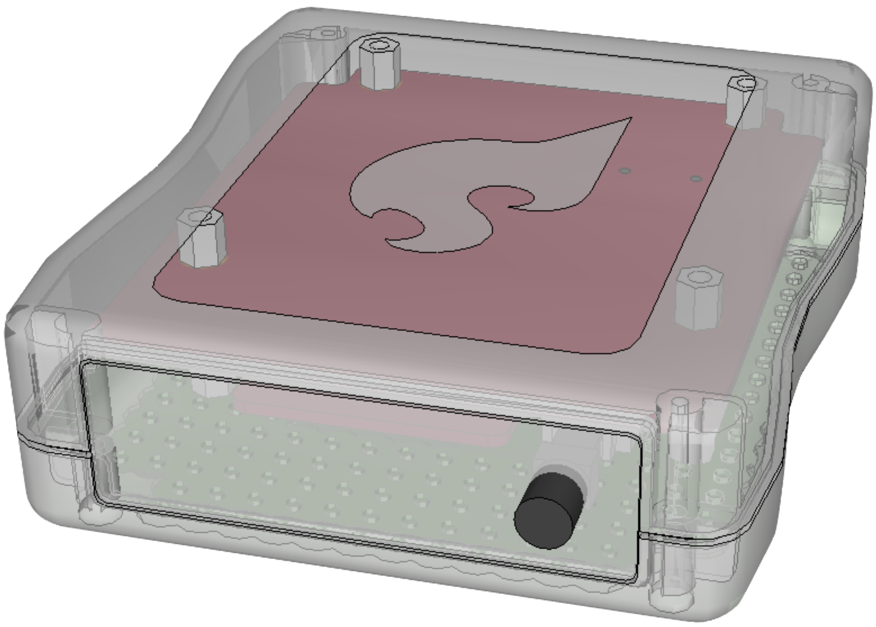
X8

PCB Assembly and Test

In this lab you will assemble and test your mixed digital and analogue circuit PCB that you designed in your previous lab X7. You will compare the performance of your PCB circuit to the simulated performance in lab X6.



Schedule

Preparation Time	:	1 hour
Lab Time	:	3 hours

Items provided

Tools	:	Soldering Iron, Iron stand, Solder, Desoldering tool, Blue-tac
Components	:	RFID tags (card and key fob)
Equipment	:	Oscilloscope [5], Logic Analyser [4], Bench PSU [3], Multimeter [1]
Software	:	`AVRDUDE`, `avr-gcc`

Items to bring

RFID components

C232HM cable

Identity card

Laboratory logbook

Version: April 21, 2014

Steve R. Gunn and David Oakley
Electronics and Computer Science
University of Southampton

Before entering the laboratory you should read through this document and complete the preparatory tasks detailed in section 2.

> **Academic Integrity** – *If you wish you may undertake the preparation jointly with other students. If you do so it is important that you acknowledge this fact in your logbook. Similarly, you will probably want to use sources from the internet to help answer some of the questions. Again, record any sources in your logbook.*

You will undertake the exercise working with your laboratory partner. During the exercise you should use your logbook to record your observations, which you can refer to in the future – perhaps to write a formal report on the exercise, or to remind you about the procedures. As such it should be legible, and observations should be clearly referenced to the appropriate part of the exercise. As a guide the ✎ symbol has been used to indicate a mandatory entry in your logbook. However, you should always record additional observations whenever something unexpected occurs, or when you discover something of interest.

Notation

This document uses the following conventions:

✎	An entry should be made in your logbook
`command input`	Command to be entered at the command line
Remarkable text	A point of note

1 Introduction

This laboratory exercise aims to:

- Enhance your skills in surface mount assembly
- Develop your skills in fault finding
- Illustrate the limitations of simulation

1.1 Outcomes

At the end of the exercise you should be able to:

- Assemble a reasonable sized mixed signal circuit on a PCB
- Perform system tests and validate correct operation of a mixed signal circuit
- Contrast the limitations of simulation with a real circuit

2 Preparation

The preparation for this exercise was done in X6 and X7. The only additional preparation required is to read through the remainder of these notes and collect together your components and the C232HM programming cable.

3 Laboratory Work

3.1 Construction

1. Begin by assembling all of the passive 0603 components (R1-11, C1-12). If the components are not marked you can use your multimeter set to the resistance or capacitance range to measure the values.
2. Assemble D1, IC1, IC2 and BZ1 making sure that they are orientated correctly.
3. Assemble X1.
4. Assemble S1 and S2; ensure that the latching switch is S1.
5. Assemble the battery clips for B1 and B2.
6. Assemble the display, LCD1.

Carefully check that all components are assembled to a good standard.

Designing a PCB is a challenging task and you should not be disheartened if your circuit does not work first time. There are hundreds of requirements which must be met, and just one error can result in failure. Many errors can be corrected with a small modification of the PCB by breaking incorrectly routed tracks with a knife or a dremel and patching with wire-wrap wire; it is not uncommon to see a few bits of wire on the back of a prototype PCB.

3.2 Power Test

Connect up a bench PSU (3.0V, current limit 100mA) and verify that the power LED and back light of the display illuminate. Record the power consumption. If the circuit draws more than 100mA there is either an error in your design, error in PCB fabrication, or a faulty component. You will then need to use your fault-finding skills to try to isolate the problem.

3.3 Programming Test

Connect up your FTDI cable in the same manner that you did in the X2 exercise. At this point you can power the device from the FTDI cable rather than the batteries if you wish.

Set the fuses to ensure correct clock speed by executing[1]

```
avrdude -p attiny25 -P usb -c c232hm -B 128 -U lfuse:w:0xE2:m
```

If you receive an error whilst programming, carefully check that you have connected the programming lines correctly. If so, you will need to investigate whether there is an error with the layout of the digital circuit. Verify that the microcontroller is orientated correctly and has power and ground connected correctly. Then verify that the programming signals are reaching the correct pins on the microcontroller by probing the microcontroller pins with an oscilloscope.

3.4 Digital Test

You will find the source code and pre-built hex file for the firmware in the `Firmware` directory of the `eagle.zip` archive that you downloaded for X7. Program the firmware by executing

```
avrdude -p attiny25 -P usb -c c232hm -B 128 -U flash:w:rfid.hex
```

You should hear a 'beep' and see 'Scanning' appear on the display. If not you will need to focus your fault-finding on the buzzer, I^2C interface and the display.

Verfiy that you can cycle through the five modes by pressing the mode switch.

[1] `-B 128` is necessary to slow down the programming speed as the programming lines have other components connected across them which limits the rise and fall times of the programming signals.

3.5 Analogue Test

Set the mode to *scanning* and verify that you can observe a square waveform at TP1 with a peak-to-peak value of around 3.0V at 125kHz. Capture a screenshot from the oscilloscope and enter it in your logbook. Comment on any differences from the simulation in X6. Repeat this for the waveform at

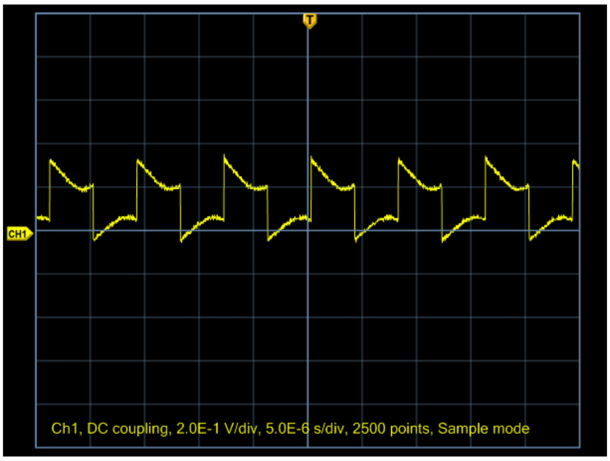


FIGURE 1: Test Point 1 (2V/Div)

TP2 which should be a 125kHz sine wave with a peak-to-peak amplitude greater than 3.0V. Record the peak-to-peak value and comment on the difference from the simulation. Finally, observe the signal

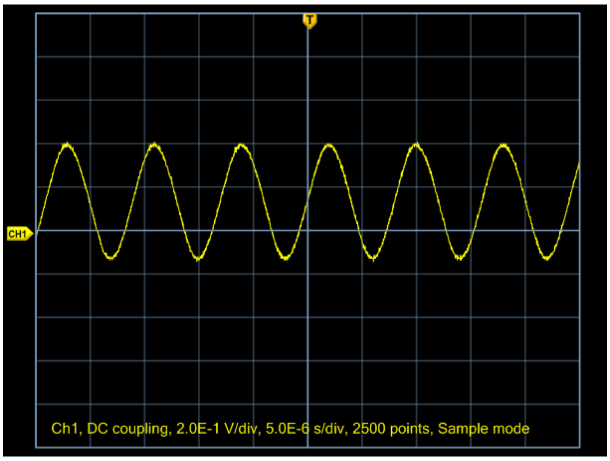


FIGURE 2: Test Point 2 (2V/Div)

at TP3 and record the capture in your logbook. The signal should have a DC component of around 1.5V and a small peak-to-peak amplitude when no tag is present. If the peak-to-peak amplitude is larger than 300mV you may have to reduce the gain of the active filter to ensure reliable operation. You can do this by reducing the value of R4. Place a tag in the vicinity of the reader and see whether the reader is able to recognise it. If you receive an error code, use the 'User Manual' or the source code to identify the error.

Observing the waveform at TP3 when a tag is placed on the reader should show a manchester encoded square wave with a peak-to-peak amplitude of approximately 3.0V. As the tag is moved away this amplitude will decay.

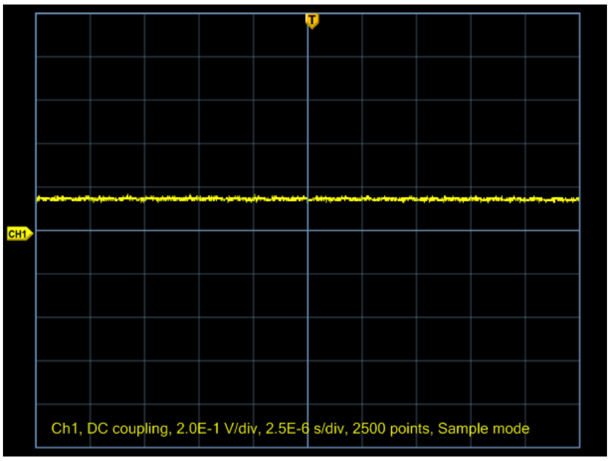


FIGURE 3: Test Point 3 (2V/Div)

If you need to debug the data stream that is captured by the sample buffer you can switch to diagnostic mode and connect your C232HM cable as a simplex UART with the 'buzzer' pin providing output in 8N1 at 19200 baud.

3.6 System Test

Work through the user manual [2] and confirm that your system works successfully for all five modes.

What is the lowest power supply voltage that the circuit will reliably work at?

What is the current consumption for the circuit in *scanning* mode?

What is the maximum range (cm) that your circuit will detect the tag?

What is the maximum range (cm) that your circuit will spoof a tag. You can use your partner's system to test this.

What is the difference in range (cm) for a card and key fob tag? Why are they different?

4 Optional Additional Work

Tune your resonant circuit by adjusting the values of C1 and C1A to optimise the range of the detector. This is achieved by maximising the peak-to-peak amplitude at TP2.

Modify the program to customise it. You will likely have to replace existing code since the current firmware occupies 2040 bytes of the 2048 bytes available (compiled using gcc version 4.8.1).

References

[1] Amprobe. Professional Digital Multimeter (30XR-A). Users Manual 9/06, 2006. URL https://secure.ecs.soton.ac.uk/notes/ellabs/reference/equipment/std-bench/DMM%20-%20Amprobe%2030XR-A.pdf.

[2] S.R. Gunn. 125kHz RFID Reader. User Manual 1.0, 2014. URL https://secure.ecs.soton.ac.uk/notes/ellabs/1/x8/rfid-um.pdf.

[3] Thurlby Thandar Instruments. Bench Power Supply (EL302P). Instruction Manual, 2009. URL https://secure.ecs.soton.ac.uk/notes/ellabs/reference/equipment/std-bench/PSU%20EL302P%20Instruction%20Manual%20-%20Iss%205.pdf.

[4] Saleae. Logic Analyser (Logic). User's Guide 1.1.15, 2012. URL https://secure.ecs.soton.ac.uk/notes/ellabs/reference/equipment/std-bench/Logic%20Analyser-%20Saleae%20User%20Guide.pdf.

[5] Tektronix. Digital Storage Oscilloscope (TDS1000C). User Manual Rev. A, 2006. URL https://secure.ecs.soton.ac.uk/notes/ellabs/reference/equipment/std-bench/Tektronix_TDS1000C_2000C_User_Manual.pdf.

A Firmware

```
//----------------------------------------------------------------------------//
// Program: 125kHz RFID Reader                                                //
//  Author: Steve Gunn                                                        //
// Licence: Creative Commons Attribution License                              //
//          See http://creativecommons.org/about/licenses/                    //
//    Date: 10th Feb 2014                                                     //
//----------------------------------------------------------------------------//

//----------------------------------------------------------------------------//
// DEVICE ATtiny25/45/85                                                      //
//                                                                            //
// AVR Memory Usage (avr-gcc 4.8.1)      |  Fuse  | Value | Default           //
// --------------------------------      |  ----------------------            //
// Device: attiny25                      |  Low    | 0x62  | 0xE2             //
//                                       |  High   | 0xDF  | 0xDF             //
// Program:    2040 bytes (99.6% Full)   |  Ext.   | 0xFF  | 0xFF             //
// (.text + .data + .bootloader)         |  (disable divide by 8, F_CPU=8Mhz) //
//                                       |                                    //
// Data:          20 bytes (15.6% Full)  |  Device    | Flash | SRAM         //
// (.data + .bss + .noinit)              |  ---------|-------|-----          //
//                                       |  ATtiny25 | 2048  | 128           //
// EEPROM:         1 bytes (0.8% Full)   |  ATtiny45 | 4096  | 256           //
// (.eeprom)                             |  ATtiny85 | 8192  | 512           //
//----------------------------------------------------------------------------//

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <avr/pgmspace.h>

//----------------------------------------------------------------------------//
// PINOUT                                                                     //
// ISP Programming on pins PB0-2,5                                            //
//----------------------------------------------------------------------------//
#define SDA          PB0           // MOSI/DI/SDA/AIN0/OC0A/~OC1A/AREF/PCINT0
#define BUZZER       PB1           // MISO/DO/AIN1/OC0B/OC1A/PCINT1
#define SCL          PB2           // SCK/USCK/SCL/ADC1/T0/INT0/PCINT2
#define DEMOD        PB3           // ADC3/~OC1B/CLKI/XTAL1/PCINT3
#define RFOUT        PB4           // ADC2/OC1B/CLKO/XTAL2/PCINT4
#define RESET        PB5           // dW/ADC0/~RESET/PCINT5

#define LOW(pin)     PORTB &= ~_BV(pin)
#define HIGH(pin)    PORTB |= _BV(pin)
#define INPUT(pin)   DDRB  &= ~_BV(pin)
#define OUTPUT(pin)  DDRB  |= _BV(pin)
#define IN(pin)      PINB  & _BV(pin)
```

A.1 Delay

```
//------------------------------------------------------------------------------//
// DELAY                                                                        //
// Routines taken from <util/delay_basic.h>                                     //
// Smaller code size than using routines in <util/delay.h>                      //
// Make them static to squeeze a bit more space                                 //
//------------------------------------------------------------------------------//
#define DELAY_LOOP_CLKS     3
static void delay(uint8_t __count)
{
    __asm__ volatile (
        "1: dec %0" "\n\t"
        "brne 1b"
        : "=r" (__count)
        : "0" (__count)
    );
}

#define DELAY2_LOOP_CLKS    4
static void delay2(uint16_t __count)
{
    __asm__ volatile (
        "1: sbiw %0,1" "\n\t"
        "brne 1b"
        : "=w" (__count)
        : "0" (__count)
    );
}
```

A.2 UART Communication

```
//----------------------------------------------------------------------------//
// UART                                                                       //
// Routines to communicate over UART (8N1) for debugging purposes            //
// Debug information over the BUZZER pin                                      //
//----------------------------------------------------------------------------//
#define UART_BAUD       19200
#define UART_DELAY      (F_CPU/DELAY_LOOP_CLKS)/UART_BAUD - 3

static void uart_tx(uint8_t c)
{
    uint8_t n;
    LOW(BUZZER);                    // Start bit
    delay(UART_DELAY);
    for(n=0x01; n; n<<=1) {         // Data bits
        if (c & n)
            HIGH(BUZZER);
        else
            LOW(BUZZER);
        delay(UART_DELAY);
    }
    HIGH(BUZZER);                   // Stop bit
    delay(UART_DELAY);
}
```

A.3 I²C Communication

```
//----------------------------------------------------------------------------//
// I2C                                                                        //
// Routines to communicate over I2C                                           //
// There is the USI but we only need single master write so simple bit-banging  //
// gets the job done just as effectively with a smaller code base             //
//----------------------------------------------------------------------------//
#define F_I2C          100000
#define I2C_DELAY      F_CPU/(2*DELAY_LOOP_CLKS*F_I2C)

static void i2c_start(void)
{
    delay(I2C_DELAY);
    LOW(SDA);
    delay(I2C_DELAY);
    LOW(SCL);
}

static void i2c_stop(void)
{
    LOW(SDA);
    LOW(SCL);
    delay(I2C_DELAY);
    HIGH(SCL);
    delay(I2C_DELAY);
    HIGH(SDA);
}

static uint8_t i2c_tx(uint8_t c)
{
    uint8_t ack;
    uint8_t n;
    for(n=0x80; n; n>>=1) {
        if (c & n)
            HIGH(SDA);
        else
            LOW(SDA);
        delay(I2C_DELAY);
        HIGH(SCL);
        delay(I2C_DELAY);
        LOW(SCL);
    }
    INPUT(SDA);
    delay(I2C_DELAY);
    HIGH(SCL);
    ack = IN(SDA);
    delay(I2C_DELAY);
    LOW(SCL);
    LOW(SDA);
    OUTPUT(SDA);
    return ack;
}
```

A.4 Signal Strength and System Health

```
//----------------------------------------------------------------------------//
// ADC                                                                        //
//----------------------------------------------------------------------------//
static void adc_read(uint8_t admux)
{
    ADMUX = admux;
    // Enable ADC with 8MHz/64 clock
    ADCSRA = _BV(ADEN) | _BV(ADPS2) | _BV(ADPS1);
    // Perform conversion
    ADCSRA |= _BV(ADSC);
    while(ADCSRA & _BV(ADSC));
    // Sec. 17.6.2: "The first ADC conversion after switching voltage reference
    // source may be inaccurate, and the user is advised to discard this result"
    ADCSRA |= _BV(ADSC);
    while(ADCSRA & _BV(ADSC));
}

//----------------------------------------------------------------------------//
// SIGNAL STRENGTH                                                            //
//----------------------------------------------------------------------------//
static uint8_t signal_strength(void)
{
    uint8_t min = 0xFF;
    uint8_t max = 0x00;
    uint8_t r;
    // Compute range of signal values
    for(r=0; r<128; r++) {
        // Select internal VCC reference with no external cap and PB3 input
        // Left shift result as we only require 8-bit accuracy
        adc_read(_BV(ADLAR) | _BV(MUX1) | _BV(MUX0));
        if (ADCH < min)
            min = ADCH;
        if (ADCH > max)
            max = ADCH;
    }
    return max - min;
}

//----------------------------------------------------------------------------//
// SYSTEM HEALTH                                                              //
//----------------------------------------------------------------------------//
static uint8_t battery_voltage(void)
{
    // Ensure RFOUT is disabled for accurate measurement
    // Select internal 2.56V reference with no external cap and PB3
    // Measurement only reliable whilst VCC > 2.56V
    // Left shift result as we only require 8-bit accuracy
    adc_read(_BV(REFS2) | _BV(REFS1) | _BV(ADLAR) | _BV(MUX1) | _BV(MUX0));
    // Answer stored in ADCH with 10mV precision
    // Measuring VCC/2 so VCC returned at 5mV precision
    return ADCH;
}

static uint8_t temperature(void)
{
    // Use value from datasheet (section 17.12) to estimate typical offset
    // You can adjust this value for one point calibration
    const uint16_t t_offset = 285;
    // Select internal 1.1V reference and select temp sensor
    adc_read(_BV(REFS1) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1) | _BV(MUX0));
    // Answer stored in ADC with ~1 degree C precision
    // Must read ADCH otherwise ADC registers locked and next conversion fails
    return (uint8_t)(ADC - t_offset);
}
```

A.5 LCD

```
//----------------------------------------------------------------------------//
// LCD                                                                        //
// Routines to display output on the LCD                                      //
// LCD driver is Sitronix ST7032i (I2C variant)                               //
//----------------------------------------------------------------------------//
#define SLAVE_ADDRESS   0x7C
#define WRITE           0x00
#define READ            0x01
#define CMDSEND         0x00
#define DATASEND        0x40
#define MULTIPLE        0x80
#define LCD_CLEAR       0x01
#define LCD_HOME        0x02
#define TOP_ROW         0x00
#define BOT_ROW         0x40
#define DDRAM_SIZE      0x28
#define SET_DDRAM       0x80
#define SET_CGRAM       0x40
#define LCD_MODE        0x08
#define LCD_ON          0x04
#define CURSOR          0x02
#define BLINK           0x01
#define ENTRY_MODE      0x04
#define INC             0x02
#define DEC             0x00
#define SHIFT           0x01
#define INSTR_TABLE     0x38
#define IS0             0x00
#define IS1             0x01
#define LCD_SHIFT       0x18
#define INT_OSC         0x10
#define BS              0x08
#define F_183HZ         0x04
#define BIAS_5          0x00
#define POWER           0x50
#define ICON_ON         0x08
#define BOOST_ON        0x04
#define CONTRAST        0x70
#define FOLLOWER        0x60
#define FON             0x08

static void lcd_init(void)
{
#define AUTO_CONTRAST
#ifdef AUTO_CONTRAST
    // Adjust the LCD contrast to suit the supply voltage
    // V_0 = (rab[2] << 1) x (1 + rab[1:0]/4) x (contrast + 36) x VCC/100
    // The following equation was experimentally determined with a prototype
    // and worked well over the supply range 2.5V to 3.5V. If your display is
    // difficult to read you could calibrate by adjusting the 106 value.
    const uint8_t contrast = 106 - (battery_voltage() >> 1);
    const uint8_t rab = 5;
#else
    // Use a fixed value good for around 3.0V
    const uint8_t contrast = 32;
    const uint8_t rab = 5;
#endif
    HIGH(SCL);
    OUTPUT(SCL);
    HIGH(SDA);
    OUTPUT(SDA);
    i2c_start();
    i2c_tx(SLAVE_ADDRESS | WRITE);
    // Create 5x4 pixel block character for bar display
    // Store in character position 0x00 of CGRAM
    i2c_tx(CMDSEND | MULTIPLE);
    i2c_tx(SET_CGRAM | 2);
    i2c_tx(DATASEND | MULTIPLE);
```

```
    i2c_tx(0x1F);
    i2c_tx(DATASEND | MULTIPLE);
    i2c_tx(0x1F);
    i2c_tx(DATASEND | MULTIPLE);
    i2c_tx(0x1F);
    i2c_tx(DATASEND | MULTIPLE);
    i2c_tx(0x1F);
    i2c_tx(CMDSEND);
    // Configure the display hardware
    i2c_tx(INSTR_TABLE | IS1);
    i2c_tx(INT_OSC | F_183HZ | BIAS_5);
    i2c_tx(CONTRAST | (contrast & 0x0F) );
    i2c_tx(POWER | ICON_ON | BOOST_ON | ((contrast & 0x30) >> 4));
    i2c_tx(FOLLOWER | FON | (rab & 0x07));
    i2c_tx(INSTR_TABLE | IS0);
    i2c_tx(LCD_MODE | LCD_ON);
    i2c_stop();
}

static void lcd_pos_dir(uint8_t pos, uint8_t dir)
{
    i2c_tx(CMDSEND | MULTIPLE);
    i2c_tx(ENTRY_MODE | dir);
    i2c_tx(CMDSEND | MULTIPLE);
    i2c_tx(SET_DDRAM | pos);
}

static void lcd_str(const char *str, uint8_t pos)
{
    uint8_t c = pgm_read_byte(str);
    i2c_start();
    i2c_tx(SLAVE_ADDRESS | WRITE);
    lcd_pos_dir(pos, INC);
    i2c_tx(DATASEND);
    while(c) {
        i2c_tx(c);
        c = pgm_read_byte(++str);
    }
    i2c_stop();
}

#define BASE10  0x20
#define BASE16  0x40
#define DP1     0x80

static void lcd_num(uint32_t n, uint8_t pos, uint8_t format)
{
    uint8_t digits = format & 0x0F;
    uint8_t digit;
    i2c_start();
    i2c_tx(SLAVE_ADDRESS | WRITE);
    lcd_pos_dir(pos, DEC);          // Number right justified
    i2c_tx(DATASEND);
    do {
        if (format & BASE10) {
            digit = n % 10;
            n /= 10;
        } else {  // BASE16
            digit = n & 0x0F;
            if (digit > 9)
                digit += 7;         // Compute offset for ascii hex letters
            n >>= 4;
        }
        i2c_tx('0' + digit);
        if ((format & DP1) && digits==2)
            i2c_tx('.');
    } while(--digits);
    i2c_stop();
}

static void lcd_bar(uint8_t length, uint8_t pos)
```

```
{
    static uint8_t bar;
    uint8_t i;
    i2c_start();
    i2c_tx(SLAVE_ADDRESS | WRITE);
    lcd_pos_dir(pos + bar, length > bar ? INC : DEC);
    i2c_tx(DATASEND);
    for(i=bar; i<=length; i++)
        i2c_tx(0x00);
    for(i=bar; i>length; i--)
        i2c_tx(' ');
    i2c_stop();
    bar = length;
}
```

A.6 Data Capture

```
//----------------------------------------------------------------------------//
// SAMPLING                                                                    //
// Routines to generate the RF signal and do the data capture                 //
// Data is Manchester decoded on-the-fly                                      //
// SRAM limit in attiny25 is 128 bytes so pack 8 databits per byte            //
// Timer 1 (8-bit) is used to generate the 125kHz Carrier                     //
// Carrier output is on OC1B                                                  //
// Pulse timing is done using sampling                                        //
// Preferred to edge triggered interrupts as it is more robust here           //
//----------------------------------------------------------------------------//
#define SAMPLES         128         // Size of sample buffer (max:255)
uint8_t data[SAMPLES>>3];           // Sample buffer

static uint8_t read_databit(uint8_t i)
{
    return (data[i>>3] & (1 << (i & 0x7)) ? 1 : 0);
}

static void write_databit(uint8_t i, uint8_t v)
{
    if (v)
        data[i>>3] |= (1 << (i & 0x7));
    else
        data[i>>3] &= ~(1 << (i & 0x7));
}

#define F_RFID          125000      // RFID Frequency

static void sampler_init(void)
{
    TCCR1  = _BV(CS10);             // Pre-scaler set to 1
    GTCCR  = _BV(PWM1B)             // PWM mode
           | _BV(COM1B1);           // OC1B clear on match, set when counter 0
    OCR1B  = F_CPU/(2*F_RFID);      // Set Timer to RFID frequency
    OCR1C  = (F_CPU/(F_RFID))-1;    // Set Timer to RFID frequency
    INPUT(DEMOD);                   // Enable demodulator input
    OUTPUT(RFOUT);                  // Enable RF output on OC1B
}

volatile uint8_t count = 0;         // Count 125kHz pulses
volatile uint8_t last_in = 0;       // Last demodulator state
volatile uint8_t pulse = 0;         // Number of counts between last two edges

ISR(TIMER1_OVF_vect)
{
    uint8_t in = IN(DEMOD);         // Get demodulator state
    if (count < 0xFF)               // Avoid count overflow
        count++;                    // Count 8us periods
    if (in != last_in) {            // Do we have an edge?
        pulse = count;              // Save the pulse length
        count = 0;                  // Reset counter
        last_in = in;               // Update the demodulator state
    }
}

#define BIT_CLKS        64          // Number of clocks cycles per bit
#define TOL             16          // Tolerance for clock cycles per bit
#define SHORT_PULSE()   pulse >= (BIT_CLKS/2 - TOL) && pulse < (BIT_CLKS/2 + TOL)
#define LONG_PULSE()    pulse >= (BIT_CLKS - TOL) && pulse < (BIT_CLKS + TOL)

static void sample_capture(void)
{
    uint8_t synced = 0;             // Wait for sync before filling buffer
    uint8_t sample = 0;             // Iterator for sample buffer
    uint8_t second_half = 0;        // Second half of short pulse
    uint8_t last_data = 0;          // Last databit value
    sampler_init();
    TIMSK |= _BV(TOIE1);            // Enable timer overflow interrupt
```

```c
    while(sample < SAMPLES)
        if (pulse) {
              if (synced) {
                  if (SHORT_PULSE()) {
                     if (second_half) {
                          second_half = 0;
                        write_databit(sample++, last_data);
                    } else           // Wait for second half pulse before write
                        second_half = 1;
                } else if(LONG_PULSE()) {
                    last_data = (last_data ? 0 : 1);
                    write_databit(sample++, last_data);
                  } else             // Unknown pulse width
                      synced = 0;    // Resynchronise
            } else                   // Look for long low pulse to synchronise
                if (last_in && LONG_PULSE()) {
                    synced = 1;      // Data can be Manchester decoded by
                    second_half = 0;// pulse length and previous bit
                    data[0] = 2;     // Long low pulse means first bits are 01
                    sample = 2;      // First two bits of buffer populated here
                    last_data = 1;   // Last Manchester databit decoded was a 1
                }
            pulse = 0;               // Pulse has been processed
        }
    TIMSK &= ~_BV(TOIE1);            // Disable timer overflow interrupt
}

static void sample_dump(void)
{
    uint8_t sample;                  // Iterator for sample buffer
    HIGH(BUZZER);                    // Make sure the output is high when turned on
    OUTPUT(BUZZER);                  // Enable the buzzer pin for UART tx data
    delay(0xFF);                     // Discard any buzzer corruption of tx line
    uart_tx('\n');
    uart_tx('\r');
    for(sample = 0; sample < SAMPLES; sample++)
        uart_tx('0' + read_databit(sample));
}
```

A.7 Data Analysis

```
//--------------------------------------------------------------------------------//
// ANALYSIS                                                                       //
// Routines to extract the RFID tag information from the sample buffer            //
// 64-bit tag data format:                                                        //
//  1 1 1 1 1 1 1 1 1   9-bit header (all 1)                                      //
// M00 M01 M02 M03 PR0    8-bit version number                                    //
// M04 M05 M06 M07 PR1                                                            //
// D00 D01 D02 D03 PR2  32-bit tag identifier                                     //
// D04 D05 D06 D07 PR3                                                            //
// D08 D09 D10 D11 PR4                                                            //
// D12 D13 D14 D15 PR5  PRr row parity (even)                                     //
// D16 D17 D18 D19 PR6                                                            //
// D20 D21 D22 D23 PR7                                                            //
// D24 D25 D26 D27 PR8                                                            //
// D28 D29 D30 D31 PR9                                                            //
// PC0 PC1 PC2 PC3      PCc Column parity (even)                                  //
//  0                   1 stop bit (0)                                            //
//--------------------------------------------------------------------------------//
#define TAG_BITS        64
#define HEADER_LENGTH   9
typedef struct {
    uint8_t version;
    uint32_t data;
} rfid_tag;

static uint16_t analyse(rfid_tag *tag, uint8_t *off)
{
    uint8_t pc[4] = {0, 0, 0, 0};   // Column parity
    uint8_t pr;                     // Row parity
    uint8_t row, col;               // Row and column counters
    uint8_t offset = 0;             // Offset to data in sample buffer
    uint8_t in_a_row = 0;           // Counter for consecutive bits
    uint8_t last_bit = 0;
    uint8_t bit;
    uint16_t error = 0x0000;        // 16-bit error code
    uint16_t err_mask = 0x8000;     // Pointer to current error bit
    while(!(in_a_row == HEADER_LENGTH && last_bit)) {
        bit = read_databit(offset++);
        if (bit != last_bit)        // Search for 9-bit header of 1's
            in_a_row = 0;
        last_bit = bit;
        in_a_row++;
        if (offset >= SAMPLES - TAG_BITS) {
            error |= err_mask;      // Header not found (bit 15)
            return error;
        }
    }
    err_mask >>= 1;
    for(row=0; row<10; row++) {     // Extract tag version number and ID
        pr = 0;
        for(col=0; col<4; col++) {
            bit = read_databit(offset++);
            pc[col] += bit;
            pr += bit;
            if (row<2) {
                tag->version <<= 1;
                tag->version |= bit;
            } else {
                tag->data <<= 1;
                tag->data |= bit;
            }
        }
        pr += read_databit(offset++);
        if (pr % 2)
            error |= err_mask;      // Row parity error (bit 14 - r)
        err_mask >>= 1;
    }
    for(col=0; col<4; col++) {      // Perform column parity check
```

```
        pc[col] += read_databit(offset++);
        if (pc[col] % 2)
            error |= err_mask;      // Column parity error (bit 4 - c)
        err_mask >>= 1;
    }
    if (read_databit(offset))       // Test stop bit
        error |= err_mask;          // Stop bit error (bit 0)
    *off = offset;                  // return offset (points to the last databit)
    return error;
}
```

A.8 Spoofing

```c
//----------------------------------------------------------------------------//
// SPOOF                                                                      //
// Routines to simulate a tag by switching the RFOUT pin between ground and   //
// high impedance states. Timer 1 is used to generate the timing information. //
// The demodulator pin must be disabled to avoid the ISR resetting the count. //
//----------------------------------------------------------------------------//

static void wait_256us(void)
{
    while(count < BIT_CLKS/2);
    count = 0;
}

static void manchester(uint8_t bit)
{
    if (bit) {
        INPUT(RFOUT);
        wait_256us();
        OUTPUT(RFOUT);
    } else {
        OUTPUT(RFOUT);
        wait_256us();
        INPUT(RFOUT);
    }
    wait_256us();
}

static void spoof(uint8_t offset)
{
    uint8_t i;                      // uint8_t ok, since offset >= TAG_BITS
    GTCCR &= ~_BV(COM1B1);          // Disable OC1B from timer
    LOW(RFOUT);                     // Ground RFOUT when output on
    DIDR0 = _BV(DEMOD);             // Disable demodulator input
    TIMSK |= _BV(TOIE1);            // Enable timer 1 overflow interrupt
    while(1)
        for(i = offset - TAG_BITS + 1; i <= offset; i++)
            manchester(read_databit(i));
}
```

A.9 Mode Selection

```
//---------------------------------------------------------------------------//
// MODE                                                                      //
// Since the ATtiny is tight on pins we re-purpose the external reset pin    //
// as a mode switch by storing the current mode in non-volatile memory       //
// Remember life-cycle for EEPROM: write is 100,000, read is unlimited       //
//---------------------------------------------------------------------------//
#define MODES    5
typedef enum {SCAN, SPOOF, SIGNAL, HEALTH, DIAGNOSTIC} mode;

mode EEMEM saved_mode = SCAN;       // Store the mode variable in EEPROM

static void set_mode(mode m)
{
    eeprom_write_byte(&saved_mode, m);
}

static mode get_mode(void)
{
    return eeprom_read_byte(&saved_mode);
}

static mode mode_init(void)
{
    mode m = SCAN;                  // Reset the mode for all reset conditions
    if (MCUSR & _BV(EXTRF))         // Except external reset
        m = get_mode() + 1;         // In which case increment mode
    if (m >= MODES)                 // Cycle through modes
        m = SCAN;
    set_mode(m);
    MCUSR = 0x00;                   // Clear reset flags
    return m;
}
```

A.10 Buzzer

```
//--------------------------------------------------------------------------------//
// BUZZER                                                                        //
// Disabled in diagnostic mode to avoid UART corruption                          //
// Routines to make simple sounds                                                //
// Timer 0 (8-bit) is used to generate the sounds                                //
// Use CTC mode 2 with OCR0A controlling the output frequency                    //
// Output is on OC0B                                                             //
//--------------------------------------------------------------------------------//
#define OCR_FROM_FREQ(f)    (F_CPU/(f))/128
#define TONE_DUR_UNIT_TIME  0.01
typedef enum {FAIL_SND, SUCCESS_SND, START_SND} sound;

static void tone(uint8_t ocra, uint8_t dur)
{
    OCR0A = ocra;
    while(dur--)
        delay2(F_CPU*TONE_DUR_UNIT_TIME/DELAY2_LOOP_CLKS);
    OCR0A = 0;
}

static void buzzer(sound s)
{
    if (get_mode() != DIAGNOSTIC) {
        TCCR0A = _BV(WGM01)         // Mode 2, CTC
               | _BV(COM0B0);       // Toggle OC0B on match
        TCCR0B = _BV(CS01)          // Pre-scaler set to 64
               | _BV(CS00);
        OUTPUT(BUZZER);
        switch(s) {
            case FAIL_SND:
                tone(OCR_FROM_FREQ(2000), 80);
                break;
            case SUCCESS_SND:
                tone(OCR_FROM_FREQ(4000), 40);
                break;
            case START_SND:
                tone(OCR_FROM_FREQ(2000), 20);
                tone(OCR_FROM_FREQ(4000), 10);
        }
        INPUT(BUZZER);
        TCCR0A = 0x00;              // Disable timer
    }
}
```

A.11 LED

```
//--------------------------------------------------------------------------------//
// LED                                                                           //
// Disabled in diagnostic mode to avoid UART corruption                          //
//--------------------------------------------------------------------------------//
static void led_on(void)
{
    if (get_mode() != DIAGNOSTIC) {
        HIGH(BUZZER);
        OUTPUT(BUZZER);
    }
}

static void led_off(void)
{
    if (get_mode() != DIAGNOSTIC)
        INPUT(BUZZER);
}
```

A.12 Main Program

```
//-----------------------------------------------------------------------------//
// MAIN PROGRAM                                                                //
//-----------------------------------------------------------------------------//
int main(void)
{
    rfid_tag       tag = {0, 0};
    uint16_t     error = 0xFFFF;
    uint8_t     offset = 0;
    mode working_mode = mode_init();
    mode current_mode = working_mode;
    buzzer(START_SND);
    lcd_init();
    sei();
    do {
        switch(current_mode) {
            case SCAN:
                if (working_mode == SCAN)
                    lcd_str(PSTR("Scanning"),     TOP_ROW |  0);
                led_on();
                sample_capture();
                led_off();
                error = analyse(&tag, &offset);
                if (error) {
                    lcd_str(PSTR("Error Code "), BOT_ROW |  0);
                    lcd_num(error,               BOT_ROW | 14, BASE16 |  4);
                    buzzer(FAIL_SND);
                } else {
                    lcd_str(PSTR("ID = "),       BOT_ROW |  0);
                    lcd_num(tag.data,            BOT_ROW | 14, BASE10 | 10);
                    buzzer(SUCCESS_SND);
                }
                current_mode = working_mode;
                break;
            case SPOOF:
                lcd_str(PSTR("Spoof"),           TOP_ROW |  0);
                if (!error) {
                    lcd_str(PSTR("ing"),         TOP_ROW |  5);
                    led_on();
                    spoof(offset);
                }
                current_mode = SCAN;
                break;
            case SIGNAL:
                lcd_str(PSTR("Signal Strength"), TOP_ROW |  0);
                sampler_init();
                lcd_bar(signal_strength() >> 4,  BOT_ROW |  0);
                break;
            case HEALTH:
                lcd_str(PSTR("Battery at"),      TOP_ROW |  0);
                lcd_num(battery_voltage() << 1,  TOP_ROW | 14, BASE10 | DP1 |  3);
                lcd_str(PSTR("V"),               TOP_ROW | 15);
                lcd_str(PSTR("Cooking at"),      BOT_ROW |  0);
                lcd_num(temperature(),           BOT_ROW | 12, BASE10 |  2);
                lcd_str(PSTR("\xDF\x43"),        BOT_ROW | 13);
                tone(0, 50);         // cheap delay - update at 2Hz
                break;
            case DIAGNOSTIC:
                lcd_str(PSTR("Diagnostic Mode"), TOP_ROW |  0);
                sample_dump();
                current_mode = SCAN;
                break;
        }
    } while(1);
}
```