

```

1  /*****
2  /* Serial Bootloader for Atmel megaAVR Controllers */
3  /* */
4  /* tested with ATmega8, ATmega128 and ATmega168 */
5  /* should work with other mega's, see code for details */
6  /* */
7  /* ATmegaBOOT.c */
8  /* */
9  /* */
10 /* 20090308: integrated Mega changes into main bootloader */
11 /* source by D. Mellis */
12 /* 20080930: hacked for Arduino Mega (with the 1280 */
13 /* processor, backwards compatible) */
14 /* by D. Cuartielles */
15 /* 20070626: hacked for Arduino Diecimila (which auto- */
16 /* resets when a USB connection is made to it) */
17 /* by D. Mellis */
18 /* 20060802: hacked for Arduino by D. Cuartielles */
19 /* based on a previous hack by D. Mellis */
20 /* and D. Cuartielles */
21 /* */
22 /* Monitor and debug functions were added to the original */
23 /* code by Dr. Erik Lins, chip45.com. (See below) */
24 /* */
25 /* Thanks to Karl Pitrich for fixing a bootloader pin */
26 /* problem and more informative LED blinking! */
27 /* */
28 /* For the latest version see: */
29 /* http://www.chip45.com/ */
30 /* */
31 /* ----- */
32 /* */
33 /* based on stk500boot.c */
34 /* Copyright (c) 2003, Jason P. Kyle */
35 /* All rights reserved. */
36 /* see avrl.org for original file and information */
37 /* */
38 /* This program is free software; you can redistribute it */
39 /* and/or modify it under the terms of the GNU General */
40 /* Public License as published by the Free Software */
41 /* Foundation; either version 2 of the License, or */
42 /* (at your option) any later version. */
43 /* */
44 /* This program is distributed in the hope that it will */
45 /* be useful, but WITHOUT ANY WARRANTY; without even the */
46 /* implied warranty of MERCHANTABILITY or FITNESS FOR A */
47 /* PARTICULAR PURPOSE. See the GNU General Public */
48 /* License for more details. */
49 /* */
50 /* You should have received a copy of the GNU General */
51 /* Public License along with this program; if not, write */
52 /* to the Free Software Foundation, Inc., */
53 /* 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA */
54 /* */
55 /* License can be viewed at */
56 /* http://www.fsf.org/licenses/gpl.txt */
57 /* */
58 /* Target = Atmel AVR m128,m64,m32,m16,m8,m162,m163,m169, */
59 /* m8515,m8535. ATmega161 has a very small boot block so */
60 /* isn't supported. */
61 /* */
62 /* Tested with m168 */
63 /*****/
64
65
66 /* some includes */
67 /*
68
69 inclusão de bibliotecas*/
70
71
72 #include <inttypes.h>
73 #include <avr/io.h>
74 #include <avr/pgmspace.h>
75 #include <avr/interrupt.h>
76 #include <avr/wdt.h>
77 #include <util/delay.h>
78
79 /* the current avr-libc eeeprom functions do not support the ATmega168 */
80 /* own eeeprom write/read functions are used instead */
81 #if !defined( __AVR_ATmega168__ ) || !defined( __AVR_ATmega328P__ ) || !defined( __AVR_ATmega328__ )
82 //se nenhum estiver definido, ele faz a inclusão
83 #include <avr/eeeprom.h>
84 #endif

```

```

85
86 /* Use the F_CPU defined in Makefile */
87
88 /* 20060803: hacked by DoioCorp */
89 /* 20070626: hacked by David A. Mellis to decrease waiting time for auto-reset */
90 /* set the waiting time for the bootloader */
91 /* get this from the Makefile instead */
92 /* #define MAX_TIME_COUNT (F_CPU>>4) */
93
94 /* 20070707: hacked by David A. Mellis - after this many errors give up and launch application */
95 #define MAX_ERROR_COUNT 5
96
97 /* set the UART baud rate */
98 /* 20060803: hacked by DoioCorp */
99
100 /*Quanto mais perto estao os dispositivos mais alta a velocidade de transmissao*/
101
102 // #define BAUD_RATE 115200 // taxa alta espera que o canal de dados seja o mais limpo possivel
103
104 #ifndef BAUD_RATE // se a comunicação não puder ser estabelecida, define uma taxa mais baixa
105 #define BAUD_RATE 19200
106 #endif
107
108
109 /* SW MAJOR and MINOR needs to be updated from time to time to avoid warning message from AVR
Studio */
110 /* never allow AVR Studio to do an update !!!! */
111 #define HW_VER 0x02
112 #define SW MAJOR 0x01
113 #define SW MINOR 0x10
114
115
116 /* Adjust to suit whatever pin your hardware uses to enter the bootloader */
117 /* ATmega128 has two UARTS so two pins are used to enter bootloader and select UART */
118 /* ATmega1280 has four UARTS, but for Arduino Mega, we will only use RXD0 to get code */
119 /* BL0... means UART0, BL1... means UART1 */
120
121
122 /*
123 Cada placa tem uma característica diferente:
124 Depende da pinagem de cada placa. Aqui determina as portas de comunicação
125 */
126
127 #ifdef __AVR_ATmega128__
128 #define BL_DDR DDRF
129 #define BL_PORT PORTF
130 #define BL_PIN PINF
131 #define BL0 PINF7
132 #define BL1 PINF6
133 #elif defined __AVR_ATmega1280__
134 /* we just don't do anything for the MEGA and enter bootloader on reset anyway */
135 #else
136 /* other ATmegs have only one UART, so only one pin is defined to enter bootloader */
137 #define BL_DDR DDRD
138 #define BL_PORT PORTD
139 #define BL_PIN PIND
140 #define BL PIND6
141 #endif
142
143
144 /* onboard LED is used to indicate, that the bootloader was entered (3x flashing) */
145 /* if monitor functions are included, LED goes on after monitor was entered */
146 #if defined __AVR_ATmega128__ || defined __AVR_ATmega1280__
147 /* Onboard LED is connected to pin PB7 (e.g. Crumb128, PROBOmega128, Savvy128, Arduino Mega) */
148 #define LED_DDR DDRB
149 #define LED_PORT PORTB
150 #define LED_PIN PINB
151 #define LED PINB7
152 #else
153 /* Onboard LED is connected to pin PB5 in Arduino NG, Diecimila, and Duomilanuove */
154 /* other boards like e.g. Crumb8, Crumb168 are using PB2 */
155 #define LED_DDR DDRB
156 #define LED_PORT PORTB
157 #define LED_PIN PINB
158 #define LED PINB5
159 #endif
160
161
162 /* monitor functions will only be compiled when using ATmega128, due to bootblock size
constraints */
163 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
164 #define MONITOR 1
165 #endif
166

```

```

167
168 /* define various device id's */
169 /* manufacturer byte is always the same */
170 #define SIG1 0x1E // Yep, Atmel is the only manufacturer of AVR micros. Single source :(
171
172 #if defined __AVR_ATmega1280__
173 #define SIG2 0x97
174 #define SIG3 0x03
175 #define PAGE_SIZE 0x80U //128 words (tamanho da quantidade de palavras que podem ser
endereçadas)
176
177 #elif defined __AVR_ATmega1281__
178 #define SIG2 0x97
179 #define SIG3 0x04
180 #define PAGE_SIZE 0x80U //128 words
181
182 #elif defined __AVR_ATmega128__
183 #define SIG2 0x97
184 #define SIG3 0x02
185 #define PAGE_SIZE 0x80U //128 words
186
187 #elif defined __AVR_ATmega64__
188 #define SIG2 0x96
189 #define SIG3 0x02
190 #define PAGE_SIZE 0x80U //128 words
191
192 #elif defined __AVR_ATmega32__
193 #define SIG2 0x95
194 #define SIG3 0x02
195 #define PAGE_SIZE 0x40U //64 words
196
197 #elif defined __AVR_ATmega16__
198 #define SIG2 0x94
199 #define SIG3 0x03
200 #define PAGE_SIZE 0x40U //64 words
201
202 #elif defined __AVR_ATmega8__
203 #define SIG2 0x93
204 #define SIG3 0x07
205 #define PAGE_SIZE 0x20U //32 words
206
207 #elif defined __AVR_ATmega88__
208 #define SIG2 0x93
209 #define SIG3 0x0a
210 #define PAGE_SIZE 0x20U //32 words
211
212 #elif defined __AVR_ATmega168__
213 #define SIG2 0x94
214 #define SIG3 0x06
215 #define PAGE_SIZE 0x40U //64 words
216
217 #elif defined __AVR_ATmega328P__
218 #define SIG2 0x95
219 #define SIG3 0x0F
220 #define PAGE_SIZE 0x40U //64 words
221
222 #elif defined __AVR_ATmega328__
223 #define SIG2 0x95
224 #define SIG3 0x14
225 #define PAGE_SIZE 0x40U //64 words
226
227 #elif defined __AVR_ATmega162__
228 #define SIG2 0x94
229 #define SIG3 0x04
230 #define PAGE_SIZE 0x40U //64 words
231
232 #elif defined __AVR_ATmega163__
233 #define SIG2 0x94
234 #define SIG3 0x02
235 #define PAGE_SIZE 0x40U //64 words
236
237 #elif defined __AVR_ATmega169__
238 #define SIG2 0x94
239 #define SIG3 0x05
240 #define PAGE_SIZE 0x40U //64 words
241
242 #elif defined __AVR_ATmega8515__
243 #define SIG2 0x93
244 #define SIG3 0x06
245 #define PAGE_SIZE 0x20U //32 words
246
247 #elif defined __AVR_ATmega8535__
248 #define SIG2 0x93
249 #define SIG3 0x08

```

```

250 #define PAGE_SIZE    0x20U    //32 words
251 #endif
252
253
254 /* function prototypes */
255
256 /*
257     Coloca no inicio, antes de ser
258     usado, para que não ocorra problema
259     posteriormente, nas chamadas de funções.
260 */
261 void putch(char);
262 char getch(void);
263 void getNch(uint8_t);
264 void byte_response(uint8_t);
265 void nothing_response(void);
266 char gethex(void);
267 void puthex(char);
268 void flash_led(uint8_t);
269
270 /* some variables */
271 union address_union { /*representação de mesma area de memoria por tipos de dados diferentes
272                        word dividido em 2 bytes. Facilita para pegar as partes alta e baixa do
273                        endereço*/
274     uint16_t word;
275     uint8_t  byte[2];
276 } address;
277
278 union length_union {
279     uint16_t word;
280     uint8_t  byte[2];
281 } length;
282
283 struct flags_struct {/*
284                        Quando é necessario uso de flags, e, nessa estrutura, pegar 1 ou dois
285                        bits,
286                        é necessario mascaramento. Para evitar isso, é usado mapa de bits.
287                        Estrutura com nome flag, com
288                        tamanho de 2 bits, sendo um chamado eeprom e outro rampz. Vai ser 0 ou
289                        1, conforme seu uso. Nunca vai
290                        passar de 1 bit. Não pode ser maior conforme seu tipo de dado, mas pode
291                        ser maior.
292 */
293     unsigned eeprom : 1;
294     unsigned rampz  : 1;
295 } flags;
296
297 uint8_t buff[256];
298 uint8_t address_high; /*endereço alto para manipulação*/
299
300 uint8_t pagesz=0x80; /*128*/
301
302 uint8_t i;
303 uint8_t bootuart = 0;
304
305 uint8_t error_count = 0;
306
307 void (*app_start)(void) = 0x0000;
308
309
310 /* main program starts here */
311 int main(void)
312 {
313     uint8_t ch,ch2;
314     uint16_t w;
315
316 #ifdef WATCHDOG_MODS
317     ch = MCUSR;
318     MCUSR = 0;
319
320     WDTCSR |= _BV(WDCE) | _BV(WDE);
321     WDTCSR = 0;
322     /*Verifica se o WDT foi usado para reset*/
323     // Check if the WDT was used to reset, in which case we dont bootload and skip straight to
324     the code.
325     if (! (ch & _BV(EXTRF))) // if its a not an external reset...
326         app_start(); // skip bootload
327 #else
328     asm volatile("nop\n\t");
329 #endif
330
331     /* set pin direction for bootloader pin and enable pullup */
332     /* for ATmega128, two pins need to be initialized */
333 #ifdef __AVR_ATmega128__ /*Se definido ATmega128: */

```

```

328     BL_DDR &= ~_BV(BL0); /*Faz uma operação AND, e joga em BL_DDR*/
329     BL_DDR &= ~_BV(BL1); /*Preciso negar pra fazer o AND para tentar desligar o BIT apontado
330                             pelo valor de BL. Em seguida liga.
331                             */
332     BL_PORT |= _BV(BL0);
333     BL_PORT |= _BV(BL1);
334 #else
335     /* We run the bootloader regardless of the state of this pin. Thus, don't
336        put it in a different state than the other pins. --DAM, 070709
337        This also applies to Arduino Mega -- DC, 080930
338        BL_DDR &= ~_BV(BL);
339        BL_PORT |= _BV(BL);
340        */
341 #endif
342
343
344 #ifdef __AVR_ATmega128__
345     /* check which UART should be used for booting */
346     if(bit_is_clear(BL_PIN, BL0)) {
347         bootuart = 1;
348     }
349     else if(bit_is_clear(BL_PIN, BL1)) {
350         bootuart = 2;
351     }
352 #endif
353
354 #if defined __AVR_ATmega1280__
355     /* the mega1280 chip has four serial ports ... we could eventually use any of them, or not?
356        */
357     /* however, we don't wanna confuse people, to avoid making a mess, we will stick to RXD0,
358        TXD0 */
359     bootuart = 1;
360 #endif
361
362     /* check if flash is programmed already, if not start bootloader anyway */
363     if(pgm_read_byte_near(0x0000) != 0xFF) { /*Verifica se a memoria esta vazia*/
364
365 #ifdef __AVR_ATmega128__
366         /* no UART was selected, start application */
367         if(!bootuart) {
368             app_start();
369         }
370     #else
371         /* check if bootloader pin is set low */
372         /* we don't start this part neither for the m8, nor m168 */
373         //if(bit_is_set(BL_PIN, BL)) {
374         //    app_start();
375         //}
376     #endif
377
378 #ifdef __AVR_ATmega128__
379     /* no bootuart was selected, default to uart 0 */
380     if(!bootuart) {
381         bootuart = 1;
382     }
383 #endif
384
385     /* initialize UART(s) depending on CPU defined */
386     #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
387         if(bootuart == 1) {
388             UBRR0L = (uint8_t) (F_CPU/(BAUD_RATE*16L)-1);
389             UBRR0H = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
390             UCSR0A = 0x00;
391             UCSR0C = 0x06;
392             UCSR0B = _BV(TXEN0) | _BV(RXEN0); /*Transmissao esta no TXEN0*/
393         }
394         if(bootuart == 2) {
395             UBRR1L = (uint8_t) (F_CPU/(BAUD_RATE*16L)-1);
396             UBRR1H = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
397             UCSR1A = 0x00;
398             UCSR1C = 0x06;
399             UCSR1B = _BV(TXEN1) | _BV(RXEN1); /*Transmissao esta em TXEN1*/
400         }
401     #elif defined __AVR_ATmega163__
402         UBRR = (uint8_t) (F_CPU/(BAUD_RATE*16L)-1);
403         UBRRH = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
404         UCSRA = 0x00;
405         UCSRB = _BV(TXEN) | _BV(RXEN);
406     #elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__)
407
408     #ifdef DOUBLE_SPEED
409         UCSRA = (1<<U2X0); //Double speed mode USART0

```

```

410     UBRRL = (uint8_t) (F_CPU/(BAUD_RATE*8L)-1); /*Acertar os registradores para que a
411     comunicacao funcione. Desloca os bits para esquerda com
412     o valor definido na constante. F_CPU
413     determina a freq do controlador, transforma em bytes
414     e determina a velocidade de comunicacao
415     efetiva em que a CPU consegue trabalhar.
416     Se a freq da CPU for menor, recebe uma
417     quantidade de dados maior que o que consegue trabalhar.
418     A taxa tem que ser menor para que haja tempo
419     de receber os dados e trabalhar
420     */
421     UBRRL = (F_CPU/(BAUD_RATE*8L)-1) >> 8;
422 #else
423     UBRRL = (uint8_t) (F_CPU/(BAUD_RATE*16L)-1);
424     UBRRL = (F_CPU/(BAUD_RATE*16L)-1) >> 8; /* Determina em termos binarios a velocidade de
425     comunicacao
426     */
427 #endif
428
429     UCSRB = (1<<RXEN0) | (1<<TXEN0); /* Liga valores binarios dentro do registrador*/
430     UCSRC = (1<<UCSZ00) | (1<<UCSZ01);
431
432     /* Enable internal pull-up resistor on pin D0 (RX), in order
433     to suppress line noise that prevents the bootloader from
434     timing out (DAM: 20070509) */
435     DDRC &= ~_BV(PIND0);
436     PORTD |= _BV(PIND0);
437 #elif defined __AVR_ATmega8__
438     /* m8 */
439     UBRRL = (((F_CPU/BAUD_RATE)/16)-1)>>8; // set baud rate
440     UBRRL = (((F_CPU/BAUD_RATE)/16)-1);
441     UCSRB = (1<<RXEN) | (1<<TXEN); // enable Rx & Tx
442     UCSRC = (1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0); // Config USART; 8N1
443 #else
444     /* m16,m32,m169,m8515,m8535 */
445     UBRRL = (uint8_t) (F_CPU/(BAUD_RATE*16L)-1);
446     UBRRL = (F_CPU/(BAUD_RATE*16L)-1) >> 8;
447     UCSRA = 0x00;
448     UCSRC = 0x06;
449     UCSRB = _BV(TXEN) | _BV(RXEN);
450 #endif
451
452 #if defined __AVR_ATmega1280__
453     /* Enable internal pull-up resistor on pin D0 (RX), in order
454     to suppress line noise that prevents the bootloader from
455     timing out (DAM: 20070509) */
456     /* feature added to the Arduino Mega --DC: 080930 */
457     DDRE &= ~_BV(PINE0);
458     PORTE |= _BV(PINE0);
459 #endif
460
461     /* set LED pin as output */
462     LED_DDR |= _BV(LED); /*Define a porta do LED*/
463
464     /* flash onboard LED to signal entering of bootloader */
465 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
466     // 4x for UART0, 5x for UART1
467     flash_led(NUM_LED_FLASHES + bootuart);
468 #else
469     flash_led(NUM_LED_FLASHES); /*Conta o numero de vezes que o LED deve piscar*/
470 #endif
471
472     /* 20050803: by DojoCorp, this is one of the parts provoking the
473     system to stop listening, cancelled from the original */
474     //putch('\0');
475
476     /* forever loop */
477     for (;;) {
478
479         /* get character from UART */
480         ch = getch();
481
482         /* A bunch of if...else if... gives smaller code than switch...case ! */
483
484         /* Hello is anyone home ? */
485         if(ch=='0') {
486             nothing_response(); /*Nada para responder, não recebeu nada*/
487         }
488
489         /* Request programmer ID */

```

```

488     /* Not using PROGMEM string due to boot block in m128 being beyond 64kB boundry */
489     /* Would need to selectively manipulate RAMPZ, and it's only 9 characters anyway so who
cares. */
490     else if(ch=='1') {
491         if (getch() == ' ') {
492             putchar(0x14); /*
493                 Se recebeu 1:
494                 responde "AVR ISP"
495             */
496             putchar('A');
497             putchar('V');
498             putchar('R');
499             putchar(' ');
500             putchar('I');
501             putchar('S');
502             putchar('P');
503             putchar(0x10);
504         } else {
505             if (++error_count == MAX_ERROR_COUNT)
506                 app_start();
507         }
508     }
509
510
511     /* AVR ISP/STK500 board commands DON'T CARE so default nothing_response */
512     else if(ch=='@') {
513         ch2 = getch();
514         if (ch2>0x85) getch();
515         nothing_response();
516     }
517
518
519     /* AVR ISP/STK500 board requests */
520     else if(ch=='A') {
521         ch2 = getch();
522         if(ch2==0x80) byte_response(HW_VER); // Hardware version
523         else if(ch2==0x81) byte_response(SW_MAJOR); // Software major version
524         else if(ch2==0x82) byte_response(SW_MINOR); // Software minor version
525         else if(ch2==0x98) byte_response(0x03); // Unknown but seems to be required by avr
studio 3.56
526         else byte_response(0x00); // Covers various unnecessary responses we
don't care about
527     }
528
529
530     /* Device Parameters DON'T CARE, DEVICE IS FIXED */
531     else if(ch=='B') {
532         getNch(20);
533         nothing_response();
534     }
535
536
537     /* Parallel programming stuff DON'T CARE */
538     else if(ch=='E') {
539         getNch(5);
540         nothing_response();
541     }
542
543
544     /* P: Enter programming mode */
545     /* R: Erase device, don't care as we will erase one page at a time anyway. */
546     else if(ch=='P' || ch=='R') {
547         nothing_response();
548     }
549
550
551     /* Leave programming mode */
552     else if(ch=='Q') {
553         nothing_response();
554 #ifdef WATCHDOG_MODS
555         // via via watchdog (sneaky!)
556         WDTCR = _BV(WDE);
557         while (1); // 16 ms
558 #endif
559     }
560
561
562     /* Set address, little endian. EEPROM in bytes, FLASH in words */
563     /* Perhaps extra address bytes may be added in future to support > 128kB FLASH. */
564     /* This might explain why little endian was used here, big endian used everywhere else. */
565     else if(ch=='U') {
566         address.byte[0] = getch();
567         address.byte[1] = getch();
568         nothing_response();

```

```

569     }
570
571
572     /* Universal SPI programming command, disabled. Would be used for fuses and lock bits. */
573     else if(ch=='V') {
574         if (getch() == 0x30) {
575             getch();
576             ch = getch();
577             getch();
578             if (ch == 0) {
579                 byte_response(SIG1);
580             } else if (ch == 1) {
581                 byte_response(SIG2);
582             } else {
583                 byte_response(SIG3);
584             }
585         } else {
586             getNch(3);
587             byte_response(0x00);
588         }
589     }
590
591
592     /* Write memory, length is big endian and is in bytes */
593     else if(ch=='d') {
594         length.byte[1] = getch(); /*Joga os valores dentro da memoria*/
595         length.byte[0] = getch();
596         flags.eeprom = 0;
597         if (getch() == 'E') flags.eeprom = 1;
598         for (w=0;w<length.word;w++) {
599             buff[w] = getch(); // Store data in buffer, can't keep up
600             with serial data stream whilst programming pages
601         }
602         if (getch() == ' ') {
603             if (flags.eeprom) { //Write to EEPROM one byte at a time
604                 address.word <=<= 1;
605                 for(w=0;w<length.word;w++) {
606                     #if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__)
607                         while(EECR & (1<<EEPE));
608                         EEAR = (uint16_t)(void *)address.word;
609                         EEDR = buff[w];
610                         EECR |= (1<<EEMPE);
611                         EECR |= (1<<EEPE);
612                     #else
613                         eeprom_write_byte((void *)address.word,buff[w]);
614                         /* Código é jogado em uma memoria para ser executado */
615                         */
616                     #endif
617                     address.word++;
618                 }
619             }
620             else { //Write to FLASH one page at a time
621                 if (address.byte[1]>127) address_high = 0x01; //Only possible with m128,
622                 m256 will need 3rd address byte. FIXME
623                 else address_high = 0x00;
624                 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__)
625                     RAMPZ = address_high;
626                 #endif
627                 address.word = address.word << 1; //address * 2 -> byte location
628                 /* if ((length.byte[0] & 0x01) == 0x01) length.word++; //Even up an odd
629                 number of bytes */
630                 if ((length.byte[0] & 0x01)) length.word++; //Even up an odd number of bytes
631                 cli(); //Disable interrupts, just to be sure
632                 #if defined(EEPE)
633                     while(bit_is_set(EECR,EEPE)); //Wait for previous EEPROM writes to
634                     complete
635                 #else
636                     while(bit_is_set(EECR,EEWE)); //Wait for previous EEPROM writes to
637                     complete
638                 #endif
639                 asm volatile(
640                     "clr r17, 0\n\t" //page word count
641                     "lds r30,address\n\t" //Address of FLASH location (in bytes)
642                     "lds r31,address+1\n\t"
643                     "ldi r28,lo8(buff)\n\t" //Start of buffer array in RAM
644                     "ldi r29,hi8(buff)\n\t"
645                     "lds r24,length\n\t" //Length of data to be written (in bytes)
646                     "lds r25,length+1\n\t"
647                     "length_loop:\n\t" //Main loop, repeat for number of words in block
648                     "cpi r17,0x00\n\t" //If page_word_count=0 then erase page
649                     "brne no_page_erase\n\t"
650                     "wait_spm1:\n\t"
651                     "lds r16,%0\n\t" //Wait for previous spm to complete

```



```

648         "andi    r16,1           \n\t"
649         "cpi     r16,1           \n\t"
650         "breq    wait_spm1       \n\t"
651         "ldi     r16,0x03         \n\t" //Erase page pointed to by Z
652         "sts     %0,r16          \n\t"
653         "spm                     \n\t"
654     #ifdef __AVR_ATmega163__
655         ".word 0xFFFF             \n\t"
656         "nop                     \n\t"
657     #endif
658         "wait_spm2:              \n\t"
659         "lds     r16,%0           \n\t" //Wait for previous spm to complete
660         "andi    r16,1           \n\t"
661         "cpi     r16,1           \n\t"
662         "breq    wait_spm2       \n\t"
663
664         "ldi     r16,0x11         \n\t" //Re-enable RWW section
665         "sts     %0,r16          \n\t"
666         "spm                     \n\t"
667     #ifdef __AVR_ATmega163__
668         ".word 0xFFFF             \n\t"
669         "nop                     \n\t"
670     #endif
671         "no_page_erase:          \n\t"
672         "ld      r0,Y+            \n\t" //Write 2 bytes into page buffer
673         "ld      r1,Y+            \n\t"
674
675         "wait_spm3:              \n\t"
676         "lds     r16,%0           \n\t" //Wait for previous spm to complete
677         "andi    r16,1           \n\t"
678         "cpi     r16,1           \n\t"
679         "breq    wait_spm3       \n\t"
680         "ldi     r16,0x01         \n\t" //Load r0,r1 into FLASH page buffer
681         "sts     %0,r16          \n\t"
682         "spm                     \n\t"
683
684         "inc     r17              \n\t" //page_word_count++
685         "cpi     r17,%1           \n\t"
686         "brlo    same_page        \n\t" //Still same page in FLASH
687         "write_page:             \n\t"
688         "clr     r17              \n\t" //New page, write current one first
689         "wait_spm4:              \n\t"
690         "lds     r16,%0           \n\t" //Wait for previous spm to complete
691         "andi    r16,1           \n\t"
692         "cpi     r16,1           \n\t"
693         "breq    wait_spm4       \n\t"
694     #ifdef __AVR_ATmega163__
695         "andi    r30,0x80         \n\t" // m163 requires Z6:Z1 to be zero during page
write
696     #endif
697         "ldi     r16,0x05         \n\t" //Write page pointed to by Z
698         "sts     %0,r16          \n\t"
699         "spm                     \n\t"
700     #ifdef __AVR_ATmega163__
701         ".word 0xFFFF             \n\t"
702         "nop                     \n\t"
703         "ori     r30,0x7E         \n\t" // recover Z6:Z1 state after page write (had
to be zero during write)
704     #endif
705         "wait_spm5:              \n\t"
706         "lds     r16,%0           \n\t" //Wait for previous spm to complete
707         "andi    r16,1           \n\t"
708         "cpi     r16,1           \n\t"
709         "breq    wait_spm5       \n\t"
710         "ldi     r16,0x11         \n\t" //Re-enable RWW section
711         "sts     %0,r16          \n\t"
712         "spm                     \n\t"
713     #ifdef __AVR_ATmega163__
714         ".word 0xFFFF             \n\t"
715         "nop                     \n\t"
716     #endif
717         "same_page:              \n\t"
718         "adiw    r30,2            \n\t" //Next word in FLASH
719         "sbw     r24,2            \n\t" //length-2
720         "breq    final_write      \n\t" //Finished
721         "rjmp    length_loop      \n\t"
722         "final_write:            \n\t"
723         "cpi     r17,0            \n\t"
724         "breq    block_done       \n\t"
725         "adiw    r24,2            \n\t" //length+2, fool above check on length after
short page write
726         "rjmp    write_page       \n\t"
727         "block_done:            \n\t"
728         "clr     __zero_reg__     \n\t" //restore zero register

```

```

729 #if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__) ||
defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__)
730 : "m" (SPMCSR) : "M" (PAGE_SIZE) :
"r0", "r16", "r17", "r24", "r25", "r28", "r29", "r30", "r31"
731 #else
732 : "m" (SPMCR) : "M" (PAGE_SIZE) :
"r0", "r16", "r17", "r24", "r25", "r28", "r29", "r30", "r31"
733 #endif
734 );
735 /* Should really add a wait for RWW section to be enabled, don't actually need
it since we never */
736 /* exit the bootloader without a power cycle anyhow */
737 }
738 putch(0x14);
739 putch(0x10);
740 } else {
741     if (++error_count == MAX_ERROR_COUNT)
742         app_start();
743 }
744 }
745
746
747 /* Read memory block mode, length is big endian. */
748 else if(ch=='t') {
749     length.byte[1] = getch();
750     length.byte[0] = getch();
751 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
752     if (address.word > 0x7FFF) flags.rampz = 1; // No go with m256, FIXME
753     else flags.rampz = 0;
754 #endif
755     address.word = address.word << 1; // address * 2 -> byte location
756     if (getch() == 'E') flags.eeprom = 1;
757     else flags.eeprom = 0;
758     if (getch() == ' ') { // Command terminator
759         putch(0x14);
760         for (w=0; w < length.word; w++) { // Can handle odd and even lengths okay
761             if (flags.eeprom) { // Byte access EEPROM read
762 #if defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined(__AVR_ATmega328__)
763                 while(EEDR & (1<<EEPE));
764                 EEAR = (uint16_t)(void *)address.word;
765                 EEDR |= (1<<EERE);
766                 putch(EEDR);
767 #else
768                 putch(eeprom_read_byte((void *)address.word));
769 #endif
770                 address.word++;
771             }
772             else {
773                 if (!flags.rampz) putch(pgm_read_byte_near(address.word));
774 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
775                 else putch(pgm_read_byte_far(address.word + 0x10000));
776                 // Ummmm, yuck FIXME when m256 arrives
777 #endif
778                 address.word++;
779             }
780         }
781     }
782     putch(0x10);
783 }
784 }
785
786
787 /* Get device signature bytes */
788 else if(ch=='u') {
789     if (getch() == ' ') {
790         putch(0x14);
791         putch(SIG1);
792         putch(SIG2);
793         putch(SIG3);
794         putch(0x10);
795     } else {
796         if (++error_count == MAX_ERROR_COUNT)
797             app_start();
798     }
799 }
800
801
802 /* Read oscillator calibration byte */
803 else if(ch=='v') {
804     byte_response(0x00);
805 }
806
807
808 #if defined MONITOR

```

```

809
810     /* here come the extended monitor commands by Erik Lins */
811
812     /* check for three times exclamation mark pressed */
813     else if(ch=='!') {
814         ch = getch();
815         if(ch=='!') {
816             ch = getch();
817             if(ch=='!') {
818                 PGM_P welcome = "";
819                 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
820                     uint16_t extaddr;
821                 #endif
822                     uint8_t addr1, addrh;
823
824                 #ifdef CRUMB128
825                     welcome = "ATmegaBOOT / Crumb128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
826                 #elif defined PROBOMEGA128
827                     welcome = "ATmegaBOOT / PROBOmega128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
828                 #elif defined SAVVY128
829                     welcome = "ATmegaBOOT / Savvy128 - (C) J.P.Kyle, E.Lins - 050815\n\r";
830                 #elif defined __AVR_ATmega1280__
831                     welcome = "ATmegaBOOT / Arduino Mega - (C) Arduino LLC - 090930\n\r";
832                 #endif
833
834                 /* turn on LED */
835                 LED_DDR |= _BV(LED);
836                 LED_PORT &= ~ BV(LED);
837
838                 /* print a welcome message and command overview */
839                 for(i=0; welcome[i] != '\0'; ++i) {
840                     putchar(welcome[i]);
841                 }
842
843                 /* test for valid commands */
844                 for(;;) {
845                     putchar('\n');
846                     putchar('\r');
847                     putchar(':');
848                     putchar(' ');
849
850                     ch = getch();
851                     putchar(ch);
852
853                     /* toggle LED */
854                     if(ch == 't') {
855                         if(bit_is_set(LED_PIN,LED)) {
856                             LED_PORT &= ~ BV(LED);
857                             putchar('1');
858                         } else {
859                             LED_PORT |= _BV(LED);
860                             putchar('0');
861                         }
862                     }
863
864                     /* read byte from address */
865                     else if(ch == 'r') {
866                         ch = getch(); putchar(ch);
867                         addrh = gethex();
868                         addr1 = gethex();
869                         putchar('=');
870                         ch = *(uint8_t *) ((addrh << 8) + addr1);
871                         puthex(ch);
872                     }
873
874                     /* write a byte to address */
875                     else if(ch == 'w') {
876                         ch = getch(); putchar(ch);
877                         addrh = gethex();
878                         addr1 = gethex();
879                         ch = getch(); putchar(ch);
880                         ch = gethex();
881                         *(uint8_t *) ((addrh << 8) + addr1) = ch;
882                     }
883
884                     /* read from uart and echo back */
885                     else if(ch == 'u') {
886                         for(;;) {
887                             putchar(getch());
888                         }
889                     }
890                 #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
891                     /* external bus loop */
892                     else if(ch == 'b') {

```

```

893         putchar('b');
894         putchar('u');
895         putchar('s');
896         MCUCR = 0x80;
897         XMCRA = 0;
898         XMCRB = 0;
899         extaddr = 0x1100;
900         for(;;) {
901             ch = *(volatile uint8_t *)extaddr;
902             if(++extaddr == 0) {
903                 extaddr = 0x1100;
904             }
905         }
906     }
907 #endif
908
909     else if(ch == 'j') {
910         app_start();
911     }
912
913     } /* end of monitor functions */
914
915 }
916 }
917
918 /* end of monitor */
919 #endif
920 else if (++error_count == MAX_ERROR_COUNT) {
921     app_start();
922 }
923 } /* end of forever loop */
924
925 }
926
927
928 char gethexnib(void) {
929     char a;
930     a = getch(); putchar(a);
931     if(a >= 'a') {
932         return (a - 'a' + 0x0a);
933     } else if(a >= '0') {
934         return(a - '0');
935     }
936     return a;
937 }
938
939
940 char gethex(void) {
941     return (gethexnib() << 4) + gethexnib();
942 }
943
944
945 void puthex(char ch) {
946     char ah;
947
948     ah = ch >> 4;
949     if(ah >= 0x0a) {
950         ah = ah - 0x0a + 'a';
951     } else {
952         ah += '0';
953     }
954
955     ch &= 0x0f;
956     if(ch >= 0x0a) {
957         ch = ch - 0x0a + 'a';
958     } else {
959         ch += '0';
960     }
961
962     putchar(ah);
963     putchar(ch);
964 }
965
966
967 void putch(char ch) /*Recebe o caracter e joga na porta serial
968                     PROTOCOLO DE COMUNICAÇÃO SERIAL TRADICIONAL
969                     */
970 {
971     #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
972         if(bootuart == 1) {
973             while (!(UCSR0A & _BV(UDRE0))); /*Testar os valores*/
974             UDR0 = ch;
975         }
976         else if (bootuart == 2) {

```

```

977     while (!(UCSR1A & _BV(UDRE1)));
978     UDR1 = ch;
979 }
980 #elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
981     while (!(UCSR0A & _BV(UDRE0)));
982     UDR0 = ch;
983 #else
984     /* m8,16,32,169,8515,8535,163 */
985     while (!(UCSRA & _BV(UDRE)));
986     UDR = ch;
987 #endif
988 }
989
990
991 char getch(void)
992 {
993     #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
994         uint32_t count = 0;
995         if(bootuart == 1) {
996             while(!(UCSR0A & _BV(RXC0))) {
997                 /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
998                 /* HACKME:: here is a good place to count times*/
999                 count++;
1000                 if (count > MAX_TIME_COUNT)
1001                     app_start();
1002             }
1003
1004             return UDR0; /*Retorna o valor UDR0*/
1005         }
1006         else if(bootuart == 2) {
1007             while(!(UCSR1A & _BV(RXC1))) {
1008                 /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
1009                 /* HACKME:: here is a good place to count times*/
1010                 count++;
1011                 if (count > MAX_TIME_COUNT)
1012                     app_start();
1013             }
1014
1015             return UDR1;
1016         }
1017         return 0;
1018     #elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
1019         uint32_t count = 0;
1020         while(!(UCSR0A & _BV(RXC0))) {
1021             /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
1022             /* HACKME:: here is a good place to count times*/
1023             count++;
1024             if (count > MAX_TIME_COUNT)
1025                 app_start();
1026         }
1027         return UDR0;
1028     #else
1029         /* m8,16,32,169,8515,8535,163 */
1030         uint32_t count = 0;
1031         while(!(UCSRA & _BV(RXC))) {
1032             /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
1033             /* HACKME:: here is a good place to count times*/
1034             count++;
1035             if (count > MAX_TIME_COUNT)
1036                 app_start();
1037         }
1038         return UDR;
1039     #endif
1040 }
1041
1042
1043 void getNch(uint8_t count)
1044 {
1045     while(count--){
1046         #if defined(__AVR_ATmega128__) || defined(__AVR_ATmega1280__)
1047             if(bootuart == 1) {
1048                 while(!(UCSR0A & _BV(RXC0)));
1049                 UDR0;
1050             }
1051             else if(bootuart == 2) {
1052                 while(!(UCSR1A & _BV(RXC1)));
1053                 UDR1;
1054             }
1055         #elif defined(__AVR_ATmega168__) || defined(__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
1056             getch();
1057         #else
1058             /* m8,16,32,169,8515,8535,163 */
1059             /* 20060803 DojoCorp:: Addon coming from the previous Bootloader*/
1060             //while(!(UCSRA & _BV(RXC)));

```

```

1061         //UDR;
1062         getch(); // need to handle time out
1063     #endif
1064     }
1065 }
1066
1067
1068 void byte_response(uint8_t val)
1069 {
1070     if (getch() == ' ') {
1071         putch(0x14);
1072         putch(val);
1073         putch(0x10);
1074     } else {
1075         if (++error_count == MAX_ERROR_COUNT)
1076             app_start();
1077     }
1078 }
1079
1080
1081 void nothing_response(void)
1082 {
1083     if (getch() == ' ') {
1084         putch(0x14);
1085         putch(0x10);
1086     } else {
1087         if (++error_count == MAX_ERROR_COUNT)
1088             app_start();
1089     }
1090 }
1091
1092 void flash_led(uint8_t count)
1093 {
1094     while (count--) {
1095         LED_PORT |= BV(LED); /*Liga a porta*/
1096         _delay_ms(100); /*Dá um delay de 100 ms*/
1097         LED_PORT &= ~ BV(LED); /*Desliga a porta*/
1098         _delay_ms(100); /*Delay de 100 ms*/
1099     }
1100 }
1101
1102
1103 /* end of file ATmegaBOOT.c */
1104

```