Sure Electronics AVRDEM2 Board

Introduction

The AVRDEM2 board from Sure Electronics is an AVR ATmega 16 based 12 MHz development board with a USB interface for communications and power. It includes a 20x2 character LCD (no backlight), four 7 segment LED displays, four LEDs, a piezo buzzer, a temperature sensor, a 512 byte external EEPROM, and two pushbuttons. At the time of this writing it is available on eBay for about $26, shipping (from China) included.

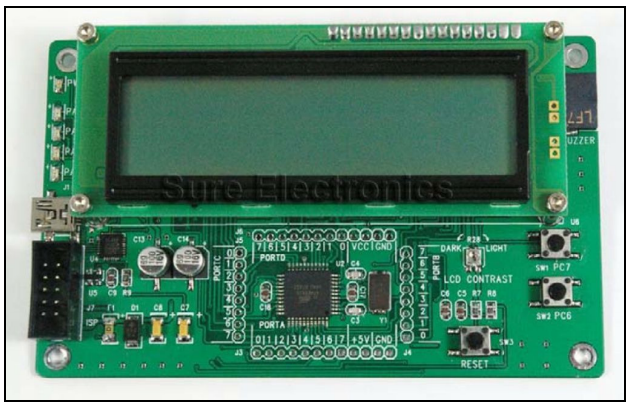

Photo 1: Front view of AVRDEM2 with LCD installed

The AVRDEM comes with a preloaded application which reads the LM75 temperature device (located beneath the LCD) and displays its temperature reading on the LCD as well as sending it via USB to a terminal emulation program such as HyperTerminal. Because the LCD covers the sensor, it is a challenge to warm it by touch as a test, although blowing under the LCD can demonstrate that it is in fact working.

Sample code and a user's manual (including schematics) are available from Sure Electronics. The schematics do not exactly match the version of the board I received, although most of the information appears to be accurate. The board layout has also changed somewhat from the photos included here (taken from the Sure documentation).

One discouraging design flaw is that the power requirements for the board (derived via USB) are in excess of the USB specification, at least on the initial surge. If you plug the AVRDEM directly into the motherboard of your computer (that is, plug it into a USB socket located on your computer's case), it will probably work fine. If you plug it into an external powered USB hub it may or may not work, depending on the sophisication of the hub. If it doesn't (and reports an excessive hub surge current), try plugging it in somewhere else. I have some USB sockets that can support it, and some that can't.

We had a lively discussion on an AVRfreaks forum about this, and the consensus was that the board is out of spec and works where adherence to the USB power specifications are relaxed. Suggestions included downsizing some large capacitors on the power line, adding a small series resistor in the power line to limit current, removing the power from the USB line and providing a separate supply, and simply finding a USB socket that works. I have chosen the latter method, although it's less than optimal.

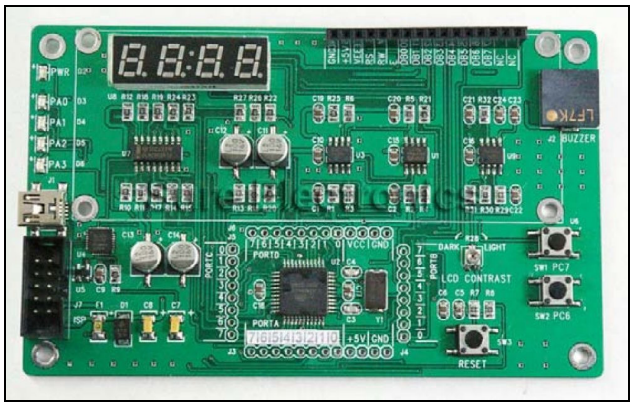

Photo 2: Front view with LCD removed

Another unfortunate shortcoming is the quality of the example code that is available from Sure (and only as a pdf file). As a consequence I have developed accompanying code which allows easy access to every feature of the board. You are free to use this document and its code in any way you see fit, although the demon dogs of hell will pursue you through eternity if you don't properly credit its author (that would be me).

ISP Programming

Programming the ATmega 16 is achieved through ISP using a 10 wire cable. If you have an STK500 this is one of its standard ISP interfaces, and the STK500 is capable of supplying the necessary power to the board. Countless other ISP programming alternatives exist, although with the STK500 and AVR Studio it is trivial. However, there's a small problem with programming the board: **the LCD must be removed to use ISP**. It uses Port B, home of MISO/MOSI/SCK, the ISP pins. The LCD can be replaced once the flash programming has completed (be sure to power down the board first, and double check that the pins socketed correctly).

So, design flaw number two – the LCD daughter board messes up the ability of ISP to reliably function, at least most of the time. It's not that hard to unplug the LCD for programming, but it is yet another irritant, and yes, you will forget to do so and the programming will fail. Usually that means it's time to give up for the night and go to bed.

And that first time you're going to reprogram it? Dump the application program hex file out of the device before you erase it, in case you want to restore it later for some reason. Sure Electronics does not seem to make the hex file available, or at least I couldn't locate it. It's not rocket science, but if you save the hex file first you won't need to rewrite it.

Hardware Configuration

Here is the hardware layout taken from the schematics via my code:

```
// hardware configuration:
//
// Port A:
//   bit 0 - standalone LED (bottom)     0 = on
//   bit 1 - standalone LED (third)      0 = on
//   bit 2 - standalone LED (second)     0 = on
//   bit 3 - standalone LED (top)        0 = on
//   bit 4 - thousands 7 seg LED select  1 = on
//   bit 5 - hundreds 7 seg LED select   1 = on
//   bit 6 - tens 7 seg LED select       1 = on
//   bit 7 - ones 7 seg LED select       1 = on
//
// Port B: LCD data lines
//         LED segments (1 = on)
//           -        bit 0
//          | |       bits 5, 1
//           -        bit 6
//          | |       bits 4, 2
//           -  X     bits 3, 7
//
// Port C:
//   bit 0 - SCL, temp and EEPROM (with ext pullup)
//   bit 1 - SDA, temp and EEPROM (with ext pullup)
//   bit 2 -
//   bit 3 -
//   bit 4 -
//   bit 5 -
//   bit 6 - switch 2 (1 if open, 0 if closed)
//   bit 7 - switch 1 (1 if open, 0 if closed)
//
// Port D:
//   bit 0 - RxD
//   bit 1 - TxD
//   bit 2 - LCD - RS
//   bit 3 - LCD - R/W
//   bit 4 - LCD - E
//   bit 5 - LCD contrast
//   bit 6 -
//   bit 7 - speaker (OC2)
```

Notice that Port B is shared by the 7 segment devices and the LCD. Since the LCD physically covers the 7 segment display it is unlikely you would use both simultaneously, but with some mounting modifications you may want to. Consequently the code allows this, playing some time sharing tricks on this port if necessary.

The TWI interface (C0 and C1) goes to both the LM75 temperature sensor and the external 2404 EEPROM. The piezo element connects to timer 2's OC2 output and can be driven directly by timer 2 via PWM. The USART's RxD and TxD pins go through the USB interface and are available as a virtual com device to the PC. LCD contrast can be adjusted via software or manually with a mounted potentiometer.

The Sample Code

The code is offered as-is, with no warranties or any kind, expressed or implied. I have made a good faith effort to write error free code, but we all know how those good intentions go. Use it at your own risk.

The sample code is written in ImageCraft C, primarily because that's the compiler I own. You may download a free 45 day demo version of the compiler, after which it reverts to a size limited version. It should be fairly easy to port the code to some other compiler.

One thing about ICC – all *char* variables are unsigned. I should have defined a *uchar* type, but I didn't and I'm too lazy to go do it now. Just remember to consider whether you'll introduce problems in your ported code if your compiler supports signed characters.

I split all the features of the board into separate source files, each with its own header. There is also a mainline file which includes the main() routine, interrupt handlers, and some odds and ends. I didn't create a library (this is sample code, after all), so here's the way you proceed:

- Make the mainline code (*avrdem.c*) the primary file in your project
- Add the virtual timer source file (*ad_vtimer.c*) to the project
- Decide which of the optional features you're going to use
- Add the source file for each feature to your project
- Reference the include files for those features in the mainline
- Put your code into main(), along with support functions, etc.
- Instantly get rich, better looking, and popular as a result

The presence of the header file in the mainline code will automatically cause the feature's initialization code to be executed, and may (depending on the feature) alter one or more parts of the interrupt handler(s). That's what all those conditional preprocessor directives are doing – when a feature is included, its support code gets compiled, but otherwise it is ignored.

Again, from the code, a list of the support files:

```
// support files (.c and .h)
//
//   ad_usart  - USART support (original default is 19.2)
//   ad_7seg   - 7 segment LED support
//   ad_switch - switch support
//   ad_leds   - standalone LED support
//   ad_lcd    - LCD support
//   ad_spkr   - speaker support
//   ad_temp   - temperature support
//   ad_EEPROM - standalone EEPROM support
//   ad_twi    - TWI support (needed for temp and EEPROM -
//               the .h file is automatically included when
//               needed. the .c file has to be in the project)
```

As noted, the TWI header file is automatically included if you use the EEPROM or temperature header files, but you will need to include the TWI source file (*ad_twi.c*) in your project in those cases.

For each of the following features, refer to the appropriate source and header files for additional information, function definitions, and how to use the code.

Minimum Code

A bare bones project using this code will be something along the lines of the following. This is essentially the *avrdem.c* source file, whichyou then modify to fit your needs.

```
#include "avrdem.h"             // mainline include file
<include files for any features you use>

char i10x;                      // virtual timer interrupt counter
<your global variable declarations>

// ------------------------------------------------------------
// mainline

void main(void) {
  <your local variable declarations>

  init();                       // various initializations
  SEI();                        // start the timer interrupts

  <any "one time" code you want>

  while (1) {
    <any "infinitely executing" code you want>
  }
}

<the rest of avrdem.c follows (interrupt handlers, etc.)>
```

Then add the virtual timer support source file (*ad_vtimers.c*) and the source files for any of the other features you will be using (and have inserted #include statements for their header files).

Timers

The code uses the two ATmega 16 eight bit timers, Timer 0 and Timer 2. The sixteen bit timer (Timer 1) is available for your use.

Timer 0 is used to generate an interrupt approximately every millisecond, and most of the service routines for the various features are executed within this interrupt handler. Virtual timer management, switch debouncing, LED and 7 segment display multiplexing, and sound generation control are examples.

Timer 2 is used to generate PWM frequencies to drive the speaker. There are 6 octaves of alleged musical notes defined, and while the quality isn't good, it's passable for a piezo device. This code does not include volume control since the PWM waveform is fixed at a 50% duty cycle. Volume control is left as an exercise for the student.

The code includes built-in support for (up to 10, initially set at 5) virtual timers. These have a resolution of about 0.01 seconds over a range of about 10 minutes. Your code can start one of the timers, then a flag will be set when that timer runs out. This can be done on a once only basis, or you can let the timer run indefinitely, setting the flag each time the interval expires. You reset the flag when you acknowledge the timer run out, as shown below.

Code to flash an LED every second:

```
#include "avrdem.h"             // mainline include file
#include "ad_leds.h"            // standalone LED support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline
```

```
void main(void) {
  char flag1;                   // virtual timer run out flag

  init();                       // various initializations
  SEI();                        // start the timer interrupts

  set_vtime(1, 100, V_NEXT | V_GO, &flag1);  // 1 second timer

  while (1) {
    while (!flag1) ;            // wait for timer run out
    flag1 = 0;                  // reset flag
    leds(1, LED_TOG);           // toggle LED #1
  }
}
```

This project would consist of the code shown above, the *ad_leds.c* source file, and the *ad_vtimer.c* source file. The above code would become part of the mainline source file (*avrdem.c*) which would also include the init() function and the timer0 interrupt handler, among other things.

The include files are for the main code (which references the virtual timer header file) and the LED support. The presence of these include files causes appropriate initialization code to be run (as a part of the *init()* function). They also cause the timer0 interrupt handler to be expanded to support LED multiplexing.

In the program *flag1* is a variable that is set to 0 by the call to *set_vtime()*. When the virtual timer (timer #1 in this case) elapses (after 100 units of 0.01 seconds, or 1 second), then *flag1* is set to 1. The third argument of *set_vtime()* specifies that the timer is to start immediately and run forever.

Within the infinite loop typical of embedded programs, when the program sees *flag1* go nonzero, it resets *flag1* back to 0 and toggles the LED. Thus the LED will flash every second until the cows come home for dinner.

USART

USART support is minimal, partly because it's so easy to write and partly because everyone seems to want something slightly different. The routines included here are polled (both transmit and receive), and the receiver returns "no character ready" rather than waiting as it does in some implementations. Interrupt driven serial routines require buffering that eats up SRAM which is already in short supply.

The ICC libraries require a user written *getchar()* and *putchar()* since each hardware platform varies on where the Rx/Tx pins are and what all the USART register names are. In addition, I've written functions to send a CR/LF pair, a string from SRAM, and a string from flash. Additional USART support is easy to implement, so I left it at these basic routines.

The ICC library routines support a text mode indicated by the *_textmode* variable. If *_textmode* is nonzero, newline characters are replaced with CR/LF pairs in *putchar()*.

Seven Segment Display

The 7 segment display consists of 4 digits cleverly hidden beneath the LCD display. Support routines include the ability to write a number (signed integer, -999 to 9999, with or without decimal point(s) and/or zero fill), hexadecimal digits, and raw segments.

The display itself is multiplexed in 8 passes (handled by the timer0 interrupt handler), although this does not affect the casual user.

The following example counts from -999 to 9999 (then displays an error until it wraps around to -999), incrementing every 1/10th of a second.

```
#include "avrdem.h"
#include "ad_7seg.h"            // 7 segment LED support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline

void main(void) {
  char flag1;
  int cntr = -999;              // starting count

  init();                       // all initialization
  SEI();                        // run the timer interrupts

  set_vtime(1, 10, V_NEXT | V_GO, &flag1);

  while (1) {
    if (flag1) {                // tenth second pass?
      flag1 = 0;                // yup, we saw it
      seg7_num(cntr++, 0);      // display & incr our number
    }
  }
}
```

Pushbutton Switches

The AVRDEM board has two user pushbuttons (and a reset pushbutton), and they are identified in the code as 1 and 2, corresponding to the board markings of *sw1* and *sw2*.

There are no support functions other than the automatically called initialization to be concerned with. Rather, there are four volatile *char* variables named *sw1*, *sw2*, *sw1_chg*, and *sw2_chg* which you use to access the state of the switches.

The *sw1* and *sw2* variables show the current state of the two switches. If the variable is equal to *SW_OPEN*, then the switch is open and debounced. If the variable is equal to *SW_CLOSED*, then the switch is closed and debounced. Any other value indicates that the switch is in transition and should not be considered reliable.

The *sw1_chg* variable (and *sw2_chg* for switch #2) is set nonzero when the switch changes state and is stable. The transition from closed to open sets the variable to *SW_C2O*, and the transition from open to closed sets the variable to *SW_O2C*. The variable will contain the last recognized transition, unless the user wishes to zero it when the transition is acknowledged (as shown in the example code below).

Debouncing is achieved by 8 consecutive reads of the switch on every other 1ms interrupt, giving a debounce interval of 16ms. This is probably longer than necessary.

In this example code, LED #1 (the one closest to the top edge of the board) follows switch 1 – on when the switch is pressed, and off when it is released. LED #2 toggles each time the switch is released (goes from closed to open).

```
#include "avrdem.h"
#include "ad_switch.h"          // switch support
#include "ad_leds.h"            // standalone LED support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline

void main(void) {
  init();                       // various initializations
  SEI();                        // start the timer interrupts

  while (1) {
    if (sw1 == SW_CLOSED) leds(1, LED_ON); // turn on if closed
    if (sw1 == SW_OPEN) leds(1, LED_OFF);  // turn off if open

    if (sw1_chg == SW_C2O) {    // did we go from closed to open?
      sw1_chg = 0;              // only do this once per change
      leds(2, LED_TOG);         // toggle LED 2
    }
  }
}
```

LEDs

There are 4 standalone LEDs on the AVRDEM, and I have numbered them 1 to 4 (from the top of the board). Functions are supplied to read the current state and set an LED on, off, or toggle it. See the example above (under *Pushbutton Switches*) for a typical use.

The actual lighting of the LEDs is multiplexed within the timer0 interrupt handler, although this is tranparent to the user.

LCD

The LCD is a standard HD44780 compatible device. The Hitachi datasheet is sort of the gold standard for a description of its operation. The AVRDEM implementation does not include a backlight.

The display has two lines of 20 columns, and the contrast can be altered with an on-board potentiometer and via software. Standard LCD functionality is included in the code: clear, home, write to location, read and write both character and data memory, etc. The initialization leaves the display and cursor turned off, so be sure to turn at least the display on before you panic and decide your LCD doesn't work.

Also note that because the code expects a response from the HD44780 controller, **this code will hang if the LCD is not plugged in**. Beware.

The display RAM (80 bytes, addresses 0x00 – 0x4F) and graphics RAM (64 bytes, addresses 0x00 – 0x3F) can be used as normal volatile memory if desired. Forty bytes of the display RAM are mapped to the display itself (addresses 0x00 – 0x13 for the top line, and 0x40 – 0x3B for the second line).

The graphics RAM is used to define special characters 0 – 7 which may then be displayed like the built-in character set (once defined they can be referred to as either 0 – 7 or 9 – 15). This is done by passing a 7 byte pattern to a function, where the low 5 bits of each byte are mapped to a pixel of the special character. The bottom line, line 8, is used as the cursor position and is not included in the pattern.

Because the 7 segment displays and the LCD share a port, it is necessary to keep them playing together nicely. This is achieved by shutting down the 7 segment display (and disabling interrupts) during those microseconds that the LCD is being accessed. The effect should be invisible to the casual observer.

Here's an example of using the LCD and 7 segment display together. Again, this won't work unless the LCD is plugged in (even the seven segment display part won't work).

```
#include "avrdem.h"
#include "ad_7seg.h"            // 7 segment LED support
#include "ad_lcd.h"             // LCD support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline

void main(void) {
  char flag1, buf[7];
  int cnt = 0;

  init();
  SEI();

  lcd_display(L_DON, L_COFF, L_BOFF);
  set_vtime(1, 100, V_NEXT | V_GO, &flag1);

  while (1) {
    while (!flag1) ;            // wait for timer runout
    flag1 = 0;                  // reset the timer flag

    seg7_num(cnt, 0);           // write to the 7 seg display
    sprintf(buf, "%d", cnt);    // num --> ascii string
    lcd_str(0, 0, buf);         // write to the LCD
    cnt++;                      // bump the counter
  }
}
```

The functions pretty much follow the HD44780 command set in structure and capability. Refer to the LCD datasheet for more details on addressing, custom character generation, memory usage, display and cursor shifting, and the like.

Speaker (Piezo Buzzer)

The speaker is driven by the PWM output of timer 2. The support functions are written with standard musical sounds in mind, and routines to play (and stop playing) a song stored in flash memory are included.

There are #defines for 6 octaves of notes, and a song is stored as a sequence of notes. Each song includes a repeat count as its first value, and may contain embedded indications of rests, changes in tempo and duration, and indications of eighth, quarter, half, and whole notes. Two songs are included in the source code – a run up of all the defined notes, and a little Beethoven "borrowed" from the AVR Butterfly source code.

The address of the note sequence is passed to a function to start playing, and a variable is altered to indicate when it has completed. No intervention from the user is required once the song starts, with the possible exception of leaving the room.

Example highbrow code:

```
#include "avrdem.h"
#include "ad_spkr.h"            // speaker support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline

void main(void) {
  init();                       // various initializations
  SEI();                        // start the timer interrupts

  start_song(elise);            // start playing
  while (1) ;                   // wait for Godot
}
```

Temperature (LM75)

The LM75 temperature sensor has features which are not supported either by the AVRDEM hardware or this software. My code treats it only as a Celcius thermometer returning readings in 0.5 degrees units. Have a look at the LM75 datasheet to see the chip's additional capabilities.

The two user callable functions are *read_temp()* and *celcius()*. The first starts a read which results in raw data being stored into a two character buffer. The second converts the data in the buffer into an integral Celcius reading in units of ½ degree. The Fahrenheit conversion requires floating point math, an option many user would not want to include in the example code due to the large overhead such support requires.

The LM75 is an I^2C device (called TWI in the AVR universe), and as such requires the TWI support described below. Refer to it for more details on how TWI is implemented.

External EEPROM

This EEPROM is a 512 byte chip external to the ATmega 16 (which contains 512 bytes of internal EEPROM). Like the LM75, the external EEPROM is an I^2C device, so read the TWI section below for additional information.

The memory is organized as 32 pages of 16 bytes. This makes no difference for reads, but for certain types of multiple byte writes all the data must be contained in the same page. While I had planned to include a multi-byte write routine to store across page boundaries, I decided it wasn't really needed in this version of the code. The primary reason again is the limitation of the SRAM size; large buffers which lend themselves to big writes to EEPROM simply cannot be maintained on a memory limited device.

Otherwise the access functions are pretty much as expected and are explained in comments in the code. The only gotcha is the delayed culmination of the I/O due to the interrupt driven TWI implementation; see below.

TWI

TWI support is required if you use the LM75 temperature sensor and/or the external EEPROM. The include files for both of those will also load the TWI header file, but the TWI source file (*ad_twi.c*) will have to be manually added to the project.

As an interesting activity I decided to make the TWI support interrupt driven. Many applications simply poll (and wait) during each step of the TWI process. Doing it this way is far simpler, and at the end of the process the activity (write the EEPROM, read the temperature, etc.) is complete, but several milliseconds may have been wasted in the process. The interrupt driven approach frees up the processor, although it does require being notified that the TWI interaction has finished.

For this implementation there is a simple scripting language (essentially a state machine) that walks the TWI interactions through their steps. Included in the code are scripts to read the LM75 and various types of reads and writes to the EEPROM. The user makes a function call (such as to write a byte to EEPROM) which starts a script that eventually results in either success or failure. The user polls a variable (*twi_sts*) to find out what happens. Possible values are:

TWI_DIS – twi hardware disabled, no script
TWI_OK – hardware enabled, no script running
TWI_NRDY – like TWI_OK, but EEPROM returned a busy status
TWI_RUN – script running
TWI_ERR – script error; hardware not enabled (user must set twi_sts = TWI_DIS when the error is recognized)

In the simplest case the user starts a script and simply polls *twi_sts* until the process completes. More complicated and useful scenarios are obviously possible.

Here's some code to read the raw temperature data and display it on the 7 segment display in hex. See the LM75 data sheet to interpret the raw data.

```
#include "avrdem.h"
#include "ad_7seg.h"            // 7 segment LED support
#include "ad_temp.h"            // temperature support

char i10x;                      // virtual timer interrupt counter

// ------------------------------------------------------------
// mainline

void main(void) {
  char flag1, strt = 0;

  init();
  SEI();

  set_vtime(1, 10, V_NEXT | V_GO, &flag1);

  while (1) {
    if (flag1) {                // one second elapse?
      flag1 = 0;                // we saw the flag; reset
      strt = 1;                 // we're starting TWI script
      if (read_temp()) {
        seg7_hex1(14, 3);       // error ("E") in pos 3, blanked
        while (1) ;             // die
      }
    }

    // wait for the read script to finish
```

```
      if (strt) {                    // are we running a script?
        if ((twi_sts == TWI_OK) || (twi_sts == TWI_NRDY)) {
          seg7_hex2(lm75[0], 0);     // display raw readings
          seg7_hex2(lm75[1], 1);     // display raw readings
          strt = 0;                  // all done until next time
        }
      }
    }
  }
```

You've seen all of this before, with the exception of watching the TWI flag for an indication of completion. In this example we only check *twi_sts* when we've started a script, although there are countless other ways to have done this.

Summary

This code has been lightly tested, but not exhaustively. You may well discover some "interesting" anomolies, and if so I'd like to hear about them. I am not likely to do maintenance on this code, although I'm always happy to accept suggestions and/or corrections. I can be contacted through private message at AVRfreaks.net (user name zbaird). If you are not familiar with AVRfreaks, it is an invaluable resource for anyone doing AVR development work.

Because of the length of the combined source listing I have not included it in this document. The actual individual source files are included in the accompanying zip file.

If your flash space is tight you can certainly remove the functions you are not using in any of the support source code files. In my experience flash size is not typically the limiting factor, at least not in a development and experimenting environment.

Good luck with your experiments with your AVRDEM board!