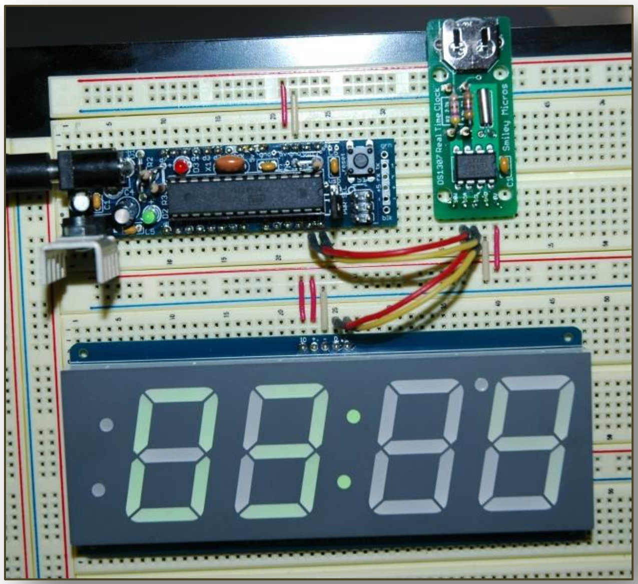


Add a Seven-Segment LED Display to your AVR microcontroller

Bruce E. Hall, W8BH

1) INTRODUCTION

Character-based LCD modules are the most popular display units for microcontrollers. So why would you ever want to use an old-fashioned seven segment display? They can't do letters. They can't do punctuation. And it takes a lot of current to drive them. Here's why:

- Seven-segment displays do numbers *very well*.
- The displays are bright, high-contrast units that can be read at distance.
- Seven-segment digits can be much larger than typical LCD characters.
- They look cool.

If you would like to add a seven-segment display to your AVR micro, read on. I will describe a series of routines that can be used with Adafruit I2C backpack displays. The displays come in digit heights of 0.56" or 1.2". I wanted a BIG display, so I chose the 1.2" display. These four digit units have a colon (:) between the middle digits, making them especially well-suited for clock displays. If you'd like to make a simple clock with an LED display, read more.

2) THE I^{2}C INTERFACE

Atmel calls their version of I2C the "two-wire" interface, or TWI. It is a serial-data protocol which uses two data lines for communication: a data line (SDA) and a clock (SCL). Please see my DS1307 article, which includes all the I2C routines we will need to communicate with our display.

3) THE HT16K33 CONTROLLER

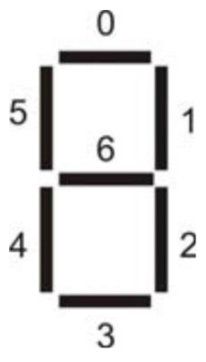
Four digit displays typically have a 12 pin interface: eight pins for the segments +/- decimal point, and one common pin for each digit. That's a lot of I/O lines! Our microcontroller has enough pins, but there would be few left for other functions. Further, our micro would need to multiplex this display, sequentially driving the digits at a rate faster than human vision can process. See my smart necklace article for an example of LED multiplexing.

Adding a dedicated controller to our display reduces the number of I/O lines and handles the display multiplexing for us. All we need are two spare data lines, and the ability to send a few commands. The controller does the rest.

These display modules use the HT16K33 controller. This chip is a versatile device, capable of driving up to eight 16-segment digits and reading a keyboard matrix of 39 keys *at the same time*. When driving this four-digit display, we are using only a part of its capabilities. Data is transferred to/from the device in I2C format, up to a 400 kHz clock rate.

Display Address	Digit (function)
0x00 – 0x01	0 (10 hour digit)
0x02 – 0x03	1 (1 hour digit)
0x04 – 0x05	2 (":" + dp's)
0x06 – 0x07	3 (10 min. digit)
0x08 – 0x09	4 (1 min digit)

The HT16K33 controller contains 16 bytes of display memory. The first two bytes represent the 16 segments of the first digit, the next two bytes represent the second digit, etc. Our display uses 7 of the 16 segments, and 5 of the 8 digits. On a clock display, digits 0-1 are hours, digit 2 is ":", and digits 3-4 are minutes.



As above, the 16 available segments map to two bytes of display memory. The seven segments that we use map to the first byte, and the second byte is not used. Bit 0 maps to the top display segment, usually called segment 'a'. Following around the display clockwise are bits 1, 2, 3, 4, and 5. Bit 6 represents the middle segment 'g'. On my display bit 7 is not used (but could be decimal point on other displays).

So, how do we make a '3'? We need to light up segments 0, 1, 2, 3, and 6:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	1	0	0	1	1	1	1

The corresponding binary value would be 0100.1111 or 0x4F. Here is a table for all 16 hexadecimal digits, and the values needed to turn on the proper segments.

Digit	Hex Value	g	f	e	d	c	b	a
0	0×3F	0	1	1	1	1	1	1
1	0×06	0	0	0	0	1	1	0
2	0×5B	1	0	1	1	0	1	1
3	0×4F	1	0	0	1	1	1	1
4	0×66	1	1	0	0	1	1	0
5	0×6D	1	1	0	1	1	0	1
6	0×7D	1	1	1	1	1	0	1
7	0×07	0	0	0	0	1	1	1
8	0×7F	1	1	1	1	1	1	1
9	0×6F	1	1	0	1	1	1	1
A	0×77	1	1	1	0	1	1	1
b	0×7C	1	1	1	1	1	0	0
C	0×39	0	1	1	1	0	0	1
d	0×5E	1	0	1	1	1	1	0
E	0×79	1	1	1	1	0	0	1
F	0×71	1	1	1	0	0	0	1

4) CODING:

The controller is initialized by sending three commands: turning on its internal oscillator, enabling display output, and setting the brightness level. Brightness is adjusted from 0 (dimmest) to 15 (brightest). See my article on the DS1307 real-time clock for information on I2C communication.

```
#define HT16K33             0xE0        // I2C bus address for Ht16K33 backpack
#define HT16K33_ON          0x21        // turn device oscillator on
#define HT16K33_STANDBY     0x20        // turn device oscillator off
#define HT16K33_DISPLAYON   0x81        // turn on output pins
#define HT16K33_DISPLAYOFF  0x80        // turn off output pins
#define HT16K33_BLINKON     0x85        // blink rate 1 Hz (-2 for 2 Hz)
#define HT16K33_BLINKOFF    0x81        // same as display on
#define HT16K33_DIM         0xE0        // add level (15=max) to byte

void SS_Init()
{
    I2C_WriteByte(HT16K33,HT16K33_ON);          // turn on device oscillator
    I2C_WriteByte(HT16K33,HT16K33_DISPLAYON);   // turn on display, no blink
    I2C_WriteByte(HT16K33,HT16K33_DIM + 15);    // set max brightness
}
```

Lighting segments on each digit is just a matter of writing the data to the appropriate display address. The colon “:” is a special case: bit1 at address 0x04. I prefix all of my seven-segment routines with “SS_”.

```
void SS_SetDigitRaw(byte digit, byte data)      // digits (L-to-R) are 0,1,2,3
// Send segment-data to specified digit (0-3) on LED display
{
    if (digit>4) return;                        // only digits 0-4
    if (digit>1) digit++;                       // skip over colon @ position 2
    digit <<= 1;                                // multiply by 2
    I2C_WriteRegister(HT16K33,digit,data);      // send segment-data to display
}

void SS_BlankDigit(byte digit)
// Blanks out specified digit (0-3) on LED display
{
    SS_SetDigitRaw(digit,0x00);                 // turn off all segments on specified digit
}

void SS_SetColon(byte data)                     // 0=off, 1=on
//  the colon is represented by bit1 at address 0x04.  There are three other single LED
//  "decimal points" on the display, which are at the following bit positions
//  bit2=top left, bit3=bottom left, bit4=top right
{
    I2C_WriteRegister(HT16K33,0x04,data<<1);
}
```

Usually we want to display numbers, not segment patterns. The (green) conversion table above is coded as an array of bytes.

```
static const byte numberTable[] =       // convert number to lit-segments
{
    0x3F,  // 0
    0x06,  // 1
    0x5B,  // 2
    0x4F,  // 3
    0x66,  // 4
    0x6D,  // 5
    0x7D,  // 6
    0x07,  // 7
    0x7F,  // 8
    0x6F,  // 9
    0x77,  // A
    0x7C,  // b
    0x39,  // C
    0x5E,  // d
    0x79,  // E
    0x71   // F   };


void SS_SetDigit(byte digit, byte data)
// display data value (0-F) on specified digit (0-3) of LED display
{
    if (data>0x10) return;                      // only values <=16
    SS_SetDigitRaw(digit,numberTable[data]);    // show value on display
}
```

Now that we can display individual digits, it would be helpful to display integers up to 4 digits in size (0-9999). The following routine uses the itoa() routine to get each digit of the number. A simple way to display numbers might be something like this:

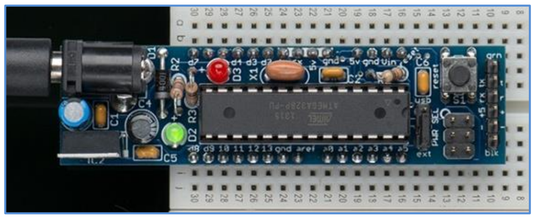
```
void SimpleInteger(int data)
{
    char st[10]="";
    if (data>9999) return;                      // too big??
    itoa(data,st,10);                           // convert to string
    byte len = strlen(st);
    for (byte digit=0; digit<len; digit++)      // for all digits in string
    {
        char ch = st[digit];                    // get char for this digit
        ch-='0';                                // ascii -> numeric value
        SS_SetDigit(digit,ch);                  // display digit
    }
}
```

The itoa() routines give us the ability to choose decimal (base 10), hex (base 16), or even octal (base 8) representation, so let’s add it. I also added right-justification, because I think it looks better for numbers.

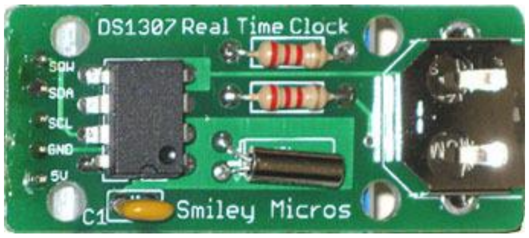
```
void SS_Integer(int data, byte base)
{
    char st[10]="";
    itoa(data,st,base);                         // convert to string
    byte len = strlen(st);
    if (len>4) return;                          // integer too large??
    for (byte digit=0; digit<4; digit++)        // for all 4 digits
    {
        byte blanks = 4-len;                    // number of blanks
        if (digit<blanks)                       // right-justify display
            SS_BlankDigit(digit);               // padding with blanks
        else
        {
            char ch = st[digit-blanks];         // get char for this digit
            if (ch>='a') ch-=87;                // correct for hex digits
            else ch-='0';                       // ascii -> numeric value
            SS_SetDigit(digit,ch);              // display digit
        }
    }
}
```

That’s it for the basic seven-segment routines. The source code includes a few “testing” routines

5) CONSTRUCTION:



1) Instead of breadboarding an ATmega328 directly, I use the DC boarduino by Adafruit: it is breadboard friendly, and puts a DC power supply, microprocessor, external oscillator, ISP programming header, status LED, and reset switch all on a very small circuit board.



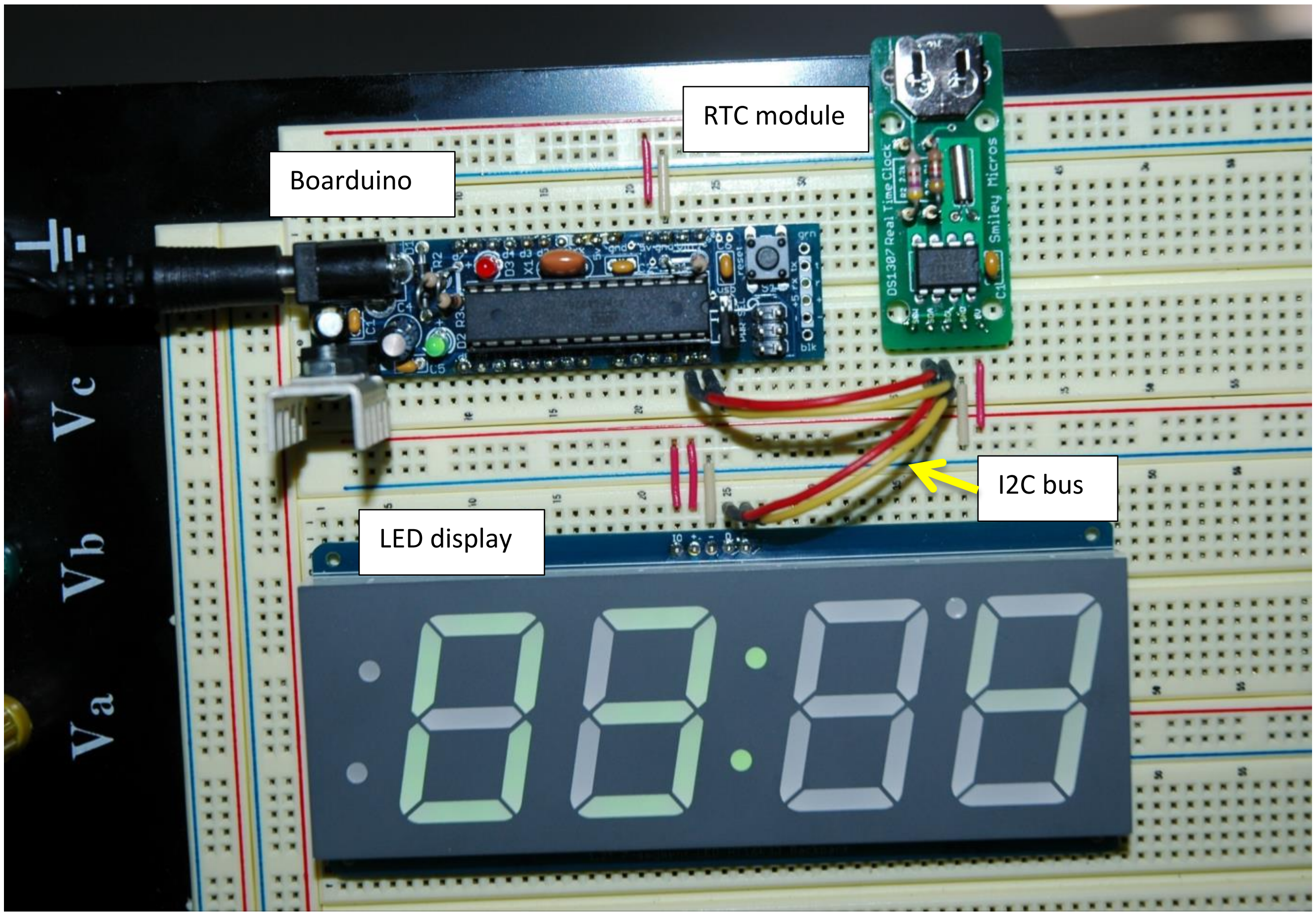
2) Next, you need a DS1307. I used a small circuit module rather than the chip. The module I used is the $15 RTC kit by Smiley Micros. The module adds the required external oscillator and battery backup. Other good ones are available from SparkFun and Adafruit.

3) Get a huge 1.2" seven segment display from Adafruit, in red, green, or yellow. The five pins are labeled IO, +, - , D, and C. The first two pins, IO and "+" go to +5V, and "-" goes to ground. Please remember that this display can require a lot of current! Make sure your DC input can supply at least 500mA at 5VDC. The regulator on my boarduino gets pretty warm, so I added a heat sink.

4) Wire the I^{2}C bus as follows:

DC Boarduino/328	DS1307 module	Seven-Segment module
A4 (PC4)	SDA	"D" (data)
A5 (PC5)	SCL	"C" (clock)

Remember than each data line needs a pull-up resistor. Check your RTC module for these resistors. Mine uses a pair of 2.2K (red/red/red) resistors. If your module does not include these resistors, install them on your breadboard between +5V and SDA/SCL. I also tried a pair of 4.7K resistors and they worked fine.



Here is the breadboard layout. The I^2C bus is represented by the red (SDA) and yellow (SCL) wires. Note DC power connections to the Boarduino, RTC, and LED.

6) SOURCE CODE:

```
//-----------------------------------------------------------------------------
//  ss01:  Experiments with interfacing ATmega328 to an Seven-Segment display
//
//  Author   :  Bruce E. Hall <bhall66@gmail.com>
//  Website  :  w8bh.net
//  Version  :  1.0
//  Date     :  10 Sep 2013
//  Target   :  ATTmega328P microcontroller
//  Language :  C, using AVR studio 6
//  Size     :  1338 bytes, using -O1 optimization
//
//  Fuse settings:  8 MHz osc with 65 ms Delay, SPI enable; *NO* clock/8


//      ---------------------------------------------------------------------------
//      GLOBAL DEFINES

#define F_CPU   16000000L                // run CPU at 16 MHz
#define LED     5                        // Boarduino LED on PB5
#define ClearBit(x,y) x &= ~_BV(y)       // equivalent to cbi(x,y)
#define SetBit(x,y) x |= _BV(y)          // equivalent to sbi(x,y)

//      ---------------------------------------------------------------------------
//      INCLUDES

#include <avr/io.h>                      // deal with port registers
#include <util/delay.h>                  // used for _delay_ms function
#include <string.h>                      // string manipulation routines
#include <stdlib.h>

//      ---------------------------------------------------------------------------
//      TYPEDEFS

typedef uint8_t byte;                    // I just like byte & sbyte better
typedef int8_t sbyte;

//      ---------------------------------------------------------------------------
//      MISC ROUTINES

void InitAVR()
{
    DDRB = 0x3F;                        // 0011.1111; set B0-B5 as outputs
    DDRC = 0x00;                        // 0000.0000; set PORTC as inputs
}

void msDelay(int delay)                 // put into a routine
{                                       // to remove code inlining
    for (int i=0;i<delay;i++)           // at cost of timing accuracy
    _delay_ms(1);
}

void FlashLED()
{
    SetBit(PORTB,LED);
    msDelay(250);
    ClearBit(PORTB,LED);
    msDelay(250);
}


//      ---------------------------------------------------------------------------
//      I2C (TWI) ROUTINES
//
//  On the AVRmega series, PA4 is the data line (SDA) and PA5 is the clock (SCL
//  The standard clock rate is 100 KHz, and set by I2C_Init.  It depends on the AVR osc. freq.
```

```
#define F_SCL         100000L           // I2C clock speed 100 KHz
#define READ          1
#define TW_START      0xA4              // send start condition (TWINT,TWSTA,TWEN)
#define TW_STOP       0x94              // send stop condition (TWINT,TWSTO,TWEN)
#define TW_ACK        0xC4              // return ACK to slave
#define TW_NACK       0x84              // don't return ACK to slave
#define TW_SEND       0x84              // send data (TWINT,TWEN)
#define TW_READY     (TWCR & 0x80)      // ready when TWINT returns to logic 1.
#define TW_STATUS    (TWSR & 0xF8)      // returns value of status register
#define I2C_Stop()    TWCR = TW_STOP    // inline macro for stop condition

void I2C_Init()
// at 16 MHz, the SCL frequency will be 16/(16+2(TWBR)), assuming prescalar of 0.
// so for 100KHz SCL, TWBR = ((F_CPU/F_SCL)-16)/2 = ((16/0.1)-16)/2 = 144/2 = 72.
{
    TWSR = 0;                           // set prescalar to zero
    TWBR = ((F_CPU/F_SCL)-16)/2;        // set SCL frequency in TWI bit register
}

byte I2C_Detect(byte addr)
//  look for device at specified address; return 1=found, 0=not found
{
    TWCR = TW_START;                     // send start condition
    while (!TW_READY);                   // wait
    TWDR = addr;                         // load device's bus address
    TWCR = TW_SEND;                      // and send it
    while (!TW_READY);                   // wait
    return (TW_STATUS==0x18);            // return 1 if found; 0 otherwise
}

byte I2C_FindDevice(byte start)
// returns with address of first device found; 0=not found
{
    for (byte addr=start;addr<0xFF;addr++)  // search all 256 addresses
    {
        if (I2C_Detect(addr))           // I2C detected?
            return addr;                // leave as soon as one is found
    }
    return 0;                           // none detected, so return 0.
}

void I2C_Start (byte slaveAddr)
{
    I2C_Detect(slaveAddr);
}

byte I2C_Write (byte data)              // sends a data byte to slave
{
    TWDR = data;                        // load data to be sent
    TWCR = TW_SEND;                     // and send it
    while (!TW_READY);                  // wait
    return (TW_STATUS!=0x28);
}

byte I2C_ReadACK ()                     // reads a data byte from slave
{
    TWCR = TW_ACK;                      // ack = will read more data
    while (!TW_READY);                  // wait
    return TWDR;
    //return (TW_STATUS!=0x28);
}

byte I2C_ReadNACK ()                    // reads a data byte from slave
{
    TWCR = TW_NACK;                     // nack = not reading more data
    while (!TW_READY);                  // wait
    return TWDR;
    //return (TW_STATUS!=0x28);
}
```

```c
void I2C_WriteByte(byte busAddr, byte data)
{
    I2C_Start(busAddr);                 // send bus address
    I2C_Write(data);                    // then send the data byte
    I2C_Stop();
}

void I2C_WriteRegister(byte busAddr, byte deviceRegister, byte data)
{
    I2C_Start(busAddr);                 // send bus address
    I2C_Write(deviceRegister);          // first byte = device register address
    I2C_Write(data);                    // second byte = data for device register
    I2C_Stop();
}

byte I2C_ReadRegister(byte busAddr, byte deviceRegister)
{
    byte data = 0;
    I2C_Start(busAddr);                 // send device address
    I2C_Write(deviceRegister);          // set register pointer
    I2C_Start(busAddr+READ);            // restart as a read operation
    data = I2C_ReadNACK();              // read the register data
    I2C_Stop();                         // stop
    return data;
}


//      ---------------------------------------------------------------------------
//      DS1307 RTC ROUTINES

#define DS1307              0xD0             // I2C bus address of DS1307 RTC
#define SECONDS_REGISTER    0x00
#define MINUTES_REGISTER    0x01
#define HOURS_REGISTER      0x02
#define DAYOFWK_REGISTER    0x03
#define DAYS_REGISTER       0x04
#define MONTHS_REGISTER     0x05
#define YEARS_REGISTER      0x06
#define CONTROL_REGISTER    0x07
#define RAM_BEGIN           0x08
#define RAM_END             0x3F

void DS1307_GetTime(byte *hours, byte *minutes, byte *seconds)
// returns hours, minutes, and seconds in BCD format
{
    *hours = I2C_ReadRegister(DS1307,HOURS_REGISTER);
    *minutes = I2C_ReadRegister(DS1307,MINUTES_REGISTER);
    *seconds = I2C_ReadRegister(DS1307,SECONDS_REGISTER);
    if (*hours & 0x40)                  // 12hr mode:
       *hours &= 0x1F;                  // use bottom 5 bits (pm bit = temp & 0x20)
    else *hours &= 0x3F;                // 24hr mode: use bottom 6 bits
}

void DS1307_GetDate(byte *months, byte *days, byte *years)
// returns months, days, and years in BCD format
{
    *months = I2C_ReadRegister(DS1307,MONTHS_REGISTER);
    *days = I2C_ReadRegister(DS1307,DAYS_REGISTER);
    *years = I2C_ReadRegister(DS1307,YEARS_REGISTER);
}

void SetTimeDate()
// simple, hard-coded way to set the date.
{
    I2C_WriteRegister(DS1307,MONTHS_REGISTER,  0x08);
    I2C_WriteRegister(DS1307,DAYS_REGISTER,    0x31);
    I2C_WriteRegister(DS1307,YEARS_REGISTER,   0x13);
    I2C_WriteRegister(DS1307,HOURS_REGISTER,   0x08+0x40);  // add 0x40 for PM
    I2C_WriteRegister(DS1307,MINUTES_REGISTER, 0x51);
    I2C_WriteRegister(DS1307,SECONDS_REGISTER, 0x00);
}


//      ---------------------------------------------------------------------------
//      7-SEGMENT BACKPACK (HT16K33) ROUTINES
//
//  The HT16K33 driver contains 16 bytes of display memory, mapped to 16 row x 8 column output
//  Each column can drive an individual 7-segment display; only 0-4 are used for this device.
//  Each row drives a segment of the display; only rows 0-6 are used.
//
//        0               For example, to display the number 7, we need to light up segments
//     -------            0, 1, 2, 3, and 6.  This would be binary 0100.1111 or 0x4F.
//   5 |     | 1
//     |  6  |            Mapping to the display address memory:
//     -------              0x00   Digit 0 (left most digit)
//   4 |     | 2            0x02   Digit 1
//     |  3  |              0x04   colon ":" on bit1
//     -------              0x06   Digit 2
//                          0x08   Digit 4 (right-most digit)
//

#define HT16K33             0xE0        // I2C bus address for Ht16K33 backpack
#define HT16K33_ON          0x21        // turn device oscillator on
#define HT16K33_STANDBY     0x20        // turn device oscillator off
#define HT16K33_DISPLAYON   0x81        // turn on output pins
#define HT16K33_DISPLAYOFF  0x80        // turn off output pins
#define HT16K33_BLINKON     0x85        // blink rate 1 Hz (-2 for 2 Hz)
#define HT16K33_BLINKOFF    0x81        // same as display on
#define HT16K33_DIM         0xE0        // add level (15=max) to byte

static const byte numberTable[] =       // convert number to lit-segments
{
    0x3F,  // 0
    0x06,  // 1
    0x5B,  // 2
    0x4F,  // 3
    0x66,  // 4
    0x6D,  // 5
    0x7D,  // 6
    0x07,  // 7
    0x7F,  // 8
    0x6F,  // 9
    0x77,  // A
    0x7C,  // b
    0x39,  // C
    0x5E,  // d
    0x79,  // E
    0x71,  // F
    0x00,  //<blank>
};

void SS_Init()
{
    I2C_WriteByte(HT16K33,HT16K33_ON);          // turn on device oscillator
    I2C_WriteByte(HT16K33,HT16K33_DISPLAYON);   // turn on display, no blink
    I2C_WriteByte(HT16K33,HT16K33_DIM + 15);    // set max brightness
}


void SS_SetDigitRaw(byte digit, byte data)      // digits (L-to-R) are 0,1,2,3
// Send segment-data to specified digit (0-3) on LED display
{
    if (digit>4) return;                        // only digits 0-4
    if (digit>1) digit++;                       // skip over colon @ position 2
    digit <<= 1;                                // multiply by 2
    I2C_WriteRegister(HT16K33,digit,data);      // send segment-data to display
}
```

```
void SS_BlankDigit(byte digit)
// Blanks out specified digit (0-3) on LED display
{
    SS_SetDigitRaw(digit,0x00);         // turn off all segments on specified digit
}


void SS_SetDigit(byte digit, byte data)
// display data value (0-F) on specified digit (0-3) of LED display
{
    if (data>0x10) return;                      // only values <=16
    SS_SetDigitRaw(digit,numberTable[data]);    // show value on display
}

void SS_SetColon(byte data)                     // 0=off, 1=on
//  the colon is represented by bit1 at address 0x04.  There are three other single LED
//  "decimal points" on the display, which are at the following bit positions
//  bit2=top left, bit3=bottom left, bit4=top right
{
    I2C_WriteRegister(HT16K33,0x04,data<<1);
}

void SS_SetDigits(byte d0, byte d1, byte d2, byte d3, byte colon)
{
    SS_SetDigit(0,d0);
    SS_SetDigit(1,d1);
    SS_SetDigit(2,d2);
    SS_SetDigit(3,d3);
    SS_SetColon(colon);
}

void SS_Integer(int data, byte base)
{
    char st[5]="";
    itoa(data,st,base);                         // convert to string
    byte len = strlen(st);
    if (len>4) return;
    for (byte digit=0; digit<4; digit++)        // for all 4 digits
    {
        byte blanks = 4-len;                    // number of blanks
        if (digit<blanks)                       // right-justify display
            SS_SetDigit(digit,0x10);            // padding with blanks
        else
        {
            char ch = st[digit-blanks];         // get char for this digit
            if (ch>='a') ch-=87;                // correct for hex digits
            else ch-='0';                       // ascii -> numeric value
            SS_SetDigit(digit,ch);              // display digit
        }
    }
}


//      ---------------------------------------------------------------------------
//      APPLICATION ROUTINES

void SS_IntegerTest()                           // count 0 to 255 on display
{
    for (int i=0;i<256;i++)
    {
        SS_Integer(i,16);                       // choose your base here: 16=hex
        msDelay(150);
    }
}

void SS_BeefTest()                              // displays pulsating 'beeF' message
{
    SS_SetDigits(0x0b,0x0e,0x0e,0x0f,0);        // write 'beeF'
    for (byte count=0; count<3; count++)
    {
        for (byte j=15; j>0; j--)               // gradually dim the display
        {
            I2C_WriteByte(HT16K33,HT16K33_DIM + j);
            msDelay(100);
        }
        for (byte j=0; j<16; j++)               // gradually brighten the display
        {
            I2C_WriteByte(HT16K33,HT16K33_DIM + j);
            msDelay(100);
        }
    }
}

void SS_CircleTest()                           // show rotating circle on each digit
{
    for (byte count=0; count<15; count++)
    {
        for (byte i=0; i<6; i++)               // display each segment in turn
        {
            SS_SetDigitRaw(0,1<<i);
            SS_SetDigitRaw(1,1<<i);
            SS_SetDigitRaw(2,1<<i);
            SS_SetDigitRaw(3,1<<i);
            msDelay(100);
        }
    }
}

void LED_Time()
// display current time on 7-segment LED display from BCD input
{
    byte hours, minutes, seconds;
    DS1307_GetTime(&hours,&minutes,&seconds);
    SS_SetDigits(
    hours >> 4,                         // 10 hour digit
    hours & 0x0F,                       //  1 hour digit
    minutes >> 4,                       // 10 minute digit
    minutes & 0x0F,                     //  1 minute digit
    1);                                 // turn on colon
}


//      ---------------------------------------------------------------------------
//      MAIN PROGRAM

void MainLoop()
{
    while(1)
    {
        LED_Time();                     // put time on LED
        msDelay(1000);                  // one second between updates
    }
}


int main(void)
{
    InitAVR();                          // set port direction
    I2C_Init();                         // set I2C clock frequency
    SS_Init();                          // initialize HT16K33 LED controller
    SS_BeefTest();                      // pulsating 'beef' message
    SS_IntegerTest();                   // count in hexadecimal
    SS_CircleTest();                    // circle animation
    MainLoop();                         // display time on LCD & LED
}
```