```
/**********************************************************/
/* Serial Bootloader for Atmel megaAVR Controllers        */
/*                                                        */
/* tested with ATmega8, ATmega128 and ATmega168           */
/* should work with other mega's, see code for details    */
/*                                                        */
/* ATmegaBOOT.c                                           */
/*                                                        */
/*                                                        */
/* 20090308: integrated Mega changes into main bootloader */
/*           source by D. Mellis                          */
/* 20080930: hacked for Arduino Mega (with the 1280       */
/*           processor, backwards compatible)             */
/*           by D. Cuartielles                            */
/* 20070626: hacked for Arduino Diecimila (which auto-    */
/*           resets when a USB connection is made to it)  */
/*           by D. Mellis                                 */
/* 20060802: hacked for Arduino by D. Cuartielles         */
/*           based on a previous hack by D. Mellis        */
/*           and D. Cuartielles                           */
/*                                                        */
/* Monitor and debug functions were added to the original */
/* code by Dr. Erik Lins, chip45.com. (See below)         */
/*                                                        */
/* Thanks to Karl Pitrich for fixing a bootloader pin     */
/* problem and more informative LED blinking!             */
/*                                                        */
/* For the latest version see:                            */
/* http://www.chip45.com/                                 */
/*                                                        */
/* ------------------------------------------------------ */
/*                                                        */
/* based on stk500boot.c                                  */
/* Copyright (c) 2003, Jason P. Kyle                      */
/* All rights reserved.                                   */
/* see avr1.org for original file and information         */
/*                                                        */
/* This program is free software; you can redistribute it */
/* and/or modify it under the terms of the GNU General    */
/* Public License as published by the Free Software       */
/* Foundation; either version 2 of the License, or        */
/* (at your option) any later version.                    */
/*                                                        */
/* This program is distributed in the hope that it will   */
/* be useful, but WITHOUT ANY WARRANTY; without even the  */
/* implied warranty of MERCHANTABILITY or FITNESS FOR A   */
/* PARTICULAR PURPOSE.  See the GNU General Public        */
/* License for more details.                              */
/*                                                        */
/* You should have received a copy of the GNU General     */
/* Public License along with this program; if not, write  */
/* to the Free Software Foundation, Inc.,                 */
/* 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA */
/*                                                        */
/* Licence can be viewed at                               */
/* http://www.fsf.org/licenses/gpl.txt                    */
/*                                                        */
/* Target = Atmel AVR m128,m64,m32,m16,m8,m162,m163,m169, */
/* m8515,m8535. ATmega161 has a very small boot block so  */
/* isn't supported.                                       */
/*                                                        */
/* Tested with m168                                       */
/**********************************************************/


/* some includes */
/*

inclusão de bibliotecas*/


#include <inttypes.h>
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/delay.h>

/* the current avr-libc eeprom functions do not support the ATmega168 */
/* own eeprom write/read functions are used instead */
#if !defined(__AVR_ATmega168__) || !defined(__AVR_ATmega328P__) || !defined(__AVR_ATmega328__)
//se nenhum estiver definido, ele faz a inclusão
#include <avr/eeprom.h>
#endif
```

```c

/* Use the F_CPU defined in Makefile */

/* 20060803: hacked by DojoCorp */
/* 20070626: hacked by David A. Mellis to decrease waiting time for auto-reset */
/* set the waiting time for the bootloader */
/* get this from the Makefile instead */
/* #define MAX_TIME_COUNT (F_CPU>>4) */

/* 20070707: hacked by David A. Mellis - after this many errors give up and launch application */
#define MAX_ERROR_COUNT 5

/* set the UART baud rate */
/* 20060803: hacked by DojoCorp */

/*Quanto mais perto estao os dispositivos mais alta a velocidade de transmissao*/

//#define BAUD_RATE   115200 //taxa alta espera que o canal de dados seja o mais limpo possivel

#ifndef BAUD_RATE //se a comunição nao puder ser estabelecida, define uma taxa mais baixa
#define BAUD_RATE   19200
#endif


/* SW_MAJOR and MINOR needs to be updated from time to time to avoid warning message from AVR Studio */
/* never allow AVR Studio to do an update !!!! */
#define HW_VER    0x02
#define SW_MAJOR 0x01
#define SW_MINOR 0x10


/* Adjust to suit whatever pin your hardware uses to enter the bootloader */
/* ATmega128 has two UARTS so two pins are used to enter bootloader and select UART */
/* ATmega1280 has four UARTS, but for Arduino Mega, we will only use RXD0 to get code */
/* BL0... means UART0, BL1... means UART1 */


/*
    Cada placa tem uma caracteristica diferente:
    Depende da pinagem de cada placa. Aqui determina as portas de comunicação
*/

#ifdef __AVR_ATmega128__
#define BL_DDR  DDRF
#define BL_PORT PORTF
#define BL_PIN  PINF
#define BL0     PINF7
#define BL1     PINF6
#elif defined __AVR_ATmega1280__
/* we just don't do anything for the MEGA and enter bootloader on reset anyway*/
#else
/* other ATmegas have only one UART, so only one pin is defined to enter bootloader */
#define BL_DDR  DDRD
#define BL_PORT PORTD
#define BL_PIN  PIND
#define BL      PIND6
#endif


/* onboard LED is used to indicate, that the bootloader was entered (3x flashing) */
/* if monitor functions are included, LED goes on after monitor was entered */
#if defined __AVR_ATmega128__ || defined __AVR_ATmega1280__
/* Onboard LED is connected to pin PB7 (e.g. Crumb128, PROBOmega128, Savvy128, Arduino Mega) */
#define LED_DDR  DDRB
#define LED_PORT PORTB
#define LED_PIN  PINB
#define LED      PINB7
#else
/* Onboard LED is connected to pin PB5 in Arduino NG, Diecimila, and Duomilanuove */
/* other boards like e.g. Crumb8, Crumb168 are using PB2 */
#define LED_DDR  DDRB
#define LED_PORT PORTB
#define LED_PIN  PINB
#define LED      PINB5
#endif


/* monitor functions will only be compiled when using ATmega128, due to bootblock size constraints */
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
#define MONITOR 1
#endif

```

```

/* define various device id's */
/* manufacturer byte is always the same */
#define SIG1    0x1E    // Yep, Atmel is the only manufacturer of AVR micros.  Single source :(

#if defined __AVR_ATmega1280__
#define SIG2    0x97
#define SIG3    0x03
#define PAGE_SIZE   0x80U   //128 words (tamanho da quantidade de palavras que podem ser armazenadas)

#elif defined __AVR_ATmega1281__
#define SIG2    0x97
#define SIG3    0x04
#define PAGE_SIZE   0x80U   //128 words

#elif defined __AVR_ATmega128__
#define SIG2    0x97
#define SIG3    0x02
#define PAGE_SIZE   0x80U   //128 words

#elif defined __AVR_ATmega64__
#define SIG2    0x96
#define SIG3    0x02
#define PAGE_SIZE   0x80U   //128 words

#elif defined __AVR_ATmega32__
#define SIG2    0x95
#define SIG3    0x02
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega16__
#define SIG2    0x94
#define SIG3    0x03
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega8__
#define SIG2    0x93
#define SIG3    0x07
#define PAGE_SIZE   0x20U   //32 words

#elif defined __AVR_ATmega88__
#define SIG2    0x93
#define SIG3    0x0a
#define PAGE_SIZE   0x20U   //32 words

#elif defined __AVR_ATmega168__
#define SIG2    0x94
#define SIG3    0x06
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega328P__
#define SIG2    0x95
#define SIG3    0x0F
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega328__
#define SIG2    0x95
#define SIG3    0x14
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega162__
#define SIG2    0x94
#define SIG3    0x04
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega163__
#define SIG2    0x94
#define SIG3    0x02
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega169__
#define SIG2    0x94
#define SIG3    0x05
#define PAGE_SIZE   0x40U   //64 words

#elif defined __AVR_ATmega8515__
#define SIG2    0x93
#define SIG3    0x06
#define PAGE_SIZE   0x20U   //32 words

#elif defined __AVR_ATmega8535__
#define SIG2    0x93
#define SIG3    0x08
```

```
#define PAGE_SIZE    0x20U   //32 words
#endif


/* function prototypes */]

/*
    Coloca no inicio, antes de ser
    usado, para que não ocorra problema
    posteriormente, nas chamadas de funções.
*/
void putch(char);
char getch(void);
void getNch(uint8_t);
void byte_response(uint8_t);
void nothing_response(void);
char gethex(void);
void puthex(char);
void flash_led(uint8_t);

/* some variables */
union address_union { /*representação de mesma area de memoria por tipos de dados diferentes
                        word dividido em 2 bytes. Facilita para pegar as partes alta e baixa do endereco*/
    uint16_t word;
    uint8_t  byte[2];
} address;

union length_union {
    uint16_t word;
    uint8_t  byte[2];
} length;

struct flags_struct {/*
                        Quando é necessario uso de flags, e, nessa estrutura, pegar 1 ou dois bits,
                        é necessario mascaramento. Para evitar isso, é usado mapa de bits. Estrutura com nome flag, com
                        tamanho de 2 bits, sendo um chamado eeprom e outro rampz. Vai ser 0 ou 1, conforme seu uso. Nunca vai
                        passar de 1 bit. Não pode ser maior conforme seu tipo de dado, mas pode ser maior.
*/
    unsigned eeprom : 1;
    unsigned rampz  : 1;
} flags;

uint8_t buff[256];
uint8_t address_high; /*endereco alto para manipulação*/

uint8_t pagesz=0x80;/*128*/

uint8_t i;
uint8_t bootuart = 0;

uint8_t error_count = 0;

void (*app_start)(void) = 0x0000;


/* main program starts here */
int main(void)
{
    uint8_t ch,ch2;
    uint16_t w;

#ifdef WATCHDOG_MODS
    ch = MCUSR;
    MCUSR = 0;

    WDTCSR |= _BV(WDCE) | _BV(WDE);
    WDTCSR = 0;
    /*Verifica se o WDT foi usado para reset*/
    // Check if the WDT was used to reset, in which case we dont bootload and skip straight to the code. woot.
    if (! (ch &  _BV(EXTRF))) // if its a not an external reset...
        app_start();  // skip bootloader
#else
    asm volatile("nop\n\t");
#endif

    /* set pin direction for bootloader pin and enable pullup */
    /* for ATmega128, two pins need to be initialized */
#ifdef __AVR_ATmega128__  /*Se definido ATmega128: */
```

```
    BL_DDR &= ~_BV(BL0);  /*Faz uma operação AND, e joga em BL_DDR*/
    BL_DDR &= ~_BV(BL1);    /*Preciso negar pra fazer o AND para tentar desligar o BIT apontado
                                pelo valor de BL. Em seguida liga.
                            */
    BL_PORT |= _BV(BL0);
    BL_PORT |= _BV(BL1);
#else
    /* We run the bootloader regardless of the state of this pin.  Thus, don't
    put it in a different state than the other pins.  --DAM, 070709
    This also applies to Arduino Mega -- DC, 080930
    BL_DDR &= ~_BV(BL);
    BL_PORT |= _BV(BL);
    */
#endif


#ifdef __AVR_ATmega128__
    /* check which UART should be used for booting */
    if(bit_is_clear(BL_PIN, BL0)) {
        bootuart = 1;
    }
    else if(bit_is_clear(BL_PIN, BL1)) {
        bootuart = 2;
    }
#endif

#if defined __AVR_ATmega1280__
    /* the mega1280 chip has four serial ports ... we could eventually use any of them, or not? */
    /* however, we don't wanna confuse people, to avoid making a mess, we will stick to RXD0, TXD0 */
    bootuart = 1;
#endif

    /* check if flash is programmed already, if not start bootloader anyway */
    if(pgm_read_byte_near(0x0000) != 0xFF) { /*Verifica se a memoria esta vazia*/

#ifdef __AVR_ATmega128__
    /* no UART was selected, start application */
    if(!bootuart) {
        app_start();
    }
#else
    /* check if bootloader pin is set low */
    /* we don't start this part neither for the m8, nor m168 */
    //if(bit_is_set(BL_PIN, BL)) {
    //      app_start();
    //    }
#endif
    }

#ifdef __AVR_ATmega128__
    /* no bootuart was selected, default to uart 0 */
    if(!bootuart) {
        bootuart = 1;
    }
#endif


    /* initialize UART(s) depending on CPU defined */
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
    if(bootuart == 1) {
        UBRR0L = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
        UBRR0H = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
        UCSR0A = 0x00;
        UCSR0C = 0x06;
        UCSR0B = _BV(TXEN0)|_BV(RXEN0); /*TRansmissao esta no TXEN0*/
    }
    if(bootuart == 2) {
        UBRR1L = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
        UBRR1H = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
        UCSR1A = 0x00;
        UCSR1C = 0x06;
        UCSR1B = _BV(TXEN1)|_BV(RXEN1);/*Transmissao esta em TXEN1*/
    }
#elif defined __AVR_ATmega163__
    UBRR = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
    UBRRHI = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
    UCSRA = 0x00;
    UCSRB = _BV(TXEN)|_BV(RXEN);
#elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)

#ifdef DOUBLE_SPEED
    UCSR0A = (1<<U2X0); //Double speed mode USART0
    UBRR0L = (uint8_t)(F_CPU/(BAUD_RATE*8L)-1);/*Acertar os registradores para que a comunicacao funcione. Desloca os bits para esquerda com
                                                    o valor definido na constante. F_CPU determina a freq do controlador, transforma em bytes
                                                    e determina a velocidade de comunicação efetiva em que a CPU consegue trabalhar.
                                                    Se a freq da CPU for menor, recebe uma quantidade de dados maior que o que consegue trabalhar.
                                                    A taxa tem que ser menor para que haja tempo de receber os dados e trabalhar
                                                    */
    UBRR0H = (F_CPU/(BAUD_RATE*8L)-1) >> 8;
#else
    UBRR0L = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
    UBRR0H =   (F_CPU/(BAUD_RATE*16L)-1) >> 8;/* Determina em termos binarios a velocidade de comunicação

                                                */
#endif

    UCSR0B = (1<<RXEN0) | (1<<TXEN0); /* Liga valores binarios dentro do registrador*/
    UCSR0C = (1<<UCSZ00) | (1<<UCSZ01);

    /* Enable internal pull-up resistor on pin D0 (RX), in order
    to supress line noise that prevents the bootloader from
    timing out (DAM: 20070509) */
    DDRD &= ~_BV(PIND0);
    PORTD |= _BV(PIND0);
#elif defined __AVR_ATmega8__
    /* m8 */
    UBRRH = (((F_CPU/BAUD_RATE)/16)-1)>>8;   // set baud rate
    UBRRL = (((F_CPU/BAUD_RATE)/16)-1);
    UCSRB = (1<<RXEN)|(1<<TXEN);  // enable Rx & Tx
    UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);  // config USART; 8N1
#else
    /* m16,m32,m169,m8515,m8535 */
    UBRRL = (uint8_t)(F_CPU/(BAUD_RATE*16L)-1);
    UBRRH = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
    UCSRA = 0x00;
    UCSRC = 0x06;
    UCSRB = _BV(TXEN)|_BV(RXEN);
#endif

#if defined __AVR_ATmega1280__
    /* Enable internal pull-up resistor on pin D0 (RX), in order
    to supress line noise that prevents the bootloader from
    timing out (DAM: 20070509) */
    /* feature added to the Arduino Mega --DC: 080930 */
    DDRE &= ~_BV(PINE0);
    PORTE |= _BV(PINE0);
#endif


    /* set LED pin as output */
    LED_DDR |= _BV(LED); /*Define a porta do LED*/


    /* flash onboard LED to signal entering of bootloader */
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
    // 4x for UART0, 5x for UART1
    flash_led(NUM_LED_FLASHES + bootuart);
#else
    flash_led(NUM_LED_FLASHES); /*Conta o numero de vezes que o LED deve piscar*/
#endif

    /* 20050803: by DojoCorp, this is one of the parts provoking the
         system to stop listening, cancelled from the original */
    //putch('\0');

    /* forever loop */
    for (;;) {

    /* get character from UART */
    ch = getch();

    /* A bunch of if...else if... gives smaller code than switch...case ! */

    /* Hello is anyone home ? */
    if(ch=='0') {
        nothing_response(); /*Nada para responder, não recebeu nada*/
    }


    /* Request programmer ID */
    /* Not using PROGMEM string due to boot block in m128 being beyond 64kB boundry  */
    /* Would need to selectively manipulate RAMPZ, and it's only 9 characters anyway so who cares.  */
    else if(ch=='1') {
        if (getch() == ' ') {
            putch(0x14); /*
                        Se recebeu 1:
                            responde "AVR ISP"
                        */
            putch('A');
            putch('V');
            putch('R');
            putch(' ');
            putch('I');
            putch('S');
            putch('P');
            putch(0x10);
        } else {
            if (++error_count == MAX_ERROR_COUNT)
                app_start();
        }
    }


    /* AVR ISP/STK500 board commands  DON'T CARE so default nothing_response */
    else if(ch=='@') {
        ch2 = getch();
        if (ch2>0x85) getch();
        nothing_response();
    }


    /* AVR ISP/STK500 board requests */
    else if(ch=='A') {
        ch2 = getch();
        if(ch2==0x80) byte_response(HW_VER);     // Hardware version
        else if(ch2==0x81) byte_response(SW_MAJOR);  // Software major version
        else if(ch2==0x82) byte_response(SW_MINOR);  // Software minor version
        else if(ch2==0x98) byte_response(0x03);      // Unknown but seems to be required by avr studio 3.56
        else byte_response(0x00);                // Covers various unnecessary responses we don't care about
    }


    /* Device Parameters  DON'T CARE, DEVICE IS FIXED  */
    else if(ch=='B') {
        getNch(20);
        nothing_response();
    }


    /* Parallel programming stuff  DON'T CARE  */
    else if(ch=='E') {
        getNch(5);
        nothing_response();
    }


    /* P: Enter programming mode  */
    /* R: Erase device, don't care as we will erase one page at a time anyway.  */
    else if(ch=='P' || ch=='R') {
        nothing_response();
    }


    /* Leave programming mode  */
    else if(ch=='Q') {
        nothing_response();
#ifdef WATCHDOG_MODS
        // autoreset via watchdog (sneaky!)
        WDTCSR = _BV(WDE);
        while (1); // 16 ms
#endif
    }


    /* Set address, little endian. EEPROM in bytes, FLASH in words  */
    /* Perhaps extra address bytes may be added in future to support > 128kB FLASH.  */
    /* This might explain why little endian was used here, big endian used everywhere else.  */
    else if(ch=='U') {
        address.byte[0] = getch();
        address.byte[1] = getch();
        nothing_response();
    }


    /* Universal SPI programming command, disabled.  Would be used for fuses and lock bits.  */
    else if(ch=='V') {
        if (getch() == 0x30) {
            getch();
            ch = getch();
            getch();
            if (ch == 0) {
                byte_response(SIG1);
            } else if (ch == 1) {
                byte_response(SIG2);
            } else {
                byte_response(SIG3);
            }
        } else {
            getNch(3);
            byte_response(0x00);
        }
    }


    /* Write memory, length is big endian and is in bytes  */
    else if(ch=='d') {
        length.byte[1] = getch();  /*Joga os valores dentro da memoria*/
        length.byte[0] = getch();
        flags.eeprom = 0;
        if (getch() == 'E') flags.eeprom = 1;
        for (w=0;w<length.word;w++) {
            buff[w] = getch();                        // Store data in buffer, can't keep up with serial data stream whilst programming pages
        }
        if (getch() == ' ') {
            if (flags.eeprom) {                    //Write to EEPROM one byte at a time
                address.word <<= 1;
                for(w=0;w<length.word;w++) {
#if defined(__AVR_ATmega168__)  || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__)
                    while(EECR & (1<<EEPE));
                    EEAR = (uint16_t)(void *)address.word;
                    EEDR = buff[w];
                    EECR |= (1<<EEMPE);
                    EECR |= (1<<EEPE);
#else
                    eeprom_write_byte((void *)address.word,buff[w]);
                    /*
                       Código é jogado em uma memoria para ser executado
                    */
#endif
                    address.word++;
                }
            }
            else {                          //Write to FLASH one page at a time
                if (address.byte[1]>127) address_high = 0x01;    //Only possible with m128, m256 will need 3rd address byte. FIXME
                else address_high = 0x00;
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__)
                RAMPZ = address_high;
#endif
                address.word = address.word << 1;            //address * 2 -> byte location
                /* if ((length.byte[0] & 0x01) == 0x01) length.word++;    //Even up an odd number of bytes */
                if ((length.byte[0] & 0x01)) length.word++;  //Even up an odd number of bytes
                cli();                  //Disable interrupts, just to be sure
#if defined(EEPE)
                while(bit_is_set(EECR,EEPE));            //Wait for previous EEPROM writes to complete
#else
                while(bit_is_set(EECR,EEWE));            //Wait for previous EEPROM writes to complete
#endif
                asm volatile(
                     "clr   r17      \n\t"   //page_word_count
                     "lds   r30,address \n\t"   //Address of FLASH location (in bytes)
                     "lds   r31,address+1    \n\t"
                     "ldi   r28,lo8(buff)    \n\t"   //Start of buffer array in RAM
                     "ldi   r29,hi8(buff)    \n\t"
                     "lds   r24,length   \n\t"   //Length of data to be written (in bytes)
                     "lds   r25,length+1     \n\t"
                     "length_loop:       \n\t"   //Main loop, repeat for number of words in block
                     "cpi   r17,0x00     \n\t"   //If page_word_count=0 then erase page
                     "brne  no_page_erase    \n\t"
                     "wait_spm1:         \n\t"
                     "lds   r16,%0       \n\t"   //Wait for previous spm to complete
                     "andi  r16,1           \n\t"
                     "cpi   r16,1           \n\t"
                     "breq  wait_spm1       \n\t"
                     "ldi   r16,0x03    \n\t"   //Erase page pointed to by Z
                     "sts   %0,r16      \n\t"
                     "spm           \n\t"
#ifdef __AVR_ATmega163__
                     ".word 0xFFFF      \n\t"
                     "nop           \n\t"
#endif
                     "wait_spm2:        \n\t"
                     "lds   r16,%0      \n\t"   //Wait for previous spm to complete
                     "andi  r16,1           \n\t"
                     "cpi   r16,1           \n\t"
                     "breq  wait_spm2       \n\t"

                     "ldi   r16,0x11    \n\t"   //Re-enable RWW section
                     "sts   %0,r16      \n\t"
                     "spm           \n\t"
#ifdef __AVR_ATmega163__
                     ".word 0xFFFF      \n\t"
                     "nop           \n\t"
#endif
                     "no_page_erase:    \n\t"
                     "ld    r0,Y+       \n\t"   //Write 2 bytes into page buffer
                     "ld    r1,Y+       \n\t"

                     "wait_spm3:        \n\t"
                     "lds   r16,%0      \n\t"   //Wait for previous spm to complete
                     "andi  r16,1           \n\t"
                     "cpi   r16,1           \n\t"
                     "breq  wait_spm3       \n\t"
                     "ldi   r16,0x01    \n\t"   //Load r0,r1 into FLASH page buffer
                     "sts   %0,r16      \n\t"
                     "spm           \n\t"

                     "inc   r17     \n\t"   //page_word_count++
                     "cpi r17,%1            \n\t"
                     "brlo  same_page   \n\t"   //Still same page in FLASH
                     "write_page:       \n\t"
                     "clr   r17     \n\t"   //New page, write current one first
                     "wait_spm4:        \n\t"
                     "lds   r16,%0      \n\t"   //Wait for previous spm to complete
                     "andi  r16,1           \n\t"
                     "cpi   r16,1           \n\t"
                     "breq  wait_spm4       \n\t"
#ifdef __AVR_ATmega163__
                     "andi  r30,0x80    \n\t"   // m163 requires Z6:Z1 to be zero during page write
#endif
                     "ldi   r16,0x05    \n\t"   //Write page pointed to by Z
                     "sts   %0,r16      \n\t"
                     "spm           \n\t"
#ifdef __AVR_ATmega163__
                     ".word 0xFFFF      \n\t"
                     "nop           \n\t"
                     "ori   r30,0x7E    \n\t"   // recover Z6:Z1 state after page write (had to be zero during write)
#endif
                     "wait_spm5:        \n\t"
                     "lds   r16,%0      \n\t"   //Wait for previous spm to complete
                     "andi  r16,1           \n\t"
                     "cpi   r16,1           \n\t"
                     "breq  wait_spm5       \n\t"
                     "ldi   r16,0x11    \n\t"   //Re-enable RWW section
                     "sts   %0,r16      \n\t"
                     "spm           \n\t"
#ifdef __AVR_ATmega163__
                     ".word 0xFFFF      \n\t"
                     "nop           \n\t"
#endif
                     "same_page:        \n\t"
                     "adiw  r30,2       \n\t"   //Next word in FLASH
                     "sbiw  r24,2       \n\t"   //length-2
                     "breq  final_write \n\t"   //Finished
                     "rjmp  length_loop \n\t"
                     "final_write:      \n\t"
                     "cpi   r17,0       \n\t"
                     "breq  block_done  \n\t"
                     "adiw  r24,2       \n\t"   //length+2, fool above check on length after short page write
                     "rjmp  write_page  \n\t"
                     "block_done:       \n\t"
                     "clr   __zero_reg__    \n\t"   //restore zero register
```

```c
#if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__) || defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__)
                     : "=m" (SPMCSR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","r25","r28","r29","r30","r31"
#else
                     : "=m" (SPMCR) : "M" (PAGE_SIZE) : "r0","r16","r17","r24","r25","r28","r29","r30","r31"
#endif
                     );
                /* Should really add a wait for RWW section to be enabled, don't actually need it since we never */
                /* exit the bootloader without a power cycle anyhow */
            }
            putch(0x14);
            putch(0x10);
        } else {
            if (++error_count == MAX_ERROR_COUNT)
                app_start();
        }
    }


    /* Read memory block mode, length is big endian.  */
    else if(ch=='t') {
        length.byte[1] = getch();
        length.byte[0] = getch();
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
        if (address.word>0x7FFF) flags.rampz = 1;         // No go with m256, FIXME
        else flags.rampz = 0;
#endif
        address.word = address.word << 1;            // address * 2 -> byte location
        if (getch() == 'E') flags.eeprom = 1;
        else flags.eeprom = 0;
        if (getch() == ' ') {                        // Command terminator
            putch(0x14);
            for (w=0;w < length.word;w++) {              // Can handle odd and even lengths okay
                if (flags.eeprom) {                          // Byte access EEPROM read
#if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__)
                    while(EECR & (1<<EEPE));
                    EEAR = (uint16_t)(void *)address.word;
                    EECR |= (1<<EERE);
                    putch(EEDR);

#else
                    putch(eeprom_read_byte((void *)address.word));
#endif
                    address.word++;
                }
                else {

                    if (!flags.rampz) putch(pgm_read_byte_near(address.word));
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
                    else putch(pgm_read_byte_far(address.word + 0x10000));
                    // Hmmmm, yuck  FIXME when m256 arrvies
#endif
                    address.word++;
                }
            }
            putch(0x10);
        }
    }


    /* Get device signature bytes  */
    else if(ch=='u') {
        if (getch() == ' ') {
            putch(0x14);
            putch(SIG1);
            putch(SIG2);
            putch(SIG3);
            putch(0x10);
        } else {
            if (++error_count == MAX_ERROR_COUNT)
                app_start();
        }
    }


    /* Read oscillator calibration byte */
    else if(ch=='v') {
        byte_response(0x00);
    }


#if defined MONITOR
```

```

    /* here come the extended monitor commands by Erik Lins */

    /* check for three times exclamation mark pressed */
    else if(ch=='!') {
        ch = getch();
        if(ch=='!') {
        ch = getch();
        if(ch=='!') {
            PGM_P welcome = "";
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
            uint16_t extaddr;
#endif
            uint8_t addrl, addrh;

#ifdef CRUMB128
            welcome = "ATmegaBOOT / Crumb128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
#elif defined PROBOMEGA128
            welcome = "ATmegaBOOT / PROBOmega128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
#elif defined SAVVY128
            welcome = "ATmegaBOOT / Savvy128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
#elif defined __AVR_ATmega1280__
            welcome = "ATmegaBOOT / Arduino Mega - (C) Arduino LLC - 090930\n\r";
#endif

            /* turn on LED */
            LED_DDR |= _BV(LED);
            LED_PORT &= ~_BV(LED);

            /* print a welcome message and command overview */
            for(i=0; welcome[i] != '\0'; ++i) {
                putch(welcome[i]);
            }

            /* test for valid commands */
            for(;;) {
                putch('\n');
                putch('\r');
                putch(':');
                putch(' ');

                ch = getch();
                putch(ch);

                /* toggle LED */
                if(ch == 't') {
                    if(bit_is_set(LED_PIN,LED)) {
                        LED_PORT &= ~_BV(LED);
                        putch('1');
                    } else {
                        LED_PORT |= _BV(LED);
                        putch('0');
                    }
                }

                /* read byte from address */
                else if(ch == 'r') {
                    ch = getch(); putch(ch);
                    addrh = gethex();
                    addrl = gethex();
                    putch('=');
                    ch = *(uint8_t *)((addrh << 8) + addrl);
                    puthex(ch);
                }

                /* write a byte to address  */
                else if(ch == 'w') {
                    ch = getch(); putch(ch);
                    addrh = gethex();
                    addrl = gethex();
                    ch = getch(); putch(ch);
                    ch = gethex();
                    *(uint8_t *)((addrh << 8) + addrl) = ch;
                }

                /* read from uart and echo back */
                else if(ch == 'u') {
                    for(;;) {
                        putch(getch());
                    }
                }
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
                /* external bus loop  */
                else if(ch == 'b') {
                    putch('b');
                    putch('u');
                    putch('s');
                    MCUCR = 0x80;
                    XMCRA = 0;
                    XMCRB = 0;
                    extaddr = 0x1100;
                    for(;;) {
                        ch = *(volatile uint8_t *)extaddr;
                        if(++extaddr == 0) {
                            extaddr = 0x1100;
                        }
                    }
                }
#endif

                else if(ch == 'j') {
                    app_start();
                }

            } /* end of monitor functions */

        }
        }
    }
    /* end of monitor */
#endif
    else if (++error_count == MAX_ERROR_COUNT) {
        app_start();
    }
    } /* end of forever loop */

}


char gethexnib(void) {
    char a;
    a = getch(); putch(a);
    if(a >= 'a') {
        return (a - 'a' + 0x0a);
    } else if(a >= '0') {
        return(a - '0');
    }
    return a;
}


char gethex(void) {
    return (gethexnib() << 4) + gethexnib();
}


void puthex(char ch) {
    char ah;

    ah = ch >> 4;
    if(ah >= 0x0a) {
        ah = ah - 0x0a + 'a';
    } else {
        ah += '0';
    }

    ch &= 0x0f;
    if(ch >= 0x0a) {
        ch = ch - 0x0a + 'a';
    } else {
        ch += '0';
    }

    putch(ah);
    putch(ch);
}


void putch(char ch) /*Recebe o caracter e joga na porta serial
                    PROTOCOLO DE COMUNICAÇÃO SERIAL TRADICIONAL
                        */
{
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
    if(bootuart == 1) {
        while (!(UCSR0A & _BV(UDRE0))); /*Testar os valores*/
        UDR0 = ch;
    }
    else if (bootuart == 2) {
        while (!(UCSR1A & _BV(UDRE1)));
        UDR1 = ch;
    }
#elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
    while (!(UCSR0A & _BV(UDRE0)));
    UDR0 = ch;
#else
    /* m8,16,32,169,8515,8535,163 */
    while (!(UCSRA & _BV(UDRE)));
    UDR = ch;
#endif
}


char getch(void)
{
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
    uint32_t count = 0;
    if(bootuart == 1) {
        while(!(UCSR0A & _BV(RXC0))) {
            /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
            /* HACKME:: here is a good place to count times*/
            count++;
            if (count > MAX_TIME_COUNT)
                app_start();
        }

        return UDR0; /*Retorna o valor UDR0*/
    }
    else if(bootuart == 2) {
        while(!(UCSR1A & _BV(RXC1))) {
            /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
            /* HACKME:: here is a good place to count times*/
            count++;
            if (count > MAX_TIME_COUNT)
                app_start();
        }

        return UDR1;
    }
    return 0;
#elif defined(__AVR_ATmega168__)  || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
    uint32_t count = 0;
    while(!(UCSR0A & _BV(RXC0))){
        /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
        /* HACKME:: here is a good place to count times*/
        count++;
        if (count > MAX_TIME_COUNT)
            app_start();
    }
    return UDR0;
#else
    /* m8,16,32,169,8515,8535,163 */
    uint32_t count = 0;
    while(!(UCSRA & _BV(RXC))){
        /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
        /* HACKME:: here is a good place to count times*/
        count++;
        if (count > MAX_TIME_COUNT)
            app_start();
    }
    return UDR;
#endif
}


void getNch(uint8_t count)
{
    while(count--) {
#if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
        if(bootuart == 1) {
            while(!(UCSR0A & _BV(RXC0)));
            UDR0;
        }
        else if(bootuart == 2) {
            while(!(UCSR1A & _BV(RXC1)));
            UDR1;
        }
#elif defined(__AVR_ATmega168__)  || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
        getch();
#else
        /* m8,16,32,169,8515,8535,163 */
        /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
        //while(!(UCSRA & _BV(RXC)));
        //UDR;
        getch(); // need to handle time out
#endif
    }
}


void byte_response(uint8_t val)
{
    if (getch() == ' ') {
        putch(0x14);
        putch(val);
        putch(0x10);
    } else {
        if (++error_count == MAX_ERROR_COUNT)
            app_start();
    }
}


void nothing_response(void)
{
    if (getch() == ' ') {
        putch(0x14);
        putch(0x10);
    } else {
        if (++error_count == MAX_ERROR_COUNT)
            app_start();
    }
}

void flash_led(uint8_t count)
{
    while (count--) {
        LED_PORT |= _BV(LED); /*Liga a porta*/
        _delay_ms(100);       /*Dá um delay de 100 ms*/
        LED_PORT &= ~_BV(LED); /*Desliga a porta*/
        _delay_ms(100); /*Delay de 100 ms*/
    }
}


/* end of file ATmegaBOOT.c */

```