INTRODUCTION

Chess Prodigy is a dedicated chess computer based on ATmega644 microcontroller. It is a nostalgia project to recall the age of standalone chess computers.

The final official release is v1.2 and features the following characteristics:

HW:

- ROM: 64kB (opening book included)
- RAM: 4kB
- CPU speed: 8Mz
- Input: 4 x push buttons
- Output: LCD display, buzzer

SW:

- Code size: 48kB
- Opening book: 10kB (hosting around 3k moves)
- Data size: 3kB
- Playing strength:
    - Colditz: 1856 ELO (as indicated by EloStatTS)
    - Tournament (3 min fixed time / move) estimated: 1600 ELO

ABOUT

The dream of forging a dedicated chess computer

Hello there, I am Cristian Zaslo the author of Chess Prodigy chess computer (hereinafter referred to as **CP1**). I developed this project over a period of about five years, in my spare time. But the dream of programming a computer to play chess goes far back to the mid of 80s when my parents purchased my first personal computer: a ZX Spectrum 48k. It did cost a little fortune since those amazing units were typically "smuggled in" by some foreign students in Timisoara. If my memory serves me well, they paid as much as 15k ROL for it, which at that time meant a quarter of a Dacia passenger car.

In the first year I did nothing else but playing the classics (Manic Miner, Bruce Lee, Hungry Horace, Invaders, Fist, and so on). Then, gradually, I developed interest for board games such as chess, reversi or backgammon. Soon after, I came across a very special one: Superchess 3.0 since it was by far the best chess program at that time, capable of playing good and aggressive moves in a fair amount of time (5 - 10 min) and ultimately, strong enough to put in trouble, the most experienced chess player in family my uncle Gheorghe.

The close encounter with Superchess 3.0, determined me to roll up my sleeves and make my own chess program in ZX BASIC. The first release came out approximately one year later and in spite of fully obeying the chess rules, it could not examine more than 100 nodes in a five-minute timebox. Simply the BASIC interpreter was too slow for chess programming, let alone my programming skills.

But the golden age of ZX Spectrum was soon to be turned into sweet memories by the first PC units (x86 family) with much more RAM, CPU power and high performance C / Pascal compilers; and lest I forget: floppy drives instead of tape cassette a major data storage breakthrough; goodbye "R Tape Loading Error", welcome "F2 - Save" !

So, time for a new chess program: it was named "ProChess 1.0", was written in Pascal and later on, integrated into a Dephi 3.0 environment, for a more convenient graphical interface.

Although written completely from scratch, I tend to regard it as "inspired" since it incorporated some of the ideas implemented in Turbo Chess, a didactic chess program by Kaare Danielsen, written in Turbo Pascal and published as source code in 1985 on floppy-disk along with the book Turbo GameWorks by Borland International. ProChess 1.0 was capable of playing a decent level of chess, being able to outperform Cyrus-IS Chess, under tournament conditions. Of course, ProChess 1.0 despite its Pascal origins, was running on a much more powerful machine, say a 486@120Mz processor with enough memory to accommodate exotic ideas such as transposition tables.

But the victory against Cyrus-IS Chess was short lived - it was merely unfair, due to the sheer difference in computing power - and soon after I started envisioning a chess engine running on a low-end hardware, in fact a dedicated chess computer.

Meanwhile, I got an embedded SW developer job at a multinational company, got a family and for almost twenty years I had to keep at bay my interest in chess programming.

But it was in the early 2000 when Ed Schroeder retired from competitions and made public the anatomy of Rebel at his Programmer Corner - an invaluable gift for the entire community of chess programming aficionados. Coming across the Rebel description, re-ignited my interest in chess programming. Inspired now, and enarmed with new ideas to try, I decided to roll again my sleeves and consequently launched the low-end chess computer project.

Down here a list of release dates, as follows:

Chess Prodigy 1.0: Jan 2020, released

Chess Prodigy 1.1: Oct 2020, released

Chess Prodigy 1.2: Jul 2021 released

For implementation insights, please refer the "Concept" section.

Concept – SW

Board and piece representation

For board representation the **0x88** approach is used as described here. For coding of the chess men, the following values are assigned:

0 -> empty square

1 .. 6 -> white pawn up to white king

7 .. 12 -> black pawn up to black king

Thus, 4 nibbles are enough to code each man on the chessboard. In addition, one extra bit (bit #4) is used to store the "has moved" information. When moved for the first time, bit #4 gets set. For example, after pushing the white pawn to A3, the content of Board [0x20] becomes 0x11.

In addition, **CP1** uses two sorted Piece-Lists, one for white and one for black to the location of each chessman on board, for more convenient access. Down here the content of the white piece-list corresponding to the initial position:

WPieceTable [16] = {0x04, 0x03, 0x00, 0x07, 0x02, 0x05, 0x01, 0x06, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17}

Here, WPieceTab [0] points to the square occupied by white king, WPieceTab [1] to white queen, and so on.

Whenever a piece gets captured its place within PieceTab structure is replaced with **0xFF** (out of board) information.

Besides, **CP1** keeps track of some more state variables, needed in the move generation process:

- side to move (white or black)
- en-passant capture square (filled with 0xFF if none)
- 50 move counter (to detect draws by 50-move rule)

WPieceTable	BPieceTable
0x04	0x74
0x03	0x75
0x00	0x70
0x07	0x77
0x02	0x72
0x05	0x75
0x01	0x52
0x33	0x76
0x10	0x60
0x11	0x61
0x12	0x63
0x34	0x64
0x15	0x65
0x16	0x66
0x17	0x67
0xFF	0xFF

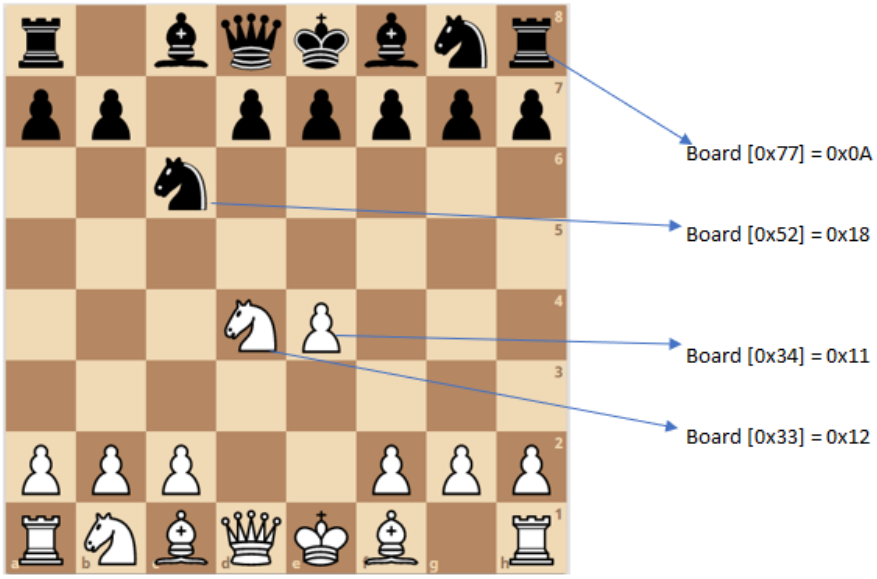

Fig.1 - board representation

Move generator:

Because of the limited memory resources, **CP1** generates the moves for a given position one at a time (staged move generation). This approach makes the move generation process somehow more complex, but there's no other way around, when RAM is scarce. Moves are generated according to the following sequence:

main variation: moves along the best line as determined by the previous iteration

winning captures: at each ply in the search, the program records a list of up to two vulnerable pieces for both sides; they are called **Hang_1**/**Hang_2** for the side to move, and **Threat_1/Threat_2** for the other side, and they are sorted by the estimated exchange outcome (kind of SEE algorithm). Then, captures on Threat_1 are searched, followed by captures on Threat_2. And of course, the lowest capturer is tried first (MVV-LVA approach). In the example below, the bishop capture a4-b5, will be tried before the rook capture, f5-g6.

Threats	Hangings
0x41	0x30
0x56	0xFF

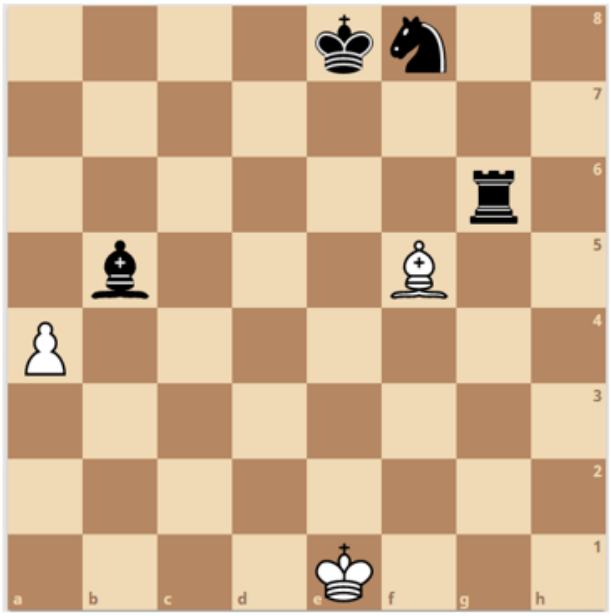

Fig.2 - threats and captures, white's perspective

pawn promotions: Q-promotions first, followed by R/B/N promotions

good captures as follows: QxQ, RxR, B/NxB/N, PxP

en-passant captures

killer moves: CP1 stores two (distinct) killer moves per ply, say **Kill_1** and **Kill_2**. As soon a move is found to improve Alpha, Kill_1 content is transferred into Kill_2 slot and then, Kill_1 is replaced by move. Because of move ordering strategy, the following categories are never allowed to pollute the killer slots: main variation, winning captures, promotions, good captures, or en-passant captures

castlings: next come castlings, if available

hangings: moving away a piece under attack is typically a good idea, and hereby, they are tried out next

others: no special remark here, normal moves (including losing captures) are tried out according to the order they are supplied by the move generator

Remark: most of the times a null move with **R = 2** is tried right before winning captures; adding a null move to a chess program may speed up the search tremendously, especially with higher iterations. Nonetheless, null move generation is skipped if one of the following conditions is met:

- the side to move is in check
- the opponent was in check on the ply before
- null move was already generated along the current branch (no "double null" move allowed)

- if one side has no officers left
- *Pos_Val – Max_Loss < Beta*; on each ply, the *Max_Loss* value is estimated by a SEE algorithm, and in practice lies between 0 and *Value (Hang_1)*. In plain words, it gives a wide berth to null moves that are unlikely to fail high. *PosVal* describes the static evaluation of the node under analysis.

Alpha Beta search

CP1 uses an iterative search strategy based on Alpha-Beta algorithm, with some flavors, as follows:

- initial search depth is one ply, with wide-open *(+/- INFINITY*) window; then it is gradually increased by one ply until time is up
- the higher and lower windows are defined based on the outcome of the previous search:

  ➔ *HIGH = Old_Score + 0.5 * Pawn*
  ➔ *LOW = Old_Score - 1 * Pawn*

- in case the search fails high (*Score >= HIGH*) no research is done; instead, the search depth is increased, and the analysis carries on
- conversely, in case of a fail low (*Score <= LOW*) the two windows are set to *+/- INFINITY* a research is done; generally we fear the fail low situations (because of extra time spent) and this accounts for setting the *LOW* window to a rather conservative value.
- if the main variation returns a *Score < Old_Score* (but greater than *Alpha*) the *Beta* limit is reduced by *0.2 * Pawn* (since the remaining moves probably won't make it above the newly set *Beta*)
- when a new better score (greater than *Alpha*) is backup-ed at the root level (resulting in a new principal variation), a small tolerance of *0.1 * Pawn* is added upon (tolerance search). This means that the *Alpha* limit gets slightly higher, and consequently would discard all variations that don't feel like significantly improving the best score so far.

Tree-Searching (general)

While analyzing a position, **CP1** adjusts the search based on the horizon proximity; the closer it gets the more selective it becomes. As a result, we can figure out four regions of interest, along with their specific attributes:

- Region "A" (away from horizon):
  - Stretches from Root to Ply_3 (up to two plies before horizon)
  - All moves are considered
  - Reductions and extensions are enabled
  - Null move (with R = 2) is used

- Region “B” (close to horizon):
  - Ply_2 and Ply_1 (the two plies before horizon)
  - All moves considered, but futility pruning applies
  - Extensions enabled; reductions disabled
  - Null move skipped; static pruning used instead

- Region “C” (horizon, quiescence search):
  - Ply_0 and Ply_-1 (sometimes extendable down to Ply_-3)
  - Standing Pat applied
  - Only checks and queen promotions considered
  - Extensions and reductions disabled

- Region “D” (beyond horizon, quiescence search)
  - Ply_-2 and beyond (sometimes starting with Ply_-4)
  - Standing Pat applied
  - Only queen promotions considered
  - Extensions and reductions disabled

See in Fig 3 a typical tree branch (no extensions / reductions) in a 3-ply search.

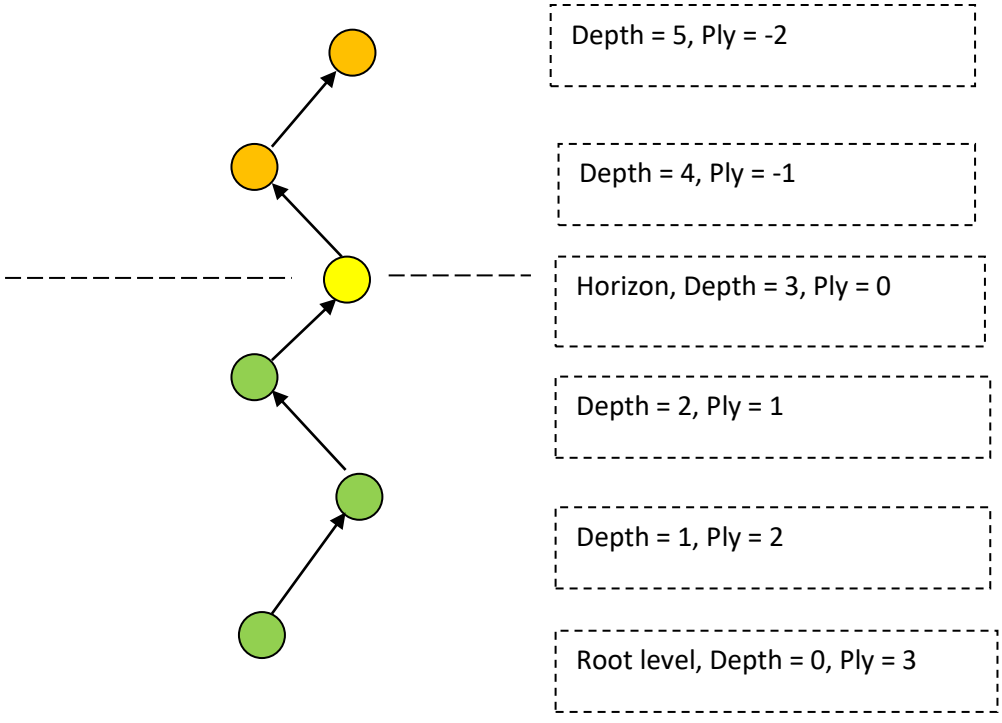

Fig 3 – a three-ply search tree

Tree-Searching (Null Move)

In region "A" (up to 2-plies before horizon) a null-move with R = 2 is tried immediately after the main variation; in case *Score >= Beta* the remaining moves on that ply are discarded (beta cut-off) and *Score* is returned.

In some particular situations the null-move is skipped:
- The side to move is in check
- The side to move on the previous *Ply* was in check
- A null-move was already played along that line ("double null-moves" forbidden)
- One side is left without officers
- *Pos_Val – Max_Loss < Beta*. If this condition is met, a null-move would probably not result in cut-off so why wasting the time on it ? Of course, *PosVal* describes the static evaluation of the node under analysis.

Tree-Searching (Static Pruning)

Static pruning forms the basis of selective search in **CP1**, the approach was to a certain extent inspired from Rebel and it applies to region "B" (Ply_1 and Ply_2). Essentially if the following conditions are met, the entire subtree is pruned off and *PosVal* returned:

- Not on root level
- No null-move was played along the line
- Side to move is not in check
- The side to move on previous ply was not in check
- The move on previous ply is not a pawn push to 6^{th} or 7^{th} rank
- The move on previous ply does not attack (directly or X-ray) the king area (one of the eight squares around the enemy king)
- The move on previous ply does not place the piece close to king (*Manhattan distance to king <= 4*)
- *PosVal – Max_Loss – 0.25 * Pawn >= Beta*. Here *Max_Loss* is a value that tries to anticipate the maximum loss for the side to move, by the means of a static exchange evaluator; in practice *Max_Loss <= Val (Hung_1)*.

Tree-Searching (Futility Pruning)

Futility pruning is a technique inspired from Rebel and Dark Thought and applied to region "B" (two plies before horizon). The benefit comes from discarding the moves with low potential of raising A*lpha*.

A candidate move on Ply_1 is discarded if:

- Side to move is in not check
  AND

- At least one legal move has been found
  AND
- The candidate move is not a mainline, winning capture, promotion, good
- capture or enpassant.
  AND
- Best score so far (on Ply_1) is not close to mate value (imminent loss)
  AND
- Move does not check
  AND
- Move is not a capture
  AND
- Move is not a pawn push to 7th rank
  AND
- *PosVal* + 3 * *Pawn_Val* <= *Alpha*

Likewise, a candidate move on Ply_2 is discarded if:

- Side to move is in not check
  AND
- At least one legal move has been found
  AND
- The candidate move is not a mainline, winning capture, promotion, good
- capture or enpassant.
  AND
- Best score so far (on Ply_2) is not close to mate value (imminent loss)
  AND
- Move does not check
  AND
- Move is not a capture
- AND
  Move is not a pawn push to 7th rank
- AND
- *PosVal + 5 * Pawn_Val <= Alpha*

Tree-Searching (Check evasions)

CP1 employs a special technique to deal with check evasions in regions C and D (horizon and beyond) namely, where standing pat is not allowed. Essentially it generates all check evasions until a standing pat value is borne out (e.g., *Score.Mtrl >= PosVal.Mtrl*) and as a result, the remaining moves are skipped as soon as this condition holds true. Worth mentioning that *PosVal.Mtrl* indicates the material balance (sum of piece values of each side) of the respective position. Because of move ordering (winning captures tried first), in general this approach gives a nice node reduction in practice.

Tree-Searching (Q-Search)

Because Q-Search (regions C and D) is expensive by nature, **CP1** was designed to skimp on some resources typically used in this phase. Unless in check, it initializes the standing pat score not with *PosVal*, but rather with *PosVal + MaxGain*, where *MaxGain* tries to estimate the maximum material gain possible by the means of a SEE, provided that the side to move has a threat at hand; in practice *MaxGain <= Threat_1*. If *PosVal + MaxGain >= Beta*, the cut-off condition is met, and no other moves will be generated; otherwise, the program looks only at checks and queen promotions on Ply_0 and Ply_-1, and solely at queen promotions starting with Ply_-2 and beyond. In other words, **CP1** restricts the checks to the first two plies of the Q-search. However, this threshold is furthermore increased by two plies (i.e., Ply_-2, Ply_-3) in case the opponent has only one legal reply. This is a simplified variation of the successful scheme called "long checks", as described on Rebel's webpage.

In the following situations the Q-search is regarded futile and hereby no moves are generated:

- Side to move is not in check and *Score* (standing pat) >= *Beta* (fail high, *Score* returned)
- Side to move is not in check, but an *Alpha* increase is unlikely (opponent was not in check before, no pawn promotion possible and *PosVal + 9 * PawnVal <= Alpha*)
- Side to move is in check, but an Alpha increase is unlikely (*PosVal + MaxGain + 1 * PawnVal <= Alpha*)

Tree-Searching (Extensions)

Extensions are powerful and in my opinion is a "must have" feature in any program running on a low-end HW.

CP1 employs the following extension techniques in regions A and B, with the purpose of identifying tactical lines at low depths. Extensions are permitted as long as the current depth is lower than iteration value multiplied by two; e.g. if we are in iteration #4 then extensions are allowed up to Depth 7.

- Checks: each check evasion move is extended one ply; additionally, a single reply to check gets rewarded one more ply; for the later a function that counts the number of legal moves in a given position is of course necessary
- Single reply in endgame: if the game phase has reached the endgame and there's only one legal move for the side to move, that line is extended with one ply
- Promotion threat: a pawn push to 7th rank triggers a one-ply extension; a passed pawn push to 6th rank is also one-ply extended, provided the opponent is in possession of no more than one officer
- Pawn endgame: entering a pawn endgame results in a two-ply extension

- Horizon threat: this extension comes into play on Ply_0 only, and helps deal with the horizon turbulences; aside from that, it is used only once along the current line, conditions are described below:

    - *PosVal* + *MaxGain* > *Alpha* (current line may become a new main variation)
      AND
    - *MaxLoss* > *0* (at least one hanging piece)
      AND
    - *PosVal* + *MaxGain* – *MaxLoss* < *Beta* (otherwise the threat would probably fail high)

Tree-Searching (Reductions)

Reductions (as opposite to extensions, sometimes known as "negative extensions"), describe a set of heuristics used to decrease the nominal search depth. They are powerful tools because of the extra depth owing to a slimmer search tree, but hard to master; using an inappropriate formula might cause more harm than benefit.

CP1 uses the following reductions inspired from Rebel's web page (region A only):

Winning capture reduction: a one-ply reduction is applied provided that:

- A winning capture is possible
    - AND
- Not in the last two plies before horizon (*Ply* > *2*)
    - AND
- Move ordering phase > Pawn Promotions (see chapter move generator)
    - AND
- Move does not threat an opponent piece, or does so, but will be eventually lost (SEE estimation)

Bad position reduction: a one-ply reduction is applied provided that:

- Same reduction has not been used yet along the current branch
    - AND
- Side to move not in check
    - AND
- Move does not give check
    - AND
- Not in the last two plies before horizon (*Ply* > *2*)
    - AND

- Move is not capture
    - AND
- *PosVal + Table [Ply] <= Alpha*, where Ply describes the distance to horizon and Table = {0, 0, 0, 5 x Pawn, 5 x Pawn, 7 x Pawn, 7 x Pawn, 9 x Pawn, 9 x Pawn, 15 x Pawn, 15 x Pawn, 15 x Pawn, …}

Two threatened pieces reduction: a one-ply reduction is applied provided that:

- Not in the last two plies before horizon (*Ply > 2*)
    - AND
- Move ordering phase > Pawn Promotions (see chapter move generator)
    - AND
- Not in check
    - AND
- Move is not check
    - AND
- Side to move has two hanging pieces
    - AND
- *PosVal - Val (Hang_2) <= Alpha*

To wrap up with, a node during search can undergo:

- Normal search
- One or two plies extension
- One ply reduction
- Static pruning

Evaluation (General)

To score a position **CP1** uses the following formula:

White_Score = SUM_i MatVal (WPiece_i) + PosVal (WPiece_i)
Black_Score = SUM_i MatVal (BPiece_i) + PosVal (BPiece_i)

Score = White_Score – Black_Score

where:

MatVal (WPiece_i) indicates the material value of white man "i" (e.g., Pawn = 1, Knight = 3, Bishop = 3, Rook = 5, Queen = 9, King = 18) and *PosVal (WPiece_i)* describes the positional value of white man "i". Similarly, for *MatVal (BPiece_i)* and *PosVal (BPiece_i)*.

Additionally:

PosVal (WPiece_i) = PieceValTab [WPiece_i, Square] + DynVal (WPiece_i, Square)
PosVal (BPiece_i) = PieceValTab [BPiece_i, Square] + DynVal (BPiece_i, Square)

where *PieceValTab [WPiece_i, Square]* indicates the piece value table of white piece "i" placed on a Square. In practice, this translates into 12 x 64 bytes *PieceValTab* arrays (six for white and six for black) that are calculated and filled in at the beginning of each search. The *DynVal (WPiece_i, Square)* factor is recalculated each time for the piece being moved and accounts for positional factors such as mobility, pins, center control, outposts, rooks on open files, connected rooks, pawn blockers and so on).

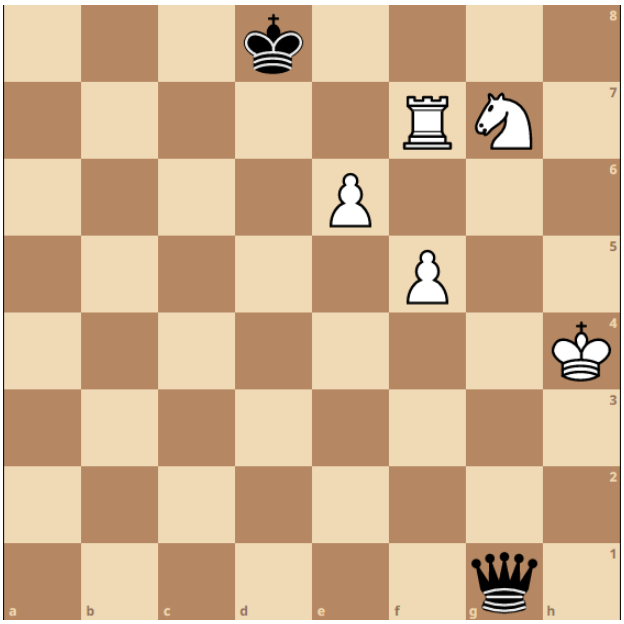

Fig.4 – score calculation example

In plain words, for example in Fig. 4:

White_Score = 18 + 5 + 3 + 1 + 1 + PosVal (WK) + PosVal (WR) + PosVal (WN) + PosVal (WP_E6) + PosVal (WP_F5)

and

Black_Score = 18 + 9 + PosVal (BK) + PosVal (BQ)

To speed up the calculation process an incremental formula is implemented, and hereby:

PosValPosition (n+1) = PosValPosition (n) + MatVal (Capture_Piece) + PosVal (Captured_Piece.Source) + PosVal (Moved_Piece.Dest) – PosVal (Moved_Piece.Source).

For example, the positional score after Qg1xg7 reads:

PosValPosition (after_move) = PosValPosition (before_move) + 5 + PosVal (WN.G7) + PosVal (BQ.G7) - PosVal (BQ.G1)

To score a position, **CP1** uses the following structure, where 1 x Pawn = 32 positional points.

```
typedef struct
{
sint8_t si8_Mtrl; /* material */
sint8_t si8_Psnl; /* positional */
}score_t
```

The *Score* variable is initialized on each ply either with *PosVal* (in Q-Search if side to move is not in check) or with maximum possible loss, that's *s_Score.si8_Mtrl = -127*, *s_Score.si8_Psnl = -127 + Depth* where *Depth* indicates the current depth (distance to root)

Evaluation (Max Depth)

To limit the search explosion, maximum search depth (*Max_Depth*) is made function of iteration as follows *Max_Depth* = {4, 6, 8, 10, 12, 12, 12, …} which means that in theory, *CP1* can detect forced lines up to mate in 6. When *Max_Depth* is reached, its *Score* is adjusted with maximum material gain (if any) for the side to move.

Evaluation (Game phases)

CP1 makes use of multiple evaluation functions, according to the game stage which is computed at the beginning of each search. The main criteria for defining the game phase are listed below.

Phase	Criterion
Opening	More than 20 pieces on initial square
Pawn endgame	No officers on board
KNB vs K endgame	One side has king, knight, and bishop whereas the other has a king only
KQ vs K endgame	One side has king and queen whereas the other side has a lonely king

KR vs K endgame	One side has king and rook whereas the other side has a lonely king
Endgame	Total number of officers less than 7
Middlegame	Otherwise

Evaluation (Positional Factors)

As explained before, the new positional value as result of *Make_Move* function, is determined by the following formula:

PosValPosition (n+1) = PosValPosition (n) + MatVal (Capture_Piece) + PosVal (Captured_Piece.Source) + PosVal (Moved_Piece.Dest) – PosVal (Moved_Piece.Source).

The *PosVal (Piece.Square)* function, employs two sub-components:

- **PVTab** factor, computed and stored in a signed char 64-byte array, at the beginning of search, for each piece type and both colors (piece square tables)
- **Dynamic** factor, calculated "on the fly", during move generation process

The main positional factors considered by **CP1** are listed below, along with the associated evaluation method.

Piece type	**Positional factors**	**Method**
Pawn	advanced	PVTab
	attack on enemy piece	dynamic
	isolated	dynamic
	doubled	dynamic
	quadrant rule (in pawn endgames)	dynamic

	passed	PVTab
	king shield	PVTab
	preserve castling pawns (when castling possible)	PVTab
Knight	advance	PVTab
	center control	dynamic
	mobility	dynamic
	attack on enemy piece	dynamic
	outpost	dynamic
	blocking enemy pawn	dynamic
	blocking d2/e2 pawns (and d7/e7 respectively)	dynamic
	enemy king tropism (in KNB vs K) endgame	dynamic
	king tropism	PVTab
	attack of squares around enemy king	PVTab
Bishop	positional (good squares)	PVTab
	center control	dynamic
	mobility	dynamic
	pinning a piece against R, Q or K	dynamic
	attack on enemy piece	dynamic
	outpost	dynamic
	blocking enemy pawn	dynamic

	blocking d2/e2 pawns (and d7/e7 respectively)	dynamic
	bonus for bishop pair	PVTab
	king tropism	PVTab
	attack of squares around enemy king	PVTab
Rook	positional (occupation of 7/8th rank, or d6/e6 squares)	PVTab
	center control	dynamic
	mobility	dynamic
	pinning a piece against Q or K	dynamic
	attack on enemy piece	dynamic
	avoid blocking friendly passed pawn	dynamic
	positional (on open / half-open files)	dynamic
	connectivity between rooks	dynamic
	keep distance from enemy king (in KR vs K endgames)	dynamic
	king tropism	PVTab
	attack of squares around enemy king	PVTab
Queen	center control	dynamic
	mobility	dynamic
	attack on enemy piece	dynamic
	king tropism	PVTab
	attack of squares around enemy king	PVTab

	avoid moving in opening	dynamic
King	proximity to passed pawns	PVTab
	guarded by friendly pawn	dynamic
	on direct opposition	dynamic
	getting closer enemy king (for winning side)	dynamic
	centering in endgame	PVTab
	averting losing corners (in case of KNB vs K) endgames	PVTab
	go castling – if pawn shield in good shape	dynamic
	avoid losing right to castle – when safe castling possible	dynamic
General	Encourage exchanges when ahead in material	dynamic
	giving check	dynamic

Draw detection

According to the rules of the game, a draw is declared if stalemate occurs, if a position is repeated three times, or if no pawn is moved and no piece is captured during 50 consecutives moves. Before starting a new search, **CP1** looks if draw conditions have been met and informs the user accordingly. It can detect draws by stalemate, insufficient material, 50 moves rule and repetition. Due to low RAM resources, repetition is detected only for do-undo move patterns such as: b1c3 f8e7 / c3b1 e7f8 / b1c3 f8e7 / c3b1 e7f8. During the search, repetition detection is qualified and draw score (*Score.Mtrl = 0*, *Score.Psnl = 0*) is return as soon as a two-fold repetition is detected (e.g. b1c3 f8e7 / c3b1 e7f8).

Toolchain

Program development relied upon the following toolchain:

Design and modeling	Office tools and UMlet 11.5
C code editor:	X32 v 3.00.01

C cross-compiler:	IAR v 2.28 A
C compiler	Dev C/C++ v5.11 (for tests purposes)
Static code analyzer:	PC- Lint v 8.00v

Basic SW architecture:

CP1 employs a task structure based on a 25ms fast task and a background loop. The 25ms fast task is responsible mainly for I/O operations whereas the background loop carries the chess engine.

Conclusions (General, Continue, Stop, Start)

This paragraph ends the brief description of **CP1** software concept; of course, looking back, I can mention traits I'm content with, and yet, aspects to improve in an eventually major release change.

Firstly, the goal of designing a low-end hardware chess engine, strong enough to outwit the mighty SuperChess 3.0 has been successfully attained; in a series of 10 games, **CP1.2** set to 3 min/move scored an undeniable victory (7 wins / 1 loss / 2 draws) against its opponent set to level 9. In another 10-game test against Colossus 4.0 it displayed an on-par performance (4 wins / 4 loses / 2 draws). On the other hand it had marginally lost against Chess Challenger 9 (6.5 to 3.5) and clearly to Mephisto Europa. This result makes very clear the name of the next mountain to climb.

So, having this said, what I like about **CP1**:

- It's selective; it achieves 5 – 6 plies in a 3 min time box
- Can find long winning combinations; this factor probably accounts for the 1856 ELO tactical performance as concluded by Colditz test
- Speed: although not extraordinary (150 - 200 nps in middle game), is more than I expected for a not extremely optimized C code (personal goal was 100 nps @8Mz)

What to improve:

- Pawn structure evaluation: needs a more effective & efficient concept; the program tends much too often to under underestimate the importance of hefty pawn formations.

- I plan to replace the arcane SEE to appraise the terminal nodes (horizon and beyond) with a clever Q-Search; relying on SEE for standing pat estimation is simply too risky in spite of the positive node reduction.

- Speed: some evaluation factors (e.g. pins, attacks, mobility) are dynamically calculated; nonetheless, I feel the benefit of this approach is merely marginal; next **CP** incarnation will employ a different strategy, most likely PV-Tab based, here; moreover there is room for code optimization along with codding the critical sections in assembly language; no surprise, it feels like speed is the key, when it comes to low-end HW chess algorithms; an extra ply may give you the upper hand.

- Time management: **CP1** has only fixed time levels (2, 5, 15, 30, 60, 120, 180, 300, 900, INFINITE seconds), a handicap when playing other programs in tournament mode.

- Opening library (now aggregating 2656 moves): needs review and updates; there are too many non-standard variations inside that often lead to ineffective positions.

- Reductions: give a try to a LMR formula, in order to achieve deeper depths (6 -7 plies in middle game under tournament time control makes it the new dream to pursue)

Concept - HW

As presented below, the HW concept revolves around the ATMega 644 microcontroller (64kb Flash / 4kb RAM) running at 8Mz. For user input, a set of four push buttons is used, whereas the output is realized by the means of a buzzer and a 1602 5V LCD type, HD44780 compatible. The whole system is powered by a 4 x 1.5V AA battery pack. There is no such thing as data exchange with the outer world, but a serial SW update mechanism (flashloader) is planned to introduce with the next major project increment.

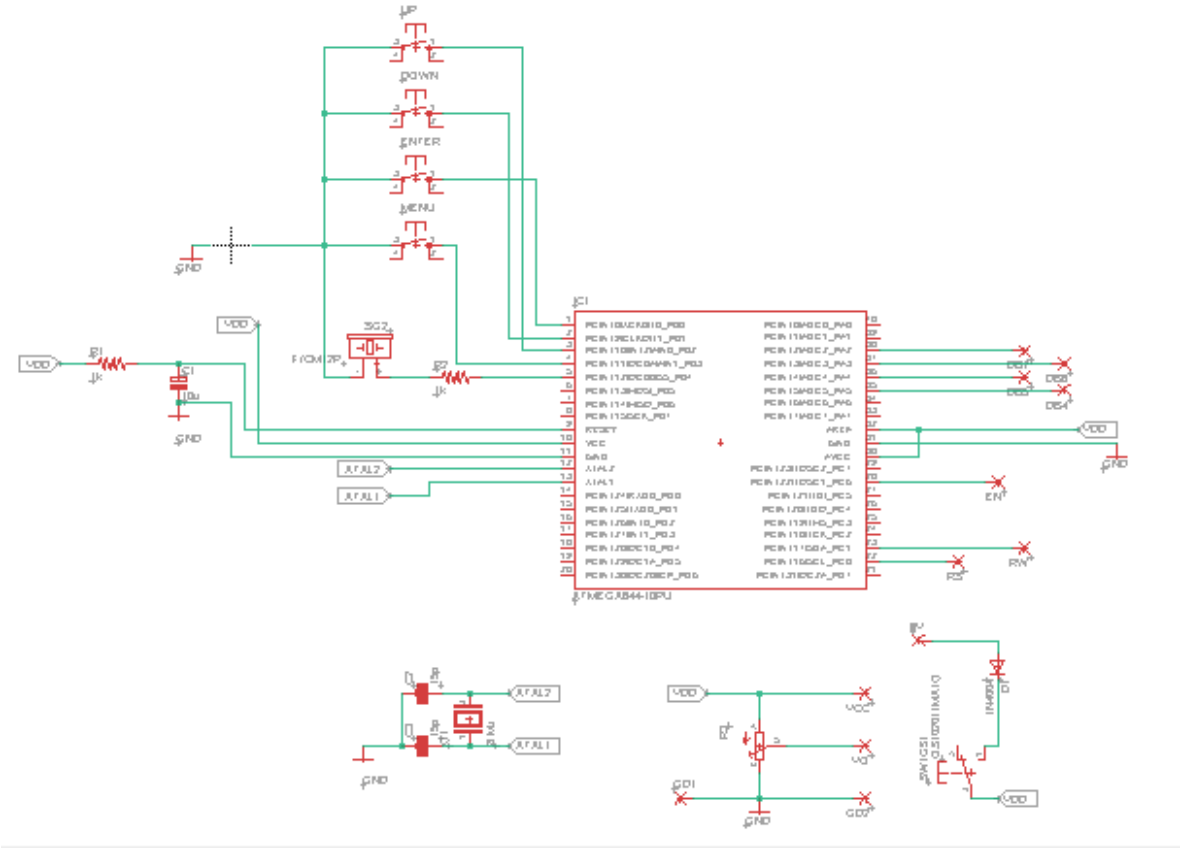

Fig.5 – HW design

To test the layout design (single layer), an A-sample prototype was designed using the "tonner & varnish remover" method (Fig. 6). The Eagle - Autodesk schematic edition and PCB design tool (v 9.6, free version) was subsequently used for design and generation of Gerber files.

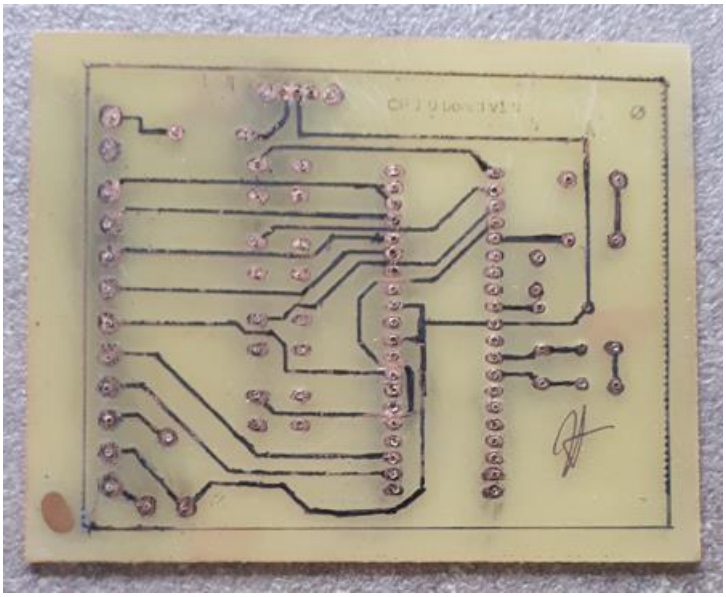

Fig.6 – A – sample layout

Then, a professional PCM maker company was hired to print the released samples.

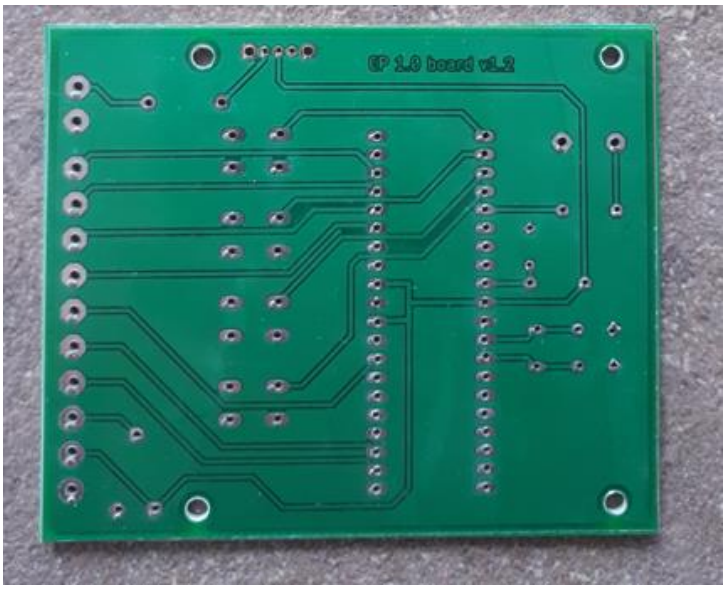

Fig.7 – B – sample layout

The whole HW assembly comes finally into one piece (actually the PCB in picture is already fit for serial communication and implements an extra push button – to drive the LCD backlight control):

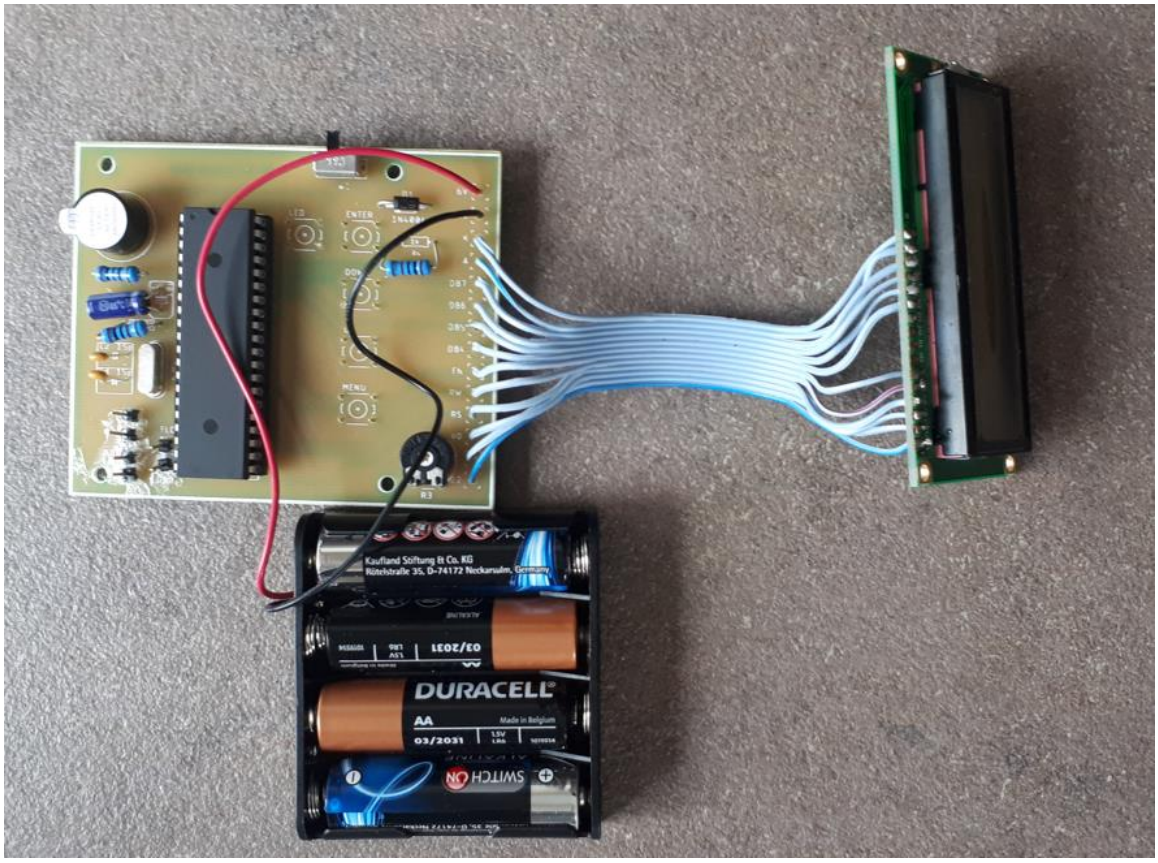

Fig.8 – Final HW assembly

Concept – ME

Initially, the whole HW assembly was merely stuffed into an aftershave metal case. Later on, a good colleague of mine, Mr. Ilie Atanasoe, mechanical engineer by profession, came up with a dedicated 3D design and successfully printed a couple of samples on his personal 3D-printer.

Fig.9 – ME concepts left – A sample, mid & right – B samples, 3D printed

User parameters:

The user input interface is made up of four push buttons (MENU, UP, DOWN, ENTER), and enables the user to configure some game parameters according to the following table:

Parameter	Meaning	Configurations
Move	Move to enter	-
Hint	Recommended countermove	-
Level	Move time (fixed)	Ten levels (0 – 9) corresponding to 2s, 5s, 15s, 30s, 60s, 180s, 300s, 900, INFINITE
Guess	Thinking on opponent's time	1 = enabled, 0 = disabled
Back	Take move back (up to three moves)	-
Swap	Swap colours	-
Verify	Print squares with chessmen	-
Analyze	Set up new position	Clear board / Square to alter
Save	Save position into EEPROM	-
Load	Load position from EEPROM	-
New	Start new game	-
About	Print about message	-

Acknowledgements:

It would be rather unfair to close **CP1** description without listing down several names that without knowing, have influenced, and inspired me in pursuit of designing a dedicated chess computer:

Gheorghe Opriciu - my uncle, who taught me the game of chess; **CP1** is dedicated to his everlasting memory

My Parents – who purchased my 1st ZX Spectrum personal computer, a very complicated undertaking in the 80s; and yes, it was a hell of financial effort behind

Viorel Darie – for his relentless efforts meant to promote the art of chess programming in Romania

Chris Whittington – for the coming up (probably) with the first really strong engine on ZX Spectrum, Superchess 3.0

Kaare Danielsen – for his didactic chess program Turbo Chess, so nicely described in the "Turbo Gameworks" handbook

Frans Morsch – for bearing out that writing excellent chess programs on very low-end hardware is possible; Mephisto Europa is such a remarkable example

Ed Schroder – for making public his knowledge about chess programming

Mr. Ilie Atanasoe – for designing and printing the mechanical parts, on his own time and costs.

E.N.D.

14.05.2022, Timisoara