/\*

chessengine.c

"Little Rook Chess" (lrc)

Port to u8g library

chess for embedded 8-Bit controllers

Copyright (c) 2012, olikraus@gmail.com

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,

are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this list

of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice, this

list of conditions and the following disclaimer in the documentation and/or other

materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND

CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR

CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,

SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT

NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER

CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,

STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF

ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Note:

UNIX\_MAIN --> unix console executable

Current Rule Limitation

- no minor promotion, only "Queening" of the pawn

- threefold repetition is not detected (same board situation appears three times)

Note: Could be implemented, but requires tracking of the complete game

- Fifty-move rule is not checked (no pawn move, no capture within last 50 moves)

Words

Ply a half move

General Links

http://chessprogramming.wikispaces.com/

Arduino specific

http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1260055596

Prefixes

chess\_ Generic Chess Application Interface

ce\_ Chess engine, used internally, these function should not be called directly

cu\_ Chess utility function

stack\_ Internal function for stack handling

Issues

10.01.2011

- castling to the right does not move the rook

--> done

- castling to the left: King can only move two squares

--> done

11.01.2011

Next Steps:

- replace stack\_NextCurrentPos with cu\_NextPos, cleanup code according to the loop variable

--> done

- Castling: Need to check for fields under attack

--> done

- Check for WIN / LOOSE situation, perhaps call ce\_Eval() once on the top-level board setup

just after the real move

- cleanup cu\_Move

--> almost done

- add some heuristics to the eval procedure

- add right side menu

--> done

- clean up chess\_ManualMove

--> done

- finish menu (consider is\_game\_end, undo move)

- end condition: if KING is under attack and if KING can not move to a field which is under attack...

then the game is lost. What will be returned by the Eval procedure? is it -INF?

--> finished

- reduce the use of variable color, all should be reduced to board\_orientation and ply&1

- chess\_GetNextMarked shoud make use of cu\_NextPos

--> done

- chess\_ManualMove: again cleanup, solve draw issue (KING is not in check and no legal moves are available)

--> done

22.01.2011

- simplify eval\_t ce\_Eval(void)

- position eval does not work, still moves side pawn :-(

maybe because all pieces are considered

--> done

\*/

#include "u8g.h"

//#ifndef \_\_unix\_\_

//#else

//#include <assert.h>

//#define U8G\_NOINLINE

//#endif

/\*

SAN identifies each piece by a single upper case letter. The standard English

values: pawn = "P", knight = "N", bishop = "B", rook = "R", queen = "Q", and

king = "K".

\*/

/\* numbers for the various pieces \*/

#define PIECE\_NONE 0

#define PIECE\_PAWN 1

#define PIECE\_KNIGHT 2

#define PIECE\_BISHOP 3

#define PIECE\_ROOK 4

#define PIECE\_QUEEN 5

#define PIECE\_KING 6

/\* color definitions \*/

#define COLOR\_WHITE 0

#define COLOR\_BLACK 1

/\* a mask, which includes COLOR and PIECE number \*/

#define COLOR\_PIECE\_MASK 0x01f

#define CP\_MARK\_MASK 0x20

#define ILLEGAL\_POSITION 255

/\* This is the build in upper limit of the search stack \*/

/\* This value defines the amount of memory allocated for the search stack \*/

/\* The search depth of this chess engine can never exceed this value \*/

#define STACK\_MAX\_SIZE 5

/\* chess half move stack: twice the number of undo's, a user can do \*/

#define CHM\_USER\_SIZE 6

/\* the CHM\_LIST\_SIZE must be larger than the maximum search depth \*/

/\* the overall size of ste half move stack \*/

#define CHM\_LIST\_SIZE (STACK\_MAX\_SIZE+CHM\_USER\_SIZE+2)

typedef int16\_t eval\_t; /\* a variable type to store results from the evaluation \*/

//#define EVAL\_T\_LOST -32768

#define EVAL\_T\_MIN -32767

#define EVAL\_T\_MAX 32767

//#define EVAL\_T\_WIN 32767

/\* for maintainance of our own stack: this is the definition of one element on the stack \*/

struct \_stack\_element\_struct

{

/\* the current source position which is investigated \*/

uint8\_t current\_pos;

uint8\_t current\_cp;

uint8\_t current\_color; /\* COLOR\_WHITE or COLOR\_BLACK: must be predefines \*/

/\* the move which belongs to that value, both values are game positions \*/

uint8\_t best\_from\_pos;

uint8\_t best\_to\_pos;

/\* the best value, which has been dicovered so far \*/

eval\_t best\_eval;

};

typedef struct \_stack\_element\_struct stack\_element\_t;

typedef struct \_stack\_element\_struct \*stack\_element\_p;

/\* chess half move history \*/

struct \_chm\_struct

{

uint8\_t main\_cp; /\* the main piece, which is moved \*/

uint8\_t main\_src; /\* the source position of the main piece \*/

uint8\_t main\_dest; /\* the destination of the main piece \*/

uint8\_t other\_cp; /\* another piece: the captured one, the ROOK in case of castling or PIECE\_NONE \*/

uint8\_t other\_src; /\* the delete position of other\_cp. Often identical to main\_dest except for e.p. and castling \*/

uint8\_t other\_dest; /\* only used for castling: ROOK destination pos \*/

/\* the position of the last pawn, which did a double move forward \*/

/\* this is required to check en passant conditions \*/

/\* this array can be indexed by the color of the current player \*/

/\* this is the condition BEFORE the move was done \*/

uint8\_t pawn\_dbl\_move[2];

/\* flags for the movement of rook and king; required for castling \*/

/\* a 1 means: castling is (still) possible \*/

/\* a 0 means: castling not possible \*/

/\* bit 0 left side white \*/

/\* bit 1 right side white \*/

/\* bit 2 left side black \*/

/\* bit 3 right side black \*/

/\* this is the condition BEFORE the move was done \*/

uint8\_t castling\_possible;

};

typedef struct \_chm\_struct chm\_t;

typedef struct \_chm\_struct \*chm\_p;

/\* little rook chess, main structure \*/

struct \_lrc\_struct

{

/\* half-move (ply) counter: Counts the number of half-moves so far. Starts with 0 \*/

/\* the lowest bit is used to derive the color of the current player \*/

/\* will be set to zero in chess\_SetupBoard() \*/

uint8\_t ply\_count;

/\* the half move stack position counter, counts the number of elements in chm\_list \*/

uint8\_t chm\_pos;

/\* each element contains a colored piece, empty fields have value 0 \*/

/\* the field with index 0 is black (lower left) \*/

uint8\_t board[64];

/\* the position of the last pawn, which did a double move forward \*/

/\* this is required to check en passant conditions \*/

/\* this array can be indexed by the color of the current player \*/

uint8\_t pawn\_dbl\_move[2];

/\* flags for the movement of rook and king; required for castling \*/

/\* a 1 means: castling is (still) possible \*/

/\* a 0 means: castling not possible \*/

/\* bit 0 left side white \*/

/\* bit 1 right side white \*/

/\* bit 2 left side black \*/

/\* bit 3 right side black \*/

uint8\_t castling\_possible;

/\* board orientation \*/

/\* 0: white is below COLOR\_WHITE \*/

/\* 1: black is below COLOR\_BLACK \*/

/\* bascially, this can be used as a color \*/

uint8\_t orientation;

/\* exchange colors of the pieces \*/

/\* 0: white has an empty body, use this for bright background color \*/

/\* 1: black has an empty body, use this for dark backround color \*/

uint8\_t strike\_out\_color;

/\* 0, when the game is ongoing \*/

/\* 1, when the game is stopped (lost or draw) \*/

uint8\_t is\_game\_end;

/\* the color of the side which lost the game \*/

/\* this value is only valid, when is\_game\_end is not 0 \*/

/\* values 0 and 1 represent WHITE and BLACK, 2 means a draw \*/

uint8\_t lost\_side\_color;

/\* checks are executed in ce\_LoopRecur \*/

/\* these checks will put some marks on the board \*/

/\* this will be used by the interface to find out \*/

/\* legal moves \*/

uint8\_t check\_src\_pos;

uint8\_t check\_mode; /\* CHECK\_MODE\_NONE, CHECK\_MODE\_MOVEABLE, CHECK\_MODE\_TARGET\_MOVE \*/

/\* count of the attacking pieces, indexed by color \*/

uint8\_t find\_piece\_cnt[2];

/\* sum of the attacking pieces, indexed by color \*/

uint8\_t find\_piece\_weight[2];

/\* points to the current element of the search stack \*/

/\* this stack is NEVER empty. The value 0 points to the first element of the stack \*/

/\* actually "curr\_depth" represent half-moves (plies) \*/

uint8\_t curr\_depth;

uint8\_t max\_depth;

stack\_element\_p curr\_element;

/\* allocated memory for the search stack \*/

stack\_element\_t stack\_memory[STACK\_MAX\_SIZE];

/\* the half move stack, used for move undo and depth search, size is stored in chm\_pos \*/

chm\_t chm\_list[CHM\_LIST\_SIZE];

};

typedef struct \_lrc\_struct lrc\_t;

#define CHECK\_MODE\_NONE 0

#define CHECK\_MODE\_MOVEABLE 1

#define CHECK\_MODE\_TARGET\_MOVE 2

/\*==============================================================\*/

/\* global variables \*/

/\*==============================================================\*/

u8g\_t \*lrc\_u8g;

lrc\_t lrc\_obj;

/\*==============================================================\*/

/\* forward declarations \*/

/\*==============================================================\*/

/\*

apply no inline to some of the functions:

avr-gcc very often inlines functions, however not inline saves a lot of program memory!

On the other hand there are some really short procedures which should be inlined (like cp\_GetColor)

These procedures are marked static to prevent the generation of the expanded procedure, which

also saves space.

\*/

uint8\_t stack\_Push(uint8\_t color) U8G\_NOINLINE;

void stack\_Pop(void) U8G\_NOINLINE;

void stack\_InitCurrElement(void) U8G\_NOINLINE;

void stack\_Init(uint8\_t max) U8G\_NOINLINE;

void stack\_SetMove(eval\_t val, uint8\_t to\_pos) U8G\_NOINLINE;

uint8\_t cu\_NextPos(uint8\_t pos) U8G\_NOINLINE;

static uint8\_t cu\_gpos2bpos(uint8\_t gpos);

static uint8\_t cp\_Construct(uint8\_t color, uint8\_t piece);

static uint8\_t cp\_GetPiece(uint8\_t cp);

static uint8\_t cp\_GetColor(uint8\_t cp);

uint8\_t cp\_GetFromBoard(uint8\_t pos) U8G\_NOINLINE;

void cp\_SetOnBoard(uint8\_t pos, uint8\_t cp) U8G\_NOINLINE;

void cu\_ClearBoard(void) U8G\_NOINLINE;

void chess\_SetupBoard(void) U8G\_NOINLINE;

eval\_t ce\_Eval(void);

void cu\_ClearMoveHistory(void) U8G\_NOINLINE;

void cu\_ReduceHistoryByFullMove(void) U8G\_NOINLINE;

void cu\_UndoHalfMove(void) U8G\_NOINLINE;

chm\_p cu\_PushHalfMove(void) U8G\_NOINLINE;

void ce\_CalculatePositionWeight(uint8\_t pos);

uint8\_t ce\_GetPositionAttackWeight(uint8\_t pos, uint8\_t color);

void chess\_Thinking(void);

void ce\_LoopPieces(void);

/\*==============================================================\*/

/\* search stack \*/

/\*==============================================================\*/

/\* get current element from stack \*/

stack\_element\_p stack\_GetCurrElement(void)

{

return lrc\_obj.curr\_element;

}

uint8\_t stack\_Push(uint8\_t color)

{

if ( lrc\_obj.curr\_depth == lrc\_obj.max\_depth )

return 0;

lrc\_obj.curr\_depth++;

lrc\_obj.curr\_element = lrc\_obj.stack\_memory+lrc\_obj.curr\_depth;

/\* change view for the evaluation \*/

color ^= 1;

stack\_GetCurrElement()->current\_color = color;

return 1;

}

void stack\_Pop(void)

{

lrc\_obj.curr\_depth--;

lrc\_obj.curr\_element = lrc\_obj.stack\_memory+lrc\_obj.curr\_depth;

}

/\* reset the current element on the stack \*/

void stack\_InitCurrElement(void)

{

stack\_element\_p e = stack\_GetCurrElement();

e->best\_eval = EVAL\_T\_MIN;

e->best\_from\_pos = ILLEGAL\_POSITION;

e->best\_to\_pos = ILLEGAL\_POSITION;

}

/\* resets the search stack (and the check mode) \*/

void stack\_Init(uint8\_t max)

{

lrc\_obj.curr\_depth = 0;

lrc\_obj.curr\_element = lrc\_obj.stack\_memory;

lrc\_obj.max\_depth = max;

lrc\_obj.check\_mode = CHECK\_MODE\_NONE;

stack\_InitCurrElement();

stack\_GetCurrElement()->current\_color = lrc\_obj.ply\_count;

stack\_GetCurrElement()->current\_color &= 1;

}

/\* assign evaluation value and store the move, if this is the best move \*/

/\* assumes, that current\_pos contains the source position \*/

void stack\_SetMove(eval\_t val, uint8\_t to\_pos)

{

stack\_element\_p e = stack\_GetCurrElement();

if ( e->best\_eval < val )

{

e->best\_eval = val;

e->best\_from\_pos = e->current\_pos;

e->best\_to\_pos = to\_pos;

}

}

/\*

calculate next position on a 0x88 board

loop is constructed in this way:

i = 0;

do

{

...

i = cu\_NextPos(i);

} while( i != 0 );

next pos might be started with an illegal position like 255

\*/

uint8\_t cu\_NextPos(uint8\_t pos)

{

/\* calculate next gpos \*/

pos++;

if ( ( pos & 0x08 ) != 0 )

{

pos+= 0x10;

pos&= 0xf0;

}

if ( ( pos & 0x80 ) != 0 )

pos = 0;

return pos;

}

uint8\_t cu\_PrevPos(uint8\_t pos)

{

/\* calculate prev gpos \*/

pos--;

if ( ( pos & 0x80 ) != 0 )

pos = 0x077;

else if ( ( pos & 0x08 ) != 0 )

{

pos &= 0xf0;

pos |= 0x07;

}

return pos;

}

/\*==============================================================\*/

/\* position transltion \*/

/\*==============================================================\*/

/\*

there are two positions

1. game position (gpos): BCD encoded x-y values

2. board position (bpos): a number between 0 and 63, only used to access the board.

\*/

/\*

gpos: game position value

returns: board position

note: does not do any checks

\*/

static uint8\_t cu\_gpos2bpos(uint8\_t gpos)

{

uint8\_t bpos = gpos;

bpos &= 0xf0;

bpos >>= 1;

gpos &= 0x0f;

bpos |= gpos;

return bpos;

}

#define gpos\_IsIllegal(gpos) ((gpos) & 0x088)

/\*==============================================================\*/

/\* colored piece handling \*/

/\*==============================================================\*/

#define cp\_IsMarked(cp) ((cp) & CP\_MARK\_MASK)

/\*

piece: one of PIECE\_xxx

color: COLOR\_WHITE or COLOR\_BLACK

returns: A colored piece

\*/

static uint8\_t cp\_Construct(uint8\_t color, uint8\_t piece)

{

color <<= 4;

color |= piece;

return color;

}

/\* inline is better than a macro \*/

static uint8\_t cp\_GetPiece(uint8\_t cp)

{

cp &= 0x0f;

return cp;

}

/\*

we could use a macro:

#define cp\_GetColor(cp) (((cp) >> 4)&1)

however, inlined functions are sometimes much better

\*/

static uint8\_t cp\_GetColor(uint8\_t cp)

{

cp >>= 4;

cp &= 1;

return cp;

}

/\*

pos: game position

returns the colored piece at the given position

\*/

uint8\_t cp\_GetFromBoard(uint8\_t pos)

{

return lrc\_obj.board[cu\_gpos2bpos(pos)];

}

/\*

pos: game position

cp: colored piece

\*/

void cp\_SetOnBoard(uint8\_t pos, uint8\_t cp)

{

/\*printf("cp\_SetOnBoard gpos:%02x cp:%02x\n", pos, cp);\*/

lrc\_obj.board[cu\_gpos2bpos(pos)] = cp;

}

/\*==============================================================\*/

/\* global board access \*/

/\*==============================================================\*/

void cu\_ClearBoard(void)

{

uint8\_t i;

/\* clear the board \*/

for( i = 0; i < 64; i++ )

lrc\_obj.board[i] = PIECE\_NONE;

lrc\_obj.ply\_count = 0;

lrc\_obj.orientation = COLOR\_WHITE;

lrc\_obj.pawn\_dbl\_move[0] = ILLEGAL\_POSITION;

lrc\_obj.pawn\_dbl\_move[1] = ILLEGAL\_POSITION;

lrc\_obj.castling\_possible = 0x0f;

lrc\_obj.is\_game\_end = 0;

lrc\_obj.lost\_side\_color = 0;

/\* clear half move history \*/

cu\_ClearMoveHistory();

}

/\*

test setup

white wins in one move

\*/

void chess\_SetupBoardTest01(void)

{

cu\_ClearBoard();

lrc\_obj.board[7+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_KING);

lrc\_obj.board[7+5\*8] = cp\_Construct(COLOR\_WHITE, PIECE\_PAWN);

lrc\_obj.board[3] = cp\_Construct(COLOR\_WHITE, PIECE\_KING);

lrc\_obj.board[0+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_ROOK);

lrc\_obj.board[6] = cp\_Construct(COLOR\_WHITE, PIECE\_QUEEN);

}

/\* setup the global board \*/

void chess\_SetupBoard(void)

{

uint8\_t i;

register uint8\_t bp, wp;

/\* clear the board \*/

cu\_ClearBoard();

/\* precronstruct pawns \*/

wp = cp\_Construct(COLOR\_WHITE, PIECE\_PAWN);

bp = cp\_Construct(COLOR\_BLACK, PIECE\_PAWN);

/\* setup pawn \*/

for( i = 0; i < 8; i++ )

{

lrc\_obj.board[i+8] = wp;

lrc\_obj.board[i+6\*8] = bp;

}

/\* assign remaining pieces \*/

lrc\_obj.board[0] = cp\_Construct(COLOR\_WHITE, PIECE\_ROOK);

lrc\_obj.board[1] = cp\_Construct(COLOR\_WHITE, PIECE\_KNIGHT);

lrc\_obj.board[2] = cp\_Construct(COLOR\_WHITE, PIECE\_BISHOP);

lrc\_obj.board[3] = cp\_Construct(COLOR\_WHITE, PIECE\_QUEEN);

lrc\_obj.board[4] = cp\_Construct(COLOR\_WHITE, PIECE\_KING);

lrc\_obj.board[5] = cp\_Construct(COLOR\_WHITE, PIECE\_BISHOP);

lrc\_obj.board[6] = cp\_Construct(COLOR\_WHITE, PIECE\_KNIGHT);

lrc\_obj.board[7] = cp\_Construct(COLOR\_WHITE, PIECE\_ROOK);

lrc\_obj.board[0+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_ROOK);

lrc\_obj.board[1+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_KNIGHT);

lrc\_obj.board[2+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_BISHOP);

lrc\_obj.board[3+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_QUEEN);

lrc\_obj.board[4+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_KING);

lrc\_obj.board[5+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_BISHOP);

lrc\_obj.board[6+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_KNIGHT);

lrc\_obj.board[7+7\*8] = cp\_Construct(COLOR\_BLACK, PIECE\_ROOK);

//chess\_SetupBoardTest01();

}

/\*==============================================================\*/

/\* checks \*/

/\*==============================================================\*/

/\*

checks if the position is somehow illegal

\*/

uint8\_t cu\_IsIllegalPosition(uint8\_t pos, uint8\_t my\_color)

{

uint8\_t board\_cp;

/\* check, if the position is offboard \*/

if ( gpos\_IsIllegal(pos) != 0 )

return 1;

/\* get the piece from the board \*/

board\_cp = cp\_GetFromBoard(pos);

/\* check if hit our own pieces \*/

if ( board\_cp != 0 )

if ( cp\_GetColor(board\_cp) == my\_color )

return 1;

/\* all ok, we could go to this position \*/

return 0;

}

/\*==============================================================\*/

/\* evaluation procedure \*/

/\*==============================================================\*/

/\*

basic idea is to return a value between EVAL\_T\_MIN and EVAL\_T\_MAX

\*/

/\*

the weight table uses the PIECE number as index:

#define PIECE\_NONE 0

#define PIECE\_PAWN 1

#define PIECE\_KNIGHT 2

#define PIECE\_BISHOP 3

#define PIECE\_ROOK 4

#define PIECE\_QUEEN 5

#define PIECE\_KING 6

the king itself is not counted

\*/

uint8\_t ce\_piece\_weight[] = { 0, 1, 3, 3, 5, 9, 0 };

uint8\_t ce\_pos\_weight[] = { 0, 1, 1, 2, 2, 1, 1, 0};

/\*

evaluate the current situation on the global board

\*/

eval\_t ce\_Eval(void)

{

uint8\_t cp;

uint8\_t is\_my\_king\_present = 0;

uint8\_t is\_opposit\_king\_present = 0;

eval\_t material\_my\_color = 0;

eval\_t material\_opposit\_color = 0;

eval\_t position\_my\_color = 0;

eval\_t position\_opposit\_color = 0;

eval\_t result;

uint8\_t pos;

pos = 0;

do

{

/\* get colored piece from the board \*/

cp = cp\_GetFromBoard(pos);

if ( cp\_GetPiece(cp) != PIECE\_NONE )

{

if ( stack\_GetCurrElement()->current\_color == cp\_GetColor(cp) )

{

/\* this is our color \*/

/\* check if we found our king \*/

if ( cp\_GetPiece(cp) == PIECE\_KING )

is\_my\_king\_present = 1;

material\_my\_color += ce\_piece\_weight[cp\_GetPiece(cp)];

if ( cp\_GetPiece(cp) == PIECE\_PAWN || cp\_GetPiece(cp) == PIECE\_KNIGHT )

{

position\_my\_color += ce\_pos\_weight[pos&7]\*ce\_pos\_weight[(pos>>4)&7];

}

}

else

{

/\* this is the opposit color \*/

if ( cp\_GetPiece(cp) == PIECE\_KING )

is\_opposit\_king\_present = 1;

material\_opposit\_color += ce\_piece\_weight[cp\_GetPiece(cp)];

if ( cp\_GetPiece(cp) == PIECE\_PAWN || cp\_GetPiece(cp) == PIECE\_KNIGHT )

{

position\_opposit\_color += ce\_pos\_weight[pos&7]\*ce\_pos\_weight[(pos>>4)&7];

}

}

}

pos = cu\_NextPos(pos);

} while( pos != 0 );

/\* decide if we lost or won the game \*/

if ( is\_my\_king\_present == 0 )

return EVAL\_T\_MIN; /\*\_LOST\*/

if ( is\_opposit\_king\_present == 0 )

return EVAL\_T\_MAX; /\*\_WIN\*/

/\* here is the evaluation function \*/

result = material\_my\_color - material\_opposit\_color;

result <<= 3;

result += position\_my\_color - position\_opposit\_color;

return result;

}

/\*==============================================================\*/

/\* move backup and restore \*/

/\*==============================================================\*/

/\* this procedure must be called to keep the size as low as possible \*/

/\* if the chm\_list is large enough, it could hold the complete history \*/

/\* but for an embedded controler... it is deleted for every engine search \*/

void cu\_ClearMoveHistory(void)

{

lrc\_obj.chm\_pos = 0;

}

void cu\_ReduceHistoryByFullMove(void)

{

uint8\_t i;

while( lrc\_obj.chm\_pos > CHM\_USER\_SIZE )

{

i = 0;

for(;;)

{

if ( i+2 >= lrc\_obj.chm\_pos )

break;

lrc\_obj.chm\_list[i] = lrc\_obj.chm\_list[i+2];

i++;

}

lrc\_obj.chm\_pos -= 2;

}

}

void cu\_UndoHalfMove(void)

{

chm\_p chm;

if ( lrc\_obj.chm\_pos == 0 )

return;

lrc\_obj.chm\_pos--;

chm = lrc\_obj.chm\_list+lrc\_obj.chm\_pos;

lrc\_obj.pawn\_dbl\_move[0] = chm->pawn\_dbl\_move[0];

lrc\_obj.pawn\_dbl\_move[1] = chm->pawn\_dbl\_move[1];

lrc\_obj.castling\_possible = chm->castling\_possible;

cp\_SetOnBoard(chm->main\_src, chm->main\_cp);

cp\_SetOnBoard(chm->main\_dest, PIECE\_NONE);

if ( chm->other\_src != ILLEGAL\_POSITION )

cp\_SetOnBoard(chm->other\_src, chm->other\_cp);

if ( chm->other\_dest != ILLEGAL\_POSITION )

cp\_SetOnBoard(chm->other\_dest, PIECE\_NONE);

}

/\*

assumes, that the following members of the returned chm structure are filled

uint8\_t main\_cp; the main piece, which is moved

uint8\_t main\_src; the source position of the main piece

uint8\_t main\_dest; the destination of the main piece

uint8\_t other\_cp; another piece: the captured one, the ROOK in case of castling or PIECE\_NONE

uint8\_t other\_src; the delete position of other\_cp. Often identical to main\_dest except for e.p. and castling

uint8\_t other\_dest; only used for castling: ROOK destination pos

\*/

chm\_p cu\_PushHalfMove(void)

{

chm\_p chm;

chm = lrc\_obj.chm\_list+lrc\_obj.chm\_pos;

if ( lrc\_obj.chm\_pos < CHM\_LIST\_SIZE-1)

lrc\_obj.chm\_pos++;

chm->pawn\_dbl\_move[0] = lrc\_obj.pawn\_dbl\_move[0];

chm->pawn\_dbl\_move[1] = lrc\_obj.pawn\_dbl\_move[1];

chm->castling\_possible = lrc\_obj.castling\_possible;

return chm;

}

char chess\_piece\_to\_char[] = "NBRQK";

/\*

simple moves on empty field: Ka1-b2

capture moves: Ka1xb2

castling: 0-0 or 0-0-0

\*/

static void cu\_add\_pos(char \*s, uint8\_t pos) U8G\_NOINLINE;

static void cu\_add\_pos(char \*s, uint8\_t pos)

{

\*s = pos;

\*s >>= 4;

\*s += 'a';

s++;

\*s = pos;

\*s &= 15;

\*s += '1';

}

const char \*cu\_GetHalfMoveStr(uint8\_t idx)

{

chm\_p chm;

static char buf[7]; /\*Ka1-b2\*/

char \*p = buf;

chm = lrc\_obj.chm\_list+idx;

if ( cp\_GetPiece(chm->main\_cp) != PIECE\_NONE )

{

if ( cp\_GetPiece(chm->main\_cp) > PIECE\_PAWN )

{

\*p++ = chess\_piece\_to\_char[cp\_GetPiece(chm->main\_cp)-2];

}

cu\_add\_pos(p, chm->main\_src);

p+=2;

if ( cp\_GetPiece(chm->other\_cp) == PIECE\_NONE )

\*p++ = '-';

else

\*p++ = 'x';

cu\_add\_pos(p, chm->main\_dest);

p+=2;

}

\*p = '\0';

return buf;

}

/\*==============================================================\*/

/\* move \*/

/\*==============================================================\*/

/\*

Move a piece from source position to a destination on the board

This function

- does not perform any checking

- however it processes "en passant" and casteling

- backup the move and allow 1x undo

2011-02-05:

- fill pawn\_dbl\_move[] for double pawn moves

--> done

- Implement casteling

--> done

- en passant

--> done

- pawn conversion/promotion

--> done

- half-move backup

--> done

- cleanup everything, minimize variables

--> done

\*/

void cu\_Move(uint8\_t src, uint8\_t dest)

{

/\* start backup structure \*/

chm\_p chm = cu\_PushHalfMove();

/\* these are the values from the board at the positions, provided as arguments to this function \*/

uint8\_t cp\_src, cp\_dest;

/\* Maybe a second position is cleared and one additional location is set \*/

uint8\_t clr\_pos2;

uint8\_t set\_pos2;

uint8\_t set\_cp2;

/\* get values from board \*/

cp\_src = cp\_GetFromBoard(src);

cp\_dest = cp\_GetFromBoard(dest);

/\* fill backup structure \*/

chm->main\_cp = cp\_src;

chm->main\_src = src;

chm->main\_dest = dest;

chm->other\_cp = cp\_dest; /\* prepace capture backup \*/

chm->other\_src = dest;

chm->other\_dest = ILLEGAL\_POSITION;

/\* setup results as far as possible with some suitable values \*/

clr\_pos2 = ILLEGAL\_POSITION; /\* for en passant and castling, two positions might be cleared \*/

set\_pos2 = ILLEGAL\_POSITION; /\* only used for castling \*/

set\_cp2 = PIECE\_NONE; /\* ROOK for castling \*/

/\* check for PAWN \*/

if ( cp\_GetPiece(cp\_src) == PIECE\_PAWN )

{

/\* double step: is the distance 2 rows \*/

if ( (src - dest == 32) || ( dest - src == 32 ) )

{

/\* remember the destination position \*/

lrc\_obj.pawn\_dbl\_move[cp\_GetColor(cp\_src)] = dest;

}

/\* check if the PAWN is able to promote \*/

else if ( (dest>>4) == 0 || (dest>>4) == 7 )

{

/\* do simple "queening" \*/

cp\_src &= ~PIECE\_PAWN;

cp\_src |= PIECE\_QUEEN;

}

/\* is it en passant capture? \*/

/\* check for side move \*/

else if ( ((src + dest) & 1) != 0 )

{

/\* check, if target field is empty \*/

if ( cp\_GetPiece(cp\_dest) == PIECE\_NONE )

{

/\* this is en passant \*/

/\* no further checking required, because legal moves are assumed here \*/

/\* however... the captured pawn position must be valid \*/

clr\_pos2 = lrc\_obj.pawn\_dbl\_move[cp\_GetColor(cp\_src) ^ 1];

chm->other\_src = clr\_pos2;

chm->other\_cp = cp\_GetFromBoard(clr\_pos2);

}

}

}

/\* check for the KING \*/

else if ( cp\_GetPiece(cp\_src) == PIECE\_KING )

{

/\* disallow castling, if the KING has moved \*/

if ( cp\_GetColor(cp\_src) == COLOR\_WHITE )

{

/\* if white KING has moved, disallow castling for white \*/

lrc\_obj.castling\_possible &= 0x0c;

}

else

{

/\* if black KING has moved, disallow castling for black \*/

lrc\_obj.castling\_possible &= 0x03;

}

/\* has it been castling to the left? \*/

if ( src - dest == 2 )

{

/\* let the ROOK move to pos2 \*/

set\_pos2 = src-1;

set\_cp2 = cp\_GetFromBoard(src-4);

/\* the ROOK must be cleared from the original position \*/

clr\_pos2 = src-4;

chm->other\_cp = set\_cp2;

chm->other\_src = clr\_pos2;

chm->other\_dest = set\_pos2;

}

/\* has it been castling to the right? \*/

else if ( dest - src == 2 )

{

/\* let the ROOK move to pos2 \*/

set\_pos2 = src+1;

set\_cp2 = cp\_GetFromBoard(src+3);

/\* the ROOK must be cleared from the original position \*/

clr\_pos2 = src+3;

chm->other\_cp = set\_cp2;

chm->other\_src = clr\_pos2;

chm->other\_dest = set\_pos2;

}

}

/\* check for the ROOK \*/

else if ( cp\_GetPiece(cp\_src) == PIECE\_ROOK )

{

/\* disallow white left castling \*/

if ( src == 0x00 )

lrc\_obj.castling\_possible &= ~0x01;

/\* disallow white right castling \*/

if ( src == 0x07 )

lrc\_obj.castling\_possible &= ~0x02;

/\* disallow black left castling \*/

if ( src == 0x70 )

lrc\_obj.castling\_possible &= ~0x04;

/\* disallow black right castling \*/

if ( src == 0x77 )

lrc\_obj.castling\_possible &= ~0x08;

}

/\* apply new board situation \*/

cp\_SetOnBoard(dest, cp\_src);

if ( set\_pos2 != ILLEGAL\_POSITION )

cp\_SetOnBoard(set\_pos2, set\_cp2);

cp\_SetOnBoard(src, PIECE\_NONE);

if ( clr\_pos2 != ILLEGAL\_POSITION )

cp\_SetOnBoard(clr\_pos2, PIECE\_NONE);

}

/\*

this subprocedure decides for evaluation of the current board situation or further (deeper) investigation

Argument pos is the new target position if the current piece

\*/

uint8\_t ce\_LoopRecur(uint8\_t pos)

{

eval\_t eval;

/\* 1. check if target position is occupied by the same player (my\_color) \*/

/\* of if pos is somehow illegal or not valid \*/

if ( cu\_IsIllegalPosition(pos, stack\_GetCurrElement()->current\_color) != 0 )

return 0;

/\* 2. move piece to the specified position, capture opponent piece if required \*/

cu\_Move(stack\_GetCurrElement()->current\_pos, pos);

/\* 3. \*/

/\* if depth reached: evaluate \*/

/\* else: go down next level \*/

/\* no eval if there had been any valid half-moves, so the default value (MIN) will be returned. \*/

if ( stack\_Push(stack\_GetCurrElement()->current\_color) == 0 )

{

eval = ce\_Eval();

}

else

{

/\* init the element, which has been pushed \*/

stack\_InitCurrElement();

/\* start over with ntext level \*/

ce\_LoopPieces();

/\* get the best move from opponents view, so invert the result \*/

eval = -stack\_GetCurrElement()->best\_eval;

stack\_Pop();

}

/\* 4. store result \*/

stack\_SetMove(eval, pos);

/\* 5. undo the move \*/

cu\_UndoHalfMove();

/\* 6. check special modes \*/

/\* the purpose of these checks is to mark special pieces and positions on the board \*/

/\* these marks can be checked by the user interface to highlight special positions \*/

if ( lrc\_obj.check\_mode != 0 )

{

stack\_element\_p e = stack\_GetCurrElement();

if ( lrc\_obj.check\_mode == CHECK\_MODE\_MOVEABLE )

{

cp\_SetOnBoard(e->current\_pos, e->current\_cp | CP\_MARK\_MASK );

}

else if ( lrc\_obj.check\_mode == CHECK\_MODE\_TARGET\_MOVE )

{

if ( e->current\_pos == lrc\_obj.check\_src\_pos )

{

cp\_SetOnBoard(pos, cp\_GetFromBoard(pos) | CP\_MARK\_MASK );

}

}

}

return 1;

}

/\*==============================================================\*/

/\* move pieces which can move one or more steps into a direction \*/

/\*==============================================================\*/

/\*

subprocedure to generate various target positions for some pieces

special cases are handled in the piece specific sub-procedure

Arguments:

d: a list of potential directions

is\_multi\_step: if the piece can only do one step (zero for KING and KNIGHT)

\*/

static const uint8\_t ce\_dir\_offset\_rook[] PROGMEM = { 1, 16, -16, -1, 0 };

static const uint8\_t ce\_dir\_offset\_bishop[] PROGMEM = { 15, 17, -17, -15, 0 };

static const uint8\_t ce\_dir\_offset\_queen[] PROGMEM = { 1, 16, -16, -1, 15, 17, -17, -15, 0 };

static const uint8\_t ce\_dir\_offset\_knight[] PROGMEM = {14, -14, 18, -18, 31, -31, 33, -33, 0};

void ce\_LoopDirsSingleMultiStep(const uint8\_t \*d, uint8\_t is\_multi\_step)

{

uint8\_t loop\_pos;

/\* with all directions \*/

for(;;)

{

if ( u8g\_pgm\_read(d) == 0 )

break;

/\* start again from the initial position \*/

loop\_pos = stack\_GetCurrElement()->current\_pos;

/\* check direction \*/

do

{

/\* check next position into one direction \*/

loop\_pos += u8g\_pgm\_read(d);

/\*

go further to ce\_LoopRecur()

0 will be returned if the target position is illegal or a piece of the own color

this is used to stop walking into one direction

\*/

if ( ce\_LoopRecur(loop\_pos) == 0 )

break;

/\* stop if we had hit another piece \*/

if ( cp\_GetPiece(cp\_GetFromBoard(loop\_pos)) != PIECE\_NONE )

break;

} while( is\_multi\_step );

d++;

}

}

void ce\_LoopRook(void)

{

ce\_LoopDirsSingleMultiStep(ce\_dir\_offset\_rook, 1);

}

void ce\_LoopBishop(void)

{

ce\_LoopDirsSingleMultiStep(ce\_dir\_offset\_bishop, 1);

}

void ce\_LoopQueen(void)

{

ce\_LoopDirsSingleMultiStep(ce\_dir\_offset\_queen, 1);

}

void ce\_LoopKnight(void)

{

ce\_LoopDirsSingleMultiStep(ce\_dir\_offset\_knight, 0);

}

/\*==============================================================\*/

/\* move king \*/

/\*==============================================================\*/

uint8\_t cu\_IsKingCastling(uint8\_t mask, int8\_t direction, uint8\_t cnt) U8G\_NOINLINE;

/\*

checks, if the king can do castling

Arguments:

mask: the bit-mask for the global "castling possible" flag

direction: left castling: -1, right castling 1

cnt: number of fields to be checked: 3 or 2

\*/

uint8\_t cu\_IsKingCastling(uint8\_t mask, int8\_t direction, uint8\_t cnt)

{

uint8\_t pos;

uint8\_t opponent\_color;

/\* check if the current board state allows castling \*/

if ( (lrc\_obj.castling\_possible & mask) == 0 )

return 0; /\* castling not allowed \*/

/\* get the position of the KING, could be white or black king \*/

pos = stack\_GetCurrElement()->current\_pos;

/\* calculate the color of the opponent \*/

opponent\_color = 1;

opponent\_color -= stack\_GetCurrElement()->current\_color;

/\* if the KING itself is given check... \*/

if ( ce\_GetPositionAttackWeight(pos, opponent\_color) > 0 )

return 0;

/\* check if fields in the desired direction are emtpy \*/

for(;;)

{

/\* go to the next field \*/

pos += direction;

/\* check for a piece \*/

if ( cp\_GetPiece(cp\_GetFromBoard(pos)) != PIECE\_NONE )

return 0; /\* castling not allowed \*/

/\* if some of the fields are under attack \*/

if ( ce\_GetPositionAttackWeight(pos, opponent\_color) > 0 )

return 0;

cnt--;

if ( cnt == 0 )

break;

}

return 1; /\* castling allowed \*/

}

void ce\_LoopKing(void)

{

/\*

there is an interessting timing problem in this procedure

it must be checked for castling first and as second step the normal

KING movement. If we would first check for normal moves, than

any marks might be overwritten by the ROOK in the case of castling.

\*/

/\* castling (this must be done before checking normal moves (see above) \*/

if ( stack\_GetCurrElement()->current\_color == COLOR\_WHITE )

{

/\* white left castling \*/

if ( cu\_IsKingCastling(1, -1, 3) != 0 )

{

/\* check for attacked fields \*/

ce\_LoopRecur(stack\_GetCurrElement()->current\_pos-2);

}

/\* white right castling \*/

if ( cu\_IsKingCastling(2, 1, 2) != 0 )

{

/\* check for attacked fields \*/

ce\_LoopRecur(stack\_GetCurrElement()->current\_pos+2);

}

}

else

{

/\* black left castling \*/

if ( cu\_IsKingCastling(4, -1, 3) != 0 )

{

/\* check for attacked fields \*/

ce\_LoopRecur(stack\_GetCurrElement()->current\_pos-2);

}

/\* black right castling \*/

if ( cu\_IsKingCastling(8, 1, 2) != 0 )

{

/\* check for attacked fields \*/

ce\_LoopRecur(stack\_GetCurrElement()->current\_pos+2);

}

}

/\* reuse queen directions \*/

ce\_LoopDirsSingleMultiStep(ce\_dir\_offset\_queen, 0);

}

/\*==============================================================\*/

/\* move pawn \*/

/\*==============================================================\*/

/\*

doppelschritt: nur von der grundlinie aus, beide (!) felder vor dem bauern müssen frei sein

en passant: nur unmittelbar nachdem ein doppelschritt ausgeführt wurde.

\*/

void ce\_LoopPawnSideCapture(uint8\_t loop\_pos)

{

if ( gpos\_IsIllegal(loop\_pos) == 0 )

{

/\* get the piece from the board \*/

/\* if the field is NOT empty \*/

if ( cp\_GetPiece(cp\_GetFromBoard(loop\_pos)) != PIECE\_NONE )

{

/\* normal capture \*/

ce\_LoopRecur(loop\_pos);

/\* TODO: check for pawn conversion/promotion \*/

}

else

{

/\* check conditions for en passant capture \*/

if ( stack\_GetCurrElement()->current\_color == COLOR\_WHITE )

{

if ( lrc\_obj.pawn\_dbl\_move[COLOR\_BLACK]+16 == loop\_pos )

{

ce\_LoopRecur(loop\_pos);

/\* note: pawn conversion/promotion can not occur \*/

}

}

else

{

if ( lrc\_obj.pawn\_dbl\_move[COLOR\_WHITE] == loop\_pos+16 )

{

ce\_LoopRecur(loop\_pos);

/\* note: pawn conversion/promotion can not occur \*/

}

}

}

}

}

void ce\_LoopPawn(void)

{

uint8\_t initial\_pos = stack\_GetCurrElement()->current\_pos;

uint8\_t my\_color = stack\_GetCurrElement()->current\_color;

uint8\_t loop\_pos;

uint8\_t line;

/\* one step forward \*/

loop\_pos = initial\_pos;

line = initial\_pos;

line >>= 4;

if ( my\_color == COLOR\_WHITE )

loop\_pos += 16;

else

loop\_pos -= 16;

if ( gpos\_IsIllegal(loop\_pos) == 0 )

{

/\* if the field is empty \*/

if ( cp\_GetPiece(cp\_GetFromBoard(loop\_pos)) == PIECE\_NONE )

{

/\* TODO: check for and loop through piece conversion/promotion \*/

ce\_LoopRecur(loop\_pos);

/\* second step forward \*/

/\* if pawn is on his starting line \*/

if ( (my\_color == COLOR\_WHITE && line == 1) || (my\_color == COLOR\_BLACK && line == 6 ) )

{

/\* the place before the pawn is not occupied, so we can do double moves, see above \*/

if ( my\_color == COLOR\_WHITE )

loop\_pos += 16;

else

loop\_pos -= 16;

if ( cp\_GetPiece(cp\_GetFromBoard(loop\_pos)) == PIECE\_NONE )

{

/\* this is a special case, other promotions of the pawn can not occur \*/

ce\_LoopRecur(loop\_pos);

}

}

}

}

/\* capture \*/

loop\_pos = initial\_pos;

if ( my\_color == COLOR\_WHITE )

loop\_pos += 15;

else

loop\_pos -= 15;

ce\_LoopPawnSideCapture(loop\_pos);

loop\_pos = initial\_pos;

if ( my\_color == COLOR\_WHITE )

loop\_pos += 17;

else

loop\_pos -= 17;

ce\_LoopPawnSideCapture(loop\_pos);

}

/\*==============================================================\*/

/\* attacked \*/

/\*==============================================================\*/

/\*

from a starting position, search for a piece, that might jump to that postion.

return:

the two global variables

lrc\_obj.find\_piece\_weight[0];

lrc\_obj.find\_piece\_weight[1];

will be increased by the weight of the attacked pieces of that color.

it is usually required to reset these global variables to zero, before using

this function.

\*/

void ce\_FindPieceByStep(uint8\_t start\_pos, uint8\_t piece, const uint8\_t \*d, uint8\_t is\_multi\_step)

{

uint8\_t loop\_pos, cp;

/\* with all directions \*/

for(;;)

{

if ( u8g\_pgm\_read(d) == 0 )

break;

/\* start again from the initial position \*/

loop\_pos = start\_pos;

/\* check direction \*/

do

{

/\* check next position into one direction \*/

loop\_pos += u8g\_pgm\_read(d);

/\* check if the board boundary has been crossed \*/

if ( (loop\_pos & 0x088) != 0 )

break;

/\* get the colored piece from the board \*/

cp = cp\_GetFromBoard(loop\_pos);

/\* stop if we had hit another piece \*/

if ( cp\_GetPiece(cp) != PIECE\_NONE )

{

/\* if it is the piece we are looking for, then add the weight \*/

if ( cp\_GetPiece(cp) == piece )

{

lrc\_obj.find\_piece\_weight[cp\_GetColor(cp)] += ce\_piece\_weight[piece];

lrc\_obj.find\_piece\_cnt[cp\_GetColor(cp)]++;

}

/\* in any case, break out of the inner loop \*/

break;

}

} while( is\_multi\_step );

d++;

}

}

void ce\_FindPawnPiece(uint8\_t dest\_pos, uint8\_t color)

{

uint8\_t cp;

/\* check if the board boundary has been crossed \*/

if ( (dest\_pos & 0x088) == 0 )

{

/\* get the colored piece from the board \*/

cp = cp\_GetFromBoard(dest\_pos);

/\* only if there is a pawn of the matching color \*/

if ( cp\_GetPiece(cp) == PIECE\_PAWN )

{

if ( cp\_GetColor(cp) == color )

{

/\* the weight of the PAWN \*/

lrc\_obj.find\_piece\_weight[color] += 1;

lrc\_obj.find\_piece\_cnt[color]++;

}

}

}

}

/\*

find out, which pieces do attack a specified field

used to

- check if the KING can do castling

- check if the KING must move

may be used in the eval procedure ... once...

the result is stored in the global array

uint8\_t lrc\_obj.find\_piece\_weight[2];

which is indexed with the color.

lrc\_obj.find\_piece\_weight[COLOR\_WHITE] is the sum of all white pieces

which can directly move to this field.

example:

if the black KING is at "pos" and lrc\_obj.find\_piece\_weight[COLOR\_WHITE] is not zero

(after executing ce\_CalculatePositionWeight(pos)) then the KING must be protected or moveed, because

the KING was given check.

\*/

void ce\_CalculatePositionWeight(uint8\_t pos)

{

lrc\_obj.find\_piece\_weight[0] = 0;

lrc\_obj.find\_piece\_weight[1] = 0;

lrc\_obj.find\_piece\_cnt[0] = 0;

lrc\_obj.find\_piece\_cnt[1] = 0;

if ( (pos & 0x088) != 0 )

return;

ce\_FindPieceByStep(pos, PIECE\_ROOK, ce\_dir\_offset\_rook, 1);

ce\_FindPieceByStep(pos, PIECE\_BISHOP, ce\_dir\_offset\_bishop, 1);

ce\_FindPieceByStep(pos, PIECE\_QUEEN, ce\_dir\_offset\_queen, 1);

ce\_FindPieceByStep(pos, PIECE\_KNIGHT, ce\_dir\_offset\_knight, 0);

ce\_FindPieceByStep(pos, PIECE\_KING, ce\_dir\_offset\_queen, 0);

ce\_FindPawnPiece(pos+17, COLOR\_BLACK);

ce\_FindPawnPiece(pos+15, COLOR\_BLACK);

ce\_FindPawnPiece(pos-17, COLOR\_WHITE);

ce\_FindPawnPiece(pos-15, COLOR\_WHITE);

}

/\*

calculate the summed weight of pieces with specified color which can move to a specified position

argument:

pos: the position which should be analysed

color: the color of those pieces which should be analysed

e.g. if a black piece is at 'pos' and 'color' is white then this procedure returns the white atting count

\*/

uint8\_t ce\_GetPositionAttackWeight(uint8\_t pos, uint8\_t color)

{

ce\_CalculatePositionWeight(pos);

return lrc\_obj.find\_piece\_weight[color];

}

uint8\_t ce\_GetPositionAttackCount(uint8\_t pos, uint8\_t color)

{

ce\_CalculatePositionWeight(pos);

return lrc\_obj.find\_piece\_cnt[color];

}

/\*==============================================================\*/

/\* depth search starts here: loop over all pieces of the current color on the board \*/

/\*==============================================================\*/

void ce\_LoopPieces(void)

{

stack\_element\_p e = stack\_GetCurrElement();

/\* start with lower left position (A1) \*/

e->current\_pos = 0;

do

{

e->current\_cp = cp\_GetFromBoard(e->current\_pos);

/\* check if the position on the board is empty \*/

if ( e->current\_cp != 0 )

{

/\* only generate moves for the current color \*/

if ( e->current\_color == cp\_GetColor(e->current\_cp) )

{

chess\_Thinking();

/\* find out which piece is used \*/

switch(cp\_GetPiece(e->current\_cp))

{

case PIECE\_NONE:

break;

case PIECE\_PAWN:

ce\_LoopPawn();

break;

case PIECE\_KNIGHT:

ce\_LoopKnight();

break;

case PIECE\_BISHOP:

ce\_LoopBishop();

break;

case PIECE\_ROOK:

ce\_LoopRook();

break;

case PIECE\_QUEEN:

ce\_LoopQueen();

break;

case PIECE\_KING:

ce\_LoopKing();

break;

}

}

}

e->current\_pos = cu\_NextPos(e->current\_pos);

} while( e->current\_pos != 0 );

}

/\*==============================================================\*/

/\* user interface \*/

/\*==============================================================\*/

/\*

eval\_t chess\_EvalCurrBoard(uint8\_t color)

{

stack\_Init(0);

stack\_GetCurrElement()->current\_color = color;

ce\_LoopPieces();

return stack\_GetCurrElement()->best\_eval;

}

\*/

/\* clear any marks on the board \*/

void chess\_ClearMarks(void)

{

uint8\_t i;

for( i = 0; i < 64; i++ )

lrc\_obj.board[i] &= ~CP\_MARK\_MASK;

}

/\*

Mark all pieces which can do moves. This is done by setting flags on the global board

\*/

void chess\_MarkMovable(void)

{

stack\_Init(0);

//stack\_GetCurrElement()->current\_color = color;

lrc\_obj.check\_mode = CHECK\_MODE\_MOVEABLE;

ce\_LoopPieces();

}

/\*

Checks, if the piece can move from src\_pos to dest\_pos

src\_pos: The game position of a piece on the chess board

\*/

void chess\_MarkTargetMoves(uint8\_t src\_pos)

{

stack\_Init(0);

stack\_GetCurrElement()->current\_color = cp\_GetColor(cp\_GetFromBoard(src\_pos));

lrc\_obj.check\_src\_pos = src\_pos;

lrc\_obj.check\_mode = CHECK\_MODE\_TARGET\_MOVE;

ce\_LoopPieces();

}

/\*

first call should start with 255

this procedure will return 255 if

- there are no marks at all

- it has looped over all marks once

\*/

uint8\_t chess\_GetNextMarked(uint8\_t arg, uint8\_t is\_prev)

{

uint8\_t i;

uint8\_t pos = arg;

for(i = 0; i < 64; i++)

{

if ( is\_prev != 0 )

pos = cu\_PrevPos(pos);

else

pos = cu\_NextPos(pos);

if ( arg != 255 && pos == 0 )

return 255;

if ( cp\_IsMarked(cp\_GetFromBoard(pos)) )

return pos;

}

return 255;

}

/\* make a manual move: this is a little bit more than cu\_Move() \*/

void chess\_ManualMove(uint8\_t src, uint8\_t dest)

{

uint8\_t cp;

/\* printf("chess\_ManualMove %02x -> %02x\n", src, dest); \*/

/\* if all other things fail, this is the place where the game is to be decided: \*/

/\* ... if the KING is captured \*/

cp = cp\_GetFromBoard(dest);

if ( cp\_GetPiece(cp) == PIECE\_KING )

{

lrc\_obj.is\_game\_end = 1;

lrc\_obj.lost\_side\_color = cp\_GetColor(cp);

}

/\* clear ply history here, to avoid memory overflow \*/

/\* may be the last X moves can be kept here \*/

cu\_ReduceHistoryByFullMove();

/\* perform the move on the board \*/

cu\_Move(src, dest);

/\* update en passant double move positions: en passant position is removed after two half moves \*/

lrc\_obj.pawn\_dbl\_move[lrc\_obj.ply\_count&1] = ILLEGAL\_POSITION;

/\* update the global half move counter \*/

lrc\_obj.ply\_count++;

/\* make a small check about the end of the game \*/

/\* use at least depth 1, because we must know if the king can still move \*/

/\* this is: King moves at level 0 and will be captured at level 1 \*/

/\* so we check if the king can move and will not be captured at search level 1 \*/

stack\_Init(1);

ce\_LoopPieces();

/\* printf("chess\_ManualMove/analysis best\_from\_pos %02x -> best\_to\_pos %02x\n", stack\_GetCurrElement()->best\_from\_pos, stack\_GetCurrElement()->best\_to\_pos); \*/

/\* analyse the eval result \*/

/\* check if the other player has any moves left \*/

if ( stack\_GetCurrElement()->best\_from\_pos == ILLEGAL\_POSITION )

{

uint8\_t color;

/\* conditions: \*/

/\* 1. no King, should never happen, opposite color has won \*/

/\* this is already checked above at the beginning if this procedure \*/

/\* 2. King is under attack, opposite color has won \*/

/\* 3. King is not under attack, game is a draw \*/

uint8\_t i = 0;

color = lrc\_obj.ply\_count;

color &= 1;

do

{

cp = cp\_GetFromBoard(i);

/\* look for the King \*/

if ( cp\_GetPiece(cp) == PIECE\_KING )

{

if ( cp\_GetColor(cp) == color )

{

/\* check if KING is attacked \*/

if ( ce\_GetPositionAttackCount(i, color^1) != 0 )

{

/\* KING is under attack (check) and can not move: Game is lost \*/

lrc\_obj.is\_game\_end = 1;

lrc\_obj.lost\_side\_color = color;

}

else

{

/\* KING is NOT under attack (check) but can not move: Game is a draw \*/

lrc\_obj.is\_game\_end = 1;

lrc\_obj.lost\_side\_color = 2;

}

/\* break out of the loop \*/

break;

}

}

i = cu\_NextPos(i);

} while( i != 0 );

}

}

/\* let the computer do a move \*/

void chess\_ComputerMove(uint8\_t depth)

{

stack\_Init(depth);

//stack\_GetCurrElement()->current\_color = lrc\_obj.ply\_count;

//stack\_GetCurrElement()->current\_color &= 1;

cu\_ReduceHistoryByFullMove();

ce\_LoopPieces();

chess\_ManualMove(stack\_GetCurrElement()->best\_from\_pos, stack\_GetCurrElement()->best\_to\_pos);

}

/\*==============================================================\*/

/\* unix code \*/

/\*==============================================================\*/

#ifdef UNIX\_MAIN

#include <stdio.h>

#include <string.h>

char \*piece\_str[] = {

/\* 0x00 \*/

" ",

"wP",

"wN",

"wB",

/\* 0x04 \*/

"wR",

"wQ",

"wK",

"w?",

/\* 0x08 \*/

"w?",

"w?",

"w?",

"w?",

/\* 0x0c \*/

"w?",

"w?",

"w?",

"w?",

/\* 0x10 \*/

"b ",

"bP",

"bN",

"bB",

"bR",

"bQ",

"bK",

"b?",

"b?",

"b?",

"b?",

"b?",

"b?",

"b?",

"b?",

"b?"

};

void chess\_Thinking(void)

{

uint8\_t i;

uint8\_t cp = cp\_GetPiece(stack\_GetCurrElement()->current\_cp);

printf("Thinking: ", piece\_str[cp], stack\_GetCurrElement()->current\_pos);

for( i = 0; i <= lrc\_obj.curr\_depth; i++ )

printf("%s ", piece\_str[(lrc\_obj.stack\_memory+i)->current\_cp]);

printf(" \r");

}

void board\_Show(void)

{

uint8\_t i, j, cp;

char buf[10];

for ( i = 0; i < 8; i++ )

{

printf("%1d ", 7-i);

for ( j = 0; j < 8; j++ )

{

/\* get piece from global board \*/

cp = lrc\_obj.board[(7-i)\*8+j];

strcpy(buf, piece\_str[cp&COLOR\_PIECE\_MASK]);

if ( (cp & CP\_MARK\_MASK) != 0 )

{

buf[0] = '#';

}

/\* mask out any bits except color and piece index \*/

cp &= COLOR\_PIECE\_MASK;

printf("%s %02x ", buf, cp);

}

printf("\n");

}

}

int main(void)

{

uint8\_t depth = 3;

chess\_SetupBoard();

board\_Show();

puts("");

/\*

chess\_ClearMarks();

chess\_MarkMovable(COLOR\_WHITE);

board\_Show();

\*/

chess\_ManualMove(0x006, 0x066);

printf("lrc\_obj.is\_game\_end: %d\n" , lrc\_obj.is\_game\_end);

printf("lrc\_obj.lost\_side\_color: %d\n" , lrc\_obj.lost\_side\_color);

chess\_ComputerMove(2);

printf("lrc\_obj.is\_game\_end: %d\n" , lrc\_obj.is\_game\_end);

printf("lrc\_obj.lost\_side\_color: %d\n" , lrc\_obj.lost\_side\_color);

board\_Show();

}

#else

/\*==============================================================\*/

/\* display menu \*/

/\*==============================================================\*/

//#define MNU\_FONT font\_5x7

#define MNU\_FONT u8g\_font\_5x8r

//#define MNU\_FONT font\_6x9

#define MNU\_ENTRY\_HEIGHT 9

char \*mnu\_title = "Little Rook Chess";

char \*mnu\_list[] = { "New Game (White)", "New Game (Black)", "Undo Move", "Return" };

uint8\_t mnu\_pos = 0;

uint8\_t mnu\_max = 4;

void mnu\_DrawHome(uint8\_t is\_highlight)

{

uint8\_t x = lrc\_u8g->width - 35;

uint8\_t y = (lrc\_u8g->height-1);

uint8\_t t;

u8g\_SetFont(lrc\_u8g, u8g\_font\_5x7r);

u8g\_SetDefaultForegroundColor(lrc\_u8g);

t = u8g\_DrawStrP(lrc\_u8g, x, y -1, U8G\_PSTR("Options"));

if ( is\_highlight )

u8g\_DrawFrame(lrc\_u8g, x-1, y - MNU\_ENTRY\_HEIGHT +1, t, MNU\_ENTRY\_HEIGHT);

}

void mnu\_DrawEntry(uint8\_t y, char \*str, uint8\_t is\_clr\_background, uint8\_t is\_highlight)

{

uint8\_t t, x;

u8g\_SetFont(lrc\_u8g, MNU\_FONT);

t = u8g\_GetStrWidth(lrc\_u8g, str);

x = u8g\_GetWidth(lrc\_u8g);

x -= t;

x >>= 1;

if ( is\_clr\_background )

{

u8g\_SetDefaultBackgroundColor(lrc\_u8g);

u8g\_DrawBox(lrc\_u8g, x-3, (lrc\_u8g->height-1) - (y+MNU\_ENTRY\_HEIGHT-1+2), t+5, MNU\_ENTRY\_HEIGHT+4);

}

u8g\_SetDefaultForegroundColor(lrc\_u8g);

u8g\_DrawStr(lrc\_u8g, x, (lrc\_u8g->height-1) - y, str);

if ( is\_highlight )

{

u8g\_DrawFrame(lrc\_u8g, x-1, (lrc\_u8g->height-1) - y -MNU\_ENTRY\_HEIGHT +1, t, MNU\_ENTRY\_HEIGHT);

}

}

void mnu\_Draw(void)

{

uint8\_t i;

uint8\_t t,y;

/\* calculate hight of the complete menu \*/

y = mnu\_max;

y++; /\* consider also some space for the title \*/

y++; /\* consider also some space for the title \*/

y \*= MNU\_ENTRY\_HEIGHT;

/\* calculate how much space will be left \*/

t = u8g\_GetHeight(lrc\_u8g);

t -= y;

/\* topmost pos start half of that empty space from the top \*/

t >>= 1;

y = u8g\_GetHeight(lrc\_u8g);

y -= t;

y -= MNU\_ENTRY\_HEIGHT;

mnu\_DrawEntry(y, mnu\_title, 0, 0);

y -= MNU\_ENTRY\_HEIGHT;

for( i = 0; i < mnu\_max; i++ )

{

y -= MNU\_ENTRY\_HEIGHT;

mnu\_DrawEntry(y, mnu\_list[i], 0, i == mnu\_pos);

}

}

void mnu\_Step(uint8\_t key\_cmd)

{

if ( key\_cmd == CHESS\_KEY\_NEXT )

{

if ( mnu\_pos+1 < mnu\_max )

mnu\_pos++;

}

else if ( key\_cmd == CHESS\_KEY\_PREV )

{

if ( mnu\_pos > 0 )

mnu\_pos--;

}

}

uint8\_t chess\_key\_code = 0;

uint8\_t chess\_key\_cmd = 0;

#define CHESS\_STATE\_MENU 0

#define CHESS\_STATE\_SELECT\_START 1

#define CHESS\_STATE\_SELECT\_PIECE 2

#define CHESS\_STATE\_SELECT\_TARGET\_POS 3

#define CHESS\_STATE\_THINKING 4

#define CHESS\_STATE\_GAME\_END 5

uint8\_t chess\_state = CHESS\_STATE\_MENU;

uint8\_t chess\_source\_pos = 255;

uint8\_t chess\_target\_pos = 255;

const uint8\_t chess\_pieces\_body\_bm[] PROGMEM =

{

/\* PAWN \*/ 0x00, 0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00, /\* 0x00, 0x00, 0x00, 0x0c, 0x0c, 0x00, 0x00, 0x00, \*/

/\* KNIGHT \*/ 0x00, 0x00, 0x1c, 0x2c, 0x04, 0x04, 0x0e, 0x00,

/\* BISHOP \*/ 0x00, 0x00, 0x1c, 0x1c, 0x1c, 0x08, 0x00, 0x00, /\* 0x00, 0x00, 0x08, 0x1c, 0x1c, 0x08, 0x00, 0x00, \*/

/\* ROOK \*/ 0x00, 0x00, 0x00, 0x1c, 0x1c, 0x1c, 0x1c, 0x00,

/\* QUEEN \*/ 0x00, 0x00, 0x14, 0x1c, 0x08, 0x1c, 0x08, 0x00,

/\* KING \*/ 0x00, 0x00, 0x00, 0x08, 0x3e, 0x1c, 0x08, 0x00,

};

#ifdef NOT\_REQUIRED

/\* white pieces are constructed by painting black pieces and cutting out the white area \*/

const uint8\_t chess\_white\_pieces\_bm[] PROGMEM =

{

/\* PAWN \*/ 0x00, 0x00, 0x0c, 0x12, 0x12, 0x0c, 0x1e, 0x00,

/\* KNIGHT \*/ 0x00, 0x1c, 0x22, 0x52, 0x6a, 0x0a, 0x11, 0x1f,

/\* BISHOP \*/ 0x00, 0x08, 0x14, 0x22, 0x22, 0x14, 0x08, 0x7f,

/\* ROOK \*/ 0x00, 0x55, 0x7f, 0x22, 0x22, 0x22, 0x22, 0x7f,

/\* QUEEN \*/ 0x00, 0x55, 0x2a, 0x22, 0x14, 0x22, 0x14, 0x7f,

/\* KING \*/ 0x08, 0x1c, 0x49, 0x77, 0x41, 0x22, 0x14, 0x7f,

};

#endif

const uint8\_t chess\_black\_pieces\_bm[] PROGMEM =

{

/\* PAWN \*/ 0x00, 0x00, 0x18, 0x3c, 0x3c, 0x18, 0x3c, 0x00, /\* 0x00, 0x00, 0x0c, 0x1e, 0x1e, 0x0c, 0x1e, 0x00, \*/

/\* KNIGHT \*/ 0x00, 0x1c, 0x3e, 0x7e, 0x6e, 0x0e, 0x1f, 0x1f,

/\* BISHOP \*/ 0x00, 0x1c, 0x2e, 0x3e, 0x3e, 0x1c, 0x08, 0x7f, /\*0x00, 0x08, 0x1c, 0x3e, 0x3e, 0x1c, 0x08, 0x7f,\*/

/\* ROOK \*/ 0x00, 0x55, 0x7f, 0x3e, 0x3e, 0x3e, 0x3e, 0x7f,

/\* QUEEN \*/ 0x00, 0x55, 0x3e, 0x3e, 0x1c, 0x3e, 0x1c, 0x7f,

/\* KING -\*/ 0x08, 0x1c, 0x49, 0x7f, 0x7f, 0x3e, 0x1c, 0x7f,

};

#if defined(DOGXL160\_HW\_GR)

#define BOXSIZE 13

#define BOXOFFSET 3

#else

#define BOXSIZE 8

#define BOXOFFSET 1

#endif

u8g\_uint\_t chess\_low\_edge;

uint8\_t chess\_boxsize = 8;

uint8\_t chess\_boxoffset = 1;

void chess\_DrawFrame(uint8\_t pos, uint8\_t is\_bold)

{

u8g\_uint\_t x0, y0;

x0 = pos;

x0 &= 15;

if ( lrc\_obj.orientation != COLOR\_WHITE )

x0 ^= 7;

y0 = pos;

y0>>= 4;

if ( lrc\_obj.orientation != COLOR\_WHITE )

y0 ^= 7;

x0 \*= chess\_boxsize;

y0 \*= chess\_boxsize;

u8g\_SetDefaultForegroundColor(lrc\_u8g);

u8g\_DrawFrame(lrc\_u8g, x0, chess\_low\_edge - y0 - chess\_boxsize+1, chess\_boxsize, chess\_boxsize);

if ( is\_bold )

{

x0--;

y0++;

u8g\_DrawFrame(lrc\_u8g, x0, chess\_low\_edge - y0 - chess\_boxsize +1, chess\_boxsize+2, chess\_boxsize+2);

}

}

void chess\_DrawBoard(void)

{

uint8\_t i, j, cp;

const uint8\_t \*ptr; /\* pointer into PROGMEM \*/

if ( U8G\_MODE\_GET\_BITS\_PER\_PIXEL(u8g\_GetMode(lrc\_u8g)) > 1 )

{

for( i = 0; i < 8; i++ )

for( j = 0; j < 8; j++ )

{

uint8\_t x,y;

x = i;

x\*=chess\_boxsize;

y = j;

y\*=chess\_boxsize;

if ( ((i^j) & 1) == 0 )

u8g\_SetDefaultMidColor(lrc\_u8g);

else

u8g\_SetDefaultBackgroundColor(lrc\_u8g);

u8g\_DrawBox(lrc\_u8g, x,chess\_low\_edge-y-chess\_boxsize+1,chess\_boxsize,chess\_boxsize);

}

//u8g\_SetDefaultForegroundColor(lrc\_u8g);

}

else

{

uint8\_t x\_offset = 1;

u8g\_SetDefaultForegroundColor(lrc\_u8g);

for( i = 0; i < 8\*8; i+=8 )

{

for( j = 0; j < 8\*8; j+=8 )

{

if ( ((i^j) & 8) == 0 )

{

u8g\_DrawPixel(lrc\_u8g, j+0+x\_offset, chess\_low\_edge - i-0);

u8g\_DrawPixel(lrc\_u8g, j+0+x\_offset, chess\_low\_edge - i-2);

u8g\_DrawPixel(lrc\_u8g, j+0+x\_offset, chess\_low\_edge - i-4);

u8g\_DrawPixel(lrc\_u8g, j+0+x\_offset, chess\_low\_edge - i-6);

u8g\_DrawPixel(lrc\_u8g, j+2+x\_offset, chess\_low\_edge - i-0);

u8g\_DrawPixel(lrc\_u8g, j+2+x\_offset, chess\_low\_edge - i-6);

u8g\_DrawPixel(lrc\_u8g, j+4+x\_offset, chess\_low\_edge - i-0);

u8g\_DrawPixel(lrc\_u8g, j+4+x\_offset, chess\_low\_edge - i-6);

u8g\_DrawPixel(lrc\_u8g, j+6+x\_offset, chess\_low\_edge - i-0);

u8g\_DrawPixel(lrc\_u8g, j+6+x\_offset, chess\_low\_edge - i-2);

u8g\_DrawPixel(lrc\_u8g, j+6+x\_offset, chess\_low\_edge - i-4);

u8g\_DrawPixel(lrc\_u8g, j+6+x\_offset, chess\_low\_edge - i-6);

}

}

}

}

for ( i = 0; i < 8; i++ )

{

for ( j = 0; j < 8; j++ )

{

/\* get piece from global board \*/

if ( lrc\_obj.orientation == COLOR\_WHITE )

{

cp = lrc\_obj.board[i\*8+j];

}

else

{

cp = lrc\_obj.board[(7-i)\*8+7-j];

}

if ( cp\_GetPiece(cp) != PIECE\_NONE )

{

ptr = chess\_black\_pieces\_bm;

ptr += (cp\_GetPiece(cp)-1)\*8;

u8g\_SetDefaultForegroundColor(lrc\_u8g);

u8g\_DrawBitmapP(lrc\_u8g, j\*chess\_boxsize+chess\_boxoffset-1, chess\_low\_edge - (i\*chess\_boxsize+chess\_boxsize-chess\_boxoffset), 1, 8, ptr);

if ( cp\_GetColor(cp) == lrc\_obj.strike\_out\_color )

{

ptr = chess\_pieces\_body\_bm;

ptr += (cp\_GetPiece(cp)-1)\*8;

u8g\_SetDefaultBackgroundColor(lrc\_u8g);

u8g\_DrawBitmapP(lrc\_u8g, j\*chess\_boxsize+chess\_boxoffset-1, chess\_low\_edge - (i\*chess\_boxsize+chess\_boxsize-chess\_boxoffset), 1, 8, ptr);

}

}

}

}

if ( (chess\_source\_pos & 0x88) == 0 )

{

chess\_DrawFrame(chess\_source\_pos, 1);

}

if ( (chess\_target\_pos & 0x88) == 0 )

{

chess\_DrawFrame(chess\_target\_pos, 0);

}

}

void chess\_Thinking(void)

{

}

void chess\_Init(u8g\_t \*u8g, uint8\_t body\_color)

{

lrc\_u8g = u8g;

chess\_low\_edge = u8g\_GetHeight(lrc\_u8g);

chess\_low\_edge--;

if ( U8G\_MODE\_GET\_BITS\_PER\_PIXEL(u8g\_GetMode(lrc\_u8g)) == 1 )

{

chess\_boxsize = 8;

chess\_boxoffset = 1;

}

else

{

/\*

if ( u8g\_GetHeight(lrc\_u8g) >= 12\*8 )

{

chess\_boxsize = 12;

chess\_boxoffset = 3;

}

else \*/ if ( u8g\_GetHeight(lrc\_u8g) >= 11\*8 )

{

chess\_boxsize = 10;

chess\_boxoffset = 2;

}

else

{

chess\_boxsize = 8;

chess\_boxoffset = 1;

}

if ( u8g\_GetHeight(lrc\_u8g) > 64 )

chess\_low\_edge -= (u8g\_GetHeight(lrc\_u8g)-chess\_boxsize\*8) / 2;

}

lrc\_obj.strike\_out\_color = body\_color;

chess\_SetupBoard();

}

void chess\_Draw(void)

{

if ( chess\_state == CHESS\_STATE\_MENU )

{

if ( lrc\_obj.ply\_count == 0)

mnu\_max = 2;

else

mnu\_max = 4;

mnu\_Draw();

}

else

{

chess\_DrawBoard();

{

uint8\_t i;

uint8\_t entries = lrc\_obj.chm\_pos;

if ( entries > 4 )

entries = 4;

u8g\_SetFont(lrc\_u8g, u8g\_font\_5x7);

u8g\_SetDefaultForegroundColor(lrc\_u8g);

for( i = 0; i < entries; i++ )

{

#if defined(DOGXL160\_HW\_GR) || defined(DOGXL160\_HW\_BW)

dog\_DrawStr(u8g\_GetWidth(lrc\_u8g)-35, u8g\_GetHeight(lrc\_u8g)-8\*(i+1), font\_5x7, cu\_GetHalfMoveStr(lrc\_obj.chm\_pos-entries+i));

#else

u8g\_DrawStr(lrc\_u8g, u8g\_GetWidth(lrc\_u8g)-35, 8\*(i+1), cu\_GetHalfMoveStr(lrc\_obj.chm\_pos-entries+i));

#endif

}

}

if ( chess\_state == CHESS\_STATE\_SELECT\_PIECE )

mnu\_DrawHome(chess\_source\_pos == 255);

else if ( chess\_state == CHESS\_STATE\_SELECT\_TARGET\_POS )

mnu\_DrawHome(chess\_target\_pos == 255);

else

mnu\_DrawHome(0);

if ( chess\_state == CHESS\_STATE\_GAME\_END )

{

switch( lrc\_obj.lost\_side\_color )

{

case COLOR\_WHITE:

mnu\_DrawEntry(u8g\_GetHeight(lrc\_u8g) / 2-2, "Black wins", 1, 1);

break;

case COLOR\_BLACK:

mnu\_DrawEntry(u8g\_GetHeight(lrc\_u8g) / 2-2, "White wins", 1, 1);

break;

default:

mnu\_DrawEntry(u8g\_GetHeight(lrc\_u8g) / 2-2, "Stalemate", 1, 1);

break;

}

}

}

}

void chess\_Step(uint8\_t keycode)

{

if ( keycode == CHESS\_KEY\_NONE )

{

chess\_key\_cmd = chess\_key\_code;

chess\_key\_code = CHESS\_KEY\_NONE;

}

else

{

chess\_key\_cmd = CHESS\_KEY\_NONE;

chess\_key\_code = keycode;

}

//chess\_ComputerMove(2);

switch(chess\_state)

{

case CHESS\_STATE\_MENU:

mnu\_Step(chess\_key\_cmd);

if ( chess\_key\_cmd == CHESS\_KEY\_SELECT )

{

if ( mnu\_pos == 0 )

{

chess\_SetupBoard();

lrc\_obj.orientation = 0;

chess\_state = CHESS\_STATE\_SELECT\_START;

}

else if ( mnu\_pos == 1 )

{

chess\_SetupBoard();

lrc\_obj.orientation = 1;

chess\_state = CHESS\_STATE\_THINKING;

}

else if ( mnu\_pos == 2 )

{

if ( lrc\_obj.ply\_count >= 2 )

{

cu\_UndoHalfMove();

cu\_UndoHalfMove();

lrc\_obj.ply\_count-=2;

if ( lrc\_obj.ply\_count == 0 )

mnu\_pos = 0;

}

chess\_state = CHESS\_STATE\_SELECT\_START;

}

else if ( mnu\_pos == 3 )

{

chess\_state = CHESS\_STATE\_SELECT\_START;

}

}

break;

case CHESS\_STATE\_SELECT\_START:

chess\_ClearMarks();

chess\_MarkMovable();

chess\_source\_pos = chess\_GetNextMarked(255, 0);

chess\_target\_pos = ILLEGAL\_POSITION;

chess\_state = CHESS\_STATE\_SELECT\_PIECE;

break;

case CHESS\_STATE\_SELECT\_PIECE:

if ( chess\_key\_cmd == CHESS\_KEY\_NEXT )

{

chess\_source\_pos = chess\_GetNextMarked(chess\_source\_pos, 0);

}

else if ( chess\_key\_cmd == CHESS\_KEY\_PREV )

{

chess\_source\_pos = chess\_GetNextMarked(chess\_source\_pos, 1);

}

else if ( chess\_key\_cmd == CHESS\_KEY\_SELECT )

{

if ( chess\_source\_pos == 255 )

{

chess\_state = CHESS\_STATE\_MENU;

}

else

{

chess\_ClearMarks();

chess\_MarkTargetMoves(chess\_source\_pos);

chess\_target\_pos = chess\_GetNextMarked(255, 0);

chess\_state = CHESS\_STATE\_SELECT\_TARGET\_POS;

}

}

break;

case CHESS\_STATE\_SELECT\_TARGET\_POS:

if ( chess\_key\_cmd == CHESS\_KEY\_NEXT )

{

chess\_target\_pos = chess\_GetNextMarked(chess\_target\_pos, 0);

}

else if ( chess\_key\_cmd == CHESS\_KEY\_PREV )

{

chess\_target\_pos = chess\_GetNextMarked(chess\_target\_pos, 1);

}

else if ( chess\_key\_cmd == CHESS\_KEY\_BACK )

{

chess\_ClearMarks();

chess\_MarkMovable();

chess\_target\_pos = ILLEGAL\_POSITION;

chess\_state = CHESS\_STATE\_SELECT\_PIECE;

}

else if ( chess\_key\_cmd == CHESS\_KEY\_SELECT )

{

chess\_ManualMove(chess\_source\_pos, chess\_target\_pos);

if ( lrc\_obj.is\_game\_end != 0 )

chess\_state = CHESS\_STATE\_GAME\_END;

else

chess\_state = CHESS\_STATE\_THINKING;

/\* clear marks as some kind of feedback to the user... it simply looks better \*/

chess\_source\_pos = ILLEGAL\_POSITION;

chess\_target\_pos = ILLEGAL\_POSITION;

chess\_ClearMarks();

}

break;

case CHESS\_STATE\_THINKING:

chess\_ComputerMove(2);

if ( lrc\_obj.is\_game\_end != 0 )

chess\_state = CHESS\_STATE\_GAME\_END;

else

chess\_state = CHESS\_STATE\_SELECT\_START;

break;

case CHESS\_STATE\_GAME\_END:

if ( chess\_key\_cmd != CHESS\_KEY\_NONE )

{

chess\_state = CHESS\_STATE\_MENU;

chess\_SetupBoard();

}

break;

}

}

#endif