Programming the Sure Electronics 0832 LED matrix

Chuck Baird
avrfreaks.net, 'zbaird'

Introduction

The Sure Electronics 0832 LED matrix is a board containing four single color, 1.5 inch 8x8 LED modules driven by a Holtek HT1632 LED controller. Each board has two identical 16 pin headers and a 4 position DIP switch that permits up to 4 boards to be daisy chained together. Each board comes with a short 16 conductor ribbon cable with connectors on each end.

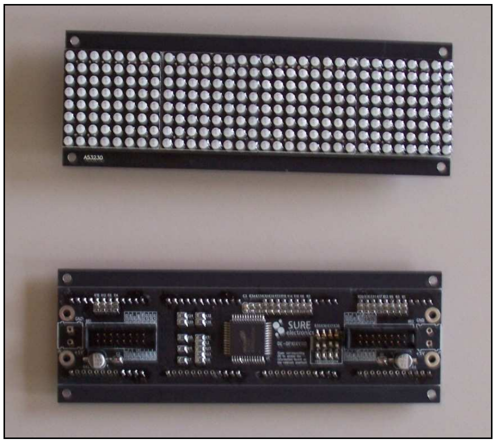

Photo 1: The Sure Electronics 0832 LED board

At the time of this writing, the boards were available through eBay, shipping from Hong Kong included, tax free, for $10 each. They are available in green, red, and yellow LEDs. Search for "0832" on eBay.

A single board requires 4 I/O lines: read, write, chip select (all active low inputs), and data (read/write). For each additional daisy chained board, an additional chip select I/O is required, for a total of 7 I/O lines to drive 4 boards (for a matrix of 8x128 LEDs). The DIP switch on each board is used to select which chip select line that board will respond to.

The board contains a volatile memory, each cell of which is mapped to an LED. For the configuration used on this board, the memory is seen by the MCU as consisting of 4 by 128 bits.

Interfacing

In general, the target board's chip select line is pulled low, and then bits are clocked (via the data line) by pulsing the read or write lines. At the end of the interaction, the chip select line is brought high which deselects the board.

Each command starts with writing 3 bits of ID followed by 9 bits (the interpretation of which depends on the command – it may be a command code, memory address, or other information). For some commands this is all that is needed. For others, additional bits are written, or bits are read (by clocking the read line and sampling the data line). Multiple memory nibble reads and writes are allowed within single command frames, as are read/alter/rewrite memory accesses. These implement an auto increment of the memory addressing, thus allowing efficient multiple memory access commands.

The boards may connect directly to a 5 volt AVR's I/O pins. The board(s) will need a separate 5 volt supply, with a common ground connection. I used an AVR ATMega16 MCU running at 8 MHz plugged into an Atmel STK500 development board, and a regulated 5 volt wall wart supply for the display. Using the supplied cables, I daisy chained four 0832 boards together, and slid them into a simple wood frame ("U" shaped with table saw kerfs to receive the board edges).

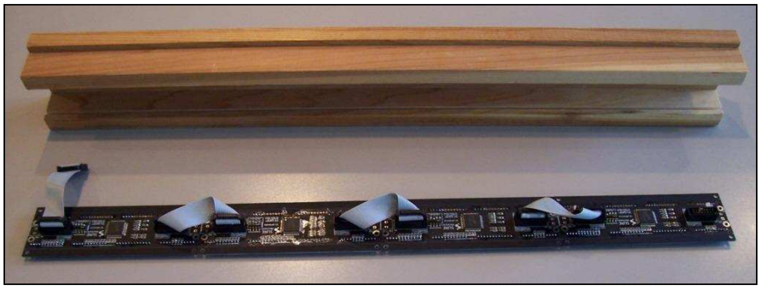

Photo 2: Back view, daisy chained boards

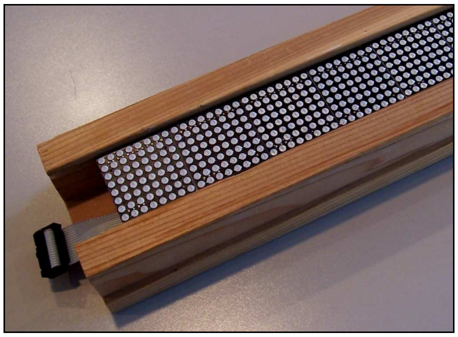

Photo 3: Front view of boards in holder

On perfboard I put a 16 pin header to receive the 0832 array cable, and a 10 pin header for a cable to connect to an AVR port on the STK500. The external supply also connects to the board, and the two headers are wired together.

Other placement options and documentation

As pointed out in the Sure Electronics documentation, the boards can be arranged in a variety of ways. I placed mine end of end (what I call “horizontal” in the code) because I wanted to play with a standard LED sign format. Each 0832’s long edges are scored and may be snapped off even with the display, so the boards can be placed in a 2x2, 4x1, or about any other arrangement. The supplied cables and the placement of the end connectors support several of these placements.

While the Sure Electronics documentation includes the controller commands and a board schematic, you may also wish to download the HT1632 datasheet from Holtek. It goes into considerably more detail and may be useful for completing your programming information.

Programming the 0832

As mentioned, the code shown below was developed on an Atmel 8 bit AVR, the ATMega16. I used the ImageCraft C compiler (a 45 day free trial version of the full compiler is available, and after 45 days it reverts to a limited code size version). Other embedded C compilers will have some syntactical differences.

One thing about the ICCAVR compiler that may be unexpected is that **chars** are unsigned. While this isn’t usually a problem one way or the other, I have tried to remember to explicitly use the **unsigned** designation where it may be important. I may have missed some, so if you get unusual behavior in a code port, this would be something to check.

My approach was to write high level routines which treat the four boards as one large board, one byte (8 LEDs) high and 128 columns long. Because I wanted to do some text output, I modified a 5x7 font to be 5x8 with partial descenders. There are some character manipulation functions using this built-in font for outputting text.

The high level routines are:

dm_init(#)	initialization
dm_blankall()	blank all boards
dm_blankcols()	blank columns
dm_fill()	write pattern to multiple columns
dm_rcol()	read single column
dm_wcol()	write single column
dm_rbytes()	read multiple columns
dm_wbytes()	write multiple columns
dm_lscroll()	scroll display left 1 column
dm_rscroll()	scroll display right 1 column
dm_rbit()	read single bit
dm_wbit()	write single bit
dm_blink()	set blink on/off
dm_enable()	set board enable on/off

dm_setpwm()......... set PWM duty cycle (brightness)

The call to the initialization routine (dm_init) sets the number of boards (1 – 4) that will be available. It also enables those boards, blanks them, and sets them to a PWM level of 7 (mid-level brightness).

There are low level routines which perform similar functions at the individual board level. For the most part, the high level routines decide which board or boards are affected, then call the lower level routines to perform the task(s).

The character functions consist of:

dm_cload() .... return character definition
dm_ccol() .... return column of char definition
dm_char().............. display single character
dm_str()................. display string

Each character of the font (ASCII 32 – 127) is defined on a 5x8 grid. These may be used as is for a monospaced font, or they may be “squeezed” for a proportional font. The last two functions listed allow character and string output to the display, while the first two allow for direct manipulation of the font definitions.

The mainline (test routine)

There is a mainline that tests most of the functions above. It initializes the four boards and draws a rectangle around the perimeter. It then draws and erases a zigzag pattern within the rectangle, then erases the display and places a monospaced message at various places.

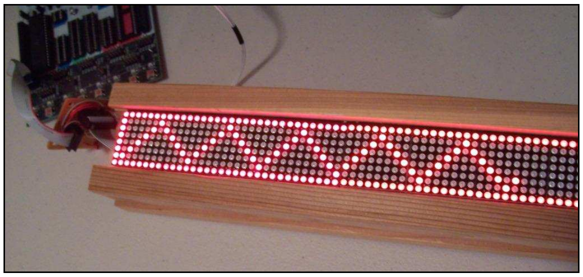

Photo 4: Dot addressing

It writes a blinking proportional message and scrolls it right and left, then draws a pattern across the full display. It reads the pattern, flips it upside down, then rewrites it. Then it toggles all the LEDs (so the display is mostly lit) and varies the brightness through its full range.

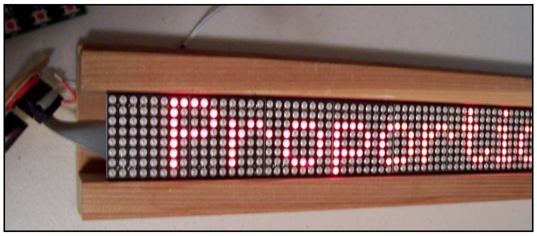

Photo 5: Proportional font

Finally it writes manually scrolled messages (using the character primitives) while changing the number of active boards from four down to one. The sequence then repeats.

Timing is performed via a simple one millisecond timer interrupt with a “kill time” function that waits for a certain number of milliseconds to pass. The code is written for an 8 MHz clock, so this may need to be adjusted for other configurations.

The small print

This code is presented for use without any guarantees, warrantees, or other promises to respect you in the morning. Use it at your own risk. You are free to incorporate it in other projects or situations as you see fit, although original author attribution is always appreciated and will bring you good karmic return. The demon dogs of hell will enthusiastically pursue you throughout eternity if you don’t.

I may be reached through the private message feature of the AVRfreaks.net website. My user name is “zbaird.”

The code

There are 5 files: mainline (test routine and timer functions), support functions and header, and character functions and header. In addition, the device specific header for the ATmega16 and some ICCAVR macros are referenced as header files.

LED_matrix2.c (mainline)

```
//      LED_matrix2 - test routine for the SURE LED matrix
//      multiple board version, horizontal layout
//
//     boards are: CS0 -> CS1 -> CS2 -> CS3, CS0 is master
//
//      the definitions in dm2.h define the wiring:
//
//      portA:
//        bit 0 - CS0 (master)
//        bit 1 - RD
//        bit 2 - WR
//        bit 3 - data
//        bit 4 - CS1
//        bit 5 - CS2
//        bit 6 - CS3
//        bit 7 - available

#include <iom16v.h>             // device (ATMega16) definitions
#include <AVRdef.h>             // ICCAVR macros
#include "dm2.h"                // header for display matrix routines
```

```
#include "dm_chars.h"           // header for display character routines

//      prototypes
char reverse(char);             // reverse bit order
void init(void);                // general initialization
void wait(void);                // burn time

//      demo routines
void outline(void);             // outline 4 boards
void zigzag(void);              // slo-mo zigzag
void monofont(void);            // monospaced font
void proportional(void);        // proportional font
void block_write(void);         // write array of bytes
void active(void);              // vary active boards

volatile unsigned int pause;    // delay counter (ms to kill)

// ------------------------------------------------------------
//      main() - mainline

void main(void) {

  init();                       // initialize timers, etc.
  pause = 100;                  // 0.1 seconds
  wait();                       // wait until it passes

  while (1) {
    dm_init(4);                 // initialize 4 boards
    dm_blankall();              // and blank them

    outline();                  // draw an outline
    zigzag();                   // slo-mo zigzag
    monofont();                 // monospaced font message
    proportional();             // proportional font message
    block_write();              // block write
    active();                   // vary active boards
  }
}

// ------------------------------------------------------------
//      outline() - draw an outline around 4 boards
//
//      using fill and column writes, draw an outline

void outline(void) {

  dm_fill(1, 126, 0x81);        // fill 126 columns
  dm_wcol(0, 0xff);             // write the left column
  dm_wcol(127, 0xff);           // and right
  pause = 1000;
  wait();
}

// ------------------------------------------------------------
//      zigzag() - draw a moving zigzag
//
//      draw and erase a zigzag line using the point
//      plotting routines

void zigzag(void) {
  char pass, y, add, i;

  for (pass = 1; pass < 3; pass++) {    // draw, then erase
    y = 1;
    add = 1;                            // going up or down
    for (i = 1; i < 127; i++) {         // cols 1-126
      pause = 20;
      wait();
      dm_wbit(i, y, pass & 0x01);       // set or clear
      y = y + add;
      if (y == 7) {                     // reach top?
```

```
        y = 5;                          // yup, start down
        add = 255;                      // -1
      } else if (y == 0) {              // reach bottom?
        y = 2;                          // yup, start up
        add = 1;                        // +1
      }
    }
  }
  pause = 1000;
  wait();
  dm_blankall();
}

// ------------------------------------------------------------
//      monofont() - monospaced font message
//
//      write a message at various places across the display

void monofont(void) {
  char i, *msg = "Mono font";

  for (i = 0; i < 128; i += 13) {      // some random spots
    dm_str(i, msg, 0x09);               // fixed font, pad=1
    pause = 600;
    wait();
    dm_blankall();                      // blank old
  }
  pause = 400;
  wait();
}

// ------------------------------------------------------------
//      proportional() - proportional font message
//
//      while blinking, write a message and scroll it
//      right and left

void proportional(void) {
  char i, *msg = "Proportional font";
  int cnt;

  cnt = dm_str(0, msg, 0x41);           // blank=4, pad=1
  pause = 600;
  wait();

  dm_blink(1);                          // start blinking
  cnt &= 0x7f;                          // drop error flags
  cnt = 127 - cnt;                      // number of places to shift

  for (i = 0; i < cnt; i++) {           // scoot it to the right
    dm_rscroll(0);
    pause = 60;
    wait();
  }

  for (i = 0; i < cnt; i++) {           // scoot it to the left
    dm_lscroll(0);
    pause = 60;
    wait();
  }

  pause = 400;
  wait();
  dm_blink(0);                          // turn off blinking
  dm_blankall();
}

// ------------------------------------------------------------
//      block_write() - write array of bytes

void block_write(void) {
  char add, w, w2, skip, blk[128], i;
  char use[] = { 0x18, 0x24, 0x42, 0x81 };  // < symbol
  char use2[] = { 0x81, 0x42, 0x24, 0x18 }; // > symbol
  char c1, msk, lng;

  add = 0;                              // space between symbols
  w = 0;                                // next column of <
  w2 = 3;                               // next column of >
  skip = 0;                             // blanks to insert

  for (i = 0; i < 64; i++) {            // half a display
    if (skip) {                         // between symbols?
      blk[i] = 0;                       // yes - blanks
      blk[127 - i] = 0;
      skip--;
    } else {                            // no - part of symbol
      blk[i] = use[w++];                // left side of display
      blk[127 - i] = use2[w2--];        // right side of display
    }
    if (w > 3) {                        // done with symbol?
      w = 0;                            // yes - start over
      w2 = 3;
      skip = add++;                     // # of blanks between them
    }
  }
  dm_wbytes(0, blk, 128);               // write full display
  pause = 2000;
  wait();

// we're going to progressively erase (fill) the middle of the display
  c1 = 50;                              // starting column
  msk = 1;                              // bottom LED on
  for (i = 0; i < 8; i++) {             // do 8 LEDs
    lng = (64 - c1) * 2;                // how wide the fill will be
    dm_fill(c1, lng, msk);              // fill it
    pause = 150;
    wait();
    msk += msk;                         // move LED up one row
    c1 -= 5;                            // and move starting point left
  }

  pause = 1000;
  wait();

  dm_rbytes(0, blk, 128);               // read full display
  for (i = 0; i < 128; i++)             // flip it vertically
    blk[i] = reverse(blk[i]);
  dm_wbytes(0, blk, 128);               // and rewrite it
  pause = 1000;
  wait();

  for (i = 0; i < 128; i++)             // now invert everything
    blk[i] ^= 0xff;                     // toggle all the bits
  dm_wbytes(0, blk, 128);               // and rewrite it
  pause = 800;
  wait();

  for (i = 8; i < 16; i++) {            // take intensity up full
    pause = 400;
    wait();
    dm_setpwm(i);
  }

  for (i = 14; i; i--) {                // and back down
    pause = 400;
    wait();
    dm_setpwm(i);
  }

  for (i = 0; i < 8; i++) {             // and back up half way
    pause = 400;
    wait();
    dm_setpwm(i);
  }
  pause = 1000;
  wait();
  dm_blankall();
}

// ------------------------------------------------------------
//      reverse(cc) - reverse the bit order for a byte
//
//      cc - incoming bit pattern
//      returns: inverted bit pattern

char reverse(char cc) {
  char i, msk, r;
  unsigned char msk2;

  msk = 0x01;
  msk2 = 0x80;
  r = 0;

  for (i = 0; i < 8; i++) {     // 8 bits in a byte
    if (msk & cc) r |= msk2;
    msk <<= 1;
    msk2 >>= 1;
  }
  return r;
}

// ------------------------------------------------------------
//      active() - vary the number of active boards
//
//      this shows how to put out characters manually
//      (as opposed to using dm_char() or dm_str())

void active(void) {
  char buf[7], *p, c, cnt, cc, pcnt, n;
  char msg[] = "  4 active boards";

  for (c = 0; c < 7; c++) buf[c] = 0;
  pcnt = 4;                     // number of active boards
  p = msg;                      // start at beginning of msg
  c = 0;                        // next column out
  cnt = 0;                      // message counter

  while (1) {                   // repeat count depends on message
    pause = 60;
    wait();                     // wait a while

    if (!c) {                   // are we done with prev char?
      cc = *p++;                // get next character out
      if (!cc) {                // are we done with message?
        p = msg;                // yes - start at beginning again
        cc = *p++;              // get first character
        cnt++;                  // count number of messages put out
        if (cnt > 1) {          // reach our preset limit?
          cnt = 0;              // yup - start over
          pcnt--;               // reduce board count
          if (!pcnt) break;     // if 1 --> 0, we're done
          dm_init(pcnt);        // initialize with reduced boards
          msg[2] = '0' + pcnt;  // insert board count into message
          if (pcnt == 1) msg[16] = 0; // for 1, drop plural
        }
      }
      n = dm_cload(cc, buf, 2);         // load up character definition
      if (cc == 32) n = 4; else n += 1; // blanks = 4, pad = 1
    }
    dm_lscroll(buf[c++]);    // put next column out

    if (c >= n) c = 0;          // last column out? start over
  }

  pause = 1000;
  wait();
  dm_init(4);                   // back to full complement of boards
  dm_blankall();
}

// ------------------------------------------------------------
//      wait() - burn time
//
//      this hangs until pause transitions from 1 to 0
//
//      to delay, set pause to the approximate number of milliseconds,
//        then call this
//      note: if you call this without setting pause, you die

void wait(void) {
  while (pause != 1) ;          // hang until pause = 1
  while (pause) ;               // hang until pause = 0
}

// ------------------------------------------------------------
//      timer0 interrupts - about 1 KHz

#pragma interrupt_handler timer0_comp_isr:iv_TIM0_COMP
void timer0_comp_isr(void)
{
  if (pause) pause--;
}

// ------------------------------------------------------------
//      init() - general initialization
void init(void) {
  CLI();
  UCSRB = 0x00;                 // disable while setting baud rate
  UCSRA = 0x00;
  UCSRC = BIT(URSEL) | 0x06;
  UBRRL = 0x19;                 // set baud rate lo
  UBRRH = 0x00;                 // set baud rate hi
  UCSRB = 0x18;

  TCCR0 = 0x00;                 // stop
  TCNT0 = 0xE1;                 // set count (about 1KHz)
  OCR0  = 0x1F;                 // set compare
  TCCR0 = 0x0C;                 // start timer

  MCUCR = 0x00;
  GICR  = 0x00;
  TIMSK = 0x02;                 // timer interrupt sources
  SEI();                        // re-enable interrupts
}
```

dm2.c (main functions)

```
//      dm2.c - source file for horizontal multi LED display matrix
//              (Sure Electronics 0832 boards)
//
//      these routines support up to 4 boards arranged horizontally.
//      CS0 is the leftmost and is the master.  CS1 is the second board,
//      and is the first slave, and so on.
//
//      there is a total of dm_nboards boards (1 - 4), set by the
//      argument in the call to dm_init().
//
//      each display memory is 4 bit x 64 for our use
//
//      display addresses:
//      ------------------
//         0  2  4  6  ...  62    <-- top half of display
//         1  3  5  7  ...  63    <-- bottom half of display
```

```
//
//        -^--             --^--
//        col1             col32
//
//
//      bit ordering:
//      -------------
//
//      4 bit r/w             8 bit r/w
//      ---------             ---------
//          3   top of display    7
//          2                     6
//          1                     5
//          0                     4
//          3                     3
//          2                     2
//          1                     1
//          0  bottom of display  0
//
//      columns on a single board number from 0 to 31.
//      columns on the group of boards number from 0 to dm_cmax,
//      where dm_cmax = dm_nboards * 32 - 1.
//
//      for ICCAVR all chars are unsigned. in the few places where
//      this is important, unsigned has been explicitly specified.
//
//      function summary
//      ----------------
//      dm_init(#)              initialization
//
//      dm_blankall()           blank all boards
//      dm_blankcols()          blank columns
//      dm_fill()               write pattern to multiple columns
//
//      dm_rcol()               read column
//      dm_wcol()               write column
//      dm_rbytes()             read multiple columns
//      dm_wbytes()             write multiple columns
//
//      dm_lscroll()            scroll display left 1 column
//      dm_rscroll()            scroll display right 1 column
//
//      dm_rbit()               read single bit
//      dm_wbit()               write single bit
//
//      dm_blink()              set blink on/off
//      dm_enable()             set board enable on/off
//      dm_setpwm()             set PWM duty cycle (brightness)
//
//      these functions act on all display boards (1 - 4, set by dm_init).
//      there are low level routines that perform these same functions on
//        the individual boards
//
//
#include <iom16v.h>
#include <AVRdef.h>
#include "dm2.h"

//      local prototypes
char dm_b_fill(char, char, char, char);    // board memory fill
char dm_r_bit(char, char, char);           // read one bit from board
void dm_w_bit(char, char, char, char);     // write one bit to board
void dm_r_scroll(char, char);              // right scroll board
char dm_l_scroll(char, char);              // left scroll board
void dm_multi(char, char);                 // general command to all boards
void dm_cmd(char, char);                   // general command to one board
char dm_r_col(char, char);                 // read column of board
void dm_w_col(char, char, char);           // write column of board
char dm_w_bytes(char, char, char, char *); // write bytes to board
char dm_r_bytes(char, char, char, char *); // read bytes from board
void dm_wakeup(char);                      // select board
```

```
char dm_read(char);                        // read 4 or 8 bits (low level)
void dm_write(char, unsigned int);         // send data out (low level)

//      the bit fiddling macros:
#define cs_high()   DM_PORT |= cs_mask
#define cs_low()    DM_PORT &= ~cs_mask
#define data_in()   DM_DDR &= ~DM_DATA; DM_PORT &= ~DM_DATA
#define data_out()  DM_DDR |= DM_DATA

//      local variables
char cs_mask;                      // mask for active board's CS line
char cs_m[] = {DM_CS0, DM_CS1, DM_CS2, DM_CS3};  // all the CS masks

//      global variables (for dm_chars.c functions - do not alter values)
char dm_nboards;                   // number of active boards (1 - 4)
unsigned char dm_cmax;             // rightmost column (31, 63, 95, or 127)


// =================================================================
// ========= the following are the public functions ================
// =================================================================

// ------------------------------------------------------------
//      dm_blink(mode) - set blink on/off
//
//      mode - bit 0 = 0 for off, = 1 for on

void dm_blink(char m) {
  char cmd;

  cmd = (m & 1) ? BLINK_ON : BLINK_OFF;
  dm_multi(1, cmd);
}

// ------------------------------------------------------------
//      dm_enable(mode) - set enable on/off
//
//      mode - bit 0 = 0 for off, = 1 for on

void dm_enable(char m) {
  char cmd;

  cmd = (m & 1) ? SYS_EN : SYS_DIS;
  dm_multi(1, cmd);
}

// ------------------------------------------------------------
//      dm_setpwm(level) - set PWM duty cycle
//
//      level - 0 to 15 (rightmost 4 bits)

void dm_setpwm(char m) {

  m &= 0x0f;                    // keep it legal (0 - 15)
  dm_multi(1, SET_PWM | m);
}

// ------------------------------------------------------------
//      dm_rcol(column) - read a column of display
//
//      column - column to read (0 - dm_cmax)

char dm_rcol(char col) {
  char who;

  if (col > dm_cmax) return 0;  // ignore if out of range
  who = (col >> 5) & 0x03;      // which board it's on
  col &= 0x1f;                  // local column (0 - 31)
  return dm_r_col(who, col);    // go read it
}
```

```
// ------------------------------------------------------------
//      dm_wcol(column, value) - write a column to display
//
//      col - column to write (0 - dm_cmax)
//      value - value to write

void dm_wcol(char col, char v) {
  char who;

  if (col <= dm_cmax) {         // only want legal columns
    who = (col >> 5) & 0x03;    // which board it's on
    col &= 0x1f;                // local column (0 - 31)
    dm_w_col(who, col, v);      // go write it
  }
}

// ------------------------------------------------------------
//      dm_rbytes(col, array, n) - read bytes from display
//
//      col - display column to start in (0 - dm_cmax)
//      array[] - where to store bytes
//      n - number of bytes to read (1 - 128)
//
//      returns: number of bytes read

int dm_rbytes(char col, char *p, int n) {
  char who, mx, rtn;
  int r;

  if (col > dm_cmax || n == 0) return 0;
  r = (int) dm_cmax + 1 - col;          // maximum from this column
  if (n > r) n = r;                     // keep it in bounds

  r = 0;                                // count number read
  who = col >> 5;                       // initial board (0 - 3)
  col &= 0x1f;                          // initial column on board

  while (n > 0 && who < dm_nboards) {   // second test is paranoia
    if (n > 32) mx = 32; else mx = n;
    rtn = dm_r_bytes(who, col, mx, p);  // write some
    n -= rtn;                           // that many fewer to go
    r += rtn;                           // that many more read
    p += rtn;                           // advance pointer
    who++;                              // move to next board
    col = 0;                            // starting at its beginning
  }
  return r;
}

// ------------------------------------------------------------
//      dm_wbytes(col, array, n) - write bytes to display
//
//      col - display column to start in (0 - dm_cmax)
//      array[] - bytes to write
//      n - number of bytes to write (1 - 128)
//
//      returns: number of bytes written

int dm_wbytes(char col, char *p, int n) {
  char who, mx, rtn;
  int r;

  if (col > dm_cmax || n == 0) return 0;
  r = (int) dm_cmax + 1 - col;          // maximum from this column
  if (n > r) n = r;                     // keep it in bounds

  r = 0;                                // count number written
  who = col >> 5;                       // initial board (0 - 3)
  col &= 0x1f;                          // initial column on board

  while (n > 0 && who < dm_nboards) {   // second test is paranoia
```

```
    if (n > 32) mx = 32; else mx = n;
    rtn = dm_w_bytes(who, col, mx, p);  // write some
    n -= rtn;                           // that many fewer to go
    r += rtn;                           // that many more written
    p += rtn;                           // advance pointer
    who++;                              // move to next board
    col = 0;                            // starting at its beginning
  }
  return r;
}

// ------------------------------------------------------------
//      dm_lscroll(val) - scroll all displays left 1 column
//
//      moves column 1 --> 0, 2 --> 1, ..., dm_cmax --> dm_cmax - 1
//      and inserts val into dm_cmax (rightmost column of all boards)

void dm_lscroll(char v) {
  unsigned char who;

  who = (dm_nboards - 1) & 0x03;  // rightmost board number
  while (who < 4) {
    v = dm_l_scroll(who, v);      // scroll single board
    who--;                        // this will wrap to 255
  }
}

// ------------------------------------------------------------
//      dm_rscroll(val) - scroll all displays right 1 column
//
//      moves column dm_cmax - 1 --> dm_cmax, ..., 0 --> 1
//      and inserts val in column 0

void dm_rscroll(char v) {
  char i, c31;

  for (i = 0; i < dm_nboards; i++) {
    c31 = dm_r_col(i, 31);       // read old column 31
    dm_r_scroll(i, v);
    v = c31;
  }
}

// ------------------------------------------------------------
//      dm_rbit(who, column, bit) - read single bit of display
//
//      column - 0 - dm_cmax
//      bit - 0 to 7 (7 at the top)
//      returns: 0 or 1

char dm_rbit(char col, char bnum) {
  char who;

  if (col > dm_cmax) return 0;        // ignore if out of range
  who = (col >> 5) & 0x03;            // which board it's on
  col &= 0x1f;                        // local column (0 - 31)
  return dm_r_bit(who, col, bnum);    // go read it
}

// ------------------------------------------------------------
//      dm_wbit(column, bit, value) - write single bit to display
//
//      column: 0 - dm_cmax
//      bit:    0 - 7
//      value:  0 or 1 (bit 0 used)

void dm_wbit(char col, char bnum, char v) {
  char who;

  if (col <= dm_cmax) {               // only want legal columns
    who = (col >> 5) & 0x03;          // which board it's on
    col &= 0x1f;                      // local column (0 - 31)
    dm_w_bit(who, col, bnum, v);      // go write it
  }
}

// ------------------------------------------------------------
//      dm_blankall() - blank all boards

void dm_blankall(void) {
  char i;

  for (i = 0; i < dm_nboards; i++)
    dm_b_fill(i, 0, 32, 0);           // blank board's memory
}

// ------------------------------------------------------------
//      dm_blankcols(col, n) - blank some columns
//
//      col - starting column number (0 - dm_cmax)
//        (dm_cmax will be 31, 63, 95, or 127)
//      n - number of columns to blank (1 - 128)
//
//      returns: number of columns blanked

int dm_blankcols(char col, int n) {
  return dm_fill(col, n, 0);
}

// ------------------------------------------------------------
//      dm_fill(col, n, fill) - write pattern to multiple columns
//
//      col - starting column number (0 - dm_cmax)
//        (dm_cmax will be 31, 63, 95, or 127)
//      n - number of columns to write (1 - 128)
//      fill - character to write
//
//      returns: number of columns written

int dm_fill(char col, int n, char f) {
  char who, mx, rtn;
  int r;

  if (col > dm_cmax || n == 0) return 0;
  r = (int) dm_cmax + 1 - col;          // maximum from this column
  if (n > r) n = r;                     // keep it in bounds

  r = 0;
  who = col >> 5;                       // initial board (0 - 3)
  col &= 0x1f;                          // initial column on board

  while (n > 0 && who < dm_nboards) {   // second test is paranoia
    if (n > 32) mx = 32; else mx = n;
    rtn = dm_b_fill(who, col, mx, f);   // fill 'er up
    n -= rtn;                           // that many fewer to go
    r += rtn;                           // that many more written
    who++;                              // move to next board
    col = 0;                            // starting at its beginning
  }
  return r;
}

// ------------------------------------------------------------
//      dm_init(nboards) - initialization
//
//      nboards - number of boards (1 - 4)
//
//      notice that all the I/O lines for the displays have
//      to be on the same port

void dm_init(char n) {
  char p, msk, i;
  if (n < 1 || n > 4) n = 1;
  dm_nboards = n;
  dm_cmax = (n << 5) - 1;               // max column (31, 63, 95, or 127)
  p = DM_RD | DM_WR;                    // the two fixed outputs
  for (i = 0; i < dm_nboards; i++) p |= cs_m[i];  // each board's CS bit

  msk = ~(p | DM_DATA);                 // the "not ours" bits
  DM_DDR &= msk;                        // drop our bits, keep theirs
  DM_PORT &= msk;                       // no pullups on ours
  DM_PORT |= p;                         // turn on pullups briefly
  DM_DDR |= p;                          // and switch to high outputs

  // at this point, RD, WR and the CS lines are high (inactive)
  // outputs, and DATA is an input without a pullup
  //
  // the cascade initialization sequence is:
  //   disable master
  //   disable slaves
  //   commons options (master and slave) - default is OK
  //   set master to master
  //   set slaves to slave
  //   sys on for master
  //   sys on for slaves
  //   blank
  //   LEDs on

  dm_multi(1, SYS_DIS);                 // disable master and slaves
  dm_multi(1, COMMONS);                 // all to PMOS 8 (the default)
  dm_cmd(0, SET_MASTER);                // tell the first one it's master
  dm_multi(0, SET_SLAVE);               // tell the others they're slaves
  dm_multi(1, SYS_EN);                  // turn all the buggers on
  dm_blankall();                        // blank everyone's memory
  dm_multi(1, SET_PWM | 7);             // set mid brightness
  dm_multi(1, LED_ON);                  // and away we go
}

// =================================================================
// ========= the following are local low level functions ===========
// =================================================================

// ------------------------------------------------------------
//      dm_b_fill(who, col, n, fill) - fill board's memory
//
//      who  - board to use (0 - 3)
//      col  - starting address (0 - 31)
//      n    - number of columns
//      fill - fill character
//
//      returns: number of bytes written

char dm_b_fill(char who, char col, char n, char fill) {
  unsigned int addr;
  char i;

  if (col > 31 || n == 0) return 0;
  if (n + col > 32) n = 32 - col;

  addr = col << 1;              // they use 4 bit addresses
  dm_wakeup(who);               // make board active
  dm_write(3, 0x05);            // send 101
  dm_write(7, addr);            // nibble address
  for (i = 0; i < n; i++)
    dm_write(8, (unsigned int) fill);   // write 8 bits
  cs_high();                    // end of command
  return n;
}

// ------------------------------------------------------------
//      dm_r_bit(who, column, bit) - read single bit from board
//
```

```
//      who - board to use (0 - 3)
//      column - 0 - 31
//      bit - 0 to 7 (7 at the top)
//
//      returns: 0 or 1

char dm_r_bit(char who, char col, char bnum) {
  unsigned int addr;
  char msk, ret;

  addr = col << 1;              // use 4 bit addresses
  if (bnum > 3) {
    msk = 1 << (bnum - 4);      // even address (top)
  } else {
    msk = 1 << bnum;            // odd address (bottom)
    addr++;
  }

  dm_wakeup(who);               // set board active
  dm_write(3, 0x06);            // send 110
  dm_write(7, addr);            // nibble address
  ret = dm_read(0);             // read 4 bits
  cs_high();                    // end of command

  if (ret & msk) ret = 1; else ret = 0;
  return ret;
}

// ------------------------------------------------------------
//      dm_w_bit(who, column, bit, value) - write single bit to board
//
//      who:    board to use (0 - 3)
//      column: 0 - 31
//      bit:    0 - 7
//      value:  0 or 1 (bit 0 used)

void dm_w_bit(char who, char col, char bnum, char v) {
  unsigned int addr, ret, a_msk, o_msk;

  addr = col << 1;              // use 4 bit addresses
  if (bnum > 3) {
    o_msk = 1 << (bnum - 4);    // even address (top)
  } else {
    o_msk = 1 << bnum;          // odd address (bottom)
    addr++;
  }

  a_msk = (~o_msk) & 0x0f;      // the AND mask, in 4 bits
  dm_wakeup(who);               // make board active
  dm_write(3, 0x05);            // send 101
  dm_write(7, addr);            // nibble address
  ret = dm_read(0);             // read 4 bits
  ret &= a_msk;                 // kill our bit
  if (v) ret |= o_msk;          // set our bit, if desired.
  dm_write(4, ret);             // and rewrite 4 bits
  cs_high();                    // end of command
}

// ------------------------------------------------------------
//      dm_r_scroll(who, val) - scroll board right 1 column
//
//      who - board to use (0 - 3)
//      moves column 30 --> 31, 29 --> 30, ..., 0 --> 1
//      and inserts val in column 0

void dm_r_scroll(char who, char v) {
  char h[2], i, t;
  unsigned int r;

  h[0] = v >> 4;                // top 4 bits, new column 0
  h[1] = v & 0x0f;              // bottom 4 bits, new col 0
  dm_wakeup(who);               // make board active
  dm_write(3, 0x5);             // read/mod/write
  dm_write(7, 0);               // address = 0x00

  for (i = 0; i < 64; i++) {
    t = i & 0x01;               // 0 for top, 1 for bottom
    r = h[t];                   // previous value
    h[t] = dm_read(0);          // read 4 bits
    dm_write(4, r);             // write 4 bits
  }
  cs_high();                    // that's all, folks
}

// ------------------------------------------------------------
//      dm_l_scroll(who, val) - scroll board left 1 column
//
//      who - board to use (0 - 3)
//      moves column 1 --> 0, 2 --> 1, ..., 31 --> 30,
//      and inserts val in column 31
//
//      returns: original value in column 0

char dm_l_scroll(char who, char v) {
  char hold[33];

  dm_r_bytes(who, 0, 32, hold);      // read current contents
  hold[32] = v;                      // new value
  dm_w_bytes(who, 0, 32, hold + 1);  // rewrite, offset
  return hold[0];                    // give them column 1
}

// ------------------------------------------------------------
//      dm_multi(who, cmd) - send command to multiple display boards
//
//      who - 0, slaves only; 1, master and slaves
//      cmd - command

void dm_multi(char w, char cmd) {
  char i, bot;

  if (w) {
    bot = 0;                    // include master
  } else {
    if (dm_nboards < 2) return; // no slaves to push around
    bot = 1;                    // exclude master
  }
  for (i = bot; i < dm_nboards; i++) dm_cmd(i, cmd);
}

// ------------------------------------------------------------
//      dm_cmd(who, command) - send command to single board
//
//      who - board to use (0 - 3)
//      command is one of:
//        LED_ON           display LEDs on
//        LED_OFF          display LEDs off
//        SYS_DIS          system disable
//        SYS_EN           system enable
//        BLINK_ON         blink on
//        BLINK_OFF        blink off
//        SET_PWM          set PWM level (see note)
//        SET_MASTER       set board as master
//        SET_SLAVE        set board as slave
//        COMMONS          PMOS 8
//
//      for SET_PWM, OR in a value of 0 - 15 to set PWM level

void dm_cmd(char who, char cmd) {
  unsigned int v;

  dm_wakeup(who);                // which one we're talking to
  v = ((unsigned int) cmd << 1); // commands are 9 bits, bit 0 = 0
  dm_write(3, 0x04);             // send binary 100 as ID
  dm_write(9, v);                // send the 9 bit command
  cs_high();                     // shut it down
}

// ------------------------------------------------------------
//      dm_r_col(who, column) - read column of board
//
//      who - board to use (0 - 3)
//      column - 0 to 31
//      returns the value of the column, bit 0 at top

char dm_r_col(char who, char col) {
  unsigned int addr;
  char ret;

  addr = col << 1;              // they're 4 bit addresses
  dm_wakeup(who);               // make board active
  dm_write(3, 0x06);            // send 110
  dm_write(7, addr);            // nibble address
  ret = dm_read(1);             // read 8 bits
  cs_high();                    // end of command

  return ret;
}

// ------------------------------------------------------------
//      dm_w_col(who, column, value) - write a column of board
//
//      who - board to use (0 - 3)
//      column - 0 to 31
//      value - 8 bits, bit 0 at top

void dm_w_col(char who, char col, char v) {
  unsigned int addr;

  addr = col << 1;                // they're 4 bit addresses
  dm_wakeup(who);                 // make board active
  dm_write(3, 0x05);              // send 101
  dm_write(7, addr);              // nibble address
  dm_write(8, (unsigned int) v);  // write 8 bits
  cs_high();                      // end of command
}

// ------------------------------------------------------------
//      dm_w_bytes(who, col, n, array) - write bytes to a board
//
//      who - board to use (0 - 3)
//      col - display column to start in (0 - 31)
//      n - number of bytes to write (1 - 32)
//      array[] - bytes to write
//
//      returns: number of bytes written

char dm_w_bytes(char who, char col, char n, char *p) {
  unsigned int addr;
  char i;

  if (col > 31 || n == 0) return 0;
  if (n + col > 32) n = 32 - col;

  addr = col << 1;              // they use 4 bit addresses
  dm_wakeup(who);               // make board active
  dm_write(3, 0x05);            // send 101
  dm_write(7, addr);            // nibble address
  for (i = 0; i < n; i++)
    dm_write(8, (unsigned int) *p++);   // write 8 bits
  cs_high();                    // end of command
  return n;
}
```

```
// ------------------------------------------------------------
//      dm_r_bytes(who, col, n, array) - read board's memory
//
//      who - board to use (0 - 3)
//      col - starting column (0 - 31)
//      n - number of columns (bytes) to read (1 - 32)
//      array[] - where to put data
//
//      returns: number of bytes read

char dm_r_bytes(char who, char col, char n, char *p) {
  unsigned int addr;
  char i;

  if (col > 31 || n == 0) return 0;
  if (n + col > 32) n = 32 - col;

  addr = col << 1;              // they use 4 bit addresses
  dm_wakeup(who);               // make board active
  dm_write(3, 0x06);            // send 110
  dm_write(7, addr);            // nibble address
  for (i = 0; i < n; i++)
    *p++ = dm_read(1);          // 32 8 bit reads
  cs_high();                    // end of command
  return n;
}

// ------------------------------------------------------------
//      dm_wakeup(who) - set active display board
//
//      who - board to use (0 - 3)

void dm_wakeup(char who) {
  cs_mask = cs_m[who & 0x03];   // mask for our CS bit
  cs_low();                     // take CS low to activate
}

// ------------------------------------------------------------
//      dm_read(w) - read 4 or 8 bits
//
//      w - 0 for 4 bits, 8 otherwise
//
//      note: we're expecting data to (always) be input
//      we read high bit first

char dm_read(char n) {
  char inp, i;

  if(n) n = 8; else n = 4;
  inp = 0;                              // build value here
  for (i = 0; i < n; i++) {
    DM_PORT &= ~DM_RD;                  // bring RD low
    DM_PORT |= DM_RD;                   // bring RD high
    inp <<= 1;                          // make room for our bit
    if (DM_PIN & DM_DATA) inp |= 1;     // if high, OR it in
  }
  return inp;
}

// ------------------------------------------------------------
//      dm_write(bits, value) - send some bits
//
//      bits - number of bits
//      value - value to send
//
//      we always leave the data line as input
//      high order bit goes out first
//
//      caller needs to call dm_wakeup() to start command, and
//      to call cs_high() to end command.
```

```
void dm_write(char nbits, unsigned int v) {
  unsigned int msk;

  msk = 1 << (nbits - 1);
  data_out();                           // we're writing

  while (msk) {
    DM_PORT &= ~DM_WR;                  // bring WR low
    if (msk & v) {
      DM_PORT |= DM_DATA;               // output a 1
    } else {
      DM_PORT &= ~DM_DATA;              // output a 0
    }
    DM_PORT |= DM_WR;                   // bring WR high
    msk >>= 1;                          // next bit
  }
  data_in();                            // leave it as input
}
```

dm2.h (header for main functions)

```
//      dm2.h - header file for horizontal multiboard LED display matrix


extern void dm_blink(char);                     // set blink on/off
extern void dm_enable(char);                    // set enable on/off
extern void dm_setpwm(char);                    // set PWM duty cycle

extern char dm_rcol(char);                      // read column of entire display
extern void dm_wcol(char, char);                // write column of entire display
extern int dm_rbytes(char, char *, int);        // read bytes from display memory
extern int dm_wbytes(char, char *, int);        // write bytes to display memory

extern void dm_lscroll(char);                   // scroll display left 1 column
extern void dm_rscroll(char);                   // scroll display right 1 column

extern char dm_rbit(char, char);                // read one bit from display
extern void dm_wbit(char, char, char);          // write one bit to display

extern void dm_blankall(void);                  // blank all boards
extern int dm_blankcols(char, int);             // blank columns
extern int dm_fill(char, int, char);            // write pattern to multiple 
columns

extern void dm_init(char);                      // initialization

//      global variables
extern char dm_nboards;                         // number of active boards (1 - 4)
extern unsigned char dm_cmax;                   // rightmost column (31, 63, 95, 
or 127)


#define COMMONS 0x28            // PMOS open, 8
#define SET_MASTER 0x14         // master mode
#define SET_SLAVE 0x10          // slave mode
#define LED_ON 0x03             // display LEDs on
#define LED_OFF 0x02            // display LEDs off
#define SYS_DIS 0x00            // system disable
#define SYS_EN 0x01             // system enable
#define BLINK_ON 0x09           // blink on
#define BLINK_OFF 0x08          // blink off
#define SET_PWM 0xa0            // set PWM level (OR 0000 - 1111)

#define DM_PORT PORTA           // port to use
#define DM_DDR  DDRA
#define DM_PIN  PINA
#define DM_CS0  (1 << 0)        // CS for master
#define DM_CS1  (1 << 4)        // CS for first slave
#define DM_CS2  (1 << 5)        // CS for second slave
#define DM_CS3  (1 << 6)        // CS for third slave
```

```
#define DM_RD   (1 << 1)        // read
#define DM_WR   (1 << 2)        // write
#define DM_DATA (1 << 3)        // data
```

dm_chars.c (character routines)

```
//      dm_chars.c - support functions for 5x8 characters
//
//      dm_str()   - write a string to the display
//      dm_char()  - write a character to the display
//      dm_ccol()  - write a column of mono font
//      dm_cload() - load mono or proportional font char

#include <iom16v.h>
#include <AVRdef.h>
#include "dm2.h"
#include "dm_chars.h"


//      these characters are defined in 5 columns on an 8
//      row cell.  the bottom line (bit 0) is off except
//      for mini descenders.

__flash char font[480] = {
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFA,0x00,0x00,   //  !
   0x00,0xC0,0x00,0xC0,0x00,0x28,0xFE,0x28,0xFE,0x28,   // "#
   0x24,0x54,0xFE,0x54,0x48,0xC4,0xC8,0x10,0x26,0x46,   // $%
   0x6C,0x92,0xAA,0x44,0x0A,0x00,0xA0,0xC0,0x00,0x00,   // &'
   0x00,0x38,0x44,0x82,0x00,0x00,0x82,0x44,0x38,0x00,   // ()
   0x28,0x10,0x7C,0x10,0x28,0x10,0x10,0x7C,0x10,0x10,   // *+
   0x00,0x05,0x06,0x00,0x00,0x10,0x10,0x10,0x10,0x10,   // ,-
   0x00,0x06,0x06,0x00,0x00,0x04,0x08,0x10,0x20,0x40,   // ./
   0x7C,0x82,0x82,0x82,0x7C,0x00,0x42,0xFE,0x02,0x00,   // 01
   0x42,0x86,0x8A,0x92,0x62,0x84,0x82,0xA2,0xD2,0x8C,   // 23
   0x18,0x28,0x48,0xFE,0x08,0xE4,0xA2,0xA2,0xA2,0x9C,   // 45
   0x3C,0x52,0x92,0x92,0x0C,0x80,0x8E,0x90,0xA0,0xC0,   // 67
   0x6C,0x92,0x92,0x92,0x6C,0x60,0x92,0x92,0x94,0x78,   // 89
   0x00,0x6C,0x6C,0x00,0x00,0x00,0x65,0x66,0x00,0x00,   // :;
   0x10,0x28,0x44,0x82,0x00,0x28,0x28,0x28,0x28,0x28,   // <=
   0x00,0x82,0x44,0x28,0x10,0x40,0x80,0x8A,0x90,0x60,   // >?
   0x4C,0x92,0x9E,0x82,0x7C,0x7E,0x88,0x88,0x88,0x7E,   // @A
   0xFE,0x92,0x92,0x92,0x6C,0x7C,0x82,0x82,0x82,0x44,   // BC
   0xFE,0x82,0x82,0x44,0x38,0xFE,0x92,0x92,0x92,0x82,   // DE
   0xFE,0x90,0x90,0x90,0x80,0x7C,0x82,0x92,0x92,0x5E,   // FG
   0xFE,0x10,0x10,0x10,0xFE,0x00,0x82,0xFE,0x82,0x00,   // HI
   0x04,0x02,0x82,0xFC,0x80,0xFE,0x10,0x28,0x44,0x82,   // JK
   0xFE,0x02,0x02,0x02,0x02,0xFE,0x40,0x30,0x40,0xFE,   // LM
   0xFE,0x20,0x10,0x08,0xFE,0x7C,0x82,0x82,0x82,0x7C,   // NO
   0xFE,0x90,0x90,0x90,0x60,0x7C,0x82,0x8A,0x84,0x7A,   // PQ
   0xFE,0x90,0x98,0x94,0x62,0x62,0x92,0x92,0x92,0x8C,   // RS
   0x80,0x80,0xFE,0x80,0x80,0xFC,0x02,0x02,0x02,0xFC,   // TU
   0xF8,0x04,0x02,0x04,0xF8,0xFC,0x02,0x1C,0x02,0xFC,   // VW
   0xC6,0x28,0x10,0x28,0xC6,0xE0,0x10,0x0E,0x10,0xE0,   // XY
   0x86,0x8A,0x92,0xA2,0xC2,0x00,0xFE,0x82,0x82,0x00,   // Z[
   0x40,0x20,0x10,0x08,0x04,0x00,0x82,0x82,0xFE,0x00,   // \]
   0x20,0x40,0x80,0x40,0x20,0x01,0x01,0x01,0x01,0x01,   // ^_
   0x00,0x80,0x40,0x20,0x00,0x04,0x2A,0x2A,0x2A,0x1E,   // `a
   0xFE,0x12,0x22,0x22,0x1C,0x1C,0x22,0x22,0x22,0x04,   // bc
   0x1C,0x22,0x22,0x12,0xFE,0x1C,0x2A,0x2A,0x2A,0x18,   // de
   0x10,0x7E,0x90,0x80,0x40,0x18,0x25,0x25,0x25,0x3E,   // fg
   0xFE,0x10,0x20,0x20,0x1E,0x00,0x22,0xBE,0x02,0x00,   // hi
   0x02,0x01,0x21,0xBE,0x00,0xFE,0x08,0x14,0x22,0x00,   // jk
   0x00,0x82,0xFE,0x02,0x00,0x3E,0x20,0x18,0x20,0x1E,   // lm
   0x3E,0x10,0x20,0x20,0x1E,0x1C,0x22,0x22,0x22,0x1C,   // no
   0x3F,0x24,0x24,0x24,0x18,0x18,0x24,0x24,0x34,0x1F,   // pq
   0x3E,0x10,0x20,0x20,0x10,0x12,0x2A,0x2A,0x2A,0x04,   // rs
   0x20,0xFC,0x22,0x02,0x04,0x3C,0x02,0x02,0x04,0x3E,   // tu
   0x38,0x04,0x02,0x04,0x38,0x3C,0x02,0x0C,0x02,0x3C,   // vw
   0x22,0x14,0x08,0x14,0x22,0x38,0x05,0x05,0x05,0x3E,   // xy
   0x22,0x26,0x2A,0x32,0x22,0x00,0x10,0x6C,0x82,0x00,   // z{
   0x00,0x00,0xFE,0x00,0x00,0x00,0x82,0x6C,0x10,0x00,   // |}
   0x08,0x10,0x10,0x08,0x10,0x00,0x00,0x40,0xA0,0x40};   // ^degree


// ------------------------------------------------------------
//      dm_cload(cc, *buf, adj) - return a character definition
//
//      cc - character (dec 32 to 127)
//      buf[] - place to return (5 bytes)
//      adj - adjustment:
//              0 = none (returns 5)
//              1 = left, minimum (returns # of bytes in char)
//                  (leaves trailing bytes as is)
//              2 = left, zero (blank) filled to 5
//                  (returns # of bytes in char)
//
//      returns: 5 or number of nonzero bytes (width of character)
//
//      while each character is defined on a 5x8 grid, the actual
//      character may take less than 5 columns.  if adj is nonzero,
//      the return is for the number of nonblank bytes returned.
//      if adj = 1, then only that number of bytes is returned.
//      if adj = 2, then 5 bytes are returned, with zeroes following
//      the nonzero bytes that make up the character.

char dm_cload(char cc, char *p, char adj) {
  int offset;
  char i, n, v, skip;

  if (cc < 32 || cc > 127) cc = 32;
  offset = (int) (cc - 32) * 5;   // blank is first in table

  if (!adj) {
    for (i = 0; i < 5; i++) *p++ = font[offset++];
    return 5;
  }

  if (adj == 2) {
    for (i = 0; i < 5; i++) *(p + i) = 0;
  }

  n = skip = 0;
  for (i = 0; i < 5; i++) {     // 5 columns per character
    v = font[offset++];         // a column of definition
    if (n) {                    // have we seen anything yet?
      if (v) {                  // yes - is this anything?
        while (skip) {          // yes - fill in any zeroes
          *p++ = 0;
          n++;                  // count them
          skip--;
        }
        *p++ = v;               // and put away this value
        n++;                    // count it
      } else {                  // no - remember we skipped it
        skip++;
      }
    } else {                    // haven't seen anything yet
      if (v) {                  // is this anything?
        *p++ = v;               // our first nonzero - save it
        n = 1;                  // and count it
      }
    }
  }
  return n;
}

// ------------------------------------------------------------
//      dm_ccol(col, cc) - return single column of a whole character
//
//      col - column (0 - 4)
//      cc - ascii character
//
```

```
//      this returns one column of the full 5x8 character definition

char dm_ccol(char col, char cc) {
  int offset;

  if (cc < 32 || cc > 127) cc = 32;
  offset = (int) (cc - 32) * 5 + col;   // blank is first in table
  return font[offset];
}

// ------------------------------------------------------------
//      dm_char(col, cc, fmt) - put out a char to board(s)
//
//      col - left hand column of where to start character:
//         1 board,  0 - 31
//         2 boards, 0 - 63
//         3 boards, 0 - 95
//         4 boards, 0 - 127
//      cc - ascii character (32 - 127)
//      fmt - format (OR in the following):
//         0-7 - right padding
//         8   - raw character (5 columns wide)
//               otherwise, the actual character width is used
//               (proportional font)
//         #   - width of blank * 16:
//               0x00 - 0 columns wide  (CC_BL_0)
//               0x10 - 1 column wide   (CC_BL_1)
//                ...   ...
//               0x70 - 7 columns wide  (CC_BL_7)
//               note: blanks are 5 columns wide in mono font,
//                 so this applies only to proportional
//         example: to have 2 columns between characters, with
//           blanks taking 4 columns, fmt = 0x42 = 66
//
//      returns: number of columns output (0 - 12), and
//          bit CC_TRUNC will be set if something was truncated
//          bit CC_TCHAR will be set if the truncation occurred within
//              the character definition itself (not in the padding)
//
//      zero will be returned if the character is outside of the
//      range (32 - 127), the column is out of range, or for blanks
//      with no padding if a blank width has not been specified.
//
//      this will truncate the character at rightmost column

char dm_char(char col, char cc, char fmt) {
  char wbl, rf, pad, buf[5], w;
  char cnt;

  if (col > dm_cmax) return 0;          // column too big?
  if (cc < 32 || cc > 127) return 0;    // character out of range

  cnt = 0;                              // number of columns written
  pad = fmt & 0x07;                     // number of padding columns (0 - 7)
  rf = (fmt & 0x08) ? 0 : 1;            // =0 for mono, =1 for proportional
  wbl = (fmt >> 4) & 0x07;              // width of blank (0 - 7)

  w = dm_cload(cc, buf, rf);            // load pattern into buf[]
  if (!w) {                             // w = 0 means proportional blank
    if (wbl) {                          // are we putting out blanks?
      cnt = dm_fill(col, (int) wbl, 0); // put out some blank columns
      if (cnt != wbl) return (cnt | CC_TRUNC | CC_TCHAR);  // truncated
    }
  } else {                              // normal character
    cnt = dm_wbytes(col, buf, (int) w); // output the columns
    if (cnt != w) return (cnt | CC_TRUNC | CC_TCHAR);  // truncated
  }

  if (pad) {                            // we got the character out OK
    col += cnt;                         // skip over what just went out
    w = dm_fill(col, (int) pad, 0);     // put out some blank columns
    cnt += w;                           // how many columns we put out
    if (w != pad) cnt |= CC_TRUNC;      // did we truncate the padding?
  }
  return cnt;
}

// ------------------------------------------------------------
//      dm_str(col, *msg, fmt) - write a string to the display
//
//      col - starting column number (0 - 31, 63, 95, or 127)
//      msg - pointer to string
//      fmt - format (OR in the following):
//         0-7 - right padding
//         8   - mono (raw characters, 5 columns wide)
//               otherwise, the actual character width is used
//               (proportional font)
//         #   - width of blank * 16:
//               0x00 - 0 columns wide  (CC_BL_0)
//               0x10 - 1 column wide   (CC_BL_1)
//                ...   ...
//               0x70 - 7 columns wide  (CC_BL_7)
//         example: to have 2 columns between characters, with
//           blanks taking 4 columns, fmt = 0x42 = 66
//
//      returns: number of columns output (includes padding on
//        final character if no trucation), plus optional:
//        CC_ST_ERR - truncation error (some sort of failure)
//        CC_ST_OK  - truncation occurred on padding of final char

int dm_str(char col, char *p, char fmt) {
  char c, r, ot;
  int rtn;

  rtn = 0;
  while (c = *p++) {                    // get next char
    r = dm_char(col, c, fmt);           // try to write it
    ot = r & 0x0f;                      // columns we got out
    rtn += ot;                          // total columns written
    col += ot;                          // advance column number
    if (r & CC_TRUNC) {                 // did we truncate?
      if (!(r & CC_TCHAR) && !(*p)) rtn |= CC_ST_OK; // yes, in final pad
      return rtn | CC_ST_ERR;
    }
  }
  return rtn;
}
```

dm_chars.h (header for character routines)

```
//      dm_chars.h - header file for 5x8 characters

extern char dm_cload(char, char *, char);       // return character definition
extern char dm_ccol(char, char);                // return column of char
definition
extern char dm_char(char, char, char);          // display single character
extern int dm_str(char, char *, char);          // display string

#define CC_TRUNC  0x10                          // truncated character or padding
#define CC_TCHAR  0x20                          // trucated the character
#define CC_BL_0   0x00                          // blank width = 0
#define CC_BL_1   0x10                          // blank width = 1
#define CC_BL_2   0x20                          // blank width = 2
#define CC_BL_3   0x30                          // blank width = 3
#define CC_BL_4   0x40                          // blank width = 4
#define CC_BL_5   0x50                          // blank width = 5
#define CC_BL_6   0x60                          // blank width = 6
#define CC_BL_7   0x70                          // blank width = 7
#define CC_ST_ERR 0x0100                        // truncated message somehow
#define CC_ST_OK  0x0200                        // truncation only in final
padding
```