

Basic book tool documentation

The following pages contain some basic control flow charts in a simplified version, just for getting some orientation. The documentation for the book tool is more extensive than for the rest of the chess application. While there are a lot of online resources on chess programming, Negascout etc., this tool is pretty much non-standard.

The book tool takes a line based opening book in ASCII text format, but UTF-8 with or without BOM (byte order mark) is also fine. This is compiled to the position based binary format, which enables the recognition of transpositions in the actual chess program.

The architecture is a simple two-pass compiler. The first pass is for checking, the second for converting the validated input file.

In the first pass, the formal correctness of the opening book is checked:

- are the characters legal?
- are the moves legal?

Besides, the raw number of moves is calculated for allocating enough memory in the second pass.

If no errors show up during the first pass, the second pass is started, without any parsing checks:

- every position is recorded with its CRC-40 along with the moves, line by line. The CRC-40 is a combination of a CRC-32 and an independent CRC-8.
- the data are sorted by CRC-40.
- transposition recognition and elimination of double moves when writing the output file.
- adding an index cache table.

The resulting binary book is an array of bytes that contains variable length records. Therefore, the random access characteristics are similar to a linked list. The additional index cache table provides entry points so that it is not necessary to scan through most of the book array.

Since the book is not used in the search tree, but only in the root position, the time saved is negligible. The point is rather a proof of concept on how to deal with the scalability of the chosen data structure.

The design to first record every position and weeding out double entries only after the CRC sorting costs more memory (though only around 2 MB) than doing the redundancy check right when recording the moves, but it scales better. While the sorting has about $O(N \log N)$ as hardest step, checking every move when recording would have $O(N^2)$.

Rasmus Althoff, April 2018

Format of the opening book

The opening book text file itself is an ASCII text file that you can edit with any text editor. Besides plain ASCII, also UTF-8 with and without BOM (byte order mark) are supported.

Allowed are:

- leading whitespaces in a line (whitespaces may be space characters or tabs).
- more than one whitespace between two moves.
- lines without moves (empty or whitespace only or comments).

It is line based, and each line goes like this:

`e2e4 e7e5 g1f3 b8c6 f1b5` (Spanish Opening)

You always give the origin and destination square of a move. Capturing moves do not require any special markup, not even for en passant. Castling is done as a pure king move. A white kingside castling thus is "e1g1". The rook relocation is implied automatically.

En passant captures work although the en passant square is not being tracked in the position CRC itself. Within the CT800's opening book logics, the suggested opening book moves are matched against the list of legal moves. So if en passant is in the book for a certain position CRC without being possible on the board, this move will not be selected.

Pawn promotions to queens work because the queen promotion is assumed implicitly. Also for a promotion pawn move, is it just from-to notation. Attaching the promotion piece to the move is not allowed. E.g. a black pawn on c2 that captures a white piece on b1 and is promoted to a queen is noted as "c2b1". Writing it as "c2b1q" will yield a syntax error.

Underpromotions are not possible in the opening book.

You can mark a move with a following '?' so that the program passively knows how to continue if the opponent plays that move, but it will never play that move by itself.

Another marker with the same consequence is 'x' instead of '?', which is used to mark moves that actually are playable, but avoided because the line results in positions that don't suit the software. The 'x' marker might be used for some kind of "wide" opening book in the future, though that is not planned to be implemented as of now.

E.g. if you don't want the program to play King's Gambit, but tell it how to react in case the opponent plays it:

`e2e4 e7e5 f2f4? ...`

Comments at the line end must start with '('. The rest of the line will be ignored, so the closing ')' bracket is just cosmetics. If a line contains only a comment, you can use '#' as first character.

The opening book compiler transforms the line based format into a position based one. This means that once you reach a position that already has been treated at the end of another variation, the same continuation applies.

Example:

```
d2d4 g7g6 e2e4 f8g7 ...  
e2e4 g7g6 d2d4
```

In the second line, you don't need to continue because the position that is reached after the e2e4 in the first line is the same as after d2d4 in the second line. So if the variation of the second line happens, the following move Bf8-g7 will apply automatically also in the second line.

Software architecture

The book tool comprises the following parts:

module	func. prefix	purpose
main.c	-	central coordination logic and resource management
booktool.h	-	definitions of data types, constants and macros
check.c / .h	Check_	first pass (validation of the book input file)
convert.c / .h	Conv_	second pass (turns the text book input file into the binary book output file)
util.c / .h	Util_	utility functions non-prefixed wrapper functions for more robust resource management (memory, files)

Scalability

The tested regular opening book was around 600 kB text: 12878 lines, 116995 moves, 109905 raw positions. It took 0.139 user seconds to process that.

Concatenating this file about a 1,000 times into a much bigger file of 600 MB text gave the following numbers: 13024791 lines, 118505009 moves, 111325514 raw positions. It took 98.4 user seconds to process that.

While the factor in the amount of data is about 1000, the time only grew by a factor of 708. Sublinear scaling seems strange, but with the 600 kB file, the constant runtime overhead (program start, file opening) is likely to dominate.

Especially the Shellsort scaled surprisingly well with such a huge list. Of course, the result file was of the same size because the redundancy was eliminated, so that skews the result a bit – but not much because the temporary file actually was that much bigger.

Conclusion: the scaling can be considered robust even with considerably more data than the intended usage scenarios will involve.

CRC-40 collisions

The regular opening book has about 12,000 unique positions. These are identified by a combined^(*) CRC-40 that consists of a CRC-32 and an independent^(**) CRC-8.

The probability that two different positions accidentally share the same CRC-40 can be calculated similarly to the "birthday paradoxon". The resulting probability for a CRC-40 collision is 0.006%.

Besides, the moves are not taken directly from the opening book. Instead, the list of legal moves for the board position is matched against the suggested moves from the opening book. Therefore, the opening book function will never suggest illegal moves even in case of CRC-40 collisions.

The probability of a position collision where the colliding position adds unintended, but legal moves to the board position is vanishingly small, and even then they still would have to actually be selected by the randomness function.

Notes:

(*) combined: in the actual code, the implementation pattern of "*bit spreading*" can be observed. This is also prevalent in other checksum related parts of the CT800 firmware, most notably in the hashtables. The main advantage is saving memory to get as much out of the device as possible.

In a data structure composed of data fields, there are often unused parts. For example, a byte value that only ranges from 0-15, leaving the upper four bits unused. A secondary checksum may be spread over several of these unused areas.

When verifying a match, the primary checksum is verified (here: the CRC-32), and only if that matches, the secondary checksum (here: the CRC-8) is re-assembled and compared for a full match.

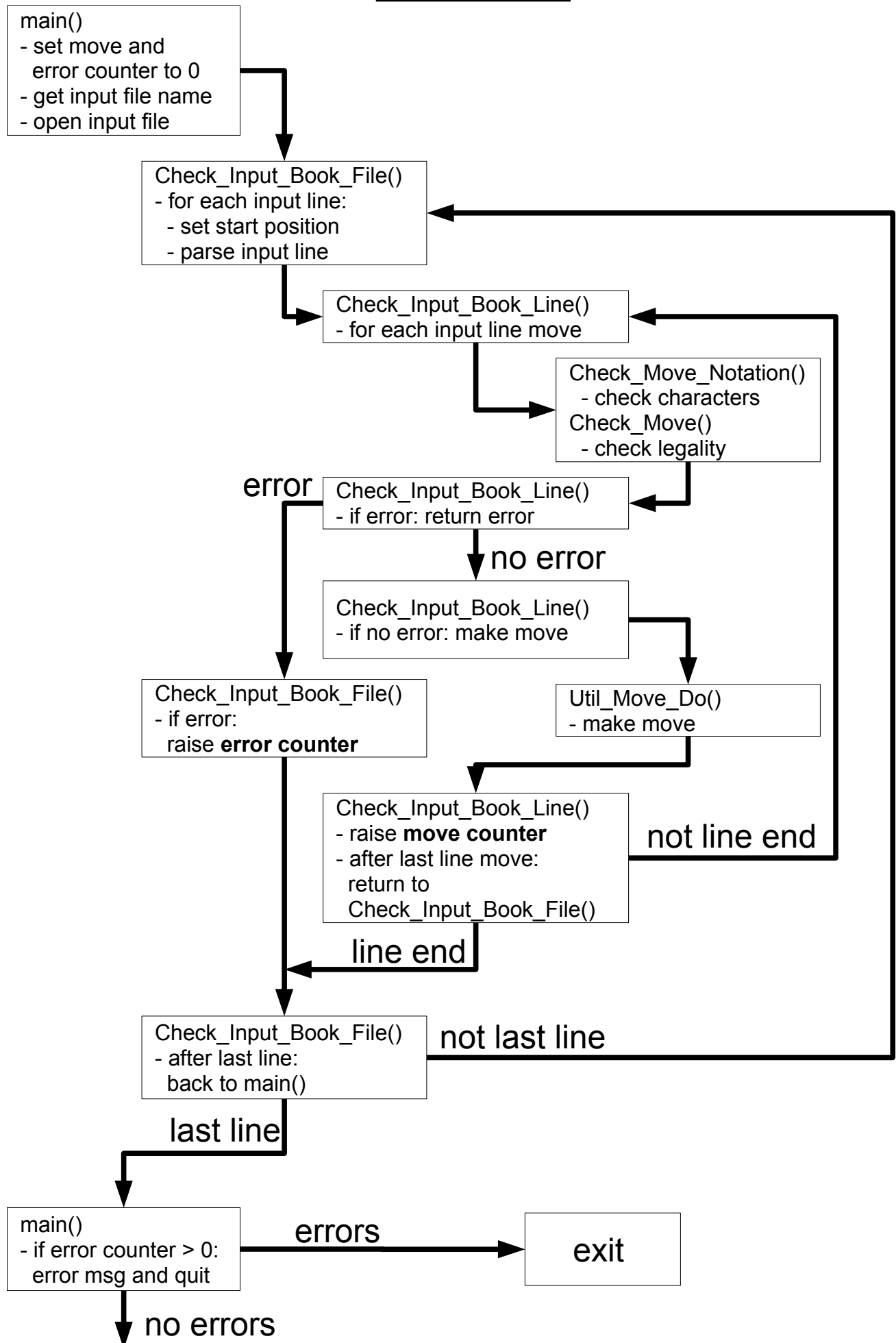
(**) independent: the mathematical background is dividing the data content by the CRC – not as integer division, but as polynomial division over GF(2). The remainder of the division is the CRC checksum.

The checksums are independent from each other if the CRC-32 and the CRC-8 polynomials are prime to each other. This is trivially fulfilled in this case because both implemented polynomials are irreducible over GF(2), i.e. they are prime.

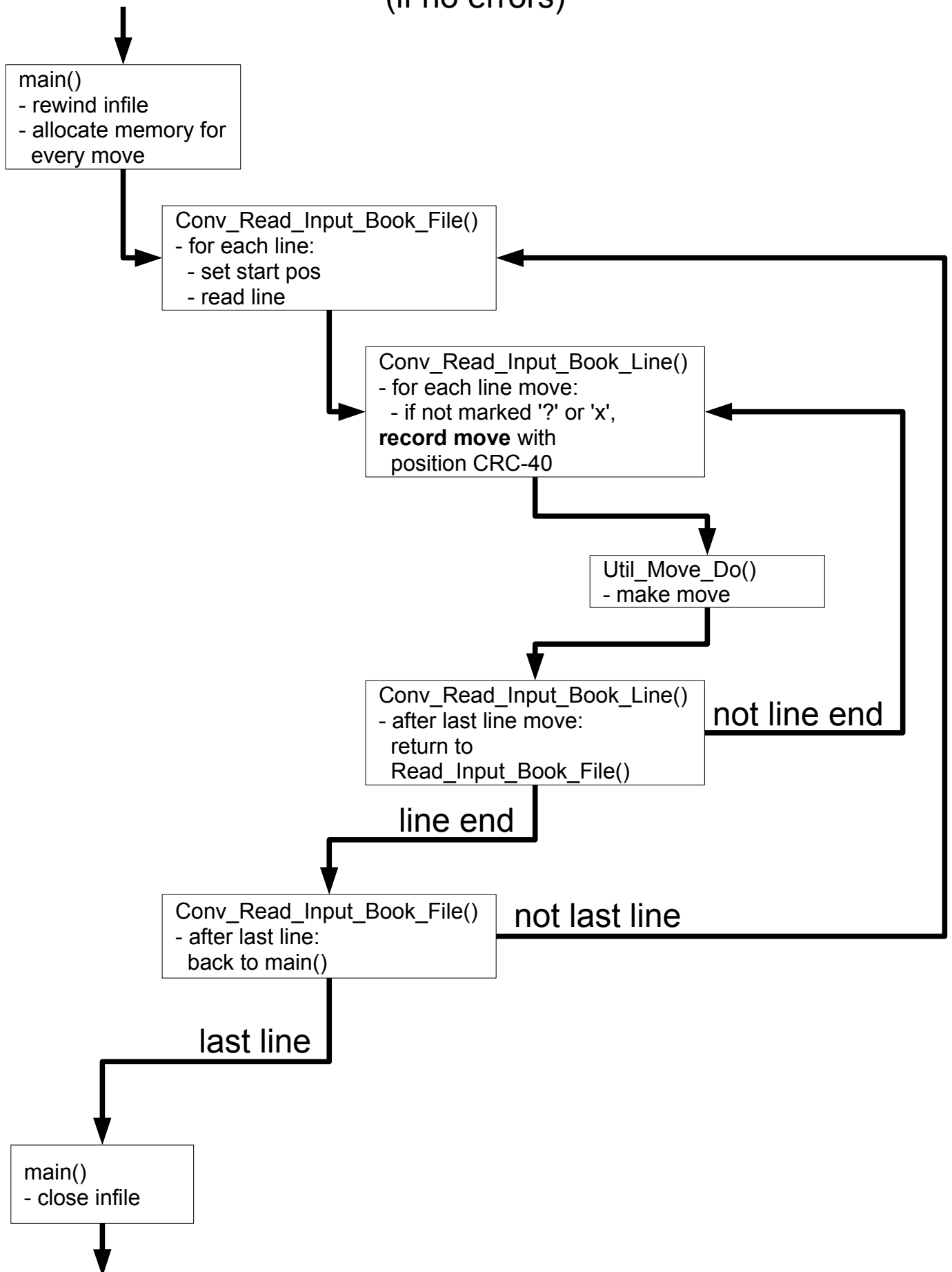
CRC-32 is the well-known Ethernet-CRC 0xEDB88320 in reversed notation, de-reversed 0x04C11DB7. The corresponding polynomial is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$.

CRC-8 is 0xB2 in reversed notation, de-reversed 0x4D. The corresponding polynomial is $x^8 + x^6 + x^3 + x^2 + 1$. For the reason of choice, see the academic paper "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks" by Philip Koopman and Tridib Chakravarty, where it figures as 0xA6 in Koopman notation.

First Pass



Second Pass (if no errors)



Second Pass (continued)

