UNIVERSAL CHESS INTERFACE

SEPTEMBER 2024 · DRAFT

Introduction

This specification governs the interaction between two processes named the *client* and the *engine*. ※ Graphical interfaces, terminal emulators, and scripts and utilities are examples of clients. The text of sections 1–4 is normative (except as described in *Conventions* below).

Getting Started

This section needs to be written! It should contain enough for a beginner to add basic UCI support to their engine.

Conventions

Sequences are written with angle brackets; for example, ⟨0, 1, 2, …⟩. An x preceding a number indicates that it is written in hexadecimal; for example, x10 = 16. Sequences of bytes encoded in ASCII are set in teal; for example, uci = ⟨x75, x63, x69⟩. Sequences of tokens may be written without commas when they contain token literals; for example, ⟨ab c *d* e⟩ rather than ⟨ab, c, *d*, e⟩.

Special terms defined by the specification are set in purple when they are first introduced and inline comments are set ※ after a reference mark.

> A blue box is used to describe a convention that clients and engines are encouraged to follow, usually to do with the interpretation or meaning of the messages that clients and engines send. A blue box is also used to provide a recommendation, usually to do with implementation-defined behavior.

> A grey box is used to provide an explanative note or comment.

> A green box is used to provide an example of conforming behavior.

Text within colored boxes is nonnormative.

1 Definitions

1·1 A violation is any violation, by the client or engine, of the requirements of the specification. When a violation occurs, or when the requirements of the specification are otherwise not met, the specification imposes no further requirements on the behavior of the client or engine.

1·2 The engine's standard input and output must be open file descriptors and the engine's standard error must be an open file descriptor until closed by the engine. The sequence of bytes that the client sends to the engine via the engine's standard input is the client stream. The sequence of bytes that the engine sends to the client via the engine's standard output is the engine stream.

> There are no requirements for an engine's standard error except that it be open. For example, the engine's standard error may be directed to a null device, to the client's standard output or standard error, or to a log file. There are accordingly no restrictions on the sequence of bytes that the engine writes to its standard error.

> No particular encoding is specified: the client and engine streams are not required to be encoded sequences of Unicode scalar values (or code points of any other character set). The specification instead governs the client and engine streams as sequences of bytes *per se*.
>
> However, clients and engines are recommended to use UTF-8 for client–engine communication so that engine, author, and option names are displayed properly. Note that clients and engines may use different encodings for other interfaces, such as for file system paths that are passed as arguments to the operating system. For example, the client and engine may be communicating over a network connexion and the client may be running on Linux (where paths are arbitrary byte sequences that do not contain `x00` or `x2f`) but the engine may be running on Windows (where NTFS is a common filesystem, which stores file names in UTF-16, but where older libraries or APIs may expect paths encoded in the local code page), and so the path of a tablebase file may require transcoding, implicitly or explicitly.

1·3 A message terminator is the pair ⟨`x0d`, `x0a`⟩ or `x0a` alone when it is not preceded by `x0d`.

1·4 Message terminators divide the client and engine streams into messages; that is, a message is a (possibly empty) sequence of bytes that does not contain a message terminator, and every byte of the client and engine streams is either part of a message or part of a message terminator. The messages within the client stream are client messages and the messages within the engine stream are engine messages.

> This precludes the proper use of some encodings for client–engine communication, such as UTF-16 or UTF-32 (since in UTF-16 and UTF-32, the byte `x0a` appears in the encodings of various scalar values, whereas in UTF-8, the byte `x0a` only appears in the encoding for U+000A LINE FEED).

> Note that `x00` is not disallowed.

1·5 The byte `x20` divides messages into tokens, that is, a token is a non-empty sequence of bytes that are not `x20`, and every byte of a message is either part of a token or is `x20`.

> In some cases, only the beginning of a message will be viewed as a collection of tokens and the remainder will be viewed as a contiguous sequence of bytes.

1·6 For a given sequence of tokens that begin a message (the prefix), the suffix is the contiguous sequence of bytes starting with the first byte that is not `x20` after the prefix and ending with the last byte of the message.

1·7 A sequence of bytes that does not contain any tokens (that is, an empty sequence or a sequence of one or more repetitions of `x20`) is void.

1·8 A position is a tuple with the following fields:

An 8×8 array of elements, each of which is either NONE or a color–kind pair (where the color is white or black and the kind is king, queen, rook, bishop, knight, or pawn). This array is called the board and is indexed along one axis by the letters “a” through “h” inclusive and along the other axis by the numerals “1” through “8” inclusive. A color–kind pair is called a piece.

A color, white or black, called the side to move.

A collection of values, called the rights, which may be empty or include one or more of the following: the white kingside castling right, the

white queenside castling right, the black kingside castling right, and the black queenside castling right.

A value called the en passant target, which is either NONE or one of a3, b3, …, h3, or a6, b6, …, h6.

A nonnegative integer called the depth from zeroing. ※ This is the number of moves that have been played (none of which are captures or pawn moves) since the last capture or pawn move, measured in ply.

The side waiting of a position is the opposite color of the side to move.

1·9 For a given position, a king is in check if it is attacked by one or more pieces of the opposite color, where "attacked" is defined analogously to article 3.1.2 of the 2023 FIDE Laws of Chess.

1·10 A position is valid if the following conditions are all satisfied:

The board contains exactly one white king and one black king.

The board does not contain any pawns at indices a1, b1, …, h1 and does not contain any pawns at indices a8, b8, …, h8.

If the rights include the white kingside castling right, the white king is at index e1 and there is a white rook at index h1. If the rights include the white queenside castling right, the white king is at index e1 and there is a white rook at index a1. If the rights include the black kingside castling right, the black king is at index e8 and there is a black rook at index h8. If the rights include the black queenside castling right, the black king is at index e8 and there is a black rook at index a8.

If the en passant target is *n*3 for some *n*, then the side to move is black, there is a white pawn at index *n*4, and at *n*2 and *n*3 the board is NONE. If the en passant target is *n*6 for some *n*, then the side to move is white, there is a black pawn at index *n*5, and at *n*6 and *n*7 the board is NONE.

The king of the color of the side waiting is not in check.

The depth from zeroing is 100 or less.

There is at least one valid move that can be applied (as described in 1·12 below). ※ This means the position is not checkmate or stalemate.

> A valid position (as defined above) need not be reachable from the starting position.
>
> Some engines designed for gameplay rather than analysis may additionally require that the positions they are sent must be reachable from the starting position. Such constraints are specific to engines, and are not constraints of the Universal Chess Interface.

> The 2023 FIDE Laws of Chess state that the game ends immediately when a player cannot checkmate the king by any series of legal moves, but this condition does not cause a position to not be valid (as defined above).

> Engines are recommended to accept positions that are checkmate or stalemate even though they are not required to handle receiving such positions. For such positions, engines should report null as the best move (see 1·18 below).

1·11 A move is a tuple of three values: a board index called the source, a board index called the destination, and a field called the promotion kind that is either NONE or queen, rook, bishop, or knight.

1·12 For a given position *P*, a move *M* is valid and the position *Q* immediately follows when *M* is applied if *P*, *M*, and *Q* fulfill requirements analogous to articles 3.1 through 3.9 inclusive of the 2023 FIDE Laws of Chess, except

- the requirement for a pawn to advance by two is instead that the pawn is white and its index is one of a2, b2, …, h2 or that the pawn is black and its index is one of a7, b7, …, h7, and
- the requirement for an en passant capture is instead that the en passant target is not none and the capturing pawn attacks the en passant target.

In particular, when *M* is valid, there is a piece of the color of the side to move in the board of *P* at the source of *M* and there is a piece of the same color and kind (or a piece of the same color and the promotion kind of *M*) in the board of *Q* at the destination of *M*.

The side to move of *Q* is the opposite color of the the side to move of *P*. If *M* is a king or rook move, the rights of *Q* do not include the corresponding castling right or castling rights. If *M* advances a pawn by two, the en

passant target of Q is the index between the source and destination of M. If M is a capture or a pawn move, the depth from zeroing of Q is zero; otherwise, the depth from zeroing of Q is the depth from zeroing of P plus one.

1·13 A Forsyth–Edwards Notation (FEN) record is a sequence of six tokens ⟨*board*, *side to move*, *rights*, *EP target*, *DFZ*, *move number*⟩ of the form

```
          board = row / row / row / row / row / row / row / row
            row = 8 | [1–7KQRBNPkqrbnp]+
side to move = w | b
          rights = - | K?Q?k?q?
      EP target = - | [a–h][36]
            DFZ = 0 | [1–9][0–9]? | 100
    move number = [1–9][0–9]{0,3}
```

※ The dash "-" is x2d.

with the following additional constraints:

> In each *row*, numerals must not be adjacent; that is, there must not be two or more immediately consecutive numerals.
>
> For each *row*, map K, Q, …, p to 1, map 1, 2, …, 8 to 1, 2, …, 8, and then define the width as the sum of these numbers. For each *row*, the width must be 8.

1·14 An FEN record describes a position if it maps to the position in a manner analogous to that described in article 16.1.3 of the Portable Game Notation Specification.

1·15 The starting position is the position described by the FEN record `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1`.

1·16 A move token is a token of the form

```
[a–h][1–8][a–h][1–8][qrbn]?
```

The first and second bytes of a move token interpreted in the natural way as a board index is the move token's source, the third and fourth bytes of a move token interpreted in the natural way as a board index is the move token's destination, and the fifth byte (if present) interpreted in the natural way as a kind is the promotion kind. If the fifth byte is not present, the promotion kind of the move token is NONE.

1·17 For a given position *P*, a move token *T* denotes a move *M* if *M* is valid for *P* and the source, destination, and promotion kind of *T* match the source, destination, and promotion kind of *M* (respectively).

1·18 A null token is a token of the form `0000`.

2 Client Messages

2·1 A client message has one of nine types indicated by the first token of the message (`uci`, `debug`, `setoption`, `ucinewgame`, `position`, `isready`, `go`, `stop`, and `quit`).

2·2 A `uci`, `ucinewgame`, `isready`, `stop`, or `quit` message is well-formed if it contains exactly one token.

2·3 A `debug` message is well-formed if it contains exactly two tokens, either ⟨`debug on`⟩ or ⟨`debug off`⟩.

> If engines have debugging information to expose via the client (that is, through standard output rather than standard error), they should feely send `info string` messages with this information after receiving `debug on` and suppress these messages after receiving `debug off`.

2·4 A `setoption` message is well-formed if it has a prefix of the form ⟨`setoption name` *tok*⁺ `value`⟩, where *tok*⁺ is one or more tokens (none of which are `value`) matching an option name listed in an `option` message enqueued by the engine, and a suffix satisfying the condition(s) for the corresponding option type:

- If `check`, the suffix contains exactly one token, either `true` or `false`.
- If `spin`, the suffix contains a single token which is a decimal integer, either `0` or of the form `-`?[`1`–`9`][`0`–`9`]*, not less than the minimum value nor greater than the maximum value specified by the engine.
- If `combo`, the suffix contains one or more tokens and the sequence of tokens in the suffix is one of the sequences specified by the engine.
- If `string`, the suffix is not void. The suffix `<empty>` indicates an empty value. ※ The value `<empty>` therefore cannot be specified.

A `setoption` message is also well-formed if it is of the form ⟨`setoption name` *tok*⁺⟩ where *tok*⁺ is one or more tokens (none of which are `value`) matching an option name listed in an `option` message enqueued by the engine for which the corresponding option type is `button`.

2·5 A `position` message is well-formed if it has the form

⟨`position` (`startpos` | `fen` *fen*⁶) (`moves` *mov*⁺)?⟩

where *fen*⁶ is an FEN record and *mov*⁺ is one or more move tokens. If `fen` is specified, the position described by *fen*⁶ must be valid. Let P_0 be the starting position if `startpos` is specified and the position described by *fen*⁶ if `fen` is specified. If `moves` is specified:

> Let N be the number of move tokens specified.
>
> For n from 0 to N–1 inclusive, the n-th token T_n of *mov*⁺ must denote a move M_n from P_n. Then let P_{n+1} be the position that immediately follows from P_n when M_n is applied.
>
> The position P_N must be valid.

The final position (P_N if `moves` is specified, otherwise P_0) is the position described by the `position` message.

2·6 A `go` message is well-formed if…

3 Engine Messages

An engine message has one of six types indicated by the first token of the message (`id`, `option`, `uciok`, `info`, `readyok`, and `bestmove`).

4 States and Transitions

4·1 The client and the engine both may always enqueue or dequeue a void message. When the client or engine dequeues a void message, it must ignore the message, behaving as if the message were not transmitted.

The remainder of section 4 is written without mention of void messages; consideration for void messages is tacit.

4·2 Once the engine dequeues a `uci` message from the client stream, at every point in time the engine is in one of six states. The engine begins in the initial state. When the engine dequeues a client message or enqueues an engine message, the state may change; this is a transition.

> There is a particular invariant that transition rules should satisfy due to the nature of standard input and standard output.
>
> Firstly, the moment a message is read necessarily occurs some duration after the moment that the message is written (that is, the sender does not directly manipulate the memory of the recipient). Secondly, a process cannot both read from a file descriptor and write to a file descriptor as an atomic transaction (that is, "time of check to time of use" race conditions cannot be prevented).
>
> In particular, it is impossible for a process to guarantee that a write to file descriptor F is not preceded by a write to file descriptor G by another process. Even if the process attempts to read G, finds it empty, and immediately writes to F, it is possible for an intervening write to G to occur before the write to F occurs.
>
> As a result, if a transition from state A to state B occurs when the engine sends a message, it is impossible to guarantee that a message sent by the client in state A can actually be read by the engine in state A (and not in state B). Therefore, any message that the client is allowed to send in state A should be allowed in state B. The same principle holds vice versa, and so we can state the invariant more generally as follows:
>
> *For distinct processes 1 and 2, for all pairs of states A and B, if a transition from A to B can be caused by process 1 sending a message, every messages that process 2 is allowed to send in state A must also be allowed in state B.*

4·3 In some states, the engine may dequeue a message. If the engine dequeues a message, the message must be well-formed and have an allowed message type given the state of the engine.

> This is not meant to imply that standard input is ever required to be empty, but meant to indicate that in some states the engine is not allowed to read standard input (or at least must behave as if it has not read standard input).

4·4 In all states, the engine may enqueue a message. If the engine enqueues a message, the message must be well-formed and have an allowed message type given the state of the engine.

4·5 The allowed message types that may be dequeued and enqueued in each state are enumerated in the following table. If the NEXT column is not empty for a row, the action listed in the row (dequeuing or enqueuing a message of a particular type) causes a transition to the listed state.

STATE	ALLOWED		NEXT
initial	ENQUEUE	`id`	
		`option`	
		`uciok`	idle
idle	DEQUEUE	`debug`	
		`setoption`	
		`ucinewgame`	
		`position`	
		`isready`	sync
		`go`	active
		`stop`	
	ENQUEUE	`info`	
sync	ENQUEUE	`info`	
		`readyok`	idle
active	DEQUEUE	`debug`	
		`isready`	ping
		`stop`	halt
	ENQUEUE	`info`	
		`bestmove`	idle
ping	ENQUEUE	`info`	
		`readyok`	active
halt	ENQUEUE	`info`	
		`bestmove`	idle

> Engines and clients are not required to ignore (or otherwise handle) unknown or unexpected tokens (these are simply violations).

4·6 The client must enqueue a `ucinewgame` message at least once before before it enqueues a `go` message for the first time.

> Engines are strongly recommended to behave identically whether or not a `ucinewgame` message is ever sent.

4·7 The client must enqueue a `position` message at least once before it enqueues a `go` message for the first time.

> Engines are strongly recommended, if a `position` message is not sent before the first `go` message, to behave as if `position startpos` were sent.

> Note that the client is not required to send `isready` before `go`.

> Note that the client may send `ucinewgame` after `position` before `go` without sending any additional intervening `position` message.

> Note that `ucinewgame` does not reset the current board state nor does it reset the move history. (Only `position` changes the current board state and move history.)

4·8 The specification ceases to govern the interaction between the client and the engine when the client enqueues a well-formed `quit` message.

> Engines are recommended to stop searching immediately and exit. Clients are recommended, if engines are their subprocesses, to provide a grace period before terminating engines.

5 Examples

```
let read byte! : File Descriptor → Byte | EOF

def withoutCR(msg : List(Byte)) : List(Byte)
  if empty?(msg) or last(msg) ≠ x0d then return msg
  return withoutLast(msg)

def read message!(mut fd : File Descriptor) : List(Byte) | EOF
  msg ← empty
  repeat
    match read byte!(fd)
      eof ⇒ return (if empty?(msg) then eof else msg)
      x0a ⇒ return withoutCR(msg)
      val ⇒ append!(msg, val)

def get token!(mut msg : List(Byte)) : List(Byte) | None
  repeat
    if empty?(msg) then return none
    if first(msg) ≠ x20 then break
    removeFirst!(msg)
  tok ← empty
  repeat
    append!(tok, first(msg))
    removeFirst!(msg)
    if empty?(msg) then return tok
    if first(msg) = x20 then break
  return tok
```