Interpreter Events

Table of Contents

1. Concept

In order to support writing debuggers, profilers, analyzers and other useful tools that are needed for efficient programming, the interpreter is offering a generic event-oriented low-level API on top of which all these tools can be built. It is similar to the `parse/trace` and `lexer/trace` instrumentation approach.

2. Interpreter instrumentation

In order to access internal interpreter states, the interpreter is generating events at key points of his internal code evaluation process. Those events can be captured using a user-provided callback function. Events are emitted only if a *tracing* mode is enabled in the interpreter using `/trace` refinement on `do`.

```
do/trace <code> <callback>

<code>     : a block of code to evaluate in tracing mode.
<callback> : a callback function, triggered on each interpreter event.
```

Example

```
logger: function [
    event  [word!]                      ;-- Event name
    code   [block! paren! none!]        ;-- Currently evaluated block
    offset [integer!]                   ;-- Offset in evaluated block
    value  [any-type!]                  ;-- Value currently processed
    ref    [any-type!]                  ;-- Reference of current call (usually word or
path)
    frame  [pair!]                      ;-- Stack frame start/top positions
][
    print [
        pad uppercase form event 8
        mold/part/flat either any-function? :value [:ref][:value] 20
    ]
]

do/trace [print 1 + 2] :logger
```

will output:

```
INIT    none                    ;-- Initializing tracing mode
ENTER   none                    ;-- Entering block to evaluate
FETCH   print                   ;-- Fetching and evaluating `print` value
OPEN    print                   ;-- Results in opening a new call stack frame
FETCH   +                       ;-- Fetching and evaluating `+` infix operator
OPEN    +                       ;-- Results in opening a new call stack frame
FETCH   1                       ;-- Fetching left operand `1`
PUSH    1                       ;-- Pushing integer! value `1` on stack
FETCH   2                       ;-- Fetching and evaluating right operand
PUSH    2                       ;-- Pushing integer! value `2`
CALL    +                       ;-- Calling `+` operator
RETURN  3                       ;-- Returning the resulting value
CALL    print                   ;-- Calling `print`
3                               ;-- Outputting 3
RETURN  unset                   ;-- Returning the resulting value
EXIT    none                    ;-- Exiting evaluated block
END     none                    ;-- Ending tracing mode
```

3. Events

When the tracing mode is active, the interpreter will trigger events described below. Events can be grouped into the following categories:

- Global events: `INIT`, `END`
- Evaluating a block/paren of code: `ENTER`, `EXIT`
- Calling any type of function: `OPEN`, `CALL`, `RETURN`
- Evaluating a function body block: `PROLOG`, `EPILOG`

- Expression evaluation: `FETCH`, `PUSH` , `SET`, `ERROR`
- Exceptions handling: `THROW`, `CATCH`

Detailed description:

Event	Code	Offset	Value	Ref	Description
`INIT`	`none`	-1	n/a (`none`)	n/a (`none`)	when the tracing mode is initiated (`do/trace` call).
`END`	`none`	-1	n/a (`none`)	n/a (`none`)	when the tracing mode is ended (`do/trace` call exiting).
`ENTER`	`block!`, `paren!`	-1	n/a (`none`)	n/a (`none`)	when a block is about to be evaluated.
`EXIT`	`block!`, `paren!`	-1	n/a (`none`)	n/a (`none`)	when current evaluated block's tail has been reached.
`OPEN`	`block!`, `paren!`	`integer!`	`any-function!`	`word!`, `path!`	when a new function (any-function!) call is initiated and a new stack frame is opened.
`CALL`	`block!`, `paren!`	`integer!`	`any-function!` to call	`word!`, `path!`, `any-function!`	a function with all arguments fetched on the stack gets called.
`RETURN`	`block!`, `paren!`	`integer!`	returned `any-type!` value	`word!`, `path!`	when a function call has returned and its stack frame has been closed.
`PROLOG`	`block!`, `paren!`	-1	called `function!` value	`word!`, `path!`	when entering a function! body.
`EPILOG`	`block!`, `paren!`	-1	called `function!` value	`word!`, `path!`	when exiting a function! body.
`FETCH`	`block!`, `paren!`	`integer!`	fetched `any-type!` value	n/a (`none`)	a value is read from the input block to be evaluated.
`PUSH`	`block!`, `paren!`	`integer!`	pushed `any-type!` value	n/a (`none`)	a value has been pushed on the stack.
`SET`	`block!`, `paren!`	`integer!`	`any-type!`	`set-word!`, `set-path!`	a set-word or set-path is set to a value.
`ERROR`	`none`	-1	`error!` value	n/a (`none`)	when an error occurs and is about to be thrown up in the stack.

Event	Code	Offset	Value	Ref	Description
`THROW`	`none`	-1	thrown `any-type!` value	n/a (`none`)	when a value is thrown using `throw` native.
`CATCH`	`none`	-1	thrown `any-type!` value	n/a (`none`)	when a value is caught using `catch` native.

Events come with extra information:

- `code`: when available, it provides the input `block!` or `paren!` series currently interpreted.
- `offset`: when different from `-1`, indicates the input series offset at the event moment.
- `value`: when available, the currently processed value.
- `ref`: when available, references the word or path from which evaluation produced the current event/value.

4. Event handler

Here is the prototype of event handlers suitable to be passed as argument to `do/trace`:

```
func [
    event  [word!]
    code   [block! paren! none!]
    offset [integer!]
    value  [any-type!]
    ref    [any-type!]
    frame  [pair!]
][
    [events]                    ;-- optional restricted event names list
    ...body...
]
```

Argument	Description
`event`	Event name.
`code`	Block of code currently evaluated.
`offset`	Offset in block currently evaluated.
`value`	Value currently processed in the event.
`ref`	Reference of the call (word or path) associated to the event.
`frame`	Pair of indexes in the Red internal stack denoting the beginning and end of the call frame. (1)

(1) Note that the `frame` index range is for the internal Red stack, not the one used in the debugger (which is managed by the debugger itself).

The body block can start with an optional filtering block, for indicating which events will be triggered. This allows to reduce the number of callback calls resulting in much better processing performance.

5. Predefined tools

5.1. `debug`

Debugger commands:

- `next` or `n` or just ENTER: evaluate next value.
- `continue` or `c`: exit debugging console but continue evaluation.
- `quit` or `q`: exit debugger and stop evaluation.
- `stack` or `s`: display the current calls and expression stack.
- `parents` or `p`: display the parents call stack.
- `:word`: outputs the value of `word`. If it is a `function!`, outputs the local context.
- `:a/b/c`: outputs the value of `a/b/c` path.
- `watch <word1> <word2>…`: watch one or more words. `w` can be used as shortcut for `watch`.
- `-watch <word1> <word2>…`: stop watching one or more words. `-w` can be used as shortcut for `-watch`.
- `+stack` or `+s`: outputs expression stack on each new event.
- `-stack` or `-s`: do not output expression stack on each new event.
- `+locals` or `+l`: output local context for each entry in the callstack.
- `-locals` or `-l`: do not output local context for each entry in the callstack.
- `+indent` or `+i`: indent the output of the expression stack.
- `-indent` or `-i`: do not indent the output of the expression stack.

5.2. `profile`

TBD

5.3. `trace`

TBD

5.4. Dumping raw events

TBD

6. Implementation notes