Duktape debugger

The debugger support is currently experimental, i.e. may change in an incompatible way even in a minor release.

Introduction

Overview

Duktape provides the following basic debugging features:

- Execution status: running/paused at file/line, call stack, local variables
- Execution control: pause, resume, step over, step into, step out

- Breakpoints: file/line pair targeted breakpoint list, "debugger" statement
- Eval in the context of the current activation when paused (can be used to implement basic watch expressions)
- Get/put variable
- Forwarding of print(), alert(), and logger writes
- Full heap dump (debugger web UI converts to JSON)

Duktape debugging architecture is based on the following major pieces:

- A standard **debug protocol**, implemented directly by Duktape.
- A reliable **debug transport** stream, implemented by the application.
- A **debug API** to attach/detach a debugger to a Duktape heap.
- A **debug client**, running off target, which implements the other debug protocol endpoint and provides a user interface.
- An optional **JSON debug protocol proxy** which provides an easier JSON-based interface for talking to the debug target.

This document describes these pieces in detail.

Getting started: debugging with "duk"

See `debugger/README.rst` for concrete instructions on how to use `duk --debugger` as a debug target and `debugger/duk_debug.js` as a debug client.

Getting started: debugging your target

To integrate debugger support into your target, you need to:

- **Check your feature options**: define `DUK_OPT_DEBUGGER_SUPPORT` and `DUK_OPT_INTERRUPT_COUNTER` to enable debugger

support in Duktape. Also consider other debugging related feature options, like forwarding `print()` / `alert()` to the debug client.

- **Implement a concrete stream transport mechanism**: needed for both the target device and the Duktape debugger. The best transport depends on the target, e.g. a TCP socket, a serial link, or embedding debug data in an existing custom protocol. An example TCP debug transport is given in `examples/debug-trans-socket/duk_debug_trans_socket.c`.
- **Add code to attach a debugger**: call `duk_debugger_attach()` when it is time to start debugging. Duktape will pause execution and process debug messages (blocking if necessary). Execution resumes under control of the debug client.
- **When done, detach the debugger**: call `duk_debugger_detach()` to stop debugging; any debug stream error also causes an automatic detach. When a detach occurs, Duktape resumes normal execution and ignores breakpoints etc. (A detach can also occur if explicitly requested by the debug client or if Duktape detects a debug stream error.)
- **If you have an eventloop**: optionally call `duk_debugger_cooperate()` once in a while when no call to Duktape is in progress. This allows debug commands to be executed outside of any Duktape calls.

You can also write your own debug client by implementing the client side of the debug protocol. The debug client is intended to adapt to the target debug protocol version, so your debug client may need changes from time to time as the Duktape debug protocol evolves. The debug protocol is versioned with the same semantic versioning principles as the Duktape API.

You can implement the binary debug protocol directly in your debug client, but an easier option is to use the JSON mapping of the debug protocol which is much more user friendly. Duktape includes a proxy server which converts between the JSON mapping and the binary debug protocol (which actually runs on the target).

Example debug client and server

The Duktape repo contains an example debugger web UI which uses TCP for the debug transport and can

communicate with the Duktape command line tool ("duk"). This running example serves to further document the concrete details of debug commands and how to implement a debug transport. The web console can also directly talk to any other debug targets that use a TCP debug transport.

The example debugger stuff includes:

- Duktape command line tool `--debugger` option which is enabled by using both `DUK_CMDLINE_DEBUGGER_SUPPORT` and `DUK_OPT_DEBUGGER_SUPPORT`. The command line tool uses a TCP socket based example transport provided in `examples/debug-trans-socket/`.
- NodeJS + ExpressJS based minimal debugger Web UI in `debugger/` directory, which uses a TCP socket for debug transport.

While TCP is a good example transport, it is not a "standard" transport: the transport is always ultimately up to the user code.

What Duktape doesn't provide

A standard debug transport

This is up to user code, because the most appropriate transport varies a great deal: Wi-Fi, serial port, etc. However, TCP is probably a good default transport if there are no specific reasons not to use it.

A standard debugger UI

User code must implement a concrete debugger interface on top of the debug commands supported by Duktape. However, Duktape does contain a fully functional debugger example. You can extend it as necessary or write your own.

Function source code

Duktape doesn't currently provide function source code over the debug protocol. The debug client is assumed to have access to matching source code, and have the ability to find a source file matching a certain filename. This also means that functions created using `eval` cannot be debugged with source present.

Impact of enabling debugger support

Performance

There should be very minimal performance impact, except when a debugger is attached and a running function has active breakpoints.

When bytecode executor restarts it quickly determines that a debugger is not attached and breakpoints don't need to be processed. The bytecode executor interrupt must be enabled to use a debugger which has some impact on bytecode execution.

Duktape enters "checked execution" when a debugger is attached and the current function has active breakpoints. Checked execution (see below for discussion) is much slower than normal execution; the interrupt handler is executed after every bytecode instruction.

Code footprint

Debugger support increases footprint by around 10kB.

Memory footprint

`duk_heap` structure size increases because of heap-level debugger state. If you're using finely tuned memory pools, memory pool sizes may need to be retuned.

Function instances will always keep their internal `_Varmap` property so that local variables can always be looked up by name. Without debugger support the `_Varmap` is only kept when it might be needed during execution (e.g. the function contains an eval call).

Otherwise memory footprint should be negligible. Duktape doesn't need to maintain any debug message buffering because all debug data is streamed in and out.

Debug API

duk_debugger_attach()

Called when the application wants to attach a debugger to the Duktape heap:

```
duk_debugger_attach(ctx,
                    my_trans_read_cb,         /* read callback */
                    my_trans_write_cb,        /* write callback */
                    my_trans_peek_cb,         /* peek callback (optional) */
                    my_trans_read_flush_cb,   /* read flush callback (optional) */
                    my_trans_write_flush_cb,  /* write flush callback (optional) */
                    my_detached_cb,           /* debugger detached callback */
                    my_udata);                /* debug udata */
```

When called, Duktape will enter debug mode, pause execution, and wait for further instructions from the debug client.

The transport callbacks are given as part of the start request. Duktape expects a new virtual stream for every debug start/stop cycle, and will send a protocol version identifier every time `duk_debugger_attach()` is called.

The detached callback is called when the debugger becomes detached. This can happen due to an explicit request (`duk_debugger_detach()`), a debug message/transport error, or Duktape heap destruction.

If Duktape debugger support is not enabled, an error is thrown.

duk_debugger_detach()

Called when the application wants to detach a debugger:

```
duk_debugger_detach(ctx);
```

When the debugger is detached, Duktape resumes normal execution. Any remaining debug state (like breakpoints) is ignored.

If Duktape debugger support is not enabled, an error is thrown.

duk_debugger_cooperate()

Optional call to process inbound debug commands when no call into Duktape is active:

```
duk_debugger_cooperate(ctx);
```

Pending debug commands are executed within the context of the `ctx` thread. All debug commands that can be executed without blocking are executed during the call. Because the call doesn't block, it is safe to embed in an event loop. The call is a no-op when debugging is not supported or active, so it can be called without a debug state check.

Note that:

- The caller is responsible for **not** calling this API function when any call to Duktape is active (for any context).
- The interval between duk_debugger_cooperate() calls affects Duktape's reaction time to pending debug commands.

This API call is needed by some applications to allow debug commands such as Eval to be executed when no call into Duktape is active. For example:

```
for (;;) {
    /* Wait for events or a timeout. */
    wait_for_events_or_timeout();

    /* Process events. */
    if (event1) {
        ...
    }
    /*...*/

    /* Cooperate with Duktape debugger. */
    duk_debugger_cooperate(ctx);
}
```

Because the API call processes all pending inbound messages (available without blocking), you can also use it like this:

```
for (;;) {
    /* Wait for events or a timeout. */
    wait_for_events_or_timeout();

    /* Process events. */
    if (got_inbound_debugger_data) {
        /* Cooperate with Duktape debugger: process all pending messages
         * until new inbound data arrives.
         */
        duk_debugger_cooperate(ctx);
    }
    /*...*/
```

```
}
```

Debug transport

Overview

Duktape debugger code sends and receives debug messages over an abstracted reliable stream transport with semantics similar to a TCP connection or a serial link. To maximize portability to different environments, Duktape expects user code to provide the concrete implementation for this transport in the form of callbacks given to `duk_debugger_attach()`.

The logical service provided by the transport is a reliable byte stream with primitives to:

- Read bytes (partial read OK, block if necessary to read at least 1 byte)
- Write bytes (partial write OK, block if necessary to write at least 1 byte)
- Peek for inbound byte(s) without blocking
- Read flush hint
- Write flush hint

Partial reads and writes are allowed to make it as easy as possible to implement the transport callbacks. Duktape will handle any "read fully" and "write fully" semantics automatically by calling read and write as many times as necessary.

Peeking allows Duktape to detect incoming debug messages without blocking. This allows debug messages to be processed even when Duktape is running normally (not in paused state).

Write flushes allow a transport implementation to reliably coalesce writes. Read flushes allow a transport implementation to manage a receive window more efficiently. The read/write flush callbacks are only needed for some types of transports.

This section covers the detailed semantics for each callback, and discusses other transport related common issues like flow control, compression, and security.

IMPORTANT: The application should assign no meaning to read/write chunk boundaries. There is no guarantee that read, write, peek, or flush calls have any correspondence to debug message boundaries.

Read callback semantics

- Read length is guaranteed to be >= 1.
- Buffer pointer is guaranteed to be non-NULL.
- Duktape is requesting that at least one and at most "length" bytes are read. Partial reads are OK but at least one byte must be read. If user code cannot read at least one byte, it MUST block until it can. If one or more bytes are available, user code MUST NOT block.
- Return value in the range [1,length] indicates how many bytes were read into the given buffer.
- Return value 0 indicates a stream error (sanity timeout, connection close, etc). Duktape will then mark the stream broken and won't do any more operations on it. Debugger will automatically detach.

Write callback semantics

- Write length is guaranteed to be >= 1.
- Buffer pointer is guaranteed to be non-NULL.
- Duktape is requesting that at least one and at most "length" bytes are written. Partial writes are OK but at least one byte must be written. If user code cannot write at least one byte, it MUST block until it can.
- Return value in the range [1,length] indicates how many bytes were written from the given buffer.
- Return value 0 indicates a stream error (sanity timeout, connection close, etc). Duktape will then mark the stream broken and won't do any more operations on it. Debugger will automatically detach.

Peek callback semantics

- Implementing a peek callback is optional (NULL can be passed in `duk_debugger_attach()`) but strongly recommended. If the callback is not provided, some features like pausing execution "out of the blue" (while Duktape is running normally) will not work.
- Peek callback has no arguments.
- Duktape is requesting a peek into the input stream, i.e. to see if at least one byte can be read without blocking.
- Return value 0 indicates no bytes can be read without blocking.
- Return value > 0 indicates the number of bytes that can be read without blocking. Right now Duktape only cares if at least one byte is available, so returning 0 or 1 is sufficient.
- Duktape will currently assume that if at least one byte is available, a whole debug message can be read (blocking and handling partial reads as necessary).

Read flush callback semantics

- Implementing a read flush callback is optional (NULL can be passed in `duk_debugger_attach()`).
- Read flush callback has no arguments.
- Duktape is indicating a "read flush" to user code. Duktape is guaranteed to indicate a "read flush" when it may not be doing any more reads on that particular occasion. (However, Duktape may indicate read flushes even when it continues to do reads immediately afterwards.)
- For most transports a read flush is not important. If the transport protocol uses a limited read window and has a protocol to update the window status to the remote peer, window control messages can be postponed to the next read flush (if there's no other pressing reason to send them, e.g. a read buffer empty condition).

Write flush callback semantics

- Implementing a write flush callback is optional (NULL can be passed in `duk_debugger_attach()`).

- Write flush callback has no arguments.
- Duktape is indicating a "write flush" to user code. Duktape is guaranteed to indicate a "write flush" when it may not be doing any more writes on that particular occasion. (However, Duktape may indicate write flushes even when it continues to do writes immediately afterwards.)
- This indication is useful if the user transport coalesces writes into larger chunks. The user code can send out chunks when buffered data becomes large enough or a write flush is indicated. User code can rely on a write flush happening when it matters.
- User code is also free to ignore this indication if it doesn't apply to the underlying transport (e.g. when using TCP, there are already mechanisms for automatic coalescing of writes) or if there's some other mechanism (e.g. a timer) in place to ensure pending bytes are eventually sent out.

Marking a transport broken

Duktape marks a transport broken if:

- User callbacks indicate a stream error
- Duktape encounters a parse error when parsing the debug stream

When the debug transport has been marked broken:

- Debugger is automatically detached so that normal Ecmascript execution will resume immediately. If a detached callback exists, it will be called.
- Duktape won't make any more calls to user callbacks for the stream.
- Duktape internal debug read calls return dummy values (zero when reading a byte, zero when reading an integer, empty string when reading a string, etc) and writes are silently ignored. This allows the implementation to read and write data without checking for errors after every read/write; an explicit check for "broken transport" can be made where it's most convenient.

Peek request notes

Duktape uses peek requests to detect incoming debug commands and process them. Peeks are used both during normal execution (when there are no relevant breakpoints and stepping is not active) and during checked execution (when there is one or more active breakpoints and/or stepping is active).

The rate of peek requests is automatically rate limited by Duktape using a Date-based timestamp, so that peeks are performed at most every 200ms.

Write flush notes

Duktape uses write flushes to indicate that it may not be sending any more data on this occasion, and that the application should send out any pending data it has queued.

Duktape writes outbound debug messages in very small pieces so it might make sense for the application to maintain a buffer for pending outbound data. When Duktape performs a write, data can be appended to the buffer. Data can be sent out when the buffer is large enough, or when Duktape performs a write flush.

A write flush is guaranteed to occur when Duktape is finished processing a set of messages so an application doesn't need to have a separate timer mechanism or similar to flush pending writes. A write flush is **not** guaranteed after every outbound debug message (although the current Duktape implementation behaves that way).

The user code should make no assumptions about when Duktape indicates a write flush, other than to send out pending bytes when it happens.

Reliability

Duktape expects the transport to be reliable, i.e. no bytes are reordered, lost, or duplicated. The concrete transport

must provide reliability by application specific means. For instance, if TCP sockets are used, reliability is automatically provided by TCP. For unreliable packet transports, user code must provide retransmission, duplicate detection, and sequencing.

Flow control

There is no flow control at the abstract transport level, though an application is free to implement flow control as part of the transport. For instance, if TCP sockets are used, there's automatic flow control as part of TCP.

Flow control may be necessary for devices with very low amount of memory so that excessive buffer reserves can be avoided.

Compression

For very slow links it may be appropriate for the application specific transport to use stream compression for the debug traffic. Compression can reduce the stream to around 10-30% of its uncompressed size.

Security

In some environments the debug transport may be security critical. In such cases the application should use authentication and encryption for the debug transport, e.g. use SSL/TLS for the transport.

Implementing on top of a packet-based transport

This topic is covered in a separate section.

Development time transport torture option

The feature option DUK_OPT_DEBUGGER_TRANSPORT_TORTURE causes Duktape to do all debug transport read/write operations in 1-byte steps, which is useful to catch any incorrect assumptions about reading or writing chunks of a certain size.

Debug stream format

Overview

The debug protocol is a conversation between Duktape internals and the debug client. User code is not aware of the contents of the debug protocol, it only provides a debug transport to carry chunks of the stream between the debug target and the debug client.

The debug protocol has a simple three-part life cycle:

- Stream connected, waiting for version identification (sent by Duktape).
- Stream connected, in active use. Debug messages are exchanged freely in each direction.
- Stream disconnected. This happens on an explicit detach request (i.e. a call to `duk_debugger_detach()`, a read/write error indicated by the user's transport callbacks, a message syntax error detected by Duktape, or Duktape heap destruction.

The protocol uses request pipelining, i.e. each peer is allowed to send multiple requests without waiting for replies to previous requests. To facilitate this, every request has a corresponding reply/error message and requests are always processed without reordering. Neither peer is required to send pipelined request, and it's perfectly fine for e.g. a debug client to wait for a response before sending another request.

Version identification

When the debug transport is attached, Duktape writes a version identification as an UTF-8 encoded line of the form:

```
<protocolversion> <SP (0x20)> <additional text, no LF> <LF (0x0a)>
```

The current protocol version is "1" and the identification line currently has the form:

```
1 <DUK_VERSION> <DUK_GIT_DESCRIBE> <target string> <LF>
```

Everything that follows the protocol version number is informative only. Example:

```
1 10099 v1.0.0-254-g2459e88 duk command built from Duktape repo
```

The debug protocol version is available as a define to the user code (defined by `duktape.h`):

```
DUK_DEBUG_PROTOCOL_VERSION
```

This may be useful e.g. when a target can advertise its debug capabilities.

The debug client should parse the line and check the protocol version first. If the protocol version is not supported, the debug connection should be closed. The debug client always adapts to the protocol version present on the target. There is no acknowledgement to the version identification, and there is no corresponding handshake message from the debug client.

When the version identification (handshake) is complete the debug stream switches to a different framing described below. The framing is potentially protocol version specific, which is why the version identification must be processed first.

Some rationale for the version identification format:

- A one-line text string is a common handshake approach, and has the benefit that you can telnet into a target (if it uses a TCP transport) and easily see that you've connected to a debugger port. It can also be easily extended to e.g. allow Duktape to advertise optional capabilities (if that becomes necessary).
- The version identification allows protocol framing to be changed in the future without changing the handshake format. If version identification used the more complex framing described below, it would make version compatibility much harder.
- Duktape just sends out the version identification blindly and doesn't need to parse a reply, so there's very little cost to having a human readable version identification line as compared to e.g. sending a single version byte.
- Adding a version identification for the debug client would mean unnecessary parsing state for Duktape. There's little benefit in making Duktape aware of the debug client version.

Dvalue

After the version identification handshake, the debug stream consists of typed values called *dvalues* sent in each direction. Dvalues represent message framing markers, integers, strings, tagged Ecmascript values, etc. They can be parsed without context which is useful for dumping and also allows dvalues (and debug messages) to be skipped without context. Debug *messages* are then constructed as a sequence of dvalues: a start marker, zero or more dvalues, and an end-of-message marker.

The following table summarizes the dvalues and their formats. The initial byte (IB) is used as both a type tag and containing parts of the value in some cases:

Byte sequence	Type	Description
0x00	EOM	End of message

Byte sequence	Type	Description
0x01	REQ	Start of request message
0x02	REP	Start of success reply message
0x03	ERR	Start of error reply message
0x04	NFY	Start of notification message
0x05...0x0f	reserved	
0x10 <int32>	integer	4-byte integer, signed 32-bit integer in network order follows initial byte
0x11 <uint32> <data>	string	4-byte string, unsigned 32-bit string length in network order and string data follows initial byte
0x12 <uint16> <data>	string	2-byte string, unsigned 16-bit string length in network order and string data follows initial byte
0x13 <uint32> <data>	buffer	4-byte buffer, unsigned 32-bit buffer length in network order and buffer data follows initial byte
0x14 <uint16> <data>	buffer	2-byte buffer, unsigned 16-bit buffer length in network order and buffer data follows initial byte
0x15	unused	Represents the internal "undefined unused" type which used to e.g. mark unused (unmapped) array entries
0x16	undefined	Ecmascript "undefined"
0x17	null	Ecmascript "null"

Byte sequence	Type	Description
0x18	true	Ecmascript "true"
0x19	false	Ecmascript "false"
0x1a <8 bytes>	number	IEEE double (network endian)
0x1b <uint8> <uint8> <data>	object	Class number, pointer length, and pointer data (network endian)
0x1c <uint8> <data>	pointer	Pointer length, pointer data (network endian)
0x1d <uint16> <uint8> <data>	lightfunc	Lightfunc flags, pointer length, pointer data (network endian)
0x1e <uint8> <data>	heapptr	Pointer to a heap object (used by DumpHeap, network endian)
0x1f	reserved	
0x20...0x5f	reserved	
0x60...0x7f <data>	string	String with length [0,31], string length is IB - 0x60, data follows
0x80...0xbf	integer	Integer [0,63], integer value is IB - 0x80
0xc0...0xff <uint8>	integer	Integer [0,16383], integer value is ((IB - 0xc0) << 8) + followup_byte

All "integer" representations are semantically the same, i.e. they can all be used wherever an integer is expected. Same applies to "string" and "buffer" representations.

The dvalue typing is sufficient to represent `duk_tval` values so that typing can be preserved (e.g. strings and buffers have separate types).

The dvalues are represented as follows in text below (not needed for all types in the text):

```
EOM
REQ
REP
ERR
NFY
<int: field name>    e.g. <int: error code>
<str: field name>    e.g. <str: error message>
<buf: field name>    e.g. <buf: buffer data>
<ptr: field name>    e.g. <ptr: prototype pointer>
<tval: field name>   e.g. <tval: eval result>
```

When a field does not relate to an Ecmascript value exactly, e.g. the field is a debugger control field, typing can be loose. For example, a boolean field can be represented sometimes as integer dvalue and an arbitrary binary string as a string dvalue. The specific types used for each command are described in per-command sections below.

The intent behind the dvalue format is to:

- Make the lowest level protocol typed so that dvalues and messages can be dumped without knowing the particular message being parsed.
- Provide a way to skip a message without understanding its contents, or ignore trailing fields in a message, by scanning for the EOM marker. This is useful for handling unsupported requests and for extending messages by appending dvalues to existing ones. However, note that reliable skipping is only possible if an implementation can parse all dvalue types so that it knows their length. In particular, zero bytes (which are used for EOM) can appear inside dvalues too, so skipping to zero byte is not a reliable way to skip.
- Provide a framing for requests and responses, which is needed to ensure both peers can distinguish replies to its own requests from requests or notifications initiated by the other party.
- Allow streamed writing of debug messages without knowing the length of the final message in advance (which

would be necessary if the framing had a leading message length field, for instance). This is useful to avoid the need to precompute message sizes or to use an accumulation buffer to create a full message before sending it out.
- Represent all `duk_tval` values without loss of information.
- Use short encoding forms for typical numbers and strings to minimize traffic for low bandwidth debug transport (like serial lines):
  - The integer range [0,63] encodes to a single byte and is useful for e.g. command numbers, status codes, booleans, etc.
  - The integer range [0,16383] encodes to two bytes and is useful for e.g. line numbers, typical array indices, loop counter values, etc.
  - Short strings with length [0,31] are encoded to a single byte plus the string data. This is useful for typical filenames, property and variable names, etc.

Notes:

- When not sending a `duk_tval` value, integer number values must always be encoded as plain integers (not the IEEE double encoding).
- When parsing a `duk_tval` value, both plain integers and IEEE double values must be accepted. The plain integers map uniquely to IEEE doubles so there's no loss of information. Note that a negative zero must be represented as an IEEE double to preserve the sign.
- Fast integers (fastint) are not distinguish from ordinary numbers in the debugger protocol.
- Plain buffer values are represented explicitly, but buffer objects (Duktape.Buffer, Node.js Buffer, ArrayBuffer, DataView, and TypedArray views) are represented as objects. This means that their contents are not transmitted, only their heap pointer and a class number.

Endianness

As a general rule all values are serialized into network order (big endian). This applies to pointer values and IEEE doubles.

When pointers or IEEE doubles are part of buffer data they are encoded in whatever order they exist in memory. This means that e.g. bytecode dumped by DumpHeap will be represented as a buffer value with platform specific byte ordering. Changing the byte order would be quite awkward because the debugger code would need to be aware of the memory layout of specific buffer values.

Representing duk_tval values

The following dvalue types are used for `duk_tval` values:

- unused (undefined/unused/none): specific dvalue
- undefined: specific dvalue
- null: specific dvalue
- boolean: specific dvalues for `true` and `false`
- number: signed 32-bit integers can be represented with the simple integer dvalues (except negative zero), other numbers are represented as literal IEEE doubles
- string: specific dvalues for a few string lengths
- buffer: specific dvalues for a few buffer lengths
- object: represented as a pointer (dangerous when sent from debug client to debug target)
- pointer: represented as a pointer
- lightfunc: represented as a point and a flags field (dangerous when sent from debug client to debug target)

The notation `<tval: field name>` allows any dvalue compatible with a `duk_tval`. However, note that some values are dangerous when sent from the debug client to the target; e.g. it's possible to send a lightfunc value as an argument to PutVar, for instance, but it's easy to segfault unless you're very careful.

Request, replies, and notifications

A request has the format:

```
REQ <int: command> <0-N dvalues> EOM
```

A success response has the format:

```
REP <0-N dvalues> EOM
```

An error response has the **fixed format** independent of command:

```
ERR <int: error code> <str: error message or empty string> EOM
```

A notification has the format:

```
NFY <int: command> <0-N dvalues> EOM
```

Notes:

- Request and replies don't have a message ID: it is not necessary. Each peer is required to response to incoming requests in order, and every request is required to have a single success or error reply, so that replies can be reliably associated with previously sent requests. Note that reply messages may still be interleaved with requests and notifications sent by the peer in the other direction.
- Only requests/notifications have a command number: the reply messages are associated with a request/notification implicitly based on their order in the debug stream.
- The error response has a fixed format so that error handling can be uniform. There's a specific error code for "unsupported command" so that a debug client can easily check if new commands are supported and if not, fall back to something else.
- Right now Duktape only sends notifications, never requests, to avoid the need to track request/reply state.

Error codes

Code	Description
0x00	Unknown or unspecified error
0x01	Unsupported command
0x02	Too many (e.g. too many breakpoints, cannot add new)
0x03	Not found (e.g. invalid breakpoint index)

Handling of inbound requests

When either peer decides something unexpected happens, it can simply drop the transport. As soon as Duktape detects this, the debugger is automatically detached and normal execution resumes. This provides uniform handling for unexpected errors, and is appropriate behavior e.g. when:

- Invalid or insane dvalue formats are encountered. There's often no way to continue reliably in these cases.
- A parse error when a supported command is being handled. Such a situation would indicate that the peer is buggy or in an inconsistent state.

The exact error handling rules are not specified in great detail here, but there are a few rules which are important for extensibility:

- If a peer receives a request with an unsupported command number, it MUST send back an error reply indicating the command is supported, and MUST NOT drop the debug connection. This behavior is important so that a peer can try a command to see if it happens to be supported, and if not, fall back to some other behavior. As a result new commands can be added without always strictly bumping the protocol version, and it's possible to add optional or

custom, target specific commands and "probe" for them.
    - Right now this only applies to Duktape: Duktape never sends requests, only notifications.
- If a peer receives a notification with an unsupported command number, it MUST ignore the notification, and MUST NOT drop the debug connection. The reason is the same as for requests.
- If a supported command is parsed and there are additional dvalues before an EOM, the trailing dvalues MUST be ignored. This allows existing commands to be extended (in some cases) without assigning new command numbers or bumping the protocol version.

These simple rules are easy to implement and allow the protocol to be extended gracefully in a few common cases (but certainly not all).

Text representation of dvalues and debug messages

This is an informative convention only used in this document and in duk_debug.js dumps.

The Duktape debug client uses the following convention for representing dvalues as text:

- Marker bytes: `EOM`, `REQ`, `REP`, `ERR`, `NFY`.
- Integers: string coerced in the obvious way, e.g. `-123`.
- Strings are mapped 1:1 from a sequence of bytes (0x00...0xff) to a sequence codepoints U+0000...U+00FF and then JSON encoded. JSON encoding ensures that the result has no unescaped newlines. Standard JSON doesn't escape all of the codepoints U+0080...U+00FF which unfortunately looks funny (ASCII only serialization would be preferable).
- Other types are JSON encoded like in the JSON mapping, see below.

Debug messages are then simply represented as one-liners containing all the related dvalues (including message type marker and EOM) separate by spaces. This makes the text dump easy to read, cut-and-paste, diagnose, etc.

As an example, consider a reply whose payload consists of the string "touché", the integer 123, and the integer -321.

The string would be represented by Duktape internally as the UTF-8 sequence:

```
74 6f 75 63 68 c3 a9
```

The raw bytes of the reply message could be (with dvalues delimited by pipes):

```
02 | 67 74 6f 75 63 68 c3 a9 | c0 7b | 10 ff ff fe bf | 00
```

This would then be rendered as a text one-liner:

```
REP "touch\u00c3\u00a9" 123 -321 EOM
```

The odd string mapping is chosen to preserve the exact bytes used by the string inside Duktape. Note that some Duktape strings are intentionally invalid UTF-8 so mapping to Unicode is not always an option. This string mapping is also used to represent buffer data.

JSON mapping for debug protocol

The mapping described in this section is used to map debug dvalues and messages into JSON values. The mapping is used to implement a JSON debug proxy which allows a debug client to interact with a debug target using clean JSON messages alone without implementing the binary protocol at all.

JSON representation of dvalues

- Unused:

```
{ "type": "unused" }
```

- Undefined:

```
{ "type": "undefined" }
```

- Null, true, and false map directly to JSON:

```
null
true
false
```

- Integers map directly to JSON number type:

```
1234
```

- Any numbers that can't be represented without loss as JSON numbers (e.g. infinity, NaN, negative zero) are expressed as:

```
// data contains IEEE double in big endian hex encoded bytes
// (here Math.PI)
{ "type": "number", "data": "400921fb54442d18" }
```

- Strings are mapped like in the text representation, i.e. bytes 0x00...0xff map to Unicode codepoints U+0000...U+00FF:

```
// the 4-byte string 0xde 0xad 0xbe 0xef
"\u00de\00ad\00be\00ef"
```

This representation is used because it is byte exact, represents non-UTF-8 strings correctly, but is still human readable for most practical (ASCII) strings.

- Buffer data is represented in hex encoded form wrapped in an object:

```
{ "type": "buffer", "data": "deadbeef" }
```

- The message framing dvalues (EOM, REQ, REP, NFY, ERR) are not visible in the JSON protocol. They are used by `duk_debug.js` internally with the format:

```
{ "type": "eom" }
{ "type": "req" }
{ "type": "rep" }
{ "type": "err" }
{ "type": "nfy" }
```

- Object:

```
// class is a number, pointer is hex-encoded
{ "type": "object", "class": 10, "pointer": "deadbeef" }
```

- Pointer:

```
// pointer is hex-encoded
```

```
{ "type": "pointer", "pointer": "deadbeef" }
```

- Lightfunc:

```
// flags is a 16-bit integer represented as a JSON number,
// pointer is hex-encoded
{ "type": "lightfunc", "flags": 1234, "pointer": "deadbeef" }
```

- Heap pointer:

```
// pointer is hex-encoded
{ "type": "heapptr", "pointer": "deadbeef" }
```

JSON representation of debug messages

Messages are represented as JSON objects, with the message type marker and the EOM marker removed, as follows.

Request messages have a 'request' key which contains the command name (if known) or "true" (if not known), a 'command' key which contains the command number, and 'args' which contains remaining dvalues (EOM omitted):

```
{
    "request": "AddBreak",
    "command": 24,
    "args": [ "foo.js", 123 ]
}

{
    "request": true,
```

```
    "command": 24,
    "args": [ "foo.js", 123 ]
}
```

Reply messages don't have a command number, so they have a 'reply' key with a "true" value to allow the message type to be distinguished. Arguments are again in 'args' (EOM omitted):

```
{
    "reply": true,
    "args": [ 3 ]
}
```

Error messages are like replies, 'error' key has a "true" value, and 'args' contain the error arguments (EOM omitted):

```
{
    "error": true,
    "args": [ 2, "no space for breakpoint" ]
}
```

Notify messages have a 'notify' key with the notify command name (if known) or "true" (if not known), a 'command' key which contains the command number, and an 'args' for arguments (EOM omitted):

```
{
    "notify": "Status",
    "command": 1,
    "args": [ 0, "foo.js", "frob", 123, 808 ]
}

{
```

```
    "notify": true,
    "command": 1,
    "args": [ 0, "foo.js", "frob", 123, 808 ]
}
```

If an argument list is empty, 'args' can be omitted from any message.

The request and notify message contain both a request/notify command name and a number. The intent is to allow debug clients to use command names (rather than numbers). The command name/number is resolved as follows:

- If command name is present, look up the command name from command metadata. If the command is known, use the command number in the command metadata and ignore a possible 'command' key.
- If command number is present, use it verbatim if the name lookup failed.
- If no command number is present, fail.

Other JSON messages

In addition to the core message formats above, there are a few custom messages for debug protocol version info and transport events. These are expressed as "notify" messages with a special command name beginning with an underscore, and no command number.

When connecting to a debug target, a version identification line is received. This line doesn't follow the dvalue format, so it is transmitted specially:

```
{
    "notify": "_Connected",
    "args": [ "1 10199 v1.1.0-173-gecd806e-dirty duk command built from Duktape repo" ]
}
```

When a transport error occurs (not necessarily a terminal error):

```
{
    "notify": "_Error",
    "args": [ "some kind of error" ]
}
```

When the JSON connection is just about to be disconnected:

```
{
    "notify": "_Disconnecting"
}
```

JSON protocol line formatting

JSON messages are sent by encoding them in compact one-liner form and terminating a message with a newline (single LF character, 0x0a). (Note that the examples above are formatted in multiline format which is **not** allowed; this is simply for clarity.)

This convention makes is easy to read and write messages. Messages can be easily cut-pasted, and message logs can be grepped effectively.

Extending the protocol and version compatibility

The version identification line provides a protocol version number which is used to make incompatible changes to the debug protocol; the debug client is always assumed to conform to the target's debug protocol version.

It is also possible to extend the protocol without bumping the protocol version number in the following basic ways:

- Add a new command. If a command is not supported, the peer will send back a specific error indicating an unknown/unsupported command.
- Add trailing field(s) to a request, response, or notification. Once a peer has read and processed the fields it supports, it's required to skip to EOM, skipping unknown trailing fields. Some messages have a variable number of fields (e.g. a list of variable name/value pairs), in which case this approach may not be possible.

These extensions are made possible by (1) the ability to skip to EOM without understanding message contents, and (2) the processing requirements for unknown messages and unknown trailing dvalues.

As a general design rule, Duktape internals should be kept clean of version specific handling and workarounds. If a feature cannot be implemented cleanly in a compatible fashion, the protocol version should be bumped instead of adding parallel variants of commands or making other awkward compromises. It's important to keep the debugger code small and clean, so that code footprint is not compromised on the target.

Commands sent by Duktape

Status notification (0x01)

Format:

```
NFY <int: 1> <int: state> <str: filename> <str: funcname> <int: linenumber> <int: pc> EOM
```

Example:

```
NFY 1 0 "foo.js" "frobValues" 101 679 EOM
```

When nothing is executing (happens e.g. when duk_debug_cooperate() is called from outside of any Duktape activation) filename and funcname are undefined (the "undefined" dvalue is used) and pc/line are zero.

State is one of:

- 0x00: running
- 0x01: paused, debug client must resume

When execution state changes (e.g. from paused to running or vice versa) Duktape always sends a Status notification.

When Duktape is running with the debugger attached, it sends a status notification from time to time to keep the debug client informed of what file/line and function is being executed.

The rate of Status updates is automatically rate limited using a Date-based timestamp, so that Status updates are sent at most every 200ms when Duktape is running in normal or checked mode.

Print notification (0x02)

Format:

```
NFY <int: 2> <str: message> EOM
```

Example:

```
NFY 2 "hello world!\n" EOM
```

String output redirected from the `print()` function.

Alert notification (0x03)

Format:

```
NFY <int: 3> <str: message> EOM
```

Example:

```
NFY 3 "hello world!\n" EOM
```

String output redirected from the `alert()` function.

Log notification (0x04)

Format:

```
NFY <int: 4> <int: log level> <str: message> EOM
```

Example:

```
NFY 4 2 "2014-12-07T23:46:27.796Z INF foo: hello world" EOM
```

Logger output redirected from Duktape logger calls.

Commands sent by debug client

BasicInfo request (0x10)

Format:

```
REQ <int: 0x10> EOM
REP <int: DUK_VERSION> <str: DUK_GIT_DESCRIBE> <str: target info> <int: endianness> EOM
```

Example:

```
REQ 16 EOM
REP 10099 "v1.0.0-254-g2459e88" "Arduino Yun" 2 EOM
```

Endianness:

- 1 = little endian
- 2 = mixed endian (doubles in ARM "mixed" endian, integers little endian)
- 3 = big endian

Endianness affects decoding of a few dvalues.

Target info is a string that can be compiled in, and can e.g. describe the device type.

TriggerStatus request (0x11)

Format:

```
REQ <int: 0x11> EOM
REP EOM
```

Example:

```
REQ 17 EOM
REP EOM
```

Duktape will then re-send a status notify.

Pause request (0x12)

Format:

```
REQ <int: 0x12> EOM
REP EOM
```

Example:

```
REQ 18 EOM
REP EOM
```

If Duktape is already paused, a no-op. If Duktape is running, Duktape will check for incoming debug messages from time to time. When Duktape notices the pause request (which can take seconds) it will reply to the request, pause execution, and send a Status notification indicating it has paused.

Resume request (0x13)

Format:

```
REQ <int: 0x13> EOM
REP EOM
```

Example:

```
REQ 19 EOM
REP EOM
```

If Duktape is already running, a no-op. If Duktape is paused, it will exit the debug message loop associated with the paused state (where control is fully in the hands of the debug client), resume execution, and send a Status notification indicating it is running.

StepInto request (0x14)

Format:

```
REQ <int: 0x14> EOM
REP EOM
```

Example:

```
REQ 20 EOM
```

```
REP EOM
```

Resume execution and pause when execution exits the current line. If a function call occurs before that, go into the function and pause execution there.

StepOver request (0x15)

Format:

```
REQ <int: 0x15> EOM
REP EOM
```

Example:

```
REQ 21 EOM
REP EOM
```

Resume execution and pause when execution exits the current line. Don't pause on function calls occuring before that.

StepOut request (0x16)

Format:

```
REQ <int: 0x16> EOM
REP EOM
```

Example:

```
REQ 22 EOM
REP EOM
```

Resume execution and pause when execution exits the current function. This can happen because:

- The current function returns, in which case execution resumes in the calling function.
- The current function, or any function called by it, throws an error which is not caught before it unwinds past the current function. Execution resumes in the error catcher.

ListBreak request (0x17)

Format:

```
REQ <int: 0x17> EOM
REP [ <str: fileName> <int: line> ]* EOM
```

Example (two breakpoints):

```
REQ 23 EOM
REP "foo.js" 102 "bar.js" 99 EOM
```

AddBreak request (0x18)

Format:

```
REQ <int: 0x18> <str: fileName> <int: line> EOM
```

```
REP <int: breakpoint index> EOM
```

Example:

```
REQ 24 "foo.js" 109 EOM
REP 3 EOM
```

If there's no space for more breakpoints, a "too many" error is sent:

```
REQ 24 "foo.js" 109 EOM
ERR 2 "no space for breakpoint" EOM
```

DelBreak request (0x19)

Format:

```
REQ <int: 0x19> <int: index> EOM
REP EOM
```

Example:

```
REQ 25 3 EOM
REP EOM
```

If an invalid index is used, an error reply is sent.

GetVar request (0x1a)

Format:

```
REQ <int: 0x1a> <str: varname> EOM
REP <int: 0/1, found> <tval: value> EOM
```

Example:

```
REQ 26 "testVar" EOM
REP 1 "myValue" EOM
```

PutVar request (0x1b)

Format:

```
REQ <int: 0x1b> <str: varname> <tval: value> EOM
REP EOM
```

Example:

```
REQ 27 "testVar" "newValue" EOM
REP EOM
```

GetCallStack request (0x1c)

Format:

```
REQ <int: 0x1c> EOM
REP [ <str: fileName> <str: funcName> <int: lineNumber> <int: pc> ]* EOM
```

Example:

```
REQ 28 EOM
REP "foo.js" "doStuff" 100 317 "bar.js" "doOtherStuff" 210 880 EOM
```

GetLocals request (0x1d)

Format:

```
REQ <int: 0x1d> EOM
REP [ <str: varName> <str: varValue> ]* EOM
```

Example:

```
REQ 29 EOM
REP "x" "1" "y" "3.1415" "foo" "bar" EOM
```

List local variable names from current function (the internal `_Varmap`).

Note

The local variable list doesn't currently include dynamically declared variables introduced by e.g. eval(), or variables with

a dynamic scope like the catch variable in try-catch. This will be fixed in future versions.

Eval request (0x1e)

Format:

```
REQ <int: 0x1e> <str: expression> EOM
REP <int: 0=success, 1=error> <tval: value> EOM
```

Example:

```
REQ 30 "1+2" EOM
REP 0 3 EOM
```

The eval expression is evaluated as if a "direct call" to eval was executed in the position where execution has paused. A direct eval call shares the same lexical scope as the function it is called from (an indirect eval call does not). For instance, suppose we're executing:

```
function foo(x, y) {
    print(x);  // (A)
    print(y);  // (B) <== paused here (before print(y))
}

foo(100, 200);
```

and you'd eval:

```
print(x + y); y = 10; "quux"
```

The Eval would execute as if the code had been:

```
function foo(x, y) {
    print(x);
    eval('print(x + y); y = 10; "quux");
    print(y);
}

foo(100, 200);
```

so that the Eval statement would:

- Print out 300 (using print).
- Assign 10 to `y` so that statement B would then print 10 (instead of 200).
- The final result of the eval would be the string `"quux"`, which would then be shown in the debug client UI.

When Eval is requested from outside any Duktape activation, e.g. while doing a duk_debugger_cooperate() call, there is no active Ecmascript activation so that a "direct" Eval is not possible. Eval will then be executed as an indirect Eval instead.

Current limitations:

- Can get stuck in an infinite loop.
- The debug code runs inside an actual `eval()` call which affects the call stack. For example, `Duktape.act()` will see the additional stack frames.

Detach request (0x1f)

Format:

```
REQ <int: 0x1f> EOM
REP EOM
```

Example:

```
REQ 31 EOM
REP EOM
```

Request that Duktape detach the debugger. Duktape requests the user transport code to close the transport connection, and then resumes normal execution.

DumpHeap request (0x20)

Format:

```
REQ <int: 0x20> EOM
REP <dvalues> EOM
```

Example:

```
REQ 32 EOM
REP <dvalues> EOM
```

Dump contents of the entire Duktape heap. The format of the heap dump is somewhat complicated; see `duk_debugger.c` for the format.

This is used to implement a debugger UI feature where you can download a JSON dump of the heap state for analysis.

Note

This command is somewhat incomplete at the moment. It will be useful to implement a heap browser, and will probably be completed together with some kind of UI.

GetBytecode request (0x21)

Format:

```
REQ <int: 0x21> EOM
REP <int: numconsts> (<tval: const>){numconsts}
    <int: numfuncs> (<tval: func>){numfuncs}
    <str: bytecode> EOM
```

Example:

```
REQ 33 EOM
REP 2 "foo" "bar" 0  "...bytecode..." EOM
```

Bytecode endianness is target specific so the debug client needs to get target endianness and interpret the bytecode based on that.

Note

This command is somewhat incomplete at the moment and may be modified once the best way to do this in the debugger UI has been figured out.

"debugger" statement

Ecmascript has a debugger statement:

```
a = 123;
debugger;
a = 234;
```

The E5 specification states that:

> Evaluating the DebuggerStatement production may allow an implementation to cause a breakpoint when run under a debugger. If a debugger is not present or active this statement has no observable effect.

Other Ecmascript engines typically treat a debugger statement as a breakpoint:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger
- http://msdn.microsoft.com/en-us/library/ie/0bwt76sk%28v=vs.94%29.aspx
- http://blog.katworksgames.com/2012/09/27/debugger-statement-makes-javascript-development-easier/

Duktape interprets it as a breakpoint too, i.e. execution is paused if a debugger statement is encountered while a debug client is attached. This allows breakpoints to be set even in anonymous eval code (though there will be no access to source code).

Implementing a debug transport on top of a packet-based

transport

Implementing a debug transport over a packet-based lower level protocol is essentially the same problem as forwarding a TCP stream or a virtual serial link over the packed-based protocol. There is very little Duktape specific in doing so, and the problem is quite well understood. This section provides some pointers.

Basic issues

- You'll need a mechanism to reliably send and receive arbitrary chunks of data with no reordering or duplication. This mechanism is needed both for the target and the debug client.
- If buffering is an issue you may need to implement a flow control mechanism. Usually buffering is only an issue on the debug target, so one way flow control is usually enough.
- To ensure data chunks sent by the debug target are reasonably sized, you may need to coalesce debug transport writes made by Duktape and use "write flush" to flush out pending bytes when no more data will be sent. Alternatively you could use a timer, similarly to what TCP does.

If you also implement your own debug client you need to parse the debug stream from the data chunks received, e.g. with trial parsing:

- Read an incoming data chunk and append it to an input byte buffer.
- Trial parse for debug messages until no more complete messages can be parsed. Then wait for next inbound data chunk.
- **Because the boundaries of debug messages are not guaranteed to align with the read/write calls Duktape makes into the transport implementation, you should not try to match debug messages to the data chunks sent/received by your transport implementation!**

Coalescing writes example

- Maintain a buffer BUF of max N bytes for outbound writes.
- For each Duktape transport write call:
  - If the write data fits into BUF, append it. If not, append as many bytes as fit in the remaining BUF space (partial write).
  - If the buffer is now full (N bytes), send and empty the buffer.
  - Return value to Duktape indicates how many values were consumed, i.e. appended to BUF.
- For each Duktape transport write flush:
  - If there are bytes in BUF, send and empty the buffer.
  - Note that you can rely on Duktape performing a write flush before it finishes writing and e.g. blocks on read or resumes execution. Write flushes may also happen at other times. **Don't assign any other meaning to the flushes, e.g. a write flush is not guaranteed to match debug message boundaries!**

One-way flow control example

A simple one-way flow control mechanism to ensure a debug target can be implemented with a fixed inbound buffer of MAXBUF bytes (MAXBUF is something small like 256):

- The debug client maintains two byte counts:
  i. SENT indicates how many bytes have been sent since the start of the debug connection.
  ii. ACKED indicates how many bytes the debug target has confirmed to have consumed. SENT - ACKED is the number of bytes potentially in the target input buffer.
- The debug client then knows that the target can buffer at least MAXBUF - (SENT - ACKED) bytes, so that it's free to send that amount.
- When the debug target receives data chunks from the debug client, it:
  - Appends the data chunk to an inbound data buffer. There should always be space for the data if the debug client behaves correctly.
- When Duktape calls the debug transport read callback:

    - Consume bytes from the inbound data buffer.
    - Send a transport specific notification to the debug client, updating the ACKED byte count (= number of bytes consumed by Duktape read calls).

Because Duktape performs a lot of small reads, it may be useful to:

- In the debug transport read callback:
    - Don't send a notification for the updated ACKED byte count unless the change to a previously sent value is large enough.
- Rely on the debug transport "read flush" indication:
    - When received, always send a notification for the updated ACKED byte count.

There are many other options too, for example, send an updated ACKED byte count when:

- Receiving bytes from the debug target.
- When Duktape reads bytes, only send an updated ACKED byte count when the read is made from a completely full input buffer (i.e., the debug client is currently not sending any data until we notify it we have space).

Implementation notes

Overview

This section contains some implementation notes on the Duktape internals.

Duktape debugger support is optional and enabled with a feature option. The bytecode executor interrupt feature is also mandatory when debugger support is enabled.

Source files

The debugger support is implemented almost entirely in the following files:

- `duk_js_executor.c` : checked execution, breakpoints, step into/over, interfacing with debugger message loop
- `duk_hthread_stacks.c` : step out handling
- `duk_debugger.c` : debug transport, debug command handling
- `duk_api_debug.c` : debugger API entrypoints

Attaching and detaching a debugger to a heap

When user code attaches a debugger using `duk_debugger_attach()` , Duktape updates the `duk_heap` state to reflect that a debugger is attached, store the callbacks etc.

The debugger operates on a Duktape heap level, as other options seem to lead to confusing outcomes. For instance, if a debugger were attached to a single thread breakpoints would only be triggered by that thread. Even so, when a breakpoint was triggered, the whole heap would be paused because there's no way to pause a single thread and resume execution of others.

Execution modes, executor interrupt, and "restart_execution"

Perhaps the most critical capability needed to implement a debugger is to have an efficient way of detecting active breakpoints, trigger on a breakpoint, and implement stepped execution. These are implemented in the Duktape bytecode executor as follows.

Debugger support relies on the executor interrupt feature, which provides the ability to interrupt bytecode execution periodically or after every bytecode instruction. This mechanism is used to implement three conceptual modes of execution:

- **Normal**: bytecode executor executes at full speed, calling into the executor interrupt once in a while. When in the

interrupt, we peek for debug client messages (this allows an out-of-the-blue pause for instance), execution timeout etc.

- **Checked**: bytecode executor executes opcodes one at a time, calling into the executor interrupt before every instruction. The interrupt detects line transitions, checks if any breakpoints or stepping related conditions are triggered, and peeks (but doesn't block waiting) for debug client messages.
- **Paused**: bytecode executor calls into executor interrupt, and the executor interrupt processes debug client messages until the debug client issues some control flow related command like step over/into/out or resume. Execution is under complete control of the debug client.

The "paused" mode is concretely implemented in the executor interrupt simply by processing debug messages until some kind of resume/detach command is encountered.

The "checked" mode is implemented by careful management of the interrupt counter. This is important so that no additional checks are introduced into the executor fast path: only a single interrupt counter check is needed. When execution is restarted, the need for checked execution is detected (e.g. there are active breakpoints or stepping is active) and the interrupt counter is configured to trigger an interrupt before any opcodes are executed. If we need to remain in checked mode, the interrupt handler will again configure the interrupt counter to ensure only one opcode is executed before again returning to the interrupt handler.

The "normal" execution mode is similar but the interrupt counter is configured into a higher value (e.g. interrupt every hundred thousand opcodes) when returning to the bytecode executor.

The `restart_execution:` label in the bytecode executor is an important control point. It is called whenever the bytecode executor is about to start executing a new activation, but can also be called explicitly e.g. when debug commands have adjusted breakpoint state. The "restart execution" operation does a lot of important things:

- It checks for debugger attached/detached state. If detached, all other debugger related checks are skipped.
- It checks for active breakpoints in the current function, and writes out the active breakpoint list to make breakpoint trigger checks faster in the executor interrupt.

- It checks for active stepping state. Both step into and step over require some handling.
- It checks for paused state too. In some cases a "paused" flag can be set outside the bytecode executor. For example, when doing a "step out", the callstack unwinding code sets a "paused" flag when unwinding the activation we're stepping out of. We detect this only when "restart execution" is called the next time.
- Ultimately, it decides whether execution should proceed in checked mode or normal mode.

After execution proceeds normally, with the help of the executor interrupt mechanism and the interrupt handler. The execution mode can only changed by the interrupt handler (e.g. if it starts setting the interrupt counter to a higher value) or if "restart_execution" is invoked again.

From the bytecode executor perspective the integration is quite simple:

- "restart_execution" does a lot of debugger processing as part of setting up execution.
- The interrupt counter mechanism is used to call into the interrupt handler, and the actual opcode executor doesn't have to worry about the rest.

Stepping and pausing

The following internal heap level state is needed:

- Pause state: heap wide flag indicating we need to talk with the debug client until it gives us as a permission to continue.
- Step state: heap wide, tracks currently active "step into", "step over", or "step out" state.

The step state is rather tricky:

- Step over: track the original thread, activation index, and starting line. Execute in checked mode until starting line has changed; then pause. If we call into other functions, the state is kept and we'll pause once we return and the line number has changed.

- Step into: track the original thread, activation index, and starting line. Execute in checked mode until starting line has changed. If we call into another function, we need to pause when entering it.
- Step out: track the original thread and activation index (starting line does not matter). Execute in normal mode (unless there are breakpoints, of course). If the activation is unwound for any reason, enter paused mode. This means that if an error is thrown, we resume execution in the catcher. Step out handling is concretely implemented as part of call stack unwinding, which differs completely from how other step commands are implemented.

  A coroutine yield does not trigger a "step out" because the callstack is not unwound.

Step over/into state is checked in executor "restart execution" operation.

Breakpoints

Breakpoints are maintained as a heap level file/line list. When the bytecode executor does a "restart execution" operation it rechecks the breakpoint list and figures out which breakpoints are active; the active breakpoint list is recorded into the heap state too. Whenever breakpoint state may have changed, e.g. as a result of executing debug commands, the bytecode executor must go through a "restart execution" operation so that breakpoints are properly re-checked and activated.

If there are one or more active breakpoints, execution resumes in checked mode. If no breakpoints are active (and there's no other reason to be in checked mode) execution resumes in normal mode. This is important to maximize execution performance when breakpoints are active but outside the currently executing function.

One key problem in figuring out the active breakpoints is how to handle inner functions. This is covered in a separate section below.

Breakpoints are handled directly by Duktape to make them reasonably efficient. Another design alternative would be to have an API or a protocol mechanism for stepped execution so that user code could implement breakpoints on its own.

This would be more flexible than an integrated breakpoint mechanism, but also much slower.

There are many design alternatives to defining a breakpoint using a file/line pair. The current file/line approach is intuitive but means that:

- There's no way to break in the middle of a single line, e.g. for one-line functions. This also affects minified Ecmascript code.
- There are potentially multiple Ecmascript function instance (i.e. `duk_hcompiledfunction` objects) that have been created from the same spot. The breakpoint will match all of them.

Line transitions

It might seem at first that a line-to-PC conversion primitive would be needed so that a line number could be translated into a PC for an active breakpoint. However, such an approach doesn't really work, for several reasons, discussed below.

Multiple instructions can be generated from a single line so that there are several instructions with the same line number in the typical case. The opcodes mapping to a certain line number can also be scattered around the code (not necessarily in a linear or localized fashion), e.g. for flow control constructs. Something like the following is entirely possible, and normal:

```
PC      Line
--      ----

50      98
51      99
52      100   <--
53      100   <--
54      100   <--
```

```
55      100   <--
56      102
57      103
58      103
59      104
60      105
61      100   <--
```

A breakpoint may also be targeted on a line number which doesn't have any matching bytecode instructions. This can happen trivially when a breakpoint is assigned to an empty line, but can also happen non-trivially when the line numbers in the generated bytecode are off by one or otherwise unintuitive. The expected behavior in this case is often that the breakpoint should be triggered when we transition to the breakpoint line *or* over it. In more concrete terms:

```
(prev_line < break_line) AND (curr_line >= break_line)
```

Implementing breakpoints in terms of line transitions also solves another related issue: once we hit a breakpoint on a certain line, how to implement "step into" / "step over"? Stepping away from the breakpoint line means we need to execute bytecode instructions until current line changes to a value different than the breakpoint line. Note that this is not necessarily the next line or even a higher line number because control flow can maka a jump backwards.

So, right now Duktape implements breakpoints as follows:

- When one or more breakpoints is active, the bytecode executor enters checked execution. In checked execution the bytecode interrupt mechanism is invoked before every opcode. Checked execution is carefully avoided when at all possible, to ensure breakpoints don't slow down performance when they don't need to.
- The interrupt mechanism tracks line information (previous line, current line) so that it can detect line transitions. This means Duktape will do a pc-to-line for every opcode executed. This is currently not optimized and will consult the pc-to-line bitstream every time; see future work for notes on how this can be improved in the future.
- Breakpoints and stepping are checked when a line transition occurs, i.e. when `prev_line != curr_line`.

Inner functions and breakpoints

A breakpoint should only be active in the innermost function in the source code. Consider for example:

```
1  function foo() {
2      print('foo 1');
3      function bar() {
4          print('bar 1');
5      }
6      print('foo 2');
7      bar();
8  }
9  foo();
```

Suppose execution was currently at line 2, and a breakpoint was added for line 4. What happens when you single step?

In a naive implementation the executor considers the line 4 breakpoint to be active for the foo() activation, and when it detects a line transition from line 2 to line 6, the breakpoint is triggered. Execution stops at line 6 before printing "foo 2".

To avoid this, a breakpoint is always associated (only) with the innermost function where it appears. This can be quickly detected by tracking the line range (smallest and largest line number) for each function. One can then determine active breakpoints for a function FUNC as follows:

- If breakpoint has a different filename, reject.
- If breakpoint has line number is outside FUNC line range, reject. (For foo() line range would be 1-8 and for bar() line range would be 3-5.)
- Loop through all inner functions IFUNC of FUNC:
  - If breakpoint line number is inside IFUNC, reject. IFUNC is considered to "capture" the breakpoint.
- Accept breakpoint as active for FUNC execution.

Avoid nested message writing

Consider the following scenario:

- Debug client requests for local variable names and values using a hypothetical GetLocalVarsAndValues request.
- Duktape starts processing the request, streaming out a REP marker, followed by variable names and values.
- One of the variable values is a getter, and the request handler just uses a naive read to get the variable value, so that the getter is invoked.
- The getter calls `print()` which gets forwarded to the debug client. The `print()` handler writes a notification message containing the print data.
- This notification ends up in the middle of the GetLocalVarsAndValues response, corrupting the debug stream.

Such nested debug messages must be avoided at all times. Some ways to achieve this:

- If the debug command only deals with a single value (and not a list of values), read and coerce any values into safe form before streaming out the response.
- As a general rule favor side effect free debug commands, e.g. read values without invoking getters.
- For unsafe primitives that may have side effects, favor debug commands that just handle a single value (instead of an arbitrarily long list of values). Such a primitive is easier to implement safely because it doesn't need to buffer a potentially unlimited list of safely obtained values before starting to write out the response.
- As a concrete example, the GetLocalVarsAndValues could be fixed either by:
    i. Changing it so that it doesn't invoke accessors.
    ii. Changing it to return only a list of variable names, and adding a separate primitive to get the local variable value (GetLocalVar). This primitive can invoke getters, but it must do so before it starts to stream out the response. Note that request pipelining allows local variables to read in two round trips: first read the variable names, then issue reads for every variable name in a big set of pipelined requests.

This issue affects various things here and there:

- If GC is invoked, it might be tempting to emit a GC notification from inside mark-and-sweep code. This would be very unsafe because GC can easily be invoked by any operation involving the value stack.

Design goals

This section provides some notes on goals behind the debugger design (this is not a comprehensive list).

Quick integration with a custom target

It should be possible integrate debugging support into a custom target very quickly, e.g. in one day.

- This should be achievable with the current solution. One needs to implement a custom transport into both the target device and duk_debug.js and can then use the debugger web UI to debug the target.

Minimize fragmentation of debug solutions

The debugger architecture should ensure that improvements for Duktape debugging capabilities are shared between users. Ideally debug clients developed for different environments could be mixed and matched.

- This is the main reason why a debug protocol is used as the basis of the design instead of a debug API. A debug API would mean every user would need to define their own debug protocol, which would fragment both the debug protocol and, as a consequence, the debug clients.
- This goal is achieved to a large extent: any debug client should be able to talk with any target. However, there may be need to adapt a transport mechanism so it's not completely automatic.

Transport neutrality

The debug protocol should be transport neutral to support embedding in very different environments and communication links (Wi-Fi, Bluetooth, serial, embedding into protocols like AllJoyn, etc).

- Concrete solution is to use assume a reliable (TCP-like) byte stream, with user code providing the concrete transport.

Transport bandwidth

The debugger must work with slow transports, e.g. slow serial links.

- This is the reason a binary protocol is used: it's reasonably compact with no compression. Compression is a possible solution but it is not preferable for very low memory devices (memory overhead).

The debugger must work with high latency transports (hundreds of milliseconds).

- This is the reason why request pipelining is used: pipelining allows multiple commands to be sent, reducing blocking round trip waits.
- Pipelining allows debug commands to be built from small, simple operations with minimal additional latency (compared to a synchronous request/reply model).

Human readable protocol

It would be nice for the protocol to be human readable, e.g. plain text.

- This is currently not achieved as the debug protocol is binary.
- A binary protocol is used at the moment because it is more compact and has a smaller code footprint than parsing a text-based protocol. Note that such parsing would need to be done without GC impact or other side effects so existing Ecmascript mechanisms (like number parsing) cannot necessarily be used as is.

Code footprint

Debugger support should be optional because it has a significant footprint.

It should be possible to enable debugger support even for very low memory devices (e.g. 256kB flash).

- At the moment the additional code footprint for debugger support is around 10kB.

Memory (RAM) footprint and minimal churn

The debugger implementation should consume a minimal amount of RAM on top of what the debug commands themselves need.

- Fixed allocations are preferable to variable allocations for low memory devices.

Debugger commands should avoid disturbing Duktape internal state. For instance, if a debug command requested a dump of the Duktape heap, the command should cause no changes to the heap during serialization of the response. Concretely this means that:

- It must be possible to read and write debug messages without doing any memory allocations that can cause a GC. This rules out, among other things, pushing values on the value stack and interning strings. Memory allocations can be done using raw calls to allocation callbacks, but it's be preferable to be able to avoid memory allocations altogether.
- Note that it is *not* a requirement that all debug commands be implemented without side effects. For instance, reading a variable may invoke a getter or use some internal mechanisms with side effects. The goal is simply that it should be *possible* to write some debug commands that are side effect free if that is necessary.

Large and variable sized buffers for parsing inbound messages or constructing outbound messages should be avoided. These would be very problematic on low memory devices.

- This goal is an important reason why the debug protocol uses a stream transport. A stream transport allows e.g. the whole heap to be serialized with no variable sized output buffering: values are simply streamed out during the heap walk with a fixed streaming buffers.
- This goal is also one reason why the debug protocol is binary instead of e.g. JSON: JSON parsing would introduce significant memory churn if the current parser were used. Adding a separate parser for debugging would be wasteful.

Performance

When a debugger is not attached (but debugger support is compiled in), performance should be as close to normal as possible.

When a debugger is attached but there are no active breakpoints, performance should be as close to normal as possible.

Performance with active breakpoints is not critical, but still matters on slow targets so that timing sensitive applications have a chance of working properly when debugged.

Miscellaneous design notes

Some design notes on miscellaneous issues, rejected alternatives, etc.

Debug commands instead of debug API

Instead of a debug protocol Duktape could provide a set of API primitives to allow user code to implement a debugger on its own. This would have several downsides:

- There would need to be a lot of new public API primitives with deep access to Duktape internals. Such an API

would be a major maintenance issue going forwards: when Duktape internals change, there would still be old API promises to keep. A debug protocol can hide the internal details more effectively.
- Every user application with a need for debugging would need to implement their own debug protocol: there would be no standard debug commands, just raw API calls which can be used to implement a debugger. Every Duktape debugger integration would be different.

Impact of being an embeddable interpreter

Embedded model means there is no standard launch like there is for a JVM for instance. The debugger needs to connect to a running instance, and the launching of the instance is up to the user. There may also not be easy access to source code: the way it is loaded is up to the user, and some of the source code is given from C code, perhaps programmatically.

It's up to the application to decide when the debugger is attached. For instance, a debugger may attached on startup (some kind of "reboot and debug" mode) or only when debugger is attached at runtime.

Packet based protocol

The debug transport could be based on delimited debug packets. Both V8 and Spidermonkey debug protocols are (JSON) packet based.

In a packet based protocol an inbound message needs to reside in memory to be processed. Similarly outbound messages are formed as full packets before being sent. This works poorly with low memory devices because it is difficult to limit the maximum debug packet size:

- For example, even if a debug packet only contained a single string (perhaps an eval result), the size of the string may vary widely. If debug packets have an upper size limit, it's quite easy to get into a situation where values that easily fit into memory cannot be sent over the debug protocol

- One can alleviate this problem by doing fragmented reads, i.e. the debug protocol allows the debug client to read in a string in chunks. This has string life cycle issues, and such a fragmentation protocol is in fact emulating a stream transport in a crude way.
- A similar approach is needed for serializing object values, and potentially many other debug commands, which is very awkward from a protocol design perspective.

Stream protocol without request/response framing

The debug protocol could also be a stream protocol with no request/response framing. This works poorly when either party may initiate messages without lock step. For example, if debug client sends a request and the target sends a notification, how can the debug client know that the bytes it receives are not a response but an unrelated notification?

Some framing is needed to at least separate responses from other messages.

Pipelining vs. asynchronous messages

The current design is to allow pipelining of requests: each request has a single reply (or error) and requests are never reordered. There is no need for request/reply identifiers in this model.

Another design would be to allow each party to send responses to incoming commands in an arbitrary order (asynchronously). This would be useful if some operations took a long time and could be handled in the background while more urgent operations could be processed in the meantime.

In practice this is difficult to implement especially on the debug target, and would require more state tracking. It would also make it more difficult to send multiple requests (compared to pipelining) because there would be no guarantee of their completion order.

Untyped debug message encoding

One alternative tried was to use untyped encoding for debug messages, i.e. debug client and target both know what exact data messages are intended to have, so there is no need to tag a value e.g. as an integer or a string.

This would be efficient but difficult to extend in a compatible fashion. Instead, the debug protocol would need hard versioning for every minor change and the debug client would need to support all protocol variants. This is not necessarily a showstopper though as the debug client will need to have version awareness anyway.

Variable size integer encoding

The debug protocol exchanges a lot of small and large integers. The extended UTF-8 encoding was used first which is consistent with other variable length integer encoding in Duktape.

However, when the current tag initial byte (IB) was added, it became very natural to use the tag byte to encode small integers and to encode the byte length of larger integers.

Accessors and proxies vs. variable get/set

- Triggering setters / getters may not be desirable.
- Perhaps return value like Object.getOwnPropertyDescriptor(), and allow debug client to invoke the getter if necessary?
- Access proxy and target separately?

Other debugger implementations

Overview

Both V8 and Spidermonkey use a packet based debug protocol with much of the protocol formatted in JSON. Although

this is quite an intuitive approach, Duktape uses a stream based binary protocol to avoid the memory churn related to using JSON, and to better support very low memory devices where forming complete debug messages in memory would be problematic.

Chrome/V8

Chrome/V8 uses a packet based debug protocol where each packet is a JSON message:

- https://code.google.com/p/v8-wiki/wiki/DebuggerProtocol

Also see:

- https://code.google.com/p/chromedevtools/wiki/ChromeDevToolsProtocol
- https://developer.chrome.com/devtools/docs/javascript-debugging

Firefox

Mozilla uses a packet based debug protocol where packets are either JSON or binary blobs. It can be mapped to a stream:

- https://wiki.mozilla.org/Remote_Debugging_Protocol_Stream_Transport

Also see:

- https://developer.mozilla.org/en/docs/Debugging_JavaScript

Eclipse

An Eclipse debugger could be implemented using the Duktape debugger protocol. Some resources for that:

- http://www.eclipse.org/articles/Article-Launch-Framework/launch.html
- http://www.eclipse.org/articles/Article-Debugger/how-to.html

Known issues

Valgrind uninitialized byte(s) warning

You may get the following when doing a DumpHeap:

```
==17318== Syscall param write(buf) points to uninitialised byte(s)
==17318==    at 0x5466700: __write_nocancel (syscall-template.S:81)
==17318==    by 0x427ADA: duk_debug_trans_socket_write (duk_debug_trans_socket.c:237)
==17318==    by 0x403538: duk_debug_write_bytes.isra.11 (duk_debugger.c:379)
==17318==    by 0x4036AC: duk_debug_write_strbuf (duk_debugger.c:463)
[...]
```

When unpacked duk_tval is in use, all bytes of a duk_tval are not necessarily set when a certain value is written into the duk_tval. This is not a safety issue because Duktape won't read or use the uninitialized bytes in ordinary situations. However, the uninitialized bytes in the 'data' area of a compiled function will be written out by DumpHeap as is, causing the above (harmless) valgrind gripe.

Future work

Error handling

Add error handling wrappers to debug code. For instance, if we run out of memory, detach automatically as a recovery

measure?

Currently unsafe behavior may be triggered by internal errors (e.g. out of memory) or, for instance, a getter error triggered by GetVar.

Fast pc-to-line for checked execution

During checked execution we need to figure out the line number for the current PC so that line transitions can be tracked accurately. Right now the pc-to-line bitstream is consulted statelessly each time, which is slow (but only affects checked execution, i.e. when there's an active breakpoint for the current function).

There are several ways to make this faster:

- Cache the pc-to-line conversion state. If the PC increases by one, we can almost always just decode a single line delta from the bitstream which is very efficient and requires no data format changes.
- When entering checked execution, create an unpacked pc-to-line array so that lookups can be done as simple array lookups.
- When debugging is enabled, store pc-to-line conversion information as a plain array in general. This has a memory footprint impact for all functions, even when a debugger is not attached (but Duktape debugger support is compiled in) so this approach is not very desirable.
- Emit explicit line transition opcodes. This has a memory and performance impact, even when a debugger is not attached, so this approach is also not very desirable.

Fix debugger statement line handling

When executing a "debugger;" statement, the debugger currently pauses after PC has already been incremented. In other words, the debugger is paused "after" the statement has been executed. In the debugger UI this looks like execution had paused on the line following the debugger statement.

Improve compiler line number accuracy

The Ecmascript compiler assigns line numbers to bytecode opcodes emitted, and doesn't always do a perfect job in doing so. There are a few cases where the line number for a statement can be off by one (matching a previous statement) which looks funny in the debugger UI.

The underlying issue is that the compiler emits bytecode opcodes both when the active token is in the "previous token" and the "current token" slot. Expression parsing usually has the active token in the previous token slot, while statement parsing (especially when parsing the initial keyword) has the active token in the current token slot. This needs some reworking to be fixed properly.

Source code

Source code handling is currently outside of Duktape scope, and we simply assume that the proper source file can be located based on a "fileName" property of a running function.

There are many future options:

- Download from target device (same as where code was originally loaded)
- Store source when compiling in debug mode, possibly using some trivial compression to reduce the memory impact
- Identify source code text using a hash computed on the target, so that the corresponding source can be located more reliably

Source maps

It's a common practice to minify Javascript code. Line number information is often lost in the process, and this makes the code difficult to debug for a variety of reasons:

- Source code readability is poor
- Breakpoint mechanisms targeting file/line work very poorly

Source maps record the original line number information:

- http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

If Duktape supported source maps, a source map could be taken into account during compilation and function pc-to-line mapping could refer to the original unminified source code which would be much more debugger friendly.

More flexible pausing

Various triggers for pausing could be added:

- Pause on function entry/exit
- Pause on next statement
- Pause on error about the be thrown
- Pause on yield/resume
- Pause on execution timeout

More flexible stepping

Additional stepping parameters could be implemented:

- Step for N bytecode instructions
- Step for roughly N milliseconds

Dynamically declared variables in local variable list

The local variable list returned by GetLocals does not include dynamically declared variables, or variables with a scope smaller than the entire function:

```
function test() {
    var foo = 123;   // 'foo' is included

    eval('var bar = 321');  // 'bar' is not included

    try {
        throw 'foo';
    } catch (e) {
        // 'e' is not included
    }
}
```

This should be fixed so that the locals include dynamic variables too. This is especially important for try-catch.

The Eval command can read/write dynamic variables too, so the current workaround is to use Eval. For instance, in the catch clause, Eval `"e"` to read the error caught.

Expression dependent breakpoints

Pause when an expression evaluates to a truthy value.

Watch expressions

Watch expressions are currently implemented by the debug client using the Eval command.

For example, the debugger web UI implements automatic eval for a single expression. The expression is automatically evaluated when Duktape becomes paused. This is easy to extend for multiple watch expressions.

Notifications on internal events

Send a notification when interesting internal events occur, like:

- Ordinary GC
- Emergency GC
- Thread creation
- Thread destruction
- Execution timeout

These must be implemented very carefully. For instance, if we're currently in the process of responding to some debug command (say, "get locals") and GC is triggered, the GC notify cannot be sent inline from the mark-and-sweep code because it might then appear in the middle of the "get locals" response. Instead, events need to be flagged, based on counters, or queued.

Reset command

Having a shared command to reset/reboot a target might be useful. It would require either a specific API callback or some form of command integration through the Duktape debugger API.

Many targets have a management protocol that can be used to implement a reset, so that it doesn't necessarily have to be in the debug protocol.

Possible new commands or command improvements

- Add callstack index to variable read/write
- Add callstack index to Eval
- More comprehensive callstack inspection, at least on par with what a stack trace provides

- Shallow value inspection, ability to inspect internal stuff like internal prototype but also state of object property tables, etc.
  - Should be debug client driven and shallow so that the client can query the value graph on its own. This avoids the need to handle reference loops on the target.
  - Deep querying needs an object reference: the client can address objects with a heap pointer. It is then critical that no side effects can be triggered during the traversal to avoid invalidating the heap pointers. All debug commands involved in traversal must be side effect free and perform no allocations.
- Resume with error, i.e. inject error
- Enumerate threads in heap
- Enumerate all objects in heap
- Status for success/failure of PutVar
- Error handling for PutVar
- Avoid side effects (getter invocation) in GetVar

Application specific messages

It would be useful to be able to send application specific commands to the target. For instance, a debug client may want to query the target's memory allocator state or file system free space.

Such messages can of course be exchanged out of band, outside the Duktape debugger protocol, and this is the cleanest option. Note that this can be done even with a single TCP connection: some minimal framing is needed to distinguish between application specific data and Duktape debugger stream chunks.

Even if an out of band solution is possible, it might be convenient to be able to add application specific commands into the debug protocol. This would make it easy for debug clients to query target specific information (e.g. the heap allocator state of a specific target device) and show it in the debugger UI in an integrated fashion.

Exposing the whole debug protocol to user code through a supported public API would mean significant versioning

issues, but perhaps a limited API could be exposed:

- An API call to send an application specific message with a string name (e.g. "my-target-heap-state") and a string value.
- A callback to receive an application specific message with a string name and a string value.

Direct support for structured values

The current mapping between `duk_tval` values and dvalues works but it cannot represent structured types. For instance, if a hypothetical debug command to set a global variable reads the value argument as a dvalue, it cannot write a value like `[1,2,3]` into the global variable.

This can of course be worked out by doing an `eval()` for the argument or by representing the value as JSON (which is more or less the same thing).

Another alternative is to add support for representing structured values directly with dvalues, so that when C code does a:

```
duk_debug_read_tval(thr);
```

an arbitrarily complex object value (perhaps even an arbitrary object graph) can be decoded and pushed to the value stack.

Heap dump viewer

The DumpHeap command provides a snapshot of all heap objects, which the debugger web UI converts into a JSON dump. It'd be nice to include a viewer for the dump, so that it'd be easy to traverse the object graph, look for strings and values, etc.

Bytecode viewer

Add a command to download function bytecode so that the executor can show source code and bytecode view side-by-side. This would be very useful when working with the Duktape compiler, for instance.

With this view it might also be useful to implement PC-by-PC stepping.

Eclipse debugger

An Eclipse debugger would be very useful as it's a very popular IDE for embedded development.

Breakpoint handling in attach/detach

Currently the list of breakpoints is not cleared by attach or detach, so if you detach and then re-attach, old breakpoints are still set. The debug client can just delete all breakpoints on attach, but it'd be cleaner to remove the breakpoints on either attach or detach.

Indicate fastint status

When debugging code that is intended to operate with fastints, it would be useful to see when a value is internally represented as a fastint vs. a full IEEE double. Currently this information is not conveyed by the protocol, and all fastints appear like any other number values.

Buffer object support

Make it easier to see buffer object contents (like for plain buffers), either by serializing them differently, or through heap walking.