```
/*
 * FILENAME:    main.c
 *
 * DESCRIPTION: Simple OS for AVR Based Systems
 *
 * Created: 9/16/2012 8:56:02 PM
 *  Author: Casey Stark <starkca@msoe.edu>
 *
 * Demonstrates usage of ATMEGA32 simple OS. Operates 4 processes (threads)
 * which currently outputs the process number to PORTB.
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include "OS.h"
#include "RFID.h"

// Functions for each process
void p0(void);
void p1(void);
void p2(void);
void p3(void);


/*************************************************************************
* Main Method. It Will:
*   1. Start OS
*   2. Spawn each new process
*   3. Execute each new process until system is shutdown
*************************************************************************/
int main(void)
{
    // Watchdog Reset
    uint8_t watchdog = MCUCSR;
    watchdog = watchdog & (1 << WDRF);
    // Watchdog was triggered
    if(!(watchdog == 0))
    {
        DDRC = 0xFF;
        PORTC = 0xFE;
        MCUCSR |= 1 << WDRF;
        while(1);
    }

    // ADC Left Adjusts, Take sample from ADC5, AVCC is reference
    ADMUX |= (1 << ADLAR) | (1 << MUX2) | (1 << MUX0) | (1 << REFS0);

    // ADC clock = 125 kHz (64 Prescaler), Enable ADC
    ADCSRA |= (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1);

    // Set PORTB to all outputs
    DDRB = 0xFF;
    PORTB = 0x00;
```

```
    // Initialize the OS
    startOS();

    // Add each process to the OS
    addProcess(p0);
    addProcess(p1);
    addProcess(p2);
    addProcess(p3);

    sei();

    while(1)
    {
        // Sit here untill context switcher kicks in
    }
}

void powerLoss(void)
{
    TCCR1B &= ~(1 << CS12);

    WDTCR |= 1 << WDTOE;
    WDTCR &= ~(1 << WDE);

    DDRB |= 0xFF;
    PORTB = 0xFF;

    while(1);
}

/*************************************************************************/
/* Process 0                                                             */
/*************************************************************************/
void p0(void)
{
    while(1)
    {
        PORTB = 0x00;

        DDRA = 0x00;

        ADCSRA |= 1 << ADSC;
        while(!(ADCSRA & (1 << ADIF)));
        ADCSRA |= 1 << ADIF;

        uint8_t adcValLow = ADCL;
        uint8_t adcValHigh = ADCH;

        if(adcValLow < 0x8F)
            powerLoss();
    }
}
```

```
/*****************************************************************************
* Process 1
*****************************************************************************/
void p1(void)
{
     // PORTA all inputs
     DDRA = 0x00;
     // AVCC as reference
     ADMUX |= (1 << REFS0);
     // Enable ADC, set prescaler to clk/128
     ADCSRA |= (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    while(1)
    {
        // Start the conversion
        ADCSRA |= 1 << ADSC;
        // Wait for the conversion to complete
        while(!(ADCSRA & (1 << ADIF)));
        // Clear the complete conversion flag
        ADCSRA |= 1 << ADIF;

        // Grab the data
        uint8_t lowData = ADCL;
        uint8_t highData = ADCH;

        if(lowData > 3)
            PORTB |= 1 << PB7;
        else
            PORTB &= ~(1 << PB7);
    }
}


/*****************************************************************************
* Process 2
*****************************************************************************/
void p2(void)
{
    // Launch the RFID system that was implemented in CE 3200
    RFIDMAIN();

    while(1)
    {
        // Just in case the RFID program fails, sit here forever
    }
}


/*****************************************************************************
* Process 3
*****************************************************************************/
void p3(void)
{
    while(1)
```

```
    {
        PORTB = 0x03;
    }
}
```

```
/*
 * FILENAME:    OS.h
 *
 * DESCRIPTION: Simple OS for AVR Based Systems
 *
 * Created: 9/16/2012 8:56:02 PM
 *  Author: Casey Stark <starkca@msoe.edu>
 *
 * Contains the interface for a very simple operating
 * system for embedded system.
 * Provides a very simple round robin based scheduler
 * based upon the AVR Timer Counter 1
 */

#ifndef OS_H_
#define OS_H_

// Stack size for each process
#define STACKSIZE 0x80

// Starting address of stack
#define STACKSTART 0x017F

// Maximum number of running processes
#define MAXPROCESSES 4

/*************************************************************************
* Starts the operation of the OS. Process control blocks will be
* initialized, and calling process given process block 0
*************************************************************************/
void startOS(void);

/*************************************************************************
* Add new process to OS. Process in passed in as a void pointer
*
* PARAMETERS:
*   void* function: Function which is to be the new process
*
* RETURNS:
*   int8_t:         Assigned Process ID (pid) or -1 if process can not
*                       be added
*************************************************************************/
int8_t addProcess(void* function);

#endif /* OS_H_ */
```

```c
/*
* FILENAME: OS.c
*
* DESCRIPTION:  Simple OS for AVR Based Systems
*
* Created: 9/16/2012 8:56:02 PM
*  Author: Casey Stark <starkca@msoe.edu>
*
* Contains the implementation for a very simple operating
* system for embedded system.
* Provides a very simple round robin based scheduler
* based upon the AVR Timer Counter 0
* Outputs the Process information to PORTB.
*/

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

#include "OS.h"

// Contains structure for a process control block.
// Includes:
//  1. Process ID
//  2. Priority setting
//  3. Number of times process ran
//  4. Process Stack Pointer
struct processStruct
{
    uint8_t PID;
    uint8_t priority;
    uint8_t switchCount;
    void* stackPtr;
};

// Holds the process control blocks (pcb) for the operating system.
// Each process gets their own pcb assigned to it
volatile struct processStruct pcbs[MAXPROCESSES];

// Holds the currently running process
volatile uint8_t currentProcess;

volatile static uint8_t firstRun;

// Tracks how many process are currently running
volatile uint8_t processCount;

// How many times has the process switched
volatile uint8_t switchCounter;

// Private Function Prototypes
void enableTimer(void);
```

```
/************************************************************************
* Start up the OS. Initializes Process Control Blocks,
* calling process is assigned block 0.
************************************************************************/
void startOS(void)
{
    firstRun = 1;

    // Set current process index
    currentProcess = 0;

    // Initialize process counter
    processCount = 0;

    // Enable Timer Interrupt
    enableTimer();
}


/************************************************************************
* Add new process to OS. Process in passed in as a void pointer
*
* PARAMETERS:
*   void* function: Function which is to be the new process
*
* RETURNS:
*   int8_t:         Assigned Process ID (pid) or -1 if process can not
*                       be added
************************************************************************/
int8_t addProcess(void* function)
{
    uint8_t retVal = -1;

    if(processCount < MAXPROCESSES)
    {
        // Setup PCB for new process
        pcbs[processCount].stackPtr = (void*)(STACKSTART + (STACKSIZE * processCount));

        // Place return to function on stack
        *(uint8_t*)pcbs[processCount].stackPtr = (0x00FF & (uint16_t)function);
        pcbs[processCount].stackPtr--;
        *(uint8_t*)pcbs[processCount].stackPtr = (0xFF00 & (uint16_t)function) >> 8;
        pcbs[processCount].stackPtr--;

        // Pad the stack
        pcbs[processCount].stackPtr = pcbs[processCount].stackPtr - 33;

        // Initialize each process block items
        pcbs[processCount].priority = processCount;
        pcbs[processCount].switchCount = 0;
        pcbs[processCount].PID = processCount + 1;
```

```
        processCount = processCount + 1;
        retVal = processCount;
    }

    return retVal;
}


/*************************************************************************
* Enables Timer 1 as periodic timer. When Fired, scheduler will
* run to determine next process to execute
*
* PARAMETERS:
*   VOID
*
* RETURNS:
*   VOID
*************************************************************************/
void enableTimer(void)
{
    // Compare register to 0x04E2
    OCR1A = 0x04E2;

    // Enable Compare Interrupt
    TIMSK |= 1 << OCIE1A;

    // Start counting at 0
    TCNT1 = 0x00;

    // Timmer 1 for CTC with Prescaler of 256
    TCCR1B |= (1 << WGM12) | (1 << CS12);
}


/*************************************************************************
* Cause scheduler to run. Determine next process to execute
*************************************************************************/
ISR(TIMER1_COMPA_vect, ISR_NAKED)
{
    asm("cli");

    // Store all registers to stack
    asm("push r0");
    asm("in r0, 0x3f");
    asm("push r0");
    asm("push r1");
    asm("push r2");
    asm("push r3");
    asm("push r4");
    asm("push r5");
    asm("push r6");
    asm("push r7");
    asm("push r8");
```

```
    asm("push r9");
    asm("push r10");
    asm("push r11");
    asm("push r12");
    asm("push r13");
    asm("push r14");
    asm("push r15");
    asm("push r16");
    asm("push r17");
    asm("push r18");
    asm("push r19");
    asm("push r20");
    asm("push r21");
    asm("push r22");
    asm("push r23");
    asm("push r24");
    asm("push r25");
    asm("push r26");
    asm("push r27");
    asm("push r28");
    asm("push r29");
    asm("push r30");
    asm("push r31");

    if(!firstRun)
    {
        PORTB = 0xFF;
        PORTB = 0x00;
        PORTB = 0xFF;

        switchCounter++;

        if(switchCounter < 800)
            asm("wdr");

        pcbs[currentProcess].stackPtr = (void*)SP;

        pcbs[currentProcess].switchCount = pcbs[currentProcess].switchCount + 1;

        if(currentProcess == (MAXPROCESSES - 1))
        {
            // Wrap back to initial process
            currentProcess = 0xFF;
        }

        currentProcess = currentProcess + 1;

        SP = (uint16_t)pcbs[currentProcess].stackPtr;

        // Output process Information as requested

        PORTB = pcbs[currentProcess].PID;
```

```
        PORTB = pcbs[currentProcess].priority;

        PORTB = pcbs[currentProcess].switchCount;

        PORTB = (uint8_t)(0x00FF & (uint16_t)pcbs[currentProcess].stackPtr);

        PORTB = (0xFF00 & ((uint16_t)pcbs[currentProcess].stackPtr)) >> 8;

        // Restore all of the registers to their previous state.
        asm("pop r31");
        asm("pop r30");
        asm("pop r29");
        asm("pop r28");
        asm("pop r27");
        asm("pop r26");
        asm("pop r25");
        asm("pop r24");
        asm("pop r23");
        asm("pop r22");
        asm("pop r21");
        asm("pop r20");
        asm("pop r19");
        asm("pop r18");
        asm("pop r17");
        asm("pop r16");
        asm("pop r15");
        asm("pop r14");
        asm("pop r13");
        asm("pop r12");
        asm("pop r11");
        asm("pop r10");
        asm("pop r9");
        asm("pop r8");
        asm("pop r7");
        asm("pop r6");
        asm("pop r5");
        asm("pop r4");
        asm("pop r3");
        asm("pop r2");
        asm("pop r1");
        asm("pop r0");
        asm("out 0x3f, r0");
        asm("pop r0");

        asm("reti");
    }

    else
    {
        firstRun = 0;

        SP = ((uint16_t)pcbs[currentProcess].stackPtr) + 33;
```

```
        switchCounter = 1;

        asm("reti");
    }

}
```

```
/*
 * RFID.h
 *
 * Created: 11/12/2012 9:37:29 PM
 *  Author: Casey
 */


#ifndef RFID_H_
#define RFID_H_

int RFIDMAIN(void);



#endif /* RFID_H_ */
```

```
/*****************************************************************************/
/* Provides an RFID system for the ATMEGA32 embedded platform that reads     */
/* an RFID card, stores its unique ID in EEPROM, then displays the ID        */
/* a serially connected device. The user can also view a complete            */
/* history of cards scanned by the system since setup.                       */
/*****************************************************************************/

#define F_CPU 16000000UL

#define RFID_BAUD_H 0x01
#define RFID_BAUD_L 0xA0

#define ENABLEPIN PD5

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

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <util/delay.h>
#include <stdbool.h>
#include <string.h>
#include <avr/sleep.h>

#include "uart.h"
#include "eeprom.h"

// Counter used in UART interrupt to keep track of ID places
volatile uint8_t IDCount = 0;

// Data from current card in process
volatile char currentCard[10];

// Array of all unique cards detected
volatile char history[50][11];

// Array of counts of each unique card detected
volatile uint8_t historyCount[50];

// How many unique cards have been seen
volatile uint8_t crdCnt = 0;

/*
* Flags to track status of separate subsystems
* of the system
*/
// Antenna On or Off
volatile bool antennaStatus = false;
// Does the user want to toggle the antenna status
volatile bool antennaBtn = false;
// Antenna is in process of toggling
```

```
volatile bool antennaTgl = false;
// Should the Heart be toggled
volatile bool pulseHrt = false;
// Is heart currently being displayed or not
volatile bool heart = false;
// Heart is in process of changing
volatile bool hrtCng = false;
// User would like history to be displayed
volatile bool historyBtn = false;
// History is in process of being displayed
volatile bool historyFlag = false;
// System encountered an event that requires something to
// be printed to UART
volatile bool print = false;
// Can the system goto sleep?
volatile bool canSleep = false;

// Counter to keep track of how long system has been inactive
volatile uint8_t sleepCnt = 0;


/*****************************************************************************/
/* Grabs the history from EEPROM upon power up                               */
/*                                                                           */
/* PARAMETERS:                                                               */
/*  - VOID                                                                   */
/*                                                                           */
/* RETURNS:                                                                  */
/*  - VOID                                                                   */
/*****************************************************************************/
void setupHistory(void)
{
    uint8_t eepromSetupCnt = 0;
    // Get the count of cards
    crdCnt = eeprom_read_byte((const uint8_t*)&eepromCrdCnt);

    // The number of cards is between a reasonable amount
    // So we will assume there is no fault in the memory
    if(crdCnt > 0 && crdCnt < 10)
    {
        for(; eepromSetupCnt < crdCnt; eepromSetupCnt++)
        {
            // Read in the ID of each card
            eeprom_read_block((void*)history[eepromSetupCnt],(const void*)eepromHistory[
            eepromSetupCnt],10);
            // Add in NULL terminator
            history[eepromSetupCnt][11] = "/0";
            // Read in count of each card
            historyCount[eepromSetupCnt] = eeprom_read_byte((const uint8_t*)eepromHistoryCnt[
            eepromSetupCnt]);
        }
    }
}
```

```
/*****************************************************************************/
/* Updates EEPROM with any new IDs seen and the count of each                */
/* ID seen                                                                   */
/*                                                                           */
/* PARAMETERS:                                                               */
/*  - VOID                                                                   */
/*                                                                           */
/* RETURNS:                                                                  */
/*  - VOID                                                                   */
/*****************************************************************************/
void updateEEPROMHistory(void)
{
    uint8_t eepromUpdateCnt = 0;
    // Update count of unique IDs seen
    eeprom_update_byte((uint8_t*)&crdCnt,eepromCrdCnt);

    for(; eepromUpdateCnt < crdCnt; eepromUpdateCnt++)
    {
        // Update list of each unique ID seen
        eeprom_update_block((const void*)&history[eepromUpdateCnt],(const void*)&eepromHistory[
        eepromUpdateCnt],10);
        // Update count of each unique ID seen
        eeprom_update_byte((uint8_t*)eepromHistoryCnt[eepromUpdateCnt],historyCount[
        eepromUpdateCnt]);
    }
}

/*****************************************************************************/
/*  Print the status and ID if available. Has to first set BAUD to           */
/*  9600 to work with Serial LCD panels, which means global interrupts       */
/*  must be disabled first.                                                  */
/*                                                                           */
/*  PARAMETERS:                                                              */
/*   - bool status: Current status of system (Antenna on or off)             */
/*                                                                           */
/*  RETURN:                                                                  */
/*   - VOID                                                                  */
/*****************************************************************************/
void printStatus(bool status)
{
    // Disable Interrupts
    cli();

    // Set BAUD to 9600 with a 16MHz Clock
    UBRRH = LCD_BAUD_H;
    UBRRL = LCD_BAUD_L;

    // We found there needs to be a delay here to let the
    // system setle in to the new BAUD
    _delay_ms(1);

    // Run defined output to clear terminal screen
```

```
    clrScr;

    // Start the status message
    printf("RFID Antenna Status: ");

    // The antenna is enabled, print out enabled
    // message
    if(status)
    {
        uint8_t prntCount = 0;
        // Inform user via UART that system is enabled
        printf("enabled");
        // Print generic message of tag ID
        printf("\nRFID Tag: ");

        // Card was presented and ready to print
        if(print)
        {
            // Print hex start
            printf("0x");

            // Iterate through holder for current card
            for(; prntCount < 10; prntCount++)
            {
                // Print each of the cards data values
                printf("%c",currentCard[prntCount]);

            }
            // Disable print flag other wise things will break
            print = false;
        }
    }

    // System is currently off, so inform the user
    else
    {
        // Complete status message
        printf("disabled");

        // Clear print flag just in case
        print = false;
    }


    // Allow rest of past message to run through the pipes before
    // resetting BAUD to RFID
    _delay_ms(2);

    // Reset BAUD
    UBRRH = RFID_BAUD_H;
    UBRRL = RFID_BAUD_L;

    // We found we needed this delay to allow for system to settle into new BAUD
```

```
    _delay_ms(1);

    // Enable global interrupts
    sei();
}

/*************************************************************************/
/* Setup port that RFID will be residing for enabling/disabling          */
/* In this implementation PD5 was used as it was free and UART is        */
/* already being used so why not used PORTD.                             */
/*                                                                       */
/* PARAMETERS:                                                           */
/*  - VOID                                                               */
/*                                                                       */
/* RETURNS:                                                              */
/*  - VOID                                                               */
/*************************************************************************/
void initRFID(void)
{
    // Enable Bit Port Output
    DDRD |= 1 << ENABLEPIN;

    // Enable Bit on/Turns RFID Off
    PORTD |= 1 << ENABLEPIN;
}

/*************************************************************************/
/* Initiate PORTD for the push button on PD2 and PD3 to trigger their    */
/* external interrupts on a falling edge                                 */
/*                                                                       */
/* PARAMETERS:                                                           */
/*  - VOID                                                               */
/*                                                                       */
/* RETURNS:                                                              */
/*  - VOID                                                               */
/*************************************************************************/
void initButtons(void)
{
    // PD2 Input
    DDRD |= (1 << PD2) | (1 << PD3);
    // Enable Pull-Up Resistors
    PORTD |= (1 << PD2) | (1 << PD3);

    // Trigger falling edge Interrupt
    MCUCR |= 1 << ISC10 | 1 << ISC00;
    // Enable INT0 Interrupts
    GICR |= (1 << INT0) | (1 << INT1);
}

/*************************************************************************/
/* Toggle the RFID antenna by writing a 0 - Enable or 1 - Disable to     */
/* corresponding pin                                                     */
/*************************************************************************/
```

```
void toggleAntenna(void)
{
    // Will be delaying in here, don't want to
    // break any interrupts
    cli();

    // Inform system the antenna is in process of changing
    antennaTgl = true;

    // Clear flag to track antenna button
    antennaBtn = false;

    // Wait for push button to be released
    while(!(PIND & 1 << PD2));

    // Let bounces stop
    _delay_ms(20);

    // CLear Interrupt just in case
    GIFR |= 1 << INTF0;

    // Toggle antenna status
    antennaStatus = !antennaStatus;

    // Change the Antenna bit
    if(antennaStatus)
        PORTD = PORTD & (~(1 << ENABLEPIN));
    else
        PORTD = PORTD | (1 << ENABLEPIN);

    // Print status message of current antenna state
    printStatus(antennaStatus);

    // Inform system the toggel has finished
    antennaTgl = false;

    // Enable interrupts
    sei();
}

/*****************************************************************************/
/* Ensure the current ID is present in the ID History, if not add it    */
/* also update the count of the ID                                           */
/*                                                                           */
/* May also add in EEPROM stuff here, or to the printHistory function -  */
/*  Not sure just yet                                                        */
/*                                                                           */
/* PARAMETERS:                                                               */
/*  - VOID                                                                   */
/*                                                                           */
/* RETURNS:                                                                  */
/*  - VOID                                                                   */
/*****************************************************************************/
```

```
void updateHistory(void)
{
    // This process may take some time, so disable
    // Interrupts just in case
    cli();

    // Some tracking variables
    uint8_t card;
    uint8_t ID;

    bool matchFound = false;

    // Iterate through each card currently in History to see if
    // it is already there
    for(card = 0; card < crdCnt; card++)
    {
        bool match = true;
        // Iterate through each ID value to see if the cards are the same
        for(ID = 0; ID < 10; ID++)
        {
            // Compare each ID
            match &= (history[card][ID] == currentCard[ID]);
        }
        // Found a match, increment the count
        if(match)
        {
            historyCount[card] = historyCount[card] + 1;
            matchFound = true;
        }
    }

    // Card not currently in History, lets add it
    if(!matchFound)
    {
        for(ID = 0; ID < 10; ID++)
        {
            history[crdCnt][ID] = currentCard[ID];
        }

        // Add in null terminator for printing
        history[crdCnt][11] = "\0";
        // Start the count of this card
        historyCount[crdCnt] = 1;
        // Increment count of each unique ID
        crdCnt++;
    }

    updateEEPROMHistory();

    // Enable interrupts
    sei();
}
```

```
/*****************************************************************************/
/* Prints each of the IDs that have been seen since startup and the     */
/* count that each card has been seen                                        */
/*                                                                           */
/* PARAMETERS:                                                               */
/*  - VOID                                                                   */
/*                                                                           */
/* RETURNS:                                                                  */
/*  - VOID                                                                   */
/*****************************************************************************/
void printHistory(void)
{
    // Will be printing, disable interrupts
    cli();

    // Tell system we are sending out history
    historyFlag = true;

    // Clear Button flag
    historyBtn = false;

    // Wait for button to be released
    while(!(PIND & 1 << PD3));

    // Allow button to debounce
    _delay_ms(20);

    // Clear interrupt flag just in case
    GIFR |= 1 << INTF1;

    // Set BAUD to LCD
    UBRRH = LCD_BAUD_H;
    UBRRL = LCD_BAUD_L;

    // Allow system to settle into new BAUD
    _delay_ms(1);

    // Clear the Screen
    clrScr;

    // Make a nice header
    printf("ID\t\tAccess Count\n");

    // Count number of card displayed
    uint8_t card;

    // Iterate through each card
    for(card = 0; card < crdCnt; card++)
    {
        // Print each card ID and count
        printf("0x%s\t\t%d\n",history[card], historyCount[card]);
    }
```

```
    // Print total number of unique cards seen
    printf("\n\n%d",crdCnt);

    // Allow final bits to travel
    _delay_ms(2);

    // Reset BAUD to RFID
    UBRRH = RFID_BAUD_H;
    UBRRL = RFID_BAUD_L;

    // Allow system to settle into new BAUD
    _delay_ms(1);

    // Done printing history
    historyFlag = false;

    // Enable interrupts
    sei();
}

/*************************************************************************/
/* Main entry point into program. Allows the user to enable an RFID     */
/* antenna connected via UART with the enable pin at ENABLEPIN on PORTD */
/*                                                                       */
/* System starts out in the disabled state, user must first trigger an  */
/*  INT0 Interrupt (PD2) which will then enable the RFID Antenna        */
/*                                                                       */
/* Once enabled, user can then scan an RFID equipped card and system    */
/*  will then display the stored ID on a serial display                 */
/*                                                                       */
/* User can also trigger an INT1 interrupt which will then display a    */
/*  history of each card the system has seen since deployment           */
/*                                                                       */
/* System also stores the history inside of EEPROM so system can be     */
/*  shutdown and history can still be viewed once power is restored     */
/*************************************************************************/
int RFIDMAIN(void)
{
    // PORTB as output
    DDRB = 0xFF;

    // Initialize Subsystems
    uart_init();
    initButtons();
    initRFID();
    setupHistory();

    printStatus(antennaStatus);

    // Enable Interrupts
    sei();

    // Loop through here to check each status flag
```

```
    while(1)
    {
        // User want to toggle antenna state and antenna is presently
        // not in the process of changing
        if(antennaBtn && !antennaTgl)
            // Toggle Antenna state
            toggleAntenna();

        // User wants to see the history and the history is currently
        // not being printed
        if(historyBtn && !historyFlag)
            // Display history
            printHistory();

        // Something wants to print the status
        if(print)
        {
            updateHistory();
            printStatus(antennaStatus);
        }

        // Set sleep mode to idle
        set_sleep_mode(SLEEP_MODE_IDLE);

        cli();
        if(canSleep)
        {
            sleep_enable();
            sei();
            sleep_cpu();
            sleep_disable();
        }

        sei();
    }
}

/*************************************************************************/
/* Interrupt to toggle Antenna Status                                    */
/*************************************************************************/
ISR(INT0_vect)
{
    canSleep = false;
    sleepCnt = 0;

    // Antenna is currently not in process of changing
    if(!antennaTgl)
        // Set flag and system will get right on that
        antennaBtn = true;
}

/*************************************************************************/
/* Interrupt to display history                                          */
```

```
/*****************************************************************************/
ISR(INT1_vect)
{
    canSleep = false;
    sleepCnt = 0;

    // History is currently not printing
    if(!historyFlag)
        // Set flag, system will get right on that
        historyBtn = true;
}

// Flag to track if an ID is on it's way in
volatile bool incomming = false;

/*****************************************************************************/
/* Interrupt for USART Receive which is used to receive the RFID ID     */
/*****************************************************************************/
ISR(USART_RXC_vect)
{
    sleepCnt = 0;
    canSleep = false;

    // Get data from UDR, store in RAM
    uint8_t data = UDR;

    // Start bit of ID
    if(data == 10)
    {
        // Inform system we are getting some data
        incomming = true;
        IDCount = 0;
    }
    // Currently receiving data, and not stop bit
    else if(incomming && data != 13)
    {
        // Store data into current card array
        currentCard[IDCount++] = data;
    }
    // Currently receiving and this is the stop bit
    else if(incomming && data == 13)
    {
        // Clear the counter
        IDCount = 0;
        // Tell system we are full
        incomming = false;
        // Print the ID
        print = true;
    }
    // Something broke
    else
    {
        // Disable RFID as a failsafe
```

```
        toggleAntenna();
    }
}
```

```
/* FILE:   uart.h
 * AUTHOR: Casey Stark <starkca@msoe.edu>
 * COURSE: CE3910
 * DATE:   3/14/12
 *
 * PURPOSE: Header file for UART API
 *          Contains declerations for
 *          functions available with this
 *          API and and constants required
 */

#ifndef uart_h
#define uart_h

#include <stdint.h>
#include <stdio.h>

#define LCD_BAUD_H 0x00
#define LCD_BAUD_L 0x67

// Size for UART buffer
#define MAX_BUFFER_SIZE 50

#define clrScr printf("\e[2J \e[H")
#define home printf("\e[H")
#define clrLn printf("\e[K")
#define hrt printf("\e[1;32f")

/*
*   PURPOSE: Initializes UART Functionality for
*               AtMega32. Enables functionality for
*               C's stdio functions as well.
*
*   PARAMETERS: None
*
*   RETURNS: None
*/
void uart_init(void);

/*
*   PURPOSE: Grabs char from UDR, if char is return char, reset
*               buffer and take appropriate actions, otherwise
*               echo all printable chars back. If buffer becomes
*               full, send a beep as a warning. Backspace is also
*               implemented appropriatly.
*
*   PARAMETERS: None
*
*   RETURNS: Character that was processed
*/
char uart_getc(void);

/*
```

```
*   PURPOSE: Add char to queue to be sent out.
*               If char is '\n', send also '\r'
*
*   PARAMETERS: char c: character to put transmited via
*                       serial connection
*
*   RETURN: None
*/
void uart_putc(char c);

/*
*   PURPOSE: Obtains the value located at regAddress
*               the prints it via stdio and returns
*               given value.
*
*   PARAMETERS: uint16_t regAddress: Address to
*                       collect data from
*
*   RETURNS:    Data at given regAddress
*/
uint8_t readIO(uint16_t regAddress);

/*
*   PURPOSE: Writes data to regAddress
*
*   PARAMETERS: uint16_t regAddress: Address of IO port
*                   to write data to.
*               uint8_t data: Data to be writen to
*                   regAddress
*
*   RETURN: None
*/
void writeIO(uint16_t regAddress,uint8_t data);

#endif
```

```
/* FILE:   uart.c
 * AUTHOR: Casey Stark <starkca@msoe.edu>
 * COURSE: CE3910
 * DATE:   3/14/12
 *
 * PURPOSE: This file contains functions
 *          that are required for
 *          UART communication.
 *          Functions include an
 *          Initializer, putc, and
 *          getc methods.
 */

#include "uart.h"
#include <avr/io.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Value to tell UART operations the clock
// speed and desired BAUD Rate.
#define UBRR_DEFAULT 416
#define UBRR_WRITE 103

volatile char RX_BUFF[MAX_BUFFER_SIZE];
volatile char* rxptr;
volatile char* cptr;

// Create FILE that allows for UART to take over C IO functions
FILE uart_str = FDEV_SETUP_STREAM(uart_putc,uart_getc,_FDEV_SETUP_RW);

/*
PURPOSE:    Initializes UART functionality for AtMega32. Takes over C's stdio
            functions.
PARAMETERS: VOID
RETURNS:    VOID
*/
void uart_init()
{
    UBRRH = 0x01;
    UBRRL = 0xA0;

    UCSRA = 0;

    // Transmit and Receive
    UCSRB = (1<<TXEN)|(1<<RXEN) |(1<<RXCIE);

    // synchronous operation, 8-bit char size
    UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);

    // initialize pointers to 0
    rxptr = 0;
    cptr = 0;
```

```
    // Finish up C IO integration
    stdout=stdin=&uart_str;

    return;
}

/*
PURPOSE:    Add char to queue to be sent out.
            If char is '\n', send also '\r'
PARAMETERS: char c: character to be transmitted
RETURN:     VOID
*/
void uart_putc(char c)
{
    // wait here until the UDR is empty
    while(!(UCSRA&(1<<UDRE)));

    // add the char to the UDR
    UDR = c;

    // if the char is a newline, also send return
    if(c == '\n')
    {
        uart_putc('\r');
    }
    return;
}

/*
PURPOSE:    Grabs char from UDR, if char is return char, reset
            buffer and take appropriate actions, otherwise
            echo all printable chars back. If buffer becomes
            full, send a beep as a warning. Backspace is also
            implemented appropriately.
PARAMETERS: VOID
RETURNS:    char: Processed Character
*/
char uart_getc(void)
{
    char c;
    // Start of new line?
    if(rxptr == 0)
    {
        // Write buffer = start of buffer
        for(cptr = RX_BUFF;;)
        {
            // poll for new character
            while(!(UCSRA&(1<<RXC)));
            c = UDR;
            // if the char is a return, replace with newline,
            // increment the pointer, send the newline, reset
            // the read pointer and break the loop
```

```
            if(c=='\r')
            {
                c = '\n';
                *cptr = c;
                cptr++;

                uart_putc(c);
                rxptr = RX_BUFF;
                break;
            }
            // if char is printable
            if((c >= ' ') && (c < 0x7F))
            {
                // if the buffer is full, send a beep to the terminal
                if(cptr == RX_BUFF + MAX_BUFFER_SIZE - 2)
                {
                    uart_putc('\a'); // beep
                }
                // otherwise set the char, increment the pointer, and send it
                else
                {
                    *cptr = c;
                    cptr++;
                    uart_putc(c);
                }

            }
            // if backspace or delete
            if((c == 0x08) || (c == 0x7F))
            {
                // if the write pointer is not at the start of the buffer
                if(cptr > RX_BUFF)
                {
                    uart_putc(0x08); // send backspace
                    uart_putc(' ');  // send space to overwrite previous char
                    uart_putc(0x08); // send backspace
                    cptr--; // decrement the buffer write pointer
                }
            }
        }
    }
    // get the character
    c = *rxptr;
    // increment the read pointer
    rxptr++;
    // if the char was a newline, reset the read pointer to 0
    if(c == '\n')
    {
        rxptr = 0;
    }
    // return the char
    return c;
}
```

```
/*
*   PURPOSE: Obtains the value located at regAddress
*               the prints it via stdio and returns
*               given value.
*
*   PARAMETERS: uint16_t regAddress: Address to
*                       collect data from
*
*   RETURNS:    Data at given regAddress
*/
uint8_t readIO(uint16_t regAddress)
{
    uint8_t regData = *(volatile uint8_t*) regAddress;
    printf("Register %u contains %u\n\n", regAddress, regData);
    return regData;
}

/*
*   PURPOSE: Writes data to regAddress
*
*   PARAMETERS: uint16_t regAddress: Address of IO port
*                   to write data to.
*               uint8_t data: Data to be writen to
*                   regAddress
*
*   RETURN: None
*/
void writeIO(uint16_t regAddress,uint8_t data)
{
    if(data <= 0xFF)
    {
        volatile uint8_t* regData = (uint8_t*) regAddress;
        *regData = data;
        if(*regData == data)
        {
            printf("Value %u now resides in %u\n", *regData, regAddress);
        }
        else
        {
            printf("Something Failed");
        }
    }
    else
    {
        printf("How Big Do You Think My Data Capacity Is? Enter A Smaller Number For Data.");
    }
    printf("\n");
    return;
}
```

```
/*
 * eeprom.h
 *
 * Created: 10/14/2012 7:39:20 PM
 *  Author: starkca
 */


#ifndef EEPROM_H_
#define EEPROM_H_

uint8_t EEMEM eepromCrdCnt = 0;
char EEMEM eepromHistory[50][11];
uint8_t EEMEM eepromHistoryCnt[50];

#endif /* EEPROM_H_ */
```