```
#------------------------------------------------------------------------
# Makefile for antBASIC on the Raspberry Pi
#
# NOTICE  You need to pre-install "libreadline-dev", "atp" and
#         "ghostscript (includes ps2pdf)" packages before "make std|max".
#------------------------------------------------------------------------
# PREP    $ sudo apt install libreadline-dev atp ghostscript
#------------------------------------------------------------------------
# BUILD   $ make                --> main applications only
#         $ make std            --> main applications plus PDFs
#         $ make max            --> main and test applications plus PDFS
# INSTALL $ sudo make install   --> install main applications and MAN file
#------------------------------------------------------------------------
# BAREMETALHACK.COM --> public domain
#------------------------------------------------------------------------

CFLAGS := -Wall -Wextra
CFLAGS += -DREADLINE ### Build antBASIC with readline extension
CFLAGS += -DPI ### Build antBASIC targeted for Raspberry Pi
#CFLAGS += -DDEBUG ### Turn on in the case of debugging
LDLIBS := -lreadline ### Link with GNU Readline library
ATPFLAGS := -t 8 -date `date "+%Y-%m-%d"`
ATPFLAGS += -T A4 -M 1,1,2,2 cm ### Paper size and margin definition
#ATPFLAGS += -N ### With line number
GROFFFLAGS := -mandoc -Tps
GROFFFLAGS += -dpaper=a4 ### Paper size
TARGET_BIN := /usr/local/bin
TARGET_MAN := /usr/local/share/man/man1
TARGET_DOC := ./DOCUMENTS

HEADERS = byteword.h token.h bcode.h program.h container.h function.h eval.h basic
.h debug.h escape.h
APPS = antbasic antcalib
TESTAPPS = test_token test_bcode test_prog test_eval test_assign
PDFS = $(TARGET_DOC)/src.pdf $(TARGET_DOC)/antbasic.pdf

.SECONDEXPANSION:

min: antbasic antcalib
std: antbasic antcalib pdfs
max: test_token test_bcode test_prog test_eval test_assign antbasic antcalib pdfs

test_token : $$@.c $(HEADERS) token.c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^)

test_bcode: $$@.c $(HEADERS) token.c bcode.c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^)

test_prog: $$@.c $(HEADERS) token.c bcode.c program.c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^)

test_eval: $$@.c $(HEADERS) token.c bcode.c container.c function.c eval.c pi_gpio.
c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^)

test_assign: $$@.c $(HEADERS) token.c bcode.c container.c function.c eval.c pi_gpi
o.c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^)

antbasic: $(HEADERS) main.c token.c bcode.c program.c container.c function.c eval.
c basic.h basic.c pi_gpio.c main.c
	gcc $(CFLAGS) -o $@ $(filter %.c, $^) -lreadline

antcalib: antcalib.c
	gcc $(CFLAGS) -o $@ antcalib.c

pdfs:
	atp $(ATPFLAGS) Makefile byteword.h debug.h escape.h token.h token.c test_
token.c bcode.h bcode.c test_bcode.c program.h program.c test_prog.c container.h c
ontainer.c function.h function.c eval.h eval.c test_eval.c test_assign.c basic.h b
asic.c pi_gpio.h pi_gpio.c main.c antcalib.c | ps2pdf - - > $(TARGET_DOC)/src.pdf
	groff $(GROFFFLAGS) antbasic.1 | ps2pdf - - > $(TARGET_DOC)/antbasic.pdf

install: antbasic antcalib
	install -v -t $(TARGET_BIN) antbasic antcalib
	install -v -D -t $(TARGET_MAN) antbasic.1

clean:
	rm -f *.o $(APPS) $(TESTAPPS)

dist-clean:
	rm -f *.o $(APPS) $(TESTAPPS) $(PDFS)
	rm -f $(TARGET_BIN)/antbasic $(TARGET_BIN)/antcalib
	rm -f $(TARGET_MAN)/antbasic.1
```

```
//
//  b y t e w o r d . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _BYTEWORD_H_
#define _BYTEWORD_H_

#include <stdint.h>             // uint8_t, int16_t

//
// Type definitions
//

typedef uint8_t byte_t;
typedef int16_t word_t;

#endif  // _BYTEWORD_H_
```

```
//
//  d e b u g . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _DEBUG_H_
#define _DEBUG_H_

#include <stdio.h>      // FILE{}

//
// Global variable
//

extern FILE *Debug;

//
// Macro definition for debugging
//

#ifdef DEBUG
    #define DMSG(...) { \
        fprintf(Debug, "%s() ", __FUNCTION__); \
        fprintf(Debug, __VA_ARGS__); \
        fprintf(Debug, "\n"); \
    }
#else
    #define DMSG(...) {}
#endif

#endif // _DEBUG_H_
```

```
//
//  e s c a p e . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _ESCAPE_H_
#define _ESCAPE_H_

//
// ANSI escape sequences
//

#define ESCAPE_HOME             "\e[1;1H"
#define ESCAPE_CLS              "\e[2J"

#define ESCAPE_DEFAULT          "\e[0m"
#define ESCAPE_BOLD             "\e[1m"
#define ESCAPE_RED              "\e[31m"
#define ESCAPE_GREEN            "\e[32m"
#define ESCAPE_YELLOW           "\e[33m"
#define ESCAPE_BLUE             "\e[34m"
#define ESCAPE_MAGENTA          "\e[35m"
#define ESCAPE_CYAN             "\e[36m"
#define ESCAPE_WHITE            "\e[37m"
#define ESCAPE_LRED             (ESCAPE_BOLD ESCAPE_RED)
#define ESCAPE_LGREEN           (ESCAPE_BOLD ESCAPE_GREEN)
#define ESCAPE_LYELLOW          (ESCAPE_BOLD ESCAPE_YELLOW)
#define ESCAPE_LBLUE            (ESCAPE_BOLD ESCAPE_BLUE)
#define ESCAPE_LMAGENTA         (ESCAPE_BOLD ESCAPE_MAGENTA)
#define ESCAPE_LCYAN            (ESCAPE_BOLD ESCAPE_CYAN)
#define ESCAPE_LWHITE           (ESCAPE_BOLD ESCAPE_WHITE)

#endif  // _ESCAPE_H_
```

```
//
//  t o k e n . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _TOKEN_H_
#define _TOKEN_H_

#include "byteword.h"           // word_t

//
// Token constants
//

#define TOKEN_MAXBUF            512     // Source text buffer capacity
#define TOKEN_MAXSTR            256     // Maximum length of string
#define TOKEN_MAXKEY            10      // Maximum length of keyword
#define TOKEN_MAXDLM            2       // Maximum length of delimiter

//
// Token types
//

#define TOKEN_TYPE_EOL          0       // End Of Line
#define TOKEN_TYPE_NUMBER10     1       // Decimal number
#define TOKEN_TYPE_NUMBER16     2       // Hexa-decimal number
#define TOKEN_TYPE_STRING       3       // String
#define TOKEN_TYPE_VARIABLE     4       // Variable or array name
#define TOKEN_TYPE_DELIMITER    5       // Delimiter
#define TOKEN_TYPE_KEYWORD      6       // Keyword

//
// Index codes for variable
//

#define TOKEN_VAR_A     0
#define TOKEN_VAR_B     1
#define TOKEN_VAR_C     2
#define TOKEN_VAR_D     3
#define TOKEN_VAR_E     4
#define TOKEN_VAR_F     5
#define TOKEN_VAR_G     6
#define TOKEN_VAR_H     7
#define TOKEN_VAR_I     8
#define TOKEN_VAR_J     9
#define TOKEN_VAR_K     10
#define TOKEN_VAR_L     11
#define TOKEN_VAR_M     12
#define TOKEN_VAR_N     13
#define TOKEN_VAR_O     14
#define TOKEN_VAR_P     15
#define TOKEN_VAR_Q     16
#define TOKEN_VAR_R     17
#define TOKEN_VAR_S     18
#define TOKEN_VAR_T     19
#define TOKEN_VAR_U     20
#define TOKEN_VAR_V     21
#define TOKEN_VAR_W     22
#define TOKEN_VAR_X     23
#define TOKEN_VAR_Y     24
#define TOKEN_VAR_Z     25

//
// Index codes for delimiter
//

#define TOKEN_EQUAL     0       // ==
#define TOKEN_ASSIGN    1       // =
#define TOKEN_GEQUAL    2       // >=
#define TOKEN_GTHAN     3       // >
#define TOKEN_LEQUAL    4       // <=
#define TOKEN_LTHAN     5       // <
#define TOKEN_NEQUAL    6       // !=
#define TOKEN_AT        7       // @
#define TOKEN_LBRACK    8       // [
#define TOKEN_RBRACK    9       // ]
#define TOKEN_LPAREN    10      // (
#define TOKEN_RPAREN    11      // )
#define TOKEN_COLON     12      // :
#define TOKEN_SCOLON    13      // ;
#define TOKEN_COMMA     14      // ,
#define TOKEN_MUL       15      // *
#define TOKEN_DIV       16      // /
#define TOKEN_MOD       17      // %
#define TOKEN_ADD       18      // +
#define TOKEN_SUB       19      // -
#define TOKEN_AND       20      // &
#define TOKEN_OR        21      // |
#define TOKEN_INQ       22      // ?

//
// Index codes for reserved keyword
//

#define TOKEN_CLEAR     0
#define TOKEN_CLS       1
#define TOKEN_DELETE    2
#define TOKEN_DIM       3
#define TOKEN_DUMP      4
#define TOKEN_END       5
#define TOKEN_FOR       6
#define TOKEN_FREE      7
#define TOKEN_GOSUB     8
#define TOKEN_GOTO      9
#define TOKEN_HELP      10
#define TOKEN_HEX2      11
#define TOKEN_HEX4      12
#define TOKEN_IF        13
#define TOKEN_INPUT     14
#define TOKEN_LIST      15
#define TOKEN_NEW       16
#define TOKEN_NEXT      17
#define TOKEN_PRINT     18
#define TOKEN_REM       19
#define TOKEN_RENUM     20
#define TOKEN_RETURN    21
#define TOKEN_RND       22
#define TOKEN_RUN       23
#define TOKEN_TO        24

// --- OS dependent section ---

#define TOKEN_FILES     25
#define TOKEN_LOAD      26
#define TOKEN_MERGE     27
#define TOKEN_MSLEEP    28
#define TOKEN_SAVE      29
#define TOKEN_USLEEP    30

#ifdef PI
#define TOKEN_IN        31
```

```
#define TOKEN_OUT       32
#define TOKEN_OUTHZ     33
#endif // PI

//
// Result codes
//

#define TOKEN_SUCCESS           0
#define TOKEN_MISSINGQUOTE      1
#define TOKEN_TOOLONGSTR        2
#define TOKEN_TOOLONGKEY        3
#define TOKEN_ILLEGALCHAR       4
#define TOKEN_ILLNUMBER         5
#define TOKEN_ILLSTRINRG        6
#define TOKEN_KEYNOTFOUND       7
#define TOKEN_DLMNOTFOUND       8
#define TOKEN_UNKNOWN           9

//
// Type definition
//

typedef struct {
    int type;
    union {
        int idx;
     word_t num;
       char str[ TOKEN_MAXSTR + 1 ];
    } body;
} token_t;

//
// Function prototypes
//

  void token_init(void);
  void token_source(char *);
   int token_read(token_t *);
char * token_keyword(int);
char * token_delimiter(int);
char * token_result(int);

#ifdef DEBUG
void token_dump(token_t *);
#endif // DEBUG

#endif // _TOKEN_H_
```

```
//
//  t o k e n . c
//
//  BAREMETALHACK.COM --> public domain
//

#ifdef DEBUG            // token_dump()
#include <stdio.h>      // FILE{}, fprintf()
#include <string.h>     // strlen()
#endif

#include "token.h"
#include "debug.h"

//
// Constants for character type
//

#define CT_EOL              1       // end of line: NULL | '\n'
#define CT_SPACE            2       // ' ' | '\t'
#define CT_DELIMITER        4       // !()*+,-/:;<=>@[]&|?
#define CT_ALPHABET         8       // A-Z | a-z
#define CT_BINARY           16      // 0-1
#define CT_DECIMAL          32      // 0-9
#define CT_HEXADECIMAL      64      // 0-9 | A-F | a-f

//
// Local type definition
//

typedef struct {
    char * string;
       int index;
} table_t;

//
// File scope variables
//

static unsigned char Ctype[ 256 ];
static char *Srcptr;
static char Next;

static table_t
Keywords[] = {
    { "CLEAR",  TOKEN_CLEAR },
    { "CLS",    TOKEN_CLS },
    { "DELETE", TOKEN_DELETE },
    { "DIM",    TOKEN_DIM },
    { "DUMP",   TOKEN_DUMP },
    { "END",    TOKEN_END },
    { "FOR",    TOKEN_FOR },
    { "FREE",   TOKEN_FREE },
    { "GOSUB",  TOKEN_GOSUB },
    { "GOTO",   TOKEN_GOTO },
    { "HELP",   TOKEN_HELP },
    { "HEX2",   TOKEN_HEX2 },
    { "HEX4",   TOKEN_HEX4 },
    { "IF",     TOKEN_IF },
    { "INPUT",  TOKEN_INPUT },
    { "LIST",   TOKEN_LIST },
    { "NEW",    TOKEN_NEW },
    { "NEXT",   TOKEN_NEXT },
    { "PRINT",  TOKEN_PRINT },
    { "REM",    TOKEN_REM },
    { "RENUM",  TOKEN_RENUM },
    { "RETURN", TOKEN_RETURN },
    { "RND",    TOKEN_RND },
    { "RUN",    TOKEN_RUN },
    { "TO",     TOKEN_TO },

    { "FILES",  TOKEN_FILES },
    { "LOAD",   TOKEN_LOAD },
    { "MERGE",  TOKEN_MERGE },
    { "MSLEEP", TOKEN_MSLEEP },
    { "SAVE",   TOKEN_SAVE },
    { "USLEEP", TOKEN_USLEEP },

#ifdef PI
    { "IN",     TOKEN_IN },
    { "OUT",    TOKEN_OUT },
    { "OUTHZ",  TOKEN_OUTHZ },
#endif // PI
    { 0, 0 }                        // table end mark
};

static table_t
Delimiters[] = {
    { "==", TOKEN_EQUAL },
    { "=",  TOKEN_ASSIGN },
    { ">=", TOKEN_GEQUAL },
    { ">",  TOKEN_GTHAN },
    { "<=", TOKEN_LEQUAL },
    { "<",  TOKEN_LTHAN },
    { "!=", TOKEN_NEQUAL },
    { "@",  TOKEN_AT },
    { "[",  TOKEN_LBRACK },
    { "]",  TOKEN_RBRACK },
    { "(",  TOKEN_LPAREN },
    { ")",  TOKEN_RPAREN },
    { ":",  TOKEN_COLON },
    { ";",  TOKEN_SCOLON },
    { ",",  TOKEN_COMMA },
    { "*",  TOKEN_MUL },
    { "/",  TOKEN_DIV },
    { "%",  TOKEN_MOD },
    { "+",  TOKEN_ADD },
    { "-",  TOKEN_SUB },
    { "&",  TOKEN_AND },
    { "|",  TOKEN_OR },
    { "?",  TOKEN_INQ },
    { 0, 0 }                        // table end mark
};

static char *
Results[] = {
    "TOKEN \"ok\"",                 // TOKEN_SUCCESS
    "TOKEN \"missing quotation\"",  // TOKEN_MISSINGQUOTE
    "TOKEN \"too long string\"",    // TOKEN_TOOLONGSTR
    "TOKEN \"too long keyword\"",   // TOKEN_TOOLONGKEY
    "TOKEN \"illegal character\"",  // TOKEN_ILLCHAR
    "TOKEN \"illegal number\"",     // TOKEN_ILLNUMBER
    "TOKEN \"illegal string\"",     // TOKEN_ILLSTRING
    "TOKEN \"keyword not found\"",  // TOKEN_KEYNOTFOUND
    "TOKEN \"delimiter not found\"",// TOKEN_DLMNOTFOUND
    "TOKEN \"unknown token\""       // TOKEN_UNKNOWN
};

//
// Procedures and functions
//
```

```
void
token_init(void)
{
    int i;

    for (i = 0; i < 256; i++) Ctype[ i ] = 0;

    Ctype[ 0 ] = Ctype[ '\n' ] = CT_EOL;

    Ctype[ ' ' ] = Ctype[ '\t' ] = CT_SPACE;

    Ctype[ '!' ] = Ctype[ '%' ] = Ctype[ '&' ] = Ctype[ '(' ] = \
    Ctype[ ')' ] = Ctype[ '*' ] = Ctype[ '+' ] = Ctype[ ',' ] = \
    Ctype[ '-' ] = Ctype[ '/' ] = Ctype[ ':' ] = Ctype[ ';' ] = \
    Ctype[ '<' ] = Ctype[ '=' ] = Ctype[ '>' ] = Ctype[ '@' ] = \
    Ctype[ '[' ] = Ctype[ ']' ] = Ctype[ '?' ] = Ctype[ '|' ] = CT_DELIMITER;

    Ctype[ 'A' ] = Ctype[ 'B' ] = Ctype[ 'C' ] = Ctype[ 'D' ] = \
    Ctype[ 'E' ] = Ctype[ 'F' ] = Ctype[ 'G' ] = Ctype[ 'H' ] = \
    Ctype[ 'I' ] = Ctype[ 'J' ] = Ctype[ 'K' ] = Ctype[ 'L' ] = \
    Ctype[ 'M' ] = Ctype[ 'N' ] = Ctype[ 'O' ] = Ctype[ 'P' ] = \
    Ctype[ 'Q' ] = Ctype[ 'R' ] = Ctype[ 'S' ] = Ctype[ 'T' ] = \
    Ctype[ 'U' ] = Ctype[ 'V' ] = Ctype[ 'W' ] = Ctype[ 'X' ] = \
    Ctype[ 'Y' ] = Ctype[ 'Z' ] = CT_ALPHABET;

    Ctype[ 'a' ] = Ctype[ 'b' ] = Ctype[ 'c' ] = Ctype[ 'd' ] = \
    Ctype[ 'e' ] = Ctype[ 'f' ] = Ctype[ 'g' ] = Ctype[ 'h' ] = \
    Ctype[ 'i' ] = Ctype[ 'j' ] = Ctype[ 'k' ] = Ctype[ 'l' ] = \
    Ctype[ 'm' ] = Ctype[ 'n' ] = Ctype[ 'o' ] = Ctype[ 'p' ] = \
    Ctype[ 'q' ] = Ctype[ 'r' ] = Ctype[ 's' ] = Ctype[ 't' ] = \
    Ctype[ 'u' ] = Ctype[ 'v' ] = Ctype[ 'w' ] = Ctype[ 'x' ] = \
    Ctype[ 'y' ] = Ctype[ 'z' ] = CT_ALPHABET;

    Ctype[ '0' ] = Ctype[ '1' ] = Ctype[ '2' ] = Ctype[ '3' ] = \
    Ctype[ '4' ] = Ctype[ '5' ] = Ctype[ '6' ] = Ctype[ '7' ] = \
    Ctype[ '8' ] = Ctype[ '9' ] = CT_DECIMAL;

    Ctype[ '0' ] |= Ctype[ '1' ] |= CT_BINARY;

    Ctype[ '0' ] |= Ctype[ '1' ] |= Ctype[ '2' ] |= Ctype[ '3' ] |= \
    Ctype[ '4' ] |= Ctype[ '5' ] |= Ctype[ '6' ] |= Ctype[ '7' ] |= \
    Ctype[ '8' ] |= Ctype[ '9' ] |= CT_HEXADECIMAL;

    Ctype[ 'A' ] |= Ctype[ 'B' ] |= Ctype[ 'C' ] |= Ctype[ 'D' ] |= \
    Ctype[ 'E' ] |= Ctype[ 'F' ] |= CT_HEXADECIMAL;

    Ctype[ 'a' ] |= Ctype[ 'b' ] |= Ctype[ 'c' ] |= Ctype[ 'd' ] |= \
    Ctype[ 'e' ] |= Ctype[ 'f' ] |= CT_HEXADECIMAL;
}

static void
read_char(void)
{
    Next = *Srcptr++;
    return;
}

void
token_source(char *ptr)
{
    Srcptr = ptr;
    read_char();        // do pre-fetch first
    return;
}

static int
check_digit(int base)
{
    if (base == 16) {
        if (Ctype[ (int) Next ] & CT_HEXADECIMAL) {
            if (Next >= 'a')
                return (Next - 'a' + 10);
            else if (Next >= 'A')
                return (Next - 'A' + 10);
            else
                return (Next - '0');
        } else
            return -1;
    } else if (base == 10) {
        if (Ctype[ (int) Next ] & CT_DECIMAL)
            return (Next - '0');
        else
            return -1;
    } else if (base == 2) {
        if (Ctype[ (int) Next ] & CT_BINARY)
            return (Next - '0');
        else
            return -1;
    } else
        return -1;
}

static word_t
number(token_t *tok)
{
    word_t base = 10, digit, val = 0;

    if (Next == '0') {
        read_char();
        if (Next == 'x' || Next == 'X') {
            base = 16;
            read_char();
        } else
            base = 10;
    }
    while ((digit = check_digit(base)) >= 0) {
        val = (val * base) + digit;
        read_char();
    };

    if (base == 16)
        tok->type = TOKEN_TYPE_NUMBER16;
    else
        tok->type = TOKEN_TYPE_NUMBER10;
    tok->body.num = val;
    return TOKEN_SUCCESS;
}

static int
string(token_t *tok)
{
    int cnt, high, low;
    char *str;

    str = tok->body.str;
    cnt = 0;
    read_char();
    while (Next && Next != '\"' && cnt < TOKEN_MAXSTR) {
        if (Next == '\\') {
            read_char();
            switch(Next) {
                case 'a':
```

```
                    Next = '\007';
                    break;

                case 'b':
                    Next = '\010';
                    break;

                case 't':
                    Next = '\011';
                    break;

                case 'n':
                    Next = '\012';
                    break;

                case 'r':
                    Next = '\015';
                    break;

                case 'e':
                    Next = '\033';
                    break;

                case 'x':
                    read_char();
                    if (! (Ctype[ (int) Next ] & CT_HEXADECIMAL))
                        return TOKEN_ILLNUMBER;
                    high = check_digit(16);
                    read_char();
                    if (! (Ctype[ (int) Next ] & CT_HEXADECIMAL))
                        return TOKEN_ILLNUMBER;
                    low = check_digit(16);
                    Next = high * 16 + low;;
                    break;

                case '\\':
                    Next = '\\';
                    break;

                default:                // unknown character
                    Next = '?';
                    break;
            }
        }
        *str++ = Next;
        cnt++;
        read_char();
    }
    *str = 0;

    if (Next == '\"')
        read_char();
    else if (cnt == TOKEN_MAXSTR)
        return TOKEN_TOOLONGSTR;
    else
        return TOKEN_MISSINGQUOTE;

    tok->type = TOKEN_TYPE_STRING;
    return TOKEN_SUCCESS;
}

static int
delimiter(token_t *tok)
{
    int idx;

    switch(Next) {
        case '=':
            read_char();
            if (Next == '=') {
                read_char();
                idx = TOKEN_EQUAL;
            } else
                idx = TOKEN_ASSIGN;
            break;

        case '>':
            read_char();
            if (Next == '=') {
                read_char();
                idx = TOKEN_GEQUAL;
            } else
                idx = TOKEN_GTHAN;
            break;

        case '<':
            read_char();
            if (Next == '=') {
                read_char();
                idx = TOKEN_LEQUAL;
            } else
                idx = TOKEN_LTHAN;
            break;

        case '!':
            read_char();
            if (Next == '=') {
                read_char();
                idx = TOKEN_NEQUAL;
            } else
                return TOKEN_UNKNOWN;
            break;

        case '@':
            read_char();
            idx = TOKEN_AT;
            break;

        case '[':
            read_char();
            idx = TOKEN_LBRACK;
            break;

        case ']':
            read_char();
            idx = TOKEN_RBRACK;
            break;

        case '(':
            read_char();
            idx = TOKEN_LPAREN;
            break;

        case ')':
            read_char();
            idx = TOKEN_RPAREN;
            break;

        case ':':
            read_char();
            idx = TOKEN_COLON;
            break;
```

```
        case ';':
            read_char();
            idx = TOKEN_SCOLON;
            break;

        case ',':
            read_char();
            idx = TOKEN_COMMA;
            break;

        case '*':
            read_char();
            idx = TOKEN_MUL;
            break;

        case '/':
            read_char();
            idx = TOKEN_DIV;
            break;

        case '%':
            read_char();
            idx = TOKEN_MOD;
            break;

        case '+':
            read_char();
            idx = TOKEN_ADD;
            break;

        case '-':
            read_char();
            idx = TOKEN_SUB;
            break;
        case '&':
            read_char();
            idx = TOKEN_AND;
            break;

        case '|':
            read_char();
            idx = TOKEN_OR;
            break;

        case '?':
            read_char();
            idx = TOKEN_INQ;
            break;
    }
    tok->type = TOKEN_TYPE_DELIMITER;
    tok->body.idx = idx;
    return TOKEN_SUCCESS;
}

static int
search_table(table_t *tbl, char *key)
{
    int idx = 0;
    char *src, *dst;

    while (tbl->string) {
        src = key;
        dst = tbl->string;
        while (*src && *dst) {
            if (*dst != *src)
                break;
            src++;
            dst++;
        }
        if ((*src == 0) && (*dst == 0))
            return idx;
        idx++;
        tbl++;
    }
    return -1;
}

static int keyword(token_t *tok)
{
    int cnt, idx;
    char key[ TOKEN_MAXKEY + 1 ], *ptr;

    ptr = key;
    cnt = 0;
    while (Ctype[ (int) Next ] & CT_ALPHABET || \
           Ctype[ (int) Next ] & CT_DECIMAL) {
        if (Next >= 'a' && Next <= 'z')
            Next -= 0x20;
        cnt++;
        if (cnt == (TOKEN_MAXKEY + 1))
            return TOKEN_TOOLONGKEY;
        *ptr++ = Next;
        read_char();
    }
    *ptr = 0;

    if (cnt == 1) {
        tok->type = TOKEN_TYPE_VARIABLE;
        tok->body.idx = key[ 0 ] - 'A';
        return TOKEN_SUCCESS;
    }

    idx = search_table(Keywords, key);
    if (idx < 0)
        return TOKEN_KEYNOTFOUND;

    tok->type = TOKEN_TYPE_KEYWORD;
    tok->body.idx = Keywords[ idx ].index;
    return TOKEN_SUCCESS;
}

int
token_read(token_t *tok)
{
    while (Ctype[ (int) Next ] & CT_SPACE)      // Skip blank
        read_char();
    if (Ctype[ (int) Next ] & CT_EOL) {
        tok->type = TOKEN_TYPE_EOL;
        return TOKEN_SUCCESS;
    }

    if (Ctype[ (int) Next ] & CT_DECIMAL)
        return number(tok);
    else if (Ctype[ (int) Next ] & CT_ALPHABET)
        return keyword(tok);
    else if (Ctype[ (int) Next ] & CT_DELIMITER)
        return delimiter(tok);
    else if (Next == '\"')
        return string(tok);
    else
        return TOKEN_UNKNOWN;
}
```

```
char *
token_keyword(int idx)
{
    return Keywords[ idx ].string;
}

char *
token_delimiter(int idx)
{
    return Delimiters[ idx ].string;
}

char *
token_result(int idx)
{
    return Results[ idx ];
}

#ifdef DEBUG
void
token_dump(token_t *tok)
{
    int i, len;

    fprintf(Debug, "    token type %d : ", tok->type);
    switch (tok->type) {
        case TOKEN_TYPE_EOL:
            fprintf(Debug, "EOL\n");
            break;

        case TOKEN_TYPE_NUMBER10:
            fprintf(Debug, "number %d (0x%0hX)\n", tok->body.num, tok->body.num);
            break;

        case TOKEN_TYPE_NUMBER16:
            fprintf(Debug, "number 0x%0hX (%d)\n", tok->body.num, tok->body.num);
            break;

        case TOKEN_TYPE_STRING:
            len = strlen(tok->body.str);
            fprintf(Debug, "string (strlen = %d)\n", len);
            for (i = 0; i < (len + 1); i++) {
                if ((i % 16) == 0)
                    fprintf(Debug, "        ");
                fprintf(Debug, "%02hhX ", (unsigned char) tok->body.str[ i ]);
                if ((i % 16) == 15)
                    fprintf(Debug, "\n");
            }
            if ((i % 16) != 15)
                fprintf(Debug, "\n");
            break;

        case TOKEN_TYPE_VARIABLE:
            fprintf(Debug, "variable %c\n", tok->body.idx + 'A');
            break;

        case TOKEN_TYPE_KEYWORD:
            fprintf(Debug, "keyword \"%s\"\n", Keywords[ tok->body.idx ].string);
            break;

        case TOKEN_TYPE_DELIMITER:
            fprintf(Debug, "delimiter \"%s\"\n", Delimiters[ tok->body.idx ].strin
g);
            break;
    }
}

#endif // DEBUG
```

```
#include <stdio.h>      // FILE{}, stdin, stderr, fgets(), printf()
#include "token.h"

FILE *Debug;

int
main()
{
    char buffer[ 256 ];
    token_t tok;
    int ret;

    Debug = stderr;
    token_init();
    while (fgets(buffer, 256, stdin) != NULL) {
        if (*buffer == 0 || *buffer == '\n')
            break;
        printf("===== %s", buffer);
        token_source(buffer);
        while (1) {
            ret = token_read(&tok);
            if (ret != TOKEN_SUCCESS) {
                fprintf(Debug, "*** Token error: %s\n", token_result(ret));
                break;
            }
#ifdef DEBUG
            token_dump(&tok);
#endif
            if (tok.type == TOKEN_TYPE_EOL)
                break;
        }
        fprintf(Debug, "\n");
    }

    return 0;
}
```

```
//
//  b c o d e . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _BCODE_H_
#define _BCODE_H_

#include "token.h"

//
// Bytecode constants
//

#define BCODE_MAXSIZE           512     // Maximum byte size in single line

//-------------------------------------------------
//
// Bytecode instruction codes
//
//   Type 0       EOL: 0x00
//   Type 1  Number10: 0x01, low, high
//   Type 2  Number16: 0x02, low, high
//   Type 3    String: 0x03, len (includes null terminator), ch1, ch2, ..., 0
//   Type 4  Variable: 0x10 - 0x29
//   Type 5 Delimiter: 0x40 - 0x56
//   Type 6   Keyword: 0x80 - 0xA1
//
//-------------------------------------------------

#define BCODE_TYPE_EOL          TOKEN_TYPE_EOL
#define BCODE_TYPE_NUMBER10     TOKEN_TYPE_NUMBER10
#define BCODE_TYPE_NUMBER16     TOKEN_TYPE_NUMBER16
#define BCODE_TYPE_STRING       TOKEN_TYPE_STRING
#define BCODE_TYPE_VARIABLE     TOKEN_TYPE_VARIABLE
#define BCODE_TYPE_DELIMITER    TOKEN_TYPE_DELIMITER
#define BCODE_TYPE_KEYWORD      TOKEN_TYPE_KEYWORD

#define BCODE_EOL               BCODE_TYPE_EOL
#define BCODE_NUMBER10          BCODE_TYPE_NUMBER10
#define BCODE_NUMBER16          BCODE_TYPE_NUMBER16
#define BCODE_STRING            BCODE_TYPE_STRING
#define BCODE_VARIABLE          0x10
#define BCODE_DELIMITER         0x40
#define BCODE_KEYWORD           0x80

#define BCODE_VAR_A             (BCODE_VAIRABLE | ('A' - 'A'))
#define BCODE_VAR_B             (BCODE_VAIRABLE | ('B' - 'A'))
#define BCODE_VAR_C             (BCODE_VAIRABLE | ('C' - 'A'))
#define BCODE_VAR_D             (BCODE_VAIRABLE | ('D' - 'A'))
#define BCODE_VAR_E             (BCODE_VAIRABLE | ('E' - 'A'))
#define BCODE_VAR_F             (BCODE_VAIRABLE | ('F' - 'A'))
#define BCODE_VAR_G             (BCODE_VAIRABLE | ('G' - 'A'))
#define BCODE_VAR_H             (BCODE_VAIRABLE | ('H' - 'A'))
#define BCODE_VAR_I             (BCODE_VAIRABLE | ('I' - 'A'))
#define BCODE_VAR_J             (BCODE_VAIRABLE | ('J' - 'A'))
#define BCODE_VAR_K             (BCODE_VAIRABLE | ('K' - 'A'))
#define BCODE_VAR_L             (BCODE_VAIRABLE | ('L' - 'A'))
#define BCODE_VAR_M             (BCODE_VAIRABLE | ('M' - 'A'))
#define BCODE_VAR_N             (BCODE_VAIRABLE | ('N' - 'A'))
#define BCODE_VAR_O             (BCODE_VAIRABLE | ('O' - 'A'))
#define BCODE_VAR_P             (BCODE_VAIRABLE | ('P' - 'A'))
#define BCODE_VAR_Q             (BCODE_VAIRABLE | ('Q' - 'A'))
#define BCODE_VAR_R             (BCODE_VAIRABLE | ('R' - 'A'))
#define BCODE_VAR_S             (BCODE_VAIRABLE | ('S' - 'A'))
#define BCODE_VAR_T             (BCODE_VAIRABLE | ('T' - 'A'))
#define BCODE_VAR_U             (BCODE_VAIRABLE | ('U' - 'A'))
#define BCODE_VAR_V             (BCODE_VAIRABLE | ('V' - 'A'))
#define BCODE_VAR_W             (BCODE_VAIRABLE | ('W' - 'A'))
#define BCODE_VAR_X             (BCODE_VAIRABLE | ('X' - 'A'))
#define BCODE_VAR_Y             (BCODE_VAIRABLE | ('Y' - 'A'))
#define BCODE_VAR_Z             (BCODE_VAIRABLE | ('Z' - 'A'))

#define BCODE_EQUAL             (BCODE_DELIMITER | TOKEN_EQUAL)
#define BCODE_ASSIGN            (BCODE_DELIMITER | TOKEN_ASSIGN)
#define BCODE_GEQUAL            (BCODE_DELIMITER | TOKEN_GEQUAL)
#define BCODE_GTHAN             (BCODE_DELIMITER | TOKEN_GTHAN)
#define BCODE_LEQUAL            (BCODE_DELIMITER | TOKEN_LEQUAL)
#define BCODE_LTHAN             (BCODE_DELIMITER | TOKEN_LTHAN)
#define BCODE_NEQUAL            (BCODE_DELIMITER | TOKEN_NEQUAL)
#define BCODE_EXCL              (BCODE_DELIMITER | TOKEN_EXCL)
#define BCODE_AT                (BCODE_DELIMITER | TOKEN_AT)
#define BCODE_LBRACK            (BCODE_DELIMITER | TOKEN_LBRACK)
#define BCODE_RBRACK            (BCODE_DELIMITER | TOKEN_RBRACK)
#define BCODE_LPAREN            (BCODE_DELIMITER | TOKEN_LPAREN)
#define BCODE_RPAREN            (BCODE_DELIMITER | TOKEN_RPAREN)
#define BCODE_COLON             (BCODE_DELIMITER | TOKEN_COLON)
#define BCODE_SCOLON            (BCODE_DELIMITER | TOKEN_SCOLON)
#define BCODE_COMMA             (BCODE_DELIMITER | TOKEN_COMMA)
#define BCODE_MUL               (BCODE_DELIMITER | TOKEN_MUL)
#define BCODE_DIV               (BCODE_DELIMITER | TOKEN_DIV)
#define BCODE_MOD               (BCODE_DELIMITER | TOKEN_MOD)
#define BCODE_ADD               (BCODE_DELIMITER | TOKEN_ADD)
#define BCODE_SUB               (BCODE_DELIMITER | TOKEN_SUB)
#define BCODE_AND               (BCODE_DELIMITER | TOKEN_AND)
#define BCODE_OR                (BCODE_DELIMITER | TOKEN_OR)
#define BCODE_INQ               (BCODE_DELIMITER | TOKEN_INQ)

#define BCODE_CLEAR             (BCODE_KEYWORD | TOKEN_CLEAR)
#define BCODE_CLS               (BCODE_KEYWORD | TOKEN_CLS)
#define BCODE_DELETE            (BCODE_KEYWORD | TOKEN_DELETE)
#define BCODE_DIM               (BCODE_KEYWORD | TOKEN_DIM)
#define BCODE_DUMP              (BCODE_KEYWORD | TOKEN_DUMP)
#define BCODE_END               (BCODE_KEYWORD | TOKEN_END)
#define BCODE_FOR               (BCODE_KEYWORD | TOKEN_FOR)
#define BCODE_FREE              (BCODE_KEYWORD | TOKEN_FREE)
#define BCODE_GOSUB             (BCODE_KEYWORD | TOKEN_GOSUB)
#define BCODE_GOTO              (BCODE_KEYWORD | TOKEN_GOTO)
#define BCODE_HELP              (BCODE_KEYWORD | TOKEN_HELP)
#define BCODE_HEX2              (BCODE_KEYWORD | TOKEN_HEX2)
#define BCODE_HEX4              (BCODE_KEYWORD | TOKEN_HEX4)
#define BCODE_IF                (BCODE_KEYWORD | TOKEN_IF)
#define BCODE_INPUT             (BCODE_KEYWORD | TOKEN_INPUT)
#define BCODE_LIST              (BCODE_KEYWORD | TOKEN_LIST)
#define BCODE_NEW               (BCODE_KEYWORD | TOKEN_NEW)
#define BCODE_NEXT              (BCODE_KEYWORD | TOKEN_NEXT)
#define BCODE_PRINT             (BCODE_KEYWORD | TOKEN_PRINT)
#define BCODE_REM               (BCODE_KEYWORD | TOKEN_REM)
#define BCODE_RENUM             (BCODE_KEYWORD | TOKEN_RENUM)
#define BCODE_RETURN            (BCODE_KEYWORD | TOKEN_RETURN)
#define BCODE_RND               (BCODE_KEYWORD | TOKEN_RND)
#define BCODE_RUN               (BCODE_KEYWORD | TOKEN_RUN)
#define BCODE_TO                (BCODE_KEYWORD | TOKEN_TO)

#define BCODE_FILES             (BCODE_KEYWORD | TOKEN_FILES)
#define BCODE_LOAD              (BCODE_KEYWORD | TOKEN_LOAD)
#define BCODE_MERGE             (BCODE_KEYWORD | TOKEN_MERGE)
#define BCODE_MSLEEP            (BCODE_KEYWORD | TOKEN_MSLEEP)
#define BCODE_SAVE              (BCODE_KEYWORD | TOKEN_SAVE)
#define BCODE_USLEEP            (BCODE_KEYWORD | TOKEN_USLEEP)
```

```
#ifdef PI
#define BCODE_IN                (BCODE_KEYWORD | TOKEN_IN)
#define BCODE_OUT               (BCODE_KEYWORD | TOKEN_OUT)
#define BCODE_OUTHZ             (BCODE_KEYWORD | TOKEN_OUTHZ)
#endif // PI

//
// Result codes
//

#define BCODE_SUCCESS           TOKEN_SUCCESS
#define BOCDE_MISSINGQUOTE      TOKEN_MISSINGQUOTE
#define BCODE_TOOLONGSTR        TOKEN_TOOLONGSTR
#define BCODE_TOOLONGKEY        TOKEN_TOOLONGKEY
#define BCODE_ILLEGALCHAR       TOKEN_ILLEGALCHAR
#define BCODE_ILLNUMBER         TOKEN_ILLNUMBER
#define BCODE_ILLSTRINRG        TOKEN_ILLSTRING
#define BCODE_KEYNOTFOUND       TOKEN_KEYNOTFOUND
#define BCODE_DLMNOTFOUND       TOKEN_DLMNOTFOUND
#define BCODE_UNKNOWN           TOKEN_UNKNOWN
#define BCODE_SIZEOVER          (BCODE_UNKNOWN + 1)

//
// Type definition
//

typedef struct {
      int pos;
      int inst;
      int type;
      int idx;
   word_t num;
 byte_t * str;
} bcode_t;

//
// Function prototypes
//

   void bcode_start(byte_t *);
   void bcode_setpc(int);
    int bcode_getpc(void);
   void bcode_skip(void);
    int bcode_next(void);
    int bcode_nextiseol(void);
    int bcode_nextisnum(void);
    int bcode_nextisstr(void);
    int bcode_nextisvar(void);
    int bcode_nextiskey(void);
    int bcode_nextisdlm(void);
    int bcode_twoahead(void);
   void bcode_read(bcode_t *);
    int bcode_compile(byte_t *);
 char * bcode_result(int);

#ifdef DEBUG
   void bcode_display(bcode_t *);
   void bcode_dump(byte_t *);
#endif // DEBUG

#endif // _BCODE_H_
```

```
//
//  b c o d e . c
//
//  BAREMETALHACK.COM --> public domain
//

#ifdef DEBUG                // bcode_dump()
#include <stdio.h>          // FILE{}, fprintf()
#endif

#include "bcode.h"

//
// Filescope variables
//

static byte_t *Code;        // points start of code area
static int Pc;              // program counter

//
// Bytecode functions
//

void
bcode_start(byte_t *ptr)
{
    Code = ptr;
    Pc = 0;
}

void
bcode_setpc(int where)
{
    Pc = where;
}

int
bcode_getpc(void)
{
    return Pc;
}

void
bcode_skip(void)
{
    bcode_t b;

    bcode_read(&b);
}

int
bcode_next(void)
{
    return (int) Code[ Pc ];
}

int
bcode_nextiseol(void)
{
    return Code[ Pc ] == BCODE_EOL ? 1 : 0;
}

int
bcode_nextisnum(void)
{
    if ((Code[ Pc ] == BCODE_NUMBER10) || (Code[ Pc ] == BCODE_NUMBER16))
        return 1;
    else
        return 0;
}

int
bcode_nextisstr(void)
{
    return Code[ Pc ] == BCODE_STRING ? 1 : 0;
}

int
bcode_nextisvar(void)
{
    if ((Code[ Pc ] >= BCODE_VARIABLE) && (Code[ Pc ] < BCODE_DELIMITER))
        return 1;
    else
        return 0;
}

int
bcode_nextisdlm(void)
{
    if ((Code[ Pc ] >= BCODE_DELIMITER) && (Code[ Pc ] < BCODE_KEYWORD))
        return 1;
    else
        return 0;
}

int
bcode_nextiskey(void)
{
    return Code[ Pc ] >= BCODE_KEYWORD ? 1 : 0;
}

int
bcode_twoahead(void)
{
    return (int) Code[ Pc + 1 ];
}

void
bcode_read(bcode_t *b)
{
    int len;
    word_t *ptr;

    b->pos = Pc;
    b->inst = Code[ Pc ];
    if (Code[ Pc ] < BCODE_VARIABLE)
        b->type = b->idx = Code[ Pc ];
    else if (Code[ Pc ] >= BCODE_VARIABLE && Code[ Pc ] < BCODE_DELIMITER) {
        b->type = BCODE_TYPE_VARIABLE;
        b->idx = Code[ Pc ] - BCODE_VARIABLE;
    } else if (Code[ Pc ] >= BCODE_DELIMITER && Code[ Pc ] < BCODE_KEYWORD) {
        b->type = BCODE_TYPE_DELIMITER;
        b->idx = Code[ Pc ] - BCODE_DELIMITER;
    } else {
        b->type = BCODE_TYPE_KEYWORD;
        b->idx = Code[ Pc ] - BCODE_KEYWORD;
    }
    b->num = 0;
    b->str = 0;
    Pc++;
    switch (b->type) {
        case BCODE_TYPE_NUMBER10:
```

```
        case BCODE_TYPE_NUMBER16:
            ptr = (word_t*) &Code[ Pc ];
            b->num = *ptr;
            Pc += 2;
            break;

        case BCODE_TYPE_STRING:
            b->idx = len = Code[ Pc++ ];
            b->str = &Code[ Pc ];
            Pc += len;
            break;
    }
}

int bcode_compile(byte_t *start) {
    byte_t *bptr, *lptr, *sptr;
    int len, ret;
    token_t tok;

    bptr = start;
    while (1) {
        ret = token_read(&tok);
        if (ret != TOKEN_SUCCESS)
            return -ret;
        switch (tok.type) {
            case TOKEN_TYPE_EOL:
                *bptr++ = BCODE_EOL;
                return bptr - start;

            case TOKEN_TYPE_NUMBER10:
                *bptr++ = BCODE_NUMBER10;
                *bptr++ = tok.body.num & 0xFF;
                *bptr++ = (tok.body.num >> 8) & 0xFF;
                break;

            case TOKEN_TYPE_NUMBER16:
                *bptr++ = BCODE_NUMBER16;
                *bptr++ = tok.body.num & 0xFF;
                *bptr++ = (tok.body.num >> 8) & 0xFF;
                break;

            case TOKEN_TYPE_STRING:
                *bptr++ = BCODE_STRING;
                lptr = bptr++;
                sptr = (byte_t *) tok.body.str;
                len = 0;
                while (*sptr) {
                    *bptr++ = *sptr++;
                    len++;
                }
                *bptr++ = 0;
                len++;
                *lptr = len;
                break;

            case TOKEN_TYPE_VARIABLE:
                *bptr++ = BCODE_VARIABLE + tok.body.idx;
                break;

            case TOKEN_TYPE_DELIMITER:
                *bptr++ = BCODE_DELIMITER + tok.body.idx;
                break;

            case TOKEN_TYPE_KEYWORD:
                *bptr++ = BCODE_KEYWORD + tok.body.idx;
                break;
        }
        if ((bptr - start + 3) > BCODE_MAXSIZE)
            return -BCODE_SIZEOVER;
    }
    return bptr - start;
}

char *
bcode_result(int idx)
{
    static char
        msg_ok[] = "BCODE \"ok\"",
        msg_sizeover[] = "BCODE \"size over\"";

    if (idx == BCODE_SUCCESS)
        return msg_ok;
    else if (idx == BCODE_SIZEOVER)
        return msg_sizeover;
    else
        return token_result(idx);
}

#ifdef DEBUG
extern FILE *Debug;

void
bcode_display(bcode_t *b)
{
    fprintf(Debug, "    0d%05d : ", b->pos);
    fprintf(Debug, "bcode 0x%02X : ", b->inst);
    fprintf(Debug, "type %1d : ", b->type);
    fprintf(Debug, "idx %2d | ", b->idx);
    if (b->type == BCODE_TYPE_NUMBER10)
        fprintf(Debug, "%d (0x%04hX)\n", b->num, b->num);
    else if (b->type == BCODE_TYPE_NUMBER16)
        fprintf(Debug, "0x%04hX (%d)\n", b->num, b->num);
    else if (b->type == BCODE_TYPE_STRING)
        fprintf(Debug, "\"%s\"\n", (char*) b->str);
    else if (b->type == BCODE_TYPE_VARIABLE)
        fprintf(Debug, "%c\n", 'A' + b->idx);
    else if (b->type == BCODE_TYPE_KEYWORD)
        fprintf(Debug, "%s\n", token_keyword(b->idx));
    else if (b->type == BCODE_TYPE_DELIMITER)
        fprintf(Debug, "%s\n", token_delimiter(b->idx));
    else if (b->type == BCODE_TYPE_EOL)
        fprintf(Debug, "EOL\n");
}

void
bcode_dump(byte_t *bytes)
{
    bcode_t b;

    fprintf(Debug, "[Compiled bytecodes]\n");
    bcode_start(bytes);
    while (1) {
        bcode_read(&b);
        bcode_display(&b);
        if (b.type == BCODE_TYPE_EOL)
            break;
    }
}
#endif
```

```
#include <stdio.h>      // FILE{}, fgets()
#include "bcode.h"

FILE *Debug;

int
main()
{
    char buffer[ 256 ];
    byte_t bytes[ 256 ];
    int i, ret;

    Debug = stderr;
    token_init();
    while (fgets(buffer, 256, stdin) != NULL) {
        if (*buffer == 0 || *buffer == '\n')
            break;
        fprintf(Debug, "===== %s", buffer);
        token_source(buffer);
        ret = bcode_compile(bytes);
        if (ret < 0) {
            ret = -ret;
            fprintf(Debug, "***** Error: %s\n", bcode_result(ret));
            break;
        }
        fprintf(Debug, ">>>>> Compiled bytecodes [ %d bytes ]\n", ret);
        for (i = 0; i < ret; i++) {
            if ((i % 16) == 0)
                fprintf(Debug, "    ");
            fprintf(Debug, "%02X ", bytes[ i ]);
            if ((i % 16) == 15)
                fprintf(Debug, "\n");
        }
        if ((i % 16) != 15)
            fprintf(Debug, "\n");
#ifdef DEBUG
            bcode_dump(bytes);
#endif // DEBUG
    }
    return ret;
}
```

```
//
//  p r o g r a m . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _PROGRAM_H_
#define _PROGRAM_H_

#include <stdio.h>              // FILE{}
#include "bcode.h"

//
// Program constants
//

#define PROG_MAXSIZE        30000
#define PROG_MAXLINE        2000
#define PROG_MAXRELOC       100
#define PROG_BLANKLINE      0x7FFF
#define PROG_BLANKDATA      0xFF
#define PROG_CMDLINE        PROG_MAXSIZE

#define PROG_LISTCOLOR      1
#define PROG_LISTPLAIN      0

//
// Result codes
//

#define PROG_SUCCESS        0
#define PROG_LINENOTFOUND   1
#define PROG_TOOMANYLINES   2
#define PROG_SIZEOVER       3
#define PROG_TOOMANYRELOCS  4

//
// Type definitions
//

typedef struct {
    int num;
    int add;
    int len;
} line_t;

typedef struct {
    int idx;
    int add;
} reloc_t;

//
// Global variables
//

extern byte_t Program[];
extern int Psize;
extern line_t Lines[];
extern int Lsize;
extern int Pnum;

//
// Function prototypes
//

  void prog_init(void);
   int prog_list(FILE *, int, int, int);
   int prog_search(int);
   int prog_delete(int, int);
   int prog_insert(int, byte_t *, int);
  void prog_cmdline(byte_t *, int);
   int prog_renum(int, int);
char * prog_result(int);
  void prog_dumpbytes(void);
  void prog_dumplines(void);

#endif // _PROGRAM_H_
```

```
//
//  p r o g r a m . c
//
//  BAREMETALHACK.COM --> public domain
//

#include "program.h"
#include "bcode.h"
#include "escape.h"

//
// Global variables
//

  byte_t Program[ PROG_MAXSIZE + BCODE_MAXSIZE ]; // Last region (BCODE_MAXSIZE)
                                                  //     stores command line data
  line_t Lines[ PROG_MAXLINE + 2 ];     // First element is COMMAND LINE which
                                        //      contains ZERO
                                        // Last element is a GUARD which contains
                                        //      PROG_BLANKLINE
     int Psize;                         // Program size in BYTES
     int Lsize;                         // How many LINES are there?
     int Pnum;                          // Current processing line number

//
// File scope variables
//

static reloc_t Relocs[ PROG_MAXRELOC ];
    static int Rsize;

static char * Results[] = {
    "PROGRAM \"ok\"",                   // PROG_SUCCESS
    "PROGRAM \"line not found\"",       // PROG_LINENOTFOUND
    "PROGRAM \"too many lines\"",       // PROG_TOOMANYLINES
    "PROGRAM \"size over\"",            // PROG_SIZEOVER
    "PROGRAM \"too many relocations\""  // PROG_TOOMANYRELOCS
};

//
// Program line processing functions
//

void
prog_init(void)
{
    int i;

    // Initialize program area and line holders

    for (i = 0; i < (PROG_MAXSIZE + BCODE_MAXSIZE); i++)
        Program[ i ] = PROG_BLANKDATA;

    for (i = 0; i <= (PROG_MAXLINE + 1); i++) {
        Lines[ i ].num = PROG_BLANKLINE;
        Lines[ i ].add = 0;
        Lines[ i ].len = 0;
    }
    Lines[ 0 ].num = 0;

    // Initialize control variables

    Psize = Lsize = 0;
}

int
prog_search(int linenum)
{
    int i = 1, find = 0;

    while (Lines[ i ].num != PROG_BLANKLINE) {
        if (Lines[ i ].num == linenum) {
            find = 1;
            break;
        } else if (Lines[ i ].num > linenum)
            break;
        i++;
    }

    // NOTE: Returns INDEX number of Lines[]
    //        postive number -> found (linenum == Lines[i].num)
    //       negative number -> not found (linenum < Lines[-i].num)

    return find ? i : -i;
}

static void
spacing(FILE *out)
{
    if (bcode_nextiseol())
        return;
    else if (bcode_nextisdlm()) {
        if (bcode_next() != BCODE_AT)
            return;
    }
    fprintf(out, " ");
}

int
prog_list(FILE *out, int color, int startnum, int endnum)
{
    int sidx, eidx, i, loop;
    bcode_t b;
    byte_t *ptr, ch, esc;

    if (Lsize == 0)
        return PROG_SUCCESS;

    sidx = prog_search(startnum);
    if (sidx < 0)
        return PROG_LINENOTFOUND;
    eidx = prog_search(endnum);
    if (eidx < 0)
        return PROG_LINENOTFOUND;

    bcode_start(Program);
    for (i = sidx; i <= eidx; i++) {
        fprintf(out, "%04d ", Lines[ i ].num);
        bcode_setpc(Lines[ i ].add);
        loop = 1;
        while (loop) {
            bcode_read(&b);
            switch (b.type) {
                case BCODE_TYPE_EOL:
                    loop = 0;
                    break;

                case BCODE_TYPE_NUMBER10:
                    fprintf(out, "%d", b.num);
                    spacing(out);
                    break;
```

```
case BCODE_TYPE_NUMBER16:
    if (b.num & 0xFF00)
        fprintf(out, "0x%04hX", b.num);
    else
        fprintf(out, "0x%02hX", b.num);
    spacing(out);
    break;

case BCODE_TYPE_STRING:
    fprintf(out, "\"");
    ptr = b.str;
    while ((ch = *ptr++)) {
        esc = 0;
        if (ch < ' ' || ch == '\\') {
            switch (ch) {
                case '\007':
                    esc = 'a';
                    break;

                case '\010':
                    esc = 'b';
                    break;

                case '\011':
                     esc = 't';
                    break;

                case '\012':
                    esc = 'n';
                    break;

                case '\015':
                    esc = 'r';
                    break;

                case '\033':
                    esc = 'e';
                    break;

                case '\\':
                    esc = '\\';
                    break;

                default:
                    esc = 'x';
                    break;
            }
        }
        if (esc) {
            if (esc == 'x')
                fprintf(out, "\\x%02X", ch);
            else
                fprintf(out, "\\%c", esc);
        } else
            fprintf(out, "%c", ch);
    }
    fprintf(out, "\"");
    break;

case BCODE_TYPE_VARIABLE:
    if (color)
        fprintf(out, ESCAPE_LBLUE);
    fprintf(out, "%c", b.idx + 'A');
    if (color)
        fprintf(out, ESCAPE_DEFAULT);
    spacing(out);
                    break;

                case BCODE_TYPE_KEYWORD:
                    if (color)
                        fprintf(out, ESCAPE_LGREEN);
                    fprintf(out, "%s", token_keyword(b.idx));
                    if (color)
                        fprintf(out, ESCAPE_DEFAULT);
                    spacing(out);
                    break;

                case BCODE_TYPE_DELIMITER:
                    if (color && b.inst == BCODE_AT)
                        fprintf(out, ESCAPE_LBLUE);
                    fprintf(out, "%s", token_delimiter(b.idx));
                    if (color && b.inst == BCODE_AT)
                        fprintf(out, ESCAPE_DEFAULT);
                    if (b.inst == BCODE_RPAREN || b.inst == BCODE_RBRACK)
                        if (! bcode_nextisdlm() && ! bcode_nextiseol())
                            fprintf(out, " ");
                    break;
            }
        }
        fprintf(out, "\n");
    }
    return PROG_SUCCESS;
}

int
prog_delete(int startnum, int endnum)
{
    int sidx, eidx, dlines, dbytes, mbytes, i;
    byte_t *src, *dst;

    if (Lsize == 0)
        return PROG_LINENOTFOUND;

    sidx = prog_search(startnum);
    if (sidx < 0)
        return PROG_LINENOTFOUND;
    eidx = prog_search(endnum);
    if (eidx < 0)
        return PROG_LINENOTFOUND;

    dlines = eidx - sidx + 1;
    dbytes = 0;
    for (i = sidx; i <= eidx; i++)
        dbytes += Lines[ i ].len;

    if (eidx != Lsize) {
        dst = &Program[ Lines[ sidx ].add ];
        src = &Program[ Lines[ eidx + 1 ].add ];
        mbytes = 0;
        for (i = eidx + 1; i <= Lsize; i++)
            mbytes += Lines[ i ].len;

        // Transfer tail program
        while (mbytes--)
            *dst++ = *src++;

        // Transfer tail line holder
        for  (i = eidx + 1; i <= Lsize; i++) {
            Lines[ i - dlines ].num = Lines[ i ].num;
            Lines[ i - dlines ].add = Lines[ i ].add - dbytes;
            Lines[ i - dlines ].len = Lines[ i ].len;
        }
```

```
    }

    for (i = Lsize - dlines + 1; i <= Lsize; i++) {
        Lines[ i ].num = PROG_BLANKLINE;
        Lines[ i ].add = 0;
        Lines[ i ].len = 0;
    }

    Psize -= dbytes;
    Lsize -= dlines;

    return PROG_SUCCESS;
}

int
prog_insert(int linenum, byte_t *src, int len)
{
    byte_t *dst, *tsrc, *tdst;
    int idx, i, start, move;

    if (Lsize == PROG_MAXLINE)
        return PROG_TOOMANYLINES;
    if ((Psize  + len) > PROG_MAXSIZE)
        return PROG_SIZEOVER;

    idx = prog_search(linenum);
    if (idx > 0) {
        prog_delete(linenum, linenum);
        idx = prog_search(linenum);
    }
    idx = -idx;

    start = 0;
    for (i = 1; i < idx; i++) {
        start += Lines[ i ].len;
    }
    move = Psize - start + 1;
    dst = &Program[ start ];

    if (idx != (Lsize + 1)) {
        tsrc = &Program[ Psize - 1 ];
        tdst = tsrc + len;
        while (move--) {
            *tdst-- = *tsrc--;
        }
        for  (i = Lsize; i >= idx; i--) {
            Lines[ i + 1 ].num = Lines[ i ].num;
            Lines[ i + 1 ].add = Lines[ i ].add + len;
            Lines[ i + 1 ].len = Lines[ i ].len;
        }
    }

    move = len;
    while (move--) {
        *dst++ = *src++;
    }

    Lines[ idx ].num = linenum;
    Lines[ idx ].add = start;
    Lines[ idx ].len = len;
    Psize += len;
    Lsize++;

    return PROG_SUCCESS;
}

void
prog_cmdline(byte_t *src, int size)
{
    int i;

    for (i = 0; i < size; i++)
        Program[ PROG_CMDLINE + i ] = *src++;
    Lines[ 0 ].num = 0;
    Lines[ 0 ].add = PROG_CMDLINE;
    Lines[ 0 ].len = size;
}

int
prog_renum(int start, int step)
{
    int i, add, scan, idx;
    bcode_t b;
    byte_t *ptr;

    Rsize = 0;
    for (i = 1; i <= Lsize; i++) {
        Pnum = Lines[ i ].num;
        add = Lines[ i ].add;
        bcode_start(&Program[ add ]);
        scan = 1;
        while (scan) {
            bcode_read(&b);
            switch (b.type) {
                case BCODE_TYPE_EOL:
                    scan = 0;
                    break;

                case BCODE_TYPE_NUMBER10:
                case BCODE_TYPE_NUMBER16:
                case BCODE_TYPE_STRING:
                case BCODE_TYPE_VARIABLE:
                case BCODE_TYPE_DELIMITER:
                    break;

                case BCODE_TYPE_KEYWORD:
                    if (b.inst == BCODE_GOSUB || b.inst == BCODE_GOTO) {
                        bcode_read(&b);
                        idx = prog_search(b.num);
                        if (idx < 0)
                            return PROG_LINENOTFOUND;
                        if ((Rsize + 1) > PROG_MAXRELOC)
                            return PROG_TOOMANYRELOCS;
                        Relocs[ Rsize ].idx = idx;
                        Relocs[ Rsize ].add = add + b.pos;
                        Rsize++;
                    }
                    break;
            }
        }
    }

    // Do renumber

    for (i = 1; i <= Lsize; i++)
        Lines[ i ].num = start + step * (i - 1);

    // Do relocation

    for (i = 0; i < Rsize; i++) {
        ptr = &Program[ Relocs[ i ].add ];
        ptr++;
```

```
        *((word_t*) ptr) = (word_t) Lines[ Relocs[ i ].idx ].num;
    }

    return PROG_SUCCESS;
}

char *
prog_result(int idx)
{
    return Results[ idx ];
}

#define BYTES_PERLINE    20
void
prog_dumpbytes(void)
{
    int i;

    printf("%sProgram contents%s\n", ESCAPE_LGREEN, ESCAPE_DEFAULT);
    printf("    Program size = %d bytes\n", Psize);
    for (i = 0; i < Psize; i++) {
        if ((i % BYTES_PERLINE) == 0)
            printf("    0d%05d : ", i);
        printf("%02hX ", Program[ i ]);
        if ((i % BYTES_PERLINE) == (BYTES_PERLINE - 1))
            printf("\n");
    }
    if ((i % BYTES_PERLINE) != (BYTES_PERLINE - 1))
        printf("\n");
}
void
prog_dumplines(void)
{
    int i;

    printf("%sLine information%s\n", ESCAPE_LGREEN, ESCAPE_DEFAULT);
    printf("    Total %d lines\n", Lsize);
    for (i = 1; i <= Lsize; i++) {
        printf("    Lines[%d].num = %d : Lines[%d].add = %d : ", \
                i, Lines[ i ].num, i, Lines[ i ].add);
        printf("Lines[%d].len = %d\n", i, Lines[ i ].len);
    }
}
```

```
#include <stdio.h>      // FILE{}, fgets()
#include "token.h"
#include "bcode.h"
#include "program.h"

FILE *Debug;

int
main()
{
    char buffer[ 256 ];
    byte_t bytes[ 256 ];
    int bsize, ret, num;
    bcode_t b;

    Debug = stderr;
    token_init();
    prog_init();

    while (fgets(buffer, 256, stdin) != NULL) {
        if (*buffer == 0 || *buffer == '\n')
            break;
        fprintf(Debug, "===== %s", buffer);
        token_source(buffer);
        ret = bsize = bcode_compile(bytes);
        if (bsize < 0) {
            fprintf(Debug, "***** Error: %s\n", bcode_result(ret));
            break;
        }
        fprintf(Debug, ">>>>> Compiled bytecodes [ %d bytes ]\n", bsize);
#ifdef DEBUG
            bcode_dump(bytes);
#endif // DEBUG

        bcode_start(bytes);
        bcode_read(&b);
        if (b.inst == BCODE_LIST) {
            fprintf(Debug, "----------------------------------------\n");
            ret = prog_list(stdout, 1, Lines[1].num, Lines[Lsize].num);
            fprintf(Debug, "----------------------------------------\n");
        } else if (b.inst == BCODE_NEW) {
            prog_init();
            continue;
        } else if (b.type == BCODE_TYPE_NUMBER10) {
            num = b.num;
            if (bcode_next() == BCODE_EOL) {
                ret = prog_delete(num,num);
                if (ret) {
                    fprintf(Debug, "***** Error: %s\n", prog_result(ret));
                    continue;
                }
                fprintf(Debug, ">>>>> Line %d deleted\n", num);
            } else {
                bcode_read(&b);
                ret = prog_insert(num, &bytes[ b.pos ], bsize - 3);
                if (ret) {
                    fprintf(Debug, "***** Error: %s\n", prog_result(ret));
                    continue;
                }
                fprintf(Debug, ">>>>> Line %d inserted\n", num);
            }
        } else {
            fprintf(Debug, "***** Error: illegal program line\n");
            continue;
        }
        prog_dumpbytes();
        printf("\n");
        prog_dumplines();
        ret = 0;
    }
    return 0;
}
```

```
//
//  c o n t a i n e r . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _CONTAINER_H_
#define _CONTAINER_H_

#include "byteword.h"

//
// Container constants
//

#define CONT_MAXARRAY   512
#define CONT_DEFARRAY   10
#define CONT_MAXSTRING  512

//
// Type definition
//

typedef struct {
    int row;
    int col;
} dim_t;

//
// Global variables
//

extern word_t Var[ 26 ];
extern word_t Array[ 26 ][ CONT_MAXARRAY ];
 extern dim_t Asize[ 26 ];
extern byte_t String[ CONT_MAXSTRING ];

//
// Function prototype
//

void cont_init(void);
void cont_dumpvar(void);
void cont_dumparr(void);
void cont_dumpstr(void);

#endif // _CONTAINER_H_
```

```
//
//  c o n t a i n e r . c
//
//  BAREMETALHACK.COM --> public domain
//

#include <stdio.h>          // printf()
#include "container.h"
#include "escape.h"

//
// Global variables
//

word_t Var[ 26 ];
word_t Array[ 26 ][ CONT_MAXARRAY ];
 dim_t Asize[ 26 ];
byte_t String[ CONT_MAXSTRING ];

//
// Container functions
//

void
cont_init(void)
{
    int i, j;

    for (i = 0; i < 26; i++)
        Var[ i ] = 0;

    for (i = 0; i < 26; i++)
        for (j = 0; j < CONT_MAXARRAY; j++)
            Array[ i ][ j ] = 0;

    for (i = 0; i < 26; i++) {
        Asize[ i ].row = 1;
        Asize[ i ].col = CONT_DEFARRAY;
    }

    for (i = 0; i < CONT_MAXSTRING; i++)
        String[ i ] = 0;
}

void
cont_dumpvar(void)
{
    int i;

    printf("%sVariables%s\n", ESCAPE_LGREEN, ESCAPE_DEFAULT);
    for (i = 0; i < 26; i++)
        printf("%c = %6d (0x%04hX)\n", 'A' + i, Var[ i ], Var[ i ]);
}

void
cont_dumparr(void)
{
    int i, j, k;

    printf("%sArrays%s\n", ESCAPE_LGREEN, ESCAPE_DEFAULT);
    for (i = 0; i < 26; i++) {
        printf("DIM %c[%d,%d] = { ", 'A' + i, Asize[ i ].row, Asize[ i ].col);
        for (j = 0; j < Asize[ i ].row; j++) {
            if (Asize[ i ].row > 1)
                printf("{ ");
            for (k = 0; k < Asize[ i ].col; k++) {
                printf("%d", Array[ i ][ Asize[ i ].col * j + k ]);
                if (k != (Asize[ i ].col - 1))
                    printf(",");
            }
            if (Asize[ i ].row > 1)
                printf(" }");
            if (j != (Asize[ i ].row - 1))
                printf(", ");
        }
        printf(" }\n");
    }
}

#define CHARS_PERLINE    20
void
cont_dumpstr(void)
{
    int i = 0, len = 0, cnt, idx;
    char ch;

    printf("%sString @%s\n", ESCAPE_LGREEN, ESCAPE_DEFAULT);
    while(String[ i++ ])
        len++;
    len++;

    idx = 0;
    cnt = len;
    while (len) {
        cnt = (len > CHARS_PERLINE) ? CHARS_PERLINE : len;
        for (i = 0; i < cnt; i++)
            printf("%02hhX ", String[ idx + i ]);
        printf("| ");
        for (i = 0; i < cnt; i++) {
            ch = String[ idx + i ];
            printf("%c", (ch < ' ') ? '?' : ch);
        }
        printf("\n");
        idx += cnt;
        len -= cnt;
    }
}
```

```
//
// f u n c t i o n . h
//
// BAREMETALHACK.COM --> public domain
//

#ifndef _FUNCTION_H_
#define _FUNCTION_H_

//
// Functions
//

 int func_rnd(void);
#ifdef PI
 int func_in(int);
#endif // PI

#endif // _FUNCTION_H_
```

```
//
// f u n c t i o n . c
//
// BAREMETALHACK.COM --> public domain
//

#include <sys/time.h>           // timeval{}, gettimeofday()
#include <stdlib.h>             // srandom(), random()

#ifdef PI
#include "pi_gpio.h"
#endif // PI

//
// BASIC functions
//

static int Seed = 0;

int
func_rnd(void)
{
/*** Classical version using modullo
    static word_t x = 1025;

    x = (257 * x + 1) % 0x8000;
    return (word_t) x;
***/
    struct timeval t;

    if (Seed == 0) {
        gettimeofday(&t, NULL);
        Seed = t.tv_sec;
        srandom(Seed);;
    }
    return random() & 0x7FFF;
}

#ifdef PI
int
func_in(int bit)
{
    return gpio_in(bit);
}
#endif // PI
```

```
//
//  e v a l . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _EVAL_H_
#define _EVAL_H_

#include <setjmp.h>         // jmp_buf{}
#include "container.h"      // dim_t{}

//
// Result codes
//

#define EVAL_SUCCESS        0
#define EVAL_ILLARRAY       1
#define EVAL_ILLFACTOR      2
#define EVAL_ILLINDEX       3
#define EVAL_INDEXOVER      4
#define EVAL_NOASSIGN       5
#define EVAL_NOCOMMA        6
#define EVAL_NOLBRACK       7
#define EVAL_NORBRACK       8
#define EVAL_NOLPAREN       9
#define EVAL_NORPAREN       10
#define EVAL_NOSTRING       11
#define EVAL_DIVBYZERO      12

//
// Function prototype
//

  void eval_init(jmp_buf *);
  void eval_dim(dim_t *);
  void eval_assign(int);
word_t eval(void);
char * eval_result(int);

#endif // _EVAL_H_
```

```
//
//  e v a l . c
//
//  BAREMETALHACK.COM --> public domain
//

#include <setjmp.h>         // jmp_buf{}, longjmp()
#include "bcode.h"
#include "container.h"
#include "function.h"
#include "eval.h"
#include "debug.h"

//
// File scope variables
//

static jmp_buf *Exit;
static bcode_t Bcode;

static char * Results[] = {
    "OK",                   // EVAL_SUCCESS
    "illegal array",        // EVAL_ILLARRAY
    "illegal factor",       // EVAL_ILLFACTOR
    "illegal index",        // EVAL_ILLINDEX
    "index OVER",           // EVAL_INDEXOVER
    "no assignment",        // EVAL_NOASSIGN
    "no comma",             // EVAL_NOCOMMA
    "no left bracket",      // EVAL_NOLBRAK
    "no right bracket",     // EVAL_NOLBRAK
    "no left parenthesis",  // EVAL_NOLPAREN
    "no right parenthesis", // EVAL_NORPAREN
    "no string",            // EVAL_NOSTRING
    "divided by ZERO"       // EVAL_DIVBYZERO
};

//
// Error handler
//

void
eval_error(int code)
{
    longjmp(*Exit, code);
}

//
// Initializer
//

void
eval_init(jmp_buf *env)
{
    Exit = env;
}

//
// Dimension evaluator
//

void
eval_dim(dim_t *dim)
{
    word_t row = 0, col = 0;

    if (bcode_next() != BCODE_LBRACK)
        eval_error(EVAL_NOLBRACK);
    bcode_skip();
    row = eval();
    if (bcode_next() == BCODE_RBRACK) {
        bcode_skip();
        if (row < 0 || row > CONT_MAXARRAY)
            eval_error(EVAL_ILLINDEX);
        col = row;
        row = 0;
    } else if (bcode_next() == BCODE_COMMA) {
        bcode_skip();
        col = eval();
        if (col < 0 || col > CONT_MAXARRAY)
            eval_error(EVAL_ILLINDEX);
        if (bcode_next() != BCODE_RBRACK)
            eval_error(EVAL_NORBRACK);
        bcode_skip();
    } else
        eval_error(EVAL_ILLARRAY);
    dim->row = row;
    dim->col = col;
}

// Assign value to variable or array
//     var == 0 -> assign to string holder @[]
//     var != 0 -> assign to variable/array (index = var - 1)

void
eval_assign(int var) {
    bcode_t b;
    byte_t *src, *dst;
    word_t val, idx;
    int array, row, col;
    dim_t dim;

    // String array (@[])

    if (var == 0) {
        if (bcode_next() == BCODE_ASSIGN) {
            bcode_skip();
            if (! bcode_nextisstr())
                eval_error(EVAL_NOSTRING);
            bcode_read(&b);
            src = b.str;
            dst = String;
            while (*src)
                *dst++ = *src++;
            *dst = 0;
            return;
        } else if (bcode_next() == BCODE_LBRACK) {
            bcode_skip();
            idx = eval();
            if (idx < 0 || idx >= CONT_MAXSTRING)
                eval_error(EVAL_ILLINDEX);
            if (bcode_next() != BCODE_RBRACK)
                eval_error(EVAL_NORBRACK);
            bcode_skip();

            if (bcode_next() != BCODE_ASSIGN)
                eval_error(EVAL_NOASSIGN);
            bcode_skip();

            val = eval();
            String[ idx ] = (byte_t) (val & 0xFF);
            DMSG("@[%d] = %d", idx, String[idx]);
            return;
```

```
        } else
            eval_error(EVAL_NOLBRACK);
    }

    // Variable or standard array

    var--;  // Recover right index
    if (bcode_next() == BCODE_LBRACK) {
        eval_dim(&dim);
        row = dim.row;
        col = dim.col;
        if (dim.row >= Asize[ var ].row || dim.col >= Asize[ var ].col)
            eval_error(EVAL_ILLINDEX);
        array = 1;
        DMSG("rowsiz = %d : colsiz = %d : row = %d : col = %d", \
                Asize[ var ].row, Asize[ var ].col, row, col);
    } else
        array = 0;

    if (bcode_next() != BCODE_ASSIGN)
        eval_error(EVAL_NOASSIGN);
    bcode_skip();

    // Variable assignment

    if (! array) {
        val = eval();
        Var[ var ] = val;
        DMSG("variable %c = %d", var+'A', val);
        return;
    }

    // Array assignment

    while (1) {
        val = eval();
        idx = Asize[ var ].col * row + col;
        Array[ var ][ idx ] = val;
        DMSG("array %c[%d,%d] = %c[%d] = %d", var+'A', row, col, var+'A', idx, val
);
        if (bcode_next() == BCODE_COMMA) {
            bcode_skip();
            col++;
            if (col >= Asize[ var ].col) {
                col = 0;
                row++;
                if (row >= Asize[ var ].row)
                    eval_error(EVAL_INDEXOVER);
            }
        } else
            break;
    }
    return;
}

//
// Recursive descent evaluator
//

static word_t
level_zero(void)
{
    word_t res = 0, v, idx;
    dim_t dim;

    if (Bcode.type == BCODE_TYPE_NUMBER10) {
        res = Bcode.num;
        DMSG("number %d", Bcode.num);
    } else if (Bcode.type == BCODE_TYPE_NUMBER16) {
        res = Bcode.num;
        DMSG("number 0x%04hX", Bcode.num);
    } else if (Bcode.inst == BCODE_LPAREN) {
        res = eval();
        if (bcode_next() != BCODE_RPAREN)
            eval_error(EVAL_NORPAREN);
        bcode_skip();
        DMSG("( %d )", res);
    } else if (Bcode.type == BCODE_TYPE_VARIABLE) {
        v = Bcode.idx;
        if (bcode_next() == BCODE_LBRACK) {
            eval_dim(&dim);
            if (dim.row >= Asize[ v ].row || dim.col >= Asize[ v ].col)
                eval_error(EVAL_ILLINDEX);
            idx = Asize[ v ].col * dim.row + dim.col;
            res = Array[ v ][ idx ];
            DMSG("array %c[%d,%d] = %c[%d] = %d",
                 v+'A', dim.row, dim.col, v+'A', idx, res);
        } else {
            res = Var[ v ];
            DMSG("variable %c = %d", v+'A', res);
        }
    } else if (Bcode.inst == BCODE_AT) {
        if (bcode_next() != BCODE_LBRACK)
            eval_error(EVAL_NOLBRACK);
        bcode_skip();
        idx = eval();
        if (bcode_next() != BCODE_RBRACK)
            eval_error(EVAL_NORBRACK);
        bcode_skip();
        res = (word_t) String[ idx ];
        DMSG("@[ %d ] = %d", idx, res);
    } else if (Bcode.inst == BCODE_RND) {
        if (bcode_next() != BCODE_LPAREN)
            eval_error(EVAL_NOLPAREN);
        bcode_skip();
        if (bcode_next() != BCODE_RPAREN)
            eval_error(EVAL_NORPAREN);
        bcode_skip();
        res = func_rnd();
        DMSG("rnd() = %d", res);
#ifdef PI
    } else if (Bcode.inst == BCODE_IN) {
        if (bcode_next() != BCODE_LPAREN)
            eval_error(EVAL_NOLPAREN);
        bcode_skip();
        idx = eval();
        if (bcode_next() != BCODE_RPAREN)
            eval_error(EVAL_NORPAREN);
        bcode_skip();
        res = func_in((int) idx);
        DMSG("in(%d) = %d", idx, res);
#endif // PI
    } else
        eval_error(EVAL_ILLFACTOR);
    return res;
}

static word_t
level_one(void)
{
    word_t unary, res;
```

```
    if (Bcode.inst == BCODE_SUB) {
        bcode_read(&Bcode);
        unary = level_zero();
        DMSG("-%d", unary);
        res = -unary;
    } else if (Bcode.inst == BCODE_ADD) {
        bcode_read(&Bcode);
        unary = level_zero();
        DMSG("+%d", unary);
        res = unary;
    } else
        res = level_zero();
    return res;
}

static word_t
level_two(void)
{
    int op;
    word_t first, second;

    first = level_one();
    while (1) {
        op = bcode_next();
        if (op == BCODE_MUL || op == BCODE_DIV || op == BCODE_MOD) {
            bcode_skip();
            bcode_read(&Bcode);
            second = level_one();
            switch (op) {
                case BCODE_MUL:
                    DMSG("%d * %d", first, second);
                    first = first * second;
                    break;

                case BCODE_DIV:
                    if (second == 0)
                        eval_error(EVAL_DIVBYZERO);
                    DMSG("%d / %d", first, second);
                    first = first / second;
                    break;

                case BCODE_MOD:
                    DMSG("%d %% %d", first, second);
                    first = first % second;
                    break;
            }
        } else
            break;
    }
    return first;
}

static word_t
level_three(void)
{
    int op;
    word_t first, second;

    first = level_two();
    while (1) {
        op = bcode_next();
        if (op == BCODE_ADD || op == BCODE_SUB) {
            bcode_skip();
            bcode_read(&Bcode);
            second = level_two();
            switch (op) {
                case BCODE_ADD:
                    DMSG("%d + %d", first, second);
                    first = first + second;
                    break;

                case BCODE_SUB:
                    DMSG("%d - %d", first, second);
                    first = first - second;
                    break;
            }
        } else
            break;
    }
    return first;
}

static word_t
level_four(void)
{
    int op;
    word_t first, second;

    first = level_three();
    while (1) {
        op = bcode_next();
        if (op == BCODE_EQUAL || op == BCODE_GEQUAL ||
            op == BCODE_GTHAN || op == BCODE_LEQUAL ||
            op == BCODE_LTHAN || op == BCODE_NEQUAL) {
            bcode_skip();
            bcode_read(&Bcode);
            second = level_three();
            switch (op) {
                case BCODE_EQUAL:
                    DMSG("%d == %d", first, second);
                    first = (first == second) ? 1 : 0;
                    break;

                case BCODE_GEQUAL:
                    DMSG("%d >= %d", first, second);
                    first = (first >= second) ? 1 : 0;
                    break;

                case BCODE_GTHAN:
                    DMSG("%d > %d", first, second);
                    first = (first > second) ? 1 : 0;
                    break;

                case BCODE_LEQUAL:
                    DMSG("%d <= %d", first, second);
                    first = (first <= second) ? 1 : 0;
                    break;

                case BCODE_LTHAN:
                    DMSG("%d < %d", first, second);
                    first = (first < second) ? 1 : 0;
                    break;

                case BCODE_NEQUAL:
                    DMSG("%d != %d", first, second);
                    first = (first != second) ? 1 : 0;
                    break;
            }
        } else
            break;
    }
    return first;
```

```
}

static word_t
level_five(void)
{
    int op;
    word_t first, second;

    first = level_four();
    while (1) {
        op = bcode_next();
        if (op == BCODE_AND || op == BCODE_OR) {
            bcode_skip();
            bcode_read(&Bcode);
            second = level_four();
            switch (op) {
                case BCODE_AND:
                    DMSG("%d & %d", first, second);
                    first = first & second;
                    break;

                case BCODE_OR:
                    DMSG("%d | %d", first, second);
                    first = first | second;
                    break;
            }
        } else
            break;
    }
    return first;
}

word_t eval(void) {
    word_t res;

    bcode_read(&Bcode);
    if (Bcode.inst == BCODE_EOL)
        return 0;
    res = level_five();
    DMSG("--> %d (0x%04X)", res, res);
    return res;
}

char *
eval_result(int idx)
{
    return Results[ idx ];
}
```

```
#include <stdio.h>      // fprintf()
#include <setjmp.h>     // setjmp()

#include "token.h"
#include "bcode.h"
#include "eval.h"

FILE *Debug;
jmp_buf Recovery;

int main() {
    char buffer[ 256 ], *ptr;
    byte_t bytes[ 256 ];
    int ret;
    word_t val;

    Debug = stderr;
    token_init();
    eval_init(&Recovery);
    while (1) {
        fprintf(stdout, "\n");
        fprintf(Debug, "\n");
        if ((ret = setjmp(Recovery)) != 0) {
            fprintf(stdout, "*** eval() error: %s\n", eval_result(ret));
            continue;
        }
        ptr = fgets(buffer, 256, stdin);
        if (ptr == 0 || *buffer == 0 || *buffer == '\n')
            break;
        token_source(buffer);
        ret = bcode_compile(bytes);
        if (ret < 0) {
            fprintf(stdout, "*** bcode_compile() error: %s ***\n", bcode_result(-r
et));
            continue;
        }
        bcode_start(bytes);
        val = eval();
        fprintf(stdout, "eval() returns %d\n", val);
    }
    return 0;
}
```

```
#include <stdio.h>      // fprintf()
#include <setjmp.h>     // setjmp()

#include "bcode.h"
#include "eval.h"

FILE *Debug;
jmp_buf Recovery;

int main() {
    char buffer[ 256 ], *ptr;
    byte_t bytes[ 256 ];
    int ret;
    word_t val;
    bcode_t b;

    Debug = stderr;
    eval_init(&Recovery);
    while (1) {
        printf("\n");
        fprintf(Debug, "\n");
        if ((ret = setjmp(Recovery)) != 0) {
            printf("*** eval() error: %s\n", eval_result(ret));
            continue;
        }
        ptr = fgets(buffer, 256, stdin);
        if (ptr == 0 || *buffer == 0 || *buffer == '\n')
            break;
        token_source(buffer);
        ret = bcode_compile(bytes);
        if (ret < 0) {
            printf("*** bcode_compile() error: %s\n", bcode_result(-ret));
            continue;
        }
        bcode_start(bytes);
        bcode_read(&b);
        if (b.inst == BCODE_INQ) {
            val = eval();
            printf("eval() returns %d\n", val);
        } else if (b.type == BCODE_TYPE_VARIABLE)
            eval_assign(b.idx + 1);
        else if (b.inst == BCODE_AT)
            eval_assign(0);
        else
            printf("*** Syntax error\n");
    }
    return 0;
}
```

```
//
//  b a s i c . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _BASIC_H_
#define _BASIC_H_

#include <setjmp.h>             // jmp_buf{}

//
// Program information
//

#ifdef PI
#define BASIC_NAME              "antBASIC for Pi"
#else
#define BASIC_NAME              "antBASIC for Unix"
#endif // PI

// Version number

#define BASIC_MAJOR             1
#define BASIC_MINOR             0
#define BASIC_PATCH             1

//
// BASIC constants
//

#define BASIC_MAXLINECHAR       512     // Maximum characters on command line
#define BASIC_MAXSTACKDEPTH     8       // Maximum depth of GOSUB/FOR stack
#define BASIC_LINESTART         100     // RENUM default start line number
#define BASIC_LINESTEP          10      // RENUM default line step

#define BASIC_MODERUN           0       // Normal execution mode (RUN)
#define BASIC_MODEDIRECT        1       // Direct execution from command line
#define BASIC_MODELOAD          2       // Program load mode (LOAD/MERGE)
#define BASIC_MODESHELL         4       // Execution from the shell

//
// Return codes
//

#define BASIC_SUCCESS           0
#define BASIC_DIRECTDENY        1
#define BASIC_DIRNOTFOUND       2
#define BASIC_FILEIOERROR       3
#define BASIC_ILLARRAY          4
#define BASIC_ILLRANGE          5
#define BASIC_NOARRAY           6
#define BASIC_NOASSIGN          7
#define BASIC_NOCONTROLVAR      8
#define BASIC_NODIRNAME         9
#define BASIC_NOLINENUM         10
#define BASIC_NOFILENAME        11
#define BASIC_NOSTACK           12
#define BASIC_NOVARIABLE        13
#define BASIC_STACKOVER         14
#define BASIC_SYNTAXERROR       15

// Error code group BIT

#define BASIC_BCODE_ERROR       0b00100000
#define BASIC_PROG_ERROR        0b01000000
#define BASIC_GPIO_ERROR        0b10000000

//
// Type definitions
//

typedef struct {
    int lidx;                   // Line index number
    int radd;                   // Return address
} gstack_t;

typedef struct {
    int lidx;                   // Line index number
    int ladd;                   // Looping entry address
    int vidx;                   // Variable index number
    int ecnt;                   // End count
} fstack_t;

//
// Global variable
//

extern int Lnum;                // Line number currently executing
extern int Mode;                // Current mode (NORMAL/DIRECT/LOAD/SHELL)
extern int Status;              // Exit status (in the case of SHELL mode)
extern char Text[];             // Source text line buffer

//
// Function prototype
//

  void basic_init(jmp_buf *);
  void basic_error(int);
   int basic_readline(char *, char *, int);
   int basic_command(int);
   int basic_load(char *);
   int basic_exec(void);
char * basic_result(int);

#endif  // _BASIC_H_
```

```
//
//  b a s i c . c
//
//  BAREMETALHACK.COM --> public domain
//

#include <stdio.h>              // FILE{}, stdin, fgets(), fflush()
#include <setjmp.h>             // jmp_buf{}, longjmp()
#include <string.h>             // strlen(), strncpy()
#include <stdlib.h>             // atoi(), free()
#include <dirent.h>             // DIR{}, dirent{}, scandir()
#include <unistd.h>             // usleep()

#ifdef READLINE
#include <readline/readline.h>  // readline()
#include <readline/history.h>   // add_history()
#endif // READLINE

#include "basic.h"
#include "token.h"
#include "bcode.h"
#include "eval.h"
#include "program.h"
#include "container.h"
#include "escape.h"
#include "debug.h"

#ifdef PI
#include "pi_gpio.h"
#endif // PI

//
// Global variable
//

 int Lnum;                  // Current executing line number
 int Mode;                  // Current executing mode (NORMAL/DIRECT/LOAD/SHELL)
 int Status = 0;            // Exit status which will be returned to the shell
char Text[ TOKEN_MAXBUF ];  // Source text line buffer

//
// File scope variables
//

static jmp_buf * Exit;
      static int Lidx;
      static int Gsize, Fsize;
 static gstack_t Gstack[ BASIC_MAXSTACKDEPTH ];
 static fstack_t Fstack[ BASIC_MAXSTACKDEPTH ];

static char * Results[] = {
    "BASIC \"ok\"",                     // BASIC_SUCCESS
    "BASIC \"direct exec is denied\"",  // BASIC_DIRECTDENY
    "BASIC \"directory not found\"",    // BASIC_DIRNOTFOUND
    "BASIC \"file I/O error\"",         // BASIC_FILEIOERROR
    "BASIC \"illegal array\"",          // BASIC_ILLARRAY
    "BASIC \"illegal range\"",          // BASIC_ILLRANGE
    "BASIC \"no array\"",               // BASIC_NOARRAY
    "BASIC \"no assignment\"",          // BASIC_NOASSIGN
    "BASIC \"no control variable\"",    // BASIC_NOCONTROLVAR
    "BASIC \"no directroy name\"",      // BASIC_NODIRNAME
    "BASIC \"no line number\"",         // BASIC_NOLINENUM
    "BASIC \"no filename\"",            // BASIC_NOFILENAME
    "BASIC \"no stack\"",               // BASIC_NOSTACK
    "BASIC \"no variable\"",            // BASIC_NOVARIABLE
    "BASIC \"stack is overflowed\"",    // BASIC_STACKOVER
    "BASIC \"syntax error\""            // BASIC_SYNTAXERROR
};

static char* help_container[] = {
    "CONTAINERS",       "",
    "Numbers",          "Signed 16bit integer (range from -32768 to 32767)$" \
                        "Decimal or hexadecimal (0x prefix is needed)$" \
                        "    ex. 1234, -1234, 0xABCD, 0xEF",
    "Strings",          "8bit ASCII characters surrounded by double quotation$" \
                        "Escaped special characters are as follows:$" \
                        "    \\a BEL, \\b BS, \\t TAB, \\n LF, \\r CR, \\e ESC, \\\\
 backslash$" \
                        "Special array @ holds string$" \
                        "    ex. @=\"hello!\":@[0]=@[0]-0x20:print @ -> Hello!$" \
                        "        @[0]=33:@[1]=7:@[2]=0:print @ -> ! with alarm",
    "Variables",        "Vaiable A to Z holds integer: ex. A=123:B=A+0x1234",
    "Arrays",           "Array A[] to Z[] holds integer (index starts from ZERO)$" \

                        "Two-dimensional array form is X[column,row]$" \
                        "    ex. DIM A[1],B[2,3]:A[0]=1:B[0,0]=0,1,2,3,4,5$" \
                        "        A[0] -> 1, B[0,2] -> 2, B[1,0] -> 3, B[1,2] -> 5",
    0
};

static char* help_operator[] = {
    "OPERATORS",        "Precedence (high to low)",
    "Unary",            "-xxx, +xxx",
    "Mul/Div/Mod",      "*, /, %",
    "Add/Sub",          "+, -",
    "Condition",        "==, !=, <, <=, >, >=",
    "Bitwise",          "&, |",
    0
};

static char* help_statement[] = {
    "STATEMENTES",      "",
    "CLS",              "Clear screen",
    "DIM",              "Define array size: DIM[ col, row ]$" \
                        "    NOTE: array size limitation, col * row <= 512",
    "END",              "Terminate program",
    "FOR/NEXT",         "Iterate statements between FOR and NEXT$" \
                        "    ex. S=0:FOR A=1 TO 10:S=S+A:NEXT$" \
                        "    NOTE: increment step is fixed to ONE",
    "GOTO",             "Jump to specified line number: ex. GOTO 100, GOTO X",
    "GOSUB/RETURN",     "Call subroutine / return to caller: ex. GOSUB 200, GOSUB Y"
,
    "IF",               "Conditional execution: if expression is ZERO then goto next
 line$" \
                        "    ex. IF A<B GOTO 100, IF A&0x80 B=128:PRINT \"MSB is on\
"",
#ifdef PI
    "IN",               "Read GPIO (B 1-14) bit level: IN(B) -> 0|1",
#endif // PI
    "INPUT",            "Input data from user: number) INPUT A, string) INPUT @",
#ifdef PI
    "OUT",              "Set GPIO (B 1-14) bit level (L 0 GND|1 Vdd): OUT(B,L)",
    "OUTHZ",            "Set GPIO (B 1-14) bit level (L 0 GND|1 HiZ),$" \
                        "    Internal pull-up (P 0 None|1 Pull): OUTHZ(B,S,P)",
#endif // PI
    "PRINT",            "Print data$" \
                        "    integer: immediate value, variable, array$" \
                        "    string: @$" \
                        "    separator: semicolon=no spacing, comma=tabulation$" \
                        "      PRINT \"H\";\"I\";\"!\" -> HI!$" \
                        "    hexadecimal format: HEX2(xxx), HEX4(xxxxx)",
    "REM",              "Remark: REM, REM \"This is comment string\"",
```

```
    "RND",          "Returns random number (0 to 32767): RND()",
    "MSLEEP",       "Suspend execution for MILLI-seconds: SLEEP(1000) -> 1sec",
    "USLEEP",       "Suspend execution for MICRO-seconds: MLEEP(1000) -> 1usec",

    0
};

static char* help_command[] = {
    "COMMANDS",     "",
    "CLEAR",        "Clear containers",
    "CLS",          "Clear screen",
    "DELETE",       "Delete statement(s): DELETE 100, DELETE 210,290",
    "DUMP",         "Dump containers: DUMP (all), DUMP V (variables), DUMP A (ar
rays),$" \
                    "     DUMP S (string), DUMP L (lines), DUMP B (bytecodes)",
    "END",          "Quit antBASIC",
    "FILES",        "List files in current working directory",
    "FREE",         "Display memory usage",
    "HELP",         "Display help information",
    "LIST",         "List all or part of program: LIST, LIST 100, LIST 210,330",

    "LOAD",         "Load a source file into memory: LOAD \"example/hello.bas\""
,
    "MERGE",        "Merge an additional file into memory: MERGE \"mylib/addon.b
as\"",
    "NEW",          "Clear program",
    "RENUM",        "Renumber program: RENUM, RENUM start, RENUM start,step",
    "RUN",          "Start-up program: CONTROL-C aborts the program",
    "SAVE",         "Save program list to a file: SAVE \"myprogram.bas\"",
    0
};

//
// Initializer
//

void
basic_init(jmp_buf *env)
{
    Exit = env;
    Lnum = Lidx = Gsize = Fsize = 0;
}

//
// Helper functions
//

void
basic_error(int code)
{
    longjmp(*Exit, code);
}

int
basic_readline(char *msg, char *buf, int size)
{
    char *ptr;
    int len;

#ifdef READLINE
    ptr = readline(msg);
    if (ptr) {
        len = strlen(ptr);
        strncpy(buf, ptr, size);
        add_history(ptr);
        free(ptr);
    }
    else
        len = 0;
#else
    printf("%s", msg);
    ptr = fgets(buf, size, stdin);
    if (ptr == NULL)
        len = 0;
    else {
        while (*ptr) {
            if (*ptr == '\n') {
                *ptr = 0;
                break;
            }
        }
        len = strlen(ptr);
    }
#endif // READLINE
    return len;
}

//
// BASIC statements
//

static void
stat_rem(void)
{
    bcode_t b;

    while (bcode_next() != BCODE_EOL) {
        bcode_read(&b);
    }
    DMSG(">>> %05d REM", Lnum);
}

static int
stat_end(void)
{
    bcode_t b;
    word_t num;

    if (bcode_nextiseol())
        num = 0;
    else if (bcode_nextisvar()) {
        bcode_read(&b);
        num = Var[ b.idx ];
    } else
        num = eval();

    return num;
}

static void
stat_dim(void)
{
    bcode_t b;
    int v;
    dim_t size;

    while (1) {
        bcode_read(&b);
        if (b.type != BCODE_TYPE_VARIABLE)
            basic_error(BASIC_NOARRAY);
        v = b.idx;
```

```
        eval_dim(&size);
        if (size.row < 0 || size.col <= 0)
            basic_error(BASIC_ILLRANGE);
        if (size.row == 0) {
            size.row = 1;
            if (size.col > CONT_MAXARRAY)
                basic_error(BASIC_ILLRANGE);
        } else if (size.row * size.col > CONT_MAXARRAY)
            basic_error(BASIC_ILLRANGE);
        Asize[ v ].row = size.row;
        Asize[ v ].col = size.col;
        DMSG(">>> %05d DIM %c[%d,%d]\n", Lnum, v+'A', Asize[ v ].row, Asize[ v ].c
ol);
        if (bcode_next() == BCODE_COLON || bcode_next() == BCODE_EOL)
            return;
        else if (bcode_next() == BCODE_COMMA)
            bcode_skip();
        else
            basic_error(BASIC_ILLARRAY);
    }
}

static void
stat_print(void)
{
    bcode_t b;
    int pos = 0, len, sp, newline = 1;
    word_t res;

    while (1) {
        if (bcode_next() == BCODE_COLON || bcode_next() == BCODE_EOL)
            break;
        else if (bcode_next() == BCODE_STRING) {
            bcode_read(&b);
            len = printf("%s", b.str);
            pos += len;
            newline = 1;
        } else if (bcode_next() == BCODE_AT && \
                   bcode_twoahead() != BCODE_LBRACK) {
            bcode_skip();
            len = printf("%s", String);
            pos += len;
            newline = 1;
        } else if (bcode_next() == BCODE_HEX2) {
            bcode_skip();
            if (bcode_next() != BCODE_LPAREN)
                basic_error(BASIC_SYNTAXERROR);
            bcode_skip();
            res = eval();
            if (bcode_next() != BCODE_RPAREN)
                basic_error(BASIC_SYNTAXERROR);
            bcode_skip();
            len = printf("%02hhX", (unsigned char) res);
            pos += len;
            newline = 1;
        } else if (bcode_next() == BCODE_HEX4) {
            bcode_skip();
            if (bcode_next() != BCODE_LPAREN)
                basic_error(BASIC_SYNTAXERROR);
            bcode_skip();
            res = eval();
            if (bcode_next() != BCODE_RPAREN)
                basic_error(BASIC_SYNTAXERROR);
            bcode_skip();
            len = printf("%04hX", res);
            pos += len;
            newline = 1;
        } else {
            res = eval();
            len = printf("%d", (int) res);
            pos += len;
            newline = 1;
        }
        if (bcode_next() == BCODE_COMMA) {
            bcode_skip();
            sp = 8 - (pos % 8);
            if (sp == 8)
                sp = 0;
            pos += sp;
            while (sp--)
                printf(" ");
            newline = 1;
        } else if (bcode_next() == BCODE_SCOLON) {
            bcode_skip();
            newline = 0;
        }
    }
    if (newline)
        printf("\n");
    fflush(stdout);
    DMSG(">>> %05d PRINT <<<", Lnum);
}

static void
stat_input(void)
{
    int len;
    bcode_t b;
    word_t val;
    char buff[ BASIC_MAXLINECHAR ];

    if (bcode_next() == BCODE_AT) {
        bcode_skip();
        basic_readline("? ", (char *) String, BASIC_MAXLINECHAR);
        DMSG(">>> %05d INPUT @", Lnum);
    } else if (bcode_nextisvar()) {
        bcode_read(&b);
        len = basic_readline("? ", buff, BASIC_MAXLINECHAR);
        if (len == 0)
            val = 0;
        else
            val = (word_t) atoi(buff);
        Var[ b.idx ] = val;
        DMSG(">>> %05d INPUT %c = %d\n", Lnum, 'A' + b.idx, val);
    } else
        basic_error(BASIC_NOVARIABLE);
}

static int
stat_goto(void)
{
    bcode_t b;
    word_t num = 0;

    if (bcode_nextisvar()) {
        bcode_read(&b);
        num = Var[ b.idx ];
    } else
        num = eval();
    DMSG(">>> %05d GOTO %d <<<", Lnum, num);
    return num;
}
```

```
static int
stat_gosub(void)
{
    bcode_t b;
    word_t num = 0;

    if (bcode_nextisvar()) {
        bcode_read(&b);
        num = Var[ b.idx ];
    } else
        num = eval();

    if (Gsize == BASIC_MAXSTACKDEPTH)
        basic_error(BASIC_STACKOVER);
    Gstack[ Gsize ].lidx = Lidx;
    Gstack[ Gsize ].radd = bcode_getpc();
    Gsize++;
    DMSG(">>> %05d GOSUB %d", Lnum, num);
    return num;
}

static void
stat_return(void)
{
    if (Gsize == 0)
        basic_error(BASIC_NOSTACK);
    else
        Gsize--;
    DMSG(">>> %05d RETURN", Lnum);
}

static int
stat_if(void)
{
    word_t res;

    res = eval();
    res = (res != 0) ? 1 : 0;
    DMSG(">>> %05d IF %d", Lnum, res);
    return res;
}

static void
stat_for(void)
{
    bcode_t b;
    int var;
    word_t start, end;

    if (! bcode_nextisvar())
        basic_error(BASIC_NOCONTROLVAR);
    bcode_read(&b);
    var = b.idx;

    if (bcode_next() != BCODE_ASSIGN)
        basic_error(BASIC_NOASSIGN);
    bcode_skip();
    start = eval();

    if (bcode_next() != BCODE_TO)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    end = eval();

    if (start > end)
        basic_error(BASIC_ILLRANGE);

    if (Fsize == BASIC_MAXSTACKDEPTH)
        basic_error(BASIC_STACKOVER);

    Var[ var ] = start;
    Fstack[ Fsize ].lidx = Lidx;
    Fstack[ Fsize ].ladd = bcode_getpc();
    Fstack[ Fsize ].vidx = var;
    Fstack[ Fsize ].ecnt = (int) end;
    Fsize++;
    DMSG(">>> %05d FOR lidx=%d : ladd=%d : var '%c' : start=%d : end=%d\n", \
            Lnum, Lidx, bcode_getpc(), var + 'A', start, end);
}

static void
stat_next(void)
{
    if (Fsize == 0)
        basic_error(BASIC_NOSTACK);
    DMSG(">>> %05d NEXT", Lnum);
}

static void
stat_cls(void)
{
    printf(ESCAPE_HOME ESCAPE_CLS);
    DMSG(">>> %05d CLS", Lnum);
}

static void
stat_usleep(void)
{
    word_t val;

    if (bcode_next() != BCODE_LPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    val = eval();
    if (bcode_next() != BCODE_RPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
#ifdef PI
    gpio_usleep(val);
#else
    usleep(val);
#endif // PI
    DMSG(">>> %05d USLEEP(%d)", Lnum, val);
}

static void
stat_msleep(void)
{
    word_t val;

    if (bcode_next() != BCODE_LPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    val = eval();
    if (bcode_next() != BCODE_RPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    usleep(val * 1000);
    DMSG(">>> %05d MSLEEP(%d)", Lnum, val);
}

#ifdef PI
static void
```

```
stat_out(void)
{
    word_t bit, val;
    int res;

    if (bcode_next() != BCODE_LPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    bit = eval();
    if (bcode_next() != BCODE_COMMA)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    val = eval();
    if (bcode_next() != BCODE_RPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    res = gpio_out(bit, val);
    if (res < 0) {
        res = -res;
        basic_error(res | BASIC_GPIO_ERROR);
    }
    DMSG(">>> %05d OUT(%d,%d)", Lnum, bit, val);
}

static void
stat_outhz(void)
{
    word_t bit, val, pull;
    int res;

    if (bcode_next() != BCODE_LPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    bit = eval();
    if (bcode_next() != BCODE_COMMA)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    val = eval();
    if (bcode_next() != BCODE_COMMA)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    pull = eval();
    if (bcode_next() != BCODE_RPAREN)
        basic_error(BASIC_SYNTAXERROR);
    bcode_skip();
    res = gpio_outhz(bit, val, pull);
    if (res < 0) {
        res = -res;
        basic_error(res | BASIC_GPIO_ERROR);
    }
    DMSG(">>> %05d OUTHZ(%d,%d,%d)", Lnum, bit, val, pull);
}
#endif // PI

//
// BASIC commands
//

#define FIRSTCOLSIZE  14
static void
display_help(char **msg)
{
    char *ptr;
    int cnt = 0, sp, i;

    ptr = *msg++;
    while (ptr) {
        if (cnt == 0)
            printf("%s%s%s", ESCAPE_LRED, ptr, ESCAPE_DEFAULT);
        else
            printf("%s%s%s", ESCAPE_LGREEN, ptr, ESCAPE_DEFAULT);
        sp = FIRSTCOLSIZE - strlen(ptr);
        while (sp--)
            printf(" ");
        if (cnt == 0)
            printf("%s", ESCAPE_LRED);
        ptr = *msg++;
        while (*ptr) {
            if (*ptr == '$') {
                printf("\n");
                for (i = 0; i < FIRSTCOLSIZE; i++)
                    printf(" ");
            } else
                printf("%c", *ptr);
            ptr++;
        }
        if (cnt == 0)
            printf("%s", ESCAPE_DEFAULT);
        printf("\n");
        ptr = *msg++;
        cnt++;
    }
}

static void
cmd_help(void)
{
    bcode_t b;
    int mode = 0;
    char buf[ 3 ], *ptr;

    if (bcode_nextisvar()) {
        bcode_read(&b);
        switch(b.idx) {
            case 'C' - 'A':     // Containers
                mode = 1;
                break;

            case 'O' - 'A':     // Operators
                mode = 2;
                break;

            case 'S' - 'A':     // Statements
                mode = 3;
                break;

            case 'B' - 'A':     // BASIC commands
                mode = 4;
                break;

            default:            // All
                mode = 5;
                break;
        }
    }

    if (mode == 0) {
        printf("---[ Help ]-----------------------------------------------------------
----------\n");
        printf("C)ontainers, O)perators, S)tatements, B)ASIC commands, A)ll (c/o/s
/b/a) ? ");
        ptr = fgets(buf, 3, stdin);;
```

```
        printf("------------------------------------------------------------------
----------\n");
        if (*ptr == 'C' || *ptr == 'c')
            mode = 1;
        else if (*ptr == 'O' || *ptr == 'o')
            mode = 2;
        else if (*ptr == 'S' || *ptr == 's')
            mode = 3;
        else if (*ptr == 'B' || *ptr == 'b')
            mode = 4;
        else
            mode = 5;
    }

    switch (mode) {
        case 1:
            display_help(help_container);
            break;

        case 2:
            display_help(help_operator);
            break;

        case 3:
            display_help(help_statement);
            break;

        case 4:
            display_help(help_command);
            break;

        default:
            display_help(help_container);
            printf("\n");
            display_help(help_operator);
            printf("\n");
            display_help(help_statement);
            printf("\n");
            display_help(help_command);
            break;
    }
}

static void
cmd_free(void)
{
    printf(ESCAPE_LBLUE);
    printf("%d lines, %d lines free\n", Lsize, PROG_MAXLINE - Lsize);
    printf("%d bytes used, %d bytes free\n", Psize, PROG_MAXSIZE - Psize);
    printf(ESCAPE_DEFAULT);
}

static void
cmd_dump(void)
{
    bcode_t b;

    if (bcode_nextisvar()) {
        bcode_read(&b);
        switch(b.idx) {
            case 'V' - 'A':
                cont_dumpvar();
                return;

            case 'A' - 'A':
                cont_dumparr();
                return;

            case 'S' - 'A':
                cont_dumpstr();
                return;

            case 'B' - 'A':
                prog_dumpbytes();
                return;

            case 'L' - 'A':
                prog_dumplines();
                return;
        }
    }
    cont_dumpvar();
    printf("\n");
    cont_dumparr();
    printf("\n");
    cont_dumpstr();
    printf("\n");
    prog_dumpbytes();
    printf("\n");
    prog_dumplines();
}

static void
cmd_new(void)
{
    prog_init();
    Lnum = Lidx = Gsize = Fsize = 0;
}

static void
cmd_clear(void)
{
    cont_init();
}

static void
cmd_list(void)
{
    bcode_t b;
    int start, end, ret;

    start = Lines[ 1 ].num;
    end = Lines[ Lsize ].num;
    if (bcode_next() == BCODE_NUMBER10) {
        bcode_read(&b);
        start = b.num;
        if (start < 1 || start >= PROG_BLANKLINE)
            basic_error(BASIC_ILLRANGE);
        end = start;
        if (bcode_next() == BCODE_COMMA) {
            bcode_skip();
            if (bcode_next() == BCODE_NUMBER10) {
                bcode_read(&b);
                end = b.num;
                if (end < 1 || end >= PROG_BLANKLINE || start > end)
                    basic_error(BASIC_ILLRANGE);
            } else
                basic_error(BASIC_ILLRANGE);
        }
    } else if (bcode_next() != BCODE_EOL)
        basic_error(BASIC_ILLRANGE);
    ret = prog_list(stdout, PROG_LISTCOLOR, start, end);
```

```
    if (ret)
        basic_error(ret | BASIC_PROG_ERROR);
}

static void
cmd_renum(void)
{
    bcode_t b;
    int start = 0, step = 0, ret;

    if (bcode_next() == BCODE_EOL) {
        start = BASIC_LINESTART;
        step = BASIC_LINESTEP;
    } else if (bcode_nextisnum()) {
        bcode_read(&b);
        start = b.num;
        if (bcode_next() == BCODE_COMMA) {
            bcode_skip();
            if (! bcode_nextisnum())
                basic_error(BASIC_ILLRANGE);
            else {
                bcode_read(&b);
                step = b.num;
            }
        } else if (bcode_next() == BCODE_EOL)
            step = BASIC_LINESTEP;
        else
            basic_error(BASIC_ILLRANGE);
    } else
        basic_error(BASIC_ILLRANGE);

    if (start < 1 || step < 1 || (start + step * Lsize) >= PROG_BLANKLINE)
        basic_error(BASIC_ILLRANGE);

    ret = prog_renum(start, step);
    if (ret)
        basic_error(ret | BASIC_PROG_ERROR);
}

static void
cmd_delete(void)
{
    bcode_t b;
    int start = 0, end = 0, ret;

    if (bcode_next() == BCODE_NUMBER10) {
        bcode_read(&b);
        start = b.num;
        if (start < 1 || start >= PROG_BLANKLINE)
            basic_error(BASIC_ILLRANGE);
        end = start;
        if (bcode_next() == BCODE_COMMA) {
            bcode_skip();
            if (bcode_next() == BCODE_NUMBER10) {
                bcode_read(&b);
                end = b.num;
                if (end < 1 || end >= PROG_BLANKLINE || start > end)
                    basic_error(BASIC_ILLRANGE);
            } else
                basic_error(BASIC_ILLRANGE);
        }
    } else
        basic_error(BASIC_NOLINENUM);
    ret = prog_delete(start, end);
    if (ret)
        basic_error(ret | BASIC_PROG_ERROR);
}

static void
cmd_files(void)
{
    bcode_t b;
    struct dirent **list;
    char *dir;
    int i, cnt;

    if (bcode_nextiseol())
        dir = "./";
    else if (! bcode_nextisstr())
        basic_error(BASIC_NODIRNAME);
    else {
        bcode_read(&b);
        dir = (char *) b.str;
    }
    cnt = scandir(dir, &list, NULL, alphasort);
    if (cnt == 0)
        basic_error(BASIC_DIRNOTFOUND);

    for (i = 0; i < cnt; i++) {
        if (list[ i ]->d_name[0] == '.')
            continue;
        if (list[ i ]->d_type == DT_DIR) {
            printf(ESCAPE_LGREEN);
            printf("%s/\n", list[ i ]->d_name);
        } else {
            printf(ESCAPE_LBLUE);
            printf("%s\n", list[ i ]->d_name);
        }
    }
    printf(ESCAPE_DEFAULT);
    free(list);
}

static void
cmd_load(void)
{
    bcode_t b;
    FILE *src;
    char *ptr;
    byte_t code[ BCODE_MAXSIZE ];
    int size, line, ret;

    if (! bcode_nextisstr())
        basic_error(BASIC_NOFILENAME);
    bcode_read(&b);
    printf("Loading file \"%s\"\n", b.str);
    src = fopen((char *) b.str, "r");
    if (src == NULL)
        basic_error(BASIC_FILEIOERROR);

    prog_init();
    Mode = BASIC_MODELOAD;
    while (1) {
        ptr = fgets(Text, TOKEN_MAXBUF, src);
        if (ptr == NULL)
            break;
        while (*ptr) {
            if (*ptr == '\n')
                *ptr = 0;
            ptr++;
        }
        token_source(Text);
```

```
        size = bcode_compile(code);
        if (size < 0) {
            fclose(src);
            size = -size;
            basic_error(size | BASIC_BCODE_ERROR);
        }
        bcode_start(code);
        if (! bcode_nextisnum()) {
            fclose(src);
            basic_error(BASIC_NOLINENUM);
        }
        bcode_read(&b);
        line = b.num;
        bcode_read(&b);
        ret = prog_insert(line, &code[ b.pos ], size - 3);
        if (ret) {
            fclose(src);
            basic_error(ret | BASIC_PROG_ERROR);
        }
    }
    fclose(src);
    printf("%d lines loaded\n", Lsize);
}

static void
cmd_merge(void)
{
    bcode_t b;
    FILE *src;
    char *ptr;
    byte_t code[ BCODE_MAXSIZE ];
    int cnt, size, line, ret;

    if (! bcode_nextisstr())
        basic_error(BASIC_NOFILENAME);
    bcode_read(&b);
    printf("Merging file \"%s\"\n", b.str);
    src = fopen((char *) b.str, "r");
    if (src == NULL)
        basic_error(BASIC_FILEIOERROR);

    cnt = 0;
    Mode = BASIC_MODELOAD;
    while (1) {
        ptr = fgets(Text, TOKEN_MAXBUF, src);
        if (ptr == NULL)
            break;
        while (*ptr) {
            if (*ptr == '\n')
                *ptr = 0;
            ptr++;
        }
        token_source(Text);
        size = bcode_compile(code);
        if (size < 0) {
            fclose(src);
            size = -size;
            basic_error(size | BASIC_BCODE_ERROR);
        }
        bcode_start(code);
        if (! bcode_nextisnum()) {
            fclose(src);
            basic_error(BASIC_NOLINENUM);
        }
        bcode_read(&b);
        line = b.num;
        bcode_read(&b);
        ret = prog_insert(line, &code[ b.pos ], size - 3);
        if (ret) {
            fclose(src);
            basic_error(ret | BASIC_PROG_ERROR);
        }
        cnt++;
    }
    fclose(src);
    printf("%d lines merged\n", cnt);
}

static void
cmd_save(void)
{
    bcode_t b;
    FILE *dst;
    int ret;

    if (! bcode_nextisstr())
        basic_error(BASIC_NOFILENAME);
    bcode_read(&b);
    DMSG("Saving file \"%s\"\n", b.str);
    dst = fopen((char *) b.str, "w");
    if (dst == NULL)
        basic_error(BASIC_FILEIOERROR);
    ret = prog_list(dst, PROG_LISTPLAIN, Lines[ 1 ].num, Lines[ Lsize ].num);
    if (ret) {
        fclose(dst);
        basic_error(ret | BASIC_PROG_ERROR);
    }
    fclose(dst);
    printf("%d lines saved\n", Lsize);
}

static void
deny_direct(void)
{
    if (Mode == BASIC_MODEDIRECT)
        basic_error(BASIC_DIRECTDENY);
}

int
basic_load(char *fname)
{
    bcode_t b;
    FILE *src;
    char *ptr;
    byte_t code[ BCODE_MAXSIZE ];
    int size, line, ret;

    src = fopen((char *) fname, "r");
    if (src == NULL)
        return 1;           // 1: File open failure

    prog_init();
    Mode = BASIC_MODELOAD;
    while (1) {
        ptr = fgets(Text, TOKEN_MAXBUF, src);
        if (ptr == NULL)
            break;
        while (*ptr) {
            if (*ptr == '\n')
                *ptr = 0;
            ptr++;
        }
```

```
        token_source(Text);
        size = bcode_compile(code);
        if (size < 0) {
            fclose(src);
            size = -size;
            return 2;       // 2: Bytecode compile error
        }
        bcode_start(code);
        if (! bcode_nextisnum()) {
            fclose(src);
            return 3;       // 3: Line number undefined
        }
        bcode_read(&b);
        line = b.num;
        bcode_read(&b);
        ret = prog_insert(line, &code[ b.pos ], size - 3);
        if (ret) {
            fclose(src);
            return 4;       // 4: Program error
        }
    }
    fclose(src);
    return 0;
}

int
basic_exec(void)
{
    bcode_t b;
    int line, jump = 0, back = 0, skip = 0, newpc, var;

    if (Mode == BASIC_MODEDIRECT)
        line = 0;
    else if (Mode == BASIC_MODERUN) {
        if (Lsize == 0)
            return 0;
        cont_init();
        line = 1;
    } else if (Mode == BASIC_MODESHELL) {
        if (Lsize == 0)
            return 0;
        line = 1;
    }

    Gsize = Fsize = 0;
    bcode_start(Program);
    while (Lines[ line ].len) {
        Lidx = line;
        Lnum = Lines[ line ].num;
#ifdef DEBUG
        if (Lnum)
            DMSG("Executing line %d", Lnum)
        else
            DMSG("Executing COMMAND line")
#endif // DEBUG
        if (back)
            bcode_setpc(newpc);
        else
            bcode_setpc(Lines[ line ].add);
        jump = back = skip = 0;
        while (1) {
            bcode_read(&b);
            if (b.type == BCODE_TYPE_EOL)
                break;
            else if (b.inst == BCODE_COLON)
                continue;
            else if (b.type == BCODE_TYPE_KEYWORD) {
                switch (b.inst) {
                    case BCODE_END:
                        return(stat_end());

                    case BCODE_CLS:
                        stat_cls();
                        break;

                    case BCODE_DIM:
                        stat_dim();
                        break;

                    case BCODE_FOR:
                        deny_direct();
                        stat_for();
                        break;

                    case BCODE_GOSUB:
                        jump = stat_gosub();
                        break;

                    case BCODE_GOTO:
                        jump = stat_goto();
                        break;

                    case BCODE_IF:
                        skip = stat_if() ? 0 : 1;
                        break;

                    case BCODE_INPUT:
                        stat_input();
                        break;

                    case BCODE_NEXT:
                        deny_direct();
                        stat_next();
                        var = Fstack[ Fsize - 1 ].vidx;
                        Var[ var ]++;
                        if (Var[ var ] <= Fstack[ Fsize - 1 ].ecnt) {
                            line = Fstack[ Fsize - 1 ].lidx;
                            newpc = Fstack[ Fsize - 1 ].ladd;
                            back = 1;
                        } else {
                            Fsize--;
                            back = 0;
                        }
                        break;
#ifdef PI
                    case BCODE_OUT:
                        stat_out();
                        break;

                    case BCODE_OUTHZ:
                        stat_outhz();
                        break;
#endif // PI
                    case BCODE_PRINT:
                        stat_print();
                        break;

                    case BCODE_REM:
                        deny_direct();
                        stat_rem();
                        break;
```

```
                    case BCODE_RETURN:
                        deny_direct();
                        stat_return();
                        line = Gstack[ Gsize ].lidx;
                        newpc = Gstack[ Gsize ].radd;
                        back = 1;
                        break;

                    case BCODE_MSLEEP:
                        stat_msleep();
                        break;

                    case BCODE_USLEEP:
                        stat_usleep();
                        break;

                    default:
                        basic_error(BASIC_SYNTAXERROR);
                } // switch
            } else if (b.type == BCODE_TYPE_VARIABLE)
                eval_assign(b.idx + 1);
            else if (b.inst == BCODE_AT)
                eval_assign(0);
            else
                basic_error(BASIC_SYNTAXERROR);
            if (jump || back || skip)
                break;
        } // inner while
        if (jump) {
            line = prog_search(jump);
            if (line < 0)
                basic_error(PROG_LINENOTFOUND | BASIC_PROG_ERROR);
        } else if (back)
            continue;
        else {
            if (line == 0)  // Terminate DIRECT mode
                return 0;
            line++;
        }
    } // outer while
    return 0;
}

int
basic_command(int inst)
{
    switch (inst) {
        case BCODE_CLEAR:
            cmd_clear();
            break;

        case BCODE_CLS:
            stat_cls();
            break;

        case BCODE_DELETE:
            cmd_delete();
            break;

        case BCODE_DUMP:
            cmd_dump();
            break;

        case BCODE_FILES:
            cmd_files();
            break;

        case BCODE_FREE:
            cmd_free();
            break;

        case BCODE_HELP:
            cmd_help();
            break;

        case BCODE_LIST:
            cmd_list();
            break;

        case BCODE_LOAD:
            cmd_load();
            break;

        case BCODE_MERGE:
            cmd_merge();
            break;

        case BCODE_NEW:
            cmd_new();
            break;

        case BCODE_RENUM:
            cmd_renum();
            break;

        case BCODE_RUN:
            Mode = BASIC_MODERUN;
            basic_exec();
            break;

        case BCODE_SAVE:
            cmd_save();
            break;

        default:
            return 1;
    }
    return 0;
}

char *
basic_result(int code)
{
    return Results[ code ];
}
```

```
//
//  p i _ g p i o . h
//
//  BAREMETALHACK.COM --> public domain
//

#ifndef _PI_GPIO_H_
#define _PI_GPIO_H_

// Raspberry Pi SOC type extracted from revision code
// NOTE: pull-up/dowun register handling is different between BCM283x and BCM2711

#define GPIO_REVISIONPATH   "/proc/device-tree/system/linux,revision"
#define GPIO_BCM283x        0       // BCM2835: Pi original, Pi Zero
                                    // BCM2836: Pi 2, Pi 3
#define GPIO_BCM2711        1       // Pi 4, Pi 400
#define GPIO_UPDN_WAIT      1       // Microseconds to wait while pull-up/down
                                    // control in BCM283x
                                    // "BCM2835 ARM Peripheral" says "150cycles"
                                    //    GPIO clock = 250MHz (GPU)
                                    //    150/250M = 0.6 microseconds

// Linux ROOTLESS access to GPIOMEM

#define GPIO_PAGEPATH       "/dev/gpiomem"
#define GPIO_PAGESIZE       4096

// Re-arranged BMH (Bare Metal Hacking) bits: BMH1-BMH14
//      GPIO2, GPIO3, GPIO4, GPIO17, GPIO27, GPIO22, GPIO10, GPIO9, GPIO11,
//      GPIO5, GPIO6, GPIO13, GPIO19, GPIO26

#define GPIO_BITS           14      // BMH1 to BMH14

// GPIO control register index (32bit WORD offset address)

#define FSEL0               0       // Function select 0 (GPIO0 to GPIO9)
#define FSEL1               1       // Function select 1 (GPIO10 to GPIO19)
#define FSEL2               2       // Function select 2 (GPIO20 to GPIO29)
#define SET0                7       // Pin output set 0 (GPIO0 to GPIO31)
#define CLR0                10      // Pin output clear 0 (GPIO0 to GPIO31)
#define LEV0                13      // Pin level 0 (GPIO0 to GPIO31)

// GPIO pull-up/down control registers in BCM2711

#define UPDN0               57      // Pull-up pull-down 0 (GPIO0 to GPIO15)
#define UPDN1               58      // Pull-up pull-down 0 (GPIO16 to GPIO31)

// GPIO pull-up/down control registers in BCM283x

#define GPPUD               37      // Gpio Pull-Up/Down enable register
#define GPPUDCLK0           38      // Gpio Pull-Up/Down enable CLocK 0 register

// Bit values

#define GPIO_HIGH           1
#define GPIO_LOW            0

// Bit definition

#define GPIO_INPUT          0
#define GPIO_OUTPUT         1
#define GPIO_OUTPUTHZ       2
#define GPIO_UNSPECIFIED    (-1)

// Register mode

#define GPIO_REGNONE        0       // NO register is connected
#define GPIO_REGUP          1       // Pull-UP register is connected
#define GPIO_REGDOWN        2       // Pull-DOWN register is connected

// Result codes

#define GPIO_SUCCESS        0
#define GPIO_SOCUNKNOWN     (-1)
#define GPIO_OPENFAIL       (-2)
#define GPIO_ILLBIT         (-3)
#define GPIO_ILLDIR         (-4)
#define GPIO_ILLREG         (-5)

//
// Environment variable which holds calibrated loop count for MICROSECOND wait
//

#define GPIO_MICROENV       "ANT_MICROWAIT"

//
// Function prototypes
//

   int gpio_init(void);
  void gpio_term(void);
   int gpio_bitmode(int, int, int);
   int gpio_bitread(int);
   int gpio_bitwrite(int, int);
   int gpio_in(int);
   int gpio_out(int, int);
   int gpio_outhz(int, int, int);
char * gpio_result(int);
  void gpio_usleep(int);

#endif // _PI_GPIO_H
```

```
//
//  p i _ g p i o . c
//
//  NOTICE: Supported SOCs are BCM2835, BCM2836, BCM2837, BCM2711
//
//  BAREMETALHACK.COM --> public domain
//

#include <fcntl.h>          // open()
#include <unistd.h>         // close()
#include <sys/mman.h>       // mmap(), munmap()
#include <stdint.h>         // uint32_t
#include <stdlib.h>         // atoi()
#include <unistd.h>         // usleep();
#include "pi_gpio.h"

//
// File scope variables
//

static int Soc;
static volatile uint32_t *Gpio_base;
static uint32_t Save_gpio_fsel0, Save_gpio_fsel1, Save_gpio_fsel2;
static uint32_t Save_gpio_set0, Save_gpio_clr0;
static uint32_t Save_gpio_updn0, Save_gpio_updn1;
static int States[ GPIO_BITS ];
static int Microsec;

// Bitmapping array: converts from re-arranged BMH (Bare Metal Hacking) number to
// original BCM (Broadcom) number

static int Bitmap[] = {
    0,
    2,      // BMH1  --> GPIO2
    3,      // BMH2  --> GPIO3
    4,      // BMH3  --> GPIO4
    17,     // BMH4  --> GPIO17
    27,     // BMH5  --> GPIO27
    22,     // BMH6  --> GPIO22
    10,     // BMH7  --> GPIO10
    9,      // BMH8  --> GPIO9
    11,     // BMH9  --> GPIO11
    5,      // BMH10 --> GPIO5
    6,      // BMH11 --> GPIO6
    13,     // BMH12 --> GPIO13
    19,     // BMH13 --> GPIO19
    26      // BMH14 --> GPIO26
};

static char *Results[] = {
    "GPIO ok",                      // GPIO_SUCCESS
    "GPIO unknown SOC",             // GPIO_SOCUNKNOWN
    "GPIO open failure",            // GPIO_OPENFAIL
    "GPIO illegal bit number",      // GPIO_ILLBIT
    "GPIO illegal direction",       // GPIO_ILLDIR
    "GPIO illegal register mode",   // GPIO_ILLREG
};

//
// GPIO control functions
//

int
gpio_init(void)
{
    int fd, ret, i;
    uint8_t rev[ 4 ];
    char *ptr;

    fd = open(GPIO_REVISIONPATH, O_RDONLY);
    if (fd < 0)
        return GPIO_SOCUNKNOWN;
    ret = read(fd, rev, sizeof(rev));
    close(fd);
    if (ret != sizeof(rev))
        return GPIO_SOCUNKNOWN;
    switch ((rev[ 2 ] & 0xF0) >> 4) {
        case 0:
        case 1:
        case 2:
            Soc = GPIO_BCM283x;
            break;

        case 3:
            Soc = GPIO_BCM2711;
            break;

        default:
            return GPIO_SOCUNKNOWN;
    };

    fd = open(GPIO_PAGEPATH, O_RDWR, O_SYNC, O_CLOEXEC);
    if (fd < 0)
        return GPIO_OPENFAIL;

    Gpio_base = (uint32_t *) mmap(NULL, GPIO_PAGESIZE, PROT_READ | PROT_WRITE, \
            MAP_SHARED, fd, 0);
    close(fd);
    if (Gpio_base == MAP_FAILED)
        return GPIO_OPENFAIL;

    for (i = 0; i < GPIO_BITS; i++)
        States[ i ] = GPIO_UNSPECIFIED;

    Save_gpio_fsel0 = Gpio_base[ FSEL0 ];
    Save_gpio_fsel1 = Gpio_base[ FSEL1 ];
    Save_gpio_fsel2 = Gpio_base[ FSEL2 ];
    Save_gpio_set0  = Gpio_base[ SET0 ];
    Save_gpio_clr0  = Gpio_base[ CLR0 ];
    Save_gpio_updn0 = Gpio_base[ UPDN0 ];
    Save_gpio_updn1 = Gpio_base[ UPDN1 ];

    ptr = getenv(GPIO_MICROENV);
    if (ptr)
        Microsec = atoi(ptr);
    else
        Microsec = 0;

    return GPIO_SUCCESS;
}

void
gpio_term(void)
{
    Gpio_base[ FSEL0 ] = Save_gpio_fsel0;
    Gpio_base[ FSEL1 ] = Save_gpio_fsel1;
    Gpio_base[ FSEL2 ] = Save_gpio_fsel2;
    Gpio_base[ SET0 ]  = Save_gpio_set0;
    Gpio_base[ CLR0 ]  = Save_gpio_clr0;
    Gpio_base[ UPDN0 ] = Save_gpio_updn0;
    Gpio_base[ UPDN1 ] = Save_gpio_updn1;
```

```
    munmap((void*) Gpio_base, GPIO_PAGESIZE);
}

int
gpio_bitmode(int bmh, int dir, int reg)
{
    int bcm, idx;
    uint32_t val, mask;

    if (bmh < 1 || bmh > GPIO_BITS)
        return GPIO_ILLBIT;
    if (dir != GPIO_INPUT && dir != GPIO_OUTPUT)
        return GPIO_ILLDIR;
    if (reg != GPIO_REGNONE && reg != GPIO_REGUP && reg != GPIO_REGDOWN)
        return GPIO_ILLREG;

    bcm = Bitmap[ bmh ];
    idx = FSEL0 + (bcm / 10);
    val = Gpio_base[ idx ];
    mask = ~(0b111 << (3 * (bcm % 10)));
    val &= mask;
    if (dir == GPIO_OUTPUT)
        val |= (0b001 << (3 * (bcm % 10)));
    Gpio_base[ idx ] = val;

    if (Soc == GPIO_BCM2711) {
        idx = UPDN0 + (bcm / 16);
        val = Gpio_base[ idx ];
        mask = ~(0b11 << (2 * (bcm % 16)));
        val &= mask;
        if (reg == GPIO_REGUP)
            val |= (0b01 << (2 * (bcm % 16)));
        else if (reg == GPIO_REGDOWN)
            val |= (0b10 << (2 * (bcm % 16)));
        Gpio_base[ idx ] = val;
    } else {    // BMC283x
        switch (reg) {
            case GPIO_REGNONE:
                val = 0b00;
                break;

            case GPIO_REGUP:
                val = 0b10;
                break;

            case GPIO_REGDOWN:
                val = 0x01;
                break;
        };
        // The following protocol is adopted from "BCM2835 ARM Peripherals"

        Gpio_base[ GPPUD ] = val;               // Update GPPUD
        gpio_usleep(GPIO_UPDN_WAIT);            // Wait more than 150 cycles
        Gpio_base[ GPPUDCLK0 ] = 1 << bcm;      // Assert clock
        gpio_usleep(GPIO_UPDN_WAIT);            // Wait more than 150 cycles
        Gpio_base[ GPPUD ] = 0;                 // Remove GPPUD control signal
        Gpio_base[ GPPUDCLK0 ] = 0;             // Remove GPPUDCLK0 clock signal
    }

    return GPIO_SUCCESS;
}

int
gpio_bitread(int bmh)
{
    int bcm;

    if (bmh < 1 || bmh > GPIO_BITS)
        return GPIO_ILLBIT;

    bcm = Bitmap[ bmh ];
    if (Gpio_base[ LEV0 ] & (1 << bcm))
        return GPIO_HIGH;
    else
        return GPIO_LOW;
}

int
gpio_bitwrite(int bmh, int value)
{
    int bcm, idx;

    if (bmh < 1 || bmh > GPIO_BITS)
        return GPIO_ILLBIT;

    bcm = Bitmap[ bmh ];
    if (value)
        idx = SET0;
    else
        idx = CLR0;
    Gpio_base[ idx ] = (1 << bcm);

    return GPIO_SUCCESS;
}

int
gpio_in(int bmh)
{
    int ret;

    if (States[ bmh ] == GPIO_INPUT)
        return gpio_bitread(bmh);
    else {
        ret = gpio_bitmode(bmh, GPIO_INPUT, GPIO_REGUP);
        if (ret < 0)
            return ret;
        else {
            States[ bmh ] = GPIO_INPUT;
            return gpio_bitread(bmh);
        }
    }
}

int
gpio_out(int bmh, int val)
{
    int ret;

    if (States[ bmh ] == GPIO_OUTPUT)
        return gpio_bitwrite(bmh, val);
    else {
        ret = gpio_bitmode(bmh, GPIO_OUTPUT, GPIO_REGNONE);
        if (ret < 0)
            return ret;
        else {
            States[ bmh ] = GPIO_OUTPUT;
            return gpio_bitwrite(bmh, val);
        }
    }
}

int
```

```
gpio_outhz(int bmh, int val, int pull)
{
    int ret, p;

    if (val) {
        p = pull ? GPIO_REGUP : GPIO_REGNONE;
        ret = gpio_bitmode(bmh, GPIO_INPUT, p);
        if (ret)
            return ret;
    } else {
        ret = gpio_bitmode(bmh, GPIO_OUTPUT, GPIO_REGNONE);
        if (ret)
            return ret;
        ret = gpio_bitwrite(bmh, 0);
        if (ret)
            return ret;
    }
    States[ bmh ] = GPIO_OUTPUTHZ;
    return GPIO_SUCCESS;
}

char *
gpio_result(int code)
{
    return Results[ code ];
}

void
gpio_usleep(int utime)
{
    int loops;
    volatile int i;     // We need "volatile" for prevention of code optimization

    if (Microsec == 0) {
        usleep(utime);
        return;
    } else {
        loops = utime * Microsec;
        for (i = 0; i < loops; i++)
            ;
    }
}
```

```
//
// m a i n . c
//
// BAREMETALHACK.COM --> public domain
//

#include <stdio.h>      // FILE{}, printf()
#include <setjmp.h>     // jmp_buf{}, setjmp()
#include <signal.h>     // signal()
#include <stdlib.h>     // atoi()
#include <string.h>     // strncpy()

#include "token.h"
#include "bcode.h"
#include "eval.h"
#include "program.h"
#include "basic.h"
#include "escape.h"
#include "debug.h"

#ifdef PI
#include "pi_gpio.h"
#endif //PI

//
// Global variables
//

FILE *Debug;
jmp_buf IntRecovery, EvalRecovery, BasicRecovery;
int Verbose = 1;

//
// Helper functions
//

static void
welcome(void)
{
    if (Verbose) {
        printf("\n=== Doctor BMH's %s%s%s v%d.%d.%d ===\n", \
                ESCAPE_LGREEN, BASIC_NAME, ESCAPE_DEFAULT, \
                BASIC_MAJOR, BASIC_MINOR, BASIC_PATCH);
        printf("  (enter %sHELP%s for summary, %sEND%s for exit)\n", \
                ESCAPE_LBLUE, ESCAPE_DEFAULT, ESCAPE_LBLUE, ESCAPE_DEFAULT);
        printf("\n");
    }
}

static void
sigint_handler()
{
    signal(SIGINT, sigint_handler);
    longjmp(IntRecovery, 0);
}

static void
location(void)
{
    printf("%s", ESCAPE_LGREEN);
    if (Mode == BASIC_MODELOAD)
        printf(" in \"%s\"", Text);
    else if (Mode == BASIC_MODEDIRECT) {
        if (Pnum)
            printf(" in line %d", Pnum);
    } else
        printf(" in line %d", Lnum);
    printf("%s\n", ESCAPE_DEFAULT);
}

static int
is_command(int inst) {
    switch (inst) {
        case BCODE_CLEAR:
        case BCODE_CLS:
        case BCODE_DUMP:
        case BCODE_DELETE:
        case BCODE_FILES:
        case BCODE_FREE:
        case BCODE_HELP:
        case BCODE_LIST:
        case BCODE_LOAD:
        case BCODE_MERGE:
        case BCODE_NEW:
        case BCODE_RENUM:
        case BCODE_RUN:
        case BCODE_SAVE:
            return 1;

        default:
            return 0;
    }
}

//
// Main function
//

int
main(int argc, char *argv[])
{
    char text[ TOKEN_MAXBUF ], *msg;
    byte_t cmd[ BCODE_MAXSIZE ];
    int ret, size, num;
    bcode_t b;
    word_t val;

    // Initializer

    if (argc >= 2)
        Verbose = 0;
    Debug = stderr;
    token_init();
    eval_init(&EvalRecovery);
    cont_init();
    prog_init();
    basic_init(&BasicRecovery);
#ifdef PI
    ret = gpio_init();
    if (ret < 0) {
        printf("%sCould not identify Raspberry Pi on this system.%s\n", \
            ESCAPE_LRED, ESCAPE_DEFAULT);
        return 255;
    }
#endif // PI
    signal(SIGINT, sigint_handler);
    welcome();

    // === Shell mode ===

    if (argc >= 2) {                    // Program filename is specified
        if (argc >= 3)                  // --- First number argument is specified
```

```
        Var[ 0 ] = atoi(argv[ 2 ]); // --- Store the value in variable 'A'
    if (argc >= 4)                  // --- Second number argument is specified

        Var[ 1 ] = atoi(argv[ 3 ]); // --- Store the value in variable 'B'
    if (argc >= 5)                  // --- String argument is specified
        strncpy((char*) String, (const char*) argv[ 4 ], CONT_MAXSTRING);
                                    // --- Store the value in string array '@'

    ret = basic_load(argv[ 1 ]);    // --- Load the program
    if (ret) {
        printf("%santBASIC: load error%s\n", ESCAPE_LRED, ESCAPE_DEFAULT);
        return 1;
    }
    if ((ret = setjmp(IntRecovery)) != 0) {
        printf("\n%santBASIC: user aborted%s\n", ESCAPE_LRED, ESCAPE_DEFAULT);

        return 2;
    }
    if ((ret = setjmp(EvalRecovery)) != 0) {
        printf("%santBASIC: eval() error%s\n", ESCAPE_LRED, ESCAPE_DEFAULT);
        return 3;
    }
    if ((ret = setjmp(BasicRecovery)) != 0) {
        printf("%santBASIC: syntax error%s\n", ESCAPE_LRED, ESCAPE_DEFAULT);
        return 4;
    }
    Mode = BASIC_MODESHELL;         // --- Switch to SHELL mode
    ret = basic_exec();             // --- Execute the program
    return ret;                     // --- return exit status to the shell
}

// Error handler

if ((ret = setjmp(IntRecovery)) != 0) {
    if (Lnum) {
        printf("\n%sAborted ", ESCAPE_LRED);
        location();
    } else
        printf("\n");
}
if ((ret = setjmp(EvalRecovery)) != 0) {
    printf("%sError: %s", ESCAPE_LRED, eval_result(ret));
    location();
}
if ((ret = setjmp(BasicRecovery)) != 0) {
    if (ret & BASIC_BCODE_ERROR) {
        ret -= BASIC_BCODE_ERROR;
        msg = bcode_result(ret);
    } else if (ret & BASIC_PROG_ERROR) {
        ret -= BASIC_PROG_ERROR;
        msg = prog_result(ret);
    }
#ifdef PI
    else if (ret & BASIC_GPIO_ERROR) {
        ret -= BASIC_GPIO_ERROR;
        msg = gpio_result(ret);
    }
#endif
    else
        msg = basic_result(ret);
    printf("%sError: %s", ESCAPE_LRED, msg);
    location();
}

// === Interactive direct mode ===

while (1) {
    Mode = BASIC_MODEDIRECT;                    // Enter direct mode
    Pnum = 0;
    ret = basic_readline("> ", text, TOKEN_MAXBUF);
    if (ret == 0)
        continue;
    token_source(text);
    size = bcode_compile(cmd);
    if (size < 0) {
        printf("%sError: %s", ESCAPE_LRED, bcode_result(-size));
        location();
        continue;
    }
#ifdef DEBUG
    bcode_dump(cmd);
    fprintf(Debug, "\n");
#endif // DEBUG

    bcode_start(cmd);
    bcode_read(&b);
    if (b.type == BCODE_TYPE_NUMBER10) {        // Edit program
        num = b.num;
        if (bcode_next() == BCODE_EOL) {        // --- delete a line
            ret = prog_delete(num, num);
            if (ret) {
                printf("%sError: %s", ESCAPE_LRED, prog_result(ret));
                location();
            }
        } else {                                // --- insert a line
            bcode_read(&b);
            ret = prog_insert(num, &cmd[ b.pos ], size - 3);
            if (ret) {
                printf("%sError: %s", ESCAPE_LRED, prog_result(ret));
                location();
            }
        }
        continue;
    } else if (b.inst == BCODE_INQ) {           // Evaluate expression
        if (bcode_next() == BCODE_AT && bcode_twoahead() != BCODE_LBRACK) {
            bcode_skip();
            printf("%s%s%s\n", ESCAPE_LBLUE, String, ESCAPE_DEFAULT);
        } else {
            val = eval();
            printf("%s%d%s\n", ESCAPE_LBLUE, val, ESCAPE_DEFAULT);
        }
        continue;
    } else if (b.type == BCODE_TYPE_KEYWORD) {
        if (b.inst == BCODE_END)                // END
            break;                              // --- exit loop
        else if (is_command(b.inst)) {          // BASIC commands
            basic_command(b.inst);              // --- execute it
            continue;
        }
    }                                           // Direct statement execution
    prog_cmdline(cmd, size);                    // --- store one liner program

    basic_exec();                               // --- execute it
}
#ifdef PI
    gpio_term();
#endif // PI
    return 0;
}
```

```
#include <stdio.h>          // printf()
#include <sys/time.h>       // *_t, timeval{}, gettimeofday()
#include <stdlib.h>         // atoi()

void microsec_wait(int count) {
    volatile int i;

    for (i = 0; i < count; i++) {
    };
}

int main(int argc, char* argv[]) {
    struct timeval start, end;
    time_t sec;
    suseconds_t usec;
    float usecs;
    int count, loops, i;

    if (argc != 3) {
        printf("Usage: antcalib loopcount loops\n");
        return 1;
    }

    count = atoi(argv[ 1 ]);
    loops = atoi(argv[ 2 ]);
    printf("Loopcount = %d\n", count);
    printf("Number of loops = %d\n\n", loops);
    gettimeofday(&start, NULL);
    for (i = 0; i < loops; i++)
        microsec_wait(count);
    gettimeofday(&end, NULL);
    sec = end.tv_sec - start.tv_sec;
    if (end.tv_usec < start.tv_usec) {
        sec--;
        usec = 1000000 + end.tv_usec - start.tv_usec;
    } else
        usec = end.tv_usec - start.tv_usec;
    usecs = sec * 1000000 + usec;
    printf("Elapsed time --> %ld sec %ld usec\n", sec, usec);
    printf("Mean time --> %f usec/loop\n", usecs / loops);
    return 0;
}
```