UNIVERSAL CHESS INTERFACE

2022 DEC 29 · DRAFT

1 Introduction

1·1 This specification governs the interaction between two processes named the client and the engine. Graphical interfaces, terminals, and scripts or utilities are examples of clients.

1·2 The engine's stdin, stdout, and stderr must all be open file descriptors.

1·3 The sequence of bytes that the client sends to the engine (via the engine's stdin) must be valid UTF-8, and the Unicode scalar values that the client sends must be U+000A LINE FEED, U+000D CARRIAGE RETURN, or in the range U+0020 to U+007E inclusive (in other words, printable ASCII characters and space). When the scalar value U+000D appears it must be immediately followed by U+000A.

1·4 The sequence of bytes that the engine sends to the client (via the engine's stdout) must be valid UTF-8. Non-ASCII characters are permitted so that the engine name, author names, and info messages can be represented. When the scalar value U+000D appears it must be immediately followed by U+000A.

1·5 No requirements are placed upon the engine's stderr (for example, the engine's stderr may be directed to a null file, to the client's stdout or stderr, or to a log file), and there are no restrictions on the sequence of bytes that the engine writes to its stderr.

1·6 When the client or engine violate the requirements of this specification, a violation is said to have occurred. When a violation occurs, or when the requirements of this specification are otherwise not met, this specification imposes no further requirements on the behavior of the client or engine.

1·7 There are conditions that resemble violations but that the client and engine are expected to handle; these are called errors.

2 Other Definitions

2·1 A message is a sequence consisting of

- zero or more scalar values that are not U+000A

followed by

- the scalar value U+000A.

2·2 The scalar value U+000A (or, if the scalar value U+000A is preceded by the scalar value U+000D, the sequence U+000D U+000A) is the message termination. Note that both client and engine are required to handle both Unix- and Windows-style line endings.

2·3 The sequence of scalar values in a message before the message termination is the message body.

2·4 A token is a maximally contiguous sequence of scalar values that are not U+0020 SPACE in the message body. Tokens are separated by whitespace. Whitespace is any contiguous sequence of U+0020 in the message body.

2·5 An algebraic token is a token of the form `[a–h][1–8][a–h][1–8][qrbn]?`.

2·6 A message whose body does not contain any tokens is empty.

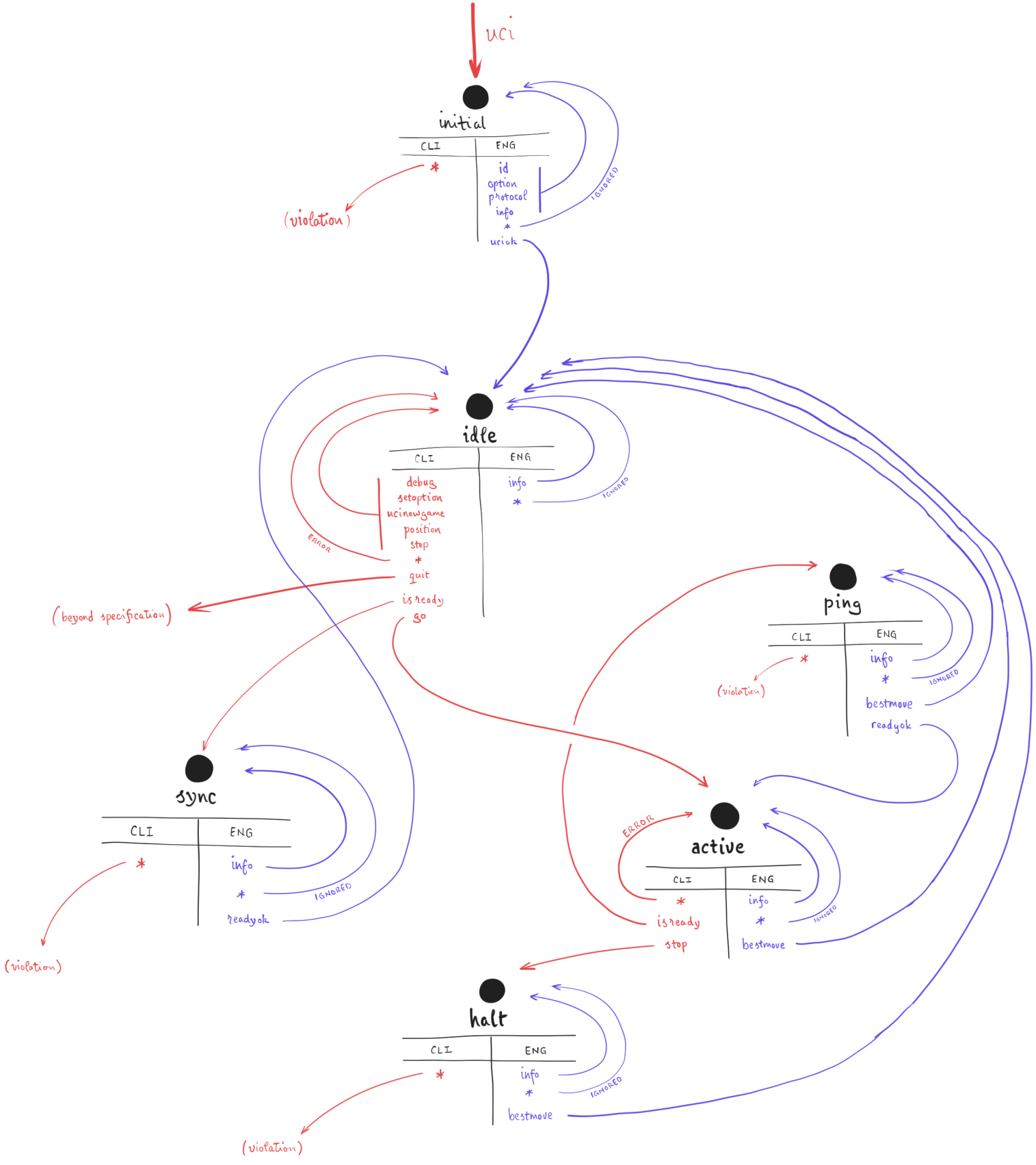
2·7 The first token of a message from the client to the engine is a command. For any token *cmd*, a message whose command is *cmd* is a *cmd* message.

2·8 The first token of a message from the engine to the client is a remark. For any token *ɾmk*, a message whose remark is *ɾmk* is an *rmk* message.

3 States and Transitions

3·1 At every point in time, given the history of messages between the client and engine, there is a corresponding state. When the client or engine sends a message (that is, writes a message to the engine's stdin or stdout), the state may change; this is a transition. Note that transitions occur when messages are written, not read.

3·2 Empty messages may always be sent and must be ignored by the client and engine. Empty messages never cause a transition.


uci
initial
CLI
ENG
*
id
option
protocol
info
*
ucioh
IGNORED
(violation)
idle
CLI
ENG
debug
setoption
ucinewgame
position
stop
*
quit
isready
go
info
*
IGNORED
ERROR
(beyond specification)
ping
CLI
ENG
*
info
*
bestmove
readyok
IGNORED
(violation)
sync
CLI
ENG
*
info
*
readyok
IGNORED
(violation)
active
CLI
ENG
*
isready
stop
info
*
bestmove
IGNORED
ERROR
halt
CLI
ENG
*
info
*
bestmove
IGNORED
(violation)

3·3 This specification begins to govern the interaction between the client and the engine when the client sends a well-formed uci message.

3·4 In the initial state, the client may not send a nonempty message to the engine. The engine may send any message. If the engine sends a message that is not a well-formed id, option, protocol, info, or uciok message, the message must be ignored by the client. When the engine sends a well-formed uciok message, the state changes to idle.

3·5 In the idle state, the client may send any message. If the client sends a nonempty message that is not a well-formed setoption, ucinewgame, isready, position, go, stop, debug, or quit message, an error has occurred, and the engine may respond by sending an info message indicating that a client error has occurred; besides sending an info message, the engine must ignore the client message (in other words, the engine must behave as if no message was sent). When the client sends a well-formed isready message, the state changes to sync. When the client sends a well-formed go message, the state changes to active. Note that the engine cannot assume that a go message will be preceded by an isready message.

3·6 In the idle state, the engine may send any message. If the engine sends a message that is not a well-formed info message, the message must be ignored by the client. In the idle state, the engine should be asleep or minimally utilize processing resources unless, for example, it is reconfiguring itself in response to a setoption message.

3·7 In the sync state, the client may not send a nonempty message to the engine. The engine may send any message. If the engine sends a message that is not a well-formed info or readyok message, the message must be ignored by the client. When the engine sends a well-formed readyok message, the state changes to idle.

3·8 In the active state, the client may send any message. If the client sends a nonempty message that is not a well-formed isready or stop message, an error has occurred, and the engine may respond by sending an info message indicating that a client error has occurred; besides sending an info message, the engine must ignore the client message. When the

client sends a well-formed isready message, the state changes to ping. When the client sends a well-formed stop message, the state changes to halt.

3·9 In the active state, the engine may send any message. If the engine sends a message that is not a well-formed info or bestmove message, the message must be ignored by the client. When the engine sends a well-formed bestmove message, the state changes to idle.

3·10 In the ping state, the client may not send a nonempty message to the engine. The engine may send any message. If the engine sends a message that is not a well-formed info, readyok, or bestmove message, the message must be ignored by the client. When the engine sends a well-formed readyok message, the state changes to active. When the engine sends a well-formed bestmove message, the state changes to idle.

3·11 In the halt state, the client may not send a nonempty message to the engine. If the engine sends a message that is not a well-formed info or bestmove message, the message must be ignored by the client. When the engine sends a well-formed bestmove message, the state changes to idle.

3·12 This specification stops governing the interaction between the client and the engine when the client sends a well-formed quit message in the idle state. Engines are recommended to shut down and terminate themselves. Clients are recommended to provide a grace period not less than 5 seconds before killing the engine.

There is an interesting invariant that the state and transition rules should satisfy due to the nature of stdin and stdout.

The time that a message is read and processed necessarily occurs some duration after the time that the message is written (*sending a message* is merely *enqueuing*, not invoking a subroutine of the recipient's) and it impossible to atomically perform both a read and a write. More particularly, it is impossible for a process to guarantee that a write to stream F is not preceded by a write to stream G by some other 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 the transition from state A to state B is caused by the client sending a message, it is impossible to guarantee that a message sent by the engine in state A can actually be read by the client in state A (and not in state B). Therefore, any message that the engine is allowed to send in state A should be allowed in state B. The same idea holds for transitions caused by the engine sending a message, so we can state the invariant 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 Timeouts

4·1 The time between the client sending a well-formed uci message and the client reading a well-formed uciok message may not exceed an implementation-defined duration. For example, this may be caused by transmission latency, by the client or engine failing to read data or flush their writes, or by the engine performing excessive operations at startup. This duration (the initialization timeout) must not be less than 5 seconds.

4·2 The time between the client sending a well-formed isready message in the idle state and the client reading a well-formed readyok message may not exceed an implementation-defined duration. This duration (the reconfiguration timeout) must not be less than 5 seconds.

4·3 The time between the client sending a well-formed isready message in the active state and the client reading a well-formed readyok message may not exceed an implementation-defined duration. This duration (the ping timeout) must not be less than 1 second.

4·4 The time between the client sending a well-formed stop message in the active state and the client reading a well-formed bestmove message may

not exceed an implementation-defined duration. This duration (the halt timeout) must not be less than 1 second.

5 Engine Remarks

5·1 An id message is well-formed when its body is a sequence of three or more tokens. It is recommended that engines notify the client of

- the engine's name using `id name` … ,
- any authors' names using `id author` … ,

where … is any nonempty sequence of tokens.

Recall that, in all states, a message sent by the engine that is not well-formed must be ignored. All id messages that are long enough are considered to be well-formed, then, so that implementation-defined id messages besides engine and author names can be added.

5·2 An option message is well-formed when its body is of the form

`option name` …[1] `type` …[2] …[3]

where

- …[1] is any nonempty sequence of tokens that does not contain the token `type` or the token `value` (the option name),
- …[2] is `check`, `spin`, `combo`, `button`, or `string` (the option type),
- …[3] is a well-formed option schema given the option type.

A well-formed option schema is a sequence of tokens of the form

- `default true` or `default false` when the option type is `check`,
- `default` …[a] `min` …[b] `max` …[c] when the option type is `spin`, where …[a], …[b], and …[c] are decimal integers in the range 0 to $2^{63}-1$ inclusive,
- `default` …[a] (`var` …[b])+ when the option type is `combo`, where …[a] and …[b] are nonempty sequences of tokens that do not contain the token `var`,
- the empty sequence when the option type is `button`,
- `default` …[a] when the option type is `string`, where …[a] is any nonempty sequence of tokens. The one-token sequence `<empty>` must be interpreted by the client as if it were the empty sequence.

It is recommended that engines use the option name `Hash` (with option type `spin`) for cache size in megabytes and the option name `Threads` (with option type `spin`) for number of worker threads.

5·3 A protocol message is well-formed when its body is of the form

`protocol` …

where … is a single token that is the protocol identifier of a version of this specification that the engine will conform to. The protocol identifier of this version of the specification (in other words, the version described in this particular document) is `2`.

Clients should interpret the protocol identifier `1` as a indication that the engine will behave informally (in other words, as a warning that the engine may cause a violation).

5·4 The only well-formed uciok message body is the single-token sequence `uciok`.

5·5 The only well-formed ready message body is the single-token sequence `readyok`.

5·6 An info message is well-formed when its body is of the form

`info` (`string`|`error`) …

where … is any nonempty sequence of tokens, or is of the form

`info` …

where … is a nonempty sequence (search information) of well-formed fields.

Engines should only send info messages beginning with `info error` when a client error has occurred. Info messages beginning with `info string` may be sent by the engine for any purpose. Both kinds of info messages may be interpreted by the client in any implementation-defined manner; in fact, a conforming client may simply ignore all such info messages (although it is recommended that clients do not do this).

A field is a sequence of two or more tokens.

- Any number of fields may be omitted (subject to the aforementioned constraint that at least one field must be present).

- Each field may appear only once.
- Fields may appear in any order, subject to the constraint that the pv field, if present, must be last.

In the field descriptions below, the metatoken *int* is any decimal integer in the range 0 to $2^{63}-1$ inclusive. The meaning of each field is implementation-defined, but recommended use is given after an equals sign as a comment. The fields are:

- depth *int* = for traditional α/β engines, the nominal length in ply of the principal variation before extensions and reductions have been applied and not including plies examined in a quiescing search,
- seldepth *int* = for traditional α/β engines, the length in ply of the principal variation after extensions and reductions have been applied, possibly including plies explored in a quiescing search, or the length in ply of the longest line examined by the engine,
- nodes *int* = the number of positions (counted with multiplicity) examined by the engine since the most recent go message was read,
- time *int* = the time in milliseconds since the most recent go message was read,
- nps *int* = the mean number of positions (counted with multiplicity) examined per second since the most recent go message was read,
- hashfull ..., where ... is a decimal integer in the range 0 to 1000 inclusive = for traditional α/β engines, the number of unique cache entries per mille written by the engine since the most recent go message was read,
- tbhits *int* = the number of tablebase accesses since the most recent go message was read,
- currmove ..., where ... is an algebraic token = for traditional engines, when interpreted as long algebraic notation, the first move of the lines currently being examined by the representative worker (the thread that will determine which move is reported with the bestmove message),
- currmovenumber *int* = for traditional engines, an index associated with the move given by currmove in the same message, assigned in order that those moves and their continuations are examined, where 1 indicates the first move,

- multipv *int* = the rank of the line given in the pv field of the same message, where 1 indicates the principal variation, 2 indicates the principal variation if the first move of line 1 were disallowed as the first move of the principal variation, 3 indicates the principal variation if the first moves of lines 1 and 2 were disallowed as the first move of the principal variation, and so on,
- score cp ...[1] (...[2])? or score mate ...[1], where ...[1] is a decimal integer in the range $-2^{63}+1$ to $2^{63}-1$ inclusive (which may be written with a leading +) and ...[2] is either lowerbound or upperbound = when cp, an estimate of the advantage in centipawn for the side-to-move, where negative values of ...[1] indicate that the side-to-move is disadvantaged; when mate, an indication that a forced mate in this many full moves exists, where positive values of ...[1] indicate that the side-to-move can deliver mate so that the game is over after ...[1]×2 − 1 additional plies have been played (including the move that the side-to-move is about to make) and negative values of ...[1] indicate that the side-waiting can deliver mate so that the game is over after ...[1]×2 additional plies have been played (including the move that the side-to-move is about to make); when lowerbound is present, an indication that ...[1] is an estimated bound in centipawn for the advantage for the side-to-move and the advantage is estimated to be ...[1] or greater; when upperbound is present, an indication that ...[1] is a likewise an estimated bound and the advantage is estimated to be ...[1] or less,
- pv ..., where ... is a sequence of algebraic tokens = the principal variation, or when the currmove field is present, the principal variation if the move given by the currmove field in the same message were required to be the first move of the principal variation.

Additional implementation-defined fields besides those listed above may be included in search information. Implementation-defined fields must not have any contiguous subsequence of tokens matching the fields listed above.

The second token of the currmove field, when interpreted as long algebraic notation, should correspond to a legal move for the side-to-move in the position defined by the last well-formed position message

sent by the client in the idle state (or in the starting position if no such position messages have been sent), but the field is still well-formed even when this is not the case (although clients may then ignore the field, since the meaning of the field is implementation-defined).

Likewise, the tokens of the pv field following the token pv, when interpreted as long algebraic notation, should correspond to a sequence of legal moves from the position defined by the last well-formed position message sent by the client in the idle state (or from the starting position if no such position messages have been sent), but the field is still well-formed even when this is not the case (although clients may then ignore the field, since the meaning of the field is implementation-defined).

Many legacy clients require the `multipv` field whenever the pv field is present, regardless of engine configuration. This behavior is strongly discouraged.

5·7 A bestmove message is well-formed when its body is of the form

`bestmove 0000`

or is of the form

`bestmove ...`

where ... is an algebraic token that, when interpreted as long algebraic notation, corresponds to a legal move for the side-to-move in the position defined by the last well-formed position message sent by the client in the idle state (or in the starting position if no such position messages have been sent).

The token 0000 (or *null move*) should generally be used to indicate that the engine was unable or is not willing to recommend a move.

All bestmove messages sent in the active, ping, and halt states must be well-formed. In other words, when the engine sends a bestmove message in these states that is not well-formed, a violation has occurred.

6 Client Commands

6·1 The only well-formed uci message body is the one-token sequence `uci`.

6·2 A `setoption` message is well-formed when its body is of the form

`setoption name` …[1] `value` …[2]

where

- …[1] is a nonempty sequence of tokens that match an option name that has been advertised by the engine whose option type is `check`, `spin`, `combo`, or `string`,
- …[2] is `true` or `false` when the option type is `check`,
- …[2] is a decimal integer not less than the minimum value nor greater than the maximum value advertised by the engine when the option type is `spin`,
- …[2] is one of the token sequences advertised by the engine when the option type is `combo`,
- …[2] is any nonempty sequence of tokens when the option type is `string`,

or is of the form

`setoption name` …

where … is a nonempty sequence of tokens that match an option name that has been advertised by the engine whose option type is `button`.

When the option type is `string`, the single-token sequence `<empty>` must be interpreted by the engine as if it were the empty sequence.

6·3 The only well-formed `debug` message bodies are the sequences `debug on` and `debug off`.

6·4 The only well-formed `ucinewgame` message body is the single-token sequence `ucinewgame`.

6·5 The only well-formed `isready` message body is the single-token sequence `isready`.

6·6 A `position` message is well-formed when its body is of the form

`position` …[1] (`moves` …[2])?

where

- …[1] is the token `startpos` or the seven-token sequence `fen` …[f], where …[f] is the description of a position in Forsyth-Edwards Notation,

- …[2] is a sequence of algebraic tokens that, when interpreted as long algebraic notation, correspond to a sequence of legal moves from the position described by …[1].

In the final position (the position described by the entirety of the position message) there must be at least one legal move available for the side-to-move (in other words, the game must not be over), otherwise the message is not well-formed.

Recall that position messages sent in idle state that are not well-formed must be ignored by the engine, so engines may not partially apply position updates.

6·7 A go message is well-formed when its body is of the form

`go infinite` (…)?

where … is a nonempty sequence of well-formed search modifiers, or of the form

`go` (…)?

where … is a nonempty sequence of well-formed search limits, search context, and/or modifiers.

Modifiers, limits, and context items are sequences of two or more tokens.

- Each limit, context item, or modifier may appear only once.
- Limits, context items, and modifiers may appear in order, subject to the constraint that the `searchmoves` modifier, if present, must be last.

The meaning of search modifiers, search limits, and search context is implementation-defined, but recommended use is given after an equals sign as a comment.

In the descriptions below, the metatoken *int16* is any decimal integer in the range 1 to $2^{15}-1$ inclusive, the metatoken *int32* is any decimal integer in the range 0 to $2^{31}-1$ inclusive, and the metatoken *int64* is any decimal integer in the range 0 to $2^{63}-1$ inclusive. Note that, unlike *int32* and *int64*, an *int16* **cannot** be zero.

The search modifiers are:

- `searchmoves` ..., where ... is a nonempty sequence of algebraic tokens that, when interpreted as long algebraic notation, each correspond to a legal move for the side-to-move in the position defined by the last well-formed position message sent by the client in the idle state (or in the starting position if no such position messages have been sent) = a collection of moves to which the engine should restrict its consideration (in other words, the move reported with the bestmove message should be one of the moves in this collection),
- `mate` *`int16`* = an indication that the engine should attempt to prove a mate in this many full moves (or twice this many plies) and may assume that it does not need to examine lines beyond this many full moves (or twice this many plies).

The search context items are:

- `wtime` *`int32`* = the number of milliseconds that the white side has on their clock,
- `btime` *`int32`* = the number of milliseconds that the black side has on their clock,
- `winc` *`int32`* = the number of milliseconds that will be added to white's clock after each move by white,
- `binc` *`int32`* = the number of milliseconds that will be added to black's clock after each move by black,
- `movestogo` *`int16`* = an indication that when the side-to-move has played this many additional moves (including the move that the side-to-move is about to make) a new time control will begin.

The search limits are:

- `depth` *`int16`* = for traditional α/β engines, the maximum length in ply of the principal variation (before extensions and reductions have been applied, and not including plies examined in a quiescing search) that the engine should explore,
- `nodes` *`int64`* = for traditional engines, the maximum number of positions (counted with multiplicity) that the engine should examine,

- movetime *int32* = the maximum time in milliseconds that the engine should spend after reading this message before sending a bestmove message.

When more than one limit is given, the halting condition should be the logical OR of the limits rather than the logical AND. In other words, the engine should stop and send a bestmove message when any of the limiting conditions are met.

Since the meaning of search modifiers, search limits, and search context are implementation-defined, engines may ignore them (for example, node limits). However, disregard or lack of support for search context and the depth and movetime limits is strongly discouraged.

6·8 The only well-formed stop message body is the single-token sequence stop.

6·9 The only well-formed quit message body is the single-token sequence quit.

Revision Notes

This specification (or SPEC) is primarily a formalization of the *Description of the universal chess interface* (or HMK) by Rudolf Huber and Stefan Meyer-Kahlen, but it differs in the following ways:

- HMK states that, when the client sends the message go infinite, the engine must "not exit the search" until it is sent a following stop message, whereas in the active state SPEC always allows engines to return a bestmove message. This is meant to simplify the specification and allow engines to signal that they have reached their maximum depth and will make no further progress.

- SPEC allows the client to send the engine a go message without a corresponding position message, whereas HMK requires the client to send a position message before it sends the first go message and in between every pair of go messages.

- HMK states that clients and engines should attempt to skip unknown or malformed tokens and continue parsing the remainder of a message, whereas SPEC states that clients and engines should ignore the message.

- HMK includes U+0009 CHARACTER TABULATION in its definition of whitespace (and possibly other scalar values as well), whereas SPEC implicitly defines whitespace as contiguous sequences of U+0020 only.

- SPEC does not include the register command or registration and copyprotection remarks. The effect of a register message may be emulated with a setoption message, and registration and copyprotection messages may be emulated with info messages.

- HMK allows search information to end with a `string` field, whereas SPEC does not. HMK also disallows the `seldepth` field when the `depth` field is not present, but SPEC does not impose this restriction.

- SPEC introduces the protocol remark.

Although SPEC is designed so that conforming engines will be usable with nonconforming, legacy clients (and vice versa), some configurations may be incompatible.

In future versions of this specification, this section will also list changes that have been made from prior versions.

Outstanding Issues

- The transition diagram needs to show that when the engine sends a bestmove message in the active, ping, and halt states that is not well-formed, a violation has occurred.

- Pondering still needs to be incorporated.

- Should `refutation` and `currline` be included in the list of governed fields? or left as implementation-defined additions?

- Should this specification introduce WDL scoring (or something similar) in addition to centipawn-advantage scoring?

‣ Should `upperbound` and `lowerbound` not be required to come after the score?

‣ Should the client be allowed to queue up messages in the halt state?

‣ Should the client be allowed to send a `quit` message anytime?

‣ Should a `stop` message in the idle state be considered an error? In other words, should the engine notify the client with `info error …` that it received a `stop` message in the idle state, rather than with a generic `info` message?

‣ Should the value for `string`-type options be a sequence of bytes rather than a seqence of tokens? – since (1) file system paths may contain U+0020, where it's not a separator and the number of contiguous U+0020 values is significant, and (2) file system paths are often just byte sequences. This latter fact means that it'd be insufficient to relax the ASCII restriction on client messages; the UTF-8 restriction needs to be relaxed as well. (In fact, it's worse than that: file system paths can also include U+000A, so an escape sequence would need to be devised.)

‣ Should this formalization of UCI be given a name? If so, what?

‣ There should be a set of conformance tests that client and engine developers can use to find ways in which their programs do not adhere to specification. (Help writing these would be welcome.)