Mattel™ Hot Wheels™ Radar Gun

Chuck Baird, http://www.cbaird.net

Introduction

The Mattel Hot Wheels Radar Gun is a low powered radar unit which retails for about $30, although Amazon carries it for a little over $21. It has a 4 digit LCD display, two switches (scale and mph/kph), a reset button, and a trigger as the user interface. The LCD display includes a low battery indicator. The radar operates at 10.525 GHz (at least that's what my American version says).

When the user pulls the trigger and holds it, the (approximate) speed of moving objects is shown on the display and updated every second or so. When the trigger is released, the maximum speed seen while the trigger was pulled is displayed. This process may be repeated. After about 20 seconds without the trigger being pulled, the unit turns itself off. The stated range is on the order of 35 feet for large objects.

Ed Paradis (http://www.edparadis.com/radar) shows how to disassemble the unit and has links to additional information. He also traced out and includes a schematic of the analog front end of the unit (a 2 stage amplifier). This has different labels on the components than actually appear on the circuit board, and his op amp chip had no part number (mine says 33078), but it is essentially correct and quite helpful. He also includes C code he wrote for driving the LCD display.

Description

The MCU is an Atmel ATmega88 (8K Flash, 1K SRAM, 512 bytes EEPROM) running at 20 MHz. The unit uses 4 AAA batteries and has a standby current drain of (presumably) about 3 micro amps. The ATmega88 drives the LCD directly, so 12 (of 23) of the AVR's general I/O pins are dedicated to the LCD. The external crystal uses 2 of the remaining 11 pins, and the reset line a third. There is a standard 6 pin ISP solder pad on the exposed side of the circuit board, so it is simple to attach a header for ISP programming. The LCD physically covers the MCU and many of the components on the MCU side of the board.

The trigger consists of 2 switches, one normally open and one normally closed. The normally closed switch connects the regulated +5 to ground through a 330K ohm resistor. This presumably drains the capacitors so the LCD shuts off abruptly rather than engaging in a slow and potentially embarrassing death scene.

The normally open switch applies power to the enable pins of the 3 voltage regulators. VR3 is used to supply the digital circuitry, VR2 supplies the amplifiers, and VR1 supplies the A/D converter input for reading the radar. The regulators always have 6 volts supplied to them but are not enabled (each is supposed to draw 1 micro amp when not enabled), and when the trigger is pulled all three regulators are enabled. One of the MCU I/O pins is also connected to the enable pins of the regulators, so when the MCU powers up it can hold and thus maintain its power when the trigger is released. This is why there is a reset button; obviously reset does nothing to a powered down unit, but it will cause the MCU to drop its hold on the regulators should it ever become necessary.

While the trigger is held in, the MCU pulses the radar unit at 8KHz (100 microseconds off, 25 microseconds on), which responds with a voltage proportional to the speed of the fastest object it senses. There has been much speculation on why the unit is designed this way, since the radar also works with a steady voltage. Ed Paradis thinks it may be to conserve batteries or prevent standing waves, either of which would justify the design.

The voltage from the radar unit is amplified and fed to one of the MCU's A/D converters. The battery voltage is also run through a voltage divider and fed to a second ADC, allowing detection of a low battery condition. The two switches (scale of 1:1 or 1:64 for Hot Wheels cars, and MPH/KPH) ground or float I/O pins.

The state of the trigger is fed to the ADC2 pin so the MCU can tell when to display the average and start the shutdown timeout. When the trigger is pulled, ADC2 should read about 750 mv. When the trigger is released (and the MCU is holding power on), ADC2 should read about 500 mv. Of course, if the MCU is not holding the power on and the trigger is released, ADC2 will be as dead as Liberace, and about as useful.

The original MCU pin assignments are as follows:

MCU Pin	Use	Comment
12	Pin B0	"Scale" switch (leftmost)
13	Pin B1	LCD backplane 4
14	Pin B2	Radar oscillator power (8KHz signal)
15	Pin B3	LCD backplane 3 (also MOSI)

MCU Pin	Use	Comment
16	Pin B4	LCD backplane 2 (also MISO)
17	Pin B5	LCD backplane 1 (also SCK)
7	Pin B6	Crystal – XTAL1
8	Pin B7	Crystal – XTAL2
23	Pin C0	Radar in (ADC1), also test point
24	Pin C1	Trigger (ADC2)
25	Pin C2	"Units" switch (rightmost)
26	Pin C3	Battery voltage (ADC3)
27	Pin C4	To regulators (power on hold, see note)
28	Pin C5	Test point with 51K resistor to ground
29	Pin C6	Reset
30	Pin D0	LCD frontplane bit 0 (note: also RxD)
31	Pin D1	LCD frontplane bit 1 (note: also TxD)
32	Pin D2	LCD frontplane bit 2
1	Pin D3	LCD frontplane bit 3
2	Pin D4	LCD frontplane bit 4
9	Pin D5	LCD frontplane bit 5
10	Pin D6	LCD frontplane bit 6
11	Pin D7	LCD frontplane bit 7

Note: If C4 is left floating (i.e., an input, the default at reset), the regulators will stay on once the trigger is pulled. Therefore, it is necessary to set the C4 pin to an output. Writing a zero to the pin allows the MCU power to follow the trigger, while writing a one holds the power on regardless of the state of the trigger.

Of course the firmware in the original radar gun is protected via the AVR lock bits. This means that once the ATmega88 is erased for reprogramming, there is no way to restore the original application software. Proceed at your own risk!

The only critical part of the original application is the calibration of the voltage to speed conversion. A moving object presents a varying voltage to the ADC. Exactly what that voltage means in miles or kilometers per hour (scaled or not) can be determined empirically, if the application you are designing requires an actual value. It may be that recognizing motion is sufficient.

Modifying the hardware

I wanted to have a serial (RS-232) interface to the unit. In the original design, the LCD is connected to the AVR's built-in USART TxD and RxD pins. While a software USART is certain feasible, so is rerouting some traces on the circuit board. I made the following changes:

Disconnected the RxD and TxD pins from the LCD and brought them out on wires for the serial communications.
Rerouted the trace from C3 (Battery voltage, ADC3) to ADC7, MCU pin 22, which was previously not connected. This trace runs under the MCU and is a bit tricky to get to.
Rerouted C3 and C5 to the LCD pads formerly used by RxD and TxD (C5 goes where D1 used to, and C3 goes where D0 used to for my version of the code). I also removed the 51K resistor (R26, to ground) connected to pin C5.

Warning! These changes are not for the feint of heart! It is tedious, frustrating, and a real head-banging ordeal to try to modify this circuit board because of the size of the traces and component leads. You may end up destroying the circuit board; please do not blame me. You have been warned. The code assumes these changes have been made. Later on I mention a couple of alternatives to making these specific modifications.

Note that RxD and TxD are at the MCU's 5v logic levels, not the standard RS-232 levels. They must be routed through a Max232, an STK500, or other equivalent to allow them to be used with a standard terminal or PC.

The urge to add an LED was almost overwhelming, but I passed. There is another unused ADC (ADC6, pin 19) which is not a general I/O pin, while the amplified radar input signal is brought to ADC0 (C0, pin 23) and the trigger to ADC1 (C1, pin 24), which are general I/O pins. It would be simple enough to reroute either (the trigger would be preferred because of noise issues) to ADC6, thus freeing up C1 for an LED, additional switch, or other use.

The modifications I made messed up the symmetry of the LCD routines, since originally the 8 bits of LCD data gets written to port D. After the modification they have to be juggled around a little so that 2 bits are written to C3 and C5 instead. Since it's a software issue, the good news is it only has to be worked out once.

The LCD

Driving the LCD required looking into how they work. This one is multiplexed with 4 backplanes, or common connections, and pins B5, B4, B3, and B1 go to them. The 8 frontplane or segment bits are written to D0-7 originally, with C3 and C5 replacing D0 and D1 in the modified version. I'm calling the backplanes 1 to 4 to coincide with the variable names (seg1 to seg4) Ed Paradis used in his C code.

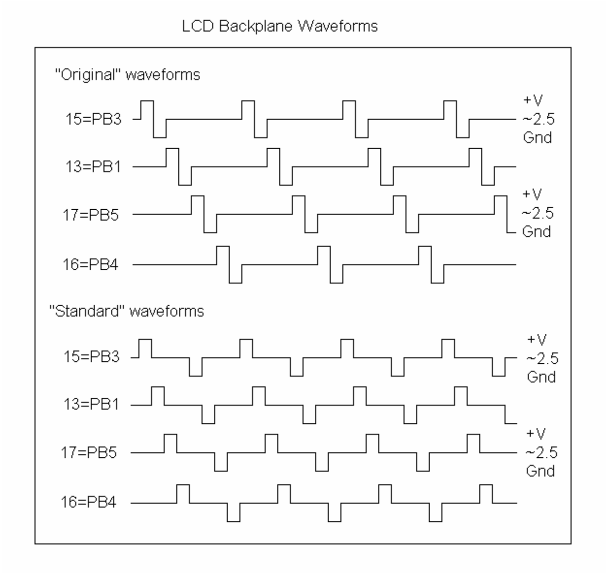
An LCD backplane needs 3 voltage levels: 0, V/2, and V. Generating 0 and V are no problem – just make the pin an output and write a 0 or 1 to it. To generate V/2 the designers put two 56K resistors on each of the 4 backplane I/O pins, one resistor to ground and the other to +V. Then, when the I/O pin is tristated by making it an input pin (with the pull-up off), the voltage divider applies V/2 to the LCD backplane.

Putting an oscilloscope on the four backplane pins revealed a waveform that is inconsistent with what I found in some searching and the one generated by Ed's C program. My routines generate waveforms in agreement with my research and Ed.

One full LCD cycle is split into 8 parts (2 per backplane). A full cycle is 6 ms, or 166.7 Hz. One subdivision is 1.5 ms. An individual backplane is given V for one division, V/2 for 3 divisions, 0 for 1 division, then V/2 for 3 division, and then the sequence repeats. The other backplanes use the same pattern, staggered by one division. Therefore one (and only one) backplane with be either V or 0 at any time, and all the rest will be at V/2.

In the waveforms I observed using the original application, each backplane went through the sequence V, 0, then 6 divisions of V/2. Again, they were staggered in such a way that only 1 was presenting a voltage other than V/2 at any given time.

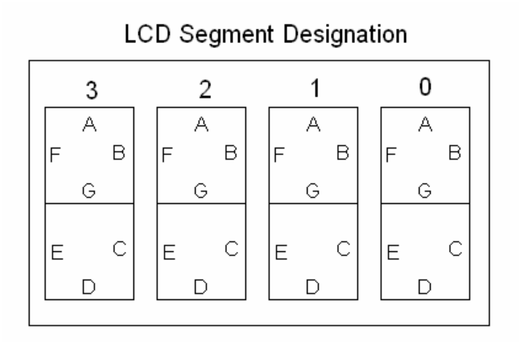
The segment data gets presented to the segment pins (port D originally) twice in each cycle, once when the corresponding backplane is V, and once when it is 0. The data is inverted when the backplane is 0. If a given segment is 1 when V is on the backplane, that segment is transparent, and if it is 0 then it will display.

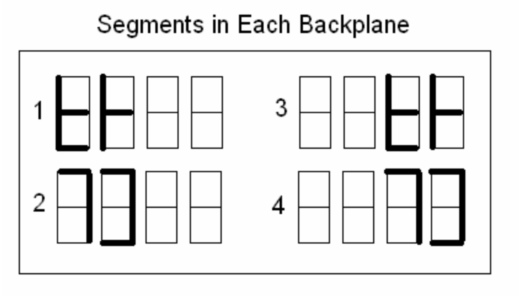


Ed Paradis worked out the segment mappings which I reproduce here:

Back plane	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1 (pin B5)	battery	2F	2G	2E	3D	3E	3G	3F
2 (pin B4)	nc	2A	2B	2C	2D	3C	3B	3A
3 (pin B3)	nc	0F	0G	0E	1D	1E	1G	1F
4 (pin B1)	nc	0A	0B	0C	0D	1C	1B	1A

where the numbers and letters refer to the segments and character positions as shown below. The battery symbol is not shown, but is to the left of position 3.





To write to the LCD, interrupts are generated about every 0.75 ms (ideally 1333 Hz). Each interrupt will manipulate one of the backplane voltages using B1, B3, B4, or B5, setting that voltage to 0 (output, write 0), V/2 (input, no pull-up), or V (output, write 1). If the voltage is set to V, then the corresponding segment bits for that backplane are written to port D (in the original, or D and C in the modified). If the voltage is set to 0, the complement of the corresponding segment bits is written. A foreground routine collects the segments and builds the 4 backplane bytes depending on which segments are to be displayed.

In summary, on each interrupt one of the four bytes or its 1's complement will be written and the corresponding backplane voltage will go to 0 or V.

Command Language

The following is an ASCII command language which makes it easy for terminal (human) interactions. Most of the commands are one byte long.

The command interpreter prompts with '?' when it is ready to receive input. If echo is on, the input characters will be returned to the host. If CRLF is on, a carriage return/line feed combination will precede the prompt and any responses to a given command.

All the features of the gun are available and can be controlled. The LCD may be directly written, either with numbers or individual segments. A cursor may be emulated which can move left to right or right to left.

ASCII command language

Command	Char	Return / Comments
Power On	`P`	Overrides trigger release
Power Off	`p`	Kills power if/when trigger released
Radar Constant	`_`	Do not pulse radar when on
Radar Pulsed	`~`	Pulse radar (8KHz) when on
Radar On	`G`	Turn radar on
Radar Off	`g`	Turn radar off
Erase LCD	`X`	Blank the LCD and home cursor
Blank LCD pos	`blank`	Blank the LCD's cursor position & move cursor
Blank LCD pos	`x`	Blank the LCD's cursor position; do not move cursor
Position cursor	`H-K`	Set cursor position (H = rightmost, 1's)
Cursor Left	`<`	Move cursor left after number, hyphen, or blank. Home will be rightmost (1's) position.
Cursor Right	`>`	Move cursor right left after number, hyphen, or blank. Home will be leftmost (1000's) position.
Hold Cursor	`.`	Do not advance cursor. Home will be rightmost.
Segment On	`h-n`	Light segment 0-6 at cursor
Digit	`0-9`	Display digit at cursor (and maybe advance)
Hyphen	`-`	Display – at cursor and advance
Battery On	`B`	Battery symbol on
Battery Off	`b`	Battery symbol off
Echo On	`E`	Echo received characters
Echo Off	`e`	Do not echo received characters
CRLF On	`C`	Precede prompt with CRLF
CRLF Off	`c`	Do not output CRLF
User hex #	`=####`	4 digit hexadecimal number, 2's complement
Output to display	`D`	User #, radar and battery to display – default Valid range is -999 to 9999
ASC output to user	`U`	User #, radar and battery to USART, decimal ASCII
Hex output to user	`u`	User #, radar and battery to USART, hex ASCII
Load table	`L`	Load the translation table from EEPROM
Save table	`S`	Save the translation table to EEPROM
Add point to table	`+`	Add point to table (followed by 3 hex bytes of value, then 4 hex bytes of return value)
Clear table	`^`	Make default translation table
Conversion On	`A`	Use table for translation. Applies to radar, battery, and user input (= cmd) numbers only.
Conversion Off	`a`	Do not translate (default)
Read Radar	`R`	Read radar, value to output "device"
Read Battery	`r`	Read battery, value to output "device"
Read Switches	`$`	Read switches, send "0" to "7" to USART
Start clock, seconds	`T`	Starts timer running, incremented every second
Start clock, 0.1 sec	`t`	Starts timer running, incremented 0.1 seconds
Stop clock	`z`	Stops the clock. Does not reset counters
Reset clock	`Z`	Resets clock counters, leaves clock on/off as it was

The clock can display (to LCD or USART) either every second or every 1/10th seconds. It isn't particularly accurate, but it's close (and could be made more accurate). The total count will be kept within the 0 to 9999 range (9,999 seconds or 999.9 seconds) and will then wrap to zero. When the clock is running and sending output to the USART, there is a high potential that echoed characters and CRLFs will get intermixed with the clock output.

It would be easy enough to make a standalone stopwatch function using the switches and trigger – this is left as an exercise to the student.

Value translation (conversion)

The user has the option of constructing a translation table within the ATmega88 to convert values, and this table may be saved in the EEPROM for nonvolatility. Thus, once the radar gun has been calibrated, its mapping from raw ADC values to display values may be kept and reused. 10 bit values (0 to 1023 decimal) can be translated, since this is the range of the A/D converters. The translation can be turned on or off (the default is off).

A value for translation may come from any of three sources: the user may input it via the serial line (in hexadecimal as 4 ASCII characters), by reading the battery ADC, or by reading the radar ADC. These values, if they are to be translated, must be in the range 0 to 1023 or an error value will result.

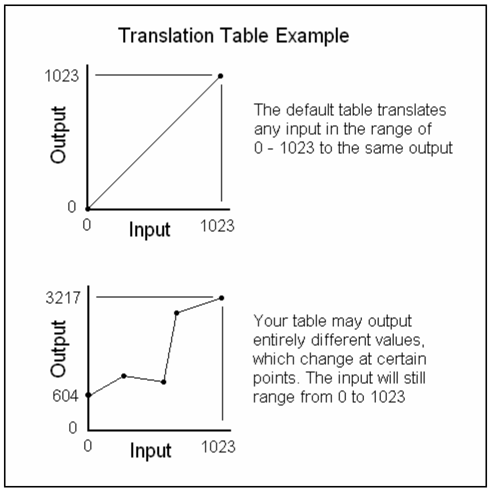
Once a value is received from one of these sources, there are three places where its (possibly translated) value will be output: on the display (the default), sent to the user via the serial line as decimal (1 to 6 ASCII characters), or sent to the user as hexadecimal (1 to 4 ASCII characters). The LCD output is limited to the decimal values -999 to 9999.

Thus the unit may be used directly as a hexadecimal to decimal converter. Set the output to serial decimal ("U" command) and type a hexadecimal number ("=0300" for example), and the equivalent decimal number will be returned. You may also use the LCD, but then the displayed values are limited to -999 to 9999.

When the MCU resets, an empty translation table is constructed, translation is turned off, and the display is set as the numeric output device. The user may construct a new table or read a saved one from EEPROM. After enabling translation, the user may verify the table by sending values and having them returned or displayed translated.

The default translation table consists of two entries, 0 and 1023, which output 0 and 1023 respectively. That is, no translation. Because of integer arithmetic and rounding, exact translations may not be possible. Additional points may be added to the table, up to 100 total, using the '+' command. Following the '+' are 7 hex digits. The first 3 give the node value (0 to 1023) and the last 4 give the value (-32,768 to +32,767) which that input maps to. Thus, if you wanted 0 to output 500 and 1023 to output 700 and anything in between to be linearly interpolated, the command sequence would be "+ 000 01F4 + 3FF 02BC" without any spaces. Each consecutive pair of points in the table defines a line segment with a starting and ending return value. Looking at a graph might make it clearer, where the X axis are the input values, and the Y axis are the returned values.

Input values landing between existing points return values which are linearly interpolated on the corresponding line segment.



In this example, the top table is the one that is automatically constructed. The user would create the bottom table by using the '+' command 5 times: twice to redefine the points 0 (to output 604) and 1023 (to output 3217), and three times to add the three points in the middle along with their output values. Thus any "x" value between 0 and 1023 will translate to its corresponding "y" value on the shown line segments.

Using the command interpreter

When the MCU resets, usually from having the trigger pulled, the command processor will output a '?' prompt. If the trigger is released without receiving a **Power On** command ('**P'**) on the USART, the MCU shuts down as the power drops. If the **Power On** is received, the MCU maintains its power and releasing the trigger has no direct effect (although it can be detected by reading the switches).

If you try to read the radar when it's not being fired, it will return -999. Again, there seems to be no easily discernable difference between firing the radar pulsed (the way Mattel does it, probably to save batteries) or steady. The default is pulsed.

A typical sequence might be (assume the translation table has been defined earlier and saved):

User pulls trigger, gets '?' prompt
'P' to hold power on
'L' to read translation table from EEPROM
'A' to turn conversion on
'G' to turn radar on
'R' to read the radar, translate (correct) the value, and display it
'g' to turn the radar off

‘p’ to turn power off, provided the trigger has been released

Other variations could include getting the user to release the trigger via some displayed symbols, then using the trigger pull/release to turn on the radar, repeatedly read and display the reading, and turning off the radar. In the original toy, once the trigger is pulled they override the power switch and use the trigger to read/display the radar readings. When the trigger is released, they display the highest reading obtained, and wait about 20 seconds before releasing the power.

It would be fairly easy to code additional (not just one) translation tables (for example, MPH and KPH), and even a simple scripting language which could be saved in EEPROM. Then automated sequences with conditional execution and flow could be saved and run on demand.

Easier hardware modifications than the ones I did could be attempted. For example, writing a software USART and using pin C5 as a half duplex communication line would probably be the easiest hardware change. There is a solder pad to C5 easily accessible, and all that would need to be done would be to remove its resistor to ground. Of course the software would need some changes, but this is likely in any case.

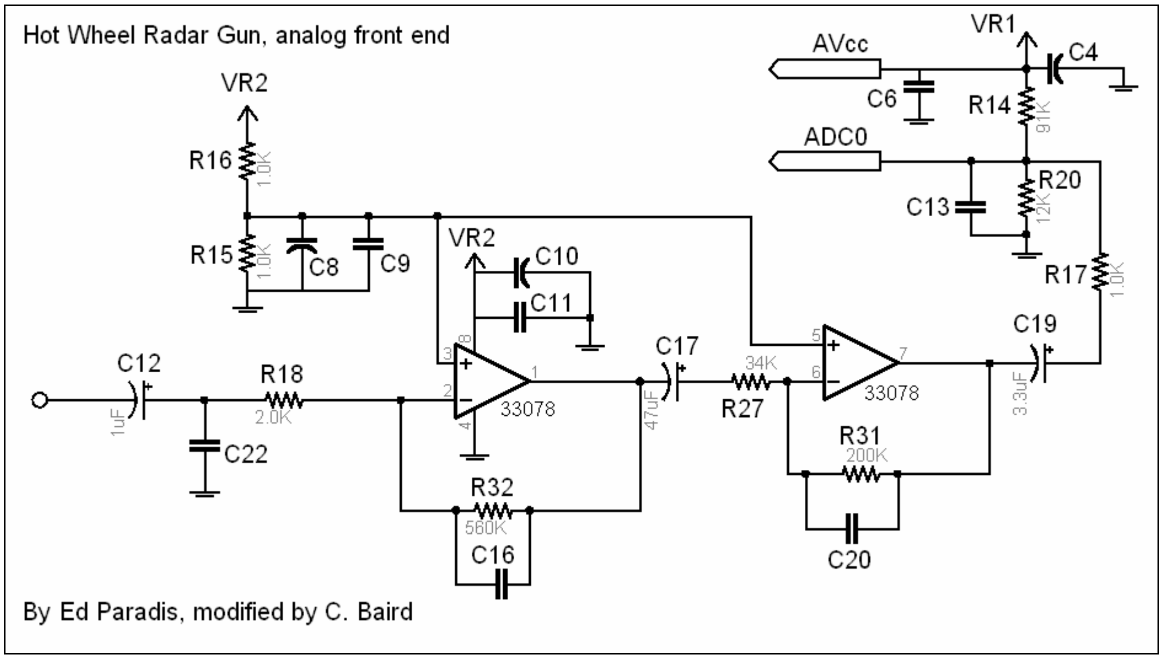
An even easier modification would be to just toss the original LCD entirely and replace it with a standard (HD44780) 2x14 character LCD. Removing the LCD reveals a long row of pcb pads, most of which go directly to the MCU I/O pins. The LCD backplane connections have the resistor networks for generating V/2 for the LCD (which could be removed if necessary), and 3 of them also go to the ISP connector. However, in 4 bit mode the standard LCD would only need 7 I/O lines, so the resistor networks could be left intact. Furthermore, it would require only soldering on the connecting pads, not cutting and rerouting traces. The two USART connections also end in pads to the original LCD. OK, I lied. You need to either disable one of the resistor networks or use C5 and remove its resistor to ground to have enough lines (you are freeing up 12, 4 of which have resistors, you need 7 for the new LCD, but RxD and TxD are two of the 12).

The software as written is more as a “proof of concept” than it is meant to be a finished product anyway. There are several places where it could be dramatically improved, including the use of macros for some of the trivial functions, the use of power saving and noise reducing (for the ADCs) techniques, and the like. I also detest obscure C (something many programmers seem to love to indulge in), so some of the coding is almost trivially written and could be tightened up.

Even if you do not choose to modify your hardware (again, please don’t blame me if you attempt it and have a disaster), hopefully this document and the attached code will prove helpful in your exploration of the radar gun.

Addendum

This is the front end diagram by Ed Paradis, with slight modification by me. I have attempted to label the various parts with the numbers off the circuit board, plus I've indicated which voltage regulators feed which parts of the circuit.



The Code

This is written in ICCavr C. The main thing that could bite you on the tail would be that ICCavr considers all **char**s to be **unsigned char**s, while others may not. You might have to change all (to be safe) my **char**s.

radar.c

```
// Mattel Hot Wheels Radar Gun
// Serial interface and command processor
//
// ATMega88, 20.000Mhz; ICCAVR compiler
//
// Written by Chuck Baird, http://www.cbaird.net
// This may be freely distributed and modified. Attribution
// would be appreciated. Outright theft will bring you bad karma.
// No warranties, gurarantees, liability, etc. implied or assumed.
// Use at your own risk.
//
// This is for a modified version of the radar gun.  A couple of
// the LCD lines have been rerouted to access the USART, and the
// trigger was moved to a different ADC input to free up an I/O bit.
// See the accompanying documentation for details.
```

```
#define EXTERN
#include "radar_decl.h"

// ----------------------------------------------------------------
void main(void)
{
  int v;
  char c;

  init();                                 // initialize some goodies
  c = 32;                               // force initial prompt

  for (;;) {
    if (c > 31) {                       // don't prompt on clock loops
        crlf_me();
      usart_out('?');                   // prompt those doggies
      }

      c = 0;                              // no clock as faked input
    while (!(UCSR0A & (1 << RXC0))) {   // watch for incoming
      if (clock == 1) {                 // are we watching seconds?
          if (t_1serv) {
              t_1serv = 0;
              c = 1;
              break;
            }
        }
        if (clock == 2) {                 // are we watching 1/10 secs?
            if (t_10serv) {
              t_10serv = 0;
              c = 2;
              break;
            }
        }
      }
    if (!c) {                           // if it's not clock, read char
        c = usart_in();
        if (c < ' ') c = '!';
        echo_me(c);
      }

      switch (c) {
        case 1:              // one second has elapsed
          num_out(t_1);
          break;
        case 2:              // one tenth of a second has elapsed
          v = (t_1 * 10 + t_10) % 10000;
            num_out(v);
          break;
        case 'P':            // power on (override trigger release)
          power(1);
          break;
        case 'p':            // power off (trigger controls power)
          power(0);
          break;
        case 'G':            // radar on
          fire_radar(1);
          break;
        case 'g':            // radar off
          fire_radar(0);
          break;
        case 'R':            // read radar, output to "device"
          if (radar) {
            v = read_adc(0);
              if (xlit) v = translate(v);
            } else v = -999;
            num_out(v);
          break;
        case 'r':            // read battery, output to "device"
          v = read_adc(7);
            if (xlit) v = translate(v);
            num_out(v);
          break;
        case '$':            // read switches, "0" - "7" to usart
          crlf_me();

            num_out(read_adc(1));

//        usart_out(read_switches());
          break;
        case 'X':            // erase LCD and home cursor
          lcd_erase();
          break;
        case ' ':            // blank LCD at cursor and advance
          lcd_cnum(11);
          break;
        case 'x':            // blank LCD at cursor, no move
          lcd_num(cursor, 11);
          break;
        case 'A':            // translation on
          xlit = 1;
          break;
        case 'a':            // translation off (default)
          xlit = 0;
          break;
        case 'L':            // load xlit table from EEPROM
          load_xlit();
          break;
        case 'S':            // save xlit table to EEPROM
          store_xlit();
          break;
        case '^':            // clear xlit table
          xlit_clear();
          break;
        case '+':            // add node to table
          add_node();
          break;
        case 'D':            // output device is LCD (default)
          outdev = 0;
          break;
        case 'U':            // output device is usart - ASCII
          outdev = 1;
          break;
        case 'u':            // output device is usart - hexadecimal
```

```
          outdev = 2;
          break;
        case '<':            // cursor moves left (home 1's; default)
          curadd = 1;
          break;
        case '>':            // cursor moves right (home 1000's)
          curadd = 7;
          break;
        case '.':            // do not advance cursor (home 1's)
          curadd = 0;
          break;
        case '=':            // input 4 char ASCII hex number from
usart
          v = user_input(4);
            if (xlit) v = translate(v);
            num_out(v);
          break;
        case '~':            // radar pulsed when fired (default)
          pulsed = 1;
          break;
        case '_':            // radar constant (not pulsed) when fired
          pulsed = 0;
          break;
      case 'E':            // echo on
          echo = 1;
          break;
        case 'e':            // echo off
          echo = 0;
          break;
        case 'C':            // CRLF on
          crlf = 1;
          break;
        case 'c':            // CRLF off
          crlf = 0;
          break;
        case 'B':            // battery symbol on
          lcd_seg(7, 1);
          break;
        case 'b':            // battery symbol off
          lcd_seg(7, 0);
          break;
        case '-':            // minus sign at cursor
          lcd_cnum(10);
            break;
        case 'T':            // seconds timer on
          s_clock(1);
            break;
        case 't':            // tenths seconds timer on
          s_clock(2);
            break;
        case 'Z':            // clock stop
          s_clock(0);
          break;
        case 'z':            // clock reset
          s_clock(clock + 4);
            break;
        default:             // something else
```

```
          if ((c >= '0') && (c <= '9')) lcd_cnum(c - '0');
            else if ((c >= 'H') && (c <= 'K')) cursor = c - 'H';
            else if ((c >= 'h') && (c <= 'n')) lcd_seg(c - 'h', 1);
            break;
      }
  }
}

// ---------------------------------------------------------------
void init(void)
{
//  In the modified wiring:
//  Pin   Dir  Function
//  ---   ---  --------
//  B0    in   "scale" switch
//  B1    var  LCD backplane
//  B2    out  radar oscillator (timer 1 drives this)
//  B3    var  LCD backplane
//  B4    var  LCD backplane
//  B5    var  LCD backplane
//  B6         crystal
//  B7         crystal
//  ---
//  C0         radar in (ADC)
//  C1         trigger in (ADC)
//  C2    in   "unit" switch
//  C3    out  LCD segment
//  C4    out  trigger override (power on)
//  C5    out  LCD segment
//  C6         reset
//  C7         does not exist
//  ---
//  D0         RxD              (timer 0 drives USART)
//  D1         TxD
//  D2-7  out  LCD segment

  DDRB  = 0x04;         // radar oscillator is output
  PORTB = 0x05;         // B0 pullup, B2 = 1 turns oscillator off

  DDRC  = 0x38;         // 2 new LCDs & power hold are outputs
  PORTC = 0x04;         // pullup for C2 (unit switch)

  DDRD  = 0xFC;         // D0 & D1 in, others (LCD segs) out

  DIDR0 = 0x03;         // digital input disable, ADCs

// set up timer 1 for 8KHz radar PWM; leave it off
  TCCR1A = 0x03;        // timer 1 = 8KHz PWM mode 15, no out now
  TCCR1B = 0x48;        // clock = 000 means not running
  OCR1AH = 0;           // top value, high byte (PWM set)
  OCR1AL = 39;          // top value, low byte
  OCR1BH = 0;           // match value, high byte (PWM cleared)
  OCR1BL = 31;          // match value, low byte

  outdev = 0;           // output to LCD (1 = ASCII, 2 = binary)
  echo = 0;             // no input echo
  crlf = 0;             // no appended CRLF
  cursor = 0;           // cursor is in the 1's position
  xlit = 0;             // translation off
  curadd = 1;           // cursor moves to the left
  pulsed = 1;           // radar is pulsed

  usart_init();         // and the usart
  xlit_clear();         // build a new xlit table
  s_clock(4);           // stop and reset clock
  lcd_init();           // set up the LCD
  SEI();                // and enable the interrupts
}

// ---------------------------------------------------------------
// power() - turn trigger power override on/off
//
// x - 0 for off, 1 for on

void power(char x)
{
  if (x) PORTC |= 0x10; else PORTC &= 0xef;
}

// ---------------------------------------------------------------
// s_clock(w) - handle the seconds counting clock
//
// w = 0  stop        add 4 to reset counters
//   = 1  seconds
//   = 2  tenths

void s_clock(char w)
{
  if (w & 0x04) {
    clock = 0;                 // stop it while resetting
    t_10 = 0;                  // tenth seconds counter
    t_1 = 0;                   // seconds counter (0 - 9999)
    t_10serv = 0;              // if 1, tenths have changed
    t_1serv = 0;               // if 1, seconds have changed
    t_small = 133;             // interrupt countdown, 0.1 secs
  }
  clock = w & 0x03;
}

// ---------------------------------------------------------------
// echo_me(w) - maybe echo the incoming character

void echo_me(char w)
{
  if (echo) usart_out(w);
}

// ---------------------------------------------------------------
// crlf_me() - maybe send out a crlf

void crlf_me(void)
{
  if (crlf) {
    usart_out('\r');
    usart_out('\n');
  }
}
```

radar_decl.h

```
// Mattel Hot Wheels Radar Gun
//
// Radar_decl.h file
//
// Written by Chuck Baird, http://www.cbaird.net
// This may be freely distributed and modified. Attribution
// would be appreciated. Outright theft will bring you bad karma.
// No warranties, gurarantees, liability, etc. implied or assumed.
// Use at your own risk.
//
#ifndef EXTERN
#define EXTERN extern
#endif

#include <iom88v.h>
#include <stdlib.h>
#include <string.h>
#include <macros.h>
#include <eeprom.h>

// ---------------------------- radar.c
void init(void);
void power(char);
void s_clock(char);
void echo_me(char);
void crlf_me(void);

// ---------------------------- Radar_IO.c
void load_xlit(void);
void store_xlit(void);
void radar(char);
void add_node(void);
int user_input(char);
int translate(int);
void xlit_clear(void);
char read_switches(void);
int read_adc(char);
void num_out(int);
void usart_init(void);
char usart_in(void);
void usart_out(char c);

// ---------------------------- Radar_LCD.c
void lcd_num(char, char);
void lcd_cnum(char);
void lcd_seg(char, char);
void lcd_init(void);
void lcd_erase(void);
```

```
// --------------------------- globals and suchnot

// MAXXLIT cannot exceed 126 for several reasons
#define MAXXLIT 100

struct trans {                    // transliteration table
  int node;                       // node point (x)
  int val;                        // value of node (y)
};

EXTERN struct trans xtbl[MAXXLIT];   // transliteration table
EXTERN char nxtbl;               // entries in xtbl[]
EXTERN char segs[4];             // current LCD output
EXTERN char echo;                // whether to echo input or not
EXTERN char crlf;                // whether to add crlf (with echo or
send)
EXTERN char cursor;              // cursor location (0 - 3)
EXTERN char curadd;              // cursor movement (0=none, 1=left,
7=right)
EXTERN char xlit;                // transliteration on (1) or off (0)
EXTERN char pulsed;              // whether radar is pulsed (=1) or not
(=0)
EXTERN char outdev;              // output (0=LCD, 1=ASC USART, 2=hex
USART)
EXTERN char radar;               // radar off (0), pulsed (1), or
steady (2)

EXTERN char clock;               // 0 = off, 1 = seconds, 2 = tenths
EXTERN volatile char t_10;       // tenth seconds counter (0 - 9)
EXTERN volatile int t_1;         // seconds counter (0 - 9999)
EXTERN volatile char t_10serv;   // if 1, tenths have changed
EXTERN volatile char t_1serv;    // if 1, seconds have changed
EXTERN char t_small;             // interrupt counter

extern const char tab[];
extern const char killer[];
extern const char smap[];
```

radar_IO.c

```
// Radar_IO.c
//
// general routines for the Mattel Hot Wheels Radar Gun
//
// Written by Chuck Baird, http://www.cbaird.net
// This may be freely distributed and modified. Attribution
// would be appreciated. Outright theft will bring you bad karma.
// No warranties, gurarantees, liability, etc. implied or assumed.
// Use at your own risk.

#include "radar_decl.h"

// local stuff:
char x_find(int);
```

```
// ---------------------------------------------------------------
// load_xlit() - read xlit table from EEPROM
//
// creates a new empty table if it doesn't like what it reads

void load_xlit(void)
{
  char i;
  int addr, v;

  EEPROM_READ(0, nxtbl);
  if (nxtbl < 2 || nxtbl > MAXXLIT) {
    xlit_clear();
      return;
  }
  addr = 1;
  for (i = 0; i < nxtbl; i++) {
    EEPROM_READ(addr, v);
      if (v < 0 || v > 1023) {
        xlit_clear();
        return;
      }
      xtbl[i].node = v;
      EEPROM_READ(addr + 2, xtbl[i].val);
      addr += 4;
  }
  if (xtbl[0].node != 0 || xtbl[nxtbl - 1].node != 1023) xlit_clear();
}

// ---------------------------------------------------------------
// store_xlit() - write xlit table to EEPROM

void store_xlit(void)
{
  char i;
  int addr;

  EEPROM_WRITE(0, nxtbl);            // number of entries
  addr = 1;                          // next address
  for (i = 0; i < nxtbl; i++) {
    EEPROM_WRITE(addr, xtbl[i].node);
      EEPROM_WRITE(addr + 2, xtbl[i].val);
      addr += 4;
  }
}

// ---------------------------------------------------------------
// fire_radar(v) - turn radar off or on, pulsing or steady
//
// v = 0 for off, 1 for on

void fire_radar(char v)
{
  TCCR1B = 0x18;               // stop the PWM timer
  TCCR1A = 0x03;               // release B2
  PORTB |= 0x04;               // radar osc off: B2 <-- 1
  radar = 0;
  if (v) {
    if (pulsed) {
        radar = 1;
        TCNT1H = 0;               // reset counter (not necessary)
        TCNT1L = 0;
      TCCR1A = 0x33;           // B2 (OC1B) under control of PWM
        TCCR1B = 0x1b;            // start the timer running
      } else {
        radar = 2;
        PORTB &= 0xfb;            // osc on steady (B2 <-- 0)
      }
  }
}

// ----------------------------------------------------------------
// add_node() - add a node to the transliteration table
//
// user must input 3 hex digits for node (000 - 3FF), followed by
// 4 hex digits for the node Y value (0000 - FFFF)

void add_node(void)
{
  char i, j;
  int nnum, v;

  nnum = user_input(3);             // node value (x)
  v = user_input(4);                // table value (y)

  if (nxtbl >= MAXXLIT) return;
  if ((j = x_find(nnum)) == 255) return;    // ignore out of range
  if (j > 127) {                   // if an exact match
    j &= 0x7f;                     // drop the high bit
      xtbl[j].val = v;               // replace the value
  } else {
    // put in a new node at xtbl[j+1]
      tomove = nxtbl - j - 1;        // number that have to scoot
    for (i = 0; i < tomove; i++) xtbl[nxtbl - i] = xtbl[nxtbl - i - 1];
      nxtbl++;                       // one more in table
    xtbl[++j].node = nnum;         // put values away in new node
      xtbl[j].val = v;
  }
}

// ----------------------------------------------------------------
// user_input(n) - get an n character hex number from the USART
//
// n is 3 or 4. length 3 will always be +, length 4 may be +/-

int user_input(char n)
{
  char buf[5], i, s, c;

  buf[0] = '0';
  s = (n == 4) ? 0 : 1;
  for (i = s; i < 4; i++) {
    echo_me(c = usart_in());
    buf[i] = c;
  }
  buf[4] = 0;
  return ((int) strtol(buf, NULL, 16));
}

// ----------------------------------------------------------------
// translate(v) - apply the transliteration table
//
// input must be 0 - 1023, or error occurs (returns as 0x7fff = 32,767)

int translate(int v)
{
  char j;
  long g;

  j = x_find(v);                             // locate "floor" node
  if (j == 255) return (0x7fff);             // out of range
  if (j > 127) return (xtbl[j & 0x7f].val);  // exact match

  g = v - xtbl[j].node;                      // interpolate
  g *= (long) (xtbl[j + 1].val - xtbl[j].val);
  g /= (long) (xtbl[j + 1].node - xtbl[j].node);
  return (xtbl[j].val + (int) g);
}

// ----------------------------------------------------------------
// xlit_clear() - clear the transliteration table
//
// builds a trivial xlit table with two entries, 0 and 1023

void xlit_clear(void)
{
  nxtbl = 2;
  xtbl[0].node = 0;
  xtbl[0].val = 0;
  xtbl[1].node = 1023;
  xtbl[1].val = 1023;
}

// ----------------------------------------------------------------
// x_find(v) - locate a node interval for a given value
//
// v - value to look up, 0 to 1023
// returns:
//   0 to (MAXXLIT - 1), plus bit 7 set if exact match
//   note: MAXXLIT cannot exceed 126
//   255 if out of range

char x_find(int v)
{
  char i;

  if (v < 0 || v > 1023) return (255);
  for (i = nxtbl - 1; i >= 0; i--) {
    if (v > xtbl[i].node) return (i);
      if (v == xtbl[i].node) return (i | 0x80);
  }
  return (0x80);             // impossible
}

// ----------------------------------------------------------------
// read_switches() - read the 3 switches
//
// returns ASCII "0" to "7" - sum up the following:
//   1 - scale (left) switch
//   2 - units (right) switch
//   4 - trigger (ADC reads at about 115 when off, 158 when pressed)
// "on" is considered top of switch pointing right (1:1 and KPH)

char read_switches(void)
{
  char v;

  v = PINB & 0x01 ? 0 : 1;         // scale switch (sw is backwards)
  if (PINC & 0x04) v += 2;         // units switch
  if (read_adc(1) > 136) v += 4;   // trigger
  return ('0' + v);
}

// ----------------------------------------------------------------
// read_adc(who) - read one of the ADCs
//
// w = 0, radar in
//   = 1, trigger
//   = 7, battery (rewired)
//
// if we're reading the radar in pulsed mode, then we want to
// time our read with the radar being on (pin low)

int read_adc(char w)
{
  int v, s, hi, lo;
  char i;

  ADMUX = 0x40 | (w & 0x0f);     // select input and reference
  ADCSRA = 0x87;                 // enable with prescale = 128

  ADCSRA |= 0x40;                // start a conversion
  while (ADCSRA & 0x40) ;        // wait for it
  v = ADCH;                      // toss it
  v = 0;                         // accumulate 10 samples here
  hi = 0;                        // get largest here
  lo = 1023;                     // get smallest here

  for (i = 0; i < 10; i++) {     // take 10, toss highest and lowest
    if (!w && pulsed) {          // are we reading pulsed radar?
        while (!(PORTB & 0x04)) ;  // wait until it goes high (radar
off)
        while (PORTB & 0x04) ;     // and then until it goes low (radar
on)
      }
    ADCSRA |= 0x40;               // start a conversion
    while (ADCSRA & 0x40) ;       // wait for it
    s = ADCL;                     // low order
      s += ADCH << 8;               // and high order
      if (s > hi) hi = s;
      if (s < lo) lo = s;
      v += s;
  }
  ADCSRA = 0x00;                  // shut it off
  return ((v - hi - lo) >> 3);    // keep 8 and average them
}

// ----------------------------------------------------------------
// num_out() - put a number to the output device
//
// out of range shows up as vertical bars on LCD

void num_out(int v)
{
  char buf[7], i, j, c;

  i = 16;                                 // assume hex
  switch (outdev) {
    case 0:                               // LCD
      if ((v < -999) || (v > 9999)) {
        for (i = 0; i < 4; i++) lcd_num(i, 13);
        return;
      }
      itoa(buf, v, 10);
      lcd_erase();
      j = strlen(buf);
      i = 0;
      while (j > 0) {
        c = buf[--j];
        if (c == '-') c = ':';            // minus sign --> 10
        lcd_num(i++, c - '0');
      }
        break;
    case 1:                              // ASCII
        i = 10;
      default:                             // hex digits
        itoa(buf, v, i);
        j = 0;
        crlf_me();
        while (buf[j]) usart_out(buf[j++]);
        break;
  }
}

// ----------------------------------------------------------------
// usart_init() - initialize the USART
//
// 19,200 baud, 8N2

void usart_init(void)
{
  UBRR0L = 64;          // baud rate lo (20MHz)
  UBRR0H = 0x00;        // baud rate hi
  UCSR0A = 0x00;        // normal speed
  UCSR0C = (1<<USBS0) | (3<<UCSZ00);
  UCSR0B = (1<<RXEN0) | (1<<TXEN0);
}

// ---------------------------------------------------------------
// usart_in() - wait for an incoming char from the USART

char usart_in(void)
{
  while (!(UCSR0A & (1 << RXC0))) ;
  return UDR0;
}

// ---------------------------------------------------------------
// usart_out() - send a char to the USART

void usart_out(char c)
{
  while (!(UCSR0A & (1 << UDRE0))) ;
  UDR0 = c;
}
```

radar_LCD.c

```
// Mattel Hot Wheels Radar Gun
//
// These are the LCD routines and support
//
// Written by Chuck Baird, http://www.cbaird.net
// This may be freely distributed and modified. Attribution
// would be appreciated. Outright theft will bring you bad karma.
// No warranties, gurarantees, liability, etc. implied or assumed.
// Use at your own risk.
//
// with rewiring mod, PD0 --> PC3
//                    PD1 --> PC5
//
// we need to diddle one of the backplane voltages about every 0.75ms
// (1333 Hz). Each goes V for an interval, V/2 for 3 intervals, 0 for
// an interval, and V/2 for 3 intervals, and repeats. The 4 backplanes
// are sequenced such that one of them is always either at V or 0.
//
// when at V, the contents of seg# for that backplane is written.  when
// at 0, the 1's complement of that value is written. thus, either seg#
// or ~seg# is written on every interrupt.
//
// to set a backplane voltage at V, the pin is set to output and a 1 is
// written. to set voltage at 0, the pin is set to output and a 0 is
// written. for V/2 the pin is set to input with no pullup.

#include "radar_decl.h"

char scnt;                   // which pass we're in for interrupts
```

```
// ---------------------------------------------------------------
// lcd_cnum(value) - write individual digit/char to LCD using cursor
//
// write a number, hyphen, blank, or other predefined pattern to LCD
// using the cursor (with cursor movement, as appropriate)
// v - 0-9, 10 for hyphen, 11 for blank, 12 for horiz bars,
//     13 for vertical bars, 14-15 undefined

void lcd_cnum(char v)
{
  lcd_num(cursor, v);
  cursor += curadd;
  cursor &= 0x03;
}

// ---------------------------------------------------------------
// lcd_num(position, value) - write individual digit/char to LCD
//
// write a number, hyphen, blank, or other predefined pattern to LCD
// pos - position; 0 = rightmost (1's), 3 = leftmost (1000's)
// v - 0-9, 10 for hyphen, 11 for blank, 12 for horiz bars,
//     13 for vertical bars, 14-15 undefined

void lcd_num(char pos, char v)
{
  unsigned char i, j;           // with ICC unsigned isn't necessary

  pos &= 0x03;                  // we like legal args
  v &= 0x0f;
  j = pos * 4;

  for (i = 0; i < 4; i++)       // drop the old bits for this position
    segs[i] &= killer[j + i];

  j = v * 12;                   // start of this one's defs
  if (pos & 0x01) j += 6;       // offset: 0=0, 1=+6, 2=+2, 3=+8
  if (pos & 0x02) j += 2;
  for (i = 0; i < 4; i++)
    segs[i] |= tab[j++];
}

// ---------------------------------------------------------------
// lcd_seg - turn a specific segment at the cursor on/off
//
// s - segment 0 - 6 (A-G) or 7 for battery symbol
// x - 0 to turn off, 1 to turn on
// no cursor movement

void lcd_seg(char s, char x)
{
  char w, m, t;

  w = smap[s * 4 + cursor];     // byte in hi nibble, bit in lo nibble
  t = (w & 0x30) >> 4;
  m = 1 << (w & 0x07);                 // build mask
  if (x) segs[t] |= m; else segs[t] &= ~m;
}

// ---------------------------------------------------------------
// lcd_init - set up the lcd stuff
//
// we'll use timer 0 to run the LCD updates

void lcd_init(void)
{
  lcd_erase();                // clear the segment bytes

  TCCR0A = 0x02;              // ctc mode 2
  TCCR0B = 0x03;              // prescaler = 64
  TCNT0 = 0;                  // counter (not necessary)
  OCR0A = 245;                // 1332.03 Hz (wanted 1333.333, but...)
  TIMSK0 = 0x02;              // compare A interrupt on




//  TCCR1A = 0;                 // ctc
//  TCCR1C = 0;
//  TCNT1H = 0;                 // zero the counter (not necessary)
//  TCNT1L = 0;                 // part deux;
//  OCR1AH = 0;                 // high part of compare (write hi then
lo)
//  OCR1AL = 0xf5;              // 245 = freq of 1332.0325
// TIMSK1 = 0x02;              // enable compare A interrupts
//  TCCR1B = 0x0b;              // ctc, 64 prescale, start timer
}

// ---------------------------------------------------------------
// lcd_erase() - blank the LCD and home cursor

void lcd_erase(void)
{
  int i;

  for (i = 0; i < 4; i++) segs[i] = 0;
  if (curadd == 7) cursor = 3; else cursor = 0;
}

// ---------------------------------------------------------------
// Timer1 Compare A Handler - 1,332.03 Hz
//
// we come here every 1/8 of a complete LCD cycle. we need to dingle
some
// backplane voltages and put out a value to port D (mostly).
//
// we may also be running a seconds and tenth-seconds (more or less)
// clock which sets flags (t_1serv and t_10serv) when they go by.
//
// entry  backplane  data
// -----  ---------  ----
//  0/4      B5      seg1
//  1/5      B4      seg2
//  2/6      B3      seg3
//  3/7      B1      seg4

#pragma interrupt_handler timer0_compa_isr:iv_TIM0_COMPA
void timer0_compa_isr(void)
{
  char dirb;                 // DDRB
  char prtb;                 // PORTB
  char v;                    // segments to light
  char w;                    // scratch

  if (clock) {               // are we timing?
    t_small--;               // 0.1 seconds counter
      if (!t_small) {
        t_small = 133;         // ran out - restart it
        t_10serv = 1;          // flag it
        t_10++;                // and one more 0.1 down
        if (t_10 > 9) {        // if it's (roughly) a second
            t_10 = 0;            // start the tenths counter over
        t_1++;               // one more second down the drain
          if (t_1 > 9999) t_1 = 0;    // rollover
          t_1serv = 1;         // service needed for seconds
        }
      }
  }

// drop all backplanes back to V/2 (3 of them already will be)
  dirb = DDRB & 0xc5;    // 4 backplanes to inputs
  prtb = PORTB & 0xc5;   // without pullups
  switch (scnt++) {
    case 0:              // send ~segs[0] to B5
        v = ~segs[0];
      dirb |= 0x20;
        prtb |= 0x20;      // output = 1
        break;
      case 1:              // send ~segs[1] to B4
        v = ~segs[1];
      dirb |= 0x10;
        prtb |= 0x10;      // output = 1
        break;
      case 2:              // send ~segs[2] to B3
        v = ~segs[2];
      dirb |= 0x08;
        prtb |= 0x08;      // output = 1
        break;
      case 3:              // send ~segs[3] to B1
        v = ~segs[3];
      dirb |= 0x02;
        prtb |= 0x02;      // output = 1
        break;
      case 4:              // send segs[0] to B5
        v = segs[0];
        dirb |= 0x20;      // B5, output = 0
        break;
      case 5:              // send segs[1] to B4
        v = segs[1];
        dirb |= 0x10;      // B4, output = 0
        break;
      case 6:              // send segs[2] to B3
        v = segs[2];
        dirb |= 0x08;      // B3, output = 0
        break;
      default:             // send segs[3] to B1
        v = segs[3];
        dirb |= 0x02;      // B1, output = 0
  }
  DDRB = dirb;           // 3 backplanes to input, 1 to output
  PORTB = prtb;          // its output is 0 or 1

// if the thing hadn't been rewired, v would go directly to port D.
// however, D0 and D1 have moved (they're the USART pins).

  PORTD = (PORTD & 0x03) | (v & 0xfc);    // top 6 bits haven't moved
  w = PORTC & 0xd7;                       // drop C3 and C5
  if (v & 0x01) w |= 0x08;                // D0 --> C3
  if (v & 0x02) w |= 0x20;                // D1 --> C5
  PORTC = w;
  scnt &= 0x07;                           // keep it 0 - 7
}

// ----------------------------------------------------------------
// table of numeric segments and friends
//
// tab: "0" in position 0. add 12 for each number beyond 0
//          +2 for position 2
//          +6 for position 1
//          +8 for position 3
  const char tab[] = {
    0x00, 0x00, 0x50, 0x78, 0x00, 0x00,
      0x00, 0x00, 0x0D, 0x07, 0x00, 0x00,  // 0
    0x00, 0x00, 0x00, 0x30, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x06, 0x00, 0x00,  // 1
    0x00, 0x00, 0x30, 0x68, 0x00, 0x00,
    0x00, 0x00, 0x0E, 0x03, 0x00, 0x00,  // 2
    0x00, 0x00, 0x20, 0x78, 0x00, 0x00,
    0x00, 0x00, 0x0A, 0x07, 0x00, 0x00,  // 3
    0x00, 0x00, 0x60, 0x30, 0x00, 0x00,
    0x00, 0x00, 0x03, 0x06, 0x00, 0x00,  // 4
    0x00, 0x00, 0x60, 0x58, 0x00, 0x00,
    0x00, 0x00, 0x0B, 0x05, 0x00, 0x00,  // 5
    0x00, 0x00, 0x70, 0x58, 0x00, 0x00,
    0x00, 0x00, 0x0F, 0x05, 0x00, 0x00,  // 6
    0x00, 0x00, 0x00, 0x70, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x07, 0x00, 0x00,  // 7
    0x00, 0x00, 0x70, 0x78, 0x00, 0x00,
    0x00, 0x00, 0x0F, 0x07, 0x00, 0x00,  // 8
    0x00, 0x00, 0x60, 0x70, 0x00, 0x00,
    0x00, 0x00, 0x03, 0x07, 0x00, 0x00,  // 9
    0x00, 0x00, 0x20, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x02, 0x00, 0x00, 0x00,  // -
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // 11 = blank
    0x00, 0x00, 0x20, 0x48, 0x00, 0x00,
    0x00, 0x00, 0x0a, 0x01, 0x00, 0x00,  // 12 = horiz bars
    0x00, 0x00, 0x50, 0x30, 0x00, 0x00,
    0x00, 0x00, 0x05, 0x06, 0x00, 0x00,  // 13 = vert bars

    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // 14 = available
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00   // 15 = available
  };

// this table masks out the bits for the given position
  const char killer[] = {
    0xff, 0xff, 0x8f, 0x87,    // position 0
    0xff, 0xff, 0xf0, 0xf8,    // position 1
    0x8f, 0x87, 0xff, 0xff,    // position 2
    0xf0, 0xf8, 0xff, 0xff     // position 3
  };

// this table maps individual segments
// order is A0, A1, ..., G3, Bat0, ... , Bat3
// high nibble is byte offset (0-3), low nibble is bit number (0-7)
  const char smap[] = {
    0x36, 0x30, 0x16, 0x10, 0x35, 0x31, 0x15, 0x11,   // A and B
    0x34, 0x32, 0x14, 0x12, 0x33, 0x23, 0x13, 0x03,   // C and D
    0x24, 0x22, 0x04, 0x02, 0x26, 0x20, 0x06, 0x00,   // E and F
    0x25, 0x21, 0x05, 0x01, 0x07, 0x07, 0x07, 0x07    // G and battery
  };
```