Home * Engines * CPW-Engine * **movegen(0x88)**

This page holds the movegenerator of the engine CPW. For function definitions see CPW-Engine_movegen_h.

Current version contains just one move generator with some conditional statements deciding whether we generate a full set of moves, or captures only. This is of course suboptimal, but the speed loss is really small. For that reason we postpone writing a separate capture generator until we are 100% sure about the exact shape of te smove struct, board representation etc.

```
#include "stdafx.h"
#include "0x88_math.h"
#include "movegen.h"

U8 movecount;

smove * m;

bool slide[5] = { 0, 1, 1, 1, 0 };
char vectors[5] = { 8, 8, 4, 4, 8 };
char vector[5][8] = {
    { SW, SOUTH, SE, WEST, EAST, NW, NORTH, NE },
    { SW, SOUTH, SE, WEST, EAST, NW, NORTH, NE },
    { SOUTH, WEST, EAST, NORTH                 },
    { SW, SE, NW, NE                           },
    { -33, -31, -18, -14, 14, 18, 31, 33       }
    };


//returns movecount
U8 movegen(smove * moves, U8 tt_move, bool captures) {

    m = moves;

    movecount = 0;

    if (!captures) {
        //Castling
        if ( b.stm == WHITE ) {
            if ( b.castle & CASTLE_WK ) {
                if ( ( b.pieces[F1] == PIECE_EMPTY ) &&
                     ( b.pieces[G1] == PIECE_EMPTY ) &&
                     ( !isAttacked(!b.stm,E1) ) &&
                     ( !isAttacked(!b.stm,F1) ) &&
```

```
                ( !isAttacked(!b.stm,G1) ) )
                movegen_push(E1,G1,KING,PIECE_EMPTY,MFLAG_CASTLE);
        }
        if ( b.castle & CASTLE_WQ ) {
            if ( ( b.pieces[B1] == PIECE_EMPTY ) &&
                 ( b.pieces[C1] == PIECE_EMPTY ) &&
                 ( b.pieces[D1] == PIECE_EMPTY ) &&
                 ( !isAttacked(!b.stm,E1) ) &&
                 ( !isAttacked(!b.stm,D1) ) &&
                 ( !isAttacked(!b.stm,C1) ) )
                movegen_push(E1,C1,KING,PIECE_EMPTY,MFLAG_CASTLE);
        }
    } else {
        if ( b.castle & CASTLE_BK ) {
            if ( ( b.pieces[F8] == PIECE_EMPTY ) &&
                 ( b.pieces[G8] == PIECE_EMPTY ) &&
                 ( !isAttacked(!b.stm,E8) ) &&
                 ( !isAttacked(!b.stm,F8) ) &&
                 ( !isAttacked(!b.stm,G8) ) )
                movegen_push(E8,G8,KING,PIECE_EMPTY,MFLAG_CASTLE);
        }
        if ( b.castle & CASTLE_BQ ) {
            if ( ( b.pieces[B8] == PIECE_EMPTY ) &&
                 ( b.pieces[C8] == PIECE_EMPTY ) &&
                 ( b.pieces[D8] == PIECE_EMPTY ) &&
                 ( !isAttacked(!b.stm,E8) ) &&
                 ( !isAttacked(!b.stm,D8) ) &&
                 ( !isAttacked(!b.stm,C8) ) )
                movegen_push(E8,C8,KING,PIECE_EMPTY,MFLAG_CASTLE);
        }
    }
}


for (S8 sq=0;sq<120;sq++) {

    if (b.color[sq] == b.stm) {

        if (b.pieces[sq] == PAWN) {
            movegen_pawn_move(sq, captures);
            movegen_pawn_capt(sq);
        } else {

            for (char dir=0;dir<vectors[b.pieces[sq]];dir++) {

                for (char pos = sq;;) {
```

```
                        pos = pos + vector[b.pieces[sq]][dir];

                        if (! IS_SQ(pos)) break;

                        if (b.pieces[pos] == PIECE_EMPTY) {
                            if (!captures)

      movegen_push(sq, pos, b.pieces[sq], PIECE_EMPTY, MFLAG_NORMAL);
                        }
                        else if (b.color[pos] != b.stm) {
                            mo
vegen_push(sq, pos, b.pieces[sq], b.pieces[pos], MFLAG_CAPTURE);
                            break;
                        }
                        else {
                            break;
                        }

                        if (! slide[b.pieces[sq]]) break;
                    }
                }
            }
        }
    }

    /* if we have a best-
move fed into movegen(), then increase its score */

    if (  ( tt_move != -1 ) && ( tt_move < movecount ) ) moves[
tt_move].score = SORT_HASH;

    return movecount;
}

U8 movegen_qs(smove * moves) {

    m = moves;

    movecount = 0;

        for (S8 sq=0;sq<120;sq++) {

        if (b.color[sq] == b.stm) {

            if (b.pieces[sq] == PAWN) {
```

```
                movegen_pawn_move(sq, 1);
                movegen_pawn_capt(sq);
            } else {

                for (char dir=0;dir<vectors[b.pieces[sq]];dir++) {

                    for (char pos = sq;;) {

                        pos = pos + vector[b.pieces[sq]][dir];

                        if (! IS_SQ(pos)) break;

                        if (b.pieces[pos] != PIECE_EMPTY) {
                            if (b.color[pos] != b.stm) {

      movegen_push(sq, pos, b.pieces[sq], b.pieces[pos]
, MFLAG_CAPTURE);
                                break;
                            }
                            else break; // hitting own piece
                        }

                        if (! slide[b.pieces[sq]]) break;
                    }
                }
            }
        }
    }

    return movecount;
}


void movegen_pawn_move(S8 sq, bool promotion_only) {

    if ( b.stm == WHITE ) {
        if (promotion_only && (ROW(sq) < 7)) return;

        if (b.pieces[sq+NORTH] == PIECE_EMPTY) {
            movegen_push(sq, sq+
NORTH, PAWN, PIECE_EMPTY, MFLAG_NORMAL);
            if ( ( ROW(sq) == 1 ) &&
                 ( b.pieces[sq+NN] == PIECE_EMPTY )
               ) {
                movegen_push(sq, sq+NN, PAWN, PIECE_EMPTY, MFLAG_EP);
            }
```

```
        }
    } else {
        if (promotion_only && (ROW(sq) > 1)) return;

        if (b.pieces[sq+SOUTH] == PIECE_EMPTY) {
            movegen_push(sq, sq+
SOUTH, PAWN, PIECE_EMPTY, MFLAG_NORMAL);
            if ( ( ROW(sq) == 6 ) &&
                 ( b.pieces[sq+SS] == PIECE_EMPTY )
               ) {
                movegen_push(sq, sq+SS, PAWN, PIECE_EMPTY, MFLAG_EP);
            }
        }
    }
}

void movegen_pawn_capt(S8 sq) {
    if (b.stm == WHITE) {
        if (IS_SQ(sq+NW) && ((b.ep==sq+NW) || (b.color[sq+NW] == (b.
stm^1)))) {
            movegen_push(sq, sq+NW, PAWN, b.pieces[sq+NW]
, MFLAG_CAPTURE);
        }
        if (IS_SQ(sq+NE) && ((b.ep==sq+NE) || (b.color[sq+NE] == (b.
stm^1)))) {
            movegen_push(sq, sq+17, PAWN, b.pieces[sq+NE]
, MFLAG_CAPTURE);
        }
    } else {
        if (IS_SQ(sq+SE) && ((b.ep==sq+SE) || (b.color[sq+SE] == (b.
stm^1)))) {
            movegen_push(sq, sq+SE, PAWN, b.pieces[sq+SE]
, MFLAG_CAPTURE);
        }
        if (IS_SQ(sq+SW) && ((b.ep==sq+SW) || (b.color[sq+SW] == (b.
stm^1)))) {
            movegen_push(sq, sq+SW, PAWN, b.pieces[sq+SW]
, MFLAG_CAPTURE);
        }
    }
}


void movegen_push(char from, char
 to, U8 piece_from, U8 piece_cap, char flags) {
```

```
    m[movecount].from = from;
    m[movecount].to = to;
    m[movecount].piece_from = piece_from;
    m[movecount].piece_to = piece_from;
    m[movecount].piece_cap = piece_cap;
    m[movecount].flags = flags;
    m[movecount].ply = b.ply;
    m[movecount].castle = b.castle;
    m[movecount].ep = b.ep;
    m[movecount].id = movecount;

    m[movecount].score = sd.history[from][to];

    /* score for capture:
        Add the value of the captured piece and the id of the attackin
g piece.
        So that if two pieces can attack the same target, the one with
 the higher id (eg. Pawn=5) gets searched first.
    */
    if (piece_cap != PIECE_EMPTY)
        m[movecount].score = SORT_CAPT + e.SORT_VALUE[
piece_cap] + piece_from;

    //score for ep-capture
    if ((piece_from == PAWN) && (to == b.ep)) {
        m[movecount].score = SORT_CAPT + e.SORT_VALUE[PAWN] + 5;
        m[movecount].flags = MFLAG_EPCAPTURE;
    }

    if ((piece_from == PAWN) && ( (ROW(to)==0)||(ROW(to)==7) )) {
        m[movecount].flags |= MFLAG_PROMOTION;

        for (char prompiece = QUEEN; prompiece <= KNIGHT;
 prompiece++) {
            m[movecount+prompiece-1] = m[movecount];
            m[movecount+prompiece-1].piece_to = prompiece;
            m[movecount+prompiece-1].score += SORT_PROM + e.
SORT_VALUE[prompiece];
            m[movecount+prompiece-1].id = movecount+prompiece-1;
        }
        movecount += 3;
    }

    movecount++;

}
```

```
void movegen_sort(U8 movecount, smove * m, U8 current) {


//find the move with the highest score - hoping for an early cutoff

    int high = current;

    for (int i=current+1; i<movecount; i++) {
        if (m[i].score > m[high].score)
            high = i;
    }

    smove temp = m[high];
    m[high] = m[current];
    m[current] = temp;
}
```

What links here?

Page	Date Edited
0x88	Nov 28, 2016
CPW-Engine	Dec 31, 2014
CPW-Engine_movegen(0x88)	Dec 30, 2014
CPW-Engine_root	Sep 27, 2008
Move Generation	Jan 29, 2018
Table-driven Move Generation	Feb 19, 2017

Up one Level