DBGP - A common debugger protocol specification

Version: 1.0
Status: draft 22
Authors: Shane Caraveo, ActiveState <shanec@ActiveState.com>
Derick Rethans <derick@derickrethans.nl>

Contents

- 1. Description
- 1.1 Issues
- 2. Requirements
- 3. Terminology
- 4. Security
- 5. Initiating a debugging session
  - 5.1 Standard DBGP port
  - 5.2 Connection Initialization
  - 5.3 Just in time debugging and debugger proxies
    - 5.3.1 Init Packet Handling
    - 5.3.2 Proxy Errors
    - 5.3.3 Proxy Ports
  - 5.4 Multiple Processes or Threads
  - 5.5 Feature Negotiation
    - Data packet negotiation
    - Asynchronous Communications
- 6. Message Packets
  - 6.1 Why not XML both ways?
  - 6.2 Packet Communications

    - 6.3 IDE to debugger engine communications
        - 6.3.1 Escaping Rules
    - 6.4 debugger engine to IDE communications
        - 6.4.1 response
        - 6.4.2 stream
        - 6.4.3 notify
    - 6.5 debugger engine errors
    - 6.5.1 Error Codes
    - 6.6 file paths
    - 6.7 Dynamic code and virtual files
- 7. Core Commands
    - 7.1 status
    - 7.2 Options and Configuration
        - 7.2.1 Feature Names
        - 7.2.2 feature_get
        - 7.2.3 feature_set
    - 7.5 continuation commands
    - 7.6 breakpoints
        - 7.6.1 breakpoint_set
        - 7.6.2 breakpoint_get
        - 7.6.3 breakpoint_update
        - 7.6.4 breakpoint_remove
        - 7.6.5 breakpoint_list
    - 7.7 stack_depth
    - 7.8 stack_get
    - 7.9 context_names
    - 7.10 context_get
    - 7.11 Properties, variables and values
        - 7.11.1 Extended Properties
    - 7.12 Data Types
        - 7.12.1 Common Data Types
        - 7.12.2 typemap_get
    - 7.13 property_get, property_set, property_value
    - 7.14 source
    - 7.15 stdout, stderr
- 8. Extended Commands
    - 8.1 stdin
    - 8.2 break
    - 8.3 eval
        - 8.3.1 expr
        - 8.3.2 exec
    - 8.4 spawnpoints
        - 8.4.1 spawnpoint_set

        - 8.4.2 spawnpoint_get
        - 8.4.3 spawnpoint_update
        - 8.4.4 spawnpoint_remove
        - 8.4.5 spawnpoint_list
    - 8.5 Notifications
        - 8.5.1 Standard Notifications
        - 8.5.2 Error Notification
    - 8.6 interact - Interactive Shell
- A. ChangeLog

1. Description

This document describes a simple protocol for use with language tools and engines for the purpose of debugging applications. It does not describe user interfaces or interactions with the debugger. The protocol provides a means of communication between a debugger engine (scripting engine, vm, etc.) and a debugger IDE (IDE, etc.). Any references to the debugger IDE UI are recommendations only, and are provided for additional explanation or as reasoning for specific design decisions.

1.1 Issues

1. The handling of proxy errors needs to be clarified. Without both IDE and debugger engine supporting commands to be received at arbitrary times, the proxy may have problems sending error or status information to either one. See section 5.3.2. We should think a bit more about what a proxy might need to do.

2. Requirements

- extensibility, allow for vendor or language specific features
- backwards and forwards compatibility
- firewall and tunneling support
- support for multiple languages
- support for multiple processes or threads
- support for dynamic and possibly for compiled languages

3. Terminology

IDE
: An IDE, or other debugger UI IDE or tool.

debugger engine

The language engine being debugged.

proxy
An intermediary demon that acts as a proxy, and may also implement support for other features such as just in time debugging, ip security, etc.

session
a single thread in an application. multiple threads in an application will attach separately.

TRUE
a value defined as TRUE should be a numeric one.

FALSE
a value defined as FALSE should be a numeric zero.

NUM
a base 10 numeric value that is stringified.

4. Security

It is expected that implementations will provide security, such as ip filtering, ssh tunneling, etc. This protocol itself does not provide a means of securing the debugging session.

5. Initiating a debugging session

The debugger engine initiates a debugging session. The debugger engine will make a connection to a listening IDE, then wait for the IDE to initiate commands. The debugger engine does not step into the first line of execution until the IDE issues one of the continuation commands. The first thing that should happen in a debug session is that the IDE negotiates features using the feature_get and feature_set commands, and sets any additional data, such as breakpoints. Debugger engine implementations should store any data it receives if it is unable to process them prior to compiling and/or executing code. Commands such as stack_get should not be expected to work during this phase, otherwise known as the 'starting' state (see section 7.1 for status levels).

Likewise, at the end of a debug session, there is a 'stopping' state. This state is entered after all execution is complete. For most debugger engine implementations, only a 'stop' command can be accepted at this point, however some implementations may provide additional commands for retrieving various data from the engine for post debug session processing.

5.1 Standard DBGP port

The IDE listens on port 9000 for debugger connections, unless the IDE is using a proxy, in which case it may listen on any port. In that case, the IDE will tell the proxy which port it is listening on, and the proxy should listen on port 9000. While this document defines port 9000 as the standard DBGP port, an implementation may support the use of any port. Current implementations accept various forms of configuration that allow this port to be defined.

5.2 Connection Initialization

When a debugger engine connects to either a IDE or proxy, it must send an init packet:

```
<init appid="APPID"
      idekey="IDE_KEY"
      session="DBGP_COOKIE"
      thread="THREAD_ID"
      parent="PARENT_APPID"
      language="LANGUAGE_NAME"
      protocol_version="1.0"
      fileuri="file://path/to/file">
```

Attributes in the init element can include:

Attribute	Description
appid	defined by the debugger engine
idekey	defined by the user. The DBGP_IDEKEY environment variable SHOULD be used if it is available, otherwise setting this value is debugger engine implementation specific. This value may be empty.
session	If the environment variable DBGP_COOKIE exists, then the init packet MUST contain a session attribute with the value of the variable. This allows an IDE to execute a debugger engine, and maintain some state information between the execution and the protocol connection. This value should not be expected to be set in 'remote' debugging situations where the IDE is not in control of the process.
thread	the systems thread id
parent	the appid of the application that spawned the process. When an application is executed, it should set it's APPID into the environment. If an APPID already exists, it should first read that value and use it as the PARENT_APPID.
language	debugger engine specific, must not contain additional information, such as version, etc.
protocol_version	The highest version of this protocol supported

Attribute	Description
fileuri	URI of the script file being debugged

The IDE responds by dropping socket connection, or starting with debugger commands.

The init packet may have child elements for additional vendor specific data. These are entirely optional and must not effect behavior of the debugger interaction. Suggested child elements include:

```
<engine version="1.abcd">product title</engine>
<author>author</author>
<company>company</company>
<license>licensing info</license>
<url>url</url>
<copyright>xxx</copyright>
```

5.3 Just in time debugging and debugger proxies

Proxies are supported to allow multiuser systems work with a defined port for debugging. Each IDE would listen on a unique port and notify the proxy what port it is listening on, along with a key value that is used by the debugger engine to specify which IDE it should be connected with.

With the exception of the init packet, all communications will be passed through without modifications. A proxy could also implement support for just in time debugging. In this case, a debugger engine would break (perhaps on an error or exception) and connect to the proxy. The proxy would then start the IDE (if it is not already running) and initiate a debugging session with it.

The method for handling just in time debugging is not defined by the protocol and is implementation specific. One example of how this may work is that the proxy has a configuration file that defines key's for each user, along with the path to the executable that will provide the UI for that user. The debugger engine would have to know this key value in advance and provide it to the proxy in the init packet (see IDE_KEY in section 5.2). The proxy would know if the IDE is running, since the IDE should have communicated with the proxy already, if it has not, the proxy could execute the IDE directly.

To support proxies and JIT daemons, the IDE should be configured with a port pointing to the proxy/JIT. The IDE then makes a connection to the proxy when it starts and sends the following command:

IDE command

```
proxyinit -p port -k ide_key -m [0|1]
```

-p	the port that the IDE listens for debugging on. The address is retrieved from the connection information.
-k	a IDE key, which the debugger engine will also use in it's debugging init command. this allows the proxy to match request to IDE. Typically the user will provide the session key as a configuration item.
-m	this tells the demon that the IDE supports (or doesn't) multiple debugger sessions. if -m is missing, zero or no support is default.

IDE command

```
proxystop -k ide_key
```

The IDE sends a proxystop command when it wants the proxy server to stop listening for it.

The proxy should respond with a simple XML statement alerting the IDE to an error, or the success of the initialization (see section 6.5 for more details on the error element).

```
<proxyinit success="[0|1]"
           idekey="{ID}"
           address="{IP_ADDRESS}"
           port="{NUM}>
    <error id="app_specific_error_code">
        <message>UI Usable Message</message>
    </error>
</proxyinit>
```

Once the IDE has sent this command, and received a confirmation, it disconnects from the proxy. The IDE will only connect to the proxy when it initially wants to start accepting connections from the proxy, or when it wants to stop accepting connections from the proxy.

The address and port attributes of the returned proxyinit element are the address and port that the proxy is configured to listen for DBGP connections on. This information is returned to the IDE so that it may pass this information on to build systems or the user via some UI.

5.3.1 Init Packet Handling

If a proxy receives the init packet (see section 5.2), it will use the idekey attribute to pass the request to the correct IDE, or to do some other operation such as which may be required to implement security or initiate just in time debugging. The proxy will add the idekey as a attribute to the init packet when it passes it through to the IDE. The proxy may also add child elements with further information, and must add an attribute to the init element called 'proxied' with the attribute value being the ip address of the debugger engine. This is the only time the proxy should modify data being passed to the IDE.

5.3.2 Proxy Errors

If the proxy must send error data to the IDE, it may send an XML message with the root element named 'proxyerror'. This message will be in the format of the error packets defined in 6.3 below.

If the proxy must send error data to the debugger engine, it may send the proxyerror command defined in section 7 below.

5.3.3 Proxy Ports

The proxy listens for IDE connections on port 9001, and for debugger engine connections on port 9000. As with section 5.1, these ports may be configurable in the implementation.

5.4 Multiple Processes or Threads

The debugger protocol is designed to use a separate socket connection for each process or thread. The IDE may or may not support multiple debugger sessions. If it does not, the debugger engine must not attempt to start debug sessions for threads, and the IDE should not accept more than one socket connection for debugging. The IDE should tell the debugger engine whether it supports multiple debugger sessions, the debugger engine should assume that the IDE does not. The IDE can use the feature_set command with the feature name of 'multiple_sessions' to notify the debugger engine that it supports multiple session debugging. The IDE may also query the debugger engine specifically for multithreaded debugging support by using the feature_get command with a feature name of 'language_supports_threads'.

5.5 Feature Negotiation

Although the IDE may at any time during the debugging session send feature_get or feature_set commands, the IDE should be designed to negotiate the base set of features up front. Differing languages and debugger engines may operate in many ways, and the IDE should be prepared to handle these differences. Likewise, the IDE may dictate certain features or capabilities be supported by the debugger engine. In

any case, the IDE should strive to work with all debugger engines that support this protocol. Therefore, this section describes a minimal set of features the debugger engine must support. These required features are outlined here in the form of discussion, actual implementation of feature arguments are detailed in section 7 under the feature_get and feature_set commands.

Data packet negotiation

IDE's may want to limit the size of data that is retrieved from debugger engines. While the debugger engines will define their own base default values, the IDE should negotiate these terms if it needs to. The debugger engine must support these requests from the IDE. This includes limits to the data size of a property or variable, and the depth limit to arrays, hashes, objects, or other tree like structures. The data size excludes the protocol overhead.

Asynchronous Communications

While the protocol does not depend on asynchronous socket support, certain design considerations may require that the IDE and/or debugger engine treat incoming and outgoing data in an asynchronous fashion.

For ease of design, some implementations may choose to utilize this protocol in a completely synchronous fashion, and not implement optional commands that require the debugger engine to behave in an asynchronous fashion. One example of this is the break command.

The break command is sent from the IDE while the debugger engine is in a run state. To support this, the debugger engine must periodically peek at the socket to see if there are any incoming commands. For this reason the break command is optional. If a command requires this type of asynchronous behavior on the part of the debugger engine it must be optional for the debugger engine to support it.

On the other hand, IDE's MUST at times behave in an asynchronous fashion. When an IDE tells the debugger engine to enter a 'run' state, it must watch the socket for incoming packets for stdout or stderr, if it has requested the data be sent to it from the debugger engine.

The form of asynchronous communications that may occur in this protocol are defined further in section 6.2 below.

6. Message Packets

The IDE sends simple ASCII commands to the debugger engine. The debugger engine responds with XML data. The XML data is prepended with a stringified

integer representing the number of bytes in the XML data packet. The length and XML data are separated by a NULL byte. The XML data is ended with a NULL byte. Neither the IDE or debugger engine packets may contain NULL bytes within the packet since it is used as a separator. The IDE to debugger engine `data` element (behind `--`) MUST be encoded using base64.

```
IDE:       command [SPACE] [args] -- data [NULL]
DEBUGGER:  [NUMBER] [NULL] XML(data) [NULL]
```

Arguments to the IDE commands are in the same format as common command line arguments, and should be parseable by common code such as getopt, or Pythons Cmd module:

```
command -a value -b value ...
```

All numbers in the protocol are base 10 string representations, unless the number is noted to be debugger engine specific (e.g. the address attribute on property elements).

6.1 Why not XML both ways?

The primary reason is to avoid the requirement that a debugger engine has an XML parser available. XML is easy to generate, but requires additional libraries for parsing.

6.2 Packet Communications

The IDE sends a command, then waits for a response from the debugger engine. If the command is not received in a reasonable time (implementation dependent) it may assume the debugger engine has entered a non-responsive state. The exception to this is when the IDE sends a 'continuation' command which may not have an immediate response.

'continuation' commands include, but may not be limited to: run, step_into, step_over, step_out and eval. When the debugger engine receives such a command, it is considered to have entered a 'run state'.

During a 'continuation' command, the IDE should expect to possibly receive stdin and/or stderr packets from the debugger engine prior to receiving a response to the command itself. It may also possibly receive error packets from either the debugger engine, or a proxy if one is in use, either prior to the 'continuation' response, or in response to any other command.

Stdout and stderr, if requested by the IDE, may only be sent during commands that have put the debugger engine into a 'run' state.

If the debugger engine supports asynchronous commands, the IDE may also send commands while the debugger engine is in a 'run' state. These commands should be limited to commands such as the 'break' or 'status' commands for performance reasons, but this protocol does not impose such limitations. The debugger engine MUST respond to these commands prior to responding to the original 'run' command.

An example of communication between IDE and debugger engines. (this is not an example of the actual protocol.)

```
IDE:  feature_get supports_async
DBG:  yes
IDE:  stdin redirect
DBG:  ok
IDE:  stderr redirect
DBG:  ok
IDE:  run
DBG:  stdin data...
DBG:  stdin data...
DBG:  reached breakpoint, done running
IDE:  give me some variables
DBG:  ok, here they are
IDE:  evaluate this expression
DBG:  stderr data...
DBG:  ok, done
IDE:  run
IDE:  break
DBG:  ok, breaking
DBG:  at breakpoint, done running
IDE:  stop
DBG:  good bye
```

6.3 IDE to debugger engine communications

A debugging IDE (IDE) sends commands to the debugger engine in the form of command line arguments.

Some comments support a `data` argument that is included at the end of the "command line". This data is included in the packet's data length. The data itself is the last part of the command line, after the -- separator. The data must be base64 encoded:

```
command [SPACE] [arguments] [SPACE] -- base64(data) [NULL]
```

Standard arguments for all commands

```
-i      Transaction ID
        unique numerical ID for each command generated by the IDE
```

All the other arguments depend on which `command` is being sent.

6.3.1 Escaping Rules

If a value for an option includes a space or NULL character, the IDE MUST encapsulate the value in double quotes (`"`):

```
property_get -i 5 -n "$x['a b']"
```

In other cases, the value MAY be encapsulated in double quotes:

```
property_get -i 6 -n "$x['ab']"
```

Inside a double-quoted argument, the double-quote (`"`), back-slash (`\`) and NULL character (*chr(0)*) MUST be escaped with a back-slash. Single quotes (`'`) MAY be escaped.

Escaping double-quote and single-quote:

```
property_get -i 7 -n "$x[\"a b\"]"
property_get -i 8 -n "$x['a b']"
property_get -i 9 -n "$x[\'a b\']"
```

Most languages don't support this, but we have if we have a NULL character in the string, like in `$x` *chr(0)* `y`, or, `$x->f` *chr(0)* `oo`, than they need to be quoted with a back-slash (`\`), as in:

```
property_get -i 10 -n "$x\0y"
property_get -i 11 -n "$x->f\0oo"
```

If the NULL character is already escaped according to a language, for example, as part of an array element key, then there is no un-escaped NULL character in the string any more, and hence does not need to be escaped again.

The Debugging Engine in this case would already have returned the XML attribute value `$x[&quot;a\0b&quot;]` **[*]**, where the *chr(0)* is already escaped as `\0`.

XML decoding this gives `$x["a\0b"]`, which can be used as argument as either one of:

```
property_get -i 12 -n $x["a\0b"]
property_get -i 13 -n "$x[\"a\\0b\"]"
```

[*]
With the `extended_properties` feature set to `1`, the Debugger Engine would not have sent `$x[&quot;a\0b&quot;]` as attribute value, but rather it would have used `<fullname encoding="base64"><![CDATA[JHhbImFcMGIiXQ==]]></fullname>`, as the *chr(0)* in the **name** property would have triggered the use of base64 encoded XML element values.

Without `extended_properties` set to `1`, the Debugger Engine would have sent the invalid XML attribute value `&#0;`.

6.4 debugger engine to IDE communications

The debugger engine always replies or sends XML data. The standard namespace for the root elements returned from the debugger engine MUST be `urn:debugger_protocol_v1`. Namespaces have been left out in the other examples in this document. The messages sent by the debugger engine must always be NULL terminated. The XML document tag must always be present to provide XML version and encoding information.

For simplification, data length and NULL bytes will be left out of the rest of the examples in this document.

Three base tags are used for the root tags:

6.4.1 response

This data packet is returned as response to continuation commands:

```
data_length
[NULL]
<?xml version="1.0" encoding="UTF-8"?>
<response xmlns="urn:debugger_protocol_v1"
```

```
            command="command_name"
            transaction_id="transaction_id"/>
[NULL]
```

6.4.2 stream

This data packet is sent when stream redirection is enabled through **7.15 stdout, stderr** redirections. As the data is base64 encoded, the `stream` packet always has the `encoding` attribute set to `base64`:

```
data_length
[NULL]
<?xml version="1.0" encoding="UTF-8"?>
<stream xmlns="urn:debugger_protocol_v1"
        type="stdout|stderr"
        encoding="base64">
    ...Base64 Data...
</stream>
[NULL]
```

6.4.3 notify

This data packet is sent when **8.5 Notifications** are enabled through setting the `notify_ok` feature (see **7.2.1 Feature names**). As the data is base64 encoded, the `notify` packet always has the `encoding` attribute set to `base64`:

```
data_length
[NULL]
<?xml version="1.0" encoding="UTF-8"?>
<notify xmlns="urn:debugger_protocol_v1"
        xmlns:customNs="http://example.com/dbgp/example"
        name="notification_name"
        encoding="base64">
    <customNs:customElement/>
    ...Base64 Data...
</notify>
[NULL]
```

6.5 debugger engine errors

A debugger engine may need to relay error information back to the IDE in response to any command. The debugger engine may add an error element as a child of the response element. Note that this is not the same as getting language error

messages, such as exception data. This is specifically a debugger engine error in response to a IDE command. IDEs and debugger engines may elect to support additional child elements in the error element, but should namespace the elements to avoid conflicts with other implementations.

```
<response command="command_name"
          transaction_id="transaction_id">
    <error code="error_code" customNs:apperr="app_specific_error_code">
        <customNs:message>UI Usable Message</customNs:message>
    </error>
</response>
```

6.5.1 Error Codes

The following are predefined error codes for the response to commands:

000 Command parsing errors

```
0 - no error
1 - parse error in command
2 - duplicate arguments in command
3 - invalid options (ie, missing a required option, invalid value for a
    passed option, not supported feature)
4 - Unimplemented command
5 - Command not available (Is used for async commands. For instance
    if the engine is in state "run" then only "break" and "status"
    are available).
```

100 File related errors

```
100 - can not open file (as a reply to a "source" command if the
      requested source file can't be opened)
101 - stream redirect failed
```

200 Breakpoint, or code flow errors

```
200 - breakpoint could not be set (for some reason the breakpoint
      could not be set due to problems registering it)
201 - breakpoint type not supported (for example I don't support
      'watch' yet and thus return this error)
202 - invalid breakpoint (the IDE tried to set a breakpoint on a
      line that does not exist in the file (ie "line 0" or lines
```

```
      past the end of the file)
203 - no code on breakpoint line (the IDE tried to set a breakpoint
      on a line which does not have any executable code. The
      debugger engine is NOT required to return this type if it
      is impossible to determine if there is code on a given
      location. (For example, in the PHP debugger backend this
      will only be returned in some special cases where the current
      scope falls into the scope of the breakpoint to be set)).
204 - Invalid breakpoint state (using an unsupported breakpoint state
      was attempted)
205 - No such breakpoint (used in breakpoint_get etc. to show that
      there is no breakpoint with the given ID)
206 - Error evaluating code (use from eval() (or perhaps
      property_get for a full name get))
207 - Invalid expression (the expression used for a non-eval()
      was invalid)
```

300 Data errors

```
300 - Can not get property (when the requested property to get did
      not exist, this is NOT used for an existing but uninitialized
      property, which just gets the type "uninitialised" (See:
      PreferredTypeNames)).
301 - Stack depth invalid (the -d stack depth parameter did not
      exist (ie, there were less stack elements than the number
      requested) or the parameter was < 0)
302 - Context invalid (an non existing context was requested)
```

900 Protocol errors

```
900 - Encoding not supported
998 - An internal exception in the debugger occurred
999 - Unknown error
```

6.6 file paths

All file paths passed between the IDE and debugger engine must be in the URI format specified by IETF RFC 1738 and 2396, and must be absolute paths.

6.7 Dynamic code and virtual files

The protocol reserves the URI scheme 'dbgp' for all virtual files generated and maintained by language engines. Such virtual files are usually managed by a

language engine for dynamic code blocks, i.e. code created at runtime, without an association with a regular file. Any IDE seeing an URI with the 'dbgp' scheme has to use the 'source' command (See section 7.14) to obtain the contents of the file from the engine responsible for that URI.

All URIs in that scheme have the form:

dbgp:engine-specific-identifier

The engine-specific-identifier is some string which the debugger engine uses to keep track of the specific virtual file. The IDE must return the URI to the debugger engine unchanged through the source command to retrieve the virtual file.

7. Core Commands

Both IDE and debugger engine must support all core commands.

7.1 status

The status command is a simple way for the IDE to find out from the debugger engine whether execution may be continued or not. no body is required on request. If async support has been negotiated using feature_get/set the status command may be sent while the debugger engine is in a 'run state'.

The status attribute values of the response may be:

starting:
State prior to execution of any code

stopping:
State after completion of code execution. This typically happens at the end of code execution, allowing the IDE to further interact with the debugger engine (for example, to collect performance data, or use other extended commands).

stopped:
IDE is detached from process, no further interaction is possible.

running:
code is currently executing. Note that this state would only be seen with async support turned on, otherwise the typical state during IDE/debugger interaction would be 'break'

break:
code execution is paused, for whatever reason (see below), and the IDE/debugger can pass information back and forth.

The reason attribute value may be:

- ok

- error
- aborted
- exception

IDE

```
status -i transaction_id
```

debugger engine

```
<response command="status"
          status="starting"
          reason="ok"
          transaction_id="transaction_id">
    message data
</response>
```

7.2 Options and Configuration

The feature commands are used to request feature support from the debugger engine. This includes configuration options, some of which may be changed via feature_set, the ability to discover support for implemented commands, and to discover values for various features, such as the language version or name.

An example of usage would be to send a feature request with the string 'stdin' to find out if the engine supports redirection of the stdin stream through the debugger socket. The debugger engine must consider all commands as keys for this command, but may also have keys that are for features that do not map directly to commands.

7.2.1 Feature Names

The following features strings MUST be available:

language_supports_threads	get	[0|1]
language_name	get	{eg. PHP, Python, Perl}
language_version	get	{version string}
encoding	get|set	current encoding in use by the debugger session. The encoding can either be (7-bit) ASCII, or a code set which contains ASCII (Ex: ISO-8859-X, UTF-8). Use the `supported_encodings` feature to query which encodings are supported
protocol_version	get	{for now, always 1}

supports_async	get	{for commands such as break}
data_encoding	get	optional, allows to turn off the default base64 encoding of data. This should only be used for development and debugging of the debugger engines themselves, and not for general use. If implemented the value 'base64' must be supported to turn back on regular encoding. the value 'none' means no encoding is in use. all elements that use encoding must include an encoding attribute.
breakpoint_languages	get	some engines may support more than one language. This feature returns a string which is a comma separated list of supported languages. **If the engine does not provide this feature, then it is assumed that the engine only supports the language defined in the feature language_name.** One example of this is an XSLT debugger engine which supports XSLT, XML, HTML and XHTML. An IDE may need this information to to know what types of breakpoints an engine will accept.
breakpoint_types	get	returns a space separated list with all the breakpoint types that are supported. See 7.6 breakpoints for a list of the 6 defined breakpoint types.
multiple_sessions	get|set	{0|1}
max_children	get|set	max number of array or object children to initially retrieve
max_data	get|set	max amount of variable data to initially retrieve.
max_depth	get|set	maximum depth that the debugger engine may return when sending arrays, hashs or object structures to the IDE.

The following features strings MAY be available, if they are not, the IDE should assume that the feature is not available:

breakpoint_details	get|set	whether breakpoint information is included in the response to a continuation command, when the debugger hits a 'break' state (for example, when encountering a breakpoint)
extended_properties	get|set	{0|1} Extended properties are required if there are property names (name, fullname or classname) that can not be represented as valid XML attribute values (such as `&#0;`). See also 7.11 Properties, variables and values.

notify_ok	get|set	[0|1] See section 8.5 Notifications.
resolved_breakpoints	get|set	whether 'breakpoint_resolved' notifications may be send by the debugging engine in case it is supported. See the *resolved* attribute under 7.6 breakpoints for further information.
supported_encodings	get	returns a comma separated list of all supported encodings that can be set through the `encoding` feature
supports_postmortem	get	[0|1] This feature lets an IDE know that there is benefit to continuing interaction during the STOPPING state (sect. 7.1).
show_hidden	get|set	[0|1] This feature can get set by the IDE if it wants to have more detailed internal information on properties (eg. private members of classes, etc.) Zero means that hidden members are not shown to the IDE.

Additionally, all protocol commands supported must have a string, such as the following examples:

```
breakpoint_set
break
eval
```

7.2.2 feature_get

arguments for feature_get include:

-n	feature_name

IDE

```
feature_get -i transaction_id -n feature_name
```

debugger engine

```
<response command="feature_get"
          feature_name="feature_name"
          supported="0|1"
          transaction_id="transaction_id">
    feature setting or available options, such as a list of
    supported encodings
</response>
```

The 'supported' attribute does NOT mean that the feature is supported, this is encoded in the text child of the response tag. The 'supported' attribute informs whether the feature with 'feature_name' is supported by feature_get in the engine, or when the command with name 'feature_get' is supported by the engine.

Example: Xdebug does not understand the 'breakpoint_languages' feature and will therefore set the supported attribute to '0'. It does however understand the feature 'language_supports_threads' and the 'supported' attribute is therefore set to '1', but as PHP does not support threads, the returned value is in this case "0".

7.2.3 feature_set

The feature set command allows a IDE to tell the debugger engine what additional capabilities it has. One example of this would be telling the debugger engine whether the IDE supports multiple debugger sessions (for threads, etc.). The debugger engine responds with telling the IDE whether it has enabled the feature or not.

Note: The IDE does not have to listen for additional debugger connections if it does not support debugging multiple sessions. debugger engines must handle connection failures gracefully.

arguments for feature_set include:

-n	feature_name
-v	value to be set

feature_set can be called at any time during a debug session to change values previously set. This allows a IDE to change encodings.

IDE

```
feature_set -i transaction_id -n feature_name -v value
```

debugger engine

```
<response command="feature_set"
          feature="feature_name"
          success="0|1"
          transaction_id="transaction_id"/>
```

If the feature is not supported, the debugger engine should return an error with the code set to 3 (invalid arguments).

7.5 continuation commands

resume the execution of the application.

run:
starts or resumes the script until a new breakpoint is reached, or the end of the script is reached.

step_into:
steps to the next statement, if there is a function call involved it will break on the first statement in that function

step_over:
steps to the next statement, if there is a function call on the line from which the step_over is issued then the debugger engine will stop at the statement after the function call in the same scope as from where the command was issued

step_out:
steps out of the current scope and breaks on the statement after returning from the current function. (Also called 'finish' in GDB)

stop:
ends execution of the script immediately, the debugger engine may not respond, though if possible should be designed to do so. The script will be terminated right away and be followed by a disconnection of the network connection from the IDE (and debugger engine if required in multi request apache processes).

detach (optional):
stops interaction with the debugger engine. Once this command is executed, the IDE will no longer be able to communicate with the debugger engine. This does not end execution of the script as does the stop command above, but rather detaches from debugging. Support of this continuation command is optional, and the IDE should verify support for it via the feature_get command. If the IDE has created stdin/stdout/stderr pipes for execution of the script (eg. an interactive shell or other console to catch script output), it should keep those open and usable by the process until the process has terminated normally.

The response to a continue command is a status response (see status above). The debugger engine does not send this response immediately, but rather when it reaches a breakpoint, or ends execution for any other reason.

IDE

```
run -i transaction_id
```

debugger engine

```
<response command="run"
          transaction_id="transaction_id"
```

```
            status="break"
            reason="ok"/>
```

A debugger engine may include extra information as sub element to *response*, as long as it is in its own name space. In the example below the debugger engine added the file and line location where the application is now paused:

```
<response xmlns="urn:debugger_protocol_v1" xmlns:xdebug="https://xdebug.org/db
          command="run"
          transaction_id="transaction_id"
          status="break"
          reason="ok">
    <xdebug:message filename="file://bug01312.inc" lineno="26"/>
</response>
```

If the *breakpoint_details* feature is enabled with **7.2.3 feature_set** then the debugger information MAY also include a sub element to described the breakpoint. The *breakpoint* element has the same format and contents as the response to a **7.6.2 breakpoint_get** command:

```
<response command="run"
          transaction_id="6"
          status="break"
          reason="ok">
    <breakpoint id="BREAKPOINT_ID"
                type="call"
                function="breaking::staticMe"
                state="enabled"
                hit_count="2"
                hit_value="0"/>
</response>
```

7.6 breakpoints

Breakpoints are locations or conditions at which a debugger engine pauses execution, responds to the IDE, and waits for further commands from the IDE. A failure in any breakpoint commands results in an error defined in **section 6.5**.

The following DBGP commands relate to breakpoints:

breakpoint_set	Set a new breakpoint on the session.
breakpoint_get	Get breakpoint info for the given breakpoint id.

breakpoint_update	Update one or more attributes of a breakpoint.
breakpoint_remove	Remove the given breakpoint on the session.
breakpoint_list	Get a list of all of the session's breakpoints.

There are six different breakpoints *types*:

Type	Req'd Attrs	Description
line	filename, lineno	break on the given lineno in the given file
call	function	break on entry into new stack for function name
return	function	break on exit from stack for function name
exception	exception	break on exception of the given name
conditional	expression, filename	break when the given expression is true at the given filename and line number or just in given filename
watch	expression	break on write of the variable or address defined by the expression argument

A breakpoint has the following attributes. Note that some attributes are only applicable for some breakpoint types.

type	breakpoint type (see table above for valid types)
filename	The file the breakpoint is effective in. This must be a "file://" or "dbgp:" (See 6.7 Dynamic code and virtual files) URI.
lineno	Line number on which breakpoint is effective. Line numbers are 1-based. If an implementation requires a numeric value to indicate that *lineno* is not set, it is suggested that -1 be used, although this is not enforced.
state	Current state of the breakpoint. This must be one of *enabled*, *disabled*.
function	Function name for *call* or *return* type breakpoints.
temporary	Flag to define if breakpoint is temporary. A temporary breakpoint is one that is deleted after its first use. This is useful for features like "Run to Cursor". Once the debugger engine uses a temporary breakpoint, it should automatically remove the breakpoint from it's list of valid breakpoints.
resolved	Flag to denote whether a breakpoint has been resolved. The value of the attribute is either *resolved* or *unresolved*. A resolved breakpoint is one where the debugger engine has established that it can actually break on: the file/line number (*line* type breakpoints), the function name (*call* and *return* type breakpoints), or the exception name (*exception* type breakpoints). For dynamic languages, that load files as the execution happens, this is useful for finding out invalid breakpoints. This is a **read only** flag. It MUST be included

	when the *resolved_breakpoints* feature has been activated, and it MUST NOT be included if the *resolved_breakpoints* feature has not been enabled. An IDE can use 7.2.3 feature_set to enable the feature (if supported), and 7.2.2 feature_get to find out whether the debugging engine supports *resolved_breakpoints*, and whether it has been enabled.
hit_count	Number of effective hits for the breakpoint in the current session. This value is maintained by the debugger engine (a.k.a. DBGP client). A breakpoint's hit count should be increment whenever it is considered to break execution (i.e. whenever debugging comes to this line). If the breakpoint is *disabled* then the hit count should NOT be incremented.
hit_value	A numeric value used together with the *hit_condition* to determine if the breakpoint should pause execution or be skipped.
hit_condition	A string indicating a condition to use to compare *hit_count* and *hit_value*. The following values are legal: **>=** break if hit_count is greater than or equal to hit_value [default] **==** break if hit_count is equal to hit_value **%** break if hit_count is a multiple of hit_value
exception	Exception name for *exception* type breakpoints.
expression	The expression used for *conditional* and *watch* type breakpoints

Breakpoints should be maintained in the debugger engine at an application level, not the thread level. Debugger engines that support thread debugging MUST provide breakpoint id's that are global for the application, and must use all breakpoints for all threads where applicable.

As for any other commands, if there is error the debugger engine should return an error response as described in **section 6.5**.

7.6.1 breakpoint_set

This command is used by the IDE to set a breakpoint for the session.

IDE to debugger engine:

```
breakpoint_set -i TRANSACTION_ID [<arguments...>] -- base64(expression)
```

where the arguments are:

-t TYPE	breakpoint *type*, see above for valid values [required]
-s STATE	breakpoint *state* [optional, defaults to "enabled"]
-f FILENAME	the *filename* to which the breakpoint belongs [optional]
-n LINENO	the line number (*lineno*) of the breakpoint [optional]
-m FUNCTION	*function* name [required for *call* or *return* breakpoint types]
-x EXCEPTION	*exception* name [required for *exception* breakpoint types]
-h HIT_VALUE	hit value (*hit_value*) used with the hit condition to determine if should break; a value of zero indicates hit count processing is disabled for this breakpoint [optional, defaults to zero (i.e. disabled)]
-o HIT_CONDITION	hit condition string (*hit_condition*); see hit_condition documentation above; BTW 'o' stands for 'operator' [optional, defaults to '>=']
-r 0|1	Boolean value indicating if this breakpoint is *temporary*. [optional, defaults to false]
-- EXPRESSION	code *expression*, in the language of the debugger engine. The breakpoint should activate when the evaluated code evaluates to *true*. [required for *conditional* breakpoint types]

An example breakpoint_set command for a conditional breakpoint could look like this:

```
breakpoint_set -i 1 -t line -f test.pl -n 20 -- base64($x > 3)
```

A unique id for this breakpoint for this session is returned by the debugger engine. This *session breakpoint id* is used by the IDE to identify the breakpoint in other breakpoint commands. It is up to the engine on how to handle multiple "similar" breakpoints, such as a double breakpoint on a specific file/line combination - even if other parameters such as hit_value/hit_condition are different.

debugger engine to IDE:

```
<response command="breakpoint_set"
          transaction_id="TRANSACTION_ID"
          state="STATE"
          resolved="RESOLVED"
          id="BREAKPOINT_ID"/>
```

where,

BREAKPOINT_ID	is an arbitrary string that uniquely identifies this breakpoint in the debugger engine.
STATE	the initial state of the breakpoint as set by the debugger engine
RESOLVED	*resolved* if the debugger engine knows the breakpoint is valid, or *unresolved* otherwise. This attribute is only present if the debugger engine implements the "resolving" feature, and the IDE has enabled it.

7.6.2 breakpoint_get

This command is used by the IDE to get breakpoint information from the debugger engine.

IDE to debugger engine:

```
breakpoint_get -i TRANSACTION_ID -d BREAKPOINT_ID
```

where,

BREAKPOINT_ID	is the unique *session breakpoint id* returned by *breakpoint_set*.

debugger engine to IDE:

```
<response command="breakpoint_get"
          transaction_id="TRANSACTION_ID">
    <breakpoint id="BREAKPOINT_ID"
                type="TYPE"
                state="STATE"
                resolved="RESOLVED"
                filename="FILENAME"
                lineno="LINENO"
                function="FUNCTION"
                exception="EXCEPTION"
                expression="EXPRESSION"
                hit_value="HIT_VALUE"
                hit_condition="HIT_CONDITION"
                hit_count="HIT_COUNT">
        <expression>EXPRESSION</expression>
    </breakpoint>
</response>
```

where each breakpoint attribute is only required if its value is relevant. E.g., the <expression/> child element need only be included if there is an expression defined, the *function* attribute need only be included if this is a *function* breakpoint.

The *lineno* attribute might be different from the one set through **7.6.1 breakpoint_set** due to breakpoint resolving, but only if the *resolved* attribute is set to *resolved*.

7.6.3 breakpoint_update

This command is used by the IDE to update one or more attributes of a breakpoint that was already set on the debugger engine via *breakpoint_set*.

IDE to debugger engine:

```
breakpoint_update -i TRANSACTION_ID -d BREAKPOINT_ID [<arguments...>]
```

where the arguments are as follows. All arguments are optional, however at least one argument should be present. See **breakpoint_set** for a description of each argument:

-s	STATE
-n	LINENO
-h	HIT_VALUE
-o	HIT_CONDITION

debugger engine to IDE:

```
<response command="breakpoint_update"
          transaction_id="TRANSACTION_ID"/>
```

7.6.4 breakpoint_remove

This command is used by the IDE to remove the given breakpoint. The debugger engine can optionally embed the removed breakpoint as child element.

IDE to debugger engine:

```
breakpoint_remove -i TRANSACTION_ID -d BREAKPOINT_ID
```

debugger engine to IDE:

```
<response command="breakpoint_remove"
          transaction_id="TRANSACTION_ID"/>
```

7.6.5 breakpoint_list

This command is used by the IDE to get breakpoint information for all breakpoints that the debugger engine has.

IDE to debugger engine:

```
breakpoint_list -i TRANSACTION_ID
```

debugger engine to IDE:

```
<response command="breakpoint_list"
          transaction_id="TRANSACTION_ID">
    <breakpoint id="BREAKPOINT_ID"
                type="TYPE"
                state="STATE"
                resolved="RESOLVED"
                filename="FILENAME"
                lineno="LINENO"
                function="FUNCTION"
                exception="EXCEPTION"
                hit_value="HIT_VALUE"
                hit_condition="HIT_CONDITION"
                hit_count="HIT_COUNT">
        <expression>EXPRESSION</expression>
    </breakpoint>
    <breakpoint ...>...</breakpoint>
    ...
</response>
```

The *lineno* attribute for each entry might be different from the one set through **7.6.1 breakpoint_set** due to breakpoint resolving, but only if the *resolved* attribute is set to *resolved*.

7.7 stack_depth

Returns the maximum stack depth that can be returned by the debugger. The optional -d argument of the *stack_get* command must be less than this number.

IDE

```
stack_depth -i transaction_id
```

debugger engine

```
<response command="stack_depth"
          depth="{NUM}"
          transaction_id="transaction_id"/>
```

7.8 stack_get

Returns stack information for a given stack depth. For extended debuggers, multiple file/line's may be returned by having child elements of the stack element. This is to allow for debuggers, such as XSLT, that have execution and data files. The filename returned should always be the local file system path translated into a file URI, and should include the system name if the engine is not connecting to an ip on the local box: **file://systemname**/c|/path. If the stack depth is specified, only one stack element is returned, for the depth requested, though child elements may be returned also. The current context is stack depth of zero, the 'oldest' context (in some languages known as 'main') is the highest numbered context.

-d	stack depth (optional)

IDE

```
stack_get -d {NUM} -i transaction_id
```

debugger engine

```
<response command="stack_get"
          transaction_id="transaction_id">
    <stack level="{NUM}"
           type="file|eval|?"
           filename="..."
           lineno="{NUM}"
           where=""
           cmdbegin="line_number:offset"
           cmdend="line_number:offset"/>
    <stack level="{NUM}"
           type="file|eval|?"
           filename="..."
           lineno="{NUM}">
        <input level="{NUM}"
               type="file|eval|?"
               filename="..."
               lineno="{NUM}"/>
```

```
    </stack>
</response>
```

Attributes for the stack element can include:

Attribute	Description
level	stack depth for this stack element
type	the type of stack frame. Valid values are file or eval.
filename	file URI
lineno	1-based line offset into the buffer
where	current command name (optional)
cmdbegin	line number and text offset from beginning of line for the current instruction (optional)
cmdend	same as cmdbegin, denotes end of current instruction

The attributes where, cmdbegin and cmdlength are primarily used for relaying visual information in the IDE. cmdbegin and cmdend can be used by the IDE for high lighting the command that is currently being debugged. The where attribute contains the name of the current stack. This could be the current function name that the user is stepping through.

7.9 context_names

The names of currently available contexts at a given stack depth, typically Local, Global and Class. These SHOULD be UI friendly names. The numerical id attribute returned with the names is used in other commands such as context_get to identify the context. The context id zero is always considered to be the 'default' context is no context id is provided. In most languages, this will the be 'local' context.

-d	stack depth (optional)

IDE

```
context_names -d {NUM} -i transaction_id
```

debugger engine

```
<response command="context_names"
          transaction_id="transaction_id">
    <context name="Local" id="0"/>
    <context name="Global" id="1"/>
```

```
    <context name="Class" id="2"/>
</response>
```

7.10 context_get

Returns an array of properties in a given context at a given stack depth. If the stack depth is omitted, the current stack depth is used. If the context name is omitted, the context with an id zero is used (generally the 'locals' context).

-d	stack depth (optional)
-c	context id (optional, retrieved by context_names)

IDE

```
context_get -d {NUM} -c {NUM} -i transaction_id
```

debugger engine

```
<response command="context_get"
          context="context_id"
          transaction_id="transaction_id">
    <property ... />
</response>
```

7.11 Properties, variables and values

Properties that have children may return an arbitrary depth of children, as defaulted by the debugger engine. A maximum depth may be defined by the IDE using the feature_set command with the max_depth argument. The IDE MAY then use the fullname attribute of a property to dig further into the tree with `property_get`, `property_set` and `property_value`. An IDE MUST NOT apply any escaping rules to the `fullname` attribute except for the escaping rules as are explained in **6.3.1 Escaping Rules**. Data types are defined further in section 7.12 below.

The number of children sent is defined by the debugger engine unless otherwise defined by sending the feature_set command with the max_children argument. If max_depth > 1, irregardless of the page argument, the childrens pages are always the first page. Children are only returned if max_depth > 0 and max_children > 0.

```
<property
    name="short_name"
    fullname="long_name"
    type="data_type"
    classname="name_of_object_class"
    constant="0|1"
    children="0|1"
    size="{NUM}"
    page="{NUM}"
    pagesize="{NUM}"
    address="{NUM}"
    key="language_dependent_key"
    encoding="base64|none"
    numchildren="{NUM}">
...encoded Value Data...
</property>
```

Attributes in the property element can include:

Attribute	Description
name	variable name. This is the short part of the name, and is meant to be displayed in the IDE's UI. For instance, in PHP: $v = 0; // short name 'v' $object->v; // short name 'v'
fullname	variable name. This is the long form of the name, which, once XML-decoded, can be used by the IDE to retrieve the value of the variable trough `property_get` or `property_value`, following the 6.3.1 Escaping Rules for IDE to Debugger Engine communications. Although it resembles the language's own syntax to describe and evaluate a variable name, IDEs SHOULD NOT use the `eval` command to retrieve nested properties with this, but instead use `property_get`. $v = 0; // long name 'v' class::$v; // short name 'v', long 'class::$v' $this->v; // short name 'v', long '$this->v'
classname	If the type is an object or resource, then the debugger engine MAY specify the class name This is an optional attribute.
page	if not all the children in the first level are returned, then the page attribute, in combination with the pagesize attribute will define where in the array or object these children should be located. The page number is 0-based.
pagesize	the size of each page of data, defaulted by the debugger engine, or negotiated with feature_set and max_children. Required when the page attribute is available.
type	language specific data type name
facet	provides a hint to the IDE about additional facets of this value. These are space separated names, such as private, protected, public, constant, etc.

Attribute	Description
size	size of property data in bytes
children	true/false whether the property has children this would be true for objects or array's.
numchildren	optional attribute with number of children for the property.
key	language dependent reference for the property. if the key is available, the IDE SHOULD use it to retrieve further data for the property, optional
address	containing physical memory address, optional
encoding	if this is binary data, it should be base64 encoded with this attribute set

7.11.1 Extended Properties

If the debugger engine and IDE have negotiated to use extended properties (through `feature_set -n extended_properties`), then a debugging engine MAY use the extended property return type defined here.

A debugging engine MUST use this format if either the name, fullname, or classname values can not be encoded in XML, but *only* when extended properties are negotiated with the IDE.

If extended properties are not negotiated, a debugger engine MUST NOT use this format, but instead use the normal format, even if this means returning invalid XML to IDEs.

An IDE can detect that a debugger engine has sent this extended format, by detecting that the `name` attribute on the `property` element is missing. If the name attribute is *not set*, then a debugger engine MUST provide name, fullname (optional), classname (optional) and value as sub elements of the <property> element, and encode their values in base64:

```
<property
    type="data_type"
    constant="0|1"
    children="0|1"
    size="{NUM}"
    page="{NUM}"
    pagesize="{NUM}"
    address="{NUM}"
    key="language_dependent_key"
    encoding="base64|none"
    numchildren="{NUM}">
    <name encoding="base64">...</name>
    <fullname encoding="base64">...</fullname>
```

```
    <classname encoding="base64">...</classname>
    <value encoding="base64">...</value>
</property>
```

The debugger engine MAY only pick this format if the extended_properties feature has been negotiated and SHOULD only pick this format if one of the attribute values for `name`, `fullname`, `classname` or `value` contain information that can not be represented as valid XML within attributes (such as `&#0;`).

7.12 Data Types

Languages may have different names or meanings for data types, however the IDE may want to be able to handle similar data types as the same type. For this reason, we define a minimal set of standard data types, and a method for specifying more explicit facets on those types. We provide three different type attributes, and a command to map those types to each other. The schema type serves as a hint to the IDE as to how to handle this specific data type, if it so chooses to, but should not be considered to be generally supported. If the debugger engine chooses to support Schema, it should handle all data validation itself.

language type:
A language specific name for a data type

common type:
A name used by the IDE to group data types that are similar or the same

schema type:
The XML Schema data type name as specified in the W3C Recommendation, XML Schema Part 2: Datatypes located at http://www.w3.org/TR/xmlschema-2/ The use of the schema type is completely optional. The language engine should not expect an IDE to support usage of this attribute. The IDE identifies support for this in the debugger engine by retrieving the data map, which would contain the schema type attribute.

7.12.1 Common Data Types

This is a list of the common data types supported by this protocol. For ease of documentation, and as hints to the IDE, they are mapped to one or more schema data types, which are documented at **http://www.w3.org/TR/xmlschema-2/**. Some non-scalar types are also defined, which do not have direct mappings to the base types defined by XML Schema.

Common Type	Schema Type
bool	boolean (The value is always 0 or 1)
int	integer, long, short, byte, and signed or unsigned variants
float	float, double, decimal

Common Type	Schema Type
string	string or other string-like data types, such as dateTime, hexBinary, etc.

Data types that do not map to schema:

null:
For example the "None" of Python or the "null" of PHP. Some languages may not have a method to specify a null type.

array:
for non-associative arrays, such as List in Python. Arrays have integers as keys, and the index is put in the name attribute of the property element.

hash:
for associative arrays, such as Dictionaries in Python. The only supported key type is a string, which would be in the name attribute of the property.

object:
An instance of a class.

resource:
Any data type the language supports that does not map into one of the common types. This could include pointers in C, various Python types such as Method or Class types, or file descriptors, database resources, etc. in PHP. Complex types may also be defined by using XML Schema, and mapping a type to the Schema type. This is a more specialized use of the type mapping, and should be considered experimental, and not generally available in implementations of this protocol.

undefined:
This is used when a variable exists in the local scope but does not have any value yet. This is optional, it is also correct to not return the property at all instead.

For the most part, this protocol treats array's and hashes in the same way, placing the key or index into the name attribute of the property element.

7.12.2 typemap_get

The IDE calls this command to get information on how to map language specific type names (as received in the property element returned by the context_get, and property_* commands). The debugger engine returns all data types that it supports. There may be multiple map elements with the same type attribute value, but the name value must be unique. This allows a language to map multiple language specific types into one of the common data types (eg. float and double can both be mapped to float).

IDE

```
typemap_get -i transaction_id
```

debugger engine

```
<response command="typemap_get"
          transaction_id="transaction_id"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <map type="common_type"
        name="language_type_name"
        xsi:type="xsd:schema_type_name"/>
</response>
```

Using the map element, a language can map a specific data type into something the IDE can handle in a more generic way. For example, if a language supports both float and double, the IDE does not necessarily need to distinguish between them (but MAY).

```
<map type="float"
     name="float"
     xsi:type="xsd:float"/>
<map type="float"
     name="double"
     xsi:type="xsd:double"/>
<map type="float"
     name="real"
     xsi:type="xsd:decimal"/>
```

Complex types may be supported if an implementation wishes to. Any implementation doing so should work without the other end having support for this:

```
<response command="typemap_get"
          transaction_id="transaction_id"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:mytypes="http://mysite/myschema.xsd">
   <map type="resource"
        name="SpecialDataType"
        xsi:type="mytypes:SpecialDataType"/>
</response>
```

7.13 property_get, property_set, property_value

Gets/sets a property value. When retrieving a property with the get method, the maximum data that should be returned is a default defined by the debugger engine unless it has been negotiated using feature_set with max_data. If the size of the property's data is larger than that, the debugger engine only returns the configured amount, and the IDE should call property_value to get the entire data. This is to prevent large data from slowing down debugger sessions. The IDE should implement UI that allows the user to decide whether they want to retrieve all the data. The IDE should not read more data than the length defined in the packet header. The IDE can determine if there is more data by using the property data length information. As per the context_get command, the depth of nested elements returned is either defaulted by the debugger engine, or negotiated using feature_set with max_children.

-d	stack depth (optional, debugger engine should assume zero if not provided)
-c	context id (optional, retrieved by context-names, debugger engine should assume zero if not provided)
-n	property long name (required)
-m	max data size to retrieve (optional, defaults to the length as negotiated through feature_set with max_data). **0** means unlimited data.
-t	data type (property_set only, optional)
-p	data page (property_get, property_value: optional for arrays, hashes, objects, etc.; property_set: not required; debugger engine should assume zero if not provided)
-k	property key as retrieved in a property element, optional, used for property_get of children and property_value, required if it was provided by the debugger engine.
-a	property address as retrieved in a property element, optional, used for property_set/value

IDE

```
property_get -n property_long_name -d {NUM} -i transaction_id
```

debugger engine

```
<response command="property_get"
          transaction_id="transaction_id">
    <property ... />
    ...
</response>
```

IDE

```
property_set -n property_long_name -d {NUM} -i transaction_id
         -l data_length -- {DATA}
```

debugger engine

```
<response command="property_set"
      success="0|1"
      transaction_id="transaction_id"/>
```

IDE

```
property_value -n property_long_name -d {NUM} -i transaction_id
```

debugger engine

```
<response command="property_value"
          size="{NUM}"
          encoding="base64|none"
          transaction_id="transaction_id">
    ...data...
</response>
```

When the encoding attribute is not present then the default value of "none" is assumed.

7.14 source

The body of the request is the URI (retrieved from the stack info), the body of the response is the data contents of the URI. If the file uri is not provided, then the file for the current context is returned.

-b	begin line (optional)
-e	end line (optional)
-f	file URI

IDE

```
source -i transaction_id -f fileURI
```

debugger engine

```
<response command="source"
          success="0|1"
          transaction_id="transaction_id">
    ...data source code...
</response>
```

7.15 stdout, stderr

Body of the request is null, body of the response is true or false. Upon receiving one of these redirect requests, the debugger engine will start to copy data bound for one of these streams to the IDE, using **6.4.2 stream** message packets.

-c	[0|1|2] 0 - disable, 1 - copy data, 2 - redirection 0 (disable) stdout/stderr output goes to regular place, but not to IDE 1 (copy) stdout/stderr output goes to both regular destination and IDE 2 (redirect) stdout/stderr output goes to IDE only.

IDE

```
stdout -i transaction_id -c 1
```

debugger engine

```
<response command="stdout"
          success="0|1"
          transaction_id="transaction_id"/>
```

8. Extended Commands

A IDE can query the debugger engine by using the feature_get command (see above). The feature strings for extended commands defined in this specification are the same as the command itself. For commands not listed in this specification, the prefix is 'xcmd_name'. Vendor or language specific commands may be prefixed with 'xcmd_vendorname_name'.

8.1 stdin

The stdin command has nearly the same arguments and responses as stdout and stderr from the core commands (section 7). Since redirecting stdin may be very difficult to support in some languages, it is provided as an optional command. Uses for this command would primarily be for remote console operations.

If an IDE wishes to redirect stdin, or cancel the stdin redirection, then it must send the stdin command with the -c argument, without any data. After the IDE has redirected stdin, it can send more stdin commands with the data. Sending both the -c argument and data in the same command is invalid.

If the IDE requests stdin, it will *always* be a redirection, and the debugger engine must not accept stdin from any other source. The debugger engine may choose to not allow stdin to be redirected in certain situations (such as when running under a web server).

-c	[0|1] 0 - disable, 1 - redirection 0 (disable) stdin is read from the regular place 1 (redirect) stdin is read from stdin packets received from the IDE.

IDE

```
stdin -i transaction_id -c 1
stdin -i transaction_id -- base64(data)
```

debugger engine

```
<response command="stdin"
          success="0|1"
          transaction_id="transaction_id"/>
```

8.2 break

This command can be sent to interrupt the execution of the debugger engine while it is in a 'run state'.

IDE

```
break -i transaction_id
```

debugger engine

```
<response command="break"
          success="0|1"
          transaction_id="transaction_id"/>
```

8.3 eval

Evaluate a given string within the current execution context. A property element MAY be returned as a child element of the response, but the IDE MUST NOT expect one. The string being evaluated may be an expression or a code segment to be executed. Languages, such as Python, which have separate statements for these, will need to handle both appropriately. For implementations that need to be more explicit, use the expr or exec commands below.

The eval and expr commands can include the following optional parameters:

-p	data page: optional for arrays, hashes, objects, etc.; debugger engine should assume zero if not provided — similar to the -p parameter for property_get.
-d	stack depth: stack depth at which the given code should be evaluated; debugger engine should assume zero if not provided.

IDE

```
eval -i transaction_id -- {DATA}
```

debugger engine

```
<response command="eval"
      success="0|1"
      transaction_id="transaction_id">
```

```
        <property .../>
    </response>
```

8.3.1 expr

expr, short for expression, uses the same command and response as eval above, except that it is limited to evaluating expressions. Only some languages support this functionality. expr should always return a property element if the expression is evaluated successfully. This command is specified for those applications that may need to implement this specific functionality. General uses of the protocol should not expect to find this command available, and should rely on eval above.

8.3.2 exec

exec, short for execute, uses the same command and response as eval above, except that it is limited to executing code fragments. Only some languages support this functionality. An IDE should not expect exec to return a value. This command is specified for those applications that may need to implement this specific functionality. General uses of the protocol should not expect to find this command available, and should rely on eval above.

8.4 spawnpoints

Spawnpoints are points in (currently Tcl) file where a new debug session might (i.e. if this position is a point in the code where a new application is created) get spawned when hit. Spawnpoints are treated much like a different type of breakpoint: They share many of the same attributes as breakpoints, using a *type*=="*spawn*" to distinguish themselves. Spawnpoints have an equivalent set of commands. A failure in any spawnpoint commands results in an error defined in **section 6.5**.

The following DBGP commands relate to spawnpoints:

spawnpoint_set	Set a new spawnpoint on the session.
spawnpoint_get	Get spawnpoint info for the given spawnpoint id.
spawnpoint_update	Update one or more attributes of a spawnpoint.
spawnpoint_remove	Remove the given spawnpoint on the session.
spawnpoint_list	Get a list of all of the session's spawnpoints.

A spawnpoint has the following attributes:

type	always set to "spawn"
filename	The file the spawnpoint is effective in. This must be a "file://" URI.

lineno	Line number on which spawnpoint is effective. Line numbers are 1-based. If an implementation requires a numeric value to indicate that *lineno* is not set, it is suggested that -1 be used, although this is not enforced.
state	Current state of the spawnpoint. This must be one of *enabled*, *disabled*.

Spawnpoints should be maintained in the debugger engine at an application level, not the thread level. Debugger engines that support thread debugging MUST provide spawnpoint id's that are global for the application, and must use all spawnpoints for all threads where applicable.

As for any other commands, if there is error the debugger engine should return an error response as described in **section 6.5**.

8.4.1 spawnpoint_set

This command is used by the IDE to set a spawnpoint for the session.

IDE to debugger engine:

```
spawnpoint_set -i TRANSACTION_ID [<arguments...>]
```

where the arguments are:

-f FILENAME	the *filename* to which the spawnpoint belongs [optional]
-n LINENO	the line number (*lineno*) of the spawnpoint [optional]
-s STATE	spawnpoint *state* [optional, defaults to "enabled"]

A unique id for this spawnpoint for this session is returned by the debugger engine. This *session spawnpoint id* is used by the IDE to identify the spawnpoint in other spawnpoint commands.

debugger engine to IDE:

```
<response command="spawnpoint_set"
          transaction_id="TRANSACTION_ID"
          state="STATE"
          id="SPAWNPOINT_ID"/>
```

where,

SPAWNPOINT_ID	is an arbitrary string that uniquely identifies this spawnpoint

	in the debugger engine.
STATE	the initial state of the spawnpoint as set by the debugger engine

8.4.2 spawnpoint_get

This command is used by the IDE to get spawnpoint information from the debugger engine.

IDE to debugger engine:

```
spawnpoint_get -i TRANSACTION_ID -d SPAWNPOINT_ID
```

where,

SPAWNPOINT_ID	is the unique *session spawnpoint id* returned by *spawnpoint_set*.

debugger engine to IDE:

```
<response command="spawnpoint_get"
          transaction_id="TRANSACTION_ID">
    <spawnpoint id="SPAWNPOINT_ID"
                state="STATE"
                filename="FILENAME"
                lineno="LINENO"/>
</response>
```

8.4.3 spawnpoint_update

This command is used by the IDE to update one or more attributes of a spawnpoint that was already set on the debugger engine via *spawnpoint_set*.

IDE to debugger engine:

```
spawnpoint_update -i TRANSACTION_ID -d SPAWNPOINT_ID [<arguments...>]
```

where the arguments are as follows. Both arguments are optional, however at least one should be provided. See **spawnpoint_set** for a description of each option:

-s STATE	
-n LINENO	

debugger engine to IDE:

```
<response command="spawnpoint_update"
          transaction_id="TRANSACTION_ID"/>
```

8.4.4 spawnpoint_remove

This command is used by the IDE to remove the given spawnpoint.

IDE to debugger engine:

```
spawnpoint_remove -i TRANSACTION_ID -d SPAWNPOINT_ID
```

debugger engine to IDE:

```
<response command="spawnpoint_remove"
          transaction_id="TRANSACTION_ID"/>
```

8.4.5 spawnpoint_list

This command is used by the IDE to get spawnpoint information for all spawnpoints that the debugger engine has.

IDE to debugger engine:

```
spawnpoint_list -i TRANSACTION_ID
```

debugger engine to IDE:

```
<response command="spawnpoint_list"
          transaction_id="TRANSACTION_ID">
    <spawnpoint id="SPAWNPOINT_ID"
                state="STATE"
                filename="FILENAME"
                lineno="LINENO"/>
    <spawnpoint .../>
    ...
</response>
```

8.5 Notifications

At times it may be desirable to receive a notification from the debugger engine for various events. This notification tag allows for some simple data to be passed from the debugger engine to the IDE. Customized implementations may add child elements with their own XML namespace for additional data.

As an example, this is useful for handling STDIN. The debugger engine interrupts all STDIN reads, and when a read is done by the application, it sends a notification to the IDE. The IDE is then able to do something to let the user know the application is waiting for input, such as placing a cursor in the debugger output window.

A new feature name is introduced: `notify_ok`. The IDE will call feature_set with the `notify_ok` name and a TRUE value (1). This lets the debugger engine know that it can send notifications to the IDE. If the IDE has not set this value, or sets it to FALSE (0), then the debugger engine MUST NOT send notifications to the IDE. If the debugger engine does not understand the notify_ok feature, the call to feature_set should return an error with the error code set to 3 (invalid arguments).

The debugger engine MUST NOT expect a notification to cause an IDE to behave in any particular way, or even to be handled by the IDE at all.

A proxy may also use notifications, during a debug session, to let the IDE know about events that happen in the proxy. To do this, the proxy will have to listen for `feature_set` commands and keep track of the values set, as well as passing them through to the debugger engine.

IDE initialization of notifications:

```
feature_set -i TRANSACTION_ID -n notify_ok -v 1
```

debugger engine notifications to IDE:

```
<notify xmlns="urn:debugger_protocol_v1"
        xmlns:customNs="https://example.com/xmlns/debug"
        name="notification_name">
    <customNs:.../>
    TEXT_NODE or CDATA
</notify>
```

8.5.1 Standard Notifications

The following list of notifications are standardized for the protocol. Other notifications may be added by other implementations. It is suggested that notification names not found in this document are preceded with 'XXX' or some

similar tag as a means of preventing name conflicts when new notifications get added to the protocol in the future.

Name	Description
stdin	notification occurs when the debugger engine is about to read the stdin pipe.
breakpoint_resolved	Notification occurs when the debugger engine has resolved a breakpoint. The returned notification includes the same elements as a return from 7.6.2 breakpoint_get. The *breakpoint* element becomes a child element of the *notify* element instead of *response*. The *resolved* attribute should always be set to *resolved*. A debugger engine MUST NOT send this notification if *resolved_breakpoints* has not been enabled with 7.2.3 feature_set. A debugger engine MAY send multiple notifications for the same breakpoint ID, but only if their attributes have changed (again).
error	notification occurs when the language engine issues debugging information.

8.5.2 Error Notification

When a language engine creates a debugging notification, the debugger engine MAY convert this to a DBGp notification. As an example, this can be used to convert PHP's Notices and Warnings to DBGp notifications.

With the `notify_ok` feature set, a notification like below COULD be returned by the debugger engine.

An `error` notification MAY include additional information in the body of the `notify` element. As notifications come straight out of the debugger engine, the data body returned in this packet is of arbitrary encoding. If body data is present, the debugger engine MUST include which encoding is used through the `encoding` attribute on the `notify` tag. Currently, only the `base64` encoding is supported.

This extensive XML snippet displays through `<xmlns:xdebug`, how XML namespaces SHOULD BE used for providing additional information:

```
<notify xmlns="urn:debugger_protocol_v1"
        xmlns:xdebug="http://xdebug.org/dbgp/xdebug"
        name="error"
        encoding="base64">
    <xdebug:message filename="file:///tmp/xdebug-dbgp-test.php"
                    lineno="5"
                    type="Notice">
        <![CDATA[Undefined variable: bar]]>
    </xdebug:message>
```

```
    Im90aWN1O1BVbmR1ZmluZWQgdmFyaWFibGU61GJnc1BpblBmaWx1O18vL3RtC94ZGV1dWctZG.
</notify>
```

8.6 interact - Interactive Shell

The interact command allows an IDE to send chunks of code to be compiled and executed by the debugger engine. While this is similar to the eval command, it has a couple important differences.

First, it buffers code sent to it until a successful compile is achieved. The buffering allows the IDE to send a line of code for each call to the interact command, which reflects a user typing code into a console. Each line is joined in the debugger engine with a newline character. As soon as a successful compile happens, the code is run and any output returned to the IDE (via stdout/stderr or otherwise).

Second, it returns a prompt string that can be used by the IDE as an input prompt for the user.

The interact command can only be called during a break or interactive state.

The debugger engine implementation MAY also be designed to work in and interactive-only mode, where there is no script being debugged, and all code is received through the interact command. This allows the protocol to be utilized for the purpose of a pure interactive shell for the language.

Control characters should be sent in the data section of the command, and the debugger engine should handle the control characters in a way that is expected by the implementation. These characters can include Ctrl-C (KeyboardInterupt in Python) and other such console like controls. The IDE should not expect the debugger engine to handle control characters in any specific way.

The IDE can query the debugger engine for interact support using the feature_get command.

The 'filename' in the stack for an interactive session should be '<console>' or some other string to denote a console stack level.

The debugger engine is not required to enable debugging of code received via the interact command, however it should provide access to other information, such as the variables retrieved via context_get.

IDE to debugger engine:

```
interact -i TRANSACTION_ID -m mode -- base64(code)
```

where,

-m mode	a mode of zero tells the interact command to clear the buffer and any other state that was maintained for previous interact commands. The prompt attribute returned should be an empty string.

debugger engine to IDE:

```
<response command="interact"
    transaction_id="TRANSACTION_ID"
    status="STATUS_NAME"
    more="CONTINUE_FLAG"
    prompt="PROMPT" />
```

where,

STATUS_NAME	A valid status name from the list of status names in section 7.1. A new name is added specificaly for this command which is 'interactive'. The interactive status is returned unless the mode in the command was zero, in which case the status will be up to the debugger engine (typically the last status before running interact), or some error has occured that causes a different status.
CONTINUE_FLAG	a boolean which is true if the interact command requires more code to compile successfully
PROMPT	a string containing the prompt for the next line of code

A. ChangeLog

2021-05-04

- 7.2.1, 7.5 Added 'breakpoint_details' feature to include breakpoint info in the response to applicable continuation commands.

2021-01-11

- 7.6 Fixed duplicate mention of **7.2.2 feature_get**, where one of them should have been **7.2.3 feature_set**.

2019-04-06

- 7.2.1, 7.6 Change the 'breakpoint_resolved' notification to an opt-in feature.

2018-01-17

- 8.5.2 Clarify that the body (and encoding attribute) is optional.

2018-01-09 - draft 21

- 6.3.1 Added section detailing escaping rules for sending arguments to the debugger engine. Specifically crafted to prevent confusion around sending property names.
- 7.11 Clarify that IDEs must not apply any default escaping rules to read/write properties through property_get, property_set, and property_value.
- 7.11.1 Added header and describe extended properties better.

2017-07-10 - draft 20

- 5.3 Fixed the text and example for proxyinit. It doesn't accept "-a ip:port" but just "-p port".

2017-02-14

- 7.2.1 Add the undocumented 'supported_encodings' feature

2017-02-06

- Created separate sections (6.4.1-6.4.3) for each data packet format
- Clarify that the `stream` and `notify` packets encode the data with base64, and that that is signalled by the `encoding="base64"` attribute on these elements

2016-12-24 - draft 19

- 8.5.2 Added section on Error Notifications

2016-08-31

- 8.3 Added -d stack depth parameter to eval command (Max Sherman)

2016-06-07

- 7.2.1, 7.6 Add 'resolved' attribute to breakpoint_set, breakpoint_get and breakpoint_list
- 7.5 Added 'breakpoint_resolved' notification.

2015-11-20

- Fixed typos in IDE commands "feature-get" to "feature_get", "feature-set" to "feature_set", and "context id" to "context_id".

2015-11-11

- 7.7 Fixed type in IDE command from "stack-depth" to "stack_depth"

2013-10-01

- 7.13 Clarified use of the -m option.

2013-06-22

- 7.2.2 / 7.11 Added the extended property format and extended_property feature negotiation.

2012-03-29

- 6 Clarified what "Pythons Cmd module" means for quoting values that contain spaces.

2010-01-20 - draft 17

- 7.6 / 7.6.2 Added the missing "expression" argument to information that can be stored for breakpoints, and returned through breakpoint_get.

2009-12-30

- 8.3 Added the -p parameter to eval and expr, to control which pages are shown in case the returned property is an array, hash or object with more than "pagesize" children.

2007-07-14 - draft 16

- 6.3 Fixed binary encoding comments regarding data.
- 6.3 Clarified that the transaction ID is supposed to be numerical.
- 6.5.1 Mention that error code three is also for "invalid values to an option".
- 7.2.1 Clarified encoding feature.
- 7.2.2 Clarified the supported attribute for feature_get.
- 7.6 Added missing required attribute "filename" to the "conditional" breakpoint type.
- 7.6 Added missing "dbgp:" URI scheme to "filename" breakpoint option.
- 7.6.1 Added a comment that it is up to the engine on how to handle duplicate breakpoints.
- 7.6.1 Clarified on how expression evaluations affect breakpoint activation.
- 7.6.4 Added Xdebug's practise of returning the deleted breakpoint's information as an optional child element.
- 7.9 Clarified that the context's ID attributes are numerical.
- 7.11 Marked "key" and "address" attributes as "optional
- 7.13 The -a option is not required if the address is provided. Implementation of this option could possible allow reading at random memory addresses which is a security issue.
- 7.13 Clarified on how the -p option is used.
- 8.5 Fixed feature_set command in example, it does not use command data, but the -v option for specifying the value.

2007-03-31

- 7.6.1 Fixed breakpoint_set example and note that the breakpoint types are listed above and not below.

2006-01-24

- 7.2.1 Added a description of the breakpoint_types feature.

2006-01-23

- 7.11 Clarified the behavior of paging regarding depths, and that paging of arrays/objects/hashes is 0 based.
- draft 15

2004-11-03

- 7.12.1 Added the 'undefined' type.

2004-10-28

- 6 Clarify encoding for data passed in commands with the -- option.
- 7.13 Clarify the default encoding for property values.

2004-05-16

- 5.3 add address and port attributes to the proxyinit element returned to the ide by the proxy.

2004-05-12

- **7.2 reorganize the feature names, add a couple missing names** (supports_postmortem, show_hidden, notify_ok).

2004-04-05

- 8.5 New notification support
- 8.6 New interact command

2004-02-20

- 1.2 moved the change log to appendix A
- 5 massive reorganization of section 5
- 5.3 expanded description of proxies and just in time debugging.
- 5.4 expand description of multisession and multithreaded debugging.
- 7.2 A new feature name, breakpoint_languages, has been added. This option is only required if the engine supports more than one language.
- 7.2 and 7.3 Remove crufty documentation that still referred to old binary protocol information.

- 7.6.1 For conditional breakpoints the expression has been moved to the data section of the command.
- 8.3 remove the length argument in the eval command, it is unnecessary.
- 8.3 be more explicit about how eval works, add 8.3.1 expr and 8.3.2 exec as additional optional commands that can be used in special implementations.
- 8.4 Remove the 'delete' state, this was old and removed in breakpoints.

2004-01-28

- 7.8 Fix cmdbegin/end attributes for stack_get

2004-01-09

- 5.1 New DBGP_IDEKEY environment variable

2004-01-07

- 7.5 renamed the stop and kill commands to detach and stop, added some clarification to the description of the commands.

2003-12-16

- 7.6, 8.4 re-write the breakpoint and spawnpoint sections to be clearer

2003-12-09

- 6.7 new section describing dbgp file protocol
- 7.6 better document breakpoints

2003-12-05

- 6 Change the deliminator for command data to '--'. This conforms to standard getopt libraries.
- 7.11 remove the recursive attribute, if an IDE wants to handle circular references, it can do so based on the address attribute if the engine provides it.

2003-12-02

- 7.6 remove breakpoint_enable/disable, and add breakpoint_update command. Enable/disable states are changed through breakpoint_update.
- 8.4 new (optional) spawnpoint commands

2003-11-25

- 7.6 Change the breakpoint *hits* and *ignore* attributes to *hit_count*, *hit_value* and *hit_condition* to add functionality available in VS.NET and to simplify usage. Also clarify some other breakpoint attribute legal values.

2003-11-24

- 7.5 correct the stop command documentation, stop is 'detach', and does not allow for continued interaction. Document how expressions are returned from breakpoint_get.
- 7.8 correct old documentation on the stack element. Add new attributes: where, cmdbegin, cmdlength. Provide further documentation about all the attributes.

2003-11-20

- 5.1 better define session keys vs. ide key for proxy, document how proxy works better.
- 7.6 better document attributes and hit option

2003-11-18

- 7.1 Clarify stopping and stopped states
- 7.5 Clarify the stop command
- 7.6 Remove 'temporary' as a status for breakpoints, make it an option in the command line. Remove the 'function' breakpoint type, provide two new types, 'call' and 'return'. Add 'hits' option to allow a breakpoint to be ignored a number of times before being used.

2003-11-12

- draft 12
- Rest markup tweaks

2003-11-09

- draft 11
- 7.12 new section inserted as 7.12. This section specifies common data types, and how to map more specific data types to the common types.
- 7.11 two new optional attributes, classname and facet, that provide additional hints to the IDE about the nature of the property. New key attribute for language specific keys to properties.
- 6.5 new section, 6.5.1 for defining common error codes.

2003-11-05

- spelling fixes
- 5.1 change proxy options
- 7.6 clarify breakpoint command options
- 7.12 fix old text about context names

2003-10-15

- 6 remove the first NULL in the command structure from IDE to debugger engine. This makes dealing with those commands easier.

- 6.6 NEW File paths must be URI's.
- 7 source command returns the source for the current context if no file uri is provided.
- 7 added sub-item numbering
- 7.1 clarify the status values
- 8 added sub-item numbering

2003-10-09

- 7 remove run_to, unnecessary
- 7 remove 'step', there is no generic step command
- 7 clarify continuation commands
- 7 clarify breakpoints

2003-10-07

- more layout changes for reStructuredText

2003-10-06

- reformat to **reStructuredText markup**
- 6 clarify message packets
- 6.3 clarify command packets
- 7 clarify feature_get/set
- 7 allow error results on breakpoints if a type of breakpoint is not supported by a debugger engine.
- 7 add recursive attribute to properties, and clarify the address attribute and how recursive data is handled.
- 7,8 moved stdin to the optional commands section

2003-10-02

- 5.1 changed proxy error to be the same as that in 6.5
- 5.1 the IDE and proxy ports have been defined to 9000/9001
- 5.3 exclude protocol overhead from data size definition
- 6.2 changed typo 'stdin and stdout' to 'stdout and stderr'
- 6.5 changed error id to error code
- 7 removed comments on 'body' from the run commands
- 7 clarified 'source' command arguments to be optional
- 7 added 'disable' option to stdin/out/err commands
- 7 breakpoint arguments and types have been better defined since not all arguments need to be required for all types
- 7 the expression breakpoint type has been removed since it is covered by the conditional breakpoint type

2003-09-30

- section numbers added, changes below are marked with the section number
- 3 Terminology changed (frontend -> IDE, backend -> debugger engine)
- 5.1 added response packet from proxy to IDE when IDE issues the proxyinit command.
- 5.1 the proxy now adds a proxyclientid to the init packet from the debugger engine when it passes the packet through to the IDE.
- 5.1 the proxy must be able to send errors to the IDE, for instance, if it looses the connection to the debugger engine.
- 5.1 the proxy must be able to send errors to the Debugger, for instance, if it looses the connection to the IDE.
- 5.3 added new section to help better define feature negotiation with feature_get/set commands.
- 6 packets have been better defined. This section has also been reorganized.
- 6.2 the communication of packets has been rewritten.
- 7 feature_get/set have some modifications.
- 7 context_get and property_* commands have been modified to better reflect negotiation of features using the feature_get/set commands.
- 7 property_* commands have been commented a bit more, and an additional argument is available for paging arrays, etc.
- 7 The definition of the property tag has been modified
- 7 stdin command has been modified, the debugger engine may choose to not redirect stdin.
- 7 status command modified to support the async state
- 7 source command now accepts begin and end line arguments for retrieving only parts of a file.
- 7 stack_get now defines an enumeration for the stack
- 8 break command clarified so it can only be sent while the debugger engine is in a run state.
- 8 eval can return a property as part of the response