```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>
#include <conio.h>
#include <ctype.h>
#include "timer.h"

#define VERSION         "1.0"
#define COPYRIGHT       "Copyright (c)2004 A-Z-E  www.a-z-e.de"

#define CODESIZE        1024
#define DATASIZE        128

#define DO_FILEREAD     1
#define DO_ERASE        2
#define DO_CHECK        4
#define DO_PROGRAM      8
#define DO_VERIFY       16
#define DO_READ         32
#define DO_PROTECT      64
#define DO_FILEWRITE    128

#define PROG_CODE       1
#define PROG_DATA       2
#define PROG_ID         4
#define PROG_CFG        8
#define PROG_MEM_MASK   15

#define PROG_VERIFY     16
#define PROG_CHECK      32
#define PROG_READ       64
#define PROG_PROTECT    128
#define PROG_MODE_MASK  240

//----------------------------
#define PIC_LOAD_CFG        0x00        // (16<-) switch to 0x2000 (0x4000)
#define PIC_INCREMENT       0x06        // next address

#define PIC_LOAD_CODE       0x02        // (16<-)
#define PIC_READ_CODE       0x04        // (->16)

#define PIC_START_PROG      0x08        // some devices ERASE & PROGRAM !

#define PIC_PROG_ONLY       0x18        //
#define PIC_END_PROG        0x0E        //

// Devices with ERASE function
//----------------------------
#define PIC_CHIP_ERASE      0x1F        // Chip Erase TYPE=1

#define PIC_CODE_ERASE      0x09        // Single mem Erase TYPE=1

#define PIC_BULK_SETUP1     0x01        // Erase Setup TYPE=0
#define PIC_BULK_SETUP2     0x07        // ChipErase if ADDR=2007

// Flash chips only
//----------------------------
#define PIC_LOAD_DATA       0x03        // (16<-)
#define PIC_READ_DATA       0x05        // (->16)

#define PIC_DATA_ERASE      0x0B        // works only if not protected

/*
** D0 = DATA out
** D1 = CLOCK
** D2 = LVP
** D3 = MCLR
** D4 = VCC on
**
** ACK= DATA in (DATA out must be H)
*/
```

```
73
74  #define DATA    1
75  #define CLOCK   2
76  #define PGM     4
77  #define MCLR    8
78  #define VCC     16
79
80  int pinstate, addr, PORT = 0x378;
81  int buffer[ 0x2100+DATASIZE ];
82  char l[81], *pos;
83  unsigned chks;
84  FILE *x;
85
86  void pin( int which, int state )
87  {
88      if( state )
89          pinstate |= which;
90      else
91          pinstate &= ~which;
92
93      outp( PORT, pinstate );
94  }
95
96  /****************************************
97  ** progmode()
98  **
99  ** put PIC into LVP programming mode
100 **
101 ****************************************/
102 void power( int state )
103 {
104     outp( PORT, pinstate = VCC );           // VCC off, all pins L
105     delay( 100 );
106
107     if( state )
108     {
109         pin( VCC, 0 );                      // Vcc on
110         us_delay( 5 );
111         pin( PGM, 1 );                      // activate LVP mode
112         us_delay( 5 );
113         pin( MCLR, 1 );                     // remove RESET
114     }
115     delay( 100 );
116 }
117
118 void command( int cmd )
119 {
120     int i;
121
122     for( i = 0; i < 6; ++i )
123     {
124         pin( CLOCK, 1 ); us_delay( 150 );
125         pin( DATA, cmd & 1 ); us_delay( 150 );
126         cmd >>= 1;
127         pin( CLOCK, 0 ); us_delay( 150 );
128     }
129     us_delay( 250 );
130 }
131
132 void writedata( int d )
133 {
134     int i;
135
136     d = ( d & 0x3FFF ) << 1;
137
138     for( i = 0; i < 16; ++i )
139     {
140         pin( CLOCK, 1 ); us_delay( 150 );
141         pin( DATA, d & 1 ); us_delay( 150 );
142         d >>= 1;
143         pin( CLOCK, 0 ); us_delay( 150 );
144     }
```

```
145         us_delay( 250 );
146     }
147
148     int readdata( void )
149     {
150         int i;
151         unsigned d = 0;
152
153         pin( DATA, 1 ); us_delay( 150 );        // make DATA readable
154
155         for( i = 0; i < 16; ++i )
156         {
157             pin( CLOCK, 1 ); us_delay( 150 );
158             if( ( inp( PORT+1 ) & 0x40 ) != 0 ) // ACK
159                 d |= 0x8000;
160             us_delay( 150 );
161             d >>= 1;
162             pin( CLOCK, 0 ); us_delay( 150 );
163         }
164         pin( DATA, 0 ); us_delay( 250 );
165         return d & 0x3FFF;
166     }
167
168     void do_load( int cmd, int d )
169     {
170         us_delay( 250 );
171         command( cmd );
172         writedata( d );
173     }
174
175     void increment( int steps )
176     {
177         while( steps-- )
178         {
179             command( PIC_INCREMENT );
180             ++addr;
181             us_delay( 500 );
182         }
183         if( ( addr & 0xFF ) ==  0 )
184             us_delay( 500 );
185     }
186
187     void progdelay( void )
188     {
189         us_delay( 20000 );              // should take max. 8ms + 5ms
190     }
191
192     int program( int mode )
193     {
194         int retry;
195
196         power( 1 );
197
198         if( mode & PROG_CODE )          // PROGRAM memory space
199         {
200             printf( "\n" );
201             for( addr = 0; addr < CODESIZE; )
202             {
203                 retry = 5;
204     RETRY1:
205                 do_load( PIC_LOAD_CODE, buffer[addr] );
206                 command( PIC_START_PROG );
207                 progdelay();
208                 printf( "\r%04X PROGRAM memory", addr );
209
210                 if( mode & PROG_VERIFY )
211                 {
212                     command( PIC_READ_CODE );
213                     if( readdata() != buffer[addr] )
214                     {
215                         if( --retry )
216                         {
```

```
217                         progdelay();
218                         goto RETRY1;
219                     }
220                     break;
221                 }
222             }
223
224             increment( 1 );
225         }
226         if( addr != CODESIZE )
227             goto ENDPROG;
228         mode &= ~PROG_CODE;
229     }
230
231     if( mode & PROG_DATA )          // DATA memory space
232     {
233         do_load( PIC_LOAD_DATA, buffer[0x2100] & 0xFF );
234         progdelay();
235
236         printf( "\n" );
237         for( addr = 0x2100; addr < 0x2100+DATASIZE; )
238         {
239             retry = 5;
240 RETRY2:
241             do_load( PIC_LOAD_DATA, buffer[addr] & 0xFF );
242             command( PIC_START_PROG );
243             progdelay();
244             printf( "\r%04X EEPROM memory", addr );
245
246             if( mode & PROG_VERIFY )
247             {
248                 command( PIC_READ_DATA );
249                 if( ( readdata() & 0xFF ) != ( buffer[addr] & 0xFF ) )
250                 {
251                     if( --retry )
252                     {
253                         progdelay();
254                         goto RETRY2;
255                     }
256                     break;
257                 }
258             }
259
260             increment( 1 );
261         }
262         if( addr != 0x2100+DATASIZE )
263             goto ENDPROG;
264         mode &= ~PROG_DATA;
265     }
266     if( mode & PROG_ID )            // USER ID memory space
267     {
268         do_load( PIC_LOAD_CFG, 0 );
269         progdelay();
270         printf( "\n" );
271
272         for( addr = 0x2000; addr < 0x2004; )
273         {
274             retry = 5;
275 RETRY3:
276             do_load( PIC_LOAD_CODE, buffer[addr] );
277             command( PIC_START_PROG );
278             progdelay();
279             printf( "\r%04X USER ID", addr );
280
281             if( mode & PROG_VERIFY )
282             {
283                 command( PIC_READ_CODE );
284                 if( readdata() != buffer[addr] )
285                 {
286                     if( --retry )
287                     {
288                         progdelay();
```

```
289                         goto RETRY3;
290                     }
291                     break;
292                 }
293             }
294
295             increment( 1 );
296         }
297         if( addr != 0x2004 )
298             goto ENDPROG;
299         mode &= ~PROG_ID;
300     }
301
302     if( mode & PROG_CFG )           // CONFIG memory space
303     {
304         int val, val2;
305
306         printf( "\n" );
307         if( addr < 0x2000 || addr > 0x2007 )
308         {
309             do_load( PIC_LOAD_CFG, 0 ); addr = 0x2000;
310             progdelay();
311         }
312         while( addr != 0x2007 )
313             increment( 1 );
314
315         retry = 5;
316 RETRY4:
317         val = buffer[addr];
318         if( mode & PROG_PROTECT )
319             val &= ~0x3D00;         // complete CODE & DATA protection
320
321         do_load( PIC_LOAD_CODE, val );
322         command( PIC_START_PROG );
323         progdelay();
324         printf( "\r%04X CONFIG WORD", addr );
325
326         if( mode & PROG_VERIFY )
327         {
328             power( 1 );
329             do_load( PIC_LOAD_CFG, 0 ); addr = 0x2000;
330             increment( 7 );
331
332             command( PIC_READ_CODE );
333             val2 = readdata();
334
335             if( val2 != val )
336             {
337                 if( --retry )
338                 {
339                     progdelay();
340                     goto RETRY4;
341                 }
342                 goto ENDPROG;
343             }
344         }
345         mode &= ~PROG_CFG;
346     }
347
348 ENDPROG:
349     power( 0 );
350     return mode;
351 }
352
353 int read_verify( int mode )
354 {
355     int val;
356
357     power( 1 );
358
359     if( mode & PROG_CODE )          // PROGRAM memory space
360     {
```

```
361             printf( "\n" );
362             for( addr = 0; addr < CODESIZE; )
363             {
364                 command( PIC_READ_CODE );
365                 printf( "\r%04X PROGRAM memory", addr );
366 
367                 val = readdata();
368                 if( mode & PROG_READ )
369                     buffer[addr] = val;
370                 else if( ( ( mode & PROG_VERIFY ) && val != buffer[addr] ) ||
371                          ( ( mode & PROG_CHECK ) && val != 0x3FFF ) )
372                 {
373                     power(0);
374                     return 0;
375                 }
376 
377                 increment( 1 );
378             }
379         }
380 
381         if( mode & PROG_DATA )          // DATA memory space
382         {
383             printf( "\n" );
384             for( addr = 0x2100; addr < 0x2100+DATASIZE; )
385             {
386                 command( PIC_READ_DATA );
387                 printf( "\r%04X EEPROM memory", addr );
388 
389                 val = readdata() & 0xFF;
390                 if( mode & PROG_READ )
391                     buffer[addr] = val;
392                 else if( ( ( mode & PROG_VERIFY ) && val != buffer[addr] ) ||
393                          ( ( mode & PROG_CHECK ) && val != 0xFF ) )
394                 {
395                     power(0);
396                     return 0;
397                 }
398 
399                 increment( 1 );
400             }
401         }
402 
403         if( mode & PROG_ID )            // USER ID memory space
404         {
405             printf( "\n" );
406             do_load( PIC_LOAD_CFG, 0 );
407             for( addr = 0x2000; addr < 0x2004; )
408             {
409                 command( PIC_READ_CODE );
410                 printf( "\r%04X USER ID", addr );
411 
412                 val = readdata();
413                 if( mode & PROG_READ )
414                     buffer[addr] = val;
415                 else if( ( ( mode & PROG_VERIFY ) && val != buffer[addr] ) ||
416                          ( ( mode & PROG_CHECK ) && val != 0x3FFF ) )
417                 {
418                     power(0);
419                     return 0;
420                 }
421 
422                 increment( 1 );
423             }
424         }
425 
426         if( mode & PROG_CFG )           // CONFIG memory space
427         {
428             printf( "\n" );
429             if( addr < 0x2000 || addr > 0x2007 )
430             {
431                 do_load( PIC_LOAD_CFG, 0 );
432                 addr = 0x2000;
```

```
433             }
434             while( addr != 0x2007 )
435                 increment( 1 );
436
437             command( PIC_READ_CODE );
438             printf( "\r%04X CONFIG WORD", addr );
439
440             val = readdata();
441             if( mode & PROG_READ )
442                 buffer[addr] = val;
443             else if( ( ( mode & PROG_VERIFY ) && val != buffer[addr] ) ||
444                         ( ( mode & PROG_CHECK ) && val != 0x3FFF ) )
445             {
446                 power(0);
447                 return 0;
448             }
449         }
450
451         power( 0 );
452         return 1;
453     }
454
455     void erase( void )
456     {
457         power( 1 );
458
459         do_load( PIC_LOAD_CFG, 0x3FFF );
460         increment( 7 );              // advance to 0x2007 = CFG WORD
461         command( PIC_BULK_SETUP1 );
462         command( PIC_BULK_SETUP2 );
463         command( PIC_START_PROG );
464         progdelay();
465         command( PIC_BULK_SETUP1 );
466         command( PIC_BULK_SETUP2 );
467         us_delay( 100 );
468
469         power( 0 );
470         us_delay( 10000 );
471     }
472
473     int getbyte( void )
474     {
475         int d1, d2;
476
477         d1 = *pos++ - '0';
478         if( d1 > 9 )
479             d1 -= 7;
480         d2 = *pos++ - '0';
481         if( d2 > 9 )
482             d2 -= 7;
483
484         d1 = ( d1 << 4 ) | d2;
485         chks += d1;
486         return d1;
487     }
488
489     int readhex( char *name )
490     {
491         int len, mode, val;
492
493         if( ( x = fopen( name, "r" ) ) == NULL )
494             return 0;
495
496         while( fgets( l, 80, x ) != NULL )
497         {
498             if( l[0] != ':' )
499                 continue;
500
501             pos = l+1;
502
503             chks = 0;
504
```

```
505             len = getbyte();
506
507             addr = getbyte() * 256;
508             addr |= getbyte();
509
510             mode = getbyte();
511             if( mode != 0 )
512                 continue;
513
514             while( len-- )
515             {
516                 val = getbyte();
517
518                 if( addr < 0x4200 )
519                 {
520                     if( addr & 1 )              // ODD (high) byte
521                     {
522                         buffer[ addr / 2 ] &= 0xFF;
523                         buffer[ addr / 2 ] |= ( val & 0x3F ) * 256;
524                     }
525                     else
526                     {
527                         buffer[ addr / 2 ] &= 0xFF00;
528                         buffer[ addr / 2 ] |= val;
529                     }
530                 }
531                 else
532                     buffer[ addr - 0x2100 ] = val;
533
534                 ++addr;
535             }
536
537             getbyte();
538             if( ( chks & 0xFF ) != 0 )
539             {
540                 fclose( x );
541                 return 0;
542             }
543         }
544
545         buffer[0x2007] |= 0x0280;       // CONFIG WORD reserved bit & LVP bit
546
547         fclose( x );
548         return 1;
549     }
550
551     int writehex( char *name )
552     {
553         int i;
554
555         if( ( x = fopen( name, "w" ) ) == NULL )
556             return 0;
557
558         // PROGRAM memory
559         for( addr = 0; addr < CODESIZE; addr += 8 )
560         {
561             fprintf( x, ":10%04X00", addr*2 );
562
563             chks = 0x10 + ((addr*2) >> 8) + ((addr*2) & 0xFF);
564             for( i = 0; i < 8; ++i )
565             {
566                 fprintf( x, "%02X", buffer[addr+i] & 0xFF );
567                 fprintf( x, "%02X", ( buffer[addr+i] >> 8 ) & 0xFF );
568                 chks += buffer[addr+i] & 0xFF;
569                 chks += ( buffer[addr+i] >> 8 ) & 0xFF;
570             }
571             fprintf( x, "%02X\n", ( 0x100 - ( chks & 0xFF ) ) & 0xFF );
572         }
573
574         // DATA memory
575         for( addr = 0x2100; addr < 0x2100+DATASIZE; addr += 16 )
576         {
```

```
577             fprintf( x, ":10%04X00", addr*2 );
578
579             chks = 0x10 + ((addr*2) >> 8) + ((addr*2) & 0xFF);
580             for( i = 0; i < 16; ++i )
581             {
582                 fprintf( x, "%02X", buffer[addr+i] & 0xFF );
583                 chks += buffer[addr+i] & 0xFF;
584             }
585             fprintf( x, "%02X\n", ( 0x100 - ( chks & 0xFF ) ) & 0xFF );
586         }
587
588         // USER ID
589         addr = 0x2000;
590         fprintf( x, ":08400000" );
591
592         chks = 0x48;
593         for( i = 0; i < 4; ++i )
594         {
595             fprintf( x, "%02X", buffer[addr+i] & 0xFF );
596             fprintf( x, "%02X", ( buffer[addr+i] >> 8 ) & 0xFF );
597             chks += buffer[addr+i] & 0xFF;
598             chks += ( buffer[addr+i] >> 8 ) & 0xFF;
599         }
600         fprintf( x, "%02X\n", ( 0x100 - ( chks & 0xFF ) ) & 0xFF );
601
602         // CHIP ID & CONFIGURATION WORD
603         addr = 0x2006;
604         fprintf( x, ":02400C00" );
605
606         chks = 0x4E;
607         for( i = 0; i < 2; ++i )
608         {
609             fprintf( x, "%02X", buffer[addr+i] & 0xFF );
610             fprintf( x, "%02X", ( buffer[addr+i] >> 8 ) & 0xFF );
611             chks += buffer[addr+i] & 0xFF;
612             chks += ( buffer[addr+i] >> 8 ) & 0xFF;
613         }
614         fprintf( x, "%02X\n", ( 0x100 - ( chks & 0xFF ) ) & 0xFF );
615
616         fprintf( x, ":00000001FF\n" );
617
618         fclose( x );
619         return 1;
620     }
621
622     void Header( void )
623     {
624         puts( "\nPIC16 Version " VERSION "  " COPYRIGHT "\n" );
625     }
626
627     volatile void Usage( void )
628     {
629         puts(   "Usage: PIC16 {cmds} [-arg]* <hexfile> [<hexfile2>]\n"
630                 "  where cmds are:\n"
631                 " E  ERASE              B  BLANKCHECK\n"
632                 " P  PROGRAM            V  VERIFY pic against buffer\n"
633                 " L  LOCK (PROTECT)     R  READ pic into <hexfile> or <hexfile2>\n"
634                 " A  ERASE + BLANKCHECK + PROGRAM + VERIFY + LOCK\n"
635                 "  and args are:\n"
636                 " -Px select printerport x=0:PRN 1=LPT1 2=LPT2 (default=LPT1)\n"
637                 " -C  exclude CODE memory from reading/programming\n"
638                 " -D  exclude DATA memory from reading/programming\n"
639                 " -I  exclude USER ID memory from reading/programming\n"
640                 " -X  exclude CONFIG word from reading/programming\n"
641                 " -V  don't verify each word after writing\n"
642                 "\n"
643                 " Example:\n"
644                 " PIC16 A file.hex     Erase, program file1.hex, verify, protect\n"
645                 " PIC16 V -D file.hex  Verify PIC against file.hex, ignoring DATA
                    memory\n"
646                 " PIC16 R -X file.hex  Read PIC into file.hex, leave CONFIG WORD out\n"
647                 " PIC16 EB             ERASE and BLANKCHECK pic\n"
```

```
        );
    exit( -1 );
}

volatile void leave( void )
{
    puts( "Press any key" );
    if( getch() == 0 )
        getch();
    exit( -1 );
}

int main( int ac, char *av[] )
{
    char *fname = NULL, *fname2 = NULL, *cmd = NULL;
    int val;
    int pmode, mode = 0;
    int progmode = PROG_CODE | PROG_DATA | PROG_ID | PROG_CFG | PROG_VERIFY;

    delay(0);

    for( addr = 0; addr < 0x2000; ++addr )
        buffer[addr] = 0x3FFF;
    for( addr = 0x2100; addr < 0x2200; ++addr )
        buffer[addr] = 0x00FF;
    for( addr = 0x2000; addr < 0x2004; ++addr )
        buffer[addr] = 0x3FFF;
    for( addr = 0x2004; addr < 0x2006; ++addr )
        buffer[addr] = 0;
    buffer[0x2006] = 0x07C0;
    buffer[0x2007] = 0x3FFF;

    power(0);

    while( --ac )
    {
        pos = *++av;

        if( pos[0] == '/' || pos[0] == '-' )
        {
            switch( toupper( pos[1] ) )
            {
            case 'P':
                switch( pos[2] )
                {
                case '0': PORT = 0x3BC; break;
                case '1': PORT = 0x378; break;
                case '2': PORT = 0x278; break;
                default : puts( "Unknown port number" ); Usage();
                }
                break;

            case 'D':       // exclude DATA from programming
                progmode &= ~PROG_DATA;
                break;

            case 'I':       // exclude ID from programming
                progmode &= ~PROG_ID;
                break;

            case 'C':       // exclude CODE from programming
                progmode &= ~PROG_CODE;
                break;

            case 'X':       // exclude CONFIGWORD from programming
                progmode &= ~PROG_CFG;
                break;

            case 'V':       // no inline verify
                progmode &= ~PROG_VERIFY;
                break;

```

```
720             default : puts( "Unknown switch" );Usage();
721             }
722         }
723         else if( cmd == NULL )
724             cmd = pos;
725         else if( fname == NULL )
726             fname = pos;
727         else if( fname2 == NULL )
728             fname2 = pos;
729         else
730         {
731             puts( "Too many arguments" );
732             Usage();
733         }
734     }
735
736     while( *cmd )
737     {
738         switch( toupper( *cmd ) )
739         {
740         case 'E': mode |= DO_ERASE; break;
741         case 'B': mode |= DO_CHECK; break;
742         case 'P': mode |= DO_PROGRAM|DO_FILEREAD; break;
743         case 'V': mode |= DO_VERIFY|DO_FILEREAD; break;
744         case 'R': mode |= DO_READ|DO_FILEWRITE; break;
745         case 'L': mode |= DO_PROTECT; break;
746         case 'A': mode |=
        DO_ERASE|DO_CHECK|DO_FILEREAD|DO_PROGRAM|DO_VERIFY|DO_PROTECT; break;
747         default : printf( "Unknown command '%c'\n", *cmd ); Usage();
748         }
749         ++cmd;
750     }
751
752     if( !fname && ( mode & (DO_FILEREAD|DO_FILEWRITE) ) != 0 )
753     {
754         puts( "Must specify a filename" );
755         Usage();
756     }
757     if( !fname2 && ( mode & (DO_FILEREAD|DO_FILEWRITE) ) ==
        (DO_FILEREAD|DO_FILEWRITE) )
758     {
759         puts( "Must specify 2 filenames" );
760         Usage();
761     }
762
763     Header();
764
765     printf( "Insert PIC and press any key (ESC to abort)\n" );
766     switch( getch() )
767     {
768     case 0: getch(); break;
769     case 0x1B:
770         return 1;
771     }
772
773     if( mode & DO_FILEREAD )
774     {
775         printf( "\rReading '%s'...\n", fname );
776         if( !readhex( fname ) )
777         {
778             printf( " - ERROR! Aborting...\n" );
779             leave();
780         }
781     }
782
783     power(1);
784     do_load( PIC_LOAD_CFG, 0 );
785     increment( 7 );
786     command( PIC_READ_CODE );
787     val = readdata();
788     power(0);
789
```

```
790         if( ( val & 0x3D00 ) != 0x3D00 &&       // some protection active ?
791             ( mode & DO_PROGRAM ) != 0 &&       // we want to write ?
792             ( mode & DO_ERASE ) == 0 )          // and erase not activated ?
793         {
794             printf( "\rChip is protected! Erasing...\n" );
795             erase();
796         }
797         else if( mode & DO_ERASE )
798         {
799             printf( "\rErasing...\n" );
800             erase();
801         }
802
803         if( mode & DO_CHECK )
804         {
805             printf( "\rBlank Checking..." );
806
807             if( !read_verify( ( progmode & PROG_MEM_MASK ) | PROG_CHECK ) )
808             {
809                 printf( " - Not empty!\n", addr );
810                 leave();
811             }
812         }
813
814         if( mode & DO_PROGRAM )
815         {
816             int retries = 5;
817
818             printf( "\rProgramming..." );
819             pmode = (progmode & ~PROG_PROTECT) | ( (mode & DO_VERIFY) ? 0 : PROG_VERIFY );
820             while( ( ( pmode = program( pmode ) ) & PROG_MEM_MASK ) != 0 )
821             {
822                 if( !--retries )
823                 {
824                     printf( " - ERROR! Aborting...\n", addr );
825                     leave();
826                 }
827             }
828         }
829
830         if( mode & DO_VERIFY )
831         {
832             printf( "\rVerifying..." );
833             if( !read_verify( ( progmode & PROG_MEM_MASK ) | PROG_VERIFY ) )
834             {
835                 printf( " - ERROR! Aborting...\n", addr );
836                 leave();
837             }
838         }
839
840         if( mode & DO_READ )
841         {
842             printf( "\rReading..." );
843             read_verify( ( progmode & PROG_MEM_MASK ) | PROG_READ );
844         }
845
846         if( mode & DO_PROTECT )
847         {
848             int retries = 3;
849
850             printf( "\rProtecting..." );
851             pmode = PROG_CFG | PROG_PROTECT | PROG_VERIFY;
852             while( ( ( pmode = program( pmode ) ) & PROG_MEM_MASK ) != 0 )
853             {
854                 if( !--retries )
855                 {
856                     printf( " - ERROR! Aborting...\n", addr );
857                     leave();
858                 }
859             }
860         }
861
```

```
862         if( mode & DO_FILEWRITE )
863         {
864             if( ( mode & DO_FILEREAD ) != 0 && fname2 )
865                 fname = fname2;
866 
867             printf( "\rWriting to file '%s'...\n", fname );
868             if( !writehex( fname ) )
869             {
870                 printf( " - ERROR! Aborting...\n" );
871                 leave();
872             }
873         }
874 
875         printf( "\nDONE - BYE!\n" );
876         return 0;
877     }
878 
```