

Table of Contents

I. Abstract	1
II. Implemented Prototype	
1. Introduction	2
2. Prototype Specifications	4
3. Prototype Performance and Evaluation	8
4. Financial Budget	13
5. Project Schedule	15
III. Conclusions	16
IV. References	17
V. Appendices	
A. Circuit Diagram	18
B. Source Code - Assembly	19
C. Source Code – Message Sender and Monitor Program	38
D. Digital Logic Diagram	54

*** Note: This report differs slightly from the actual report submitted for the project. The only differences are the locations of the appendices within the report, which were shifted for pdf conversion to be possible.

I. Abstract

This project was centered on the creation of a new modular LED signboard system. Theoretically it would be targeted for sale as a fad in the 16-25 age group. As compared to existing signboards our modules are be more flexible and easier to work with. Also key would be a variety of applications wherein an array of our modules would be linked to a computer for applications like process monitoring and instant messages.

We have created three prototype patches these patches are each 8 LEDs high and 24 LEDs wide and controlled by a Motorola HC11 microprocessor. The project can be broken down into two main parts, the hardware design of both the LED display itself, including the necessary decoder and driver hardware to control the lights, and the software used on board the patches and for the computer interface to the patches. The hardware is clear success. The decoder circuitry has only one flaw in it and that is split between it and the software. This bug is a slight “shadow effect” caused by the column select lines switching before the row select lines are fully zeroed. If treated as a hardware problem this is a matter of filtering and switching speed. However, increasing the software delays in the main loop is the best solution, which will happen as development continues in any event.

II - 1. Introduction

Our basic design is to use a network of HC11 processors to control the array of LEDs. Each patch has one processor and 192 LEDs. The entire array has a single master controller that directs the array as a whole. The slave controllers listen for commands from the master and update their own displays.

Each microcontroller will boot, announce its presence, and, if it receives no reply, assert itself as the array's master controller. The master controller has the additional task of controlling the boot process of the remainder of the array and generally does all the processing for the array as a whole. At a lower level the master must control all communications including sending commands and periodically polling for errors. The slave controllers have a much simpler task, they must process the commands from the master, reply appropriately, and update their own display.

For a communications network we have used the HC11's asynchronous serial interface. The idle detect feature on the interface allows each slave processor to wake up, decode addressing information, process the instruction if need be, and wait for the next instruction. By using a multidrop asynchronous serial line we avoid having a separate set of slave enable lines, but gain an absolute need to have the master control transmissions at all times thus avoiding collisions.

To prevent all collisions on the serial network we use some digital logic. Before any chip is allowed to transmit on the serial bus it will have to reset fully. This is handled by a completely separate set of communications lines connecting the patches in a secondary communications system. The secondary system functions by allowing the patches to trigger the reset lines of adjacent patches, which prevents them from starting to use the serial network. The reset lines are then systematically released based on commands from the array master. Three or gates will suffice to coordinate the reset initialization process.

Digital logic will also be used as a port expander. The HC11 has only a limited number of ports and our LED array will require 192 individually switched lines. Therefore the logic component takes its input from the HC11's parallel port and fans it out into a much larger set of outputs. This is done by cycling through the columns and lighting up only the proper LEDs in each column. Four decoders are necessary to control

the entire cycling process with only five output lines from the HC11. Another 8 lines connect through drivers to the columns in the array.

Features

Our project was to create a set of three prototype patches, These patches:

- Are 8 LEDs high and 24 LEDs wide – contrary to our expectations the LEDs turned out to be just as bright as we would specify for a full salable model.
- All run identical programs and function identically when connected in any order. Because of the two different evaluation boards used we loaded two different programs, differing only in the addresses where the programs are loaded and the initialization routines for the I/O ports
- Have the software logic to elect a master controller to control the array using the digital logic assisted reset method as described later. This capability is untested.
- Operate in a coordinated fashion for the implemented features

The prototype devices include a limited set of features specifically:

- light show – two different simple light shows that could be adapted to present a single unified show across several panels without any external input. Because communications are not fully implemented the shows do not actually run in this kind of unified display.
- marquee - displaying a text message across one. In the prototype we altered one patch to display a particular message.
- bitmap display - the patch array will display bit maps provided by a computer. In this mode the computer will do all the processing to create the bitmaps.

II - 2. Prototype Specifications

Parts and Circuitry

Each patch is faced with the responsibility of powering 2V, 30 mA LEDs, as well as strobing across the columns of the array at high speeds. This requires driver circuitry for both sourcing and sinking, as well as demultiplexing 5 lines to select one of the 24 columns. In addition to this, the reset logic for array initialization is to be accomplished on board.

A complete circuit diagram is given in **Appendix A**, and a layout of the decoding logic is in **Appendix D**. There are a total of 20 lines going to each patch from its HC11 board. The first connector has 8 lines in it and is used for row selection (one line is dedicated to one row in the 8x24 array). These are directly connected to the UDN2985 source driver, which provides +5V to the selected row(s). These 8 lines come from Port C of the HC11. The next connector has two purposes. The rightmost 5 lines (out of 6) are bits 4-0 for selecting the column of LED's to be selected. The leftmost line is connected to the reset line of the HC11. In the final 6-line connector, incoming and outgoing patch reset lines are connected, as well as two lines that go to Port B of the HC11 for controlling the up and right reset lines.

The bits 4 and 3 of the column select are sent to a 2x4 demultiplexer, whose outputs are connected to the Latch Enable of the three 3x8 demultiplexers. Bits 2-0 are connected directly to the three inputs of the 3x8 demultiplexers. The outputs of the demultiplexers are 24 lines, each of which is fed through a line of a sink driver. When the line is set high, the sink driver provides a path to ground for the selected column.

The LEDs are connected as follows. First, the cathodes of each row are bent over a strand of bus wire, soldered to it and clipped. Soldering the anodes in place allowed the LEDs to hold a more rigid position but was not absolutely necessary. Once all LEDs were soldered by this method, each row had to be created by connecting all of the anodes in a row. To avoid contact with the soldered columns, wirewrap was used between each anode, elevating the connection. At the end of each row, a current-limiting 100 ohm resistor is connected, the other end going directly to a channel of the source driver.

The current prototype powers the array by feeding a regulated +5V and ground from one of many power taps on the HC11 development board. This is acceptable since

the HC11 board is equipped with a heat-sinked 1A voltage regulator. A10 uF capacitor is soldered by the power connections of each chip to suppress the propagation of digital noise back to the HC11 board.

All connections are made between the LED board and the HC11 board via mini-PV connectors. These can be obtained from any electronics supplier such as Allied Electronics. A crimping tool may be required.

Below is a list of parts used on the prototype, and an explanation of each.

HC11	3	HC139	3
LED	586	HCT32	3
Source Drivers	3	PC Board	3
Sink Drivers	9	Board standoffs	3
Wire	1	Wirewrap	2
HCT259	9		
Connectors	60 lines		
IC Sockets	45		

1) HC11 Prototype boards

The HC11 prototype boards were obtained from different places, for reason of convenience. The first one that was used is an EVBPlus board, obtainable at <http://www.evbplus.com/>. The other two were obtained from Futurlec.com, named the “68HC11DEVBRD”. The EVBPlus has the advantage that it has two serial ports, one for debugging and one for communications from the HC11. The Futurlec boards have only one serial port, a handicap which made our project difficult when trying to implement communications.

2) LEDs

The LEDs used were 5mm Hi Efficiency Red Diffused, part no. 160-1087-ND from Digikey.com. The manufacturer is LITE-ON, inc. They were chosen for low price and the fact that they are diffused.

3) Source Drivers

The Source Drivers used are the UDN2985 8-channel source drivers from Allegro Micro. They can be purchased from any mainstream electronics supplier. Each channel can source up to 120 mA, and the maximum total current to be sourced is 250 mA. Each

driver is essentially a buffer, and is changed from 0V to 5 V when a 5V TTL signal is applied to its input.

3) Sink Drivers

The Sink Drivers selected were the Toshiba 62083, obtained from Digikey. They accept TTL level inputs, and provide a path to ground when excited with a 5V input. Each channel can sink 500 mA, which is more than enough for our application.

4) Wire

For wiring the ICs and LED, two types of wire were used. 30 gauge wire-wrap wire was used to string together the anodes of the LEDs, and 24 gauge bus and hook-up wire was used to distribute power to the drivers as well as connect the columns to the sink drivers.

5) Demultiplexers

Two types of demultiplexers were used. The first is the standard HC139 2x4 demultiplexer, which sets the selected input low. Second are the HCT259 8-bit addressable latches. These have an active low Latch Enable which is compatible with the HC139, and they set the selected input high which is compatible with the 62083 sink driver. These chips follow a strict standard and can be obtained almost anywhere ICs are distributed.

6) PC Board and standoffs

The standard Radio Shack pre-drilled 4.5” x 6 1/8” PC board was used for this prototype. It is the perfect size to comfortably fit the array of LEDs and all their supporting ICs. The standoffs, also from Radio Shack, keep the board off the surface it is resting on, which prevents the possibility of a short circuit.

Patch Characteristics

Power Consumption	Min: 0.5 W	Max: 2 W	with 9V supply
Refresh Rate	Min: 120 Hz	Max: 140 Hz	
LED board	Length 4.5"	Width 6.125"	
LEDs	Von = 2V	I = 30mA	
Lowest logic speed	T(on/off) = 500ns		
Signal wire gauge	30 AWG		
Power wire gauge	22 - 24 AWG		

Programming Languages

Currently, all programming on the HC11s is done in assembly language, and future changes would have to be made in this language. The computer interface and testing programs are all developed in C.

II - 3. Prototype Performance/Evaluation

The cross chip synchronization will be achieved by controlling the reset lines of the HC11 chips. As long as this reset line is held low the chip will not complete its power up sequence and thus will not cause any collisions with the other chips on the serial line. Two outputs from the HC11 will be used to assert the reset lines of the two patches immediately up and to the right and two matching inputs will accept resets in the same manner. When one of these reset lines is asserted it should assert both output lines immediately and also assert the reset line on the HC11. These reset lines are present and one of the prototypes includes the extra OR gates but this sub-system was not tested in full operation because it would require an execute on startup feature so that our program rather than a debugger would run after the reset. Only one of our EVB boards had such a feature and a proper test would have needed two.

Assembly Programming

Assembly programming was slower than expected because of the inherent obscurity of bugs in assembly code. Delays primarily stemmed from cases where a confusing bug was encountered that required multiple investigations. For example, in one phase variables seemed to be changing randomly because our data storage overlapped with the debugger. This was not apparent for several sessions of debugging.

Program Design

After the initial boot the patches will enter a continuous loop of displaying their current state with interrupts for system processes. The design and current progress is shown below.

Boot

Internal init

Send I'm here

Wait for master reply

No Reply: Enter Master Mode

Assigned an ID: Enter Slave Mode

Master Mode:

Display loop
Initialize array
Start lightshow
Clocked loop

Master mode works properly as far as it can be tested in the absence of a slave. The patches successfully detect the absence of any other patch and display their light show.

Slave Mode:

Display loop
Process message if complete
Constant processes
Interrupt on character received
If first character check addressing
Not this patch: Return to idle
This patch: Store character
Otherwise: store character

Slave mode message processing for crucial messages was written and is included in the source code section but was not integrated because the serial communications network could not be brought up at all. While the network was theoretically in working order, our second and third patches had only one serial port each and we needed that port for debugging.

In addition to the serial port issue, a serious problem was the accessibility of EEPROM on the two HC11 boards we purchased from Futurlec.com. The serial port problem would be somewhat avoidable if we could write the program to EEPROM and run it independently from the debugger. However, the EEPROM loading software simply does not work. Thus we were not able to replace the debugger with our code without risking never having the debugger available again.

Software Design

Functionally, the computer interface software, written in C and in a Linux environment, was able to send and receive messages, and was designed to boot up the array as well. This came close to our original goals for the prototype, which included the above features plus the abilities to make the master program do a few useful things (light show, scrolling marquee, and bitmap pushing) once the array was initialized.

The programs were tested in a number of ways; however, some were tested more extensively than others due to the limited communications the group was able to get operational. The first program that was created was the message receiver, or snoop, program. Originally developed to be a debugging environment for inter-patch communications, it saw limited use to prove that messages were being sent from the patches. The other program, the message sender, was also into to aid in debugging, but primarily was envisioned as a preliminary way of controlling the array manually.

These programs were tested together, with the receiver program monitoring one serial port on a computer and the sender program using the other serial port. The two serial ports were connected together with a null modem cable. As is stands in the prototype, the sender was able to successful send valid messages to the receiver, and the receiver successfully read the messages and displayed information on screen according to the message type being sent.

The master program's array initialization code was never tested because of the communication problems between patches. Theoretically, the program could have been tested by monitoring the message output it produced and sending messages back to the patch to simulate the existence of other patches. By the time the feature was completed it was evident that the group would not be able to interface the patches together or in multiples to the computer, so testing was scrapped.

One test between the patches and the computers that was successfully accomplished that was completed towards the end of the project was the confirmation that patches were sending their messages. The patch was connected with the computer and the message receiver program was run while the patch booted. It was observed that the patch did send the proper "I'm here" message to the computer as required by the specification before it decided it was master and went into light show mode.

Circuitry

The decoding circuitry on each LED board is designed for use in serial communications, so the circuitry is made for high speed switching. The driving circuitry is also made for high speeds, with the longest t_{on} being 500 ns. Currently the prototype refreshes at speeds in which it shows no sign of flicker or of distortion due to high speeds. However, it should be noted that when the software was first written and the display was refreshed as fast as the processor could handle, there was a ghost image of the display shifted one column to the right. The exact cause of this was not characterized; it was assumed that the effect would vanish with the lengthening of execution times and insertion of delays to approach a somewhat standard refresh speed.

The current refresh speed is approximately 7.8 ms/sweep, or 128 Hz. At this speed the LEDs glow visibly and flicker is undetectable. Earlier tests determined a safe refresh rate of 120 – 140 Hz, since this was the boundary below which flicker was evident if the observer shifted his eyes. The current speed is achieved using loops of wasted processor time; clearly much more computation could be accomplished while retaining the same refresh rate.

The LEDs currently glow visibly, however in the presence of sunlight are much harder to discern. The prototype's objective was to have the display be visible, though not necessarily brilliant. The LEDs are lit by simply pulsing the nominal current whenever they are to be on. This is especially useful for debugging since they can be left on accidentally with no problem. A future upgrade would be a carefully planned excitation method. This would include not only an optimal refresh rate, but a method by which the LEDs are pulsed with a current higher than their nominal current so they shine brighter. The downside to this is if they are left on without strobing they may be destroyed.

Currently, all inter-board connections are made via mini-PV connectors. These are plastic housings with spring loaded metal receptacles crimped onto wire. We employed a smaller variety for the pins on the HC11 board, and a heavier duty, stronger-grasping variety for the pins on the LED board. The smaller connectors were used on the HC11 board so as to allow for the same connector to be used on any HC11 board; each connector only contacted two pins. Thus, if pins are arranged differently on two board,

the connectors can be separately connected accordingly. The downside to this is the heightened possibility of a connector coming loose.

A final note on the prototype circuitry deals with the use of different demultiplexing on one of the patches. This was done for the first array that was built due to a parts shortage. The three 3x8 demultiplexers and one 2x4 demultiplexer were replaced by two 4x16 demultiplexers, plus a single inverter made from a transistor and two resistors. This switch was made due to the availability of the 4x16 demultiplexers at a retail location during the construction of the array. The speed and voltage characteristics are virtually the same as in the regular design, so it operates as expected.

II - 4. Financial Budget

The original budget called for \$131.13 worth of components. This did not include all the additional components that went into the final prototype that the group did not foresee to include in this figure, such as connectors, PC boards, and other paraphernalia. Below is a table of our original parts list.

Original Patch Idea

Item	Cost	Quantity	Total Price
FPGA	\$13.20	5	\$66.00
LED	\$0.06	1000	\$62.40
HC11	\$55.00	3	\$165.00
Source Drivers	\$5.00	15	\$75.00
Sink Drivers	\$5.00	5	\$25.00
Total			\$393.40
Cost per patch			\$131.13

What was actually spent turned out to be a bit better than we had predicted. After a significant design change, even with the additional unforeseen components, the price per patch turned out to be \$105.94.

Current Patch

Item	Cost	Quantity	Total Price
HC11	\$65.00	3	\$195.00
LED	\$0.07	700	\$47.74
Source Drivers	\$2.22	5	\$11.10
Sink Drivers	\$0.72	11	\$7.92
Wire	\$4.69	1	\$4.69
HCT259	\$0.50	11	\$5.50
HC139	\$0.39	5	\$1.95
HCT32	\$0.40	5	\$2.00
PC Board	\$3.99	3	\$11.97
Board standoffs	\$1.49	3	\$4.47
Wirewrap	\$2.99	2	\$5.98
Connectors	\$15.00	1	\$15.00
IC Sockets	\$0.50	9	\$4.50
Total			\$317.82
Cost per patch			\$105.94

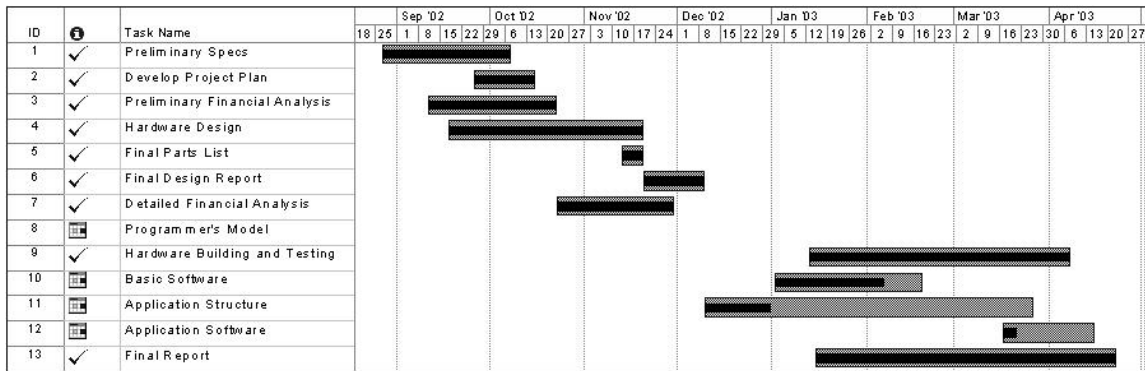
If the group were to make another set of patches, the price could be reduced somewhat more by eliminating the purchase of components that were duplicates, extras, and replacements.

Current Patch			
Item	Cost	Quantity	Total Price
HC11	\$65.00	3	\$195.00
LED	\$0.07	576	\$39.28
Source Drivers	\$2.22	3	\$6.66
Sink Drivers	\$0.72	9	\$6.48
Wire	\$4.69	1	\$4.69
HCT259	\$0.50	9	\$4.50
HC139	\$0.39	3	\$1.17
HCT32	\$0.40	3	\$1.20
PC Board	\$3.99	3	\$11.97
Board standoffs	\$1.49	3	\$4.47
Wirewrap	\$2.99	2	\$5.98
Connectors	\$15.00	1	\$15.00
IC Sockets	\$0.50	7	\$3.50
Total			\$299.90
Cost per patch			\$99.97

Finally, the HC11 in the final version of the product need not be in a development board, so that price would come down as well.

As for the cost of man-hours, just about as many man-hours were put into the project as expected: (3) engineers * (\$40/hour) * (10 hr/wk) * (15 wks) = \$18,000.

II - 5. Gantt Chart



As can be seen from the Gantt chart above, all tasks were completed with the exception of the software. The basic software is about 75% finish, the remainder lying in testing and debugging; the structure and most of the code is already developed. As for the application software, it is only about 20% done. This is a direct result of the inability for the patches to communicate properly. The programmer's model was abandoned in the meantime, since we would need the applications to be more complete to better understand the problem at hand.

The group continued to put out the 30 man-hours/week that we expected to. However, there was a shift in the distribution. 80-90 man-hours ended up being used on hardware construction, testing, and compatibility issues. Similarly, over 100 man-hours were use solely on the basic software, greatly limiting the time that could be spent on application software. These diversions were necessary to bring the prototype to working order; however if more time was given to compatibility issues earlier in the project, the communications failure may not have been an issue.

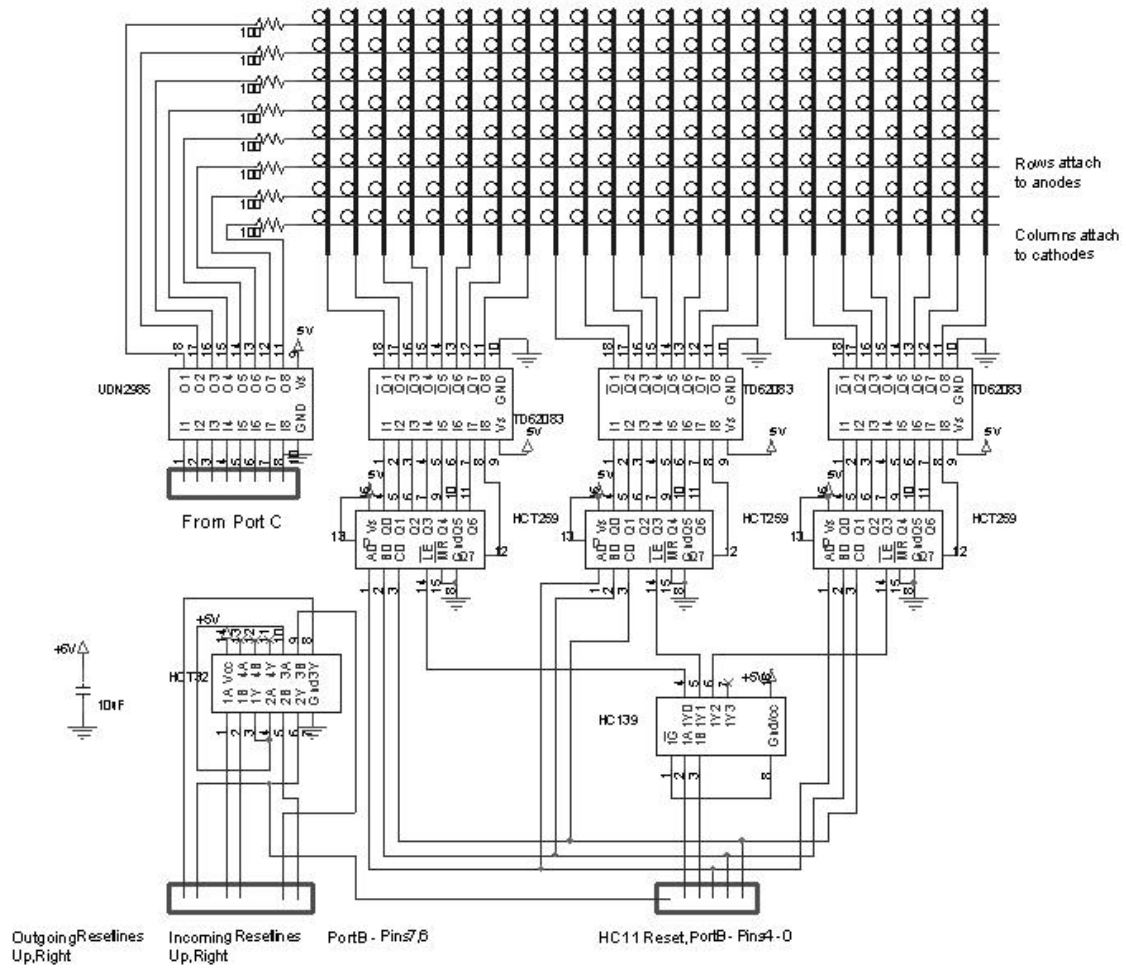
III. Conclusion

The modular LED signboard worked well but a critical weakness in some of the prototype hardware left the critical—and novel—communications system, untestable. The failure to detect this situation in the original purchase of the EVBs was the major problem with this project. A second problem arose from the difficulty of interfacing the computer to the patch's special, multi-drop serial network. Although we used a special cable to tie the send and receive lines together as they would be in such a network, meeting the computer's additional handshaking requirements was not possible.

References

1. Huang. "MC68HC11: An Introduction Software and Hardware Interfacing."
Stamford, CT:Delmar, 2001
2. "MC68HC11A8 Technical Data," Rev 6. Motorola, 1991
3. "MC68HC11A8 Reference Manual," Rev 3. USA: Motorola, 1991

Appendix A



Appendix B

```
*v.01
* this is the first draft of a program to control the signage LED
patches,
* it focuses on initialization

*'Normal' register assignments
*may or may not be consistant
*A - Main variable
*B - Scratch variable
*X - REGBLK pointer
*Y - buffer pointer

rs485_recv:      org      $7000 ; EVBplus board I/O routines
                 rmb      3      ; enables rs485 recv mode
rs485_xmit:      rmb      3      ; enables rs485 xmit mode
get_date:        rmb      3      ; gets date info. from host PC
get_time:        rmb      3      ; gets time info. from host PC
outstrg00:       rmb      3      ; print a string ended by 0
lcd_ini:         rmb      3      ; initializes the 16x2 LCD module
lcd_line1:       rmb      3      ; displays 16 char on the first line
lcd_line2:       rmb      3      ; displays 16 char on the second line
sel_inst:        rmb      3      ; selects instruction before writing
LCD module
sel_data:        rmb      3      ; selects data before writing the LCD
module
wrt_pulse:       rmb      3      ; generates a write pulse to the LCD
module
seven_segment:  rmb      3      ; convert Accu A to segment pattern,
bit 7= DP
*
                 org      $7FA0 ; buffalo i/o routines
upcase:          rmb      3
wchek:           rmb      3
dchek:           rmb      3
init_sci:        rmb      3
input:           rmb      3
output:          rmb      3
outlhlf:         rmb      3
outrhlf:         rmb      3
outa:            rmb      3
outlbyt:         rmb      3
outlbsp:         rmb      3
out2bsp:         rmb      3
outcrlf:         rmb      3
outstrg:         rmb      3
outstrg0:        rmb      3
inchar:          rmb      3

*HC11 defines
DB0:             equ      1
DB1:             equ      2
DB2:             equ      4
DB3:             equ      8
DB4:             equ      $10
DB5:             equ      $20
```

```

DB6:          equ    $40
DB7:          equ    $80
NOTDB5:       equ    $DF
NOTDB6:       equ    $BF
OL4:          equ    DB2
OM4:          equ    DB3
OL3:          equ    DB4
OM3:          equ    DB5
porta:        equ    0
portb:        equ    4
portc:        equ    3
ddrc:         equ    7
portd:        equ    8
ddrd:         equ    9
toc2:         equ    $18
tctl1:        equ    $20
toc3:         equ    $1a
tmsk1:        equ    $22
tflg1:        equ    $23
tmsk2:        equ    $24
adctl:        equ    $30
adr1:         equ    $31
adr2:         equ    $32
adr3:         equ    $33
adr4:         equ    $34
option:       equ    $39

REGBLK:       equ    $1000
PIOC:         equ    $02
BAUD:         equ    $2B
SCCR1:        equ    $2C
SCCR2:        equ    $2D
SCSR:         equ    $2E
SCDR:         equ    $2F

```

*this would be in rom somewhere w/ all 4 other bytes, testing simplification

```

MYSERIAL:     equ    $0A

```

*Message Type numberss

```

MSGTYP_IMHERE:    equ    $01
MSGTYP_URID:      equ    %00000011
MSGTYP_IMMSTR:    equ    %00000010
MSGTYP_RELRIGHT: equ    %00001100
MSGTYP_RELUP:     equ    %00000101
MSGTYP_:          equ    %0000
MSGTYP_ORERR      equ    %00001000

```

```

MSGTYP_SETMSTR:   equ    %00000111
MSGTYP_ASRTUP:    equ    %00011010
MSGTYP_ASRTRT:    equ    %00011011

```

```

MSGTYP_ERROK:     equ    %00010000
MSGTYP_ERRPOLL:   equ    %00001001
MSGTYP_GETDATA:   equ    %00001010
MSGTYP_STRDATA:   equ    %00001011
MSGTYP_DATATX:    equ    %00001100

```

```

MSGTYP_QRYROW:    equ %00001101
MSGTYP_QRYCOL:    equ %00001110
MSGTYP_ROWRTN    equ %00001111
MSGTYP_COLRTN:    equ %00010000

MSGTYP_DISPNEXT: equ %00010001
MSGTYP_NEWBMP:    equ %00010010
MSGTYP_LSIDX:     equ %00010011
MSGTYP_APPEFF:    equ %00010100
MSGTYP_NEWROWT:   equ %00010101
MSGTYP_NEWROWB:   equ %00010111
MSGTYP_NEWCOLL:   equ %00011000
MSGTYP_NEWCOLR:   equ %00011001

```

```

        ORG    $0000 ; start of internal RAM

```

```

MYID:      fcb    1 byte      ; MSB set on slave
INDEX:     fcb    1 byte      ; an extra index variable
CLOCK:     fcb    1 byte      ; a count incremented each pass
through the refresh loop
*STATUS byte    1 byte      ; a second is in the spec
STATUS:       fcb    $00
*Recv Buffer    8 bytes
RXBUFF:       fdb $0000, $0000, $0000, $0000
*Recv Count    2 byte
RXPOS:        fdb #RXBUFF
*Send Buffer    8 bytes
TXBUFF:       fdb $0000, $0000, $0000, $00ed
*Send Count 2 byte
TXPOS:        fdb #TXBUFF
*Current Display state 24 bytes
DISPLAY:      fdb $0000, $0000, $0000, $0000, $0000, $0000, $0000,
$0000, $0000, $0000, $0000, $0000
*Next Display state 24 bytes
NXTDISP:      fdb $0000, $0000, $0000, $0000, $0000, $0000, $0000,
$0000, $0000, $0000, $0000, $0000
LIGHTSHOW:    fdb 0 ;2 byte ; light show number

LIGHTINDEX:   fcb 0 ;1 byte ; an index for which stage of the
light show
*
* ; is up
COLFLAGS:     fcb DB7 ;1 byte a byte to store the operations to be
* performed on each column
* DB7 lightshow
* DB6 displaynext

*Stack space should be adjusted to the end of the internal ram 64 bytes
*here listed
* should be placed from $0083 to $00C3
        bsz    12
        fdb    $00,$00,$00,$00 , $00,$00,$00,$00, $00,$00,$00,$00,
$00,$00,$00
STACK: fdb    $00

*        ORG    $E000 ; store testing data

```



```

BCLR  SCCR1,x DB4+DB3    ; set 8 bit characters and wake on idle

cli   ; clear interrupt mask and enable interrupts
BSET  SCCR2,x DB5+DB3+DB2 ; enable SCI system transmit and
*
      recieve
      tst     SCSR,x

      ldaa   #MSGTYP_IMHERE    ; send_msg gets the type from A
*
      ; this is the top byte of the serial number
      jsr   send_msg

      jsr   delay_10ms        ;wait 30 ms for reply
      jsr   delay_10ms
      jsr   delay_10ms

      tst   RXBUFF+1         ; if the message type byte is still 0
      beq   MASTER_INIT     ; assume master

      jsr   delay_10ms      ; another 10ms for the the remainder of the
message

      ldaa   #MSGTYP_URID      ; error if some other message received
      cmpa  RXBUFF+1
      bne   INIT_ERROR

      ldaa   #TXBUFF+8        ; check for complete message recieved
      cmpa  TXPOS+1

      beq   SLAVE_INIT
      jmp   INIT_ERROR      ; error in initial communications, restart
*
      ; may need to be end expecting another reset

*The you are ID message is in the buffer but has not been decoded
SLAVE_INIT:
      ldaa  RXBUFF+2
      anda  #%00111111    ; strip out hi bits if present
      ora  #%10000000    ; set master direction flag in
*
      ; slave id
      staa MYID
      jsr  send_status

SLAVE:
      jmp  DISPLAYLOOP ; main loop

MASTER_INIT:
      ldaa  #$00
      staa MYID

      ldab  #$00 ;start with the current chip, b will be incremented
and used by
*
      ; send_mst_msg as to address
LOOPUP:
      tstb
      bne  send_rel_up
      ldab  #%01000000    ; set only right sideout
      stab  portb,x      ; MSB is up, 2nd MSB is right

      bra  wait_up:

```

```

send_rel_up:
    ldaa #MSGTYP_RELUP
    jsr  send_mst_msg

wait_up:
    incb
    jsr  delay_10ms
    lda  #RXBUFF+8 ; check for full receive buffer
    cmpa RXPOS     ; if buffer is not full
    bne  LOOPDOWN  ; loop back down the column
    jsr  init_slave
    bra  LOOPUP

LOOPDOWN:
    decb                ; Decrements here to counter increment
during wait
    andb #%111         ; loop until returned to bottom row
*
    beq  LOOPRIGHT
    ldaa #MSGTYP_RELRIGHT;
    jsr  send_mst_msg
    bra  LOOPDOWN

LOOPRIGHT:
    tstb
    bne  send_rel_right
    ldb  #%00000000 ; clear sideouts
    stab portb,x    ; MSB is up, 2nd MSB is right

    bra  wait_right
send_rel_right:
    ldaa #MSGTYP_RELRIGHT
    jsr  send_mst_msg
wait_right:
    addb #%1000        ; Move to the next column
    jsr  delay_10ms
    lda  #RXBUFF+8 ; check for full receive buffer
    cmpa RXPOS     ; if buffer is not full
    bne  MASTER    ; goto main loop
    jsr  init_slave
    bra  LOOPUP

MASTER:
    BSET COLFLAGS DB7 ; set flag for lightshow
*
    LDAB #DB7
*
    STAB COLFLAGS
    nop

DISPLAYLOOP:
    BCLR COLFLAGS DB6 ; clear display next flag
*will need logic to set queued disp next message

    inc  CLOCK

    LDAB #DISPLAY ; Y points to DISPLAY
    LDAA #0
    XGDY

```



```

*          BSR      LIGHTCOLUMN
*          BSR      NXTDSPCOLUMN

          CMPB     #$19
          BEQ     DISPLAYLOOP
          BRA     NEXTCOL

END:      JMP      END      ;end here

NXTDSPCOLUMN:
          psha
          ldaa   $18,y      ; set current column
          staa   0,y        ; y points to current col, 0x18+y
          pula
          rts

LIGHTCOLUMN:
          cmpb   LIGHTINDEX
          bne    nonLC

          psha
          ldaa   #$FF      ; set current column
          staa   $18,y     ; y points to current col, 0x18+y
          pula
          bra    endLC

nonLC:
          clr    $18,y     ; is same column in NEXTDISP
endLC:
          rts

LIGHTMAINT:
          pshb
          ldab   CLOCK
          andb   #%00011111
          bne    endLM

lsinc:
          BSET   COLFLAGS DB6 ;          When lightshow increments
*                                     ;          set to display next on this
pass

          inc    LIGHTINDEX
          ldab   #$19
          cmpb   LIGHTINDEX
          bne    endLM
          clr    LIGHTINDEX
endLM:
          pulb
          rts

lightshow:
                                     ldab   LIGHTINDEX
                                     bsr    shift_column
                                     incb

```

```

                                cmpb    #$18
    beq    lsrl    ; roll over lightshow at 0x18
    stab  LIGHTINDEX
    pulb
    rts

lsrl:
    ldab   #$FF
    STAB  NXTDISP
    clr   LIGHTINDEX
    pulb
    rts

*called with a set to the new value for the first column
*returns old last column in a
shift_column:
                                clr     INDEX
                                pshx
                                pshb

                                ldab   #$18
                                stab   INDEX
                                ldx    #NXTDISP

sc_loop:
                                ldab   0,x    ; get old value
                                staa   0,x    ; store new
value
                                inx

                                tba                ; transfer old
value to
*                                ; a as the next new value
                                dec     INDEX
                                tst     INDEX
                                bne     sc_loop

                                pulb
                                pulx
                                rts

displaynext:
                                clr     INDEX
                                pshx
                                psha
                                pshb

                                ldab   #$18
                                stab   INDEX
                                ldx    #NXTDISP
                                ldy    #DISPLAY

dnc_loop:
                                ldab   0,x
                                stab   0,y
                                iny
                                inx

                                dec     INDEX

```

```

                                tst     INDEX
                                bne     dnc_loop

                                pulb
                                pula
                                pulx
                                rts

init_slave:
    ldab     #TXBUFF+2
    ldaa    #MSGTYP_URID;
    bsr     send_mst_msg

    rts

send_status:
    ldaa    STATUS
    stab    TXBUFF+2
    ldaa    #0
    ldaa    #MSGTYP_ORERR    ; send_msg gets the type from A
    jsr     send_msg

send_mst_msg:
*           ; a contains the message number,
*           ; b contains the to address
*           ; all other bytes are already in txbuff
    pshb    ; b will used for scratch
    ldab    TXPOS+1

    ldx    #REGBLK

    cmpb    #TXBUFF+8
    bne     send_msg    ; loop if send not complete

    staa    TXBUFF+1    ; put message number in position 2
*           ; allows for abstraction and rearrangement if
needed

    ldb     #TXBUFF+1    ; reset transmit position to start of
buffer
    stab    TXPOS+1

*Multicast must be handled here ish
    pulb    ; get to address from a
    stab    SCDR,x    ; send address as first byte of message,
*           ; sci interrupt will handle the remainder byte
by byte

    rts

* a contains the message number, all other bytes are already in txbuff
send_msg:
    pshb    ; b will used for scratch

*debug hack

```

```

*      psha
*      pshb
*      pshx
*      ldx    TXPOS          ; display TXbuffer
*      ldab   #7             ; send out 7 characters
*      jsr    lcd_line1
*      pulx
*      pulb
*      pula
*end debug hack

      ldab    TXPOS+1

      ldx    #REGBLK

      cmpb   #TXBUFF+8
      bne    send_msg        ; loop if send not complete
*                                     passes 1x 3/27/03

      staa   TXBUFF+1        ; put message number in position 2
*                                     ; allows for abstraction and rearrangement if
needed

      ldb    #TXBUFF+1      ; reset transmit position to start of buffer
*                                     ; plus one for the byte we're about to send.

      stab   TXPOS+1

*Multicast must be handled here ish
*                                     ; put MYID in position 1 of buffer
      ldab   MYID           ; direction flag to master is set in ID

      BSET   SCCR2,x DB7+DB3 ; enable SCI system TCDR interrupt and
Transmitter
      TST    SCSR,x         ; read status register
      stb    SCDR,x         ; send MYID as first byte of message,
*                                     ; sci interrupt will handle the remainder byte
by byte

      pulb
      rts

*Interrupt handler  ORGed for debug ease...
      ORG    $FE80
sci_ISR:
      ldx    #REGBLK
      brset  SCSR,x DB7 send_byte    ; TDRE send next byte
*      brset  SCSR,x DB6 ; TC Transmit complete, not needed
      brset  SCSR,x DB5 recv_byte    ; RDRF move data from SCDR to
buffer
*      brset  SCSR,x DB4 ; IDLE line not needed
      ldb    SCSR,x
      andb   %0000 1010 ; mask out bits 3 and 1
      orb    STATUS        ; add them to the status byte
      stab   STATUS        ; remember status will be sent on request
*      brset  SCSR,x DB3 ; OR overrun error, should set error flag
*      brset  SCSR,x DB2 ; NF noise flag, should be ignored

```

```

*      brset SCSR,x DB1 ; FE framing error, should set error flag
dum_ISR:
    rti

*Reads from SCDR and sends new data to RXBUFF, assumes that x has
REGBLK
*receive interrupt should be disabled during processing
recv_byte:
    ldb  #(RXBUFF+8) ; check for full receive buffer
    cmpb #RXPOS      ; if buffer is full
    bge  rb_end      ; do not receive, SET ERROR
    ldy  #RXPOS      ; reference buffer
    ldb  SCDR,x      ; pass next byte from data register
through B
    stb  0,y         ; and out to the buffer
    inc  RXPOS
rb_end: rti

*Reads from TXBUFF and sends new data to RDRF, assumes that x has
REGBLK
send_byte:
    ldb  #TXBUFF+8   ; check for pointer point past end of buffer
    cmpb TXPOS+1     ; indicating a completion
    ble  send_complete ; if B <= TXPOS branch
    ldy  TXPOS        ; reference buffer
    ldb  0,y          ; pass next byte through B
    stb  SCDR,x       ; and out to the data register
    inc  TXPOS+1
    rti
send_complete:
    BCLR  SCCR2,x DB7+DB3 ; clear SCI system TCDR interrupt
*                               and transmit enable
    ldab  #SAF
    stb   SCDR,x         ; write dummy data to disabled scdr
    rti

*loop to wait for push button on port A
wtbtn:
    brset  porta,y DB0 wtbtn
    jsr    delay_10ms
    brset  porta,y DB0 wtbtn

*sends a byte in A out port B
byte_out:
    pshx
    ldx  #REGBLK
    staa portb,x
    pulx
    rts

TEN_MS:    equ    1670 ; 6 us per loop,
10000us/6us=1670
delay_10ms:
    pshx
    ldx  #TEN_MS
dell:     dex
         inx

```

```

        dex
        bne     dell
        pulx
        rts

    cmpb MSGTYP_RELUP ; Release Up routine
    ldx #REGBLK
    ldab #%01111111      ; Replace MSB of portb with "0"
    andb portb, x
    stab portb, x
    bra proc_end_nomsg

    cmpb MSGTYP_RELRT ; Release Right routine
    ldx #REGBLK
    ldab #%10111111      ; Replace 2nd MSB of portb with "0"
    andb portb, x
    stab portb, x
    bra proc_end_nomsg

    cmpb MSGTYP_ASRTUP      ; Assert Up routine
    ldx #REGBLK
    ldab #%10000000      ; Replace MSB of portb with "1"
    orab portb, x
    stab portb, x
    bra proc_end_nomsg

    cmpb MSGTYP_ASRTRT      ; Assert Up routine
    ldx #REGBLK
    ldab #%01000000      ; Replace 2nd MSB of portb with "1"
    orab portb, x
    stab portb, x
    bra proc_end_nomsg

    cmpb MSGTYP_ERRPOLL      ; Error Poll routine
    ldab STATUS
    andb (RXBUFF+2)          ; AND status byte with error mask
    stab (TXBUFF+2)          ; Place error type into "Error/OK"
message
    ldab #$00
    stab TXBUFF              ; Message is directed to the Master, address
%0000 0000
    ldaa MSGTYP_ERROK
    bra proc_end

MSGTYP_RELUP:                equ %0000 0101
MSGTYP_RELRT:                equ %0000 0110
MSGTYP_SETMSTR:              equ %0000 0111
MSGTYP_ASRTUP:              equ %0001 1010
MSGTYP_ASRTRT:              equ %0001 1011

MSGTYP_ERROK:                equ %0001 0000
MSGTYP_ERRPOLL:              equ %0000 1001
MSGTYP_GETDATA:              equ %0000 1010
MSGTYP_STRDATA:              equ %0000 1011
MSGTYP_DATATX:               equ %0000 1100
MSGTYP_QRYROW:               equ %0000 1101
MSGTYP_QRYCOL:               equ %0000 1110

```

```

MSGTYP_ROWRTN      equ %0000 1111
MSGTYP_COLRTN:    equ %0001 0000

MSGTYP_DISPNEXT:  equ %0001 0001
MSGTYP_NEWBMP:    equ %0001 0010
MSGTYP_LSIDX:     equ %0001 0011
MSGTYP_APPEFF:    equ %0001 0100
MSGTYP_NEWROWT:   equ %0001 0101
MSGTYP_NEWROWB:   equ %0001 0111
MSGTYP_NEWCOLL:   equ %0001 1000
MSGTYP_NEWCOLR:   equ %0001 1001

        ldab RXBUFF      ; Place first byte of received message into B
        tstb
        bmi MASTER_MSG   ; If MSB is 1, message is directed to
master
        ldab MYID        ; Assumes message is for slave patch
        tstb
        beq proc_end_nomsg ; Patch is master, no need to process
        ldab RXBUFF      ; Patch is slave
        andb #%01000000
        tstb
        bne proc_multicast ; If 2nd MSB is 1, message is
multicast and must be processed.
        ldab RXBUFF
        andb #%00111111 ; Erase first two bits of address byte,
leaving patch ID
        addb #%10000000 ; Since patch must be a slave, MSB of ID
must be 1
        cmpb MYID
        beq SLAVE_MSG
        bra proc_end_nomsg

SLAVE_MSG:
        ldab (RXBUFF+1) ; Message type will be in B for all tests
in this section
        cmpb MSGTYP_RELUP ; Release Up routine
        bne proc_s_1
        ldx #REGBLK
        ldab #%0111 1111 ; Replace MSB of portb with "0"
        orab portb, x
        stab portb, x
        bra proc_end_nomsg
proc_s_1:
        cmpb MSGTYP_RELRT ; Release Right routine
        bne proc_s_2
        ldx #REGBLK
        ldab #%1011 1111 ; Replace 2nd MSB of portb with "0"
        andb portb, x
        stab portb, x
        bra proc_end_nomsg
proc_s_2:
        cmpb MSGTYP_SETMSTR ; Set Master Routine
        bne proc_s_3
        ldab #%0000 0000
        stab MYID ; Set MYID to the master ID

```

```

        ldab (RXBUFF+2)
        stab MAXPTCH           ; Set MAXPTCH address to that specified
in msg (???)
        bra proc_end_nomsg
proc_s_3:
        cmpb MSGTYP_APPEFF    ; Apply Effect routine
        bne proc_s_4
        bra proc_end_nomsg
proc_s_4:
        cmpb MSGTYP_ERRPOLL   ; Error Poll routine
        bne proc_s_5
        ldab STATUS
        stab (TXBUFF+2)       ; A second STATUS byte would go in
TXBUFF+3
        ldab #$00
        stab TXBUFF           ; Message is directed to the Master, address
%0000 0000
        ldaa MSGTYP_ERROK
        bra proc_end
proc_s_5:
        cmpb MSGTYP_DISPNEXT  ; Display Next routine
        bne proc_s_6
        pshx
        ldx #DISPLAY          ; Point X at beginning of current display
        ldaa #24               ; A is counter
proc_s_5_loop:
        ldab 24,x              ; NEXTDISP is 24 bytes after DISPLAY
        stab 0,x               ; Copy byte from NEXTDISP to
corresponding byte in DISPLAY
        inx                    ; Increment pointer x
        decb                   ; Decrement counter
        bne proc_s_5_loop      ; Loop until counter = 0
        pulx
        bra proc_end_nomsg

proc_s_6:
        cmpb MSGTYP_LSIDX     ; Light Show routine (to be developed)
        bne proc_s_7
        bra proc_end_nomsg

proc_s_7:
        cmpb MSGTYP_GETDATA    ; Get Data From Address routine
        bne proc_s_8
        pshx
        ldx (RXDATA+2)         ; Get 2-byte start address
        ldab 0,x
        stab (TXBUFF+2)       ; Get Store 4-bytes from that address
into TXBUFF+2
        ldab 1,x
        stab (TXBUFF+3)
        ldab 2,x
        stab (TXBUFF+4)
        ldab 3,x
        stab (TXBUFF+5)
        ldab #$00
        stab TXBUFF           ; Set message address to master
        ldaa MSGTYP_DATATX

```

```

        pulx
        bra proc_end

proc_s_8:
        cmpb MSGTYP_STRDATA      ; Store In Address routine
        bne proc_s_9
        pshx
        ldx (RXBUFF+2)          ; Retrieve start address from RXBUFF+2
        ldd (RXBUFF+4)          ; Retrieve 2-byte data from RXBUFF+4
        std 0,x                  ; Place data from D into address pointed
to by X
        pulx
        proc_end_nomsg

proc_s_9:
        cmpb MSGTYP_NEWCOLL     ; New Column Left routine
        bne proc_s_10
        ldab (RXBUFF+2)         ; Load 1-byte column data
        stab NEXTDISP          ; Transfer from B into start of NEXTDISP
buffer
        ldab DISPLAY            ; Retrieve old left column
        stab (TXBUFF+2)
        ldab #$00
        stab TXBUFF            ; Address message to master
        proc_end_nomsg

proc_s_10
        cmpb MSGTYP_NEWCOLR     ; New Column Right routine
        bne proc_s_11
        ldab (RXBUFF+2)         ; Retrieve column data
        stab (NEXTDISP+23)      ; Place into end of NEXTDISP buffer
        proc_end_nomsg

proc_s_11
        cmpb MSGTYP_NEWROWT     ; New Top Row routine
        bne proc_s_12          ; Incoming bytes refer to top LEDs
        pshx                    ; To change this in the column-oriented display
buffer,
        pshy                    ; the first bit of each column byte must be
replaced.
        ldaa #8                  ; Use A as counter
        ldab #$00
        stab (TXBUFF+2)
        stab (TXBUFF+3)
        stab (TXBUFF+4)         ; Initialize row record in TX buffer
        ldx #DISPLAY            ; Point X at start of DISPLAY buffer
        ldy #NEXTDISP          ; Point X at start of NEXTDISP buffer
proc_s_11_loop:
        ldab #%1000 0000
        andb 0,x
        addb (TXBUFF+2)         ; Store old value in TXBUFF
        ldab #%1000 0000
        andb (RXBUFF+2)         ; Strip MSB from incoming row data
        bclr 0,y #%1000 0000    ; Clear old column bit
        addb 0,y
        stab 0,y                ; Replace with new column bit

```

```

    ldab #%1000 0000      ; Same as above, for columns 8-16
    andb 8,x
    addb (TXBUFF+3)
    ldab #%1000 0000
    andb (RXBUFF+3)
    bclr 8,y #%1000 0000
    addb 8,y
    stab 8,y

    ldab #%1000 0000      ; Same as above, for columns 17-24
    andb 16,x
    addb (TXBUFF+4)
    ldab #%1000 0000
    andb (RXBUFF+4)
    bclr 16,y #%1000 0000
    addb 16,y
    stab 16,y

    rol (RXBUFF+2)        ; Rotate bits of incoming row data and
outgoing row data,
    rol (RXBUFF+3)        ; Since all operations are performed on
MSB
    rol (RXBUFF+4)        ; After one loop (8 iterations), bits are
in correct position
    rol (TXBUFF+2)
    rol (TXBUFF+3)
    rol (TXBUFF+4)

    deca                  ; decrement counter
    inx                   ; increment pointers
    iny
    bne proc_s11_loop     ; Loop if counter not zero
    ldaa MSGTYP_ROWRTN    ; Message type into A
    ldab #$00
    stab TXBUFF           ; Set message address to master address
    puly
    pulx
    proc_end

proc_s_12
    cmpb MSGTYP_NEWROWT   ; New Bottom Row routine
    bne proc_s_13         ; Incoming bytes refer to bottom LEDs
    pshx                  ; To change this in the column-oriented display
buffer,
    pshy                  ; the last bit of each column byte must be
replaced.
    ldaa #8               ; Use A as counter
    ldab #$00
    stab (TXBUFF+2)
    stab (TXBUFF+3)
    stab (TXBUFF+4)      ; Initialize row record in TX buffer
    ldx #DISPLAY          ; Point X at start of DISPLAY buffer
    ldy #NEXTDISP        ; Point X at start of NEXTDISP buffer
proc_s_12_loop:
    ldab #%0000 0001
    andb 0,x
    addb (TXBUFF+2)      ; Store old value of LSB in TXBUFF

```

```

ldab #%1000 0000
andb (RXBUFF+2)          ; Strip MSB from incoming row data
bclr 0,y #%0000 0001    ; Clear old column bit
rolb                    ; Make MSB of incoming byte into LSB in B
addb 0,y
stab 0,y                ; Replace new column bit

ldab #%0000 0001        ; Same as above, for columns 8-16
andb 8,x
addb (TXBUFF+3)
ldab #%1000 0000
andb (RXBUFF+3)
bclr 8,y #%0000 0001
rolb
addb 8,y
stab 8,y

ldab #%0000 0001        ; Same as above, for columns 17-24
andb 16,x
addb (TXBUFF+4)
ldab #%1000 0000
andb (RXBUFF+4)
bclr 16,y #%0000 0001
rolb
addb 16,y
stab 16,y

rol (RXBUFF+2)          ; Rotate bits of incoming row data and
outgoing row data,
rol (RXBUFF+3)          ; Since all operations are performed on
MSB (Yet stored in TXBUFF in LSB)
rol (RXBUFF+4)          ; After one loop (8 iterations), bits are
in correct position
rol (TXBUFF+2)
rol (TXBUFF+3)
rol (TXBUFF+4)

deca                    ; decrement counter
inx                     ; increment pointers
iny
bne proc_s11_loop       ; Loop if counter not zero
ldaa MSGTYP_ROWRTN     ; Message type into A
ldab #$00
stab TXBUFF             ; Set message address to master address
proc_end

MASTER_MSG:            ; handles messages intended for master patch
proc_m_1:

proc_end_nomsg:
ldaa #0
proc_end:
rts

```

Appendix C

```
// Stevens Institute of Technology
// Department of Electrical and Computer Engineering
// Group 37 - Modular LED Signboard
// Computer Monitor Program
// http://www.easysw.com/~mike/serial/serial.html

#include <stdio.h>          /* Standard i/o defns. */
#include <string.h>         /* String function defns. */
#include <unistd.h>        /* UNIX standard function defns. */
#include <fcntl.h>         /* File ctrl defns. */
#include <errno.h>         /* Error number defns. */
#include <termios.h>       /* POSIX terminal ctrl defns. */

#define TYPE 0x01
#define VERS 0x01
#define SERIESID 0x000001

unsigned char inmessage[8];

int openport()
{
    int fd;          /* File descriptor for the port */

    fd = open("/dev/ttyS1", O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1)
    {
        perror("open_port: Unable to open /dev/ttyS1 - ");
    }
    else
    {
        fcntl(fd, F_SETFL, 0);
        return(fd);
    }
}

/* configure */
void configport(int fd)
{
    struct termios options;

    /* get options */
    tcgetattr(fd, &options);

    /* set baud */
    cfsetispeed(&options, B9600);
    cfsetospeed(&options, B9600);

    /* set char size */
    options.c_cflag &= ~CSIZE; /* mask the char size bits */
    options.c_cflag |= CS8;    /* select 8 data bits */

    /* set parity */
    /* none */
    options.c_cflag &= ~PARENB;
    options.c_cflag &= ~CSTOPB;
    options.c_cflag &= ~CSIZE;
```

```

options.c_cflag |= CS8;

/* enable the receiver and set local mode */
options.c_cflag |= (CLOCAL | CREAD);

/* set hardware flow control */
//options.c_cflag
options.c_cflag |= CRTSCTS;
//disable: options.c_cfla &= ~CNEW_RTSCCTS;

/* set new options */
tcsetattr(fd, TCSANOW, &options);
}

int checkmessage()
{
    int checksum, sum;

    checksum = (inmessage[6] << 8) | inmessage[7];
    sum = inmessage[0] + inmessage[1] + inmessage[2] + inmessage[3] +
inmessage[4] + inmessage[5];
    return 1;
    if (sum == checksum) return 1;
    else return 0;
}

int messdecode()
{
    unsigned char messadd;
    int messtype;

    printf("Message Received:\n");
    printf("-----\n");
    printf("raw message: 0x%02x%02x%02x%02x %02x%02x%02x%02x\n\n",
inmessage[0] & 0xFF,
        inmessage[1] & 0xFF, inmessage[2] & 0xFF, inmessage[3] &
0xFF, inmessage[4] & 0xFF,
        inmessage[5] & 0xFF, inmessage[6] & 0xFF, inmessage[7] &
0xFF);
    messadd = inmessage[0];
    printf("Direction: %2d\n", (messadd & 0x80) >> 7);
    printf("Multicast: %2d\n", (messadd & 0x40) >> 6);
    printf("X Coordinate: %d\n", (messadd & 0x38) >> 3);
    printf("Y Coordinate: %d\n", messadd & 0x07);
    messtype = inmessage[1];
    switch(messtype)
    {
        case 1:
            printf("    Message Type: I'm here\n");
            printf("    Serial Number: \n");
            printf("        Version: %02x\n", inmessage[2]);
            printf("        Series ID: %d%d%d\n\n",
inmessage[3], inmessage[4], inmessage[5]);
            break;
        case 2:
            printf("    Message Type: I'm master\n");
            printf("        Master Type: \n");

```

```

        printf("                Type: %02x\n", inmessage[2]);
        printf("                Version: %02x\n\n", inmessage[3]);
        break;
case 3:
    printf("                Message Type: You are ID#\n\n");
    break;
case 5:
    printf("                Message Type: Release up\n\n");
    break;
case 6:
    printf("                Message Type: Release right\n\n");
    break;
case 7:
    printf("                Message Type: Set master\n");
    printf("                Rectangular: %02x\n", inmessage[2] &
0x40);
    printf("                X Coordinate: %02x\n", inmessage[2] &
0x38);
    printf("                Y Coordinate: %02x\n\n", inmessage[2] &
0x07);
    break;
case 28:
    printf("                Message Type: Assert up\n\n");
    break;
case 29:
    printf("                Message Type: Assert right\n\n");
    break;

case 8:
    printf("                Message Type: Error/OK\n");
    printf("                Error Type: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 9:
    printf("                Message Type: Error poll\n");
    printf("                Error Mask: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 10:
    printf("                Message Type: Get data from address\n");
    printf("                Start Address: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 11:
    printf("                Message Type: Store in address\n");
    printf("                Start Address: %02x%02x\n",
inmessage[2], inmessage[3]);
    printf("                Data: %02x%02x\n\n",
inmessage[4], inmessage[5]);
    break;
case 12:
    printf("                Message Type: Data transmit\n");
    printf("                Data: %02x%02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4], inmessage[5]);
    break;
case 13:
    printf("                Message Type: Query row\n");

```

```

        printf("      Row: %d\n\n", inmessage[2]);
        break;
case 14:
    printf("      Message Type: Query column\n");
    printf("      Column: %d\n\n", inmessage[2]);
    break;
case 15:
    printf("      Message Type: Row return\n");
    printf("      Data: %02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4]);
    break;
case 16:
    printf("      Message Type: Column return\n");
    printf("      Data: %02x\n\n", inmessage[2]);
    break;
case 17:
    printf("      Message Type: Display next\n\n");
    break;
case 32:
    printf("      Message Type: New bitmap\n");
    printf("      Column 1 Data: %02x\n", inmessage[2]);
    printf("      Column 2 Data: %02x\n", inmessage[3]);
    printf("      Column 3 Data: %02x\n", inmessage[4]);
    printf("      Column 4 Data: %02x\n\n", inmessage[5]);
    break;
case 33:
    printf("      Message Type: New bitmap\n");
    printf("      Column 5 Data: %02x\n", inmessage[2]);
    printf("      Column 6 Data: %02x\n", inmessage[3]);
    printf("      Column 7 Data: %02x\n", inmessage[4]);
    printf("      Column 8 Data: %02x\n\n", inmessage[5]);
    break;
case 34:
    printf("      Message Type: New bitmap\n");
    printf("      Column 9 Data: %02x\n", inmessage[2]);
    printf("      Column 10 Data: %02x\n", inmessage[3]);
    printf("      Column 11 Data: %02x\n", inmessage[4]);
    printf("      Column 12 Data: %02x\n\n", inmessage[5]);
    break;
case 35:
    printf("      Message Type: New bitmap\n");
    printf("      Column 13 Data: %02x\n", inmessage[2]);
    printf("      Column 14 Data: %02x\n", inmessage[3]);
    printf("      Column 15 Data: %02x\n", inmessage[4]);
    printf("      Column 16 Data: %02x\n\n", inmessage[5]);
    break;
case 36:
    printf("      Message Type: New bitmap\n");
    printf("      Column 17 Data: %02x\n", inmessage[2]);
    printf("      Column 18 Data: %02x\n", inmessage[3]);
    printf("      Column 19 Data: %02x\n", inmessage[4]);
    printf("      Column 20 Data: %02x\n\n", inmessage[5]);
    break;
case 37:
    printf("      Message Type: New bitmap\n");
    printf("      Column 21 Data: %02x\n", inmessage[2]);
    printf("      Column 22 Data: %02x\n", inmessage[3]);

```

```

        printf("    Column 23 Data: %02x\n", inmessage[4]);
        printf("    Column 24 Data: %02x\n\n", inmessage[5]);
        break;
    case 19:
        printf("    Message Type: Light show index\n");
        printf("    Light Show Number: %02x%02x\n",
inmessage[2], inmessage[3]);
        printf("    Position: %02x\n\n", inmessage[4]);
        break;
    case 20:
        printf("    Message Type: Apply Effect\n");
        printf("    Effect Number: %02x\n", (inmessage[2] &
0xf8) >> 3);
        printf("    Subpatch: %02x\n", inmessage[2] & 0x07);
        printf("    Parameter Number: %02x\n\n",
inmessage[3]);
        break;
    case 21:
        printf("    Message Type: New row - top\n");
        printf("    Data: %02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4]);
        break;
    case 23:
        printf("    Message Type: New row - bottom\n");
        printf("    Data: %02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4]);
        break;
    case 25:
        printf("    Message Type: New column - left\n");
        printf("    Data: %02x\n\n", inmessage[2]);
        break;
    case 26:
        printf("    Message Type: New column - right\n");
        printf("    Data: %02x\n\n", inmessage[2]);
        break;
    default:
        printf("John's Mom!!!!");
    }

    fflush(stdout);
    return 1;
}

int main(int argc, char *argv[])
{
    int fd, num_errs, i, index;

    fd = openport();
    configport(fd);

    while(1){
        for(index=0; index<8; index++){
            inmessage[index]=0;
        }
        for(index=0; index<8; index++){
            read(fd, inmessage+index, 1);
        }
        num_errs = 0;

```

```

        while (!checkmessage(inmessage) && (num_errs < 3))
        {
            printf("Message invalid: ");
            for (i = 0; i <= 7; i++) printf("%02x", inmessage[i]),
fflush(stdout);
            printf("\n");
            read(fd, inmessage, 8);
            num_errs++;
        }
        if (num_errs == 3)
        {
            printf("Cannot receive valid messages.\n");
            return 1;
        }
        messsdecode(inmessage);
    }

    while(read(fd, inmessage, 8) != 0)
    {
        num_errs = 0;
        while (!checkmessage(inmessage) && (num_errs < 3))
        {
            printf("Message invalid: ");
            for (i = 0; i <= 7; i++) printf("%02x", inmessage[i]),
fflush(stdout);
            printf("\n");
            read(fd, inmessage, 8);
            num_errs++;
        }
        if (num_errs == 3)
        {
            printf("Cannot receive valid messages.\n");
            return 1;
        }
        messsdecode(inmessage);
    }
    close(fd);
    return 0;
}

```

```

// Stevens Institute of Technology
// Department of Electrical and Computer Engineering
// Group 37 - Modular LED Signboard
// Message Sender Program
// ref: http://www.easysw.com/~mike/serial/serial.html

#include <stdio.h>          /* Standard i/o defns. */
#include <string.h>        /* String function defns. */
#include <unistd.h>        /* UNIX standard function defns. */
#include <fcntl.h>         /* File ctrl defns. */
#include <errno.h>         /* Error number defns. */
#include <termios.h>       /* POSIX terminal ctrl defns. */
#include <time.h>          /* Time function defns. */

#define TYPE 0x01
#define VERS 0x01
#define SERIESID 0x000001

unsigned char outmessage[8];
unsigned char inmessage[8];
int ArraySizeX;
int ArraySizeY[0,0,0,0,0,0,0,0];

int PortOpen(void);
void PortConfig(int fd);
void Boot(int fd);
void getoutmessage(int fd);
unsigned char messdecode(void);
void ChecksumCreate(void);
int ChecksumTest(void);
void MConstruct(int fd, unsigned char messadd, unsigned char messtype,
int data);
int MFormatData2_2(short data1, short data2);
int MFormatData2_1_1(short data1, unsigned char data2, unsigned char
data3);
int MFormatData3_1(int data1, unsigned char data2);
int MFormatData1_1_1_1(unsigned char data1, unsigned char data2,
unsigned char data3, unsigned char data4);
short MFormat_ET(unsigned char req_reset, unsigned char type_reset,
unsigned char chksum_stat);
unsigned char MFormat_AE(unsigned char effect_num, unsigned char
subpatch);
unsigned char MFormat_SM(unsigned char rectangular, unsigned char x,
unsigned char y);
unsigned char MFormat_MA(unsigned char dir, unsigned char multi,
unsigned char x, unsigned char y);
unsigned char ColumnInit(int fd, unsigned char x);
void LightShow(int fd);

int main(int argc, char **argv[])
{
    int fd, num_errs, rin, data, i;
    unsigned char messadd, messtype, data1, data2, data3, data4;
    fd_set selectors;

    struct timeval now;

```

```

    fd = PortOpen();
    PortConfig(fd);
    printf("Enter 1 for snoop mode, 2 for send mode, or 3 for master
mode.\n");
    rin = getchar();
    switch (rin)
    {
        case '1':
            while(read(fd,inmessage,8) != 0)
            {
                num_errs = 0;
                while (ChecksumTest() && (num_errs < 3))
                {
                    printf("Message invalid.\n");
                    read(fd,inmessage,8);
                    num_errs++;
                }
                if (num_errs == 3)
                {
                    printf("Cannot receive valid
messages.\n");

                    return 1;
                }
                messdecode();
                for(i = 0; i <=7; i++) inmessage[i] = 0;
            }
            break;
        case '2':
            while (1) getoutmessage(fd);
            break;
        case '3':
            Boot(fd);
            LightShow(fd);
            break;
        default:
            break;
    }
}

int PortOpen()
{
    int fd;      /* File descriptor for the port */

    fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1)
    {
        perror("open_port: Unable to open /dev/ttyS0 - ");
    }
    else
    fcntl(fd, F_SETFL, 0);
    return(fd);
}

/* configure */
void PortConfig(int fd)
{
    struct termios options;

```

```

/* get options */
tcgetattr(fd, &options);

/* set baud */
cfsetispeed(&options, B9600);
cfsetospeed(&options, B9600);

/* set char size */
options.c_cflag &= ~CSIZE; /* mask the char size bits */
options.c_cflag |= CS8;    /* select 8 data bits */

/* set parity */
/* none */
options.c_cflag &= ~PARENB;
options.c_cflag &= ~CSTOPB;
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;

/* enable the receiver and set local mode */
options.c_cflag |= (CLOCAL | CREAD);

/* set hardware flow control */
//options.c_cflag
options.c_cflag |= CRTSCTS;
//disable: options.c_cfla &= ~CNEW_RTSCCTS;

/* set new options */
tcsetattr(fd, TCSANOW, &options);
}

int ChecksumTest()
//checks the checksum of incoming messages
{
    int checksum, sum;

    printf("messdecode: %02x %02x %02x %02x %02x %02x %02x %02x\n",
inmessage[0] & 0xFF,
        inmessage[1] & 0xFF, inmessage[2] & 0xFF, inmessage[3] &
0xFF,
        inmessage[4] & 0xFF, inmessage[5] & 0xFF, inmessage[6] &
0xFF, inmessage[7] & 0xFF);
    checksum = (inmessage[6] << 8) | inmessage[7];
    sum = inmessage[0] + inmessage[1] + inmessage[2] + inmessage[3] +
inmessage[4] + inmessage[5];
    if (sum == checksum) return 1;
    else return 0;
}

unsigned char MFormat_MA(unsigned char dir, unsigned char multi,
    unsigned char x, unsigned char y)
//constructs messadd
{
    unsigned char messadd;

    messadd = ((dir & 0x01) << 7) | ((multi & 0x01) << 6) | ((x &
0x07) << 3) | ((y & 0x07) << 0);
}

```

```

        return messadd;
    }

    unsigned char MFormat_SM(unsigned char rectangular, unsigned char x,
    unsigned char y)
    //returns the special set master byte
    {
        unsigned char output = ((rectangular & 0x01) << 7) | ((x & 0x07)
    << 3) | (y & 0x07);
        return output;
    }

    short MFormat_ET(unsigned char req_reset, unsigned char type_reset,
    unsigned char chksum_stat)
    //returns the special error type byte
    {
        short output = ((req_reset & 0x01) << 7) | ((type_reset & 0x01)
    << 6)
            | (chksum_stat & 0x01);
        return output;
    }

    unsigned char MFormat_AE(unsigned char effect_num, unsigned char
    subpatch)
    //returns the special apply effect byte
    {
        unsigned char output = ((effect_num & 0xF8) << 4) | (subpatch &
    0x07);
        return output;
    }

    int MFormatData1_1_1_1(unsigned char data1, unsigned char data2,
    unsigned char data3, unsigned char data4)
    //take four separate bytes and returns an int
    {
        int data = (data1 << 24) | (data2 << 16) | (data3 << 8) | data4;
        return data;
    }

    int MFormatData3_1(int data1, unsigned char data2)
    //takes 3 bytes and 1 byte and returns an int
    {
        int data = ((data1 & 0xFF0000) << 24) | ((data1 & 0xFF00) << 16)
    | ((data1 & 0xFF) << 8)
            | data2;
        return data;
    }

    int MFormatData2_1_1(short data1, unsigned char data2, unsigned char
    data3)
    //takes 2 bytes and 1 byte and 1 byte and returns an int
    {
        int data = ((data1 & 0xFF00) << 24) | ((data1 & 0xFF) << 16) |
    (data2 << 8) | data3;
        return data;
    }

```

```

int MFormatData2_2(short data1, short data2)
//takes 2 bytes and 2 bytes and returns an int
{
    int data = ((data1 & 0xFF00) << 24) | ((data1 & 0xFF) << 16) |
((data2 & 0xFF00) << 8)
    | (data2 & 0xFF);
    return data;
}

void MConstruct(int fd, unsigned char messadd, unsigned char messtype,
int data)
//takes messadd, messtype, and data and constructs and sends the
outgoing message
{
    short i = 0;

    outmessage[0] = messadd;
    outmessage[1] = messtype;
    outmessage[2] = (data & 0xFF000000) >> 24;
    outmessage[3] = (data & 0xFF0000) >> 16;
    outmessage[4] = (data & 0xFF00) >> 8;
    outmessage[5] = (data & 0xFF);
    outmessage[6] = 0;
    outmessage[7] = 0;
    ChecksumCreate();
    write(fd, outmessage, 8);
    fsync(fd);
}

void ChecksumCreate()
//takes the outgoing message and returns its checksum
{
    int sum, i;

    sum = 0;
    for (i = 0; i <= 7; i++) sum += outmessage[i];
    outmessage[6] = (sum & 0xFF00) >> 8;
    outmessage[7] = sum & 0xFF;
}

void getoutmessage(int fd)
{
    int dir, multi, x, y, temp, i;

    printf("Direction [set if to master]: ");
    scanf("%2d", &dir);
    printf("Multicast [set if multicast]: ");
    scanf("%2d", &multi);
    printf("X coordinate: ");
    scanf("%2d", &x);
    printf("Y coordinate: ");
    scanf("%2d", &y);
    outmessage[0] = MFormat_MA(dir, multi, x, y);
    printf("MessType: ");
    scanf("%2d", &temp);
    outmessage[1] = (temp & 0xff);
    printf("Data Byte 1: ");
}

```

```

scanf("%02x", &temp);
outmessage[2] = temp;
printf("Data Byte 2: ");
scanf("%02x", &temp);
outmessage[3] = temp;
printf("Data Byte 3: ");
scanf("%02x", &temp);
outmessage[4] = temp;
printf("Data Byte 4: ");
scanf("%02x", &temp);
outmessage[5] = temp;
outmessage[6] = 0;
outmessage[7] = 0;
ChecksumCreate();
for (i = 0; i <= 7; i++)
{
    printf("%02x",outmessage[i]), fflush(stdout);
}
write(fd,outmessage,8);
fsync(fd);
}

```

```

void Boot(int fd)
{
    unsigned char messadd, x, y;
    struct timeval now;
    fd_set selectors;
    int i;

    for (i = 0; i <= 7; i++) ArraySizeY[i] = 0;
    x = 0;
    y = 0;
    now.tv_sec = 5;
    now.tv_usec = 0;
    FD_ZERO(&selectors);
    FD_SET(fd, &selectors);
    while (select(fd+1,&selectors,NULL,NULL,&now))
    {
        y = ColumnInit(fd,x);
        for (y; y >= 0; y--)
        {
            messadd = MFormat_MA(0,0,x,y);
            MConstruct(fd,messadd,6,0);
        }
        ArraySizeY[x] = y;
        x += 1;
    }
    ArraySizeX = (x - 1);
}

```

```

unsigned char ColumnInit(int fd, unsigned char x)
//brings a column of patches into the array
//returns the height of the column
{
    int i, num_errs = 0;
    unsigned char messadd, y = 0;
    int data;

```

```

struct timeval now;
fd_set selectors;

now.tv_sec = 5;
now.tv_usec = 0;
FD_ZERO(&selectors);
FD_SET(fd, &selectors);
while (select(fd+1, &selectors, NULL, NULL, &now))
    {
        read(fd, inmessage, 8);
        while (!ChecksumTest() && (num_errs < 3))
            {
                printf("Message invalid: 0x");
                for (i = 0; i <= 7; i++)
printf("%02x", inmessage[i]), fflush(stdout);
                printf("\n");
                read(fd, inmessage, 8);
                num_errs++;
            }
        messadd = MFormat_MA(0, 0, 0, 0);
        data = MFormatData1_1_1_1(TYPE, VERS, 0, 0);
        MConstruct(fd, messadd, 2, data);
        messadd = MFormat_MA(0, 0, x, y);
        MConstruct(fd, messadd, 3, 0);
        messadd = MFormat_MA(0, 0, x, y);
        MConstruct(fd, messadd, 5, 0);
        y += 1;
    }
return y;
}

unsigned char messdecode()
//decodes incoming messages and prints results to stdout
{
    unsigned char messadd, messtype;

    printf("messdecode: %02x %02x %02x %02x %02x %02x %02x %02x\n",
inmessage[0] & 0xFF,
        inmessage[1] & 0xFF, inmessage[2] & 0xFF, inmessage[3] &
0xFF,
        inmessage[4] & 0xFF, inmessage[5] & 0xFF, inmessage[6] &
0xFF, inmessage[7] & 0xFF);
    messadd = inmessage[0];
    printf("Direction: %02x\n", (messadd & 0x80) >> 7);
    printf("Multicast: %02x\n", (messadd & 0x40) >> 6);
    printf("X Coordinate: %02d\n", (messadd & 0x38) >> 3);
    printf("Y Coordinate: %02d\n", messadd & 0x07);
    messtype = inmessage[1];
    switch(messtype)
    {
        case 1:
            printf("Message Type: I'm here\n");
            printf("Serial Number: \n");
            printf("    Version: %02d\n", inmessage[2]);
            printf("    Series ID: %02d%02d%02d\n\n",
inmessage[3], inmessage[4], inmessage[5]);
            break;
    }
}

```

```

case 2:
    printf("Message Type: I'm master\n");
    printf("Master Type: \n");
    printf("    Type: %02x\n", inmessage[2]);
    printf("    Version: %02x\n\n", inmessage[3]);
    break;
case 3:
    printf("Message Type: You are ID#\n\n");
    break;
case 5:
    printf("Message Type: Release up\n\n");
    break;
case 6:
    printf("Message Type: Release right\n\n");
    break;
case 7:
    printf("Message Type: Set master\n");
    printf("Rectangular: %02x\n", inmessage[2] & 0x40);
    printf("X Coordinate: %02x\n", inmessage[3] & 0x38);
    printf("Y Coordinate: %02x\n\n", inmessage[4] &
0x07);
    break;
case 28:
    printf("Message Type: Assert up\n\n");
    break;
case 29:
    printf("Message Type: Assert right\n\n");
    break;

case 8:
    printf("Message Type: Error/OK\n");
    printf("Error Type: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 9:
    printf("Message Type: Error poll\n");
    printf("Error Mask: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 10:
    printf("Message Type: Get data from address\n");
    printf("Start Address: %02x%02x\n\n",
inmessage[2], inmessage[3]);
    break;
case 11:
    printf("Message Type: Store in address\n");
    printf("Start Address: %02x%02x\n",
inmessage[2], inmessage[3]);
    printf("Data: %02x%02x\n\n",
inmessage[4], inmessage[5]);
    break;
case 12:
    printf("Message Type: Data transmit\n");
    printf("Data: %02x%02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4], inmessage[5]);
    break;
case 13:

```

```

        printf("Message Type: Query row\n");
        printf("Row: %d\n\n", inmessage[2]);
        break;
case 14:
    printf("Message Type: Query column\n");
    printf("Column: %d\n\n", inmessage[2]);
    break;
case 15:
    printf("Message Type: Row return\n");
    printf("Data: %02x%02x%02x\n\n",
inmessage[2], inmessage[3], inmessage[4]);
    break;
case 16:
    printf("Message Type: Column return\n");
    printf("Data: %02x\n\n", inmessage[2]);
    break;

case 17:
    printf("Message Type: Display next\n\n");
    break;
case 32:
    printf("Message Type: New bitmap\n");
    printf("Column 1 Data: %02x\n", inmessage[2]);
    printf("Column 2 Data: %02x\n", inmessage[3]);
    printf("Column 3 Data: %02x\n", inmessage[4]);
    printf("Column 4 Data: %02x\n\n", inmessage[5]);
    break;
case 33:
    printf("Message Type: New bitmap\n");
    printf("Column 5 Data: %02x\n", inmessage[2]);
    printf("Column 6 Data: %02x\n", inmessage[3]);
    printf("Column 7 Data: %02x\n", inmessage[4]);
    printf("Column 8 Data: %02x\n\n", inmessage[5]);
    break;
case 34:
    printf("Message Type: New bitmap\n");
    printf("Column 9 Data: %02x\n", inmessage[2]);
    printf("Column 10 Data: %02x\n", inmessage[3]);
    printf("Column 11 Data: %02x\n", inmessage[4]);
    printf("Column 12 Data: %02x\n\n", inmessage[5]);
    break;
case 35:
    printf("Message Type: New bitmap\n");
    printf("Column 13 Data: %02x\n", inmessage[2]);
    printf("Column 14 Data: %02x\n", inmessage[3]);
    printf("Column 15 Data: %02x\n", inmessage[4]);
    printf("Column 16 Data: %02x\n\n", inmessage[5]);
    break;
case 36:
    printf("Message Type: New bitmap\n");
    printf("Column 17 Data: %02x\n", inmessage[2]);
    printf("Column 18 Data: %02x\n", inmessage[3]);
    printf("Column 19 Data: %02x\n", inmessage[4]);
    printf("Column 20 Data: %02x\n\n", inmessage[5]);
    break;
case 37:
    printf("Message Type: New bitmap\n");

```

```

        printf("Column 21 Data: %02x\n", inmessage[2]);
        printf("Column 22 Data: %02x\n", inmessage[3]);
        printf("Column 23 Data: %02x\n", inmessage[4]);
        printf("Column 24 Data: %02x\n\n", inmessage[5]);
        break;
    case 19:
        printf("Message Type: Light show index\n");
        printf("Light Show Number: %02x%02x\n",
inmessage[2],inmessage[3]);
        printf("Position: %02x\n\n", inmessage[4]);
        break;
    case 20:
        printf("Message Type: Apply Effect\n");
        printf("Effect Number: %02x\n", (inmessage[2] & 0xf8)
>> 3);
        printf("Subpatch: %02x\n", inmessage[2] & 0x07);
        printf("Parameter Number: %02x\n\n", inmessage[3]);
        break;
    case 21:
        printf("Message Type: New row - top\n");
        printf("Data: %02x%02x%02x\n\n",
inmessage[2],inmessage[3],inmessage[4]);
        break;
    case 23:
        printf("Message Type: New row - bottom\n");
        printf("Data: %02x%02x%02x\n\n",
inmessage[2],inmessage[3],inmessage[4]);
        break;
    case 25:
        printf("Message Type: New column - left\n");
        printf("Data: %02x\n\n", inmessage[2]);
        break;
    case 26:
        printf("Message Type: New column - right\n");
        printf("Data: %02x\n\n", inmessage[2]);
        break;
    }
    fflush(stdout);
    return 0;
}

void LightShow(int fd)
{
    unsigned char messadd;
    int i, j;

    for (j = 0; j <= ArraySizeY[i]; j++)
    {
        messadd = MFormat_MA(0,0,0,j);
        MConstruct(fd,messadd,24,0xff000000);
    }
}

```

Appendix D

