

Readerboard and Busylight User's Guide

Includes details on assembly, installation, and programming these messaging units.



The information in this document, and the hardware and software it describes, are hobbyist works created as an educational exercise and as a matter of personal interest for recreational purposes.

It is not to be considered an industrial-grade or retail-worthy product. It is assumed that the user has the necessary understanding and skill to use it appropriately. 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. It is explicitly not designed for use where the safety of persons, animals, property, or anything of real value depends on the correct operation of the software.

As an educational exercise, it is assumed that the user has the necessary skill to appropriately set up, build, and use this project. Do not proceed to use, whether exactly as provided in the repository or with modifications of your own, unless you know exactly what you're doing and have verified that your actions and the product materials are suitable for your purpose and function as expected.

For Readerboard hardware (PCB) version 3.4.1, Busylight hardware version 2.2 (light tree version 1.1), and firmware version 2.3.7.

Copyright © 2023, 2024 by Steven L. Willoughby (aka MadScienceZone), Aloha, Oregon, USA. All Rights Reserved. This document is released under the terms and conditions of the Creative Commons “Attribution-NoDerivs 3.0 Unported” license. In summary, you are free to use, reproduce, and redistribute this document provided you give full attribution to its author and do not alter it or create derivative works from it. See

<http://creativecommons.org/licenses/bynd/3.0/>

for the full set of licensing terms.



CONTENTS

Contents	iii
List of Figures	v
List of Tables	v
1 Hardware	1
Version 1	1
Version 2	1
Version 2	2
Version 3	2
Firmware Implications	3
Construction	3
Busylight Devices	6
2 Protocol Description	11
Web Service API	11
USB vs. RS-485	12
Command Summary	15
%—Run Full Lamp Test Pattern	15
*—Strobe Lights in Sequence	17
<—Scroll Text Across Display	17
=—Set Operational Parameters	18
=#—Set Serial Number	19
=*<—Display POST Banners	19
=&—Persist Settings in EEPROM	20
?—Query Discrete LED Status	20
@—Set Column Cursor Position	21
A—Select Font	22
B—Beep/Play Sound Sequence	22
C—Clear Matrix Display	25
D—Set Dimmer Levels	25
F—Flash Lights in Sequence	26
H—Draw Bar Graph Data Point	27
I—Draw Bitmap Image	28

K—Set Current Color	28
L—Light Multiple LEDs	29
Q—Query Readerboard Status	30
M—Morse Code	31
S—Light Single LED	32
T—Display Text	32
X—Turn off Discrete LEDs	33
3 Server-Only Commands	35
current—Report Current Display State	35
post—Post a Message to the Displays	36
postlist—Report of Message Queue	38
unpost—Remove a Posted Message	38
update—Update Dynamic Content	39
4 Pinouts	41
8p8c RS-485 Connectors	41
DIN-8 Busylight Tree Connectors	43
Connectors for Version 3.4 Boards	43
Connectors for Legacy Boards	45
5 Installing and Using the Readerboard and Busylight Units	49
Power	49
Standalone Units	50
Networked Units	50
Compiling the Supporting Software	50
Setting up rbserver	50
Setting up Per-User Utilities	50
Diagnostic Displays	50
6 Font Glyphs	51
Font #0 standard.font	51
Font #1 standard_variable.font	60
Font #2 symbol.font	69
Default Glyph	76
Font Definition Files	77
7 Preparing and Updating Firmware Images	79
8 Schematics and Diagrams	85

LIST OF FIGURES

4.1	Female 8p8c Modular Jack with T-568B Color Codes	42
4.2	DIN-8 Jack (looking into socket from front)	43

LIST OF TABLES

1.1	Bill of Materials	4
1.2	Bill of Materials (Busylight)	8
2.1	Examples of RS-485 Escape Bytes.	15
2.2	Summary of All Commands	16
2.3	Control Codes in String Values	18
2.4	Baud Rate Codes	19
2.5	ASCII Encoded Integer Values (0–63)	22
2.6	Font Codes	22
2.7	Font Table for Fonts #0 and #1	23
2.8	Font Table for Font #2	24
2.9	Discrete LED Codes and Colors	26
2.10	Transition Effect Codes	29
2.11	Color codes for $\langle\text{rgb}\rangle$ parameters	29
2.12	Alignment Codes	33
3.1	Transition codes for use in post command	36
3.2	Alignment codes for use in post command	36
3.3	Color codes for use with the post command	37
3.4	Special Codes in Post Messages	38
7.1	Firmware Configuration Values	80

C H A P T E R

1

HARDWARE

People who are really serious about software should make their own hardware.

—Alan Kay

AS A HOBBY PROJECT, the design of the hardware and software for the read-erboard project “grew in the telling” as new ideas sprang to mind. As of this writing, there are three different models of hardware prototypes which were designed and created.

Version 1

The initial prototype (PCB 1.0.1) was a fairly large board, approximately $20\frac{5}{8} \times 6\frac{5}{8}$ “ with an LED pitch of $\frac{1}{8}$ “.

It requires an Arduino Mega 2560 or Arduino Due to drive it. A custom-made shield board attaches to the Arduino, which provides connectors for power and ribbon cables to drive the display (one to control the matrix, the other for the 8 status LEDs which appear in a horizontal row along the bottom right of the matrix).

Version 2

This version was based on the 1.0 design, but shrunk to a board size of about $18\frac{3}{16} \times 5\frac{3}{16}$ “ with an LED pitch of $\frac{1}{4}$ “. It also replaced the obsolete TPIC6B595 with the newer STPIC6D595 chip. Further, it relocated the ribbon cable which

connects the Arduino controller to the LED matrix, moved the eight status LEDs to a vertical column to the right of the matrix, and switched to resistor arrays for the matrix current limiting resistors, instead of the individual discrete resistors used on the 1.0.1 board.

Version 2

Version 2.1 was a significant change to the 2.0 board in terms of how it interfaces with the Arduino, although it had the same size and physical layout as the 2.0 board.

This one eliminates the need for the ribbon cables and shield board. Instead, the Arduino controller mounts directly to the back of the display board. This means that the lower portion of the left edge of the enclosure has all of the external connections in one place, including power, RS-485 in and out, and the USB connector(s) on the Arduino itself.

Version 3

Revision 3.x of the hardware expands on the version 2.1 design by replacing the single-color LEDs with RGB LEDs arranged into 24 logical rows—each of the eight physical rows is logically a row of red, green, and blue LED elements.

Note that the discrete status LEDs are mislabeled on the revision 3.2.2 PCB. L₀ should be the LED on the bottom (canonically green) and L₇ is on top (white).

Version 3.4.1 (current model)

Version 3.4 added support for playing sounds on a speaker and added a diode (D520) to prevent the Arduino from trying to power the LEDs by itself if it is powered from the USB port but the external 9 V supply is not active.

Version 3.3.1

Version 3.3 of the hardware corrects some minor issues with the 3.2.2 revision by adding a cutout to avoid colliding with the connectors on the Arduino board and adjusting the mounting holes slightly. It also corrects the mislabeled L₀–L₇ LEDs.

Additionally, it adds R51 for Arduino Due reset circuits, J3 to manually reset the Arduino, and J4, R52, R53, and U16 to support the addition of a external EEPROM chip for microcontrollers (such as the Due) which don't have one onboard.

This was further improved with revision 3.3.1 (the current version), which replaced terminal block J1 with a higher-density terminal which

allows easier interconnection of two CAT6 cables without the need for off-board splices. The power supply now connects to a new terminal block J5. The power to the RS-485 transceiver chip is also now switchable to either 3.3 V or 5 V so it can be appropriately matched to the operating voltage of the Arduino board without damaging the Arduino's input ports.

Firmware Implications

The boards before PCB revision 3.3.0 are not supported by the firmware as currently stored in the project repository. They were prototype ideas which were never built, so this shouldn't be a problem.

There are compile-time switches documented starting on p. 79 which adjust the firmware image for different microcontrollers, default settings—or the only settings if your hardware does not include an EEPROM memory—and possibly, in the future, different board revisions.

Construction

Once you have created a printed circuit board (PCB) from the fabrication files included with the readerboard Git repository and obtained the parts as listed in Figure 1.1, carefully solder the components in place.

Warning! Danger! Only perform the assembly operation if you are qualified to do so. Soldering electronic components may be hazardous due to the high temperatures, sharp objects, electrical voltages, potentially toxic materials, and other dangers. This requires skill and expertise as well as personal protection equipment to perform safely. Please also observe all precautions to avoid damaging the electronic components themselves from harm due to heat or static electricity discharge.

Suggested Order

For best results, solder components in a way that allows best access to the solder points without some parts getting in the way of connecting subsequent ones. For example:

1. Solder all the surface mount chips first: U0–U15 (and optionally, U16).
 - U0, U12, U13: CD74HCT238PWR
 - U1, U2, U14, U15: ULN2803CDWR (ULN2803A SOP-18 packages may be used if aligned to the top 18 pads, leaving the bottom two empty, if they have otherwise identical pinouts and characteristics)
 - U3–U10: STPIC6D595MTR

Part No.	Qty	Mouser	Description
	1		Printed circuit board rev 3.2.2
	1		Arduino Due or Mega 2560
C0–9,11,12	12	810-FA18X7R1H10400	0.1 µF capacitor
C10	1	667-ECA-1HM100I	10 µF electrolytic capacitor
C13	1		100 µF electrolytic capacitor
D0–511	512		LED, 5mm, RGB, common cathode
D512	1		LED, 5mm, green
D513–14	2		LED, 5mm, amber (yellow)
D515–16	2		LED, 5mm, red
D517–18	2		LED, 5mm, blue
D519	1		LED, 5mm, white
J0	1		Arduino stacking pin set
J1	2		8-position terminal block
J2,3	2		2-position jumper header
J4	1		3-position jumper header
J5	1		2-position Euro-style terminal block
P0–3	3		2-position jumper shunt
Q0–23	24	942-IRF9530NPBF	IRF9530 p-channel MOSFET
Q24	1		2N2222 or BC547 NPN transistor
R0–7,35–51	24	604-MFR-24FRF521K	1K resistor
R8–23	16		4×360Ω resistor array, isolated
R24,29–31	4	603-MFR-25FTE52-300R	300Ω resistor
R25–28	4		360Ω resistor
R32–34,52–54	6	603-MFR-25FRF5210K	10K resistor
R _{TERM}	1		120Ω resistor
U0,12,13	3	595-CD74HCT238PWR	CD74HCT238PWR non-inverting 3-to-8 decoder
U1,2,14,15	4		ULN2803CDWR octal NPN Darlington array
U3–10	8		STPIC6D595MTR power 8-bit shift register
U11	1		THVD1439 half-duplex RS-485 driver/receiver
U16	1		AT24C256 EEPROM
	24		TO-220 heat sinks for Q0–23 (optional)
	1		9V 2A DC power supply

Table 1.1: Bill of Materials

- U11: THVD1439DR (Note that this chip is oriented 180° from the other chips on the board, with pin 1 to the lower right.)
 - If needed for your microcontroller (Arduino Due, usually), solder U16: AT24C256 EEPROM chip.
2. Solder the discrete resistors: R0–R7, R24–R51, and R54 (and optionally, R52 and R53). See below first to ensure the correct values are used for R24–R31 based on your choice of corresponding LEDs.
 - R0–R7, R35–50: 1K
 - R24, R29–R31: 300Ω
 - R25–R28: 360Ω
 - R32–R34: 10K
 - If U16 is installed, solder in R52–R53: 10K
 - If using sound output, solder in R54: 10K
 3. Solder the 0.1 μF capacitors: C0–C9, C11–C12.
 4. Solder the resistor arrays: R8–R23.
 5. Solder the RGB LEDs D0–D511. Ensure that the common cathode goes to pin 2 so that pins 1, 3, and 4 are respectively the red, green, and blue anodes.
 6. Solder the single-color LEDs D512–D519 in the colors of your choice, being mindful of the polarity. The anode has a square pin on the PCB. Our recommendation:
 - D512 (bottom): green (with $R24=300\Omega$)¹
 - D513: amber² (with $R25=360\Omega$)
 - D514: amber (with $R26=360\Omega$)
 - D515: red (with $R27=360\Omega$)
 - D516: red (with $R28=360\Omega$)
 - D517: blue (with $R29=300\Omega$)
 - D518: blue (with $R30=300\Omega$)
 - D519 (top): white (with $R31=300\Omega$)
- 7. If you are using an Arduino Due:** solder R51 (1K).

¹Resistor values based on typical LED voltage and current ratings. Check to see what is appropriate for your components.

²In deference to the effort my high school drama teacher went through to get us to refer to any yellow stage lights and gels as “amber” and the preference to using “amber” for signal lamp colors in other contexts and regions, both terms will be used interchangeably in this project.

8. **If you are using an Arduino Mega 2560:** Solder the 10 μF capacitor C10, being mindful of the correct polarity. **Do not solder both R51 and C10. Only one or the other will be appropriate for a given microcontroller.**
9. Solder the Arduino bus stacking pins (six sets—1×10-pin, 5×8-pin—all labeled collectively at J0). Ensure the pins extend down and out the back of the board.
10. Solder the 2-pin headers J2 and J3, and the 3-pin header J4.
11. Solder the terminal blocks J1 (which is actually a pair of 8-pin terminal blocks J1a and J1b), and J5.

Option: Using Single-Color LEDs

If you don't need RGB LEDs for your application, the same PCB may be used with single-color 5mm LEDs instead. Simply solder them onto the board so that their cathode is on pin 2 and the anode is on pin 4 of the LED footprint. This means the LEDs will electrically all be connected to the blue circuit.

Since they won't be used, omit parts U0, U2, U12, U14, C0, C11, Q0–Q15, R0–R7, and R35–R42.

Set the firmware value HW_MODEL to MODEL_3xx_MONOCHROME and recompile the firmware. This changes the behavior of the controller so that it understands it can only drive one color which is attached to the blue color plane. Now, all colors (other than “off”) will turn the LEDs on.

Busylight Devices

The readerboard units are designed to be compatible with another project by the author. The “Busylight” is a small stand-alone indicator which can be used to indicate simple statuses like room occupancy or whether someone is busy, free, on a phone call, etc.

The busylight can drive up to seven LEDs. If directly connected to a host computer via USB, it can be powered entirely from the USB port. Starting with version 2, it can also receive commands via RS-485 network. In that case it can be powered by a 9 V supply over the same CAT5, CAT5e, or CAT6 cable carrying the serial communications.

Firmware

Starting with revision 2 of the Busylight, it is driven by a firmware image compiled from the same source code as the Readerboard units. Setting the compiler preprocessor symbol HW_MODEL to MODEL_BUSYLIGHT_2 will cause the Arduino compiler to prepare the image for a Busylight, excluding the

code that drives the readerboard-specific functions. In such firmware images, HW_MC should be set to HW_MC_PRO since the Busylight is based on the SparkFun Arduino-compatible Pro Micro microcontroller.

Also set the SERIAL_VERSION_STAMP symbol to reflect the device serial number, hardware and firmware revision numbers. See Chapter 7 starting on p. 79 for more information about preparing firmware images.

Construction

Once you have created a printed circuit board (PCB) from the fabrication files included with the Git repository and obtained the parts as listed in Figure 1.2, carefully solder the components in place.

Warning! Danger! Only perform the assembly operation if you are qualified to do so. Soldering electronic components may be hazardous due to the high temperatures, sharp objects, electrical voltages, potentially toxic materials, and other dangers. This requires skill and expertise as well as personal protection equipment to perform safely. Please also observe all precautions to avoid damaging the electronic components themselves from harm due to heat or static electricity discharge.

Suggested Order

For best results, solder components in a way that allows best access to the solder points without some parts getting in the way of connecting subsequent ones. For example:

1. Solder the surface mount components:
 - U0 ULN2003A 7-circuit NPN Darlington transistor array. Note that the board was designed for narrow SOIC-16 packages but there is an additional column of pads for a wider version of this chip.
 - U1 THVD1439 RS-485 driver/receiver.
2. Solder through-hole components:
 - R0–R2 10K resistors
 - C0 0.1 µF capacitor
3. Solder the sockets:
 - J0 8-pin DIN socket
 - J1 16-circuit Euro-style terminal block (alternatively you may choose to directly solder the CAT6 wires into the holes on the PCB.)
 - Arduino stacking pins.
4. Place the microcontroller board onto the stacking pins.

Part No.	Qty	Description
	1	Printed circuit board rev 2.1
	7	LED tier printed circuit board rev 1.1
	7	LED ring printed circuit board (if desired)
	1	SparkFun Pro Micro Arduino-compatible μ C, 5 V version
	1	Arduino stacking pin set
C0	1	0.1 μ F capacitor
J0	1	8-pin DIN connector, PCB mount
J1	1	16-pin Euro-style terminal block
P0	1	8-pin DIN plug
R0-2	3	10K resistor
U0	1	ULN2003A NPN Darlington array
U1	1	THVD1439 half-duplex RS-485 driver/receiver
	14	5-pin bussed resistor network (2/LED tier)*
	56	LED, 5 mm or 3 mm, through-hole (8/LED tier)†
	1	$\frac{1}{4}$ " screw rod
	1	Enclosure‡
	8	Nylon spacers between (and above and below) the tiers
	2	$\frac{1}{4}$ " hex nuts

*Choose each tier's pair of resistor arrays appropriately for the voltage drop for the LEDs to be used on that tier. Assuming a 5 V supply and typical LED characteristics for a desired 20 mA current draw, that would be 150 Ω or larger for red and amber LEDs and 100 Ω or larger for blue, green, and white LEDs. Check your LED datasheet to confirm.

†Choose LED colors as desired for each tier (standard is green, 2×amber, 2×red, blue, and white).

‡One idea for an inexpensive enclosure is a clear plastic tube storage container with an inside diameter large enough to hold the LED tier boards.

Table 1.2: Bill of Materials (Busylight)

5. Build each LED tier of 8 LEDs:

- Solder eight 3 mm LEDs around the edge of the board with their leads straddling the edge, so the anode is on the side of the PCB labelled “anode side” and the cathode is on the “cathode side.” Solder the leads to the exposed pads and trim the leads so they don't extend past the pads.
- Solder the two resistor arrays, making sure to select the correct values for the specific LEDs used on this tier board.

6. Assemble the tier stack by threading the tiers onto the $\frac{1}{4}$ " threaded rod with nylon spacers between them, secured in place with hex nuts on either end of the stack. The following assumes the use of a length of CAT6 cable between the PCB and the light tree stack.

- Solder one end of the cable to P0, the 8-pin DIN plug. Suggested color assignments:
 - Pin 1 (sixth tier/blue): blue

- Pin 2 (third tier/amber): brown/white
- Pin 3 (fifth tier/red): orange/white
- Pin 4 (fourth tier/red): orange
- Pin 5 (seventh tier/white): blue/white
- Pin 6 (second tier/amber): brown
- Pin 7 (+5 V supply): green
- Pin 8 (bottom tier/green): green/white
- Strip the insulation from the green wire to the height of the assembled tier tree. As each tier is added, thread this bare wire through the “V+” hole of each tier, soldering it to each tier along the way.
- Attach the wire for each tier to its “GND” hole.
- Thread the remaining wires through the larger holes to pass them up to the next tiers.

C H A P T E R



PROTOCOL DESCRIPTION

I don't stand on protocol. Just call me
your Excellency.

—Henry Kissinger

THE CONTROL PROTOCOL used to display information on the readerboard sign is very simple. Commands are expressed largely in plain ASCII characters and are executed immediately as they are received.¹

In addition to plain, printable 7-bit ASCII characters, a few control codes are recognized as described below. String data may include any 8-bit value except as otherwise indicated.

Web Service API

The organization of the rest of this chapter is oriented toward a description of the protocol used to control the hardware directly over a USB or RS-485 connection. However, you can set up a central server which is connected to all the readerboard signs and busylight indicators in a small area (via USB or RS-485). This allows clients to send commands to the server over the network for control of the readerboards.

The protocol between clients and servers uses messages which are simple HTTP requests posted to the server's listening port URL. If the server were configured to run on example.org on port 43210, then all client requests to that server would begin with

`http://example.org:43210/readerboard/v1/<message>?a=<targets>`

¹Technically, they may even be executed *while* they are being received.

Message Data Fields

As a general rule, if a message expects a field that is not provided, where possible, a suitable “zero” value will be assumed for that field. Any extra fields sent that were not expected are silently ignored.

Boolean fields may have values true or false; if they are presented without a value, they are considered to be true, and if completely missing they are assumed to be false. Thus,

```
.../readerboard/v1/<message>?a=<targets>&status=true  
is equivalent to  
.../readerboard/v1/<message>?a=<targets>&status  
and  
.../readerboard/v1/<message>?a=<targets>&status=false  
is equivalent to  
.../readerboard/v1/<message>?a=<targets>
```

Message Target Addresses

Every message also has a field called “a” which is a comma-separated list of target readerboards to which the message applies.

So, for example, the message containing a=2 would cause the server to transmit the corresponding command to the readerboard with address 2 directly connected to it via USB, or would transmit to the RS-485 network the command starting with the hex byte 92₁₆. (Why that byte value is sent will make sense after you read the protocol description that follows.)

If multiple devices are targeted, e.g., with a message containing the field a=2,5,37 with address 15 (i.e., 0F₁₆) configured as the global address ad_G , it will transmit to the RS-485 network a command starting with the hex byte sequence BF₁₆ 03₁₆ 02₁₆ 05₁₆ 25₁₆.

USB vs. RS-485

The protocol used to send commands to the readerboard is different depending on whether the host is sending directly to a single readerboard over a USB cable, or to (possibly) multiple readerboards over an RS-485 bus network.

USB

A readerboard connected via USB accepts the commands just as documented below, with the addition that each such command is terminated by a ^D byte (hex value 04₁₆).

0	...	$n - 1$	n
$\langle command \rangle$			^D

If there is an error parsing or executing a command, the readerboard will ignore all subsequent input until a ^D is received, whereupon it will expect to see the start of another command. Thus, ^D may not appear in any transmitted data except to terminate commands.

Commands which contain arbitrary-size data fields, such as text strings or lists of LEDs to light in sequence, end such fields with a terminator byte. In most cases this may be either a dollar-sign (“\$”) character or the escape control character (hex byte $1B_{16}$), indicated in the protocol diagrams below simply as “\$”. In cases where a dollar-sign could be part of the data, then only an escape character may be used to terminate the field, in which case the protocol description will show the terminator as “ESC”.

RS-485

Commands sent over RS-485 are intended to target one or more of a set of connected readerboards over a network which may also contain other Lumos-protocol-compatible devices, so they adhere to a protocol that is also compatible with those devices.

Each command begins with one of the following binary headers, depending on the set of target readerboard signs which should obey the command.

Single Target or All Readerboards

To send a command to a single sign, begin with a single byte encoded as:

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

where $\langle ad \rangle$ is the sign’s address on the bus, which must be a value in the range 0–15. This byte is followed by any command as described below. If the global address ad_G is given as the $\langle ad \rangle$ value, then all readerboards which have that set as their global address will obey the command.

Multiple Targets

Alternatively, a command may be targeted to multiple signs by starting the command with a multi-byte code:

7	6	5	4	3	2	1	0				
1	0	1	1	$\langle ad_G \rangle$							
0	0	$\langle n \rangle$									
0	0	$\langle ad_0 \rangle$									
⋮											
0	0	$\langle ad_{n-1} \rangle$									

where $\langle ad_G \rangle$ is the “global” device address which signals readerboards generally (see the = command below). This will send to the $\langle n \rangle$ devices addressed as $\langle ad_0 \rangle$ through $\langle ad_{n-1} \rangle$.

Note that device addresses are constrained to the range 0–15 if they are to be addressed in the command start byte. However, using the multiple target header, device addresses in the range 0–63 may be used.

Response Headers

For commands which send a response back to the computer, the RS-485 data stream begins with the header

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

where $\langle ad \rangle$ is the device address of the responding unit if $\langle ad \rangle < 16$. For devices with higher-numbered addresses, the following extended format is used instead:

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

All Off

As a special case, the single byte

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

will cause the readerboard addressed as $\langle ad \rangle$ to turn off all LEDs. If $\langle ad \rangle$ is the ad_G address, then all readerboards will turn off all LEDs.

No other command bytes need to follow; this byte is sufficient to turn off the sign(s).

The corresponding HTTP message is `/readerboard/v1/alloff?a=⟨ad⟩`

Subsequent Command Bytes

All subsequent bytes which follow the above binary headers *must* have their MSB cleared to 0.

To cover cases where a value sent as part of a command must have the MSB set, we use the following escape codes:

- A hex byte $7E_{16}$ causes the next byte received to have its MSB set upon receipt.
- A hex byte $7F_{16}$ causes the next byte to be accepted without any further interpretation.

Input Sequence	Resulting Byte
00	00
7D	7D
7F 7E	7E
7F 7F	7F
7E 00	80
7E 01	81
7E 7D	FD
7E 7E	FE
7E 7F	FF

Table 2.1: Examples of RS-485 Escape Bytes.

Thus, the byte C4₁₆ is sent as the two-byte sequence 7E 44, while a literal 7E is sent as 7F 7E and a literal 7F as 7F 7F.

If there is an error parsing or executing a command, the readerboard will ignore all subsequent input until a byte arrives with its MSB set to 1, whereupon it will expect to see the start of another command.

A few illustrative examples are shown in Table 2.1.

Command Summary

The eight discrete LEDs are intended for a simple display of status information in a manner analogous to the Busylight project by the same author.² To support this usage, the F, S, X, *, and ? commands are recognized in a manner compatible with how Busylight uses those same commands. These are categorized as “Busylight compatibility commands.” Unlike all other commands listed here, these are recognized regardless of case. Since the readerboard has a power supply capable of illuminating all of the status LEDs at once,³ a new command L is added which allows any arbitrary pattern of steady LEDs to be turned on.

The remaining commands are used for management of the matrix display. All commands are summarized in Table 2.2.

%—Run Full Lamp Test Pattern

0
%

/readerboard/v1/test?a=<ad>

²See github.com/MadScienceZone/busylight.

³The Busylight cannot, since it is powered from the host computer’s USB port.

Message	Cmd	Description	Notes
strobe	*	Strobe LEDs in Sequence	[1]
busy	?	Query discrete LED status	[1] [2] [5]
flash	F	Flash LEDs in Sequence	[1]
light	L	Light one or more LEDs steady	[3]
light	S	Light one LED steady	[1]
off	X	All LEDs off	[1]
	~D	Abort/terminate command	
test	%	Full lamp test	
scroll	<	Scroll text across display	
configure-device	=	Set operational parameters	[4] [5]
diag-banners	==	Display POST banners	
save	=&	Persist Settings in EEPROM	
	=#	Set serial number	[4] [5]
move	@	Move current column cursor	
font	A	Select character font	
sound	B	Beep / Play sounds	
clear	C	Clear matrix display	
dim	D	Set light dimmer levels	
graph	H	Add histogram/bargraph data point	
bitmap	I	Draw bitmap graphic image	
color	K	Set current color	
morse	M	Send message via Morse code	
query	Q	Query full device status	[2] [5]
text	T	Display text on display	
current		Report current status display	
post		Add queued display message	
postlist		Report queued display message(s)	
unpost		Remove queued display message(s)	
update		Update user-defined variables	

[1] Busylight compatible command

[2] Sends response

[3] Busylight extension (not in original Busylight)

[4] USB only

[5] Must be targeted to a specific device at a time; does not respond on the global address

Table 2.2: Summary of All Commands

Displays various test patterns on the display to show that each element of each LED is working correctly individually and in concert with others.

While running the test pattern set, the device is not responsive to other inputs or background operations.

*—**Strobe Lights in Sequence**

0	1	2	3	...	n	$n + 1$
*	$\langle led_0 \rangle$	$\langle led_1 \rangle$	$\langle led_2 \rangle$...	$\langle led_{n-1} \rangle$	\$
<code>/readerboard/v1/strobe?a=<ad>&l=<led₀><led₁>...<led_{n-1}></code>						

Each $\langle led \rangle$ value is an ASCII character corresponding to a discrete LED as shown in Table 2.9. An $\langle led \rangle$ value of “_” means there is no LED illuminated at that point in the sequence.

This command functions identically to the F command (see below), except that the lights are “strobed” (flashed very briefly with a pause between each light in the sequence).

<—**Scroll Text Across Display**

0	1	2	...	$n+1$	$n+2$
<	$\langle loop \rangle$		$\langle string \rangle$		ESC
<code>/readerboard/v1/scroll?a=<ad>&loop=<bool>&t=<string></code>					

Displays the text $\langle string \rangle$ by scrolling it across the display from right to left. If $\langle loop \rangle$ is “.”, the text is only scrolled once; if it is “L” then it repeatedly scrolls across the screen in an endless loop.

The text is rendered in the current font and may contain any 8-bit bytes except as otherwise noted (but avoiding ASCII control codes is wise to be safe from conflict with future control codes which may be added to the protocol). The string is terminated by an escape character (hex byte 1B₁₆), indicated in the protocol description as ESC.

The string may include control codes as listed in Table 2.3. These allow for special effects such as color and font changes to appear mid-message. They also include the three-character sequence $\sim X\langle hh \rangle$ to add any arbitrary 8-bit codepoint specified as the 2-digit hex value $\langle hh \rangle$ to compensate for interference from your terminal program or transport protocols which may, e.g., encode non-7-bit-ASCII characters to a multi-byte UTF-8 representation which the readerboard doesn’t grok.

Code	Hex	Description
\^@	00	Never allowed in strings (null byte)
$\text{\^C}\langle pos \rangle$	03⟨pos⟩	Move current column cursor to ⟨pos⟩
\^D	04	Never allowed in strings (command terminator)
$\text{\^F}\langle digit \rangle$	06⟨digit⟩	Switch current font
$\text{\^H}\langle pos \rangle$	08⟨pos⟩	Move cursor left ⟨pos⟩ columns
$\text{\^K}\langle rgb \rangle$	0B⟨rgb⟩	Change color to ⟨rgb⟩
$\text{\^L}\langle pos \rangle$	0C⟨pos⟩	Move cursor right ⟨pos⟩ columns
$\text{\^X}\langle hex \rangle$	18⟨hex⟩	Specify character code as 2-digit hex value
\^[1B	Never allowed in strings (string terminator)

Table 2.3: Control Codes in String Values

==Set Operational Parameters

0	1	2	3	4
=	$\langle ad' \rangle$	$\langle uspd \rangle$	$\langle rspd \rangle$	$\langle ad'_G \rangle$

/readerboard/v1/configure-device?a= $\langle ad \rangle$ &rspeed= $\langle baud \rangle$
 &uspeed= $\langle baud \rangle$ &address= $\langle ad' \rangle$ &global= $\langle ad'_G \rangle$

This command sets a few operational parameters for the sign. Once set, these will be persistent across power cycles and reboots.

If the $\langle ad' \rangle$ parameter is “.” then the RS-485 interface is disabled entirely. Otherwise it is a value from 0–63 encoded as described in Table 2.5. This enables the RS-485 interface and assigns this sign’s address to $\langle ad' \rangle$. (In the HTTP API, use “address=_” to indicate this.)

The baud rate for the USB and RS-485 interfaces is set by the $\langle uspd \rangle$ and $\langle rspd \rangle$ values respectively. Each is encoded as per Table 2.4.

The $\langle ad'_G \rangle$ value is an address in the range 0–15 which is not assigned to any other device on the RS-485 network. This is used to signal that all readerboards should pay attention to the start of the command because it might target them either as part of a list of specific readerboards or because the command is intended for all readerboards at once. This is encoded in the same way as $\langle ad' \rangle$.

This command may only be sent over the USB port.

By default, an unconfigured readerboard is set to 9,600 baud with the RS-485 port disabled, but these defaults may be changed when compiling the firmware for specific devices.

Code	Speed
0	300
1	600
2	1,200
3	2,400
4	4,800
5	9,600 (default)
6	14,400
7	19,200
8	28,800
9	31,250
A	38,400
B	57,600
C	115,200

Table 2.4: Baud Rate Codes

=#—Set Serial Number

0	1	2	...			
=	#		<i><Serial></i>	\$	#	=

To avoid recompiling the firmware image for every single device, the serial number may be set in the field using this command. The serial number is stored in EEPROM, so it won't be persistent on units that are not equipped with EEPROM.

This command may only be sent over the USB port.

The serial number may be arbitrary alphanumeric text of up to six characters. Numbers RB0000–RB0299 and B00000–B00299 are reserved for use by the author.

=*—Display POST Banners

0	1	2
=	*	=

/readerboard/v1/diag-banners?a=*<ad>*

Causes the readerboard to display the same banners as it does during power-on self-test. Busylights don't display the equivalent series of messages when they power up, but if you send them this command they will send the same information as the readerboard POST banners, but will do so by flashing out the information in Morse code on the bottom LED.

=&—Persist Settings in EEPROM

0	1	2	3
=	&	$\langle type \rangle$	=

/readerboard/v1/save?a= $\langle ad \rangle$ &type= $\langle type \rangle$

Saves current system state in EEPROM so they remain in effect across reboots. Currently, the only supported $\langle type \rangle$ is “D” which saves the current dimmer settings.

?—Query Discrete LED Status

0
?

/readerboard/v1/busy?a= $\langle ad \rangle$

This command causes the sign to send a status report back to the host to indicate what the discrete LEDs (usually showing a person’s “busy” status) are currently showing.

This cannot be sent to the global device address, although you can request the server to query the global address in which case it will individually poll each device.

This response has the form:

0	1	2	3	4	5	6	7	8
L	0	$\langle led_0 \rangle$	$\langle led_1 \rangle$	$\langle led_2 \rangle$	$\langle led_3 \rangle$	$\langle led_4 \rangle$...	$\langle led_{n-1} \rangle$
\$	F	flasher status (see below)						\$
S	strober status (see below)						\$	\n

The initial bytes “L0” identify the format of the data that follows. If the format changes in a way that breaks the previous parsing rules, the “0” will be incremented according to the conventions of 6-bit integer encoding as described in Table 2.5.

Each $\langle led_x \rangle$ value is a single character which is “_” if the corresponding LED is off, or the LED’s color code or position number if it is on. One such value is sent for each LED installed in the sign (typically eight for readerboards), followed by a “\$” to mark the end of the list.

The flasher and strober status values are variable-width fields which indicate the state of the flasher (see F command) and strober (see * command) functions. In each case, if there is no defined sequence, the status field will be:

0	1					
$\langle run \rangle$	-					
or						
0	1	2	3	4	5	6
/	$\langle up \rangle$	$\langle on \rangle$	$\langle down \rangle$	$\langle off \rangle$	$\langle run \rangle$	-

Otherwise, the state of the flasher or strober unit is indicated by:

0	1	2	3	4	...	$n + 3$
$\langle run \rangle$	$\langle pos \rangle$	@	$\langle led_0 \rangle$	$\langle led_1 \rangle$...	$\langle led_{n-1} \rangle$
or						
0	1	2	3	4	5	6
/	$\langle up \rangle$	$\langle on \rangle$	$\langle down \rangle$	$\langle off \rangle$	$\langle run \rangle$	$\langle pos \rangle$
$\langle led_0 \rangle$	$\langle led_1 \rangle$...	$\langle led_{n-1} \rangle$			@

If the flasher has custom timings defined for it, a five-character block starting with “/” will appear at the start of the status and will contain the four timing values in the same format as specified to the F command when they were set (q.v.).

In any case, $\langle run \rangle$ is the ASCII character “S” if the unit is stopped or “R” if it is currently running. If there is a defined sequence, $\langle pos \rangle$ indicates the 0-origin position within the sequence of the light currently being flashed or strobed, encoded as described in Table 2.5. The $\langle led_x \rangle$ values are as allowed for the F or * command that set the sequence. (Regardless of the actual F or * command parameters, the report will show symbolic color codes where possible, or numeric position codes otherwise.)

The status message sent to the host is terminated by a newline character (hex byte 0A), indicated in the protocol description above as “\n”.

©—Set Column Cursor Position

0	1
@	$\langle pos \rangle$

/readerboard/v1/move?a= $\langle ad \rangle$ &pos= $\langle pos \rangle$

Sets the column cursor position to the value indicated by $\langle pos \rangle$. See Table 2.5.

7	6	5	4	3	2	1	0
0	0	1	1		$\langle n \rangle$	($n < 16$)	
0	1			$\langle n \rangle - 16$	($n \geq 16$)		

Value	Code	Value	Code
0–9	0–9	17–42	A–Z
10	:	43	[
11	;	44	\
12	<	45]
13	=	46	~
14	>	47	-
15	?	48	'
16	Ø	49–63	a–o

(Each code is the numeric value plus 48.)

Table 2.5: ASCII Encoded Integer Values (0–63)

Code	Font Description
0	Fixed-width 5×7 matrix plus descenders in 8th row
1	Variable-width version of font 0
2	Bold alphanumerics and special symbols

Table 2.6: Font Codes

A—Select Font

0	1
A	$\langle digit \rangle$
/readerboard/v1/font?a= $\langle ad \rangle$ &idx= $\langle digit \rangle$	

Sets the font to use for rendering text with the `<` and `T` commands. The font codes for $\langle digit \rangle$ are listed in Table 2.6. The full text fonts support the printable ASCII characters plus a majority of the Unicode glyphs with codepoints less than 256. See Tables 2.7–2.8 for a complete font glyph listing with codepoint assignments.

B—Beep/Play Sound Sequence

0	1	2	3	4	...
B	$\langle loop \rangle$	$\langle note \rangle$	$\langle cs \text{ (hex)} \rangle$...	\$
/readerboard/v1/sound?a= $\langle ad \rangle$ &loop= $\langle bool \rangle$ ¬es= $\langle note \rangle \langle cs \rangle ...$					

	0	1	2	3	4	5	6	7	
00x				move	end		font		0x
01x	back			color	forw				
02x									1x
03x	hex			esc					
04x		!	"	#	\$	%	&	,	2x
05x	()	*	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	3x
07x	8	9	:	;	<	=	>	?	
10x	@	A	B	C	D	E	F	G	4x
11x	H	I	J	K	L	M	N	O	
12x	P	Q	R	S	T	U	V	W	5x
13x	X	Y	Z	[\]	^	_	
14x	'	a	b	c	d	e	f	g	6x
15x	h	i	j	k	l	m	n	o	
16x	p	q	r	s	t	u	v	w	7x
17x	x	y	z	{		}	~	///	
20x	"	"	'	,	†	‡	...	'	8x
21x	"	!!	-	←	→	↑	↓	≠	
22x	≤	≥	≈	Γ	Δ	Ξ	Π	Σ	9x
23x	Ω	π	ρ	σ	—	%o			
24x		í	¢	£	¤	¥	¡	§	Ax
25x	"	©	ª	«	¬	-	®		
26x	°	±	²	³		μ	¶	•	Bx
27x	'	¹	º	»	1/4	1/2	3/4	¿	
30x	À	Á	Â	Ã	Ä	Å	Æ	Ç	Cx
31x	È	É	Ê	Ë	Í	Í	Î	Ï	
32x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Dx
33x	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
34x	à	á	â	ã	ä	å	æ	ç	Ex
35x	è	é	ê	ë	ì	í	î	ï	
36x	ð	ñ	ò	ó	ô	õ	ö	÷	Fx
37x	ø	ù	ú	û	ü	ý	þ	ÿ	
	8	9	A	B	C	D	E	F	

Table 2.7: Font Table for Fonts #0 and #1

	0	1	2	3	4	5	6	7	
00x				move	end		font		
01x	back			color	forw				0x
02x									
03x	hex			esc					1x
04x	!	□	■						2x
05x			,			.	thin .		
06x	0	1	2	3	4	5	6	7	3x
07x	8	9	:						
10x	©	A	B	C	D	E	F	G	4x
11x	H	I	J	K	L	M	N	O	
12x	P	Q	R	S	T	U	V	W	5x
13x	X	Y	Z						
14x	®	←	→	↑	↓	↖	↗	↘	6x
15x	↙	←	→	↑	↓	■	■		
16x	✓	✓	∞	Œ	œ	€	⋮	⋮	7x
17x	✗	◀	▶	▲	▼	↑↓	◆	◊	
20x	λ	Θ	Φ	Ψ	♂	♀		zero	8x
21x	±	⊤	⊕	⊙	≡	clover	notes	EE	
22x	AM	PM	°F	°C	wide Œ	wide œ			9x
23x	The phases of the moon								
24x		TS1	TS2	TS3	TS4	TS5	TS6	TS7	Ax
25x	TS8								
26x	The eight trigrams 0–7								Bx
27x									
	8	9	A	B	C	D	E	F	

TS $\langle n \rangle$ = Thin space of $\langle n \rangle$ pixels

Table 2.8: Font Table for Font #2

This command causes the device to play a sequence of notes on the speaker. If $\langle loop \rangle$ is “L”, the note sequence is played repeatedly until another B command is received. If the sequence is empty, all sound is stopped. If $\langle loop \rangle$ is “.”, it is only played once.

The sequence consists of a series of $\langle note \rangle$ values, each of which is followed by the note duration $\langle cs \rangle$ which is a two-character hexadecimal integer in centiseconds (1/100th of a second). Each $\langle note \rangle$ begins with a letter A–G, optionally followed by # or b to indicate a sharp (#) or flat (b) note, respectively, followed by a digit 0–8 to indicate the octave. The notes range from B0 (B₀ at 31 Hz) to D#8 (D₈[#] at 4.978 kHz). A rest is indicated by “R” as the note letter.

An example 3-note attention tone where the first two notes are played for $\frac{1}{2}$ second and the final one is held for a full second is “G432E532C564” (note

G_4 , 32_{16} cs=0.50 s, note E_5 , 32_{16} cs=0.50 s, note C_5 , 64_{16} cs=1.00 s). Duration codes range from 00 ($0_{16}=0_{10}$ cs=0.00 s) to FF ($FF_{16}=255_{10}$ cs=2.55 s), in increments of 0.01 s.

The letters may be upper- or lower-case (the letter b is distinguished as a note name vs. a “b” half-step shift indicator from context).

Some devices are not able to play notes but can sound a beep at a fixed frequency (typically 490 Hz or just ever so slightly flat of B_4) for SparkFun Pro Micro and Arduino Mega 2560 microcontrollers and 1 kHz or between B_5 and C_6 for Arduino Due). These will play that single tone regardless of the note specified in the sequence.

C—Clear Matrix Display

0
C

/readerboard/v1/clear?a= $\langle ad \rangle$

Clears the matrix display so that no LEDs are illuminated. Does not affect the discrete LEDs.

D—Set Dimmer Levels

0	1	2	3
D	$\langle led \rangle$	$\langle level \text{ (hex)} \rangle$	

/readerboard/v1/dim?a= $\langle ad \rangle$ &l= $\langle led \rangle$ &d= $\langle level \rangle$

This command adjusts the apparent brightness of the LEDs. The brightness level is expressed as a two-digit hex value from 00_{16} (fully off) to FF_{16} (fully on). (In the HTTP API, as integers 0–255.)

The specific lamp to adjust is given by $\langle led \rangle$, and can have any of the values shown in Table 2.9 to dim one of the discrete status LEDs, or “_” to dim the matrix LEDs, or “*” to dim all the discrete status LEDs to the same value.

Not all LEDs are dimmable, depending on the specific hardware capabilities. All LEDs on readerboard models from revision 3.x and after. On the busylight hardware, only the third, fifth, and seventh, LED tiers (usually assigned amber, red, and white colors) are dimmable.

Note that the readerboards were designed to run best at full brightness. Dimming the matrix display too much may result in the display appearing to flicker too much. This effect is most pronounced with white content, a little less so with yellow, cyan, and magenta.

Code*	Light	Color
W	L ₇	white
B	L ₆	blue
b	L ₅	blue
R	L ₄	red
r	L ₃	red
Y	L ₂	amber
y	L ₁	amber
G	L ₀	green
-	—	(no LED/off)
0–9	L ₀ –L ₉	LED installed at physical position 0–9

*If a sign is built with different colors in these positions, the letter codes for those LEDs will match the custom color arrangement for that sign.

(Custom firmware modification required.)

Table 2.9: Discrete LED Codes and Colors

F—Flash Lights in Sequence

0	1	2	3	...
F	$\langle led_0 \rangle$	$\langle led_1 \rangle$	$\langle led_2 \rangle$...

or

0	1	2	3	4	5	6	...
F	/	$\langle up \rangle$	$\langle on \rangle$	$\langle down \rangle$	$\langle off \rangle$	$\langle led_0 \rangle$...

/readerboard/v1/flash?a= $\langle ad \rangle$ &l= $\langle led_0 \rangle\langle led_1 \rangle\langle led_2 \rangle\dots\langle led_{n-1} \rangle$
 $\&up=\langle sec \rangle\&on=\langle sec \rangle\&down=\langle sec \rangle\&off=\langle sec \rangle$

Each $\langle led \rangle$ value is an ASCII character corresponding to a discrete LED as shown in Table 2.9. Note that the assignment of colors to these LEDs is dependent on your particular hardware being assembled that way. As an open source project, of course, you (or whomever assembled the unit) may choose any color scheme you like when building the board.

An $\langle led \rangle$ value of “_” means there is to be no LED illuminated at the corresponding position in the sequence.

Up to 64 $\langle led \rangle$ codes may be listed. The sign will cycle through the sequence, lighting each specified LED briefly before moving on to the next one. The sequence is repeated forever in a loop until an L, S or X command is received.

If only one $\langle led \rangle$ is specified, that light will be flashed on and off. Setting an empty sequence (no codes at all) stops the flasher’s operation.

The sequence is terminated by either a dollar-sign (“\$”) character or the escape control character (hex byte 1B₁₆), indicated in the protocol diagram above simply as “\$”.

Optionally, a block of timing values may be given before the block of $\langle led \rangle$ values. This is introduced by a “/” character and consists of four 6-bit encoded integer values giving the time interval over which the light should fade up to full brightness, the time to remain on, the interval over which to dim down, and the time to remain off. These values are in units of $\frac{1}{10}$ th of a second (thus ranging from 0–6.4 seconds). For LEDs which do not support dimming, the values of $\langle up \rangle$ and $\langle down \rangle$ are treated as if they were zero, instantly turning on or off the LED. (In the HTTP API, the $\langle sec \rangle$ values are floating-point numbers of seconds; e.g., a value of 1.2 seconds would be expressed in the API as “up=1.2” and in the hardware protocol as “<”).

If you have set a dimmer level for any of the LEDs, then the maximum brightness that they will ramp up to and down from will be that dimmer level. If the dimmer is set very low, then the effect of $\langle up \rangle$ and $\langle down \rangle$ won’t be too noticeable since you didn’t leave it much room to dim up and down within that range.

This command may be given in upper- or lower-case (“f” or “F”).

H—Draw Bar Graph Data Point

			0	1		
<i>/readerboard/v1/graph?a=<ad>&v=<n></i>						
			0	1	2	3
			or	4	5	6
H	K	$\langle rgb_0 \rangle$	$\langle rgb_1 \rangle$	$\langle rgb_2 \rangle$	$\langle rgb_3 \rangle$	$\langle rgb_4 \rangle$
$\langle rgb_6 \rangle$	$\langle rgb_7 \rangle$					
<i>/readerboard/v1/graph?a=<ad>&v=<n>&colors=<rgb_0><rgb_1><rgb_2><rgb_3></i>						
<i><rgb_4><rgb_5><rgb_6><rgb_7></i>						

This command is used to draw a bar-graph element. Repeating this command causes a scrolling data display which shows a set of data samples over some period of time.

In the simplest form, it takes a single byte parameter. The value $\langle n \rangle$ is an ASCII digit character in the range “0”–“8”, and is drawn in the far-right column of the matrix display, as a column of $\langle n \rangle$ lights stacked up from the bottom row (a value of 0 results in no lights, up to 8 which is a full column of eight lights; a value of 9 is treated as if it were 8). All existing matrix data are scrolled left one column.

In the second form, it takes eight $\langle rgb \rangle$ values (see Table 2.11) which give the color to light up each row in the bar-graph column. For consistency with other drawing commands, $\langle rgb_0 \rangle$ refers to the topmost LED in the column.

In HTTP, the value of $\langle n \rangle$ is sent as an integer value.

I—Draw Bitmap Image

0	1	2	3	4	5	6	7	...
I	$\langle merge \rangle$	$\langle pos \rangle$	$\langle trans \rangle$	$\langle R\ coldata_0 \rangle$	$\langle R\ coldata_1 \rangle$			
$\langle R\ coldata_{n-1} \rangle$	\$	$\langle G\ coldata \rangle \dots$	\$	$\langle B\ coldata \rangle \dots$	\$			
$\langle flashdata \rangle \dots$	\$							

$/readerboard/v1(bitmap?a=\langle ad \rangle&merge=\langle bool \rangle&pos=\langle pos \rangle&trans=\langle trans \rangle&image=\langle R\ coldata \rangle\dots \$\langle G\ coldata \rangle\dots \$\langle B\ coldata \rangle\dots \$\langle flashdata \rangle)$

Draws an arbitrary bitmap image onto the matrix display starting at column $\langle pos \rangle$, encoded as per Table 2.5. A $\langle pos \rangle$ value of “~” represents the current column cursor position.

Each column data, from left to right, are given by $\langle coldata \rangle$ values, each of which is a two-digit ASCII hexadecimal value with the least-significant bit representing the top row of the matrix.

The column data values are terminated by either a dollar-sign (“\$”) character or the escape control character (hex byte 1B₁₆), indicated in the protocol diagram above simply as “\$”.

They may be short (or entirely empty if the corresponding color plane has no pixels set).

The first set of column data provide the bits for the red color plane. After the terminating byte (\$ or ESC), two more sets of column data are sent, each with identical format to the first, to provide the bits for the green and blue color planes, respectively. The final plane, $\langle flashdata \rangle$, causes the corresponding pixel to flash for any bits set to 1. Monochrome boards accept only two color bitplanes: the bitmap of pixels to light and then the flash plane.

The column cursor is moved to be after the end of the image.

If $\langle merge \rangle$ is the character “.” then each column’s contents is cleared before setting the pixels as per the $\langle coldata \rangle$ value. If $\langle merge \rangle$ is “M” the bits set in $\langle coldata \rangle$ are added to the lit pixels already in the column.

The $\langle trans \rangle$ value indicates the transition effect to use when adding the image to the display. See Table 2.10.

K—Set Current Color

0	1
K	$\langle rgb \rangle$

$/readerboard/v1/color?a=\langle ad \rangle&color=\langle rgb \rangle$

This command sets the color which all future commands will use by default (although some of them allow for the specification of colors directly on an *ad hoc* basis, which does not affect the current default color).

Code	Transition
.	No transition
>	Scroll in from left
<	Scroll in from right
^	Scroll up from bottom
v	Scroll down from top
L	wipe left
R	wipe right
U	wipe up
D	wipe down
	wipe left and right from middle column
-	wipe up and down from middle row
?	choose a random transition

Table 2.10: Transition Effect Codes

7	6	5	4	3	2	1	0
0	0	1	1	$\langle \text{flash} \rangle$	$\langle \text{blue} \rangle$	$\langle \text{green} \rangle$	$\langle \text{red} \rangle$

Code	Color	Code	Color
0	off	8	off
1	red	9	flashing red
2	green	:	flashing green
3	amber	;	flashing amber
4	blue	<	flashing blue
5	magenta	=	flashing magenta
6	cyan	>	flashing cyan
7	white	?	flashing white

Table 2.11: Color codes for $\langle \text{rgb} \rangle$ parameters

The $\langle \text{rgb} \rangle$ value is a value encoded as defined in Table 2.11. Note that anything drawn in black simply turns off the corresponding pixels.

L—Light Multiple LEDs

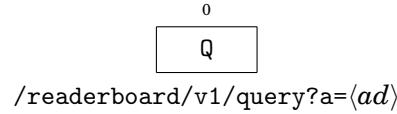
0	1	2	3	...	n	$n + 1$
L	$\langle \text{led}_0 \rangle$	$\langle \text{led}_1 \rangle$	$\langle \text{led}_2 \rangle$...	$\langle \text{led}_{n-1} \rangle$	\$
<code>/readerboard/v1/light?a=$\langle \text{ad} \rangle$&l=$\langle \text{led}_0 \rangle \langle \text{led}_1 \rangle \langle \text{led}_2 \rangle \dots \langle \text{led}_{n-1} \rangle$</code>						

This command is identical to the S command (see below), except that

multiple discrete LEDs can be specified, all of which are illuminated simultaneously. See Table 2.9. Note that if a strobe sequence is running (via a previous * command), it remains running.

The list of $\langle led \rangle$ values is terminated by either a dollar sign (\$) character or an escape byte (hex value 1B), represented in the protocol diagram as “\$”.

Q—Query Readerboard Status



This command causes the sign to send a status report back to the host to indicate the general status of the device, including the discrete LED display which may be queried using the ? command.

This command cannot be addressed to the global address.

The response has the form:

0	1	2	3	4	5	6	7	8
Q	0	$\langle model \rangle$	=	$\langle ad \rangle$	$\langle uspd \rangle$	$\langle rspd \rangle$	$\langle ad_G \rangle$	$\langle EE \rangle$
$\langle spkr \rangle$	\$	V		$\langle hwversion \rangle$	\$	R		$\langle romversion \rangle$
\$	S	$\langle serial \rangle$	\$	L	$\langle led_0 \rangle$	$\langle led_1 \rangle$	$\langle led_2 \rangle$	$\langle led_3 \rangle$
$\langle led_4 \rangle$...	$\langle led_{n-1} \rangle$	\$	F	flasher status (q.v.)			
\$	S	strober status (q.v.)				\$	D	
$\langle dim_m \rangle$		$\langle dim_0 \rangle$...	$\langle dim_n \rangle$		\$	M	
$\langle R coldata_0 \rangle$...	$\langle R coldata_{63} \rangle$	\$	$\langle G coldata_0 \rangle$...	
$\langle G coldata_{63} \rangle$	\$	$\langle B coldata_0 \rangle$...	$\langle B coldata_{63} \rangle$		\$		
$\langle flashdata_0 \rangle$...	$\langle flashdata_{63} \rangle$	\$	\n		

The leading “Q0” indicates the format of the data that follows. If a change is introduced that breaks the parsing rules needed to understand the response, the “0” will be incremented as documented in Table 2.5 for 6-bit integer encoding.

The $\langle model \rangle$ field may be “B” for the stand-alone busylight modules which only include discrete busy status lights, “M” for 64×8 matrix display hardware with monochrome LEDs, or “C” for 64×8 RGB color display boards.

$\langle hwversion \rangle$ and $\langle romversion \rangle$ indicate the versions, respectively, of the hardware the firmware was compiled to drive, and of the firmware itself. Each of these fields are variable-width and conform to the semantic version

standard 2.0.0.⁴ Each is terminated by a dollar-sign (\$) character (and thus those strings may not contain dollar signs).

The $\langle serial \rangle$ field is a variable-width alphanumeric string which was set when the firmware was compiled. It should be a unique serial number for the device (although that depends on some effort on the part of the person compiling the firmware to insert that serial number each time). Serial numbers RB0000–RB0299 are reserved for the original author’s use. This string is also terminated with a dollar sign.

The $\langle EE \rangle$ field is “X” if the device is fitted with an external EEPROM chip or module, “I” if using the EEPROM memory on the Arduino controller board, or “_” if there is no EEPROM memory at all (and thus, configuration settings made with the = command are not persistent).

The $\langle spkr \rangle$ field is “T” if the hardware supports a speaker with full tone output, “S” if the speaker can only emit a single fixed tone, or “_” if no audio output is supported.

The dimmer values appear in a block beginning with “D”. Each dimmer value is a two-digit hex value. $\langle dim_m \rangle$ is the dimmer value for the LED matrix while $\langle dim_n \rangle$ is the dimmer level for discrete status LED # n . The block is terminated with “\$”. These values are given as “_” if the device does not have the corresponding feature (e.g., matrix) or the corresponding LED is not dimmable on that device.

The discrete light, flasher, and strober fields (starting, respectively, with L, F, and S, are as documented above for the ? (busy) command. Busylight units stop here and don’t include the image bitmap data that readerboards report.

The $\langle coldata \rangle$ bytes are sent just as with the I command, as hexadecimal values indicating the LEDs lit on the matrix display, with $\langle coldata_0 \rangle$ being the leftmost column of the display and $\langle coldata_{63} \rangle$ being the rightmost. In each column, the least significant bit indicates the LED on the top row.

Note that monochrome boards only send a single color plane of data plus the flash plane.

The $\langle ad \rangle$, $\langle ad_G \rangle$, $\langle uspd \rangle$, and $\langle rspd \rangle$ values are as last set by the = command (or the factory defaults if they were never changed).

The status message sent to the host is terminated by a newline character (hex byte 0A₁₆), indicated in the protocol description above as “\n”.

For Busylight devices, serial numbers B0000–B0299 are reserved for use by the original author.

M—Morse Code

0	1	2	...	
M	$\langle led \rangle$	$\langle message \rangle$		ESC
/readerboard/v1/morse?a= $\langle ad \rangle$ &t= $\langle message \rangle$ &l= $\langle led \rangle$				

⁴See semver.org.

Sends $\langle message \rangle$ by blinking the designated status LED in Morse code. If $\langle led \rangle$ is “_”, then the message is sent audibly over the speaker.

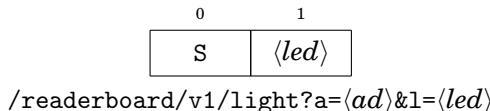
Supported characters include the letters A–Z, digits 0–9, and the symbols !, " , \$, &, ' , (,), +, ,, -, ., /, :, ;, ?, @, and _. At the end of the message, the prosign SK is transmitted (which is also available at codepoint 3).

Supported prosigns include:

Sign	Meaning	Codepoint	
<u>AR</u>	start message	2_{10}	02_{16}
<u>AS</u>	wait	19_{10}	13_{16}
<u>BT</u>	break	30_{10}	$1E_{16}$
<u>DDD</u>	relayed distress	18_{10}	12_{16}
<u>HH</u>	correction	127_{10}	$7F_{16}$
<u>KA/CT</u>	attention	7_{10}	07_{16}
<u>SK</u>	end of message	3_{10}	03_{16}
<u>SOS</u>	distress	17_{10}	11_{16}
<u>VE/SN</u>	verified	6_{10}	06_{16}

This is intended for diagnostic messages or signalling when other, easier to read, methods aren't available. As such, other device operations may be suspended while the message is being sent by Morse code.

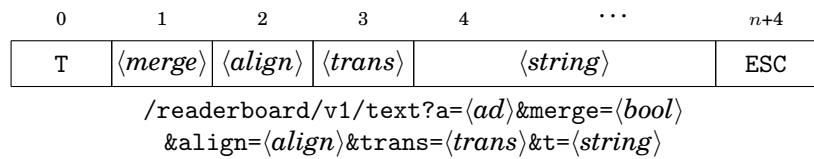
S—Light Single LED



Stops the flasher (canceling any previous F command) and turns off all discrete LEDs. The single LED indicated by $\langle led \rangle$ is turned on. See Table 2.9. Note that if a strobe sequence is running (via a previous * command), it remains running.

This command may be given in upper- or lower-case (“s” or “S”).

T—Display Text



Displays the text $\langle string \rangle$ at the current cursor position. The cursor position is then moved past the text to be ready for the next string to be printed.

Code	Alignment Behavior
.	No alignment (print at cursor)
<	Left (print at left edge of matrix)
>	Right (end of text at right edge of matrix)
~	Center (between left and right edges)
R	Local right (end of text at cursor)
C	Local center right (between cursor and right edge)
L	Local center left (between left edge and cursor)

Table 2.12: Alignment Codes

The $\langle align \rangle$ code indicates how the text is aligned on the display using one of the codes shown in Table 2.12. **Not currently implemented.**

The $\langle trans \rangle$ code specifies the transition effect to be used to add this string to the display as shown in Table 2.10.

The text is rendered in the current font and may contain any 8-bit bytes except as otherwise noted (but avoiding ASCII control codes is wise to be safe from conflict with future control codes which may be added to the protocol). The string is terminated by an escape character (hex byte 1B), indicated in the protocol description as ESC. (Since a dollar sign may appear in $\langle string \rangle$, the terminator must be an escape character for this command.)

The string may include control codes as listed in Table 2.3.

X—Turn off Discrete LEDs

```
0
X
/readerboard/v1/off?a=<ad>
```

Turns off the flasher, strober, and all discrete LEDs.

This command may be given in upper- or lower-case (“x” or “X”).

C H A P T E R



SERVER-ONLY COMMANDS

The enjoyment of one's tools is
an essential ingredient of
successful work.

—Donald E. Knuth

FOR THE MOST PART the server exists as a way to host readerboards and busylights from a central machine while allowing users on other networked machines to post messages to them. It also provides a solution to the lack of an ability for signs to coordinate content with each other. If you use the server's API to tell a set of target signs to change their display contents, that will change whatever the sign was doing to what you are now telling it to do, even if other users had put their own messages on it.

This chapter describes server-side coordination between users in a network environment by providing a set of commands which clients can post to the server to request information to be displayed for them alongside other content other users have posted. It also provides some services to keep boards updated with information known to the server which the devices don't themselves track, such as time of day.

current—Report Current Display State

`/readerboard/v1/current?a=<ad>`

Returns a JSON-formatted report as an object with device IDs as the keys and values matching the result of the busy command, which gives the server's understanding of what should be displayed on the device(s). This

Code	Name	Code	Name
.	none	R	wipe-right
>	scroll-right	U	wipe-up
<	scroll-left	D	wipe-down
^	scroll-up		wipe-horiz
v	scroll-down	-	wipe-vert
L	wipe-left	?	random

Table 3.1: Transition codes for use in post command

Code	Name	Code	Name
.	none	C	local-center-right
>	right	L	local-center-left
<	left	R	local-right
~	center		

Table 3.2: Alignment codes for use in post command

differs from busy in that this does not go through the time and effort to poll each device to ask about the hardware status. It merely reports what the server last told the devices to display.

post—Post a Message to the Displays

```
/readerboard/v1/post?a=<ad>&t=<text>&id=<id>&trans=<trans>
&until=<datetime>&hold=<duration>&color=<rgb>&visible=<duration>
&show=<duration>&repeat=<duration>
```

Adds a text message to the display list. The message may include the special tokens listed in Table 3.4. The server will maintain the list of posted images and will update the devices as needed to display dynamic content.

If any lower-level display matrix commands are received that directly manipulate the matrix, the display of posted messages is suspended in favor of the directly set content. When a clear command is received, then display of the posted message list will resume.

The *<id>* value must be unique but the server doesn't assign these in order to keep the API simple. Thus, it is recommended that you assign these as UUIDs or other scheme to keep them from colliding.

If trans is given, this message will be introduced using the specified transition effect, specified by name or number (see Table 3.1).

If until is given, the message will automatically unpost when the specified date and time is reached. (*Datetime*) has one of the formats:

Code	Name	Code	Name
0	off	8	flashing-off*
1	red	9	flashing-red
2	green	10 or :	flashing-green
3	amber/yellow	11 or ;	flashing-amber/flashing-yellow
4	blue	12 or <	flashing-blue
5	magenta	13 or =	flashing-magenta
6	cyan	14 or >	flashing-cyan
7	white	15 or ?	flashing-white

*Included for technical completeness but actually using this code would be silly.

Table 3.3: Color codes for use with the post command

$$\begin{aligned} & [\langle day \rangle @ [\langle hour \rangle] : \langle min \rangle [: \langle sec \rangle] \\ & \quad \langle day \rangle \\ & \quad \langle duration \rangle \end{aligned}$$

In the first form, an absolute time is specified for the message’s expiration. In the second form, a day of the week is specified without a time (defaulting the time to 00:00:00), and in the third form, a relative duration since posting is given.

If $\langle day \rangle$ is specified, it gives the day of the week. $\langle Hour \rangle$ and $\langle sec \rangle$ default to the current hour and 00 respectively, so “until=monday” shows the message until the beginning of Monday; “until=tuesday@12:00” expires at noon on Tuesday, “until=:30” expires after half-past the current hour, “until=17:00” expires at 5:00 PM, and “until=30m” expires after 30 minutes have passed since posting.

If hold is given, the message will remain on-screen for the specified time duration before moving on to the next posted message.

If color is given, the current color will be changed to the specified value before writing the message; otherwise (or additionally) color codes may be added in-line in the message text. (See Table 3.3 for the codes you can use here.)

The show value gives a time duration during which the message will be visible on the sign. After this time has elapsed, it will be hidden (but not necessarily removed from the queue). If the repeat value is also specified, then the message will reappear after the specified duration has elapsed since it was hidden.

The visible parameter controls absolute time ranges outside of which the message will be unconditionally hidden. Hidden messages automatically reappear at the start of any of these visible time ranges. The value is a pair of date strings in any of the forms allowed for until, separated by a hyphen (‘-’). Multiple such pairs may appear, separated by commas (‘,’).

Code	Description
{big}	Display letters and numbers in large font
{color<n>}	Switch to color number <n> (as ^K<n>)
{<colorname>}	Switch to color by name
{date}	Date as “<dd>-<mmm>-<yyyy>”
{font<n>}	Switch to font number <n> (as ^F<n>)
{normal}	Stop displaying numbers in large font
{mdy}	Date as “<mm>/<dd>/<yyyy>”
{pom}	Phase of the moon (as a symbol)
{time12}	Time as “<hh>:<mm> AM/PM” on 12-hour clock*
{time24}	Time as “<hh>:<mm>” on 24-hour clock†
{ymd}	Date as “<yyyy>-<mm>-<dd>”
{@[±]<n>}	Reposition cursor to column <n>‡
{#<n>}	Insert character with codepoint <n> (decimal)
{#\$<h>}	Insert character with codepoint <h> (hex)
{\$<name>}	User variable <name>
{}	Literal “{”
}	Literal “}”

*Hour is space-padded (e.g., “13:15”)

†Hour is zero-padded (e.g., “03:15”)

‡-<n> and +<n> move left or right by <n> columns

Table 3.4: Special Codes in Post Messages

postlist—Report of Message Queue

/readerboard/v1/postlist?a=<ad>&id=<id>

Returns a JSON-formatted list of the messages in the display queue. If <ad> is missing or is the global address, the messages for all devices will be returned; otherwise the list is filtered to include only messages targetted at the listed devices.

If id is given, only that message is included in the report; if the <id> value starts with a slash (/) then the remaining characters are interpreted as a regular expression. Only messages whose <id> values match the regular expression are included in the report.

unpost—Remove a Posted Message

/readerboard/v1/unpost?a=<ad>&id=<id>

Remove the message with the given <id> from the display list. If <ad> is missing or is the global address, the message is removed for all devices. Otherwise, it is only removed for the listed target devices.

If the $\langle id \rangle$ value starts with a slash ('/') then the remaining characters are interpreted as a regular expression. All messages whose $\langle id \rangle$ values match the regular expression are unposted. Thus, "id=/.*" deletes the entire message queue.

update—Update Dynamic Content

```
/readerboard/v1/update?<name0>=<value0>&<name1>=<value1>...
```

For each $\langle name \rangle$ and $\langle value \rangle$ pair, define or update the corresponding user variable (usable in posted messages as `{$<name>}`). The updated value will be displayed from this point until it is updated again.

CHAPTER

4

PINOUTS

The nicest thing about standards is that there are so many of them to choose from.

—Andrew S. Tanenbaum

THIS CHAPTER DESCRIBES each connector used by the readerboard devices and what signals are present on which pins. Note that this begins with the current hardware revision, which is most likely the only part you need to know about. Following that are the connector descriptions for older, legacy versions of the hardware. Be careful to know which version you need to refer to.

8p8c RS-485 Connectors

There is no universal standard for RS-485 jack wiring, but often uses an 8p8c or 6p6c modular jack.¹

A few popular *de facto* standards are in use, but mostly everyone either uses discrete screw terminal blocks or makes up their own connector wiring arrangement.

¹Commonly, but not *entirely* correctly, referred to as “RJ-45” and “RJ-11” jacks. The “registered jack” standards RJ-45 and RJ-11 refer to the wiring arrangement of signals in a modular jack, not just the shape of the jack itself.

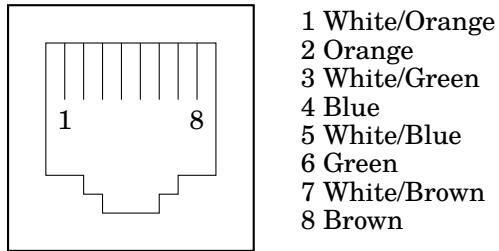


Figure 4.1: Female 8p8c Modular Jack with T-568B Color Codes

Lumos Wiring Pinout

The author's personal wiring standard used by all his RS-485-capable projects, starting with Lumos, supports full-duplex communication (i.e., a data pair that is dedicated to signals from the host PC out to all the connected units—but which could reverse direction if used with half-duplex devices—and a return data pair dedicated to signals from the networked devices back to the host) and a cable-check pair which can be used to power very small devices such as the Busylight or to verify that the entire set of daisy-chained devices are all fully connected and terminated.

The pinout is as follows:

Pin	Signal
1	Return data A (+)
2	Return data B (-)
3	Cable check send (outbound +9–16 V)
4	Data B (-)
5	Data A (+)
6	Cable check return
7	Ground
8	Ground

9-pin Connectors

Some applications use a DE-9 connector² for RS-485 signals, with pinouts such as:

²Also commonly, but incorrectly, called “DB-9” connectors.

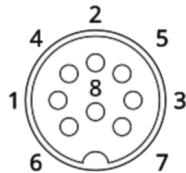


Image: adapted from original by Ancillalover, Wikimedia Commons, CC-BY-SA 4.0 int'l.

Figure 4.2: DIN-8 Jack (looking into socket from front)

Pin	Signal
1	Ground
2	Clear to Send (CTS) +
3	Request to Send (RTS) +
4	Received Data (RxD) +
5	Received Data (RxD) -
6	Clear to Send (CTS) -
7	Request to Send (RTS) -
8	Transmitted Data (TxD) +
9	Transmitted Data (TxD) -

DIN-8 Busylight Tree Connectors

The Busylight control boards use a DIN-8 jack where the light tree cable plugs in to carry the light-signal voltages out to the light tree itself. The pinout used is:

Pin	Tier	Light Color	Wire Color
1	6	blue	blue
2	3	amber	white/brown
3	5	red	white/orange
4	4	red	orange
5	7	white	white/blue
6	2	amber	brown
7		+5 V power	green
8	1	green	white/green

Connectors for Version 3.4 Boards

Arduino Bus Connectors [J0]

Two parallel rows of stacking pins, for a total of 50 pins, extend out of the back of the readerboard PCB. An Arduino Mega 2560 or Due microcontroller board is connected to these pins.

The USB connector on the Arduino board may be used to directly connect the readerboard to a PC for initial configuration and/or to use it as a directly-connected singleton device. (If there are multiple devices in use, it may be preferable to switch to the RS-485 network after initial configuration of the device.)

Version 3.4.1 Boards

The external connections changed significantly at revision 3.3.1. Now J1 is a pair of push-in terminal blocks with a combined total of 16 wire connections to make it easy to connect a pair of CAT6 cables (cable “A” which carries the incoming data signals, and cable “B” which carries the same signals out to the next daisy-chained unit on the RS-485 serial network). The expanded terminal block eliminates the need to make off-board connections or splices between the cables.

The terminals are labeled as “A1” (cable A, pin 1), “B2” (cable B, pin 2) and so forth:

Pin	Wire	Signal	Pin	Wire	Signal
1	A1	Return Data A(+) In*	9	A7	Ground In
2	B1	Return Data A(+) Out*	10	B7	Ground Out
3	A2	Return Data B(−) In*	11	A8	Ground In
4	B2	Return Data B(−) Out*	12	B8	Ground Out
5	A3	Cable Check Send + In*	13	A5	Data A(+) In
6	B3	Cable Check Send + Out*	14	B5	Data A(+) Out
7	A6	Cable Check Return In*	15	A4	Data B(−) In
8	B6	Cable Check Return Out*	16	B4	Data B(−) Out

*Signal not used by the readerboard unit (passed through).

If this device is the last in the RS-485 network chain, insert a 120Ω resistor between the A and B terminals that would have been used as the output if there had been another device connected there (e.g., B4 and B5). This properly terminates the RS-485 network at that point.

The power supply to the readerboard unit is on terminal block J5, with +9 V DC and ground connections as marked on the board.

J6 allows the connection of an external speaker, driven by a simple amplifier formed by C13, R54, and Q24.

Version 3.3.0 Boards

The previous revision, 3.3.0, used a single terminal block J1 to carry the power supply and the RS-485 signals we actually use, leaving the other wires of the serial cables to be connected externally.

The single external connector on version 3.3.0 boards is an 8-pin screw terminal block. This accepts a +9 V DC power input and ground on pins 8

and 7 respectively, which powers the entire board and the attached Arduino controller. If RS-485 communications will be used, the incoming signal is received on pins 1, 2, and 6 while the outgoing signal is on pins 3, 4, and 5. (In actuality, the “input” and “output” sense is arbitrary and either set of A and B signals may be used as input or output.)

If this device is the last in the RS-485 network chain, insert a 120Ω resistor between the A and B terminals that would have been used as the output if there had been another device connected there. This properly terminates the RS-485 network at that point.

Pin	Signal	Pin	Signal
1	A (Data in +)	5	GND
2	B (Data in -)	6	GND
3	A (Data out +)	7	GND
4	B (Data out -)	8	+9 V DC in

The pinout of J1 for revision 3.2.2 boards is different:

Pin	Signal	Pin	Signal
1	A (Data in +)	5	GND
2	B (Data in -)	6	GND
3	+9 V DC in	7	A (Data out +)
4	GND	8	B (Data out -)

Connectors for Legacy Boards

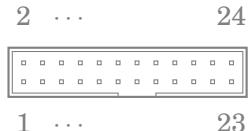
Connectors for Version 2.1 Boards

Power / RS-485 (8-pin terminal) [J0]

This is wired the same as J1 on the revision 3.2 boards.

Earlier Boards

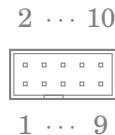
Matrix Control (24-pin ribbon cable) [J1]



For readerboards before version 2.1.0, this 24-position ribbon cable carries signals to directly drive the 64×8 LED matrix. The other end of this cable mates with J0 on the shield board. The pinout of its IDC header is:

1	D6	7	RCLK	13	+5 V DC	19	Gnd
2	D5	8	D2	14	+5 V DC	20	Gnd
3	D7	9	\bar{G}	15	+5 V DC	21	R2
4	D4	10	D1	16	+5 V DC	22	REN
5	SRCLK	11	SRCLR	17	Gnd	23	R1
6	D3	12	D0	18	Gnd	24	R0

Discrete LEDs (10-pin ribbon cable) [J2]



For readerboards before version 2.1.0, this 10-position ribbon cable carries signals to directly drive the 64×8 LED matrix. The other end of the cable mates with J1 on the shield board. The pinout of its IDC header is:

1	GND	6	L6
2	GND	7	L2
3	L0	8	L5
4	L7	9	L3
5	L1	10	L4

Board Power (3-pin screw terminal) [J0]

This 3-position screw terminal block provides power to the readerboard. Note that the +5 V supply is also connected to pins 13–16, and Ground to pins 17–20 of J1, the only power required here is the +9 V input that drives the LEDs themselves.

1	+5 V DC in
2	GND
3	+9 V DC in

Shield Power (5-pin screw terminal) [Shield J2]

This 5-position screw terminal block accepts incoming +9 V DC power and ground on pins 4 and 3 respectively. It then provides +5 V DC, +9 V DC, and ground outputs on pins 1, 2, and 5 respectively to supply power to the main display board.

1	+5 V DC out
2	GND
3	GND
4	+9 V DC in
5	+9 V DC out

Shield RS-485 (6-pin screw terminal) [Shield J4]

This 6-position screw terminal block accepts incoming RS-485 signals A, B, and ground on pins 2, 1, and 3 respectively, and outputs the network signals A, B, and ground on pins 6, 5, and 4 respectively, to go on to the next device in the chain. If this is the last device, then nothing should be connected to pins 5 and 6. Instead, install jumper J5 which connects a 120Ω resistor across those terminals to terminate the network at that point.

- 1 B (Data In -)
- 2 A (Data In +)
- 3 GND
- 4 GND
- 5 B (Data Out -)
- 6 A (Data Out +)

C H A P T E R



INSTALLING AND USING THE READERBOARD AND BUSYLIGHT UNITS

See, unlike most hackers, I get little joy out of figuring out how to install the latest toy.

—Jamie Zawinski

Power

The readerboard units require an external 9V DC power supply in order to sustain the power requirements of all of the LEDs it uses. Always be sure that this power supply is connected and operating before plugging in the readerboard to USB. Otherwise, you may overload the USB port on the host computer and/or the Arduino microcontroller as it tries to power the readerboard by itself.

Similarly, busylight units are designed to be powered from the USB connection *or* from an external power source (such as a pair of power wires on the CAT6 cable used for RS-485 signals) but not both at the same time.

Readerboard PCB revisions starting with 3.4.1 introduced diode D520 to mitigate this issue, so the LEDs don't end up being powered by the Arduino. Busylight PCB revisions starting with 2.2.0 did the same with the introduction of D0.

Standalone Units**Networked Units****Compiling the Supporting Software****Setting up rbsrvr****Setting up Per-User Utilities**

calmond

micmond

Manual Status Updates**Diagnostic Displays**

There are a few situations where the readerboards and busylights use their light displays to convey information about their own status.

EEPROM Access

When a readerboard accesses its EEPROM, a small superscripted “EE” is displayed in the upper-right corner of the matrix. The next update to the displayed matrix content will overwrite this indicator. The color of the indicator indicates the type or outcome of the operation:

RGB Units	Monochrome Units	Meaning
green	on	reading
yellow	on	writing
flashing red	flashing	error

Protocol Errors

If the device receives garbled or otherwise incorrect commands over either USB or RS-485, it lights the top discrete status LED and alternately strobos the fourth and fifth from the bottom. On units with our standard color scheme, this is the white and both red LEDs.

These will be displayed until the next status LED command changes them.

POST

During power-on self-test when a readerboard is turned on, the top (usually white) LED is illuminated until the POST operation is completed.

C H A P T E R

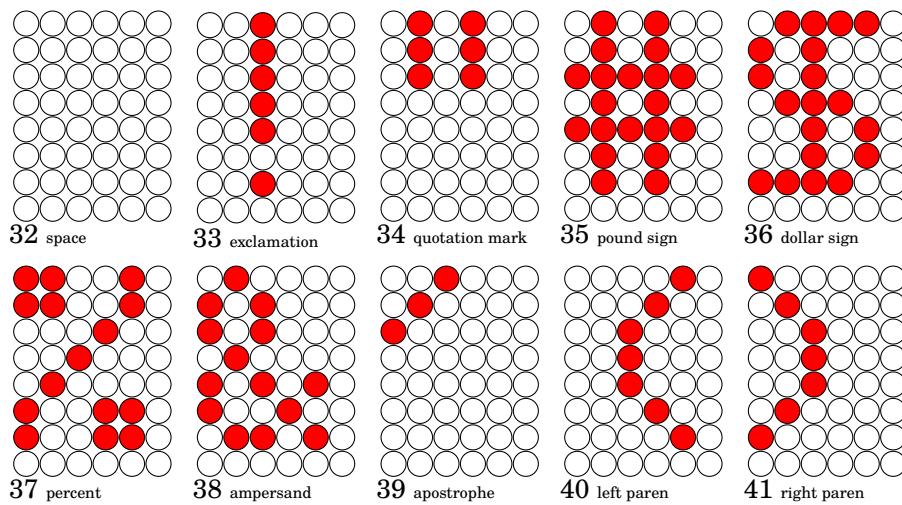
6

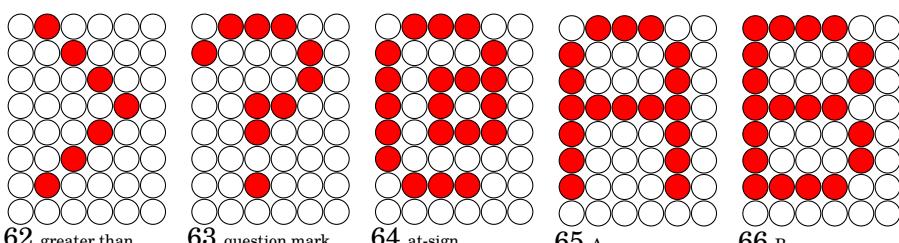
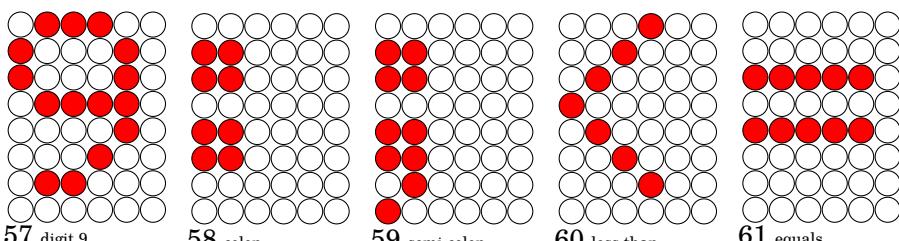
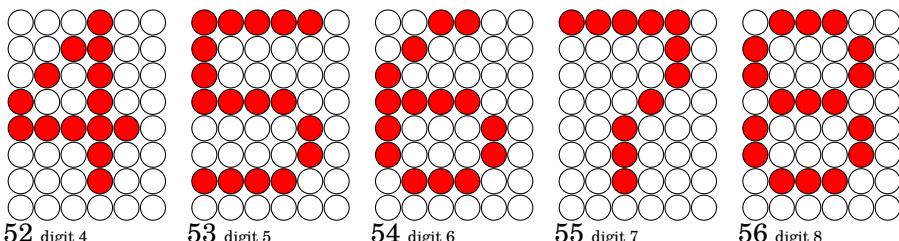
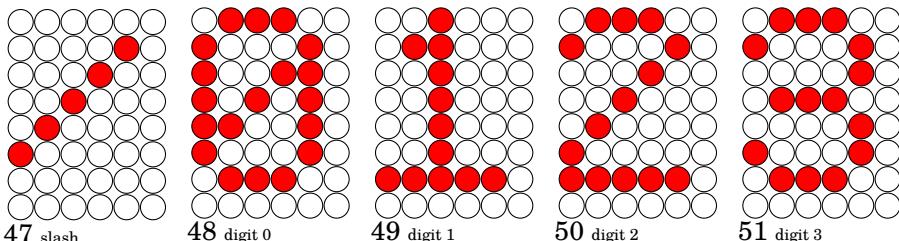
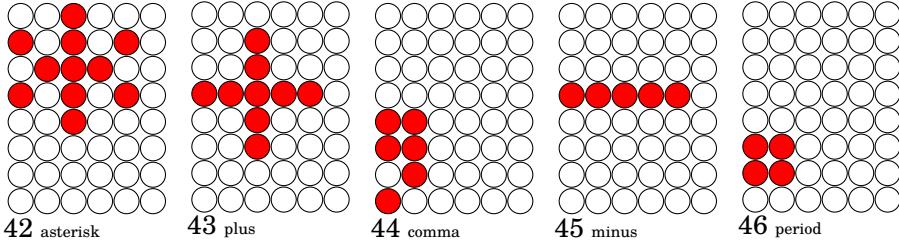
FONT GLYPHS

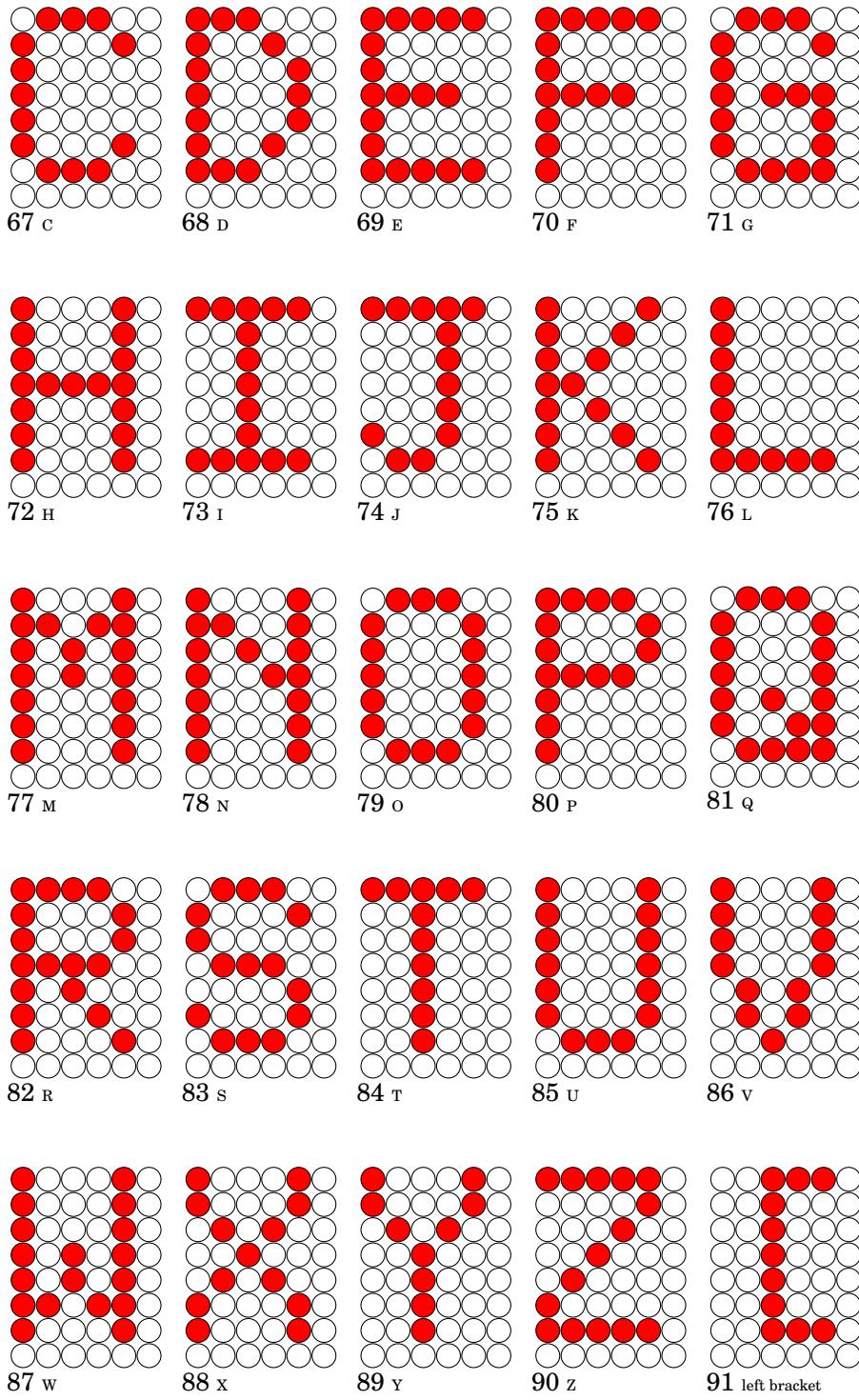
Helvetica is the sweatpants of typefaces.

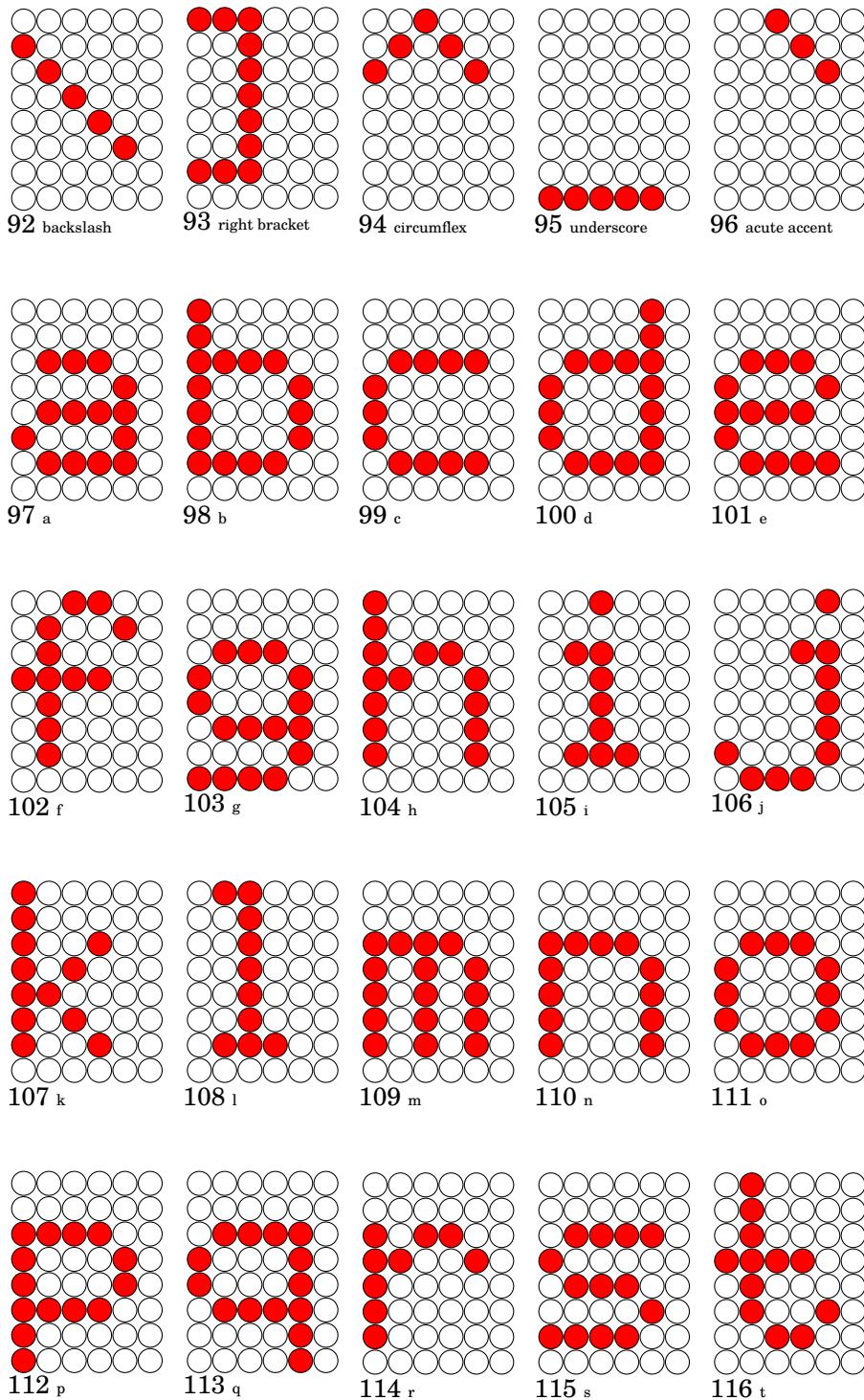
—John Boardley

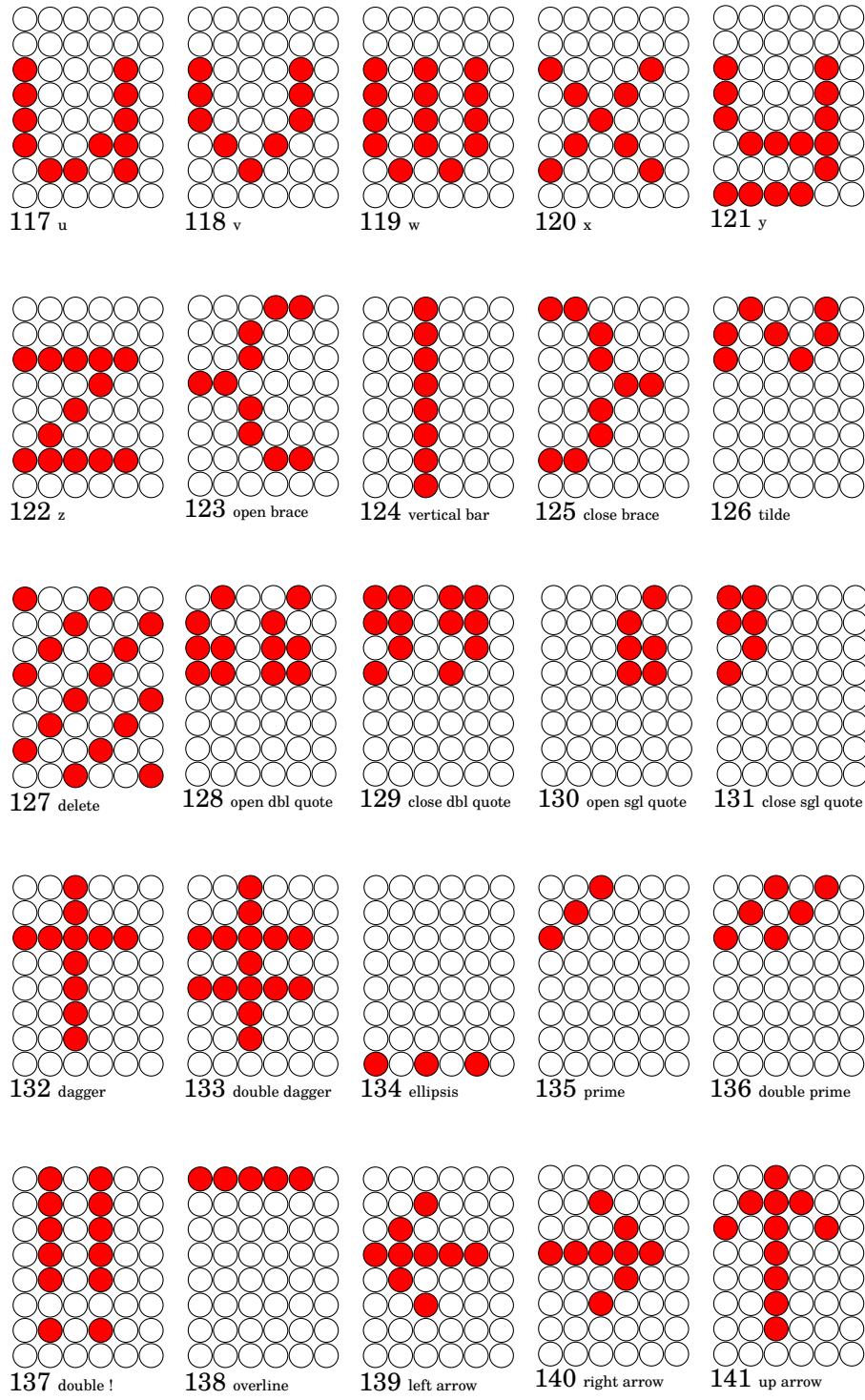
Font #0 standard.font

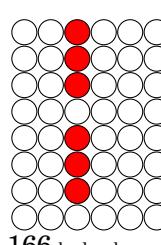
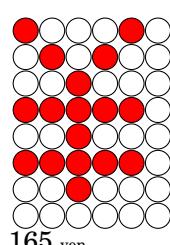
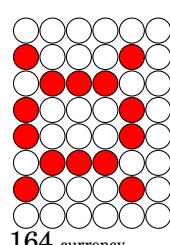
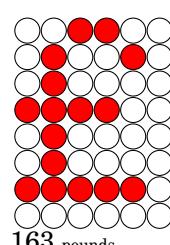
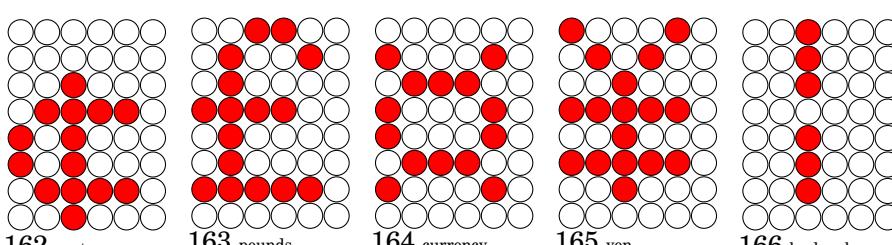
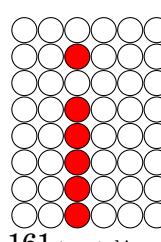
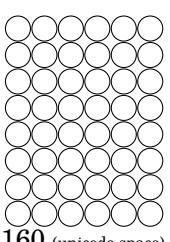
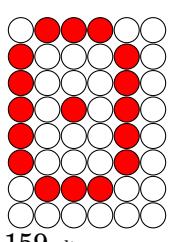
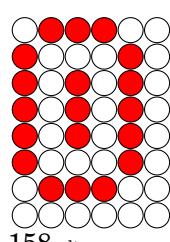
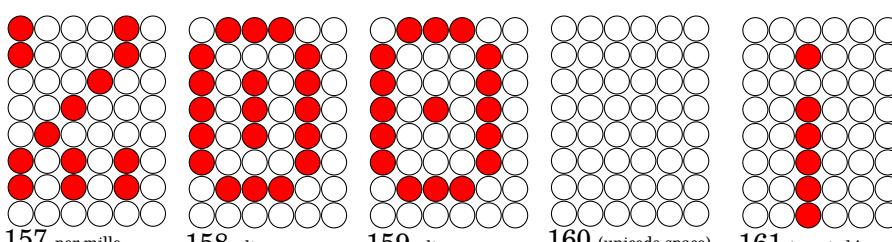
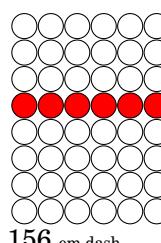
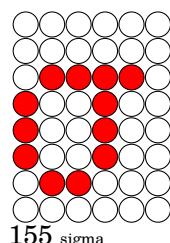
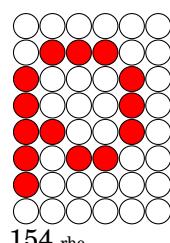
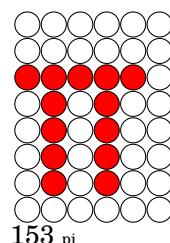
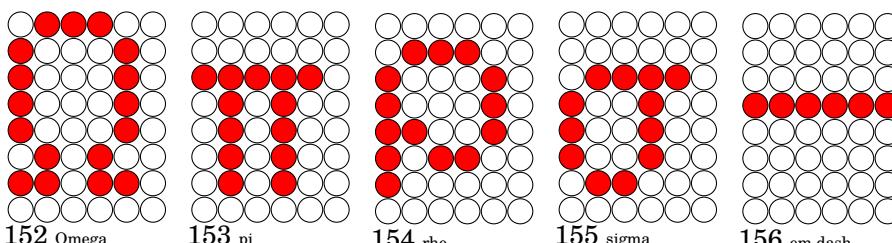
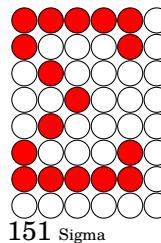
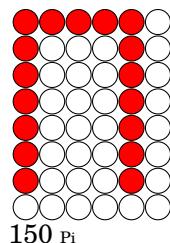
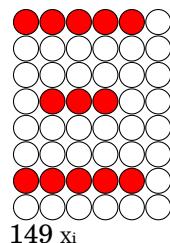
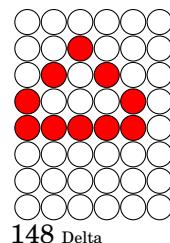
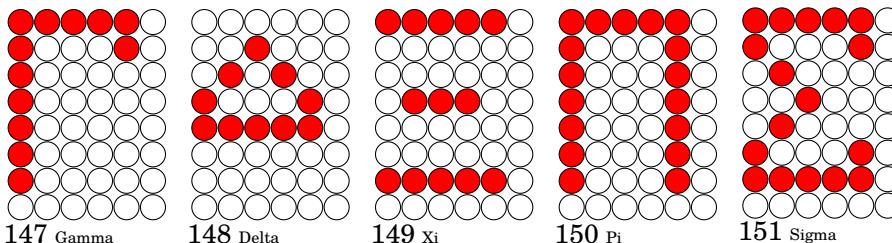
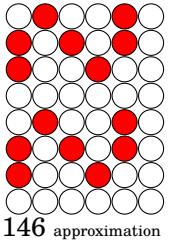
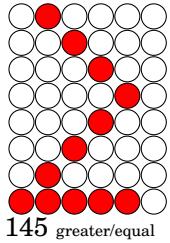
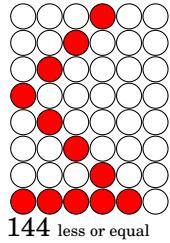
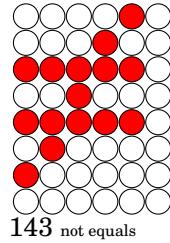
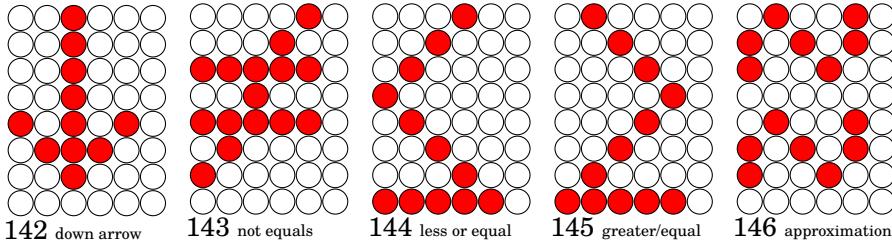


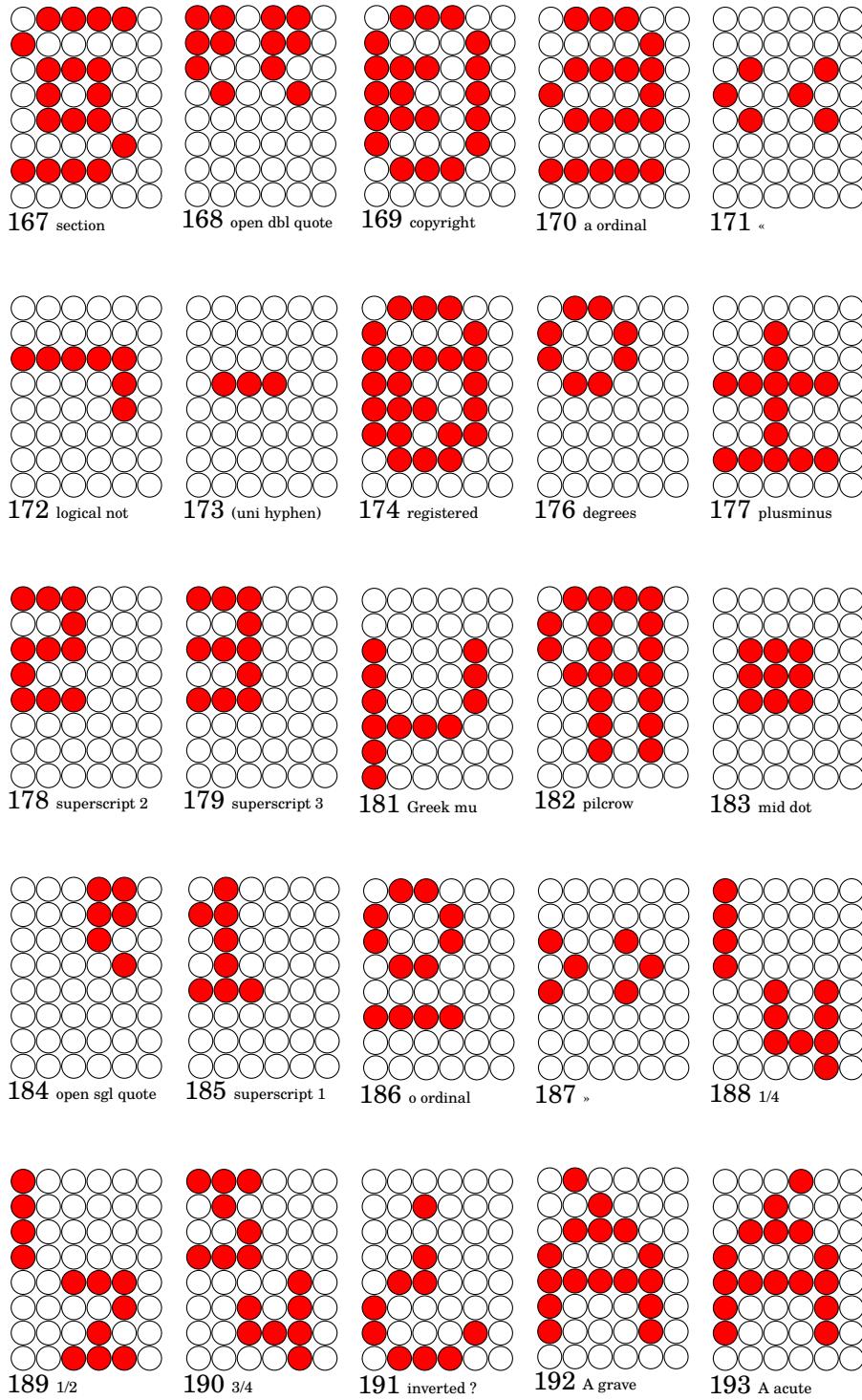


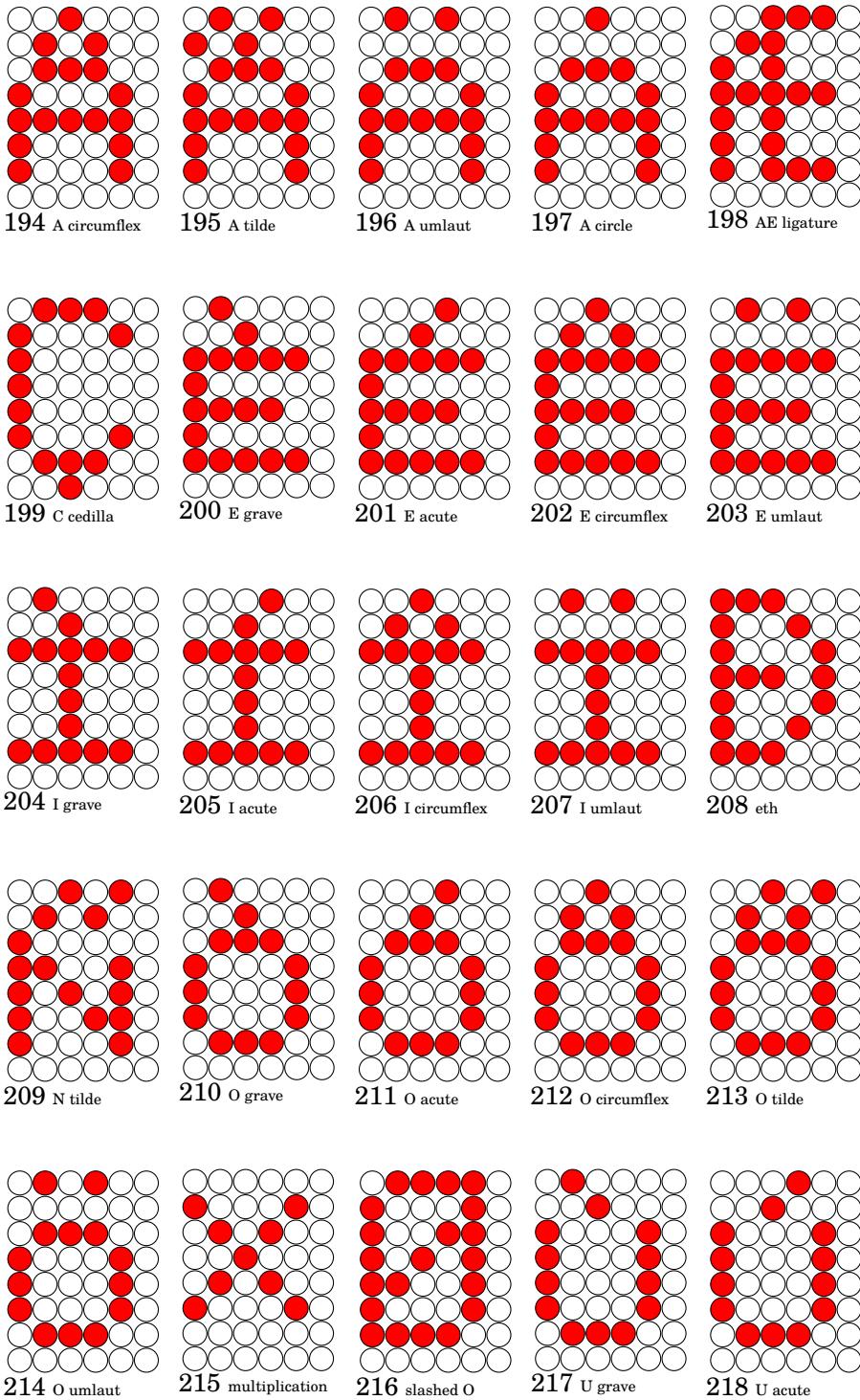


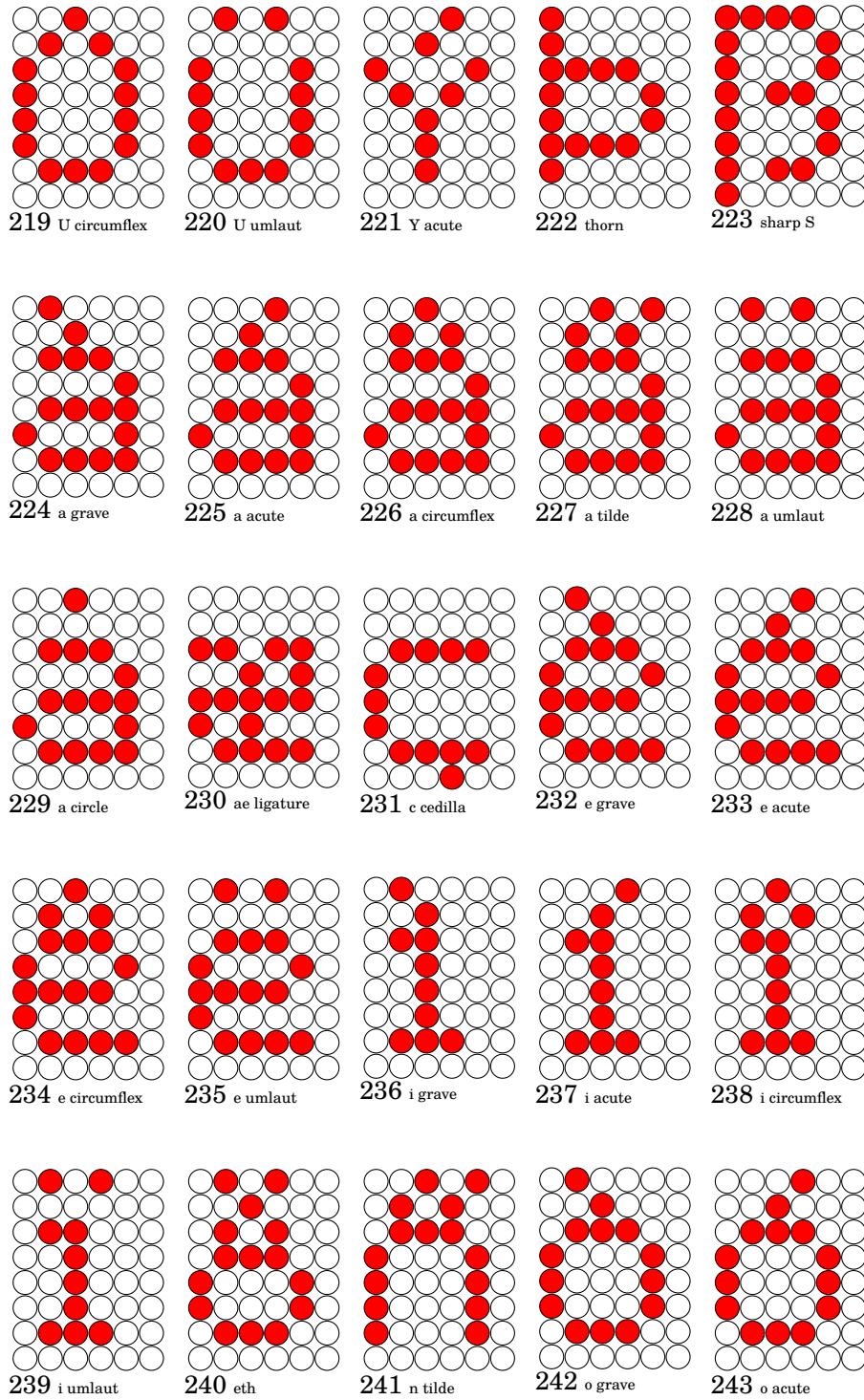


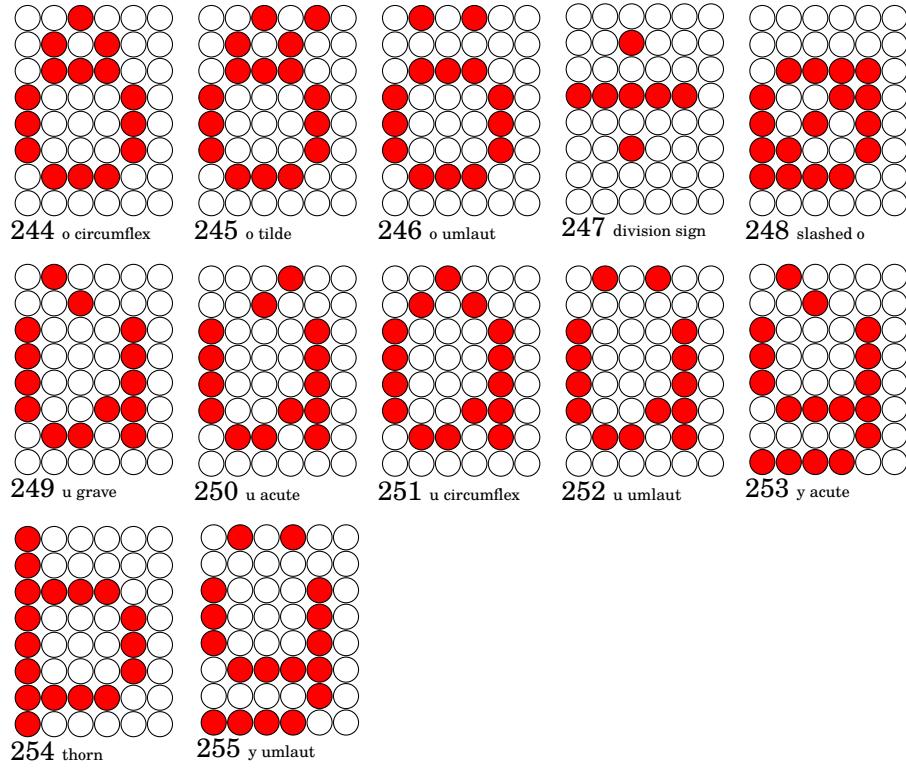
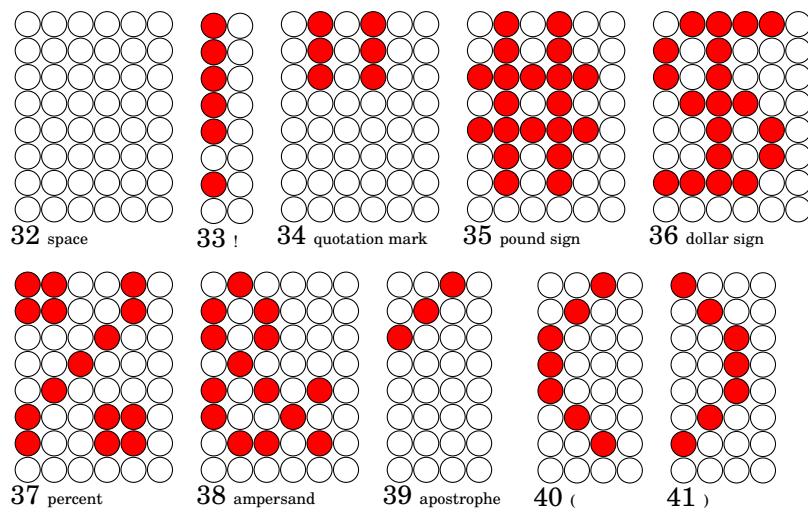


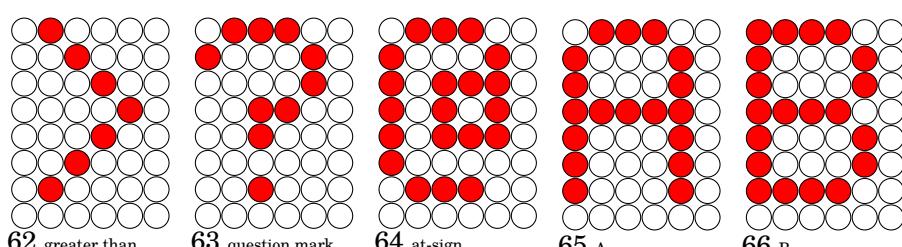
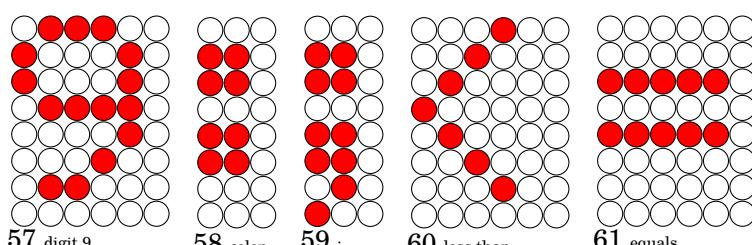
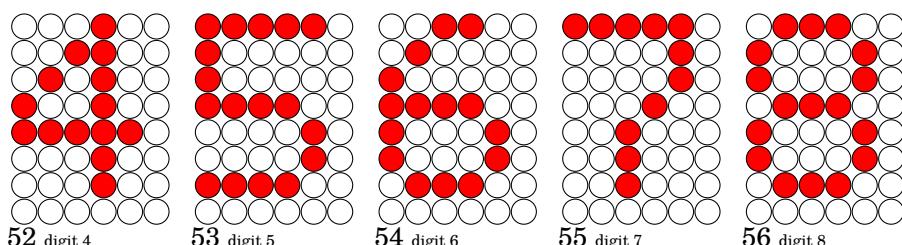
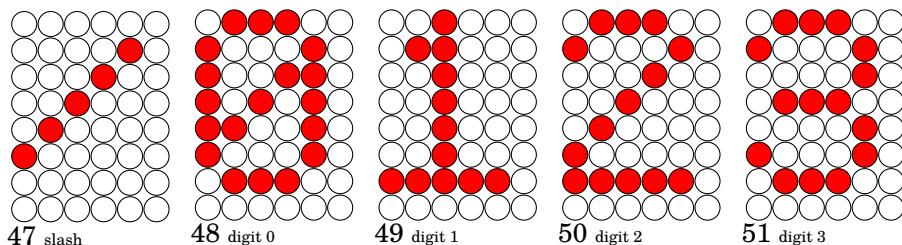
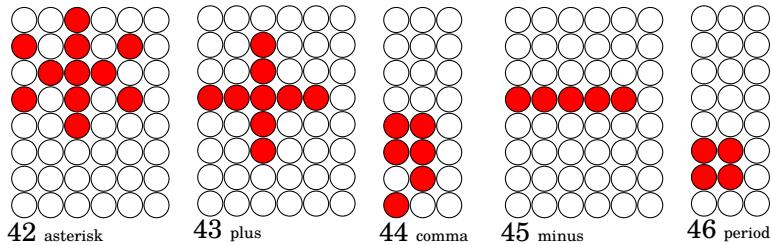


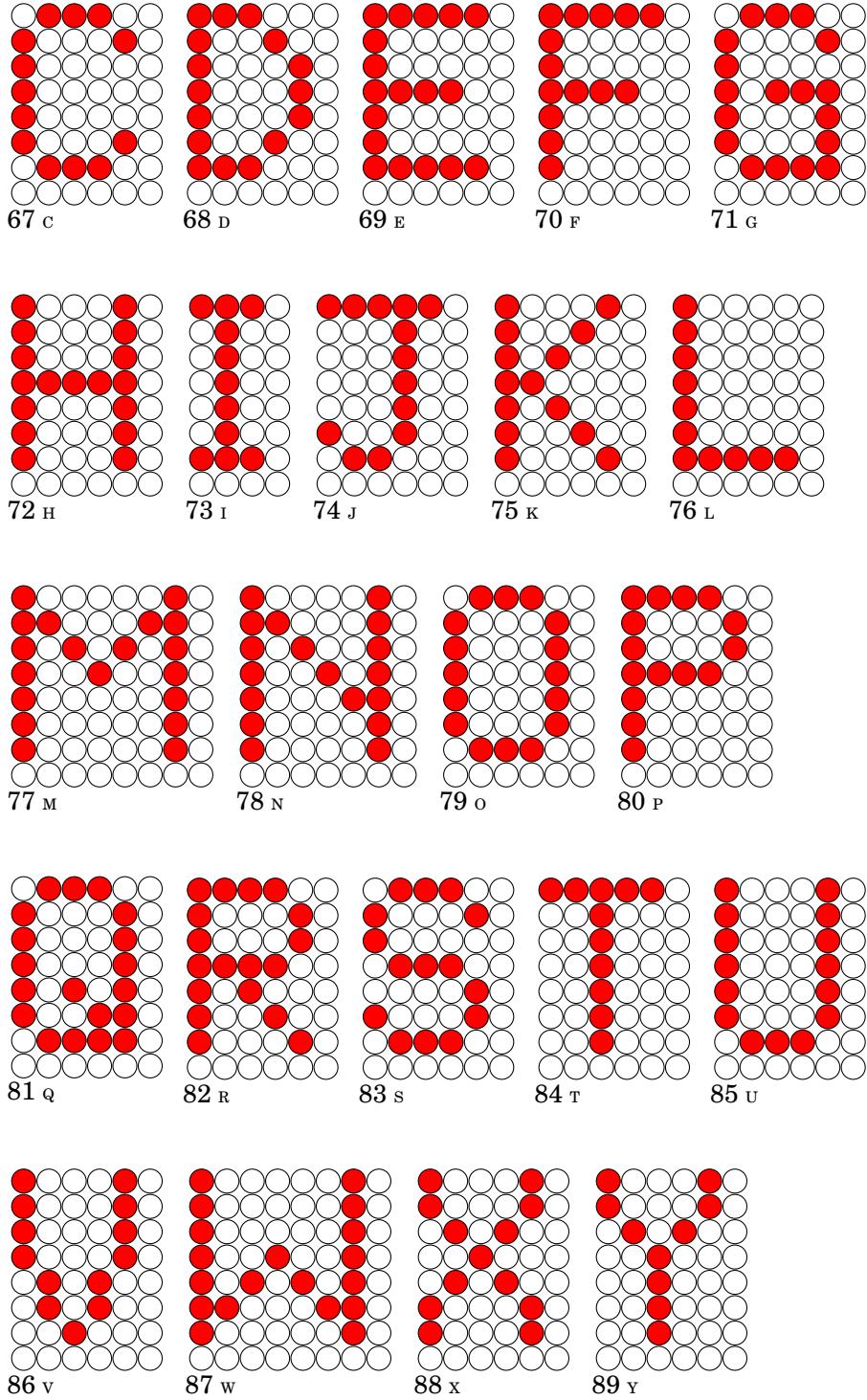


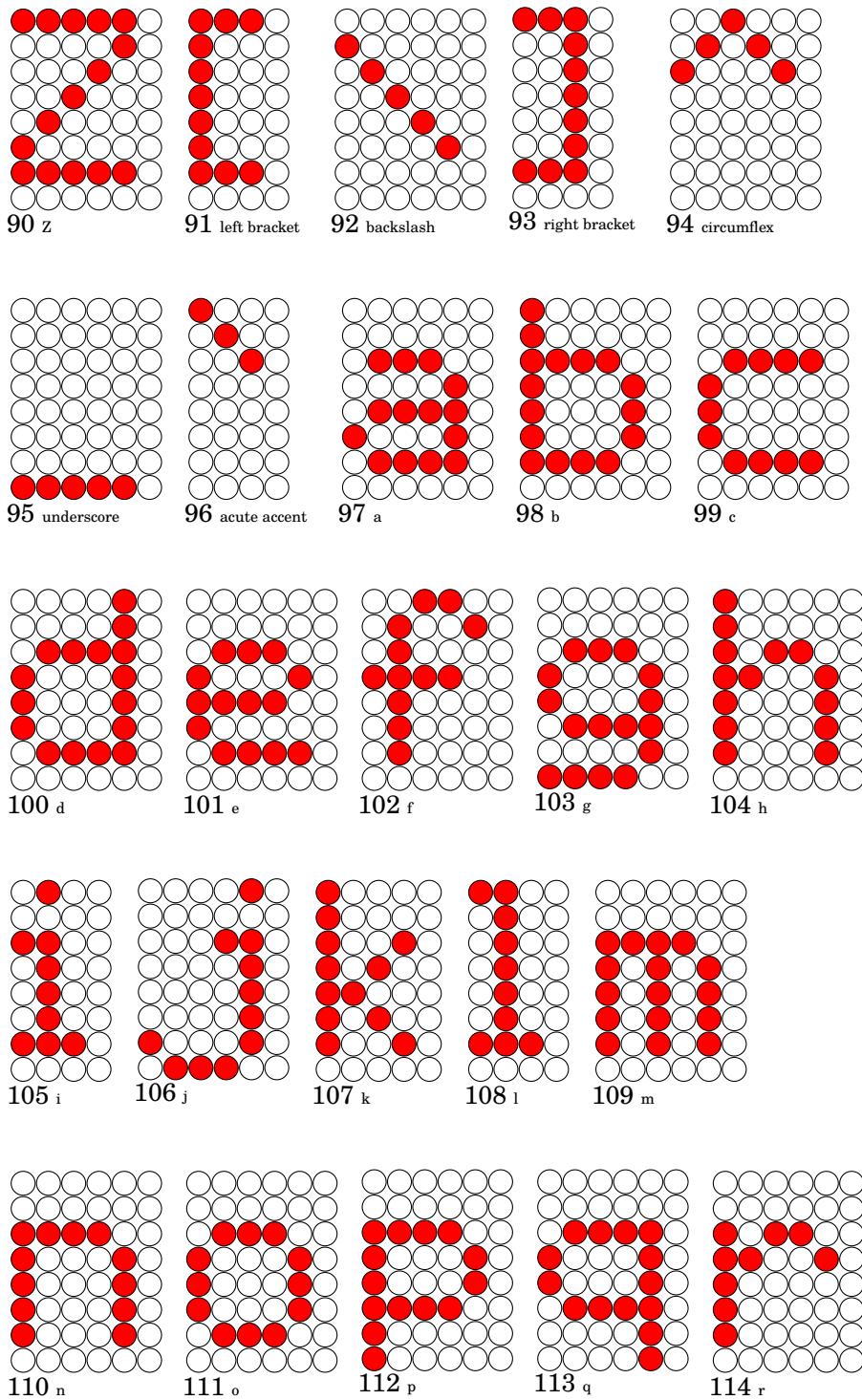


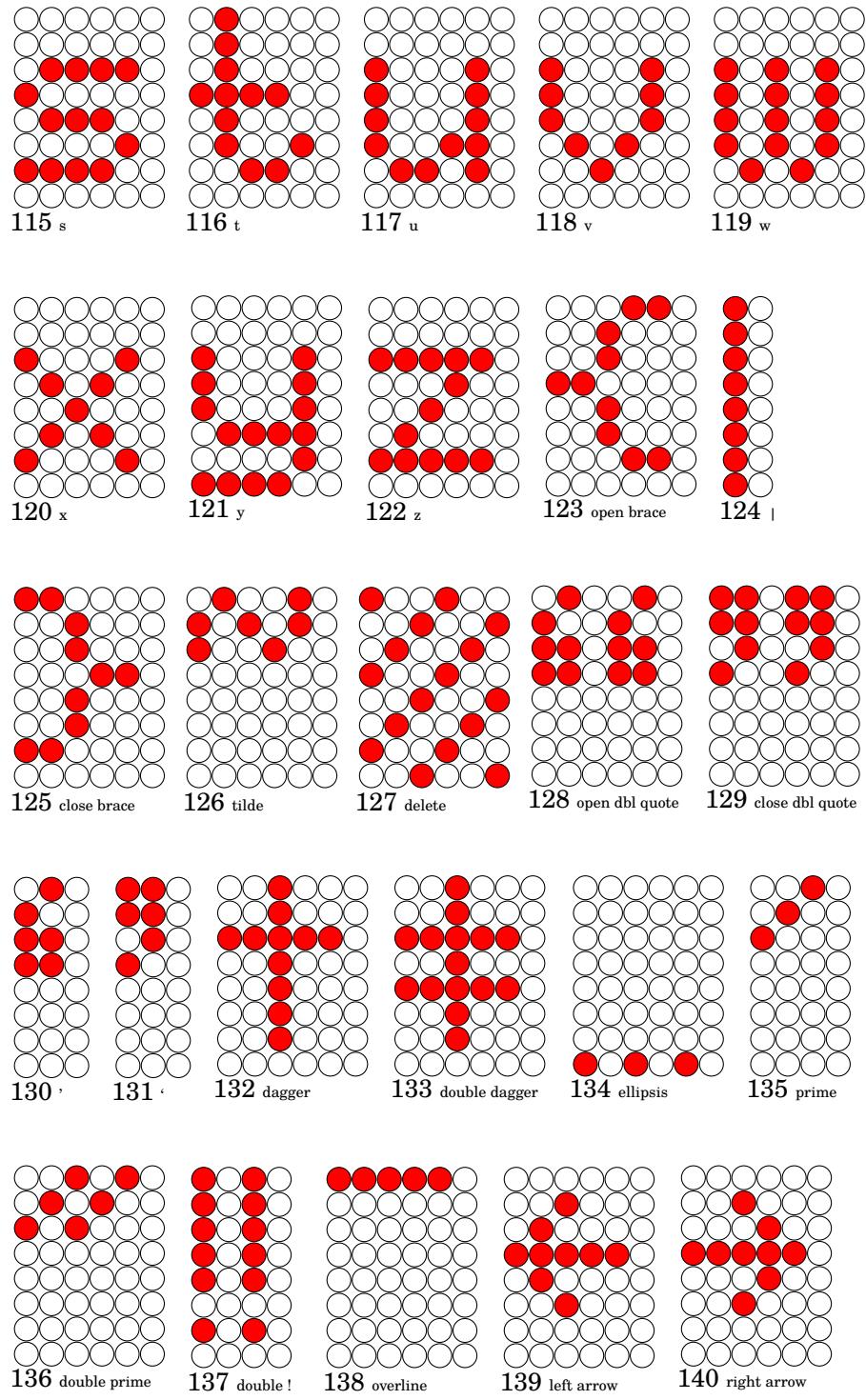


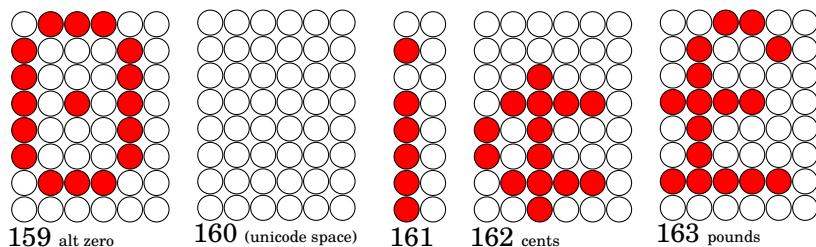
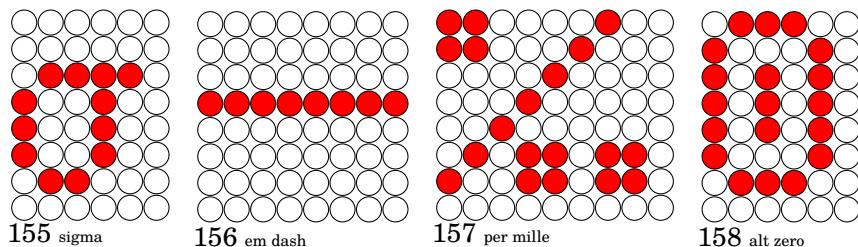
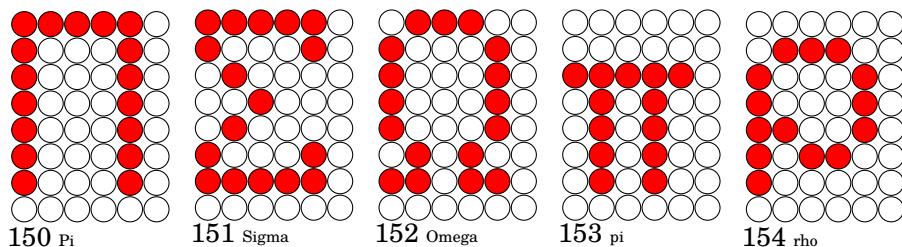
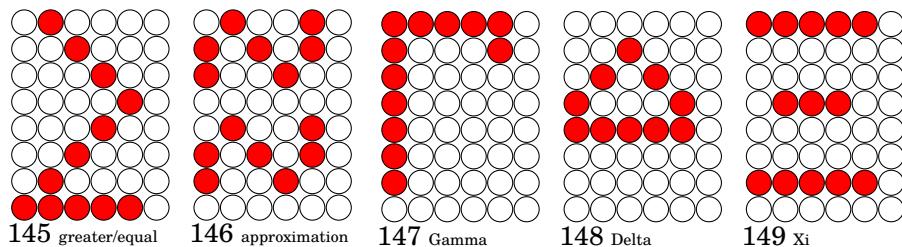
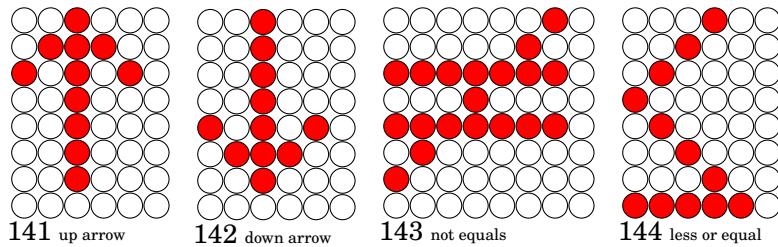
**Font #1 standard_variable.font**

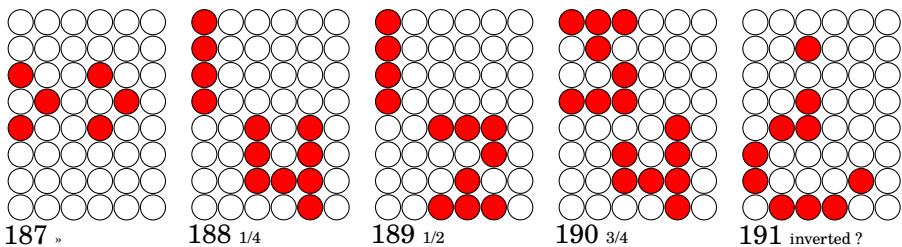
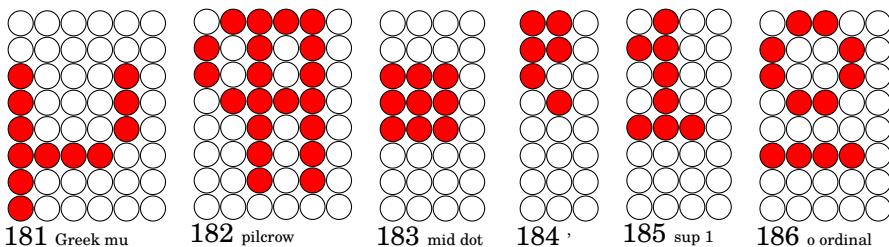
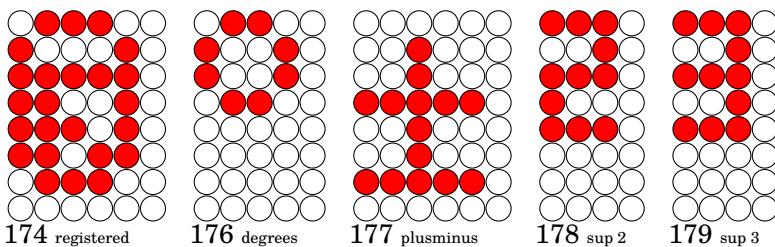
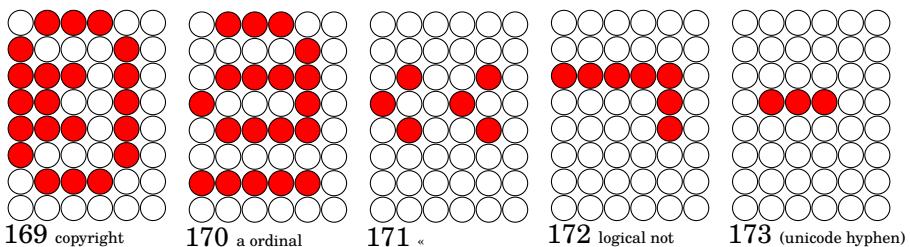
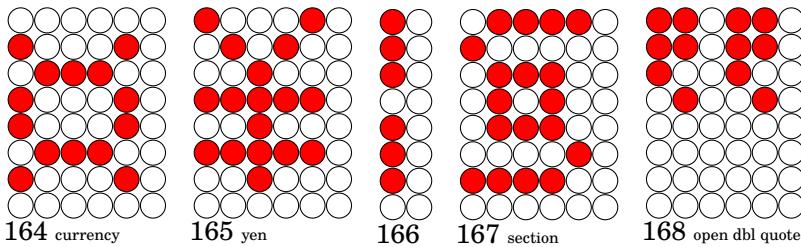


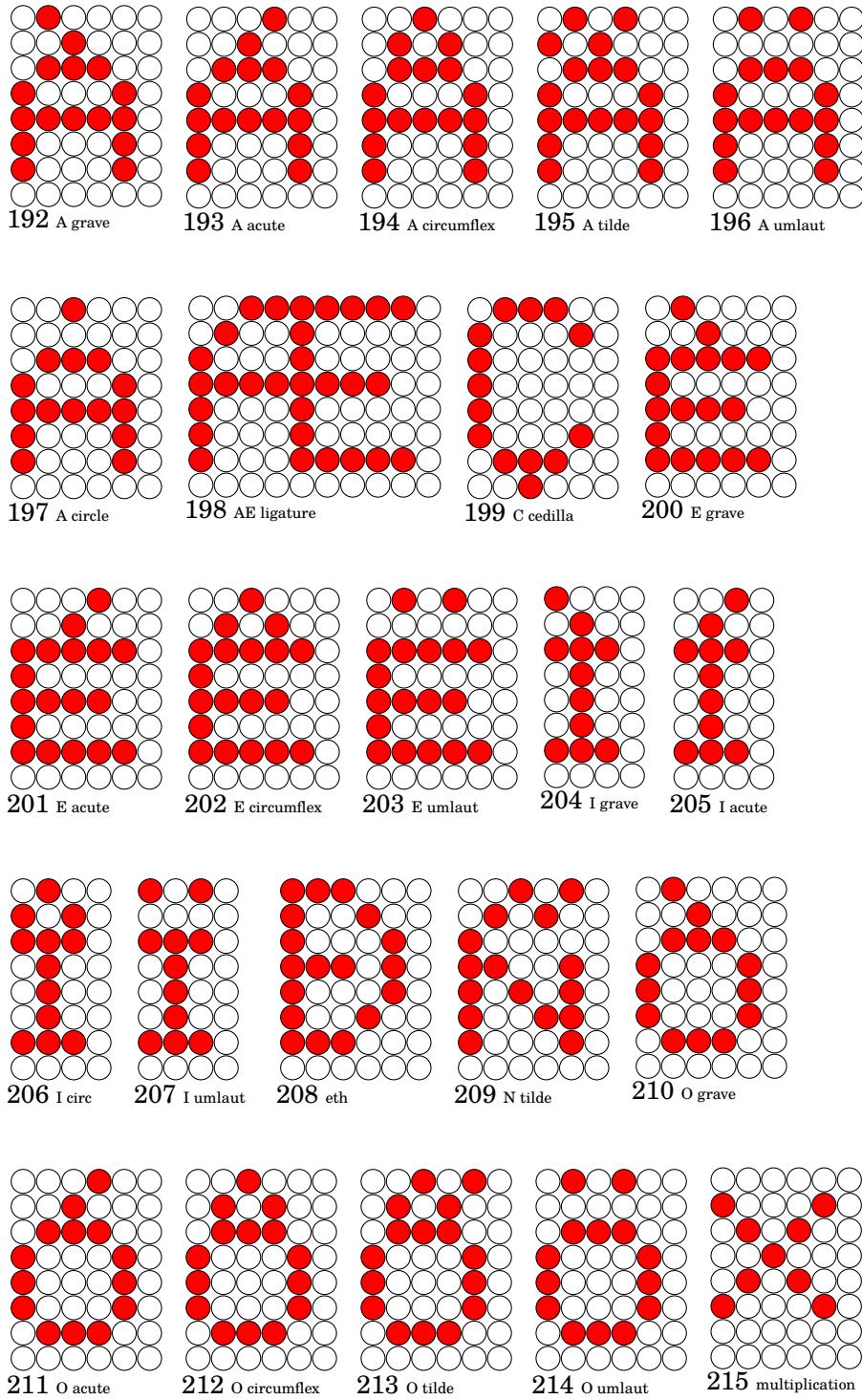


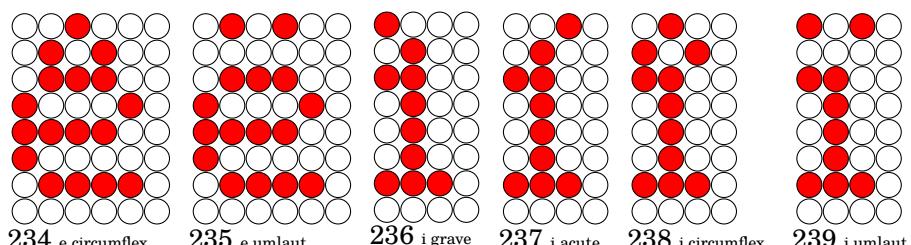
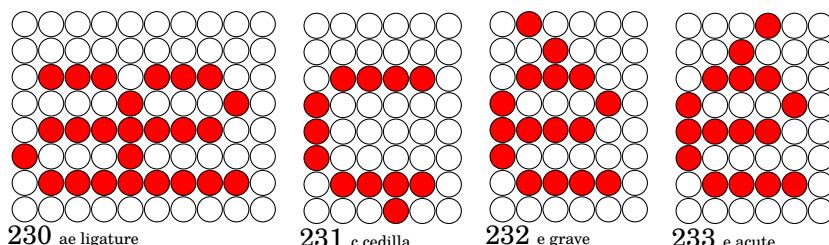
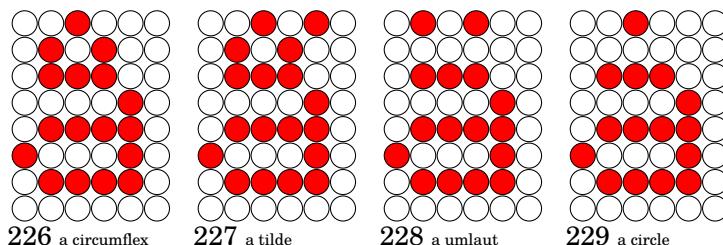
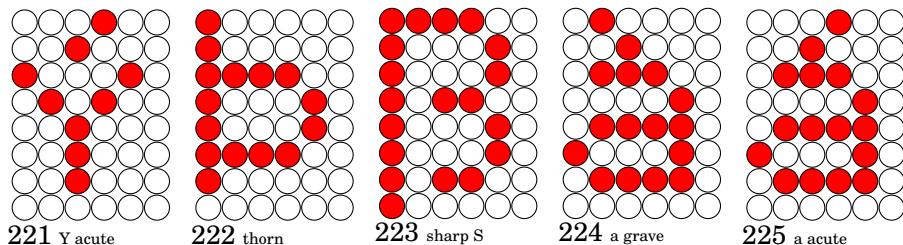
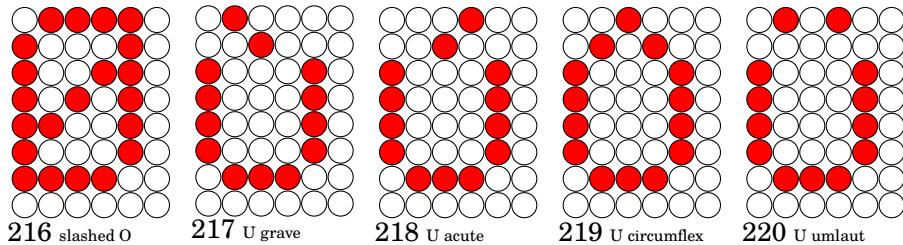


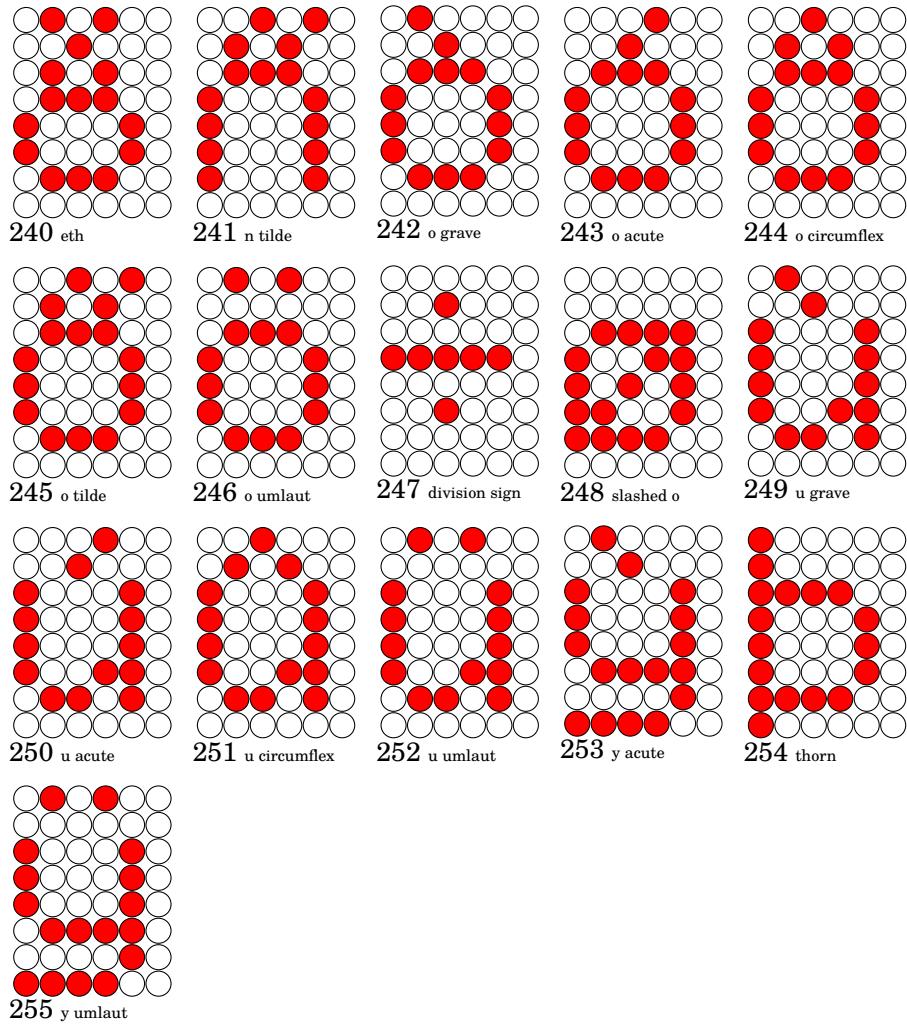
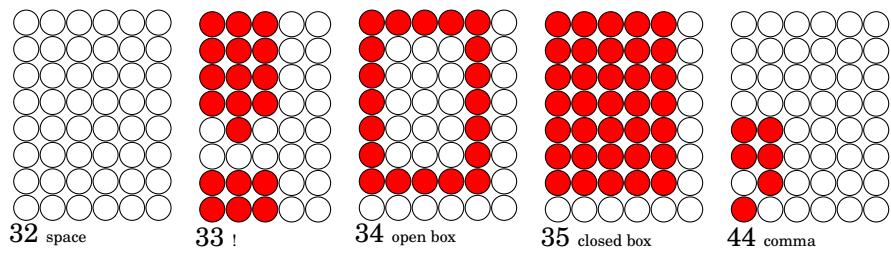


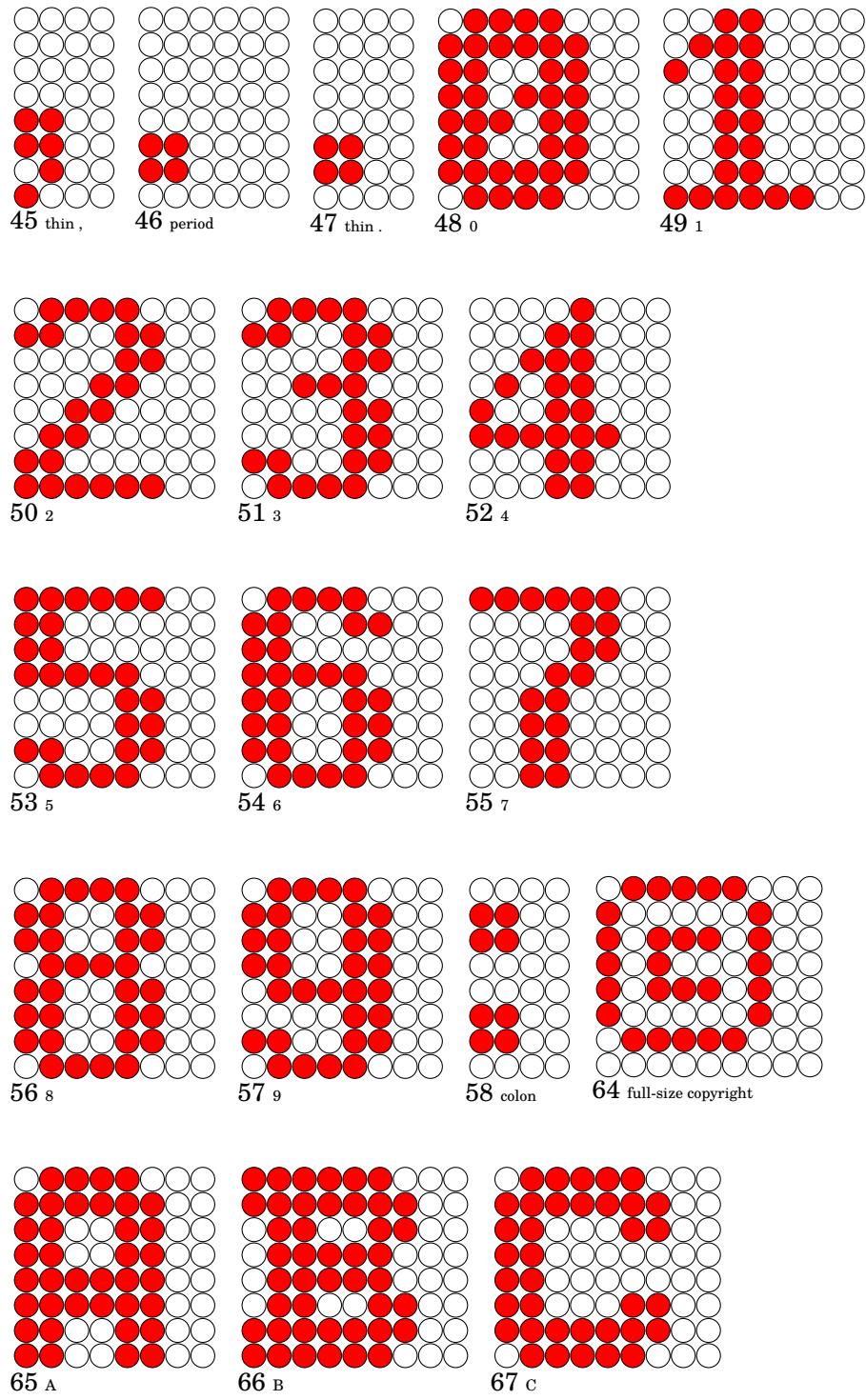


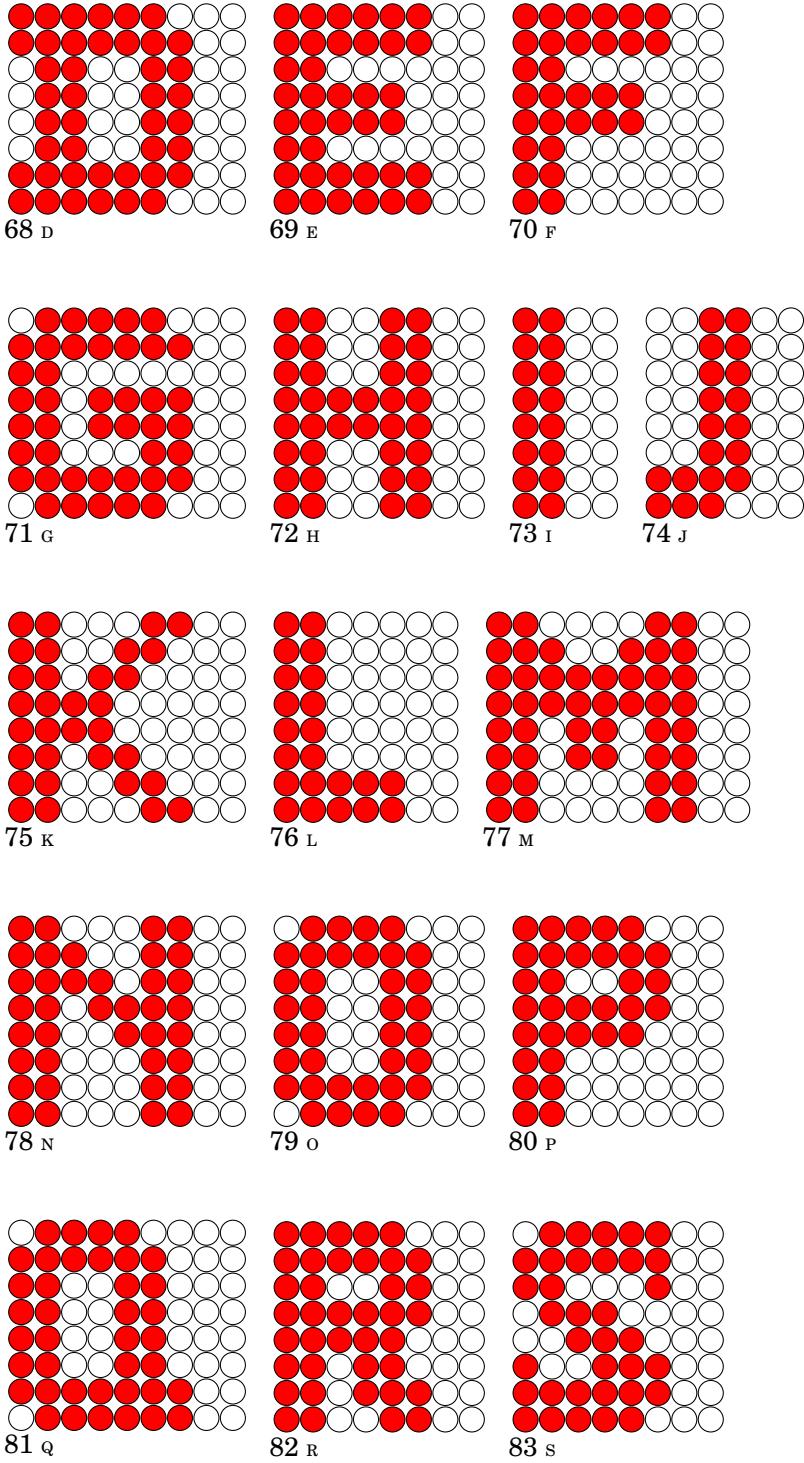


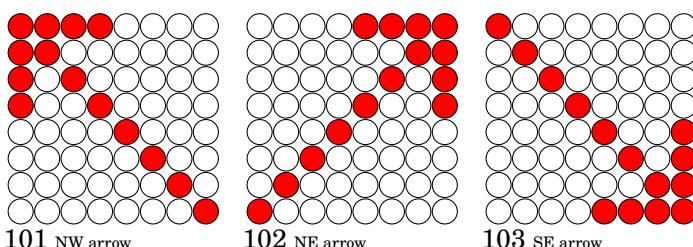
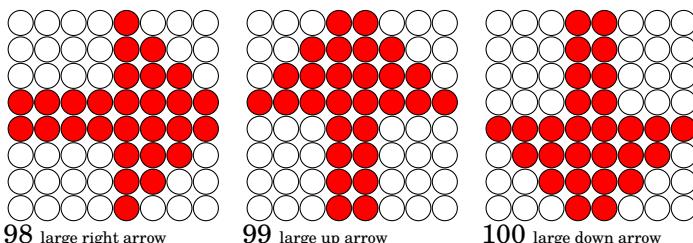
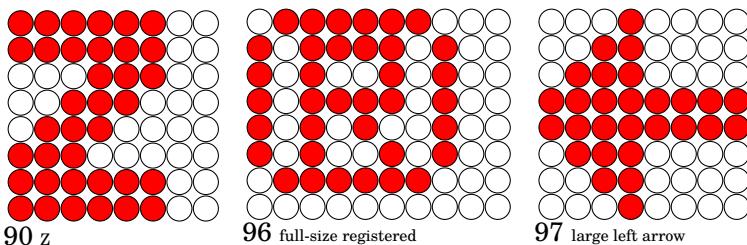
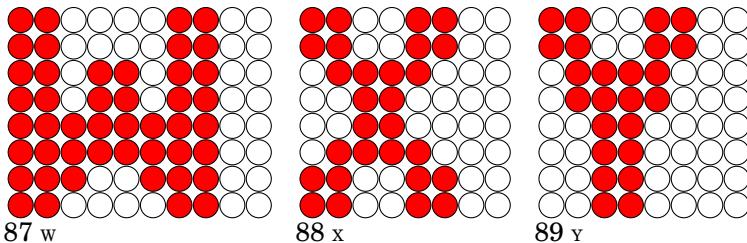
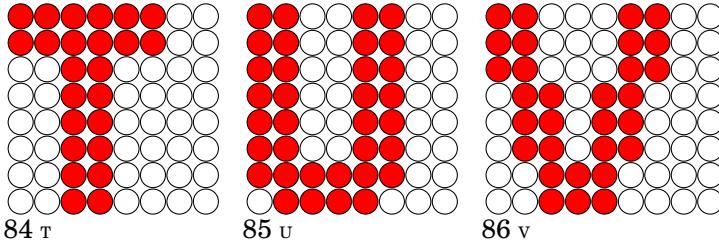


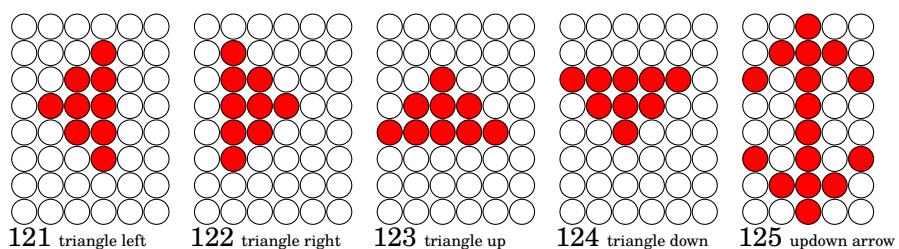
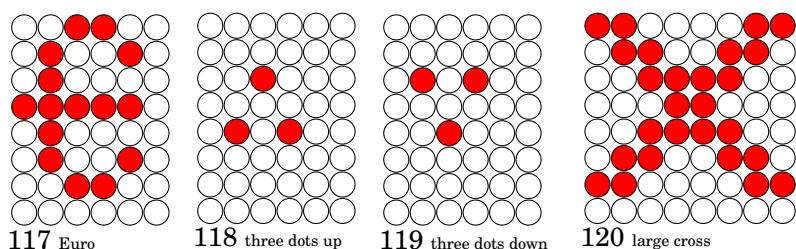
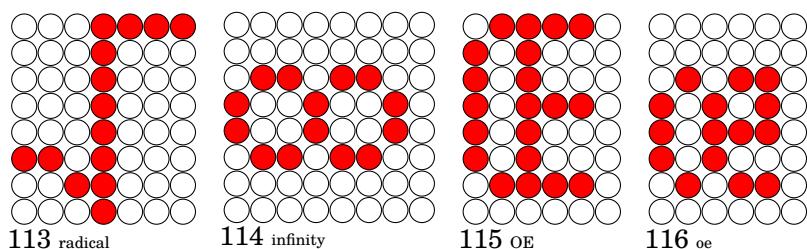
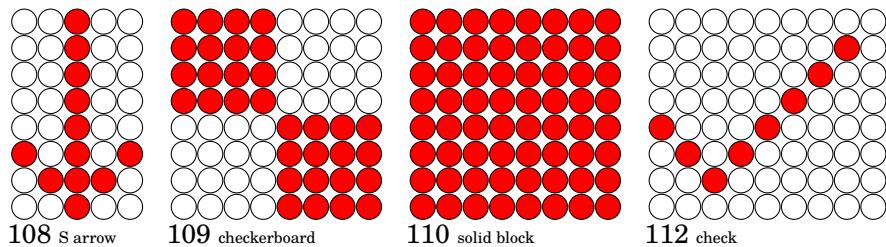
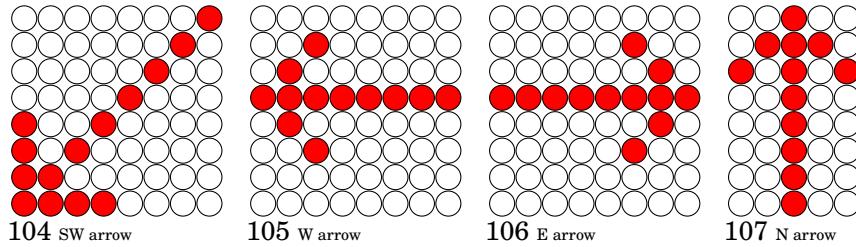


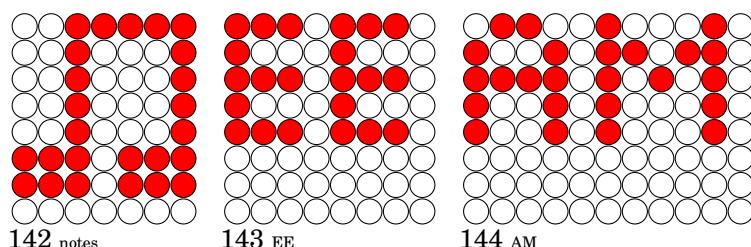
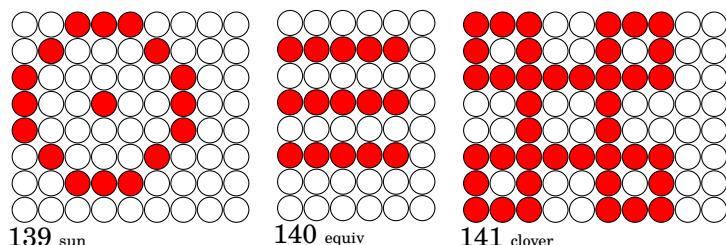
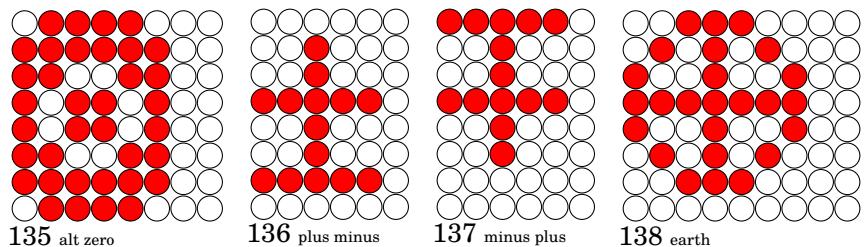
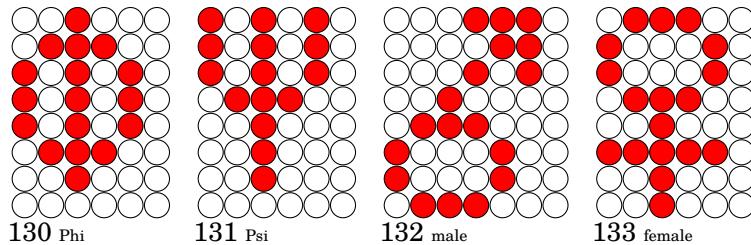
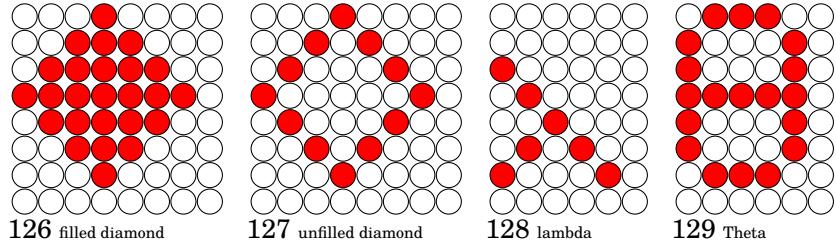
**Font #2 symbol.font**

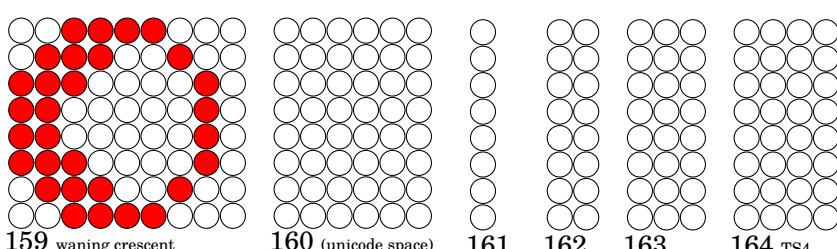
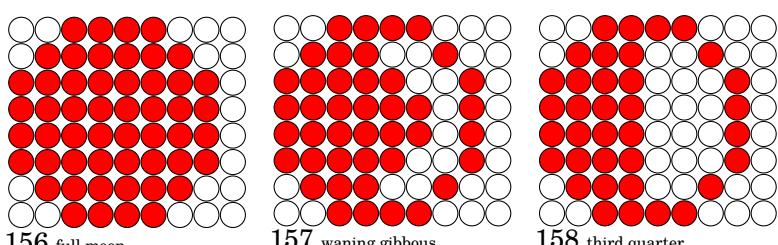
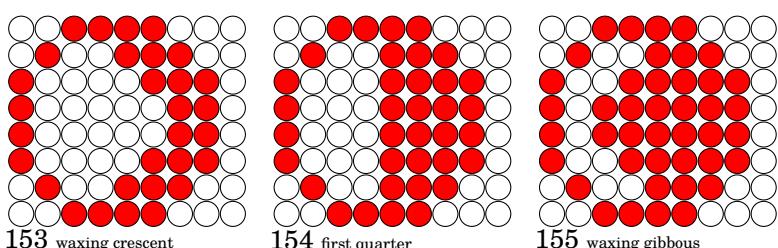
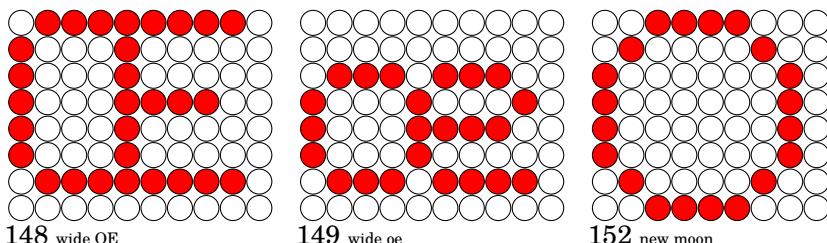
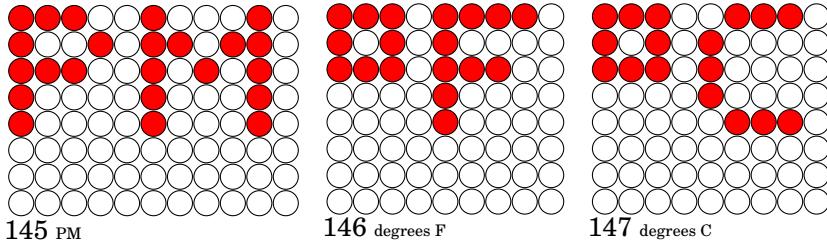


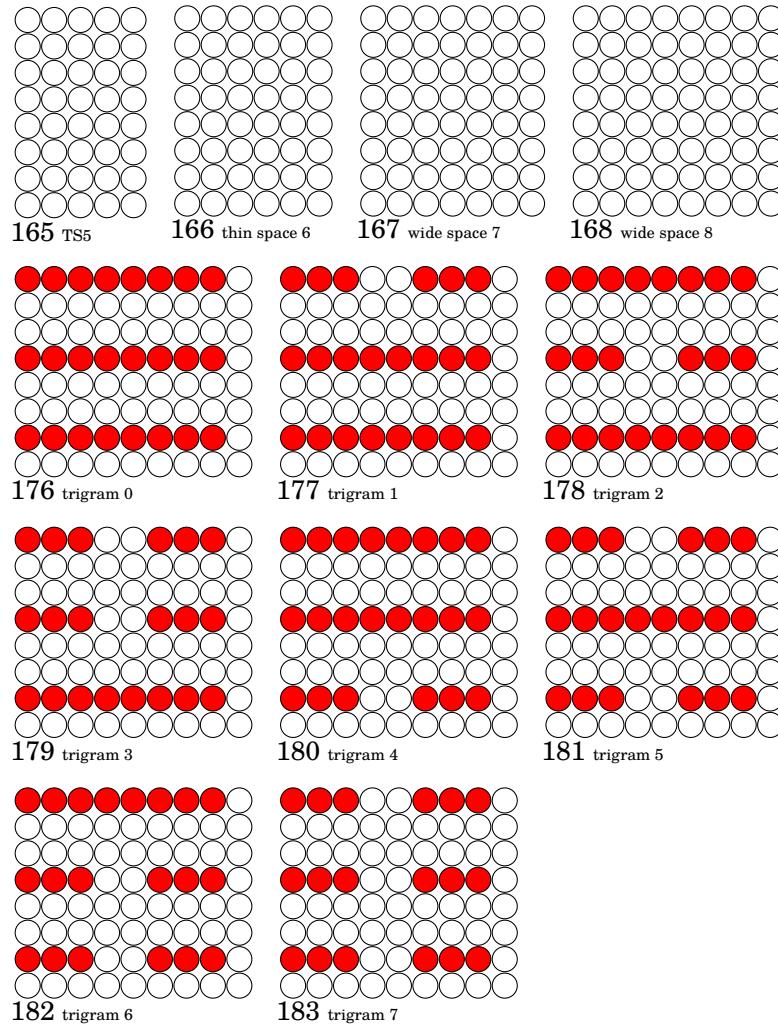




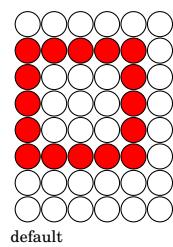








Default Glyph



Font Definition Files

In the `firmware/readerboard` directory there are a set of files with `.font` suffixes, which define the fonts that the readerboard can employ when writing text to the matrix. If you edit these files, or add more fonts, make the appropriate adjustments to the `Makefile` and run `make` in that directory to regenerate the glyph images on the preceding pages of this document, and the source files where the glyphs are actually stored in the device firmware. Then recompile the firmware image and flash it onto the hardware devices.

Each file defines up to 256 glyphs with codepoint values 0–255. The firmware will not allow some codepoints to be printed (see above, Tables 2.3, 2.7, and 2.8 on pp. 18–24), so to avoid reserved codes we recommend using codepoints ≥ 32 .

Any line which begins (possibly after leading whitespace) with a semi-colon (`' ;'`) is a comment and is ignored completely.

Each glyph definition begins with a line enclosed in square brackets, in the form

`[<codepoint> <description>]`

Where `<codepoint>` is either the codepoint expressed as a decimal integer (e.g., “42”) or a dollar sign followed by the codepoint as a hex integer (e.g., “\$2B”), and `<description>` is an arbitrary text description of the character being defined.

For example:

```

[$25 percent]
@0..@|
@0..@|
...@.| 
...@..| 
.@...| 
@..@| 
@..@| 

[$2E period]
..||||| 
..||||| 
..||||| 
..||||| 
..||||| 
@0||||| 
@0|||||

```

As a special case, a glyph introduced with the heading

[default]

is defined to be the default glyph to be used if the user tries to print a glyph that is not defined in the font. Only one default glyph may be defined in the entire set of fonts.

The non-blank lines which follow the bracketed header line define the bitmap for the glyph. In this definition, positions which are off are noted with ‘.’ or ‘|’, while positions where the LED is on are noted with ‘@’.

Any completely blank columns to the right of all lit pixels should be represented by ‘|’ rather than ‘.’. This will make note of the drawn width of the character but the columns to the right which consist of ‘|’ characters are not actually stored in the firmware bitmap image, thus conserving ROM space.

If fewer than eight rows are specified, it is assumed that the rows that are present start at the top of the display, and the remaining unspecified rows are all off.

Character Aliases

If a character glyph is to be identical to another glyph, there is no point in storing two or more identical bitmaps in the ROM. In such a case, rather than presenting the duplicate bitmap, the character definition is simply

```
-><codepoint>
or
-><codepoint>, <font index>
```

In the first case, the glyph for the indicated codepoint will be reused for this character, such as the case in Font #0 where the prime character (hex \$87) looks identical to the single-quote character (hex \$27). Only the single-quote glyph is actually specified in standard.font, so the prime character is defined only as

```
[$87 prime]
->$27
```

In the second case, the glyph to be duplicated comes from a different font file entirely. For example, many characters in Font #1 (standard_variable.font) have the same glyphs as Font #0, such as:

```
[$23 pound sign]
->$23,0
```

C H A P T E R



PREPARING AND UPDATING FIRMWARE IMAGES

Software is a great combination
between artistry and
engineering.

—Bill Gates

The firmware that runs on the readerboard hardware (or, more precisely, the Arduino single-board computer attached to the circuit board) is located in the `firmware/readerboard` directory of the readerboard project repository.

You'll need the following prerequisites in your Arduino development environment:

- EEPROM v2.0
- I2C_EEPROM v1.8.5
- TimerEvent v0.5.0
- Wire v1.0

After pulling the latest code from the repository if necessary, perform the following steps to configure and prepare the firmware image:

Step 1: Configuration. Edit the CONFIGURATION SECTION near the top of `readerboard.h`:

Values for HW_MODEL (Readerboard Hardware Model):	
MODEL_3xx_RGB	PCB revision 3.0 and later with RGB LEDs installed.
MODEL_3xx_MONOCHROME	PCB revision 3.0 and later with single-color LEDs installed with their anodes soldered into pin 4 (the blue circuit).
MODEL_BUSYLIGHT_1	PCB revision 1.x Busylight module.
MODEL_BUSYLIGHT_2	PCB revision 2.x Busylight module.

Values for HW_MC (Microcontroller Installed):	
HW_MC_MEGA_2560	Arduino Mega 2560.
HW_MC_DUE	Arduino Due.
HW_MC_PRO	SparkFun Arduino-compatible Pro Micro.

Table 7.1: Firmware Configuration Values

```
// BEGIN CONFIGURATION SECTION
// TODO: Set these before compiling for a particular hardware
//       configuration
#define HW_MODEL (MODEL_3xx_RGB)
#define HW_MC (HW_MC_DUE)
#define HAS_I2C_EEPROM (false)
```

These values should come already set up for the latest hardware model (in this example, the revision 3 RGB models which incorporate the Arduino directly onto the main circuit board). If your hardware uses a different design, HW_MODEL needs to be changed to another supported hardware name (see Table 7.1).

Also, HW_MC needs to be set to the model of Arduino microcontroller you will be using to control this unit.

If using an EEPROM chip or module external to the Arduino via the I²C bus, such as chip U16 on PCB revision 3.3.0 and later (which is recommended for use with the Arduino Due), then set HAS_I2C_EEPROM to true. If using no EEPROM or the one built-in to the Arduino, leave this set to false.

The supported values for these two configuration settings are listed in Table 7.1.

```
// TODO: Set these default settings (this will be the "factory
//       default settings"). On Due-based systems without
//       EEPROM chip U16 installed, this is the only way to
//       make these settings at all. If the EEPROM is installed
```

```

//      (or one is built-in to the microcontroller), this is
//      just the default that can be overridden by configuring
//      the unit since the new configuration values can be
//      saved in EEPROM.
//
// Default baud rate. Allowed values are '0'=300, '1'=600,
// '2'=1200, '3'=2400, '4'=4800, '5'=9600, '6'=14400,
// '7'=19200, '8'=28800, '9'=31250, 'A'=38400, 'B'=57600,
// 'C'=115200.
#define EE_DEFAULT_USB_SPEED ('5')
#define EE_DEFAULT_485_SPEED ('5')

```

These values determine the default baud rate to use for the USB direct connection and RS-485 network. Note that the value is a character literal.

```

// Default device address (may be any value from 0-63 except
// the global address, or EE_ADDRESS_DISABLED if you won't be
// using RS-485 at all.)
#define EE_DEFAULT_ADDRESS (EE_ADDRESS_DISABLED)
//
// Default global device address (may be any value from 0-15).
#define EE_DEFAULT_GLOBAL_ADDRESS (15)

```

With EE_DEFAULT_ADDRESS set to EE_ADDRESS_DISABLED, the RS-485 interface will be disabled. Otherwise the value set for this symbol is an integer in the range 0–63 which will be the device address recognized in commands received over the RS-485 network.

Set EE_DEFAULT_GLOBAL_ADDRESS to the default global device address you wish the unit to recognize. This must be in the range 0–15.

```

//
// TODO: Adjust these for the colors of discrete status LEDs on
//        this unit. These can be used for color values in
//        commands sent to the device.
//
//        READERBOARDS (8 LIGHTS)
#define R_STATUS_LED_COLOR_L0 ('G')
#define R_STATUS_LED_COLOR_L1 ('y')
#define R_STATUS_LED_COLOR_L2 ('Y')
#define R_STATUS_LED_COLOR_L3 ('r')
#define R_STATUS_LED_COLOR_L4 ('R')

```

```
#define R_STATUS_LED_COLOR_L5 ('b')
#define R_STATUS_LED_COLOR_L6 ('B')
#define R_STATUS_LED_COLOR_L7 ('W')
//
//      BUSYLIGHTS (7 LIGHTS)
#define B_STATUS_LED_COLOR_L0 ('G')
#define B_STATUS_LED_COLOR_L1 ('y')
#define B_STATUS_LED_COLOR_L2 ('Y')
#define B_STATUS_LED_COLOR_L3 ('r')
#define B_STATUS_LED_COLOR_L4 ('R')
#define B_STATUS_LED_COLOR_L5 ('B')
#define B_STATUS_LED_COLOR_L6 ('W')
```

These are set for the standard color assignments for the discrete status LEDs, allowing for the use of these single-letter color codes to refer to the status LEDs. If you build a unit with different colors, you may assign appropriate color codes here.

Since busylights and readerboards have different numbers of status light displays, they each get a separate set of color code assignments. If you are setting up a nonstandard readerboard, then edit the codes defined here for symbols starting with R_, and likewise with the B_ symbols for nonstandard busylight units.

If your unit doesn't have the full compliment of lights possible, assign the unused positions to codes with a digit that matches the light number. For example, for a three-color busylight:

```
#define B_STATUS_LED_COLOR_L0 ('G') /* green */
#define B_STATUS_LED_COLOR_L1 ('R') /* red */
#define B_STATUS_LED_COLOR_L2 ('B') /* blue */
#define B_STATUS_LED_COLOR_L3 ('O') /* orange */
#define B_STATUS_LED_COLOR_L4 ('4') /* unused */
#define B_STATUS_LED_COLOR_L5 ('5') /* unused */
#define B_STATUS_LED_COLOR_L6 ('6') /* unused */
```

Move on to the version number banners.

```
#define BANNER_HARDWARE_VERS "HW 3.4.1"
#define BANNER_FIRMWARE_VERS "FW 2.3.7"
#define SERIAL_VERSION_STAMP "V3.4.1$R2.3.7$"
//                                \___/ \___/
//                                |       |
```

```
//           Hardware version    |
//           Firmware version
```

The BANNER_FIRMWARE_VERS value should match the version of the firmware source code you are compiling.

Likewise, BANNER_HARDWARE_VERS gives the revision of the PCB in use.

These should be 10-character text strings. They will be displayed briefly when the sign is booted.

Set the SERIAL_VERSION_STAMP value to match the previous two values while still preserving the format “V...\$R...\$”. This string will be sent verbatim as part of the device’s response to being queried for its configuration and status.

If, and **only if**, you are making a custom firmware image for a board with no EEPROM storage capability, adjust the BESPOKE_SERIAL_NUMBER definition with the serial number of this device. For example:

```
#define BESPOKE_SERIAL_NUMBER "RB1234"
```

Step 2: Make. Run the make program. This will build the font tables from the various *.font files into the firmware code so the target device will be able to generate characters from those fonts.

Customizing and extending the fonts is described in Chapter 6.

Step 3: Compile. Using your Arduino compiler tools, compile the readerboard sketch files and upload the resulting binary to your readerboard hardware. This step assumes you have set up your own Arduino development environment and is beyond the scope of this document to help you with.

Step 4: Serial Number. If your unit has EEPROM memory, run the setsn command to field-configure the device’s serial number. For example:

```
% setsn -port /dev/ttyACM0 -sn RB1234 -model rb
```


C H A P T E R



SCHEMATICS AND DIAGRAMS

Whether you draw diagrams
that generate code or you type at
a browser, you are coding.

—Kent Beck

The following pages include the schematics for the current 3.4.1 version of the readerboard PCB and the board assembly diagram, showing where the components are placed, and the same for the 2.2.0 version of the Busylight circuit board and 1.1 light tree tier boards for use with the Busylight units.

Two versions of the light tree octagons are provided. Version 1.1 uses through-hole resistors, while version 1.1-S uses surface-mount resistors.

