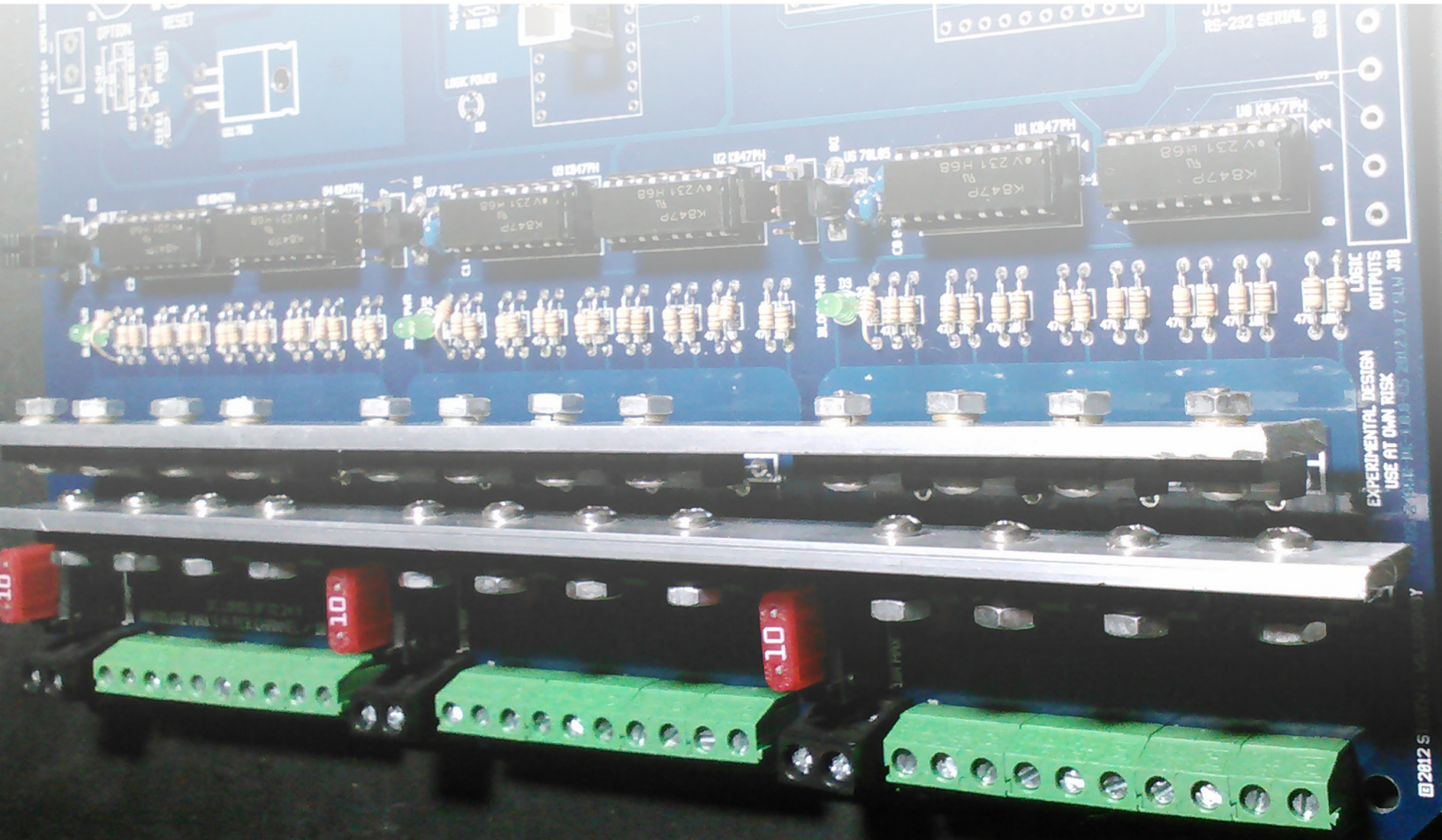
Using the Lumos™ SSR Controllers



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 2.2, for Lumos controllers with ROM version 3.1.

Contents

List of Figures

CHAPTER 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 interace 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 koow 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.madscience.zone/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.

1.6 DMX Compatibility Warning

These boards do not support the DMX512 protocol, although we have experimented with this as a feature. As such, the firmware includes a "DMX512 mode" which the board and accompanying software will recognize. However, this is still experimental and does not yet function enough for actual

production use. While this feature may be ready for use in a future upgrade to these boards, at present the Lumos boards will only work in native Lumos (not DMX512) mode.

Any and all references in the Lumos manuals to "DMX" modes should be regarded as describing experimental features not yet ready for actual production use.

CHAPTER 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 GFIC-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.

Also note that the Lumos output circuits were designed to control simple resistive loads such as incandescent lights. They are not appropriate for all kinds of loads. Some inductive loads (for example, electromagnetic relays and motors) may require a protective "snubber" circuit which would be a custom modification to the Lumos circuit requiring qualified engineering work.

Do not plug any load into a Lumos board which cannot be dimmed.

CHAPTER 3

Overview of the Lumos Controller Models

There are currently five 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.

Although a few prototypes of this board have been built and used suc-

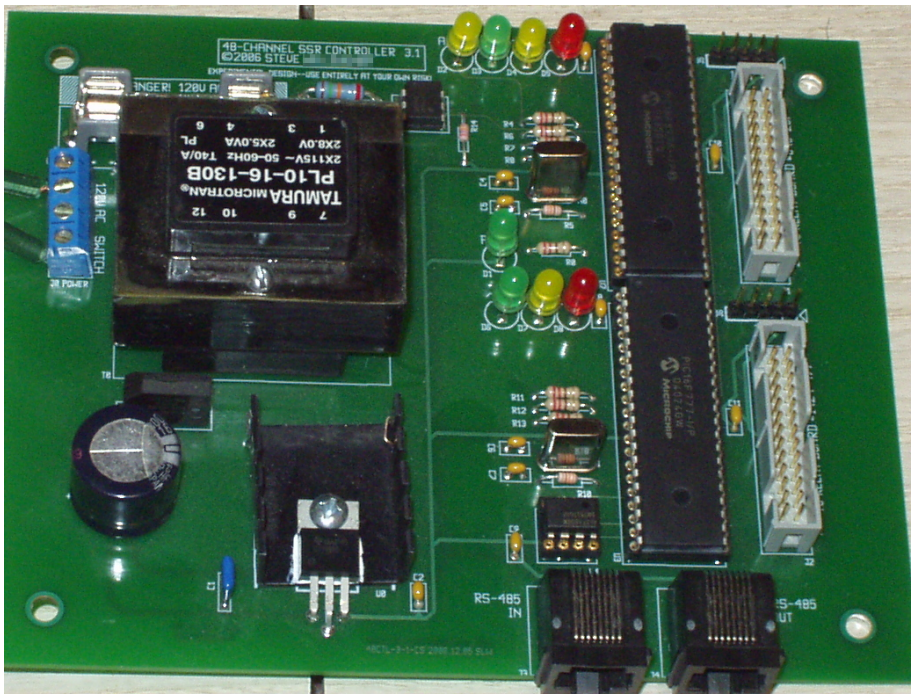


Figure 3.1: Lumos 48-Channel Controller

cessfully, 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@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.

Although a few prototypes of this board have been built and used suc-

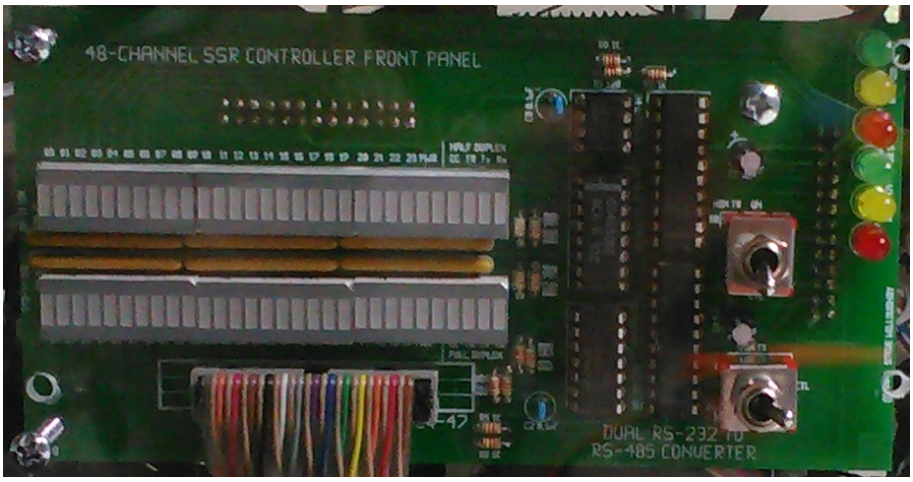


Figure 3.2: Lumos 48-Channel Front Panel

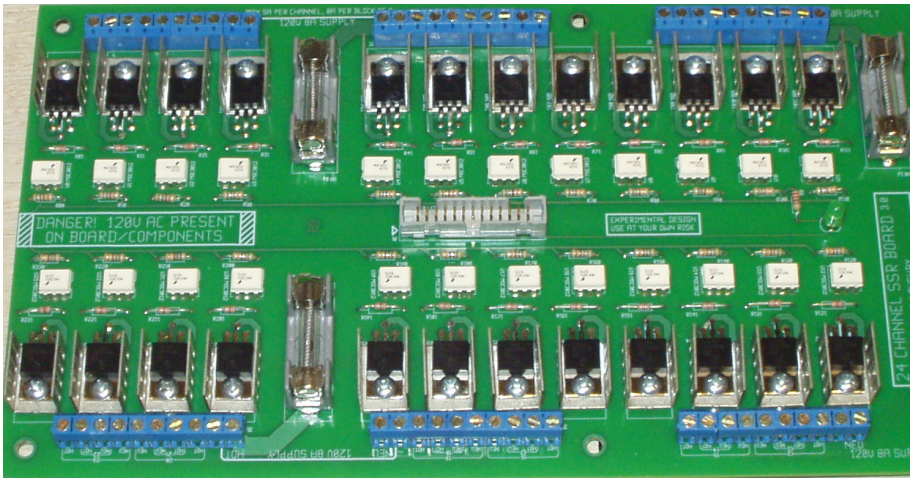


Figure 3.3: Lumos 24-Channel AC Relay

cessfully, 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

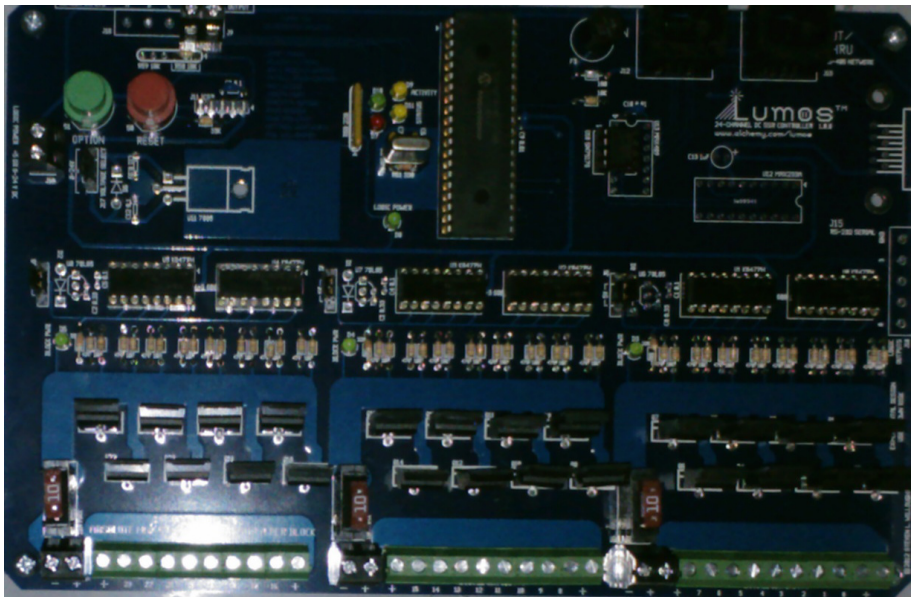


Figure 3.4: Lumos 24-Channel DC Controller/Relay

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.

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

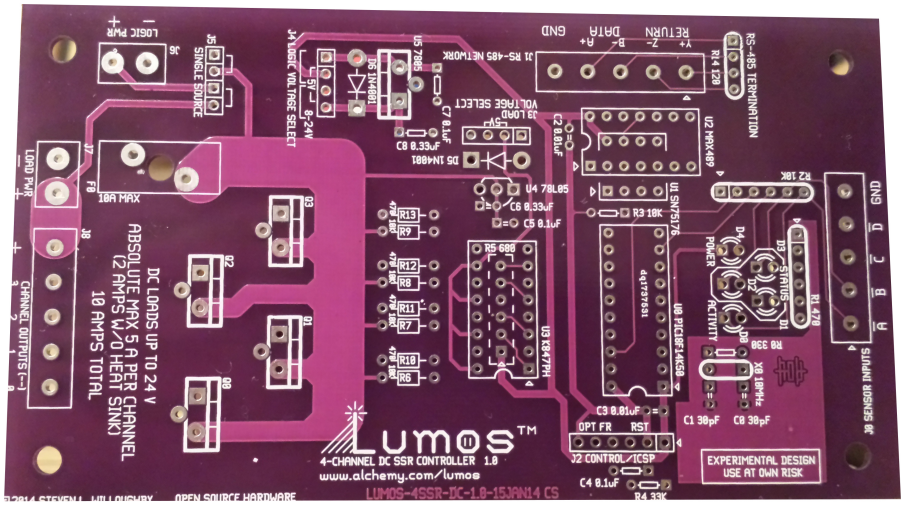


Figure 3.5: Lumos 4-Channel DC Controller

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.

3.5 4-Channel DC Controller

This board controls four output channels for low-voltage DC loads. The controlled channel block is separately powered and completely isolated from the logic side of the board It 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 at any given time.

It uses one address for Lumos-protocol communication, or 4 DMX512 slots. Communications options (selected permanently at the time the controller is assembled) include full- 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.

CHAPTER 4

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 all faults and error conditions, and re-initializes all the hardware and software components.

Four-channel boards don't include this button. To reset one of these boards, insert a jumper across pins 2–3 of J2 as shown in Figure 4.1. This will halt the board. Then remove the jumper, which will reset and restart the board.

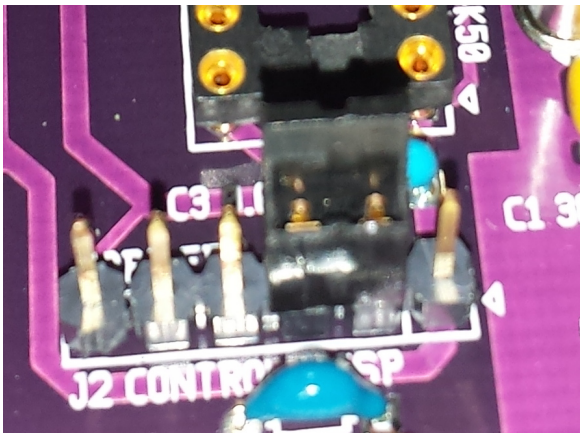
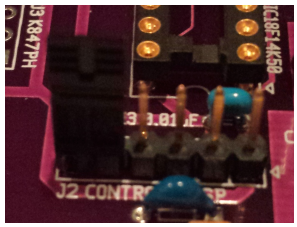


Figure 4.1: Resetting a 4-Channel DC Lumos Board Via Jumper on J2-2 and J2-3

4.2 Entering Configuration Mode

Some functions are only enabled when the controller is in a special "configuration" mode[1] to prevent potentially harmful effects such as accidentally changing the device's address, baud rate, or other configuration parameters.



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 four-channel boards do not have OPTION and RESET buttons directly (unless off-board buttons were used, connected to the board at J2). These boards are controlled by inserting jumpers onto header J2, as follows:

1. Insert a jumper across pins 5–6 of J2.
2. Wait until the LEDs start flashing rapidly (approximately 2 seconds).
3. Remove the jumper.

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

[1] In some places in other documentation, including the controller's firmware source code, this is also referred to as "privileged mode". This term is deprecated. The preferred term is now "configuration mode."

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

of rebooting the system). (For four-channel boards, follow the instructions in the previous section to reset the board.)

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 for another 2 seconds, then release. (For four-channel boards, insert and remove the jumper across pins 5–6 again, leaving the jumper in place for about 2 seconds, as you did to place the board in configuration mode.)

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 in 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 4.2 for a reference chart of the output codes. Four-channel boards use the same display pattern as the 24-channel boards.

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. Four-channel boards may be paused by briefly inserting and then removing a jumper across pins 5–6 of J2.

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. Disconnect any input sensors and power supply control wires from the Lumos controller before beginning.

24-Channel Boards

1. Turn off the Lumos controller or press and hold the red RESET button.
2. Install a jumper to short pins 4 and 5 of J11 (labeled "ICSP" on the board).
3. Power on the Lumos controller or release the RESET button.

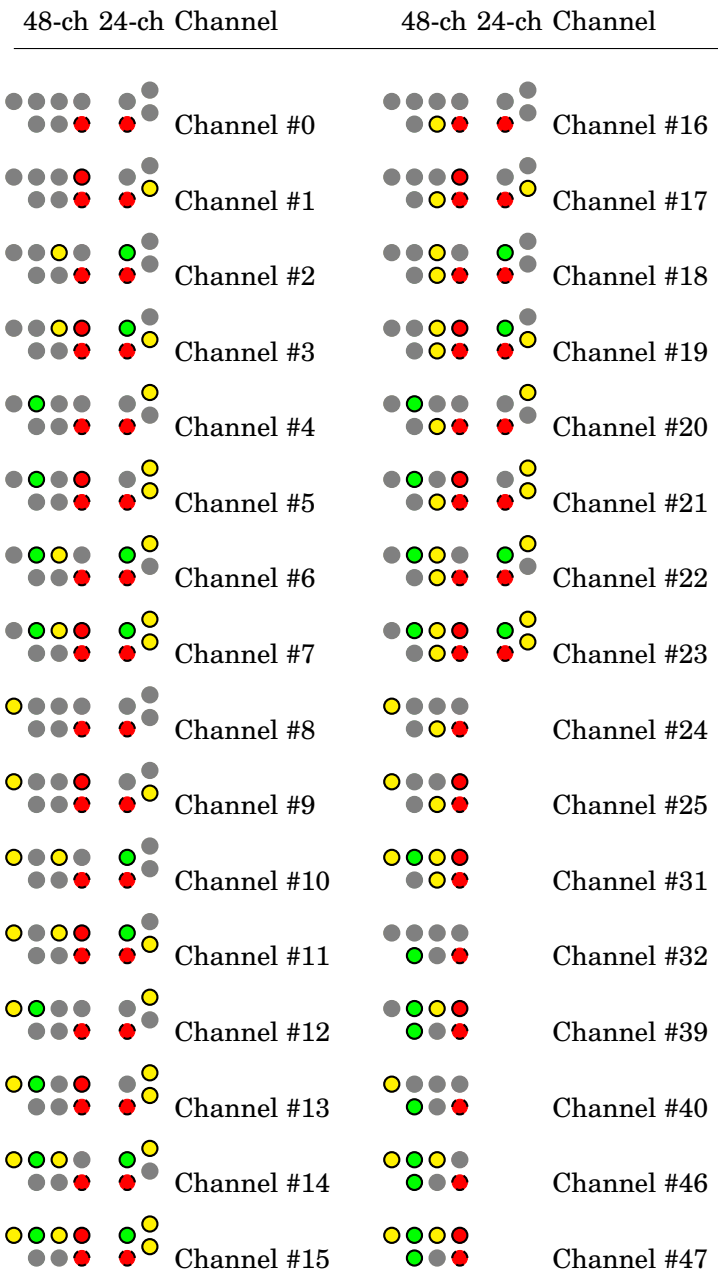
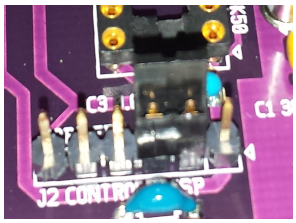


Figure 4.2: Test-mode Channel Indicators

4. When the red, green, and yellow LEDs flash rapidly, pull the jumper out.



4-Channel Boards

1. Turn off the Lumos controller or insert a jumper across pins 2–3 of J2.
2. Install a jumper across pins 5–6 of J2.
3. Remove any sensors attached to the unit.
4. Insert a jumper wire between sensor input terminals $\overline{\text{A}}$ and $\overline{\text{B}}$.
5. Power on the Lumos controller or remove the jumper from pins 2–3.
6. When the red, green, and yellow LEDs flash rapidly, pull the jumper from the sensor inputs.
7. Finally, pull the jumper off pins 5–6.

If those steps were carried out exactly as described, 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.

CHAPTER 5

CONFIGURING THE BOARD

THERE 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 in 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 Read-Only Mode

Sometimes you will use the Lumos board in situations where you can't receive a reply from the device. This may, for example, happen if you use an RS-485 converter which only transmits data, but doesn't receive any back, or if there is a problem with your PC trying to control the data direction on a half-duplex network.

In this case, use the `--read-only` option to `lumosctl`. This suppresses the features of `lumosctl` which monitor the Lumos board's configuration and current state. Normally, this helps confirm that the requested changes took effect, but if your PC is unable to receive information back from the Lumos board, `lumosctl` won't be able to work without the `--read-only` option.

5.2 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.

5.3 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.

5.4 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 the stanza tag "`[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.

The following describes a feature of the Lumos controller which is planned for a future release but not implemented today. While this description follows the expected behavior the Lumos board

will have when that feature is actually available, it is still under development and subject to change.

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 105 in the appendices.

5.5 Setting a Lumos Controller to Use DMX512

The DMX512 support in Lumos is experimental and not expected to work yet. If you wish to experiment further to refine this feature for a future release of the Lumos firmware, contact the author with your findings and recommendations.

Setting up the controller to be a DMX512 device is another task performed via the configuration file (see Section 5.4 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 new 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.

5.6 Canceling Configuration Mode

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

```
$ lumosctl --disable-configuration-mode
$ lumosctl --forbid-configuration-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.

Chapter 6

Creating Programmed Sequences

Stored programmed sequences are a planned feature for the Lumos boards but is not yet implemented. This chapter will fully describe the programming environment when that feature is ready for use.

6.1 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.

6.2 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 naïve 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
$ _
```

6.3 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-based[1] 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.

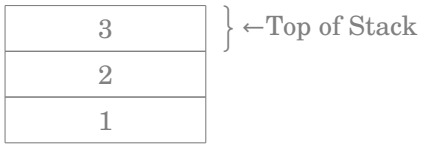
Stacks

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.

[1] Several 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.

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:



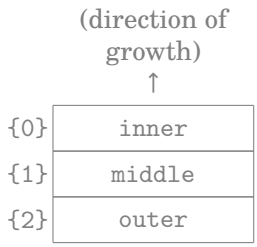
As we "pop" values off the stack, we pull them from the top of the stack, so they are retrieved in the reverse order in which they were pushed onto the stack.

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
```

Then the loop stack would look like:



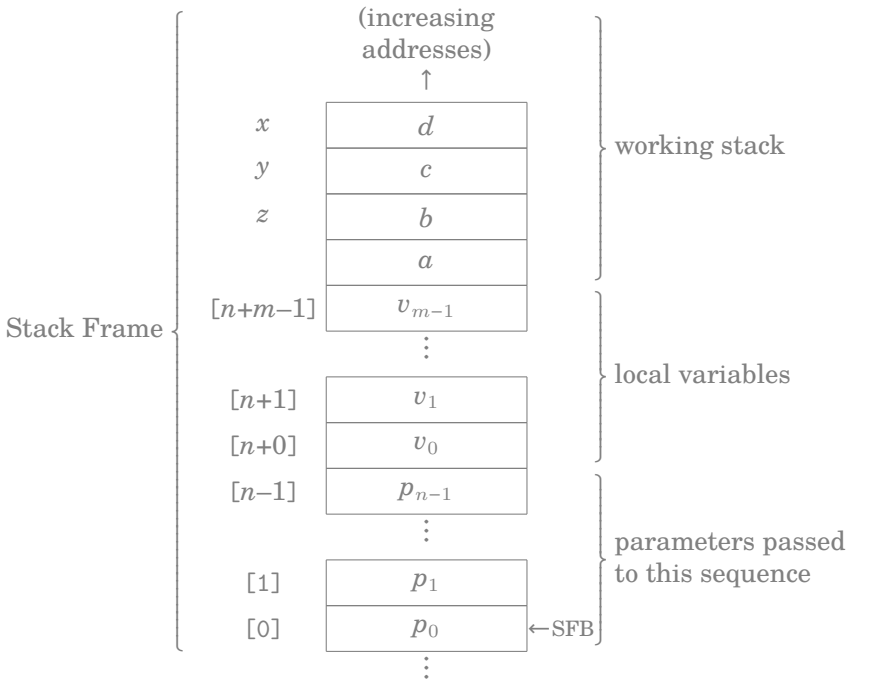
At any given time, `{0}` refers to the top of the loop stack, which is the innermost loop level currently running. Likewise, `{1}` is the counter for the next loop out from there, and so on.

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 INNER`" 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.

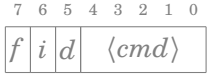
At each call, parameters are pushed onto the stack and then the stack frame boundary (SFB) pointer is moved to point to the first parameter. This way, parameters and local variables are referenced within the sequence as offsets from the frame pointer, while working values are obtained by pushing and 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 of them 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 instruciton 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 inerent 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	0x09	`push #9`
loop default	0	1	1	0x0F		`push {0}`
indirect stack	1	0	0	0x0F		`push [$]`
indirect default	1	0	1	0x0F		`push [0]`
indirect	1	1	0	0x0F	0x09	`push [9]`
loop stack	1	1	1	0x0F		`push {$}`

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 L:   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 $,$
```

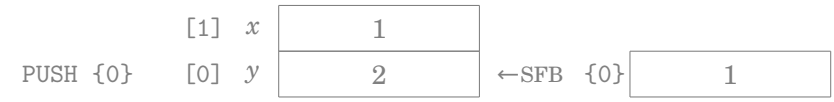
```
0010 49 64          0014      WAIT   #100
0012 AF             0015      PUSH   [0]
0013 52 03          0016      MUL    $,#3
0015 B0             0017      POP    [0]
0016 6B 02 14       0018      NEXT   L,#1,#20
```

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

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.

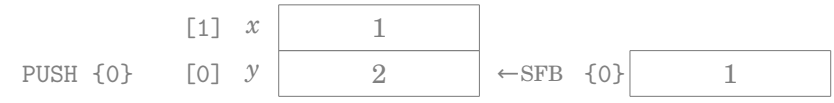
A new loop context is created with an initial counter value of 1 (now the top of the loop stack).

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.



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

Now we need to evaluate 6*x+12.



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

	[1]	*x*	6			
MUL $,#6	[0]	*y*	2	←SFB	{0}	1

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

	[1]	*x*	18			
ADD $,#12	[0]	*y*	2	←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`:

	[2]	*x*	1			
	[1]	*y*	18			
PUSH {0}	[0]	*z*	2	←SFB	{0}	1

	[2]	*x*	11			
	[1]	*y*	18			
ADD $,#10	[0]	*z*	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.

	[3]	*x*	2			
	[2]	*y*	11			
	[1]	*z*	18			
PUSH [0]	[0]		2	←SFB	{0}	1

	[2]	*x*	13			
	[1]	*y*	18			
ADD $,$	[0]	*z*	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]	*x*	2	←SFB	{0}	1

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

WAIT #100	[0]	*x*	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 the pop that

result back into memory location [0] again.[2]

	[1]	*x*	2			
PUSH [0]	[0]	*y*	2	←SFB	{0}	1
	[1]	*x*	6			
MUL $,#3	[0]	*y*	2	←SFB	{0}	1
POP [0]	[0]	*x*	6	←SFB	{0}	1

We then start the next iteration.

NEXT L,#1,#20	[0]	*x*	6	←SFB	{0}	2

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

	[0]	*x*	34	←SFB	{0}	20

(Note that the value in [0]—the local variable "base"—is multiplied by 3× every iteration, so although the value in the 8-bit memory location is currently 34, it has 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.

NEXT 2,#1,#20	[0]	*x*	34	←SFB	{0}	(empty)

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

[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.

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
```

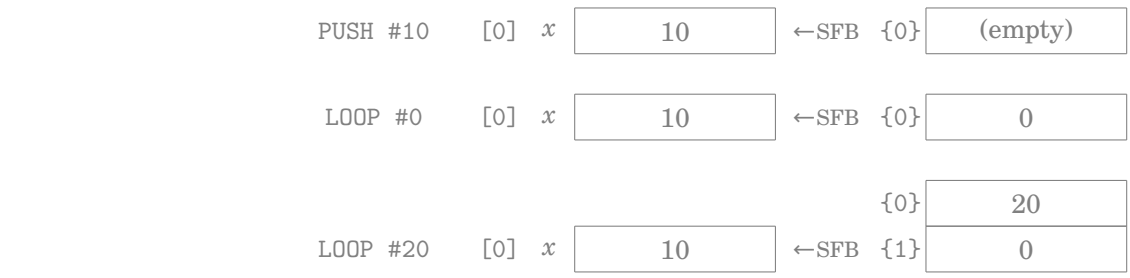
This compiles into the following bytecode:

```
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
```

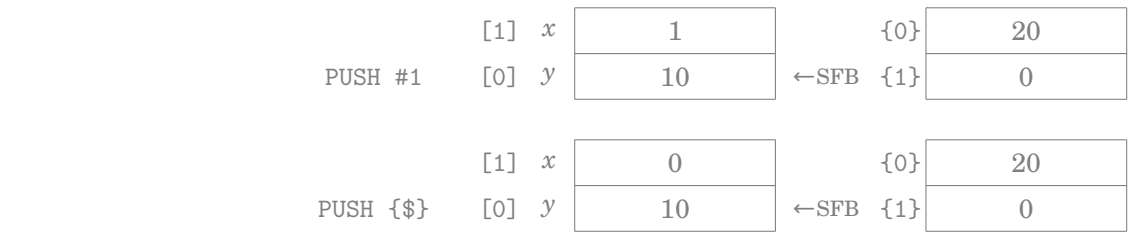
```
SEQUENCE #5 ASSEMBLY LISTING
ADDR BYTES          LINE SOURCE CODE
---- -------------- ---- ----------------------------
0000 40 02          0019  SSFB    #2
0002 4F 42          0020  PUSH    #0x42
0004 AF             0021  PUSH    [0]
0005 CF 01          0022  PUSH    [1]
0007 02             0023  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
```

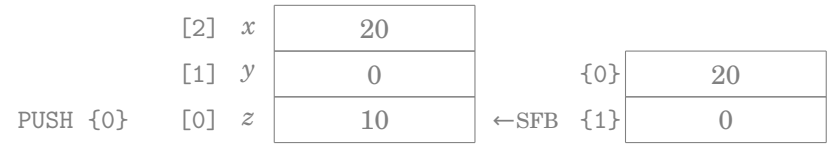
We'll step through the execution of these routines to show how the stack frames look between calls.

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

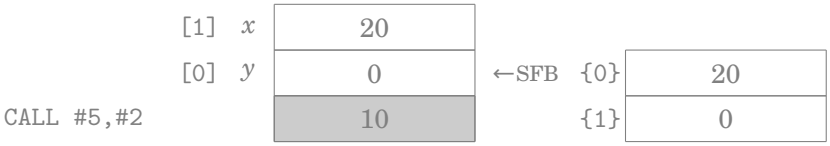


We push the two parameters onto the stack in the order shown in the source code (i.e., the deepest value will be the first parameter). Since the call is "`CALL 5(x,y)`" we need to push the outer loop counter `x` (which is at position `{1}` on the loop stack), then the inner loop counter `y` (which is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



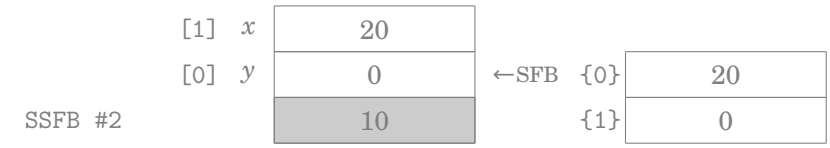


Now we make the call. The first parameter to `CALL` is the sequence number we're calling, and the second is the number of parameters being passed (2 in this case).

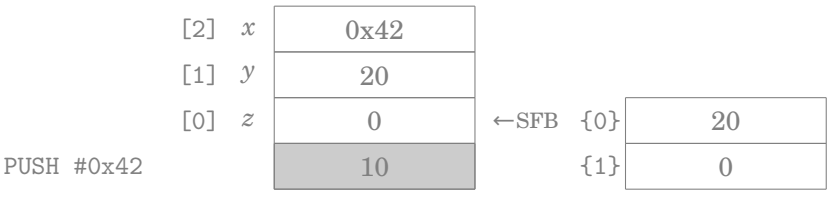


At this point, control transfers into sequence 5. Note that the stack frame boundary moved up until the frame includes only the top 2 items (the two parameters passed into this sequence). This means the parameters are directly accessible as `[0]` and `[1]`, and the local variable sequence 5 is about 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.

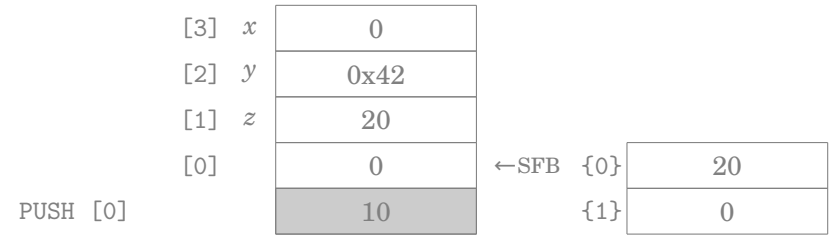


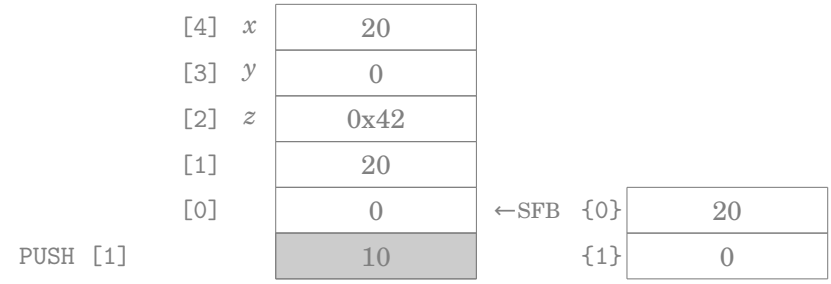
Next, it will push the initial value of the local variable `foo`, so we can get to it at location `[2]`.

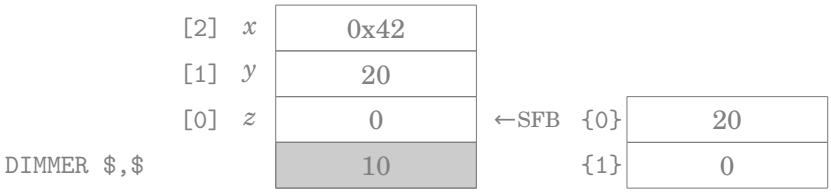


Now we set the dimmer value of channel `[1]` to level `[0]`.

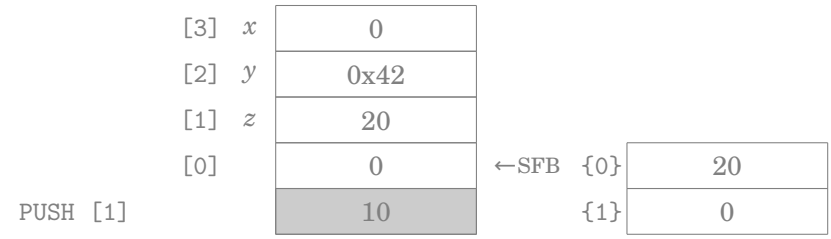
[3] Yes, if you're clever or careless enough, you can destroy the integrity of the stack(s) to retrieve or alter the values below SFB, but we're not recommending that.

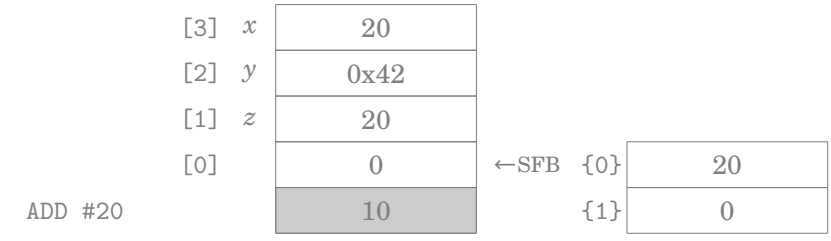


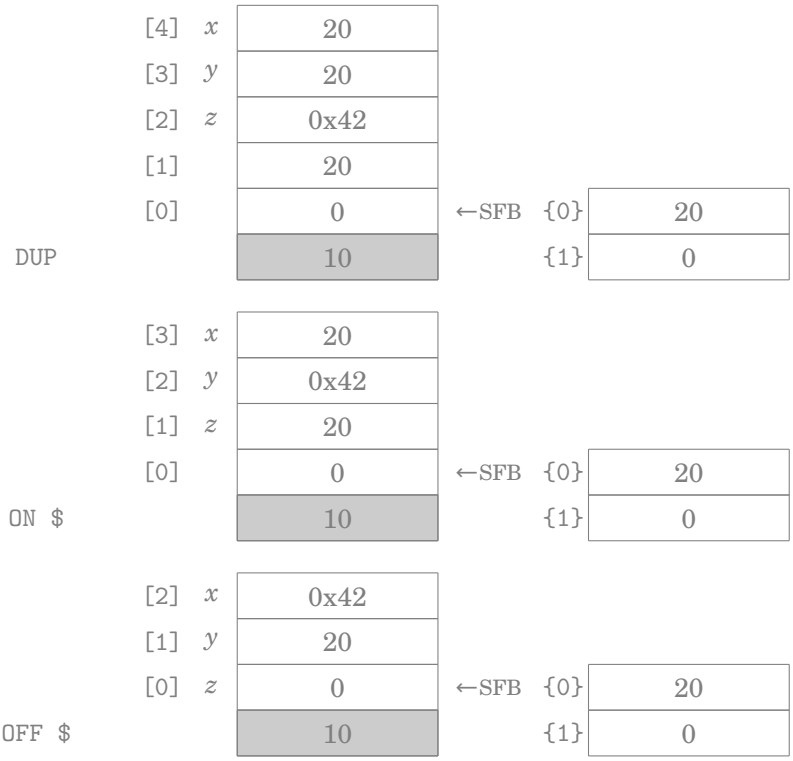




Now we calculate the value of [1]+20 which will be used for both the ON and OFF commands.

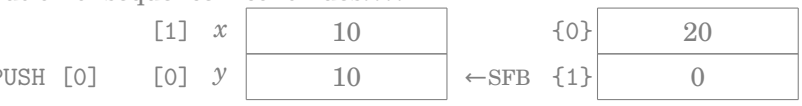






Now sequence 5 is done. As it exits, the calling sequence resumes where it left off, and the stack frame is destroyed, restoring the SFB to its previous location.

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 ⟨v⟩	ADD $,#⟨v⟩	$x + \langle v \rangle \rightarrow x$.
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 x 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 excuting 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 x with y params.
28	CALL $	Call x 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 x to level y.
42 ⟨ch⟩ ⟨lvl⟩	DIMMER #⟨ch⟩,#⟨lvl⟩	Set ⟨ch⟩ to ⟨lvl⟩.

`EXCH`: Exchange Data Stack Values

Swaps the top two values on the data stack ($x \leftrightarrow y$).

Bytes	Instruction	Description
`11`	`EXCH`	Swap x and y.

`EXEC`: Start New Sequence

Begins excuting a new sequence, possibly with parameters passed into it. This sequence replaces the current one. When the called sequence ⟨*seq*⟩ 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`	⟨*seq*⟩	⟨*np*⟩	`EXEC`	`#`⟨*seq*⟩`,#`⟨*np*⟩	Run ⟨*seq*⟩ with ⟨*np*⟩ params.
`E8`	⟨*seq*⟩		`EXEC`	`#`⟨*seq*⟩	Run ⟨*seq*⟩ 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 ⟨*ch*⟩ by subtracting ⟨*step*⟩, with a delay of ⟨*t*⟩/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`	⟨*ch*⟩	⟨*step*⟩	⟨*t*⟩	`FADE`	`#`⟨*ch*⟩`,#`⟨*step*⟩`,#`⟨*t*⟩	Fade ⟨*ch*⟩ by ⟨*step*⟩ per ⟨*t*⟩.

`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 ⟨*addr*⟩.

Most of these compare two values for equality or inequality:

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

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

Bytes		Instruction		Description
0CC*q*	⟨*addr*⟩	IFANY	⟨*addr*⟩,⟨*flags*⟩	Jump to ⟨*addr*⟩ if any of ⟨*flags*⟩ true.
4CC*q*	⟨*addr*⟩	IFALL	⟨*addr*⟩,⟨*flags*⟩	Jump to ⟨*addr*⟩ if all of ⟨*flags*⟩ true.
8CC*q*	⟨*addr*⟩	IFNONE	⟨*addr*⟩,⟨*flags*⟩	Jump to ⟨*addr*⟩ if none of ⟨*flags*⟩ true.
CCC*q*	⟨*addr*⟩	IFNALL	⟨*addr*⟩,⟨*flags*⟩	Jump to ⟨*addr*⟩ if not all of ⟨*flags*⟩ 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 ⟨*address*⟩.

Bytes			Instruction		Description
03	⟨*addr*⟩		JUMP	⟨*addr*⟩,$	Jump to ⟨*addr*⟩+x
23	⟨*addr*⟩		JUMP	⟨*addr*⟩	Jump to ⟨*addr*⟩
43	⟨*addr*⟩	⟨*offset*⟩	JUMP	⟨*addr*⟩,#⟨*offset*⟩	Jump to ⟨*addr*⟩+⟨*offset*⟩

`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 #`$\langle n \rangle$	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 $,$`	$y \times x \to x$.
`52` $\langle v \rangle$	`MUL $,#`$\langle v \rangle$	$x \times \langle v \rangle \to x$.

`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}+=$x \leq y$.
2B	⟨*addr*⟩		NEXT	⟨*addr*⟩,#1,$	Loop to ⟨*addr*⟩ if {0}+=$1 \leq x$.
4B	⟨*addr*⟩	⟨*inc*⟩ ⟨*max*⟩	NEXT	⟨*addr*⟩,#⟨*inc*⟩,#⟨*max*⟩	Loop to ⟨*addr*⟩ if {0}+=⟨*inc*⟩≤⟨*m*
6B	⟨*addr*⟩	⟨*max*⟩	NEXT	⟨*addr*⟩,#1,#⟨*max*⟩	Loop to ⟨*addr*⟩ if {0}+=1≤⟨*max*⟩.
15	⟨*addr*⟩		NEXTDEC	⟨*addr*⟩,$,$	Loop to ⟨*addr*⟩ if {0}−=$x \geq y$.
35	⟨*addr*⟩		NEXTDEC	⟨*addr*⟩,#1,$	Loop to ⟨*addr*⟩ if {0}−=$1 \geq x$.
55	⟨*addr*⟩	⟨*dec*⟩ ⟨*min*⟩	NEXTDEC	⟨*addr*⟩,#⟨*dec*⟩,#⟨*min*⟩	Loop to ⟨*addr*⟩ if {0}−=⟨*dec*⟩≥⟨*m*
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 ⟨n⟩	DROP #⟨n⟩	Drop the top ⟨n⟩ values.
70	POP {0}	Store x into loop counter {0}.
90	POP [$]	Store y into memory address in x.
B0	POP [0]	Store x into memory address 0.
D0 ⟨a⟩	POP [⟨a⟩]	Store x into address ⟨a⟩.
F0	POP {$}	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 ⟨v⟩	PUSH #⟨v⟩	Push value ⟨v⟩.
6F	PUSH {0}	Push loop counter {0}.
8F	PUSH [$]	Push contents of address in x.
AF	PUSH [0]	Push contents of address 0.
CF ⟨a⟩	PUSH [⟨a⟩]	Push contents of address ⟨a⟩.
EF	PUSH {$}	Push loop counter for level x.

RAMP: Fade a Channel Up to Full Brightness

Incrementally increase the dimmer level of channel ⟨*ch*⟩ in increments of ⟨*step*⟩, with a delay of ⟨t⟩/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 ⟨*ch*⟩ ⟨*step*⟩ ⟨t⟩	RAMP #⟨*ch*⟩,#⟨*step*⟩,#⟨t⟩	Ramp ⟨*ch*⟩ by ⟨*step*⟩ per ⟨t⟩.

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 initialze 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 $,$	$y - x \to x$.
54 $\langle v \rangle$	SUB $,#$\langle v \rangle$	$x - \langle v \rangle \to x$.
34	DECR $	$x - 1 \to x$.

`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 ⟨*time*⟩/120 seconds.

Bytes	Instruction	Description
09	WAIT $	Wait *x*/120 s.
49 ⟨*time*⟩	WAIT #⟨*time*⟩	Wait ⟨*time*⟩/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.

6.4 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 sequence, which is far more than the current Lumos boards can actually accommodate.

Each sequence begins with a line of the form: :0200000400*sskk* 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: :*nnaaaa*00⟨*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.

⟨*data*⟩ bytes are the instruction data being recorded at this address.

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

Opcode	Args	Mode	Instruction	Opcode	Args	Mode	Instruction
00		inh	exit				
01		stk	off $	81		stk	on $
02		stk	dimmer $,$				
03	a	stk	jump a,$	83	a	stk	break a,$
04		stk	fade $,$,$	84		stk	ramp $,$,$
05		inh	suspend	85		inh	suspupd
06		inh	resume	86		inh	resfade
07		inh	sleep	87		inh	wake
08		stk	call $,$	88		stk	exec $,$
09		stk	wait $				
0A		stk	loop $				
0B	a	stk	next a,$,$	8B	a	stk	nextinf a,$
0D		inh	blackout	8D		inh	fadeout
0E		inh	nop				
0F		inh	dup	8F		i/s	push [$]
10		inh	drop $	90		i/s	pop [$]
11		stk	exch				
12		stk	mul $,$				
13		stk	add $,$				
14		stk	sub $,$				
15	a	stk	nextdec a,$,$	95	a	stk	nxdcinf a,$
23	a	def	jump a	A3	a	def	break a
28		d/s	call $	A8	i	d/s	exec $
2A		def	loop #0	AA		def	loop #1
2B	a	d/s	next a,#1,$	AB	a	d/s	nextinf a,#1
2F		def	push #0	AF		i/d	push [0]
30		def	drop	B0		i/d	pop [0]
33		def	incr $				
34		def	decr $				
35	a	d/s	nextdec a,#1,$	B5	a	d/s	nxdcinf a,#1
40	n	imm	ssfb #n				
41	c	imm	off #c	C1	c	imm	on #c
42	c v	imm	dimmer #c,#v				
43	a o	imm	jump a,#o	C3	a n	imm	break a,#n
44	c s t	imm	fade #c,#s,#t	C4	c s t	imm	ramp #c,#s,#t
48	i n	imm	call #i,#n	C8	i n	imm	exec #i,#n
49	t	imm	wait #t				
4A	v	imm	loop #v				
4B	a i x	imm	next a,#i,#x	CB	a i	imm	nextinf a,#i
4F	v	imm	push #v	CF	a	ind	push [a]
50	n	imm	drop #n	D0	a	ind	pop [a]
52	x	imm	mul $,#x				
53	x	imm	add $,#x				
54	x	imm	sub $,#x				
55	a i x	imm	nextdec a,#i,#x	D5	a i	imm	nxdcinf a,#i
68	i	i/d	call #i	E8	i	i/d	fexec #i
6B	a x	i/d	next a,#1,#x				
6F		d/l	push {0}	EF		s/l	push {$}
70		d/l	pop {0}	F0		s/l	pop {$}
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	ifnall 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

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

For example, the sequence pair shown on page 35 would be encoded in the hex file as follows:

```
:020000040001F9
:100000004F0A2A4A144F01EF6F480502AF097503E2
: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
```

CHAPTER 7

Communication Protocol Details

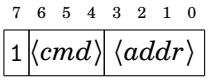
The protocol used by Lumos boards to receive commands is designed to be compact (so as to conserve the number of bytes transmitted to carry out common functions) while remaining simple and fast for the controller to interpret. It is also designed so that multiple devices can share the same RS-485 network 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 original Lumos protocol is now designated as "protocol 0" and an explicit version number is assigned to each new version of the protocol if there is a breaking change introduced by that version.

The current protocol version as of this writing is **protocol 1**.

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, ⟨*cmd*⟩, specify the command being given to the device. The least signifi-

cant four bits, ⟨*addr*⟩, 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	⟨*data*⟩

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 ⟨*addr*⟩ matches my address, interpret command code ⟨*cmd*⟩ 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	⟨*addr*⟩
0	⟨*cmd*⟩	

Note that where a value for ⟨*channel*⟩ 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 ⟨*channel*⟩ value the board does not support, it will flag the command as an error and ignore it.

If a binary value greater than 127 needs to be sent as part of a command packet, the following escape mechanism is used. Two byte values are special:

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

Intended Value	Transmitted Byte(s)
0x42	0x42
0x7D	0x7D
0x7E	0x7F 0x7E
0x7F	0x7F 0x7F
0x80	0x7E 0x00
0x81	0x7E 0x01
0xFD	0x7E 0x7D
0xFE	0x7E 0x7E
0xFF	0x7E 0x7F

Figure 7.1: Examples of Escaped 8-bit Data Values

`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`.

See the examples in Figure 7.1.

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

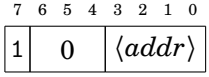
- Binary bits are shown in fixed-width type: `0`, `1`.
- Decimal numbers are shown in normal type: 123.
- Hexadecimal numbers are shown in fixed-width type with a leading `0x`: `0x42`.
- Single-bit flags are shown in Italics as single-letter names: f.
- Multi-bit fields are shown in Italics with angle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

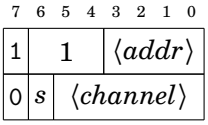
For the QS* units, output dimming is not supported, so setting a channel level of 0–127 is equivalent to turning that channel off, and a level of 128–255 is equivalent to turning it on.

`0x0`: Blackout



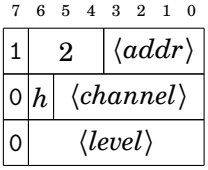
Immediately turns all output channels completely off.

0x1: Channel On/Off



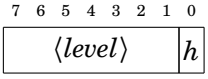
Turns output ⟨*channel*⟩ fully on (*s*=1) or off (*s*=0).

0x2: Set Channel Output Level



Sets the output level of the designated ⟨*channel*⟩ 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 ⟨*level*⟩ field and the *h* bit, with *h* forming the *least significant* bit of the resulting value:

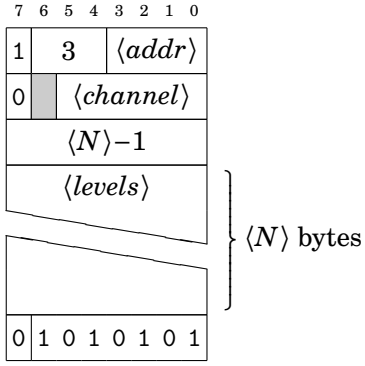


This means is that for the sake of simplicity, you may choose to ignore the *h* bit,[1] and just use a 0–127 value for ⟨*level*⟩, and it will work (although the steps between each level will be twice as large).

(This command was added to the protocol set before the data escaping rules were added, which is why it is structured as it is. This may change in the future.)

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

0x3: Bulk Channel Update

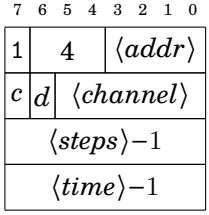


If more than a small number of channel levels are being updated at once, it is more economical to send a single bulk update command with all their new values in one payload, rather than initiate individual commands for them.

This updates ⟨*N*⟩ channels starting with ⟨*channel*⟩ and continuing to ⟨*channel*⟩+⟨*N*⟩−1. Note that the value ⟨*N*⟩−1 is passed, not ⟨*N*⟩. 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. 52 for details about this escape mechanism).

0x4: Ramp Channel Level



Smoothly increases (*d*=1) the brightness of ⟨*channel*⟩ from its current value until it reaches the maximum level, or decreases (*d*=0) it until it is fully off, changing it in increments of ⟨*steps*⟩ each time, with a delay of ⟨*time*⟩/120 seconds between each change. Since the values passed are actually ⟨*steps*⟩−1 and ⟨*time*⟩−1, the effective ranges for each of those values is 1–256.

Once the level has reached the full extent of its range, it remains there. However, if the flag *c*=1, the channel will immediately ramp back the other direction, ramping continuously up and down until another command changes the channel's state.

Note that it may be necessary to escape some of these data bytes so their

⟨*c*⟩	Color Depth	Bytes	Encoding
00	none	0	none
01	3 bits	1	00000*rgb*
10	9 bits	2	0*rrr*0*ggg* 00000*bbb*
11	12 bits	2	*rrrrgggg* 0000*bbbb*

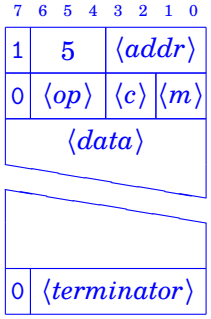
Table 7.1: Color Encoding Values

8-bit values conform to the overall communications protocol as described at the beginning of this chapter (see p. 52 for details about this escape mechanism).

For devices which don't support dimming (e.g. the QS* units), this can be used to make an output flash at a specified rate. Since there is no ability to ramp the level, these devices simply go to full on or off state immediately. However, if the ⟨*c*⟩ flag is set, the channel will cycle between on and off states every ⟨*time*⟩ intervals, which are device specific but should be approximately 1/120 s. (The QSMC, for example, interprets ⟨*time*⟩ in units of 8 ms, which is slightly shorter than 1/120 s. This gives them a flash rate of a state change of 125 per second to one state change per 2.048 seconds, or 62.5 Hz to 0.24 Hz.)

The following describes a feature of the QS controller units which is not part of the core Lumos line.

0x5: Update Scoreboard Display

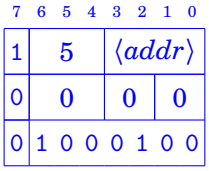


This command updates the quiz show unit's scoreboard display according to the operation code ⟨*op*⟩, as described below. In each case, ⟨*c*⟩ indicates how color information will be sent in the ⟨*data*⟩ block, and ⟨*m*⟩ contains operation-specific mode flag bits. Table 7.1 lists the possible color modes for the ⟨*c*⟩ field. Table 7.2 lists the expected values for ⟨*terminator*⟩ for each operation.

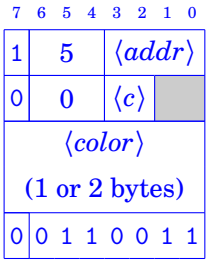
⟨*op*⟩	⟨*terminator*⟩
000	01000100 (if no color given)
000	00110011 (if color given)
001	00101110
010	01110001
011	01110001
100	01110001
101	01110001
110	00011100
111	01101010 (for segmented LED updates)
111	00001011 (for scoreboard updates)

Table 7.2: Command Terminator Bytes by Operation (Scoreboard Updates)

op=0: Clear Clear the entire screen.

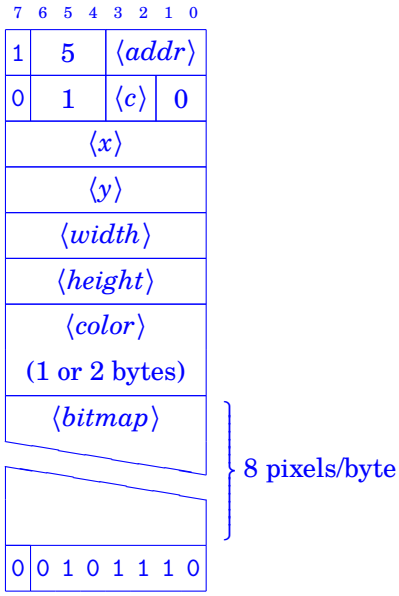


If no color is specified (⟨*c*⟩=0) the screen is cleared to black.

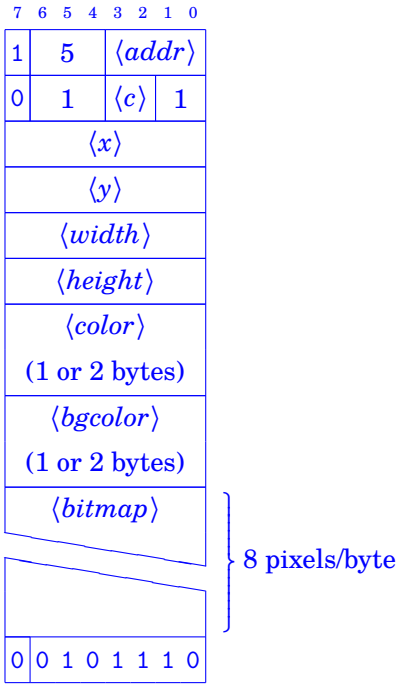


Otherwise ⟨*data*⟩ contains a single color value for the screen to be painted with.

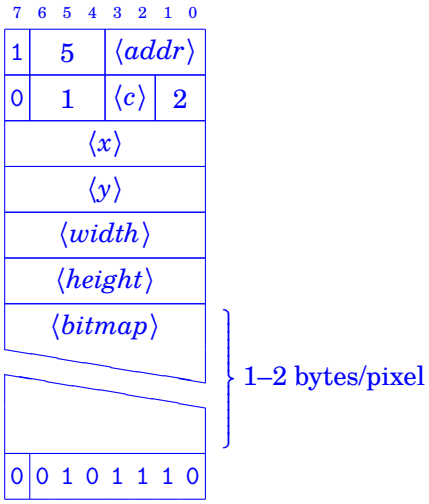
op=1: Bitmap Draws a bitmapped image on the display. In all modes, the first four bytes of ⟨*data*⟩ provide the ⟨*x*⟩ and ⟨*y*⟩ coordinates, ⟨*width*⟩ and ⟨*height*⟩ in pixels, respectively.



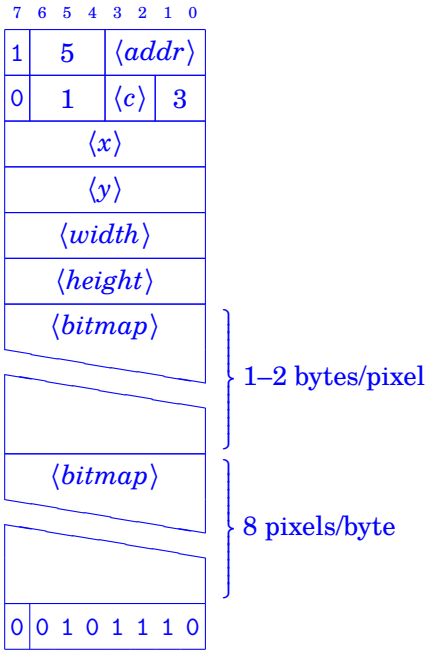
If ⟨*m*⟩=0, the next data byte specifies a color with which to draw the entire monochrome bitmap whose bits are given in the rest of the ⟨*data*⟩ block.



If ⟨*m*⟩=1, two color values are given followed by a monochrome bitmap. The first color is used to draw the bitmap, while the second is the background color to be used.



If ⟨*m*⟩=2, each value after the above-mentioned initial four are full color specifications for each pixel of the color image.



If ⟨*m*⟩=3, a color bitmap is given as with ⟨*m*⟩=2, but the color bitmap data are followed by a monochrome bitmap which gives the image mask to apply while drawing.

op=2: Pixel/Line Draw a line or a single pixel on the matrix display.

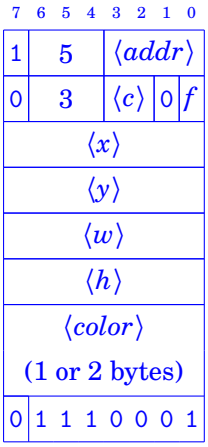
7	6	5	4	3	2	1	0
1	5			⟨*addr*⟩			
0	2			⟨*c*⟩			0
⟨*x*⟩							
⟨*y*⟩							
⟨*color*⟩ (1 or 2 bytes)							
0	1	1	1	0	0	0	1

If ⟨*m*⟩=0, ⟨*data*⟩ contains, respectively, *x* and *y* coordinates and a *color* value. A single pixel will be drawn.

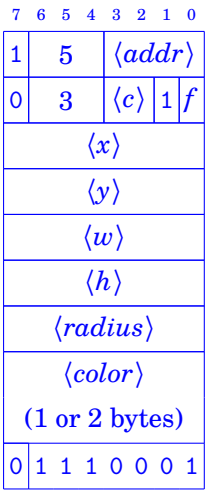
7	6	5	4	3	2	1	0
1	5			⟨*addr*⟩			
0	2			⟨*c*⟩			1
⟨x_0⟩							
⟨y_0⟩							
⟨x_1⟩							
⟨y_1⟩							
⟨*color*⟩ (1 or 2 bytes)							
0	1	1	1	0	0	0	1

Otherwise, ⟨*data*⟩ contains coordinates x_0, y_0, x_1, y_1, and a *color*. A line is drawn between the given points.

op=3: Rectangle Draws a rectangle. The first four bytes of ⟨*data*⟩ provide the *x* and *y* coordinates, *width* and *height* in pixels.



If $\langle f \rangle$=0, a rectangle will be drawn, outlined in $\langle color \rangle$. If $\langle f \rangle$=1, the rectangle will be completely filled in.



In this variant, $\langle data \rangle$ includes a *radius* byte followed by a *color* code. A rectangle will be drawn with rounded corners, outlined or filled in the color.

op=4: Circle Draws a circle on the matrix display.

7	6	5	4	3	2	1	0
1	5			⟨*addr*⟩			
0	4			⟨*c*⟩	0	*f*	
⟨*x*⟩							
⟨*y*⟩							
⟨*radius*⟩							
⟨*color*⟩ (1 or 2 bytes)							
0	1	1	1	0	0	0	1

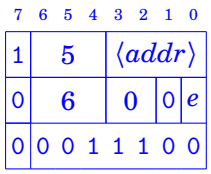
The ⟨*data*⟩ block provides ⟨*x*⟩ and ⟨*y*⟩ coordinates of the center point, the ⟨*radius*⟩, and a ⟨*color*⟩ code. A circle with those parameters is drawn, outlined if ⟨*f*⟩=0 or filled in if ⟨*f*⟩=1.

op=5: Triangle Draws triangles on the matrix display.

7	6	5	4	3	2	1	0
1	5			⟨*addr*⟩			
0	5			⟨*c*⟩	0	*f*	
⟨x_0⟩							
⟨y_0⟩							
⟨x_1⟩							
⟨y_1⟩							
⟨x_2⟩							
⟨y_2⟩							
⟨*color*⟩ (1 or 2 bytes)							
0	1	1	1	0	0	0	1

The ⟨*data*⟩ includes coordinates ⟨x_0⟩, ⟨y_0⟩, ⟨x_1⟩, ⟨y_1⟩, and ⟨x_2⟩, ⟨y_2⟩, follwed by a ⟨*color*⟩ code. A triangle is drawn between these three points, filled in if ⟨*f*⟩=1.

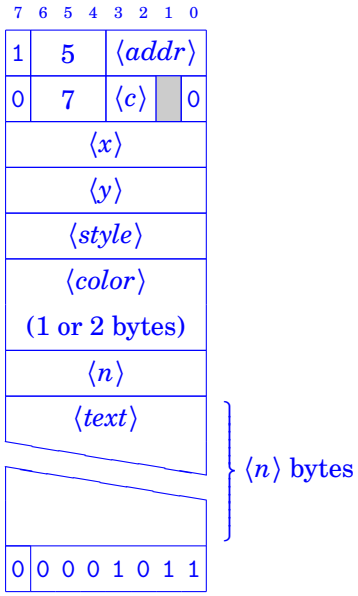
op=6: Inhibit/Enable Updates Controls whether drawing commands immediately change the matrix display.



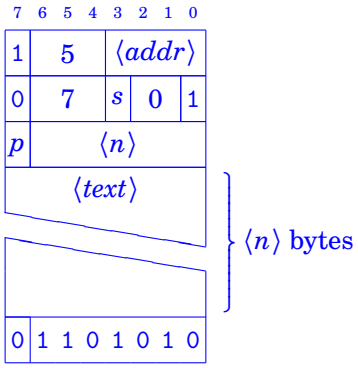
If ⟨*e*⟩=0, stops updating the display for graphics drawing commands. The drawing commands are noted internally but won't be sent to the scoreboard itself until updates are enabled again.

If ⟨*e*⟩=1, updates the display to reflect the internal state tracked by the device. All subsequent graphic updates will change the display in real time.

op=7: Alphanumeric Display This operation writes text data to the score display.

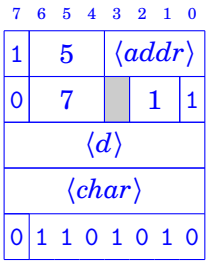


In this case, ⟨*data*⟩ provides the ⟨*x*⟩ and ⟨*y*⟩ coordinates, font style ⟨*style*⟩, a ⟨*color*⟩ code, and a string length *n*. Following this are *n* bytes containing the characters to be written.

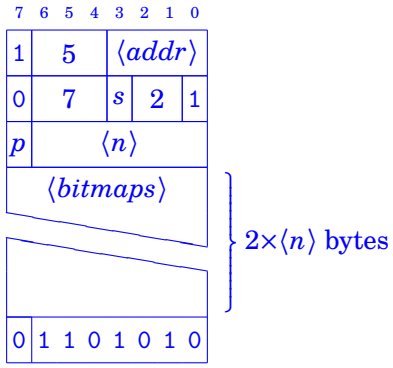


This and the following forms all write to the alphanumeric LED character display in one of several ways. Here, ⟨*data*⟩ holds a length byte n followed by n bytes of character data. These are displayed as ASCII bytes on the alphanumeric digits. The entire display is overwritten with the specified ⟨*text*⟩.

If s=1, the text will appear on the display and then scroll to the left, allowing for messages which are longer than the number of physical characters on the display to be shown. You may want to prepend spaces to the string to initially blank the display and scroll the entire message on from the right, or set p=1, which will automatically pad the message with enough blanks to ensure the message starts off-display before scrolling across.

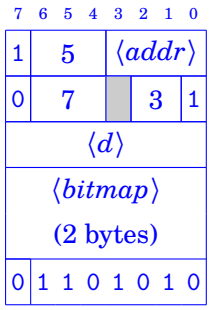


In this variant, ⟨*data*⟩ holds two bytes: ⟨*d*⟩ which gives the digit to be modified, and the ASCII character to written to that digit.



In this version, ⟨*data*⟩ holds a length byte n followed by $2n$ bytes of character data. The character data words are bitmapped to the LED digit segments, allowing completely arbitrary symbols on the displays. The entire display is overwritten by the given ⟨*text*⟩.

If s=1, the text will appear on the display and then scroll to the left, allowing for messages which are longer than the number of physical characters on the display to be shown. You may want to prepend spaces to the string to initially blank the display and scroll the entire message on from the right. Setting p=1 arranges sufficient padding to do this for you automatically.



This one allows a single character to be updated using a raw segment bitmap.

Character Sets

The ASCII characters sent to the matrix displays contain all the usual characters one would expect within the traditional 7-bit printable character range.

Table 7.3 lists the special characters available for the segmented LED displays.

Code	Description
0x80	"Z" with line through middle (Ƶ)
0x81	Digit "1" without serif
0x82	Digit "7" with diagonal downstroke (instead of straight)
0x83	Digit "7" with line through (7̶)
0x84	Digit "9" with tail
0x85	Hourglass (⧖)
0x86	Degree sign (°)
0x87	X in box (⊠)
0x88	Plus-minus (±)
0x89	Minus-plus (∓)
0x8A	Clock
0x8B	All segments on ("splat")
0x8C	Upper splat
0x8D	Lower splat
0x8E	Left splat
0x8F	Right splat
0x90	Up-pointing triangle (△)
0x91	Down-pointing triangle (▽)
0x92	Left-pointing triangle (◁)
0x93	Right-pointing triangle (▷)
0x94	Up arrow (↑)
0x95	Down arrow (↓)
0x96	Right arrow (→)
0x97	Left arrow (←)
0x98	NE arrow (↗)
0x99	NW arrow (↖)
0x9A	SE arrow (↘)
0x9B	SW arrow (↙)
0x9C	Capital Gamma (Γ)
0x9D	Lowercase mu (μ)
0x9E	Captial Xi (Ξ)
0x9F	Capital Pi (Π)
0xA0	Lowercase pi (π)
0xA1	Capital Sigma (Σ)
0xA2	Capital Phi (Φ)
0xA3	Capital Psi (Ψ)

Table 7.3: Non-standard LED Characters

Segmented LED Display Bitmaps

The raw bitmap data sent to arbitrarily light up LED segments is a 15-bit value, with the least significant bit mapped to the a segment, followed in order by b, c, d, e, f, $g1$, $g2$, h, j, k, l, m, n, and the decimal point.

7.2 0x06: Button Scanning

7	6	5	4	3	2	1	0
1	6			⟨*addr*⟩			
0	*a*	*i*	*p*	*s*	*d*	*e*	0
0	⟨*terminator*⟩						

This command controls the quiz show button scanner. This may take two different forms.

7	6	5	4	3	2	1	0
1	6			⟨*addr*⟩			
0	*a*	*i*	*p*	*s*		0	0
0	1	1	0	1	0	1	0

With ⟨*e*⟩=0, this initiates a button scan.

If ⟨*s*⟩=0, the button states are cleared and the button timer (re-)started. Normally, no reply is given and the results of the button scan may be requested when the scan is stopped (see below). However, if ⟨*i*⟩=1, a reply is sent every time any button is pressed. These replies will be simple "ping" replies if ⟨*p*⟩=1 or full status replies otherwise. Additionally, if ⟨*a*⟩=1, periodic "NAK" replies will be sent while waiting for more button presses.

Otherwise, if ⟨*s*⟩=1, this command merely re-enables the button scanner so that it starts looking for lockouts due to early button pushes, but does not perform any of the other operations listed above.

7	6	5	4	3	2	1	0
1	6			⟨*addr*⟩			
0	1	1	*p*	*s*	*d*	1	0
0	1	0	0	1	0	1	0

With ⟨*e*⟩=1, the button scanner's state is queried, and optionally stopped. The state reply will either be a simple "ping" response if ⟨*p*⟩=1, otherwise a full status will be sent.

If ⟨*s*⟩=1, the scanner is stopped. Otherwise it will continue running (if it was running at the time) after this status query is sent. If ⟨*s*⟩=1 and ⟨*d*⟩=1, the scanner is entirely disabled. No button scans will be done, and

no signals will be sent to poll the button hardware. Otherwise, if ⟨*s*⟩=1 and ⟨*d*⟩=0, the button scanning function will continue to poll the buttons, but will do so only in order to lock out buttons pressed prematurely (i.e., before a scan was initiated). (If ⟨*s*⟩=0, the value of ⟨*d*⟩ is irrelevant.)

Unimplemented Single-byte Codes

7	6 5 4	3 2 1 0
1	5	⟨*addr*⟩

7	6 5 4	3 2 1 0
1	6	⟨*addr*⟩

Codes 5 and 6 are not implemented and are reserved for future Lumos features. (Except QS* devices.)

7.3 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	⟨*addr*⟩
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. Not implemented on QS* devices.

0x701: Wake

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
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. Not implemented on QS* devices.

0x702: Shut Down

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
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. Not implemented on QS* devices.

0x703: Query

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
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 shown in Figure 7.2.

All devices using the Lumos protocol are expected to implement this command with the output as documented here; there is a section reserved in the specification for device-specific information which is up to each device

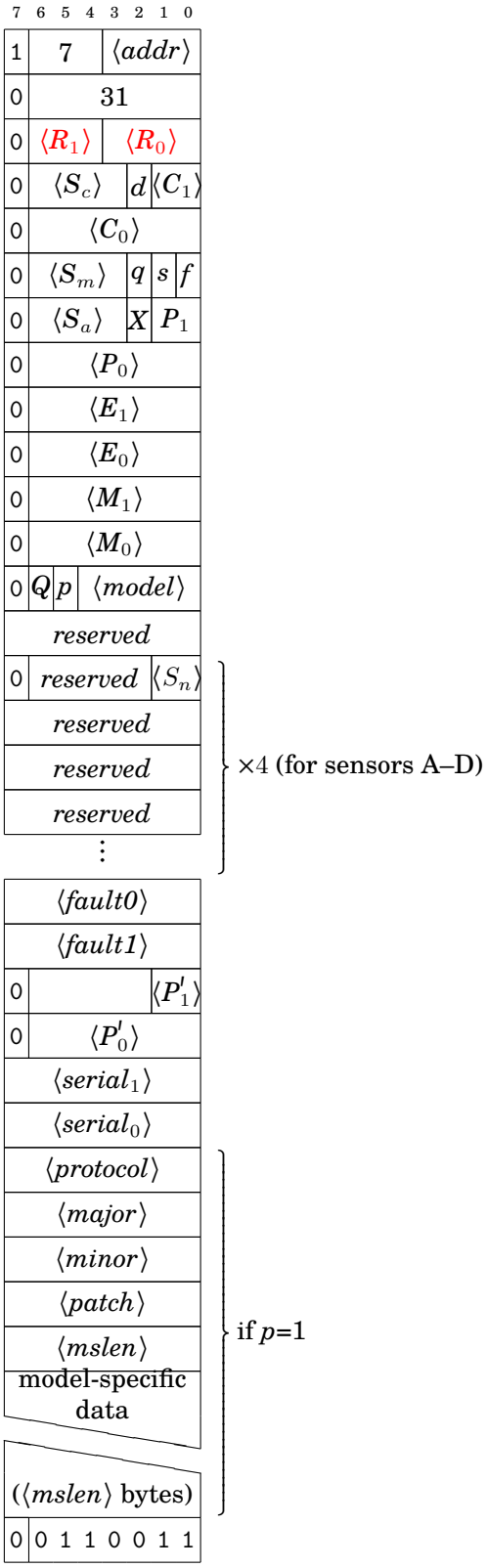


Figure 7.2: 0x71f [reply] Query Response from Lumos Controller

⟨*model*⟩	Device Type
0	48-channel controller
1	24-channel DC controller
2	4-channel DC controller
3	reserved QSCC
4	reserved QSMC
5	reserved QSRB
6	reserved QSXT
7	reserved
8	reserved
9	reserved

Table 7.4: Defined Device Model Numbers

type to define.

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

Version and Protocol Information: For Protocol 0 devices, the ⟨p⟩ flag is clear (0) and the firmware version number is contained in fields ⟨R_1⟩ and ⟨R_0⟩, with ⟨R_1⟩ giving the major revision number in the range 0–7 and ⟨R_0⟩ giving the minor revision number in the range 0–15.

All other devices set the ⟨p⟩ flag (1) to indicate that they are sending the extended form of response to this command. In this case, fields ⟨R_1⟩ and ⟨R_0⟩ should be disregarded. Instead, additional fields appear at the end of the response packet: ⟨*protocol*⟩ specifies the protocol number implemented by the device; ⟨*major*⟩, ⟨*minor*⟩, and ⟨*patch*⟩ give the full semantic version value for the firmware with each component having a range of 0–255; and ⟨*mslen*⟩ gives the length in bytes of any additional model-specific data which immediately follows the ⟨*mslen*⟩ field. Descriptions of the currently-defined model-specific data appear below.

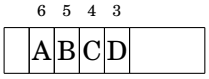
Device identity: ⟨*addr*⟩ holds the reporting device's current address on the network.

The device's serial number is held in the ⟨$serial_1$⟩ (MSB) and ⟨$serial_0$⟩ (LSB) fields. Serial numbers consisting of all 0 or all 1 bits (i.e., 0x0000 or 0xFFFF) are undefined (meaning that no serial number was assigned to this unit, probably because it was built by a hobbyist for their own use). Serial numbers ≥ 42000 are reserved for boards created by the author. Numbers below 42000 are available for others to assign. The device model code is in the ⟨*model*⟩ field. Table 7.4 lists the values that are currently defined for ⟨*model*⟩.

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 ⟨*exec*⟩. The fault code from the last failed command is contained in fields ⟨*fault0*⟩ for CPU0 and ⟨*fault1*⟩ for CPU1 (on multi-CPU devices).

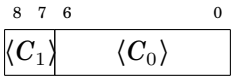
A value of 0 means there was no fault detected since the last query command (i.e., the fault condition is cleared by the query command). Starting with Protocol 1, single-CPU systems place a 16-bit fault code here, with ⟨*fault0*⟩ as the MSB and ⟨*fault1*⟩ as the LSB.

Sensor information: Three values describe the sensor configuration and state: ⟨S_c⟩ indicates the sensor lines configured as inputs (1) or as LED outputs (0); ⟨S_m⟩ shows which sensor inputs are masked out (1) or are being monitored (0); and ⟨S_a⟩ 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:

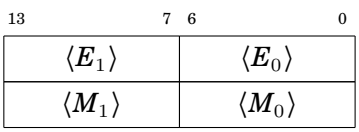


At this time, the only sensor information available is ⟨S_c⟩ and ⟨S_a⟩. The others are anticipated for a future release of the ROM. In the 16-byte block reserved for sensor trigger information, the field ⟨S_n⟩ is still expected to hold the sensor number, which is 0 for the first sensor (A), 1 for sensor B, and so on.

DMX512 configuration: The bit d indicates that the board is in normal Lumos mode (d=0) or DMX512 mode (d=1). If in DMX512 mode, its starting DMX channel is ⟨C⟩+1, which is a value in the range 1–512, sent in two fields:



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 ⟨E⟩ field, while the bytes of available RAM memory for temporary storage is in the ⟨M⟩ field. Each of these values is in the range 0–16,383, sent as two fields:



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
$\langle P_1 \rangle$							$\langle P_0 \rangle$
$\langle P'_1 \rangle$							$\langle P'_0 \rangle$

QSMC Model-Specific Data

7	6	5	4	3	2	1	0	
$\langle digits \rangle$								
$\langle scrollsz \rangle$								
$\langle scrollfree \rangle$								
γ	g	f	e	d	c	b	a	
δ	n	m	l	k	j	i	h	
ϵ	u	t	s	r	q	p	o	
ζ	β	α	z	y	x	w	v	
Lockout time (µs) MSB first (4 bytes)								
$\langle nplayers \rangle$								
0	0	d	c	b	a	l	x	} For each player
0		m	e	$\langle paddr \rangle$				
⋮								

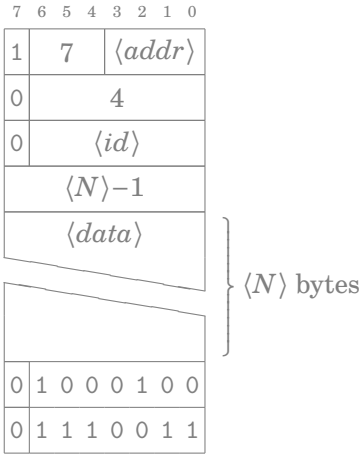
The $\langle digits \rangle$ field specifies the number of alphanumeric LED characters physically present in this device. Which ones are currently reserved for host use are indicated in the following bytes, where a–z and α–ζ represent the individual digit positions. The number of characters which may be buffered at once for scrolling across the display is given by $\langle scrollsz \rangle$, with the number of empty positions still available in that buffer in $\langle scrollfree \rangle$.

The configured lockout time is given (0 if disabled), followed by $\langle nplayers \rangle$ which is the number of player positions supported by the hardware. Then there are $\langle nplayers \rangle$ sets of two bytes giving, the current button masks for

that player, ⟨*paddr*⟩ (the Lumos device address for that player), and the *m* and *e* flags configured for that player from the most recent Player Setup command.

The following describes a feature of the Lumos controller which is planned for a future release but not implemented today. While this description follows the expected behavior the Lumos board will have when that feature is actually available, it is still under development and subject to change.

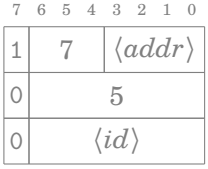
0x704: Define Sequence



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

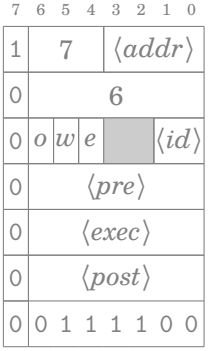
Sequences with ⟨*id*⟩ in the range 0–63 are stored in permanent EEPROM memory and will remain in the controller even after a reset or power cycle. Those with ⟨*id*⟩ in the range 64–127 are stored temporarily in RAM memory and will be lost when the device resets.

0x705: Execute Stored Sequence



Starts executing the stored sequence with the given ⟨*id*⟩. If another sequence was in progress, it is stopped. If ⟨*id*⟩=0, the current sequence is stopped without starting a new one.

0x706: Define Sensor Action

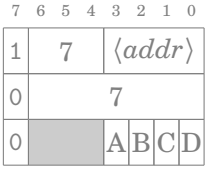


Defines the action to be taken when the sensor $\langle id \rangle$ triggers. ($\langle id \rangle$=0 is sensor A, $\langle id \rangle$=1 is B, $\langle id \rangle$=2 is C, and $\langle id \rangle$=3 is D.) The sensor is triggered on the rising edge if e=1, or the falling edge if e=0.

When that sensor is triggered, any currently executing sequence is stopped. Then the sequence number $\langle pre \rangle$ is executed once. The *exec* 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.[2] When the sensor is no longer triggering, $\langle exec \rangle$ plays on until it completes, then $\langle post \rangle$ is run once.

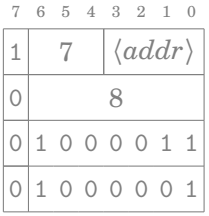
Note that if another sensor triggers or an "Execute Stored Sequence" command is run, it immediately stops the current sequence and all associated sequences. This may cause $\langle post \rangle$ to not execute.

0x707 Mask Sensors



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



[2] The action if both bits are set is undefined.

All stored sequences are erased.

0x709 Forbid Configuration Mode

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	9						

Turns off configuration mode (if the board was in that mode at the time), and prevents the board from going into configuration mode in the future. Once the board is reset or power-cycled, it may again be placed into configuration mode.

This command is intended to replace command 0x774 (see p. 89), since it can be used regardless of whether the Lumos board is already in configuration mode at the time. This makes it a better choice than 0x774 as a general-purpose initialization step to ensure boards stay in normal run mode during a performance. Command 0x774 is therefore deprecated and may disappear in the future.

The following describes a feature of the QS controller units which is not part of the core Lumos line.

0x70a Set Button Lockout Time

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	10						
Lockout time (µs) MSB first (4 bytes)							
0	0	1	1	0	0	1	0
0	1	0	1	1	0	1	0

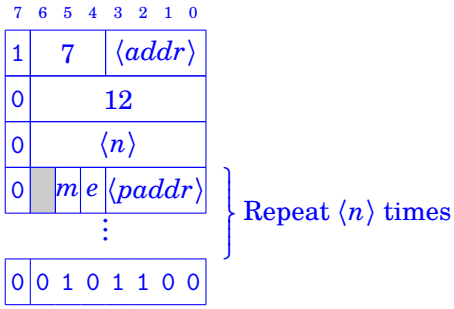
This sets the lock-out duration in microseconds. Any button pressed while the scanner is off will be locked out for this duration of time *after the button is released* before it is eligible for being counted as pressed again. Setting the lockout time to 0 (zero) disables the lockout feature entirely.

0x70b Set Button Masks

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	11						
0	1	0	0	⟨*player*⟩			
0	0	*d*	*c*	*b*	*a*	*l*	*x*
0	1	1	0	0	1	1	0

Sets the button masks for player ⟨*player*⟩. For each button ⟨*a*⟩, ⟨*b*⟩, etc., a 1 means to mask out the button so it is ignored, while a 0 returns the button to normal operation.

0x70c Player Setup



Assigns Lumos device addresses for the first ⟨*n*⟩ player positions. You may, but are not required to, include the player positions directly controlled by the QSMC unit. (You need to include them, however, if you wish to disable their lockout signalling.)

Players starting at ⟨*n*⟩+1 will be given a default behavior of direct signalling for players controlled by the QSMC and no signalling for the others.

If the flag *m*=1, this player is controlled by a QSXT unit plugged into the external bus of the QSMC. As such, this and all following players up to, but not including, player #8 (all of which must also have the *m* bit set) are mapped to player #0–#7 of the QSXT unit.

For example, if your system consists of 16 players using simple button devices, you would plug a QSXT unit as the only player device on the external bus. This means players #0–#7 are controlled by the QSXT and #8–#15 are controlled by the QSMC.

For another example, if you have, say, three QSCC units followed by a QSXT on the external bus, then players #0–#2 are controlled by the three QSCC units, players #3–#7 are controlled by the QSXT (which it will locally consider to be its player #0–#4), and as always the last 8 players will be controlled by the QSMC.

If flag *e*=1, lockout signalling is enabled for this player. The Lumos device with address ⟨*paddr*⟩ will be notified to carry out the signal in case a button for that player is locked out. If *e*=0, no lockout signalling will be sent for that player.

N.B.: This does not control whether button lockouts are enforced. It only affects whether the player stations are informed of those events.

Channel	QSMC	QSXT	QSCC
0	player 8 X red	player 0 X red	X red
1	player 8 X green	player 0 X green	X green
2	player 8 X blue	player 0 X blue	X blue
3	player 8 L light	player 0 L light	L red
4	player 9 X red	player 1 X red	L green
5	player 9 X green	player 1 X green	L yellow
6	player 9 X blue	player 1 X blue	nameplate red
7	player 9 L light	player 1 L light	nameplate green
8	player 10 X red	player 2 X red	nameplate blue
9	player 10 X green	player 2 X green	A light
10	player 10 X blue	player 2 X blue	B light
11	player 10 L light	player 2 L light	C light
12	player 11 X red	player 3 X red	D light
13	player 11 X green	player 3 X green	spare output
14	player 11 X blue	player 3 X blue	
15	player 11 L light	player 3 L light	
16	player 12 X red	player 4 X red	
17	player 12 X green	player 4 X green	
18	player 12 X blue	player 4 X blue	
19	player 12 L light	player 4 L light	
20	player 13 X red	player 5 X red	
21	player 13 X green	player 5 X green	
22	player 13 X blue	player 5 X blue	
23	player 13 L light	player 5 L light	
24	player 14 X red	player 6 X red	
25	player 14 X green	player 6 X green	
26	player 14 X blue	player 6 X blue	
27	player 14 L light	player 6 L light	
28	player 15 X red	player 7 X red	
29	player 15 X green	player 7 X green	
30	player 15 X blue	player 7 X blue	
31	player 15 L light	player 7 L light	
32	spare output 38	spare output 38	
33	spare output 39	spare output 39	
34	spare output 40	spare output 40	

Table 7.5: QS* Device Lumos Channel Assignments

0x70d Reserve Display Space

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	13						
γ	g	f	e	d	c	b	a
δ	n	m	l	k	j	i	h
ϵ	u	t	s	r	q	p	o
ζ	β	α	z	y	x	w	v
0	1	1	1	0	0	0	1

Reserves characters in the QSMC's alphanumeric display for use by the game software. The QSMC firmware will avoid overwriting those character positions as it normally would, so the host computer may write into them using the Lumos commands documented in this section.

Each character is represented here with a single bit, with the leftmost character shown as a and so on to ζ as the rightmost character. If the corresponding bit is 1, that character is reserved for host computer use; if 0, then the QSMC may use it to indicate its own operational status.

Note that regardless of this reservation, the leftmost digit will always be overwritten to show error status.

0x70e Enter Configuration Mode

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	14						
0	1	0	1	1	0	0	1
0	0	1	0	0	1	1	0
0	1	0	1	1	1	0	0

Enables configuration mode, unless a previous command forbade the unit from doing this.

This command appeared starting with Protocol 1.

0x70f Signal Button Lockout

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	15						
0				⟨*player*⟩			
0	0	*d*	*c*	*b*	*a*	*l*	*x*

This command instructs the unit to signal to the player that one or more of their buttons have been locked out due to pressing or holding them before buttons are legally in play.

The ⟨*player*⟩ will be 0 for single-player devices, or the local player number for multi-player devices. The bits *a*–*d*, *l*, and *x* are 1 if that button is locked out or 0 if it is not.

The expected behavior is for one or more of these commands to be received when the lockout status is detected, and eventually another which clears the status when the lockout period has expired.

0x710 Clear Quiz Show Displays

7	6	5	4	3	2	1	0
1	7			⟨*addr*⟩			
0	16						
0				*b*	*m*	*d*	*o*
0	0	1	1	1	1	0	0

This command clears various parts of the quiz show equipment.

If ⟨*b*⟩ is set, the button states are cleared without changing whether the scanner itself is enabled or running.

If ⟨*m*⟩ is set, the readerboard matrix is cleared to black.

If ⟨*d*⟩ is set, the LED display is cleared.

If ⟨*o*⟩ is set, all output channels are turned off.

Reserved Command Codes

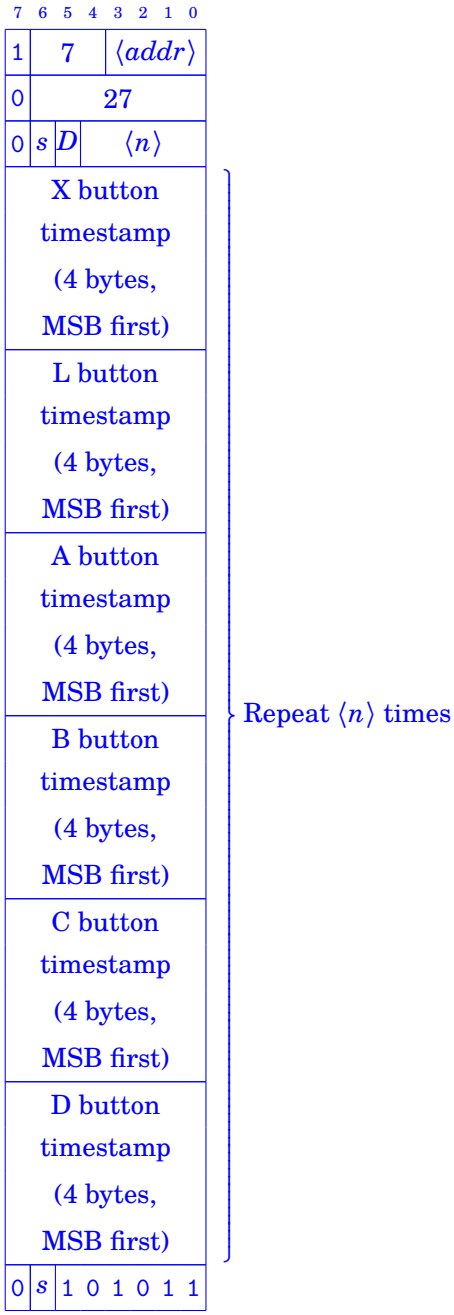
Codes 0x70a–0x70d and 0x70f–0x71d are reserved for future commands. 0x71e–0x71f are currently used for response packets from the Lumos controller. 0x70a–0x70d and 0x70f–0x710 are used for QS-specific commands. 0x71b–0x71d are used for the response packet for QS query commands.

Lumos 48-channel power controllers reserve codes 0x720–0x73F for internal use.

Codes 0x740–0x77F are for configuration-mode commands.

The following describes a feature of the QS controller units which is not part of the core Lumos line.

0x71b [reply] Button Query Full Status Response



Timestamp	Meaning
0x00000000	button not pressed
0x00000001	button masked out
0x00000002	button was locked out at time of scan
0x00000003	reserved
0x00000004	reserved
0x00000005	reserved
0x00000006	reserved
0x00000007	reserved
>0x00000007	button press time in µs

Table 7.6: Special Timestamp Values

This packet is sent upon request to indicate the full current status of all buttons in play. For each button, a 32-bit big-endian value is sent, giving the time in microseconds from the start of the button scan to the time the button was pressed.

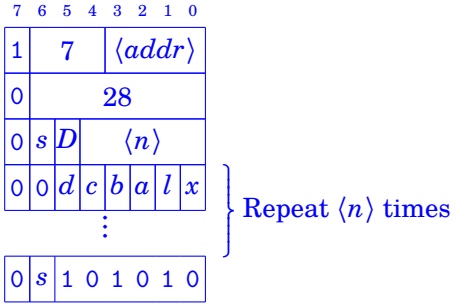
There are a few special values for the timestamp value as listed in Table 7.6.

(If a button was somehow miraculously pressed in less than 8 µs, it will be reported as 8 to distinguish it from the special cases; a button pressed after the maximum time possible to represent in 32 bits—just over 71.5 minutes—it will be given a maximum time value of 0xFFFFFFFF.)

If ⟨*s*⟩=1, the button scanner is still running; otherwise it has stopped and these results represent the final state of the buttons. (The results won't be cleared until the next scan is started, so another query may still be made to get the results again, until that happens.)

If ⟨*D*⟩=1, the button scanner is entirely disabled (so it cannot scan for illegal button presses to enforce the lockout time).

0x71c [reply] Button Query "Ping" Response



This packet is sent by the button controller upon request to show which buttons have been pressed so far. This is a shorter report than the full

button status response (above) which does not indicate which buttons were pressed first or in what order.

If ⟨*s*⟩=1, the button scanner is still running. The button values ⟨*a*⟩, ⟨*b*⟩, etc., are 1 if the button was pressed or 0 if it was not.

If ⟨*D*⟩=1, the button scanner is entirely disabled (so it cannot scan for illegal button presses to enforce the lockout time).

0x71d [reply] Button Status NAK

7	6 5 4 3 2 1 0
1	7 ⟨*addr*⟩
0	29

The host PC may request that these packets be sent while waiting for button presses to avoid a communications timeout.

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 71. 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.

The PC should assume that the full query response is forthcoming and should continue waiting for it. No additional poll is required. Any number of NAK packets may be sent by the Lumos controller between the PC's query request and the Lumos board's full query response packet. [Currently (ROM version 3.1) the Lumos board never sends NAK packets, but may do so in the future.]

7.4 0x720–0x73f: Reserved

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

7.5 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				⟨*addr*⟩		
0	4						⟨P_1⟩
0	⟨P_0⟩						
0	1	0	1	0	0	0	0
0	1	0	0	1	1	1	1

Sets the controller's phase offset to ⟨P⟩, which is a 9-bit value in the range 0–511:

8	7	6						0
⟨P_1⟩	⟨P_0⟩							

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.** Not implemented on QS* devices.

0x76x [config] Set Device Address

7	6	5	4	3	2	1	0
1	7				⟨*addr*⟩		
0	6						⟨*addr*′⟩
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 ⟨*addr*⟩ to ⟨*addr*′⟩. This is effective immediately, so the very next command must be addressed to ⟨*addr*′⟩ 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				⟨*addr*⟩		
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						⟨*addr*⟩
0	113						
0	*A*	*B*	*C*	*D*	*d*	⟨C_1⟩	
0	⟨C_0⟩						
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 ⟨*C*⟩+1, which is a 9-bit value in the range 1–512:

8	7	6							0
⟨C_1⟩	⟨C_0⟩								

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						⟨*addr*⟩
0	114						
0	⟨*speed*⟩						
0	0	1	0	0	1	1	0

Sets the Lumos controller's baud rate as indicated by ⟨*speed*⟩, according to the following table:

⟨***speed***⟩	**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
11	500,000
12	1,000,000
13	2,000,000
14	2,500,000
15	5,000,000
16	10,000,000

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			⟨*addr*⟩			
0	115						
0	0	1	0	0	1	0	0
0	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 speed 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 ⟨*addr*⟩
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,** but see command 0x709 on page 77.

This command is deprecated in favor of command 0x709, which is generally preferred since it can be used in either run mode or configuration mode. This command is only implemented on Protocol 0 devices.

0x775 [config] Update Firmware Image

The following describes a feature of the core Lumos line which is *not* implemented in the QS units.

7	6 5 4 3 2 1 0
1	7 ⟨*addr*⟩
0	117
0	0 1 1 0 0 1 1
0	1 0 0 1 1 0 0
0	0 0 1 1 1 0 0

Initiates firmware update mode on the Lumos controller. From this point forward, the controller expects to receive the new firmware image using a special protocol. If the board is reset during this process, it remains in firmware update mode, since it may have an unusable firmware image at this point.

See page 116 for full documentation about the `lumosupgrade` command used to perform this operation.

The protocol used between the `lumosupgrade` program and the Lumos board is documented in section 7.6 below, starting on page 90.

This command is only recognized if the controller is in configuration mode.

Reserved Command Codes

Codes 0x776–0x77f are reserved for future use as configuration-mode commands.

7.6 Firmware Update Protocol

The following describes a feature of the core Lumos line which is *not* implemented in the QS units.

Once the Lumos board has been placed in flash program mode via the `0x775` "Update Firmware Image" command (see above), all other normal operation of the Lumos board is suspended, including any of its command protocols (Lumos native or DMX). In this mode, the board communicates using a simple bidirectional serial protocol at a fixed speed of 9600 baud, 8 bits, no parity. This protocol is designed solely for the purpose of uploading a new firmware image into the flash memory.

All communication is 7-bit-clean ASCII which won't be confused as Lumos commands by any other listening units, but since the firmware update happens at 9600 baud, we recommend disconnecting other units from the network that aren't configured to run at that speed normally.

Encoded Byte Values

Most of the values sent using this protocol are encoded in a modified form of hexadecimal—rather than using the characters `0123456789ABCDEF` as the base-16 digits, the character set `@ABCDEFGHIJKLMNO` is used instead. The most significant nybble is sent first, as would normally be the case with hexadecimal representation of values.

This encoding makes it easier to encode and decode values, since the nybbles are simply sent with the constant upper nybble value set to `0100`:

7	6	5	4	3	2	1	0
0	1	0	0	⟨*value*⟩			

For example, the value `0x00` is encoded as `@@`, `0x12` as `AB`, and `0xFF` as `OO`.

Address Encoding

The firmware image is transmitted to the Lumos board in 64-byte blocks. Each block must be evenly aligned on a 64-byte address boundary, so the least significant six bits of the address will always be zeroes. Because of this, the least significant nybble is never transmitted and is implied to be zero.

The 64-byte blocks are specified in the protocol by their "block ID" which is simply the more significant 16 bits of the block's address in program memory. Therefore, the block at memory address `0x12580`–`0x125BF` would be known here as block ID `0x1258`.

When sent using this protocol, the nybbles of the block ID are sent from most significant nybble to least, encoded as described above. For example, block `0x1258` would be transmitted as `ABEH`.

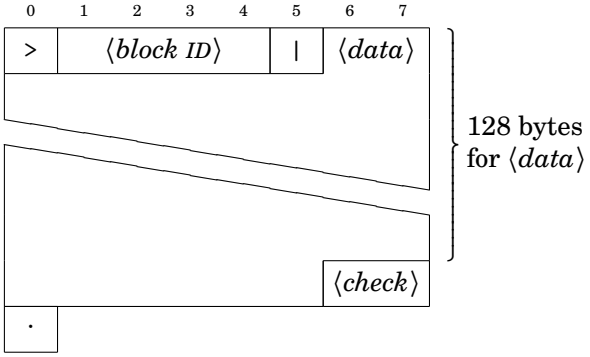
Valid block ID values may range from `0x0000`–`0x16FC`.

Command Protocol

Two commands are recognized in this mode: "query" and "data."

The "query" command is sent by the PC as a single character, 'Q'. This causes the Lumos board to respond with a result packet with status code '*' (see below). If the board was just reset and is ready to begin the process of receiving a new image, it will respond to the query with "OOOO@@@@*".

The "data" command sends a 64-byte block of firmware code to the Lumos board. The board will then burn that block into its flash memory and respond with a result packet as described in the next section. The data command packet is always 137 bytes and has the form:



The '>', '|', and '.' bytes are literal ASCII characters. The other fields are described individually below:

⟨***block ID***⟩ identifies the block being sent, encoded as described above. The blocks may be sent in any order, except for the requirement that block 0x0000 (encoded as ID @@@@) *must be the very last block to be sent*. Overwriting this block changes the boot vector for the microcontroller which then enables the device to execute the new firmware image the next time it is reset, so it is important that this be the last block written.

The act of successfully sending and burning block ID zero ends flash program mode. Once that block is written, the device will automatically reset and resume normal Lumos controller board operation.

⟨***data***⟩ is the block of 64 bytes starting at the address implied by the block ID. Since it's encoded as described above, it is transmitted as 128 ASCII characters.

⟨***checksum***⟩ is a one-byte checksum, encoded as two ASCII characters. The checksum is a running 8-bit total of the two bytes of ⟨*block ID*⟩ and each byte of ⟨*data*⟩. The two's complement of this total is encoded and sent in this field of the command. Only the least significant 8 bits of the total are used for any part of this calculation. For example, if the

⟨*st*⟩	⟨*x*⟩	⟨*y*⟩	**Description**
!	—	—	Invalid block ID given
*	—	—	Response to query command; board ready
b	—	—	Burn error writing to flash memory
k	—	—	Data received and burned successfully
n	0x00	0x00	Data packet format error
n	x	0x00	Checksum error, calculated to be x
n	x	0x40	Illegal character x received
n	0xFF	0xFF	Unrecognized command packet
v	x	y	Verification error: value y read x bytes from end

Figure 7.3: Firmware Update Protocol Response Codes

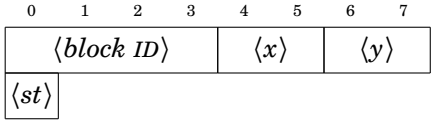
8-bit total of the block ID and data bytes comes out to 0x47, then we calculate its two's complement by inverting and adding one:

0x47	=	01000111
invert	→	10111000
+1	→	10111001

The checksum sent in this example would be 0xB9, encoded into ASCII as "KI".

Response Codes

The response to the query or data commands is always a nine-byte packet of the form:



where:

⟨***block ID***⟩ is the last-known block ID processed by the Lumos board. If no block has been processed yet, this value will be 0xFFFF (encoded as "0000") which is never a valid block ID.

⟨***x***⟩ **and** ⟨***y***⟩ are extra data bytes (encoded as two ASCII characters each) which provide additional information relevant to the particular ⟨*status*⟩ code being reported.

⟨***st***⟩ is a code indicating the result of the command. The possible status results are summarized in Figure 7.3.

For example, after successfully burning block 0x045C, the Lumos board will send the result "@DEL@@@@k".

CHAPTER 8

DMX512 Command Structure

Support for DMX512 is included in the Lumos firmware but has not been tested to be reliable yet, and in fact may not function at all. The information that follows is experimental. If you have DMX512 equipment, expertise, and willingness to help develop this feature of Lumos further, contact the author.

When configured in DMX512 mode, none of the Lumos commands documented in Chapter 7 are 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.

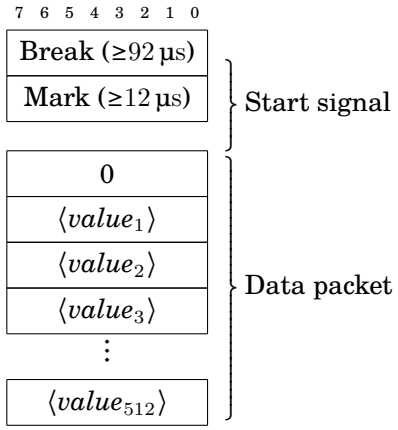


Figure 8.1: DMX Packet

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-forth 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-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.

CHAPTER 9

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.

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 intended application of these controllers—incandescent lamps for the AC boards and LEDs for the DC boards—this produces the 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] This is similar to many household

[1] The 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.

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

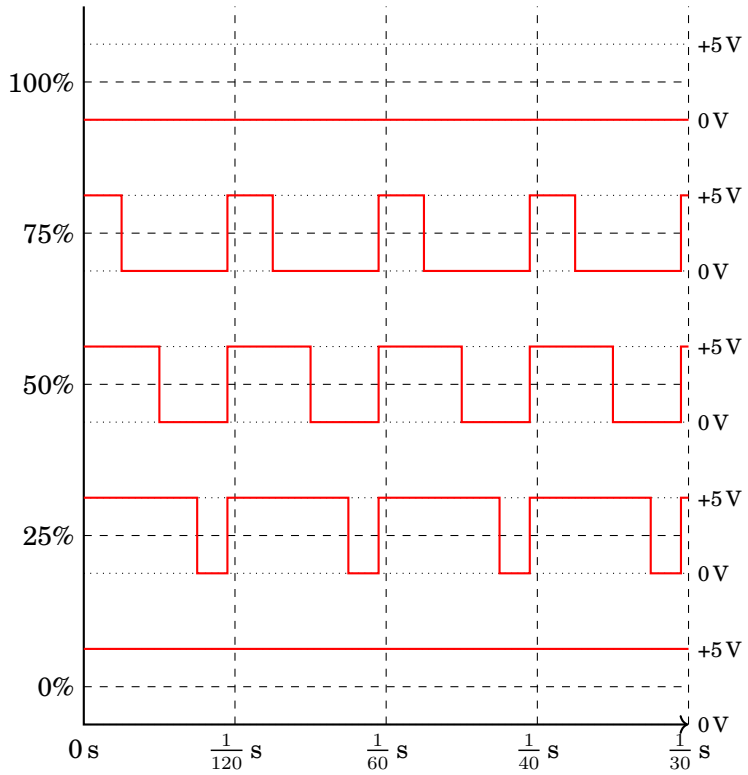


Figure 9.1: Duty Cycles of Channel Logic Drive Outputs

dimmers, so as a general rule of thumb, a light (such as a CFL) marked as "non-dimmable" should not be dimmed by a Lumos board. Damage to the Lumos board and the lamp may result.

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

trusting 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 even that a firmware bug on the controller won't cause this. It's an informed risk you can decide whether to take, but we don't officially recommend it.

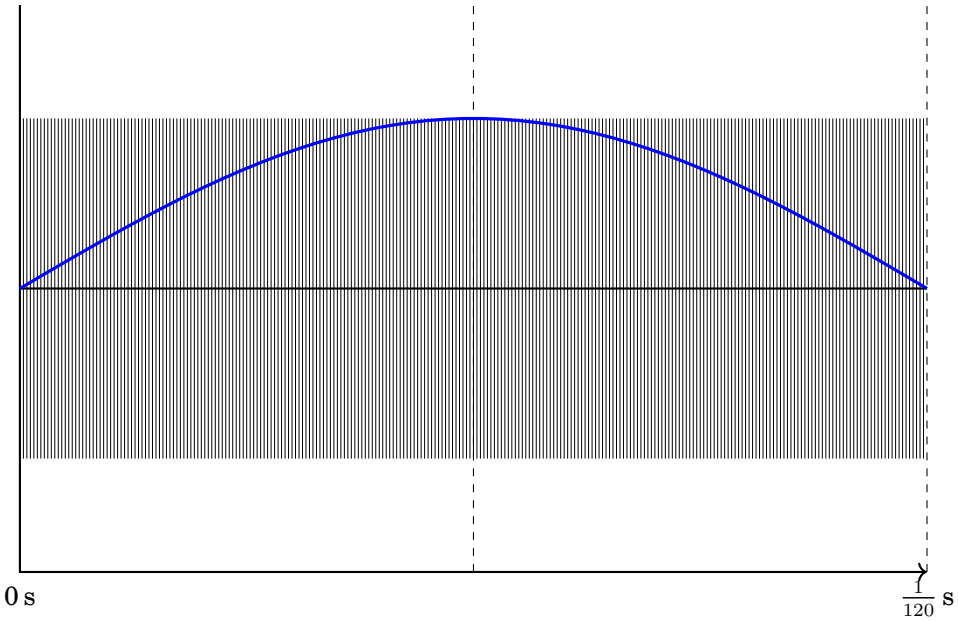


Figure 9.2: Half-AC Cycle Divided into 260 Slices

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.

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 consitent with the AC controllers, but this doesn't need to be synchronized with anything external.

9.1 Phase Offset

There is an obscure device setting called "phase offset" on the Lumos controllers which adds a 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.)

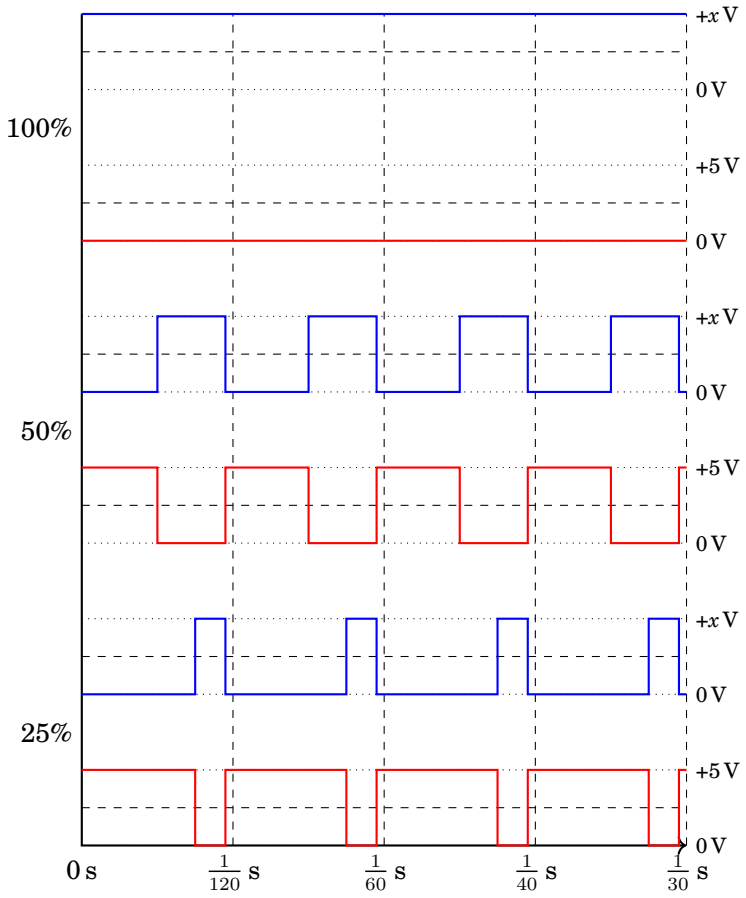


Figure 9.3: Duty Cycles of Logic (red) and DC SSR Outputs (blue)

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

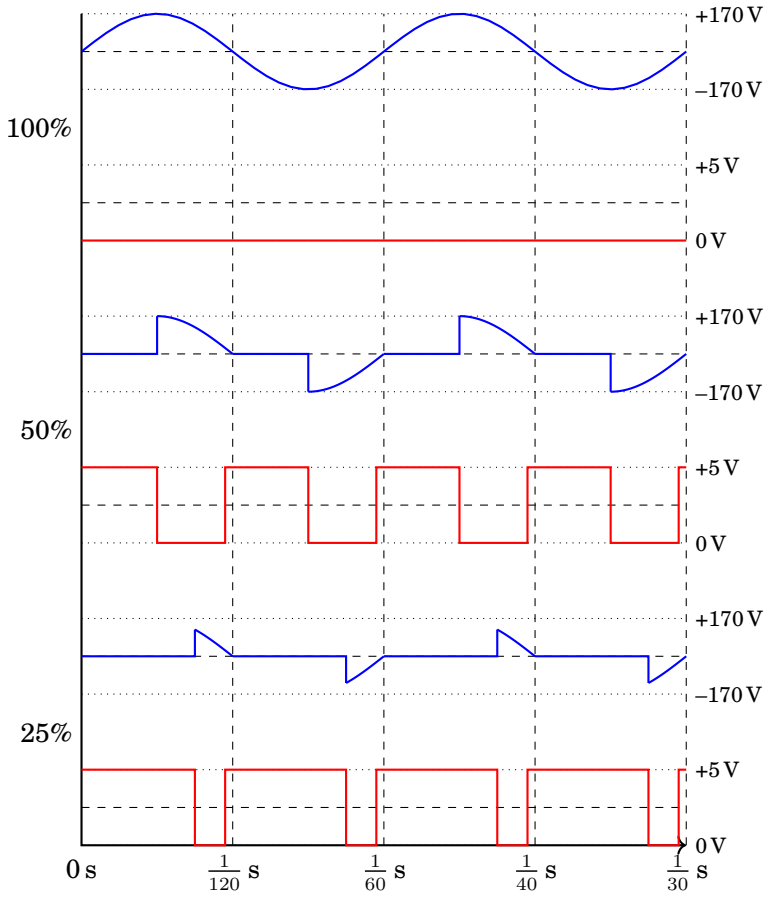


Figure 9.4: Duty Cycles of Logic (red) and AC SSR Outputs (blue)

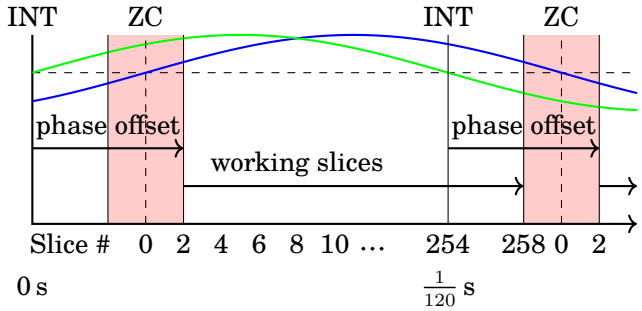


Figure 9.5: Cycle Timing with Phase Offset

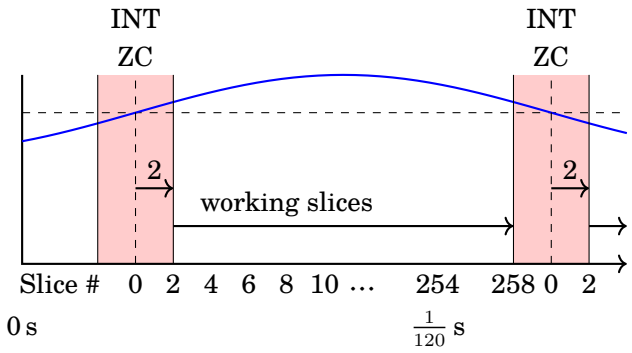


Figure 9.6: Cycle Timing with No Phase Difference

output channels are turned on in order for each of them to generate the desired PWM duty cycle.

Assuming that the interrupt arrives at the actual zero-crossing event, as it ideally should, 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 thay may creep into the cycle as well as ensuring the TRIACs settle as already described. This is shown in Figure 9.6.

9.2 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. 128). 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 flourescent 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 circuit, which is outside the design scope of the Lumos board. Such a modification requires qualified engineering design and should not be attempted by the end user.

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 A.1. 4-channel controllers use the same pattern as the 24-channel controllers.

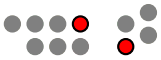
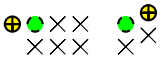
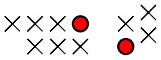
48-ch	24-ch	Description of Condition/Fault Indicated
		[boot] Boot process not yet started
		[boot] EEPROM setup stage
		[boot] EEPROM write operation
		[boot] EEPROM read operation / system initialization
		[boot] system initialization
		[boot] system initialized but main loop/timing system non-functional
		[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 A.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 A.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 flashing
- × not involved or affected; may have any value

Figure A.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/$\overline{\text{R}}$ 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 A.4: Error Condition Codes Reported Via Query Command

Lumos CLI Command Manual Entries

This chapter provides the documentation for the `lumosasm`, `lumosctl`, and `lumosupgrade` 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 – Assemble stored sequences for download to a Lumos board

The `lumosasm` program is used to assemble descriptions of stored sequences into the internal binary format recognized by the Lumos boards. Since that feature is not yet supported by Lumos boards, this section of the manual is blank until that feature is fully defined.

NAME

lumosctl – Manual control for Lumos SSR controller hardware

SYNOPSIS

`lumosctl` [`-dFhkPRrSvwXz`] [`-a` ⟨*addr*⟩] [`-A` ⟨*addr*⟩] [`-b` ⟨*speed*⟩] [`-B` ⟨*speed*⟩] [`-c` ⟨*file*⟩] [`-C` ⟨*file*⟩] [`-D` ⟨*sens*⟩] [`-E` ⟨*sens*⟩] [`-H` ⟨*hexfile*⟩] [`-L` ⟨*level*⟩] [`-m` ⟨*mS*⟩] [`-p` ⟨*port*⟩] [`-P` ⟨*phase*⟩] [`-s` ⟨*file*⟩] [`-T` ⟨*mode*⟩] [`-t` ⟨*s*⟩[`orw+`]:⟨*init*⟩:⟨*seq*⟩:⟨*term*⟩] [`-x` ⟨*duplex*⟩] ⟨*channel-outputs*⟩...

Where ⟨*channel-outputs*⟩ may be any combination of:

⟨*channel*⟩[`@`⟨*level*⟩[`,`...]]
⟨*channel*⟩`d`[⟨*:steps*⟩[⟨*:time*⟩]]
⟨*channel*⟩`u`[⟨*:steps*⟩[⟨*:time*⟩]]
`x`⟨*id*⟩
`p`⟨*time*⟩

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).

A pause in the execution of the arguments may be effected by adding an argument of the form `p`⟨*t*⟩`[s[ec[ond[s]]]]` which makes `lumosctl` pause for ⟨*t*⟩ seconds before continuing on to the next argument. The ⟨*t*⟩ 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 configuration mode
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 pa-

rameter, 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. **[This is a future feature, currently not available on Lumos boards.]**

`--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. **[This is a future feature, currently not available on Lumos boards.]**

`--drop-configuration-mode`
: (`-d`) If the Lumos device is in configuration command mode (for configuraiton of the device), this will cancel that mode. Further configuration commands will not be recognized on that device.

`--dump-configuration-file=`⟨*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`. **[This is a future feature, currently not available on Lumos boards.]**

`--factory-reset`
: (`-F`) Resets the board to its initial default settings, as it would have arrived "out of the box" as it were (of course this is a DIY project, so there's no actual "factory" but if there were one, these are the defaults the board would come shipped with). This can also be accomplished by inserting a jumper on the board in the correct sequence. See the Lumos controller user's manual for details.

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

`--kill-all`
: (`-k`) Turn off all output channels at once.

`--load-compiled-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. **[This is a future feature, currently not available on Lumos boards.]**

`--load-configuration-file=`⟨*file*⟩
: (`-c` ⟨*file*⟩) Load the device configuration from the named ⟨*file*⟩ and program that into the device.

`--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. **[This is a future feature, currently not available on Lumos boards.]**

`--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.

`--read-only`
: (`-r`) Do not query the Lumos board's status. Normally, `lumosctl` reads back the board status at the start and after each configuration change to ensure that the changes were successful. If you are using the board under conditions where getting data from the Lumos board is not possible (e.g., if your RS-485 adapter doesn't support a return channel), use the `-r` option to suppress this part of `lumosctl`'s behavior. This means that configuration requests are sent "blindly" to the Lumos board without any way to confirm that they took effect. (The name of this option seems backwards, but it was named from the Lumos board's point of view—that is, it sees its data connection as a read-only source of data and won't try to send any data back to the PC.)

`--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.

[This is a future feature, currently not available on Lumos boards.]

`--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.]

`--txdelay=`⟨mS⟩
: (`-m` ⟨mS⟩) Delay ⟨mS⟩ milliseconds between each transition from transmitting to receiving mode and vice versa. Usually only needed for half-duplex networks.

`--txlevel=`⟨*level*⟩
: (`-L` ⟨*level*⟩) Transmit mode is controlled by either the DTR or RTS lines. This option controls what logic level on that line means to engage the transmitter. The value of ⟨*level*⟩ may be either "`0`" to mean a logic low (off) indicates transmit mode, or "`1`" to mean a logic high (on) is used.

`--txmode=`⟨*line*⟩
: (`-T` ⟨*line*⟩) Specifies which serial control line is used to control the RS-485 transmitter. The value for ⟨*line*⟩ may be either "`dtr`" or "`rts`".

`--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 configration data are contained in a stanza called "`[lumos_device_settings]`". The values are detailed individually below. **Note that some of these describe anticipated future features of the Lumos hardware that are not available at this time.** These future features are recognized by `lumosctl` as documented here, but won't actually have any effect until they are fully implemented in the Lumos firmware.

`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) aciton 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 still being designed and will be documented here when it is implemented.

AUTHOR

support@madscience.zone

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”).
- Lumos 4-channel DC controller version 1.0 (boards with ID markings beginning with “LUMOS-4SSR-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

Sometimes `lumosctl` doesn't correctly track board configuration changes and incorrectly reports that the Lumos board's configuration was "not as expected" even though the operation was successful. If this happens, try running `lumosctl -R` to get a fresh report of the board's status and verify that it is configured as desired.

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@madscience.zone`.

NAME

lumosupgrade – Download new firmware image into Lumos controller hardware

SYNOPSIS

`lumosupgrade` [`-fhNnRv`] `-a` ⟨*addr*⟩ [`-b` ⟨*speed*⟩] [`-L` ⟨*txlvl*⟩] [`-m` ⟨*txdelay*⟩] [`-p` ⟨*port*⟩] [`-T` ⟨*mode*⟩] ⟨*image-file*⟩

DESCRIPTION

This command places the target Lumos controller device into "flash program mode" and downloads a new firmware image onto it. This is used in order to upgrade the Lumos firmware to a new version.

To upgrade your controller's firmware, perform the following steps:

1. Connect your Lumos controller board to the host PC (all by itself, not sharing a serial connection with other devices). The connection must be bidirectional as the Lumos board needs to be able to acknowledge receipt of the new image. (This may require setting your PC's RS-485 interface appropriately so it may send and receive data. Pay attention to whether your Lumos board is full- or half-duplex.)

2. Place the Lumos board into configuration mode (previously called "privileged mode").

3. Invoke the `lumosupgrade` program to load a new firmware image onto it, as explained in the remainder of this manual page.

Once started, this process must run to successful completion before the Lumos board may be used for normal operations again. If anything goes amiss, the Lumos board may be reset and the `lumosupgrade` program restarted using the `--resume` option. (Upon reset, the Lumos board will stay in flash program mode until a new image has been loaded into it. Should the board reset/reboot for any reason during the process, the `lumosupgrade` process must be started over to ensure a complete image is loaded.)

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. Note that downloading a new firmware image must be done when the target unit is the only device plugged in to the computer. Once the download operation is underway, the low-level protocol used to transmit the image to the device is not necessarily compatible with other units. This option is required because the command to place the device into flash program mode must be addressed to the unit.

`--dry-run`
: (`-n`) Do everything except actually burn the new firmware into the unit. This checks that the ⟨*image-file*⟩ is reasonably sane-looking (not a thorough check of correctness), and communicates with the Lumos controller up to the point where it would put it into flash program mode.

`--force`
: (`-f`) Force upgrade of the board without asking the user for confirmation.

`--help`
: (`-h`) Print a summary of these options and exit.

`--null-device`
: (`-N`) Don't actually communicate with the serial port, but still carry out the other actions including sanity checks on ⟨*image-file*⟩. Implies `--dry-run`.

`--port=`⟨*dev*⟩
: (`-p` ⟨*dev*⟩) Specifies the I/O port the Lumos device is connected to. This may be a simple integer 0, 1, 2, etc. to refer to the first, second, third, etc, standard serial port on the system, or a device name appropriate to the system such as `COM1`, `ttys1`, or `/dev/ttys1`.

`--resume`
: (`-R`) Indicates that the Lumos board was reset prematurely while attempting an upgrade. Since the Lumos board will still be in flash programming mode (and therefore won't be in a position to recognize the Lumos-protocol command to begin a flash programming operation), this option tells `lumosupgrade` to simply start downloading the image onto it, and to not try to put it into programming mode first.

`--speed=`⟨*rate*⟩
: (`-b` ⟨*rate*⟩) Set the serial port to the given baud ⟨*rate*⟩. This is the speed the Lumos board is already configured to use, and will be used for the initial command to enter flash programming mode. Once in flash programming mode, however, a fixed speed of 9600 baud will be used. [Default is 19200 baud.]

`--txdelay=`⟨*t*⟩
: (`-m` ⟨*ms*⟩) Delay ⟨*ms*⟩ milliseconds after changing the transmitter control line for half-duplex networks.

`--txlevel={0|1}`
: (`-L {0|1}`) Specifies the logic level used to signal transmit mode for half-duplex networks. A `1` indicates that the DTR or RTS line (as selected by the `--txmode` option) is asserted to transmit, while a `0` means the line is deasserted to transmit.

`--txmode={dtr|rts}`
: (`-T {dtr|rts}`) Specifies which I/O line is used to signal transmit mode on half-duplex networks.

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

FILE FORMAT

The firmware ⟨*image-file*⟩ is expected to be in standard Intel Hex format. Attempts to change memory addresses outside the supported range will be ignored, including configuration fuses and EEPROM area. Actually, only a reasonable subset of the Intel Hex format is supported; specifically, record types `00` (data record), `01` (end of file), and `04` (extended address) are recognized.

AUTHOR

Steve Willoughby, support@madscience.zone

COMPATIBILITY

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

- Lumos 24-channel DC controller version 1.0 (boards with ID markings beginning with “LUMOS-24SSR-DC-1.0”).
- Lumos 4-channel DC controller version 1.0 (boards with ID markings beginning with “LUMOS-4SSR-DC-1.0”).

The 48-channel controllers are not compatible with this program. These boards must be reprogrammed using a microcontroller programmer.

HISTORY

This program first appeared to support Lumos ROM version 3.0.

SEE ALSO

`lumosctl`(1).

Troubleshooting

While we anticipate the Lumos board will provide many hours of worry-free operation, as with any device (particularly one built as a DIY project), sometimes things don't go quite as planned. Here are a few common problems and their solutions.

Symptom	Likely Cause(s)	Solution
An entire block of outputs does not turn on	No power to the block	If the BLOCK PWR light is off, check the fuse for that block, the connection from the power supply to the block, and that the power supply is powered on.
	Power supply not told to wake up (ATX-style supplies only).	Check that the power supply's green wire is attached to the $\overline{\text{PWR CTL}}$ terminal.
		If the LEDs on the board show that it is in sleep mode, send it a "wake" command from the PC or just command it to turn on any output channel.
Some outputs don't work, or are erratic.	Loose chip.	Lightly press chips back into their sockets.
	Bad solder connection or loose chip.	Re-check all solder connections on the board, re-solder any which are cold, broken, or incomplete.
No units in serial network respond to commands.	Missing terminator	Replace terminators on both ends of the daisy chain (note the PC's RS485 converter may include a built-in terminator for that position).
One unit does not respond to commands.	Wrong address.	Use the `lumosctl` program to reconfigure the board to have the correct address.
	Blown communication fuse.	Replace fuse F3 (24-channel boards).

If all else fails, try performing a factory reset as described in section 4.4 (page 15).

Glossary

Address: The ID number assigned to a particular device to uniquely distinguish it from the other devices plugged into the same serial connection. The commands sent to Lumos boards all contain a "target address" which identifies the particular Lumos board which is to act on that command. All the other Lumos boards will ignore it.

Baud Rate: The speed, in bits per second, of data that is transferred over a serial cable. The term "baud" refers to the number of electrical state changes per second made to perform the signalling. For the kind of signalling done by Lumos boards, the baud rate is the same as the bit-per-second rate, although for other applications such as high-speed modems, it would be more correct to refer to the bit-per-second (BPS) rate only. Depending on the exact protocol settings used, one character takes approximately 10 bits to send, so a data rate of 9600 baud would send about 960 characters (bytes) per second.

Active Low: A logic signal which is considered "on" when the signal is "low" (binary 0 or 0 V), and "off" when the signal is "high" (binary 1 or +5 V). Lumos relay circuits are triggered with active-low signals.

CLI Command-line interface. A program launched on the command-line, or "shell," interface of the computer—the `cmd` window on Microsoft Windows, or the shell in a Mac OSX or Unix/Linux terminal window or `xterm`. Typically CLI tools interact with the user via keyboard, often using a combination of "options" or "switches" to control the program's behavior rather than using a mouse or other graphical interfaces. Generally, CLI tools are easier to automate (to have the computer run them autonomously on a schedule or as needed) since most of them are designed to be run unattended, specifying all of the parameters needed right on the command line.

DIY: "Do-It-Yourself."

Duplex: a feature of a serial line. On a full-duplex connection, separate data wires are present to carry data in both directions, so one device can send and receive data at the same time. On a half-duplex connection, only a single set of data wires is present, so devices must take turns transmitting over them.

Factory reset: The process of resetting all user-configurable options on a device to their original state, as the device presumably was shipped "in the box" from its factory. Lumos boards can be reset in this manner as described on page 15.

Jumper Block: A series of pins mounted to the PCB. Different options are configured for the circuit by placing a jumper over certain pairs of pins, shorting them together.

LED (Light Emitting Diode): A special kind of diode which emits light when current passes from its anode to its cathode.

MOSFET: The type of transistor which forms the major part of a Lumos DC relay channel. The name is an acronym for Metal Oxide Semiconductor Field Effect Transistor.

PCB (Printed Circuit Board): The board where electronic components are mounted to form a complete circuit. Metal traces are "printed" (actually etched) onto the surface of the board itself to make the connections between components.

Power Cycle: To turn the power off, then on, thus resetting the state of the device. On a Lumos board, the same effect can be had by pressing the RESET button momentarily, although power cycling works, too. Note that this does not undo any permanent settings such as device address or baud rate. To return all settings to their original values, you must perform a full factory reset.

RS-232: A standard hardware protocol for sending serial data between two devices (such as a computer and a modem or a single Lumos board). Shielded cable should be used for best results, and the cable length should not exceed 25 ft.

RS-485: A standard hardware protocol for sending serial data between multiple devices on a single cable length (electrically it is a single cable which each device "taps into" along the line; physically it is typically a "daisy chain" arrangement where a short cable connects one device to the next, another cable to the next, and so on). Unshielded twisted-pair cable is used (like Ethernet cable), and the cable lengths should not exceed a total of 4,000 ft (1,200 m).

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.

Terminator Plug: An RS-485 network requires a terminator at each end. This is a small plug which plugs into the last unit in the daisy chain.

TTL (Transistor-Transistor Logic): One of the ways digital logic circuits

can be constructed. For our purposes here, we consider a "TTL-level" signal to be a logic input or output where a voltage near +5 V is "high" (binary 1 or "true") and a voltage near 0 V is "low" (binary 0 or "false"). The inputs should never be above +5 nor below 0 volts.

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

Casey 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 A.
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 sites such as 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 are entirely my own original design).

Colophon

This manual was composed and typeset by the author using LaTeX with Memoir layout macros, augmented by wrapfig, lettrine, bytefield, wallpaper, Ti*k*Z, and a host of miscellaneous behind-the-scenes working packages.

It was set 10/12 pt using the TeX Gyre Schola font family created by GUST, the Polish TeX 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.