TASKING Script Debugger User Guide

TASKING Script Debugger User Guide

Table of Contents

Chapter 1. Using the Stand-alone Script Debugger

The TASKING VX-toolset for C166 contains two debuggers. One debugger is integrated in the Eclipse environment and the other is a separate program, the stand-alone script debugger. This chapter describes the stand-alone debugger.

The stand-alone script debugger is not a complete debugger; facilities such as a register or a memory window are not available. Instead, its primary purpose is to run scripts created by the user for testing purposes.

The recommended way of using the program involves the following steps:

1. Create a script file in a text editor. The directory `examples/dbg166` contains a few example scripts to get you started. For details about the script language see Chapter 2, *Debugger Script Language*. In particular Section 2.16.6, *Debugger Specific Functions* describes the script language functionality that you can use to access and control the target.

2. Test the correctness of the script (i.e. debug the script itself) using the script debugger in interactive (graphical user interface) mode. This also involves creating a configuration file (`.dcf`) for the target to be used.

3. Once the script is correct, run it from the command line (possibly from a batch file):

   ```
   dbg166 [options] name_of_script_file
   ```

1.1. Run the Debugger in Interactive Mode

To start the script debugger select **Script Debugger** from the **Start** menu. The program starts with an empty window except for a menu bar at the top. The area below that is used for so-called panes. You can resize a pane by dragging one of its four corners and you can move a pane by dragging its title.

1.1.1. Configure the Debugger

When the script debugger is started, it creates a default configuration, called "Untitled", that uses the simulator and the C167 CPU. If this default is not suitable, you need to change the configuration.

1. From the **File** menu, select **Edit configuration...**

    The Configuration dialog appears.
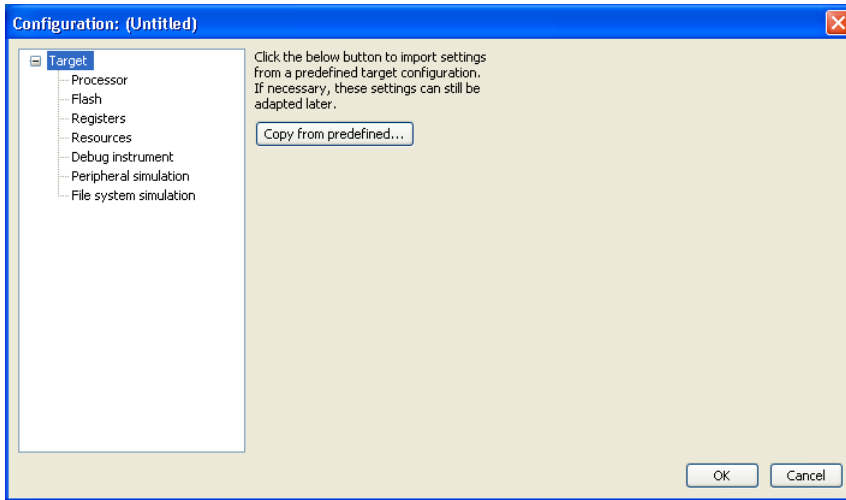
The dialog consists of several panes, which you can select on the left-hand side.

2. Select **Target** and click on the **Copy from predefined...**

    The Copy from Predefined dialog appears. This dialog shows all the predefined target configurations that come with the toolset.
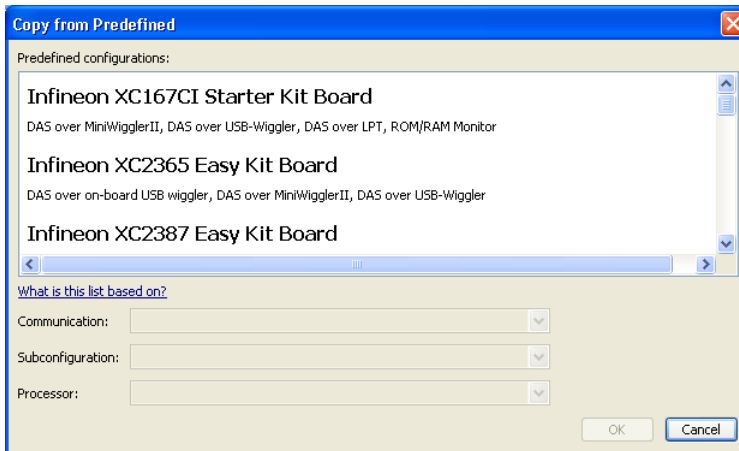
The information in this dialog is based on Debug Target Configuration (DTC) files. DTC files define all possible configurations for a debug target. The files are located in the `etc` directory of the installed product and use `.dtc` as filename suffix. For more information on DTC files, see the *TASKING VX-toolset for C166 User Guide*.

3. Select a predefined configuration and click **OK**.

    The settings will be copied to your configuration.

4.   Optionally, adapt the settings in the other panes. For example, in the **Registers** pane you can add register settings that may depend on your particular hardware situation and application program.

5.   Click **OK**.

6.   From the **File** menu, select **Save configuration as...**

     The Save Configuration dialog appears.

7.   Give your configuration file a name with extension `.dcf`.

Global program settings

A few settings that are not expected to be different for different configurations are stored separately in a "global" file `settings.dst`. For example, you can extend the list of processors.

1.   From the **File** menu, select **Settings...**

     The Settings dialog appears.
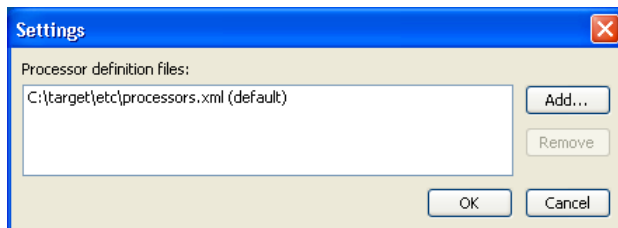
2.   Click **Add...**

     The Add Processor Definition File dialog appears.

3.   Select the file that contains additional processor definitions and click **Open**.

     The new file will be added to the list of processor definition files.

4.   Click **OK**.

     The new settings will be saved automatically when you exit the program.

1.1.1.1. Setup a Flash Device

In the Flash pane you can setup a flash device. With the TASKING script debugger you can download an application file to flash memory. Before you download the file, you must specify the type of flash devices you use in your system and the address range(s) used by these devices.

To program a flash device the debugger needs to download a flash programming monitor to the target to execute the flash programming algorithm (target-target communication). This method uses temporary target memory to store the flash programming monitor and you have to specify a temporary data workspace for interaction between the debugger and the flash programming monitor.

Setup a flash device

1. From the **File** menu, select **Edit configuration...**

    The Configuration dialog appears.

2. In the left pane, select **Flash**.

    The Flash pane appears.

3. Click **Add...** to specify a flash device.

    The Add Flash Device dialog appears.

4. In the **Device type** box, expand the name of the manufacturer of the device and select a device.

Based on your selection the other fields are filled in, but you can adapt them manually.

5. In the **Base address(es)** field enter the start address of the memory range that will be covered by the flash device. Any following addresses separated by commas are considered mirror addresses. This allows the flash device to be programmed through its mirror address before switching the flash to its base address.

6. In the **Chip width** field, select the width of the flash device.

7. In the **Number of chips** field, enter the number of flash devices that are located in parallel. For example, if you have two 8-bit devices in parallel attached to a 16-bit data bus, enter 2.

8. Fill in the **Number of unused address lines** field, if necessary.

9. Click **Add**.

   The new flash device is added to the Flash pane.

10. In the **Monitor file** field, specify the filename of the flash programming monitor, usually an Intel Hex or S-Record file.

11. In the **Sector buffer size** field, specify the buffer size for buffering a flash sector.

12. Specify the data **Workspace address** used by the flash programming monitor. This address may not conflict with the addresses of the flash devices.

To program a flash device

To program a flash device during downloading, put a `$download` command in your script file with the `flash` option enabled.

Example:

```
$failed = $download("myprogram.elf", {"flash" : 1});
```

1.1.2. Run and Debug a Script

Debugger scripts use a proprietary language described in Chapter 2, *Debugger Script Language*. The program itself does not provide script editing facilities, but you can use any text editor to create a script file. The recommended extension is `.dscr`.

Once you have created a script, you can run a script.

1. From the **File** menu, select **Run script...**

   The Run Script dialog appears.

2. Select the script file (with extension `.dscr`) you want to run and click **Open**.

   The script file opens in a separate pane. We used download.dscr as an example.
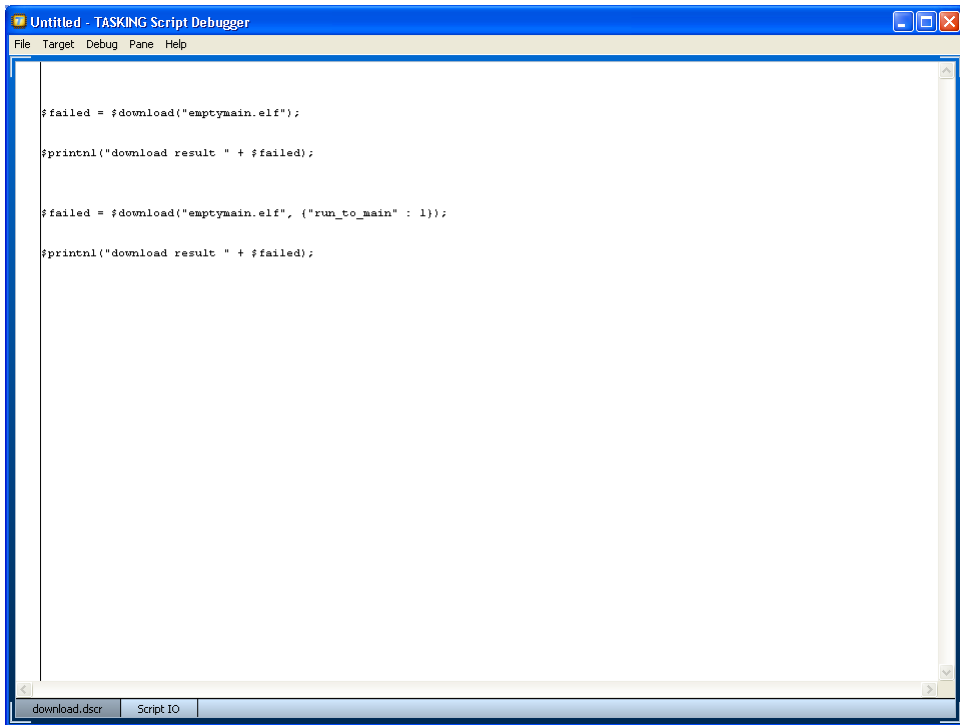
Unless this has already happened, a connection with the target will be established first. Then, provided the script contains no syntax errors, the script itself will run. Output of the script is printed in a separate pane (`Script IO` in the example above).

Debugging a script

If you suspect your script contains an error, you can debug it by inserting `$printnl` calls or by placing breakpoints. To place a breakpoint, click in the left-hand margin of the script pane. A breakpoint is represented by a yellow dot 🟡 when it has not actually been placed yet, for example before a connection to the target has been made, and a red one 🔴 thereafter.

Once your script halts on the breakpoint, you can examine the program state by selecting **Variables** or **Expressions** from the **Pane** menu. If you want to examine a higher stack level, select **Threads** and double click on the relevant stack frame. With the commands in the **Debug** menu, you can control the execution of a script thread.

1.2. Run the Debugger from the Command Line

Once your script is correct, you can run it non-interactively as well. Enter the following command:

dbg166 [*options*] *script_file*

The following options are available:

Option	Description
-? / **--help**	This option causes the program to display an overview of all command line options.
--arg=*string*	This option allows you to pass arguments to a script. In the script you can use the $getargs() function to access the string. You can use the option more than once, in which case the successive strings will appear in the array returned by $getargs() in the original order.
-c=*configuration_file*	This option allows you to specify a configuration file (.dcf). If this option is omitted, the default (simulator) configuration is used.
--fss-initial-dir[=*directory*]	This option allows you to specify the initial directory for file system simulation. If the *directory* is omitted, the current directory is used. If this option is omitted, the path from the configuration is used.
-V / **--version**	This option causes the program to display version information and then exit.

Once the script has finished executing, the debugger will terminate automatically.

Chapter 2. Debugger Script Language

The debugger features a proprietary script language that you can use to automate various tasks, for example to regularly perform a set of tests. This chapter discusses the syntax and semantics of the script language. Section 2.16.6, *Debugger Specific Functions* describes the script language functionality that you can use to access and control the target. Chapter 1, *Using the Stand-alone Script Debugger* describes how you can use the debugger script to control the debugger.

2.1. Introduction

The debugger script language is powerful, but relatively simple, borrowing concepts from various existing languages such as C/C++, Python and Java.

For the convenience of readers who are familiar with most of these languages and who want to make a quick start, the language's key properties are listed below:

- Partly object-oriented (OO): has classes, but no access control, no inheritance and no overloaded operators.

- Garbage-collected: objects are automatically deleted when they have become unreferenced.

- Strings are Unicode, not ASCII.

- Weakly typed: type compatibility is checked at run-time, not at compile-time.

- Features indexed arrays (any dimension, "jagged") and associative ("hash") arrays (in which the keys can be numbers and / or strings).

- Exceptions are thrown when certain run-time errors occur.

- Objects are passed by value or by reference.

- Control flow statements similar to those of C, plus a `foreach` statement.

- Operators similar to those of C, plus `=ref`.

- Script identifiers must begin with a dollar sign ($), which helps distinguish them from target language identifiers. Script identifiers are case-sensitive.

- Built-in regular expression pattern matching functionality.

2.2. Identifiers

The names of user-defined variables, functions, etc. must begin with a dollar sign ($), which must be followed by at least one alphabetic character ([A-Za-z]), optionally followed by one or more alpha-numeric characters or underscore characters ([A-Za-z0-9_]).

Examples of a few valid and invalid identifiers:

valid: `$a`, `$AbC`, `$a_4`.

invalid: `abc`, `$0`, `$_abc`.

There is no maximum length for identifiers and they are case-sensitive.

Identifiers beginning with `$_` are used or reserved for internal purposes. With the exception of those mentioned in this document, they are not accessible.

2.3. Special Identifiers

The following table lists identifiers that have a special meaning.

Identifier	Description
`$__FILE__`	evaluates to a string equal to the name of the script file, for example "myscript.dscr"
`$__LINE__`	evaluates to a number equal to the (one-based) line number where this identifier appears
`$args` and `$_args`	see Section 2.12.4, *Variable Argument List*
`$global`	global scope prefix; (see Section 2.12.2, *Accessing Global Variables*)
`$this`	class instance (see Section 2.13, *Classes*)

2.4. Whitespace and Comments

Whitespace is significant in essentially the same situations as it is in C. You can include C++ style source comments or use the number sign (#), which extends to (but does not include) the next line break. Comments of the form `/* … */` can be nested. Here are some examples:

```
$c =
$a + $b; // Same as $c = $a + $b;

/* Following /* code */ will be executed. */ $printnl("Hello");

$printnl("world."); # This is a comment.
```

2.5. Include Statements

Using `include` "*file name*", a script can include another script. This inclusion takes place only once, when the including script is compiled.

2.6. Nil, $defined(...) and $delete(...)

Internally, every variable and expression is represented by a so-called "handle". This handle can refer to an existing object, but may also be *nil*, which is comparable to `NULL` in C/C++. Note however that `nil` is *not* a keyword.

In many cases, if you try to use a *nil* handle, an error occurs and the script will cause an exception of type `"#NIL_OBJECT"`.

With $defined(*variable*) you can check whether a variable exists. It will yield 0 if the specified variable represents either a nil handle or does not exist at all, and 1 otherwise.

With $delete(*variable*) you delete the specified variable. After this, $defined(…) on the same variable will return zero.

You can use $defined(…) and $delete(…) with single identifiers, but also with arrays, as will be explained later.

2.7. Types, $type(...)

The types supported by the script language are shown in the following table. With $type(*variable*) you can retrieve the type of the specified variable. Unlike C/C++, but like Python, the language is weakly typed, meaning that the type of a variable is not fixed and not known at compile time. This means that if you try to execute "$c = $a + $b;", you have to make sure that the current types of $a and $b are compatible and support the addition operator; the script language compiler cannot check this when it compiles a script. In the case above, it will throw an exception if it cannot perform the addition (see Section 2.15, *Exceptions*) .

Type	Corresponding $type(…) value	See
number	"NUMBER"	Section 2.7.1, *Numbers*
string	"STRING"	Section 2.7.2, *Strings*
indexed array	"INDEXARRAY"	Section 2.7.3, *Indexed Arrays*
associative array	"ASSOCARRAY"	Section 2.7.4, *Associative Arrays*
class instance	"CLASS"	Section 2.13, *Classes* and $instance_type
reference to function	"FUNCTIONREF"	Section 2.12, *Functions*

There is no separate Boolean type: operators such as && yield a number equal to 0 or 1.

2.7.1. Numbers

All numbers in the debugger script language are floating-point numbers, i.e. there are no separate types for signed and unsigned integers, for example.

Literal numbers can be specified in three forms, as shown in the following table. Neither digits nor special characters such as e and x are case-sensitive.

Form	Optional exponent indicator	Examples
decimal	e or E	-1234 +1234.5 -1.1e+03
octal	(none)	-0172 +0411

Form	Optional exponent indicator	Examples
hexadecimal	`p` or `P`, indicating a power of two	`-0x1234abc` `+0xa.8p1` (equal to $10.5 \times 2 = 21$)

Division by zero will cause an exception of type `"#DIV_BY_ZERO"`.

See Section 2.16.2, *Functions Applicable to Numbers* for a list of built-in functions that can be used with numbers.

2.7.2. Strings

Script language strings can have an arbitrary length and can be manipulated using a wide range of built-in functionality. Strings consist of Unicode characters, although the debugger only reads ASCII input files.

Literal strings have a syntax similar to that of C. Special characters can be specified using the backslash character as an escape, as shown in the following table.

Input	Result
`\\`	single backslash character
`\'`	single quote character
`\"`	double quote character
`\a`	bell character
`\b`	backspace character
`\e`	escape character
`\f`	form feed
`\n`	line feed
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\0` plus zero or more octal characters	the character implied by the octal value, e.g. `"\0103"` for `"C"`
`\x` plus one or more hexadecimal characters (lower- or uppercase)	the character implied by the hexadecimal value, e.g. `"\x20ac"` for `"€"`
`\` followed by a newline in the source	line continuation, see below
`\` followed by any other character	that character

Strings can be concatenated with the + operator:

```
$a = "app";
$b = "le";
$printnl($a + $b); // Prints "apple".
```

This can be useful when constructing very long literal strings. You can achieve the same effect by using the backslash character as a line continuation indicator:

```
$a = "app\
le";
// $a equals "apple".
```

Without the backslash, the above example has a different meaning:

```
$a = "app
le";
// $a equals "app\nle".
```

Besides the + operator, the only operators that apply to strings are: =, +=, == and !=.

See Section 2.16.3, *Functions Applicable to Strings* for a list of built-in functions that can be used with strings.

2.7.3. Indexed Arrays

The script language supports indexed arrays which are zero-based and can have a variable length. A single array can contain elements of different types. An array element can even be an array itself, resulting in a multi-dimensional array. If an array consists of elements of different dimensions, this is sometimes referred to as a "jagged array".

You can create an array with an initializer of the form [*element 0*, *element 1*, …]:

```
$a = [];             // Empty array.
$b = [1, "hello"];   // Mixed-type array.
$c = [ [0, 1], 2 ];  // Jagged array.
```

Array elements can be accessed using the [*index*] operator. If the specified element does not exist, an "#INVALID_INDEX" exception is thrown. With the function $defined(…) you can check the existence of an array element.

Array elements do not necessarily have consecutive indices:

```
$a[2] = 3.5;
$a[4] = "hello";
$printnl($a);              // Prints [<NIL>, <NIL>, 3.5, <NIL>, "hello"].
$printnl($defined($a[2])); // Prints 1.
$printnl($defined($a[3])); // Prints 0.
```

See also Section 2.16.4, *Functions Applicable to Indexed Arrays*.

2.7.4. Associative Arrays

Associative or hash arrays are data structures tying a "key" (a number or a string) to a "value" (any type). The value can be of any type (including an array or associative array). Associative arrays are unordered.

You can create an associative array with an initializer of the form { *key 0*: *value 0*, *key 1*: *value 1*, …}. Individual elements can be accessed using the {*key*} operator. If the specified element does not

exist, a `"#KEY_NOT_FOUND"` exception is thrown. With the function `$defined(…)` you can check the existence of an array element.

```
$a = {};                                    // Empty associative array.
$b = 5;
$c = {0 : $a, "key" : "value"};             // Mixed-key associative array.
$printnl($c{"key"});                        // Prints "value".
$printnl($defined($c{"non-existent key"})); // Prints 0.
$printnl($c{"non-existent key"});           // Gives exception.
```

See also Section 2.16.5, *Functions Applicable to Associative Arrays*.

2.8. Operators

The operators provided by the script language are listed in the following table. Except where noted otherwise, they have the same meaning as in C and their precedence and associativity is an in C.

Operator	Description
(…)	function call
[…]	selection of indexed array element
{…}	selection of associative array element (precedence and associativity as […])
.	selection of class member or scope resolution
++, --	post-increment/decrement and pre-increment/decrement
+, -	unary plus and minus
!	logical negation (yields 1 or 0)
+, -, *, /, %	addition, subtraction, multiplication, division and modulus
<<, >>	bitwise shift left and right
&, |, ^	bitwise AND, OR and XOR
==, !=	comparison operators that apply to both numbers and strings
<, <=, >, >=	number-only comparison operators
&&, ||	logical AND and OR
=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	assignment by value and assignment variants of several of the above operators
=ref	assignment by reference (precedence and associativity as =) ; see Section 2.9.1, *Assignment by Value or by Reference*

2.9. Assignment

The following sections describe the details about assigning a value to a variable.

2.9.1. Assignment by Value or by Reference

When you assign an expression to a variable using the = operator, the script performs an assignment "by value". This means that a "deep copy" of the expression is made. After $b = $a; modifying $a will not affect $b. If $a is an array, all its elements will have been copied (deep copy), so modifying those will not affect the elements of $b.

Note that the source object of a deep copy can contain duplicate objects and even "cycles" (objects that directly or indirectly refer back to themselves). The deep copy will reflect this all.

The =ref operator behaves differently: the effect of $b =ref $a is to make the two variables point to the same object. If $a is an array, modifying any of its elements will affect $b.

The difference is illustrated in the following example:

```
$arr = [1, 2, 3];
$num = 5;

$arr_v = $arr;      // deep copy
$num_v = $num;

$arr_r =ref $arr;   // reference
$num_r =ref $num;

$arr[3] = 4;        // change source object
$num++;

$printnl($arr_v);   // Prints [1, 2, 3];
$printnl($num_v);   // Prints 5;

$printnl($arr_r);   // Prints [1, 2, 3, 4];
$printnl($num_r);   // Prints 6;
```

2.9.2. Assignment of Literals

For performance reasons, literal numbers and strings are treated as constants, so they cannot be modified. This is not a problem in code such as $a = 3 because, as discussed above, a (modifiable) copy will be made. You can also assign by reference to a constant value but only if you do not try to modify the object constant afterwards:

```
$a =ref "hello";
$printnl($a);        // Allowed
$a = "world";        // Throws "#MODIFIYING_CONSTANT" exception
$a =ref "world";     // Allowed
```

To use $a again for a mutable object, you either have to delete and recreate the variable $a, or you should let it refer to a mutable object:

```
$a =ref "hello";
$a = $length($a)      // Throws "#MODIFIYING_CONSTANT" exception

$a =ref $length($a); // Reference a mutable object
$a++;                 // Allowed

# OR:
$a =ref "hello";
$delete($a);          // Delete $a
$a = "world"          // Allowed
```

In general, it is recommended to use `=ref` if performance matters.

2.9.3. Assignment versus Expression

In C/C++, the result of an assignment is an expression, but in the script language this is not the case. Consequently, code such as:

```
if (($a = $length($b)) > 0)
{
…
}

$c = $b = $a;
```

gives a syntax error. The first example needs to be written as:

```
$a = $length($b);
if ($a > 0)
{
…
}
```

The second example needs to be rewritten as:

```
$a = $c;
$b = $c;  // or: $b = $a
```

Elsewhere in this document, the syntactical elements *expression* and *assignment* are used to make the difference more explicit.

2.10. Resolving of Identifiers

The script language differs from C in that identifiers are resolved at run-time. One consequence of this is that the lexical order of definitions is often irrelevant. The following example will work, even though $bar refers to $foo and $x before they are defined (in the lexical sense).

```
func $bar($a)
{
    $f = $foo($a * 2);
    return $f + $global.$x;
```

```
}

func $foo($b)
{
    $global.$x = 3;
    return $b + 1;
}
```

Another consequence is that if you load a second script that also defines a function `$foo`, this overrides the existing definition. All later invocations of `$bar` will use the new version. Invocations that are still pending keep using the old definition.

2.11. Flow Control

The following sections describe the flow control mechanisms supported by the debugger script language. The `return` statement is discussed in

2.11.1. if / elseif / else

Conditionally executing one or more statements can be done using an `if` statement, optionally with one or more `elseif` statements and at most one `else`.

Note that unlike in C/C++, the associated clauses must *always* be surrounded by curly braces.

```
if ($a > $b)
{
  if ($a > $b + 1000)
  {
    $print("much ");
  }
  $printnl("larger");
}
elseif ($a == $b)
{
  $printnl("equal");
}
else
{
  $printnl("smaller");
}

if ($a & 1)
  $printnl("odd"); // Syntax error: missing curly braces.
```

The operand of an `if` or `elseif` must be an *expression*.

2.11.2. do and while

The script language's `do` and `while` loops are the same as in C/C++. An example is shown below. Note the use of a labeled `break` statement.

```
$outer_loop: do                // label: do statement
{
  while ($x--)
  {
    $a += $arr[$x];
    if ($a == 1000)
    {
      break $outer_loop;
    }
  }
} while ($a < 100);
```

2.11.3. for

The script language's `for` loops are similar to those in C/C++. The required syntax is:

`for (` *assignment* `;` *expression* `[;` *assignment*|*expression*`] )`

where the three operands have the standard meaning.

```
for ($x = 0; $x <= 10; $x++)
{
  $a += $arr[$x];
}
```

2.11.4. foreach

You can use `foreach` to iterate over all elements of either an indexed array or an associative array, as well as all characters of a string. The required syntax is:

`foreach (` *value (identifier)* `[,` *key (identifier)*`] (` *expression* `)`

• For strings, iteration takes place from left to right.

• Elements of an indexed array are visited from lowest to highest index. Undefined elements are skipped.

• For associative arrays, the order of iteration is undefined as associative arrays are unordered.

For both types of arrays, the object assigned to *value* is the element itself, not a copy!

If the optional *key* argument is provided, `foreach` assigns the "key" to the specified variable. For an associative array, this is the element's key (a string or a number), whereas for indexed arrays and strings this is the index, a non-negative number.

```
$a[5] = "hello";
$a[7] = "there";

foreach $v, $k ($a)
{
  if ( ($a[$k] != $v) || ( ($k != 5) && ($k != 7) ) )
  {
    $printnl("Cannot happen.");
```

```
  }
  if ($a[$k] == "there")
  {
    $v = "world";  // Remember: $v is not a copy but the element itself!
  }
}

$printnl($a); // Prints [hello, world].
```

2.11.5. goto

A `goto` statement causes execution to continue at the labeled statement. This statement must be located within the current function. In addition, it is not allowed to jump into a `catch` statement.

A label is an identifier and must therefore begin with a dollar sign. Following it must be a colon and, optionally, a statement.

```
if ($a < $b)
{
  goto $label3;
}

$calculate($a);

$label3: $process($a);
```

2.11.6. continue and break

A `continue` or `break` statement can only appear in the iteration statements: `for`, `foreach`, `while` and `do`.

A `continue` statement without a label causes a jump to right after the last statement of the smallest enclosing loop. This means that the next iteration will start straight away, provided the controlling condition holds.

A `break` without a label causes execution to resume right after the end of the smallest enclosing loop.

Unlike in C or Python, a `continue` or `break` statement can have an associated label. This can be useful when you want to break out of a nested loop. The label must be attached to an enclosing iteration statement, as shown in the following example.

```
$outer_while: while ($a < 10)
{
  while ($b < 10)
  {
    $a--;
    $b--;
    if ( $a + $b == 7 )
    {
      break $outer_while;
    }
```

```
  }
}
```

2.11.7. switch

The switch statement has similar semantics to that in C. However, there are some significant differences:

- Both strings and numbers can be used for the switch argument and the case expressions. (Mixing types will not throw an exception, unless you compare a string and a number explicitly using for example "str" == 4 .)

- A case or default must be followed by a single statement or a block of statements enclosed by curly braces { }.

- With case match *regular expression*, comparison against a regular expression can take place.

- Case expressions do not need to be compile-time constants.

- There is no "fall through" behavior: once a matching case has been found, no other cases are considered.

  Because of this, the case statement block should not include a break statement (unless of course it belongs to an enclosing loop).

Examples:

```
$a = "hello";
switch ($a)
{
  case 10:             /* Does not match. No exception, even though
                        * executing $a == 10 would have thrown an exception. */
    $a++;

  case $a: // Not a constant, but valid (and true by definition)

                     // Syntax error: curly braces must be used.
    $print($a);
    $printnl(" world");

  case match "^h":  // Matches.
  {
    $a += " world";
    $printnl($a);
  }

  default:
  { }  // Not reached: there is no fall through from the preceding case.
}
```

2.12. Functions

Functions are defined using the keyword `func`, which must be followed by a valid identifier and an argument list of the form (*argument 0*, *argument 1*, …). Arguments are not declared with a specific type and the actual type at run-time may even vary from invocation to invocation. The compiler does not check type compatibility. An example is shown below.

```
$v = 3;
$a = $multiply($v, 2);      /* Will yield 6 */
$b = $multiply("hello", 2); /* Will cause exception because "hello" * 2
                             * cannot be performed */

func $multiply($x, $y)
{
  $x *= $y;
  return $x;
}
```

In the above example, `$v` is passed to `$multiply` by value, meaning that the statement `$x *= $y;` does not affect `$v`, because `$x` is initialized with a copy of `$v`.

If the argument identifier is prefixed with the `ref` keyword, it will be passed by reference, meaning that the function does not use a copy, but the actual argument object itself, as illustrated below.

```
$a = "Hello";
$append_period($a);
$printnl($a); // Prints "Hello."

func $append_period(ref $s)
{
  $s += ".";  // $s refers to the same variable as $a
}
```

Using `ref` can be useful to implement, as above, "in / out" arguments, but can also help script performance. In the following example, the function `$check_array_length()` operates on a copy of `$arr`, which may require significant resources if the array is long. Because the function does not modify the array, declaring its argument as `ref $a` gives exactly the same behavior.

```
$arr = $create_very_long_array();
$is_long_enough = $check_array_length($arr);

func $check_array_length($a)
{
  return ($length($a) > 10); // $a is a copy of $arr
}
```

If `$x` is the name of a function or a variable holding a reference to one, `$type($x)` will return `"FUNCTIONREF"`.

2.12.1. Local Variables

Local variables are not explicitly declared. Rather, the script language compiler constructs a local variable whenever an assignment is made to an identifier that does not match one of the argument identifiers. Note that it does not matter where in the function body the first assignment takes place. Also, within a function there is only one scope, thus inner blocks do not define a separate scope, as illustrated below.

```
func $foo($x, $y)
{
  $y = $defined($a); // Gives 0, but is allowed.

  $a = 0; // Defines local.
  $b = 0; // Defines local.
  $x = 0; // Modifies argument, does not define local.

  if ($x > $y)
  {
    // This is not a separate scope.
    $c = $y * 5;
  }
  if ($x > $y)
  {
    if ($c > 3) // Allowed; $c is visible within whole function.
    {
      ...
    }
  }
}
```

2.12.2. Accessing Global Variables

Normally, you can access a global variable just by its identifier. However, if the function scope already contains a variable with the same name (for example, an argument), the scope prefix $global. should be used to access the global variable.

If you want to modify a global variable from within a function body, you should always use $global., otherwise there would be no way to distinguish this from the definition of a local variable, as illustrated below.

```
// Globals.
$g = 3;
$h = 4;
$i = 5;

func $foo($g)              // $g is defined within the function scope
{
  $printnl($g);           // Prints local argument
  $printnl($global.$g);   // Prints global
  $printnl($i);           // Prints global
```

```
  $h = 5;                    // Defines new local
  $global.$h = 1;            // Modifies global
}
```

2.12.3. Return Value

Whether or not a function returns a value is not declared explicitly. In fact, you can conditionally just return from the function (`return;`) or return a specific value (`return $x[5];`). Like function arguments, the return type is not declared explicitly.

If a value is returned, it is passed by reference. Thus, in the following example, `$x[1]` is modified.

```
$x = [1, 2, 3];
($get_second_element($x))++;  // the returned element is a reference to $x[1]

func $get_second_element(ref $arr)
{
  return $arr[1];
}
```

2.12.4. Variable Argument List

Like C and C++, the script language supports functions with variable arguments specified with an ellipsis (`...`). This must always appear at the end of the argument list, but unlike in C and C++, it may be the only argument. Like for named arguments, you can use the `ref` keyword. The function itself can access the arguments via the special array `$args`, which contains the arguments in the same order as they were passed to the function.

```
$printnl($count_above_limit(3, 1, 2, 3, 4, 5)); // Prints 2.

func $count_above_limit($limit, ...) // Or $limit, ref ...
{
  $c = 0;
  for ($k = 0; $k < $length($args); $k++)
  {
    if ($args[$k] > $limit)
    {
      $c++;
    }
  }
  return $c;
}
```

Passing an array of arguments: $_args

Instead of passing a variable number of arguments individually, you can also pass the arguments in an array.

This might be useful if arguments are conditionally constructed before they are passed. For that, you must initialize the special array `$_args` and pass it to the function as its last argument. A special case

is when a variable argument function needs to pass some or all of these arguments to another function with variable arguments. In the following example, this is the function itself.

```
if ($b > 0)
{
  $_args = [$a, $b];
}
else
{
  $_args = [$a];
}

// This has the same effect as either
//    $count_odd_or_even(1, [$a, $b]);
// or
//    $count_odd_or_even(1, [$a]);

$count_odd_or_even(1, $_args);

func $count_odd_or_even($odd, ...) /* Counts number of odd/even integers in
                                    * possibly multi-dimensional array. */
{
  $c = 0;

  for ($k = 0; $k < $length($args); $k++)
  {
    if ($type($args[$k]) == "ARRAY")
    {
      $_args = $args[$k];
      $c += $count_odd_or_even($odd, $_args); // Recursion.
    }
    elseif (($args[$k] & 1) == $odd)
    {
      $c++;
    }
  }
  return $c;
}
```

2.13. Classes

Classes are the only types in the language that you can define yourself.

You can define a class with the `class` keyword followed by a valid identifier, an opening curly brace, the definitions of the data members and member functions of the class and a closing brace. An example is shown below. In the following sections several details are explained in more detail.

Note that the script language also has a number of built-in classes. See Section 2.17, *Built-in Classes*.

```
class $myclass
{
```

```
  // Constructor (always required).
  func $myclass($a, $b)
  {
    $this.$x = $a * $b;  // Defines class instance variable.
    return;
  }

  // Member function.
  func $set_y($d)
  {
    $this.$y = $d / 2;   // Defines class instance variable.
  }

  // Member function.
  func $multiply_x_by_c()
  {
    $this.$x *= $myclass.$c;

    $printnl($x); // Run-time error: no local $x, use $this.$x.
    $printnl($c); // Run-time error. no local $c, use $myclass.$c.
  }

  $c = 5;                 // Defines class variable.
}
```

Unlike C++, the script language provides neither member access restriction (`private`, etc.), nor inheritance, nor operator overloading.

2.13.1. Constructor and Other Member Functions

All member functions operate on a specific instance of the class. You can refer to the instance with the special variable `$this`. For example, `$instance_type($this)` returns a string containing the name of the class.

A class must contain at least one member function, the so-called constructor, which must have the same name as the class. Besides the constructor, a class can have zero or more other member functions. You create an instance of the class by calling the constructor:

```
$myclass_instance = $myclass(1, 2);
```

Note that although semantically the constructor returns an instance of the class, it must be defined with a simple `return;` (or no return statement at all). So, do not try to return a value.

Because the language is garbage-collected, there is no destructor function.

For other, non-constructor member functions, the syntax and semantics are the same as for functions not associated with a class.

2.13.2. Class Instance Variables

A class may have zero or more instance variables. Similar to local variables, these are not declared explicitly. Instead, the compiler constructs an instance variable whenever you assign an expression to a variable of the form $this.*identifier* in any of the class member functions. In the example in Section 2.13, *Classes*, each instance of $myclass has two instance variables, $x and $y.

Instance variables of a class instance can be accessed anywhere via *<expression yielding class instance>*. *identifier*. For example:

```
$arr[3].$x
```

if $arr is an array of instances of $myclass.

Within a member function, you must access the variables of the instance with $this. *identifier*, even when you access it within the same function as where you defined the variable. Without $this., it is assumed that you try to access a local or global variable.

2.13.3. Class Variables

The third and final kind of class member is the class variable. Assignments done within a class body, but outside any member functions define a class variable (such as $c in the example in Section 2.13, *Classes*).

There is only one copy of each class variable, which is not associated with any instance of the class. Modifying $c will affect any invocation of $multiply_x_by_c() regardless of which instance it is invoked on. You must access a class variables with *class name*. *identifier*, even when you access it within the same class as where you defined the variable.

2.14. Garbage Collection

Script language data that is no longer used is deleted (deallocated) automatically. Data is considered used when it is referred to (directly or indirectly) from a global or local variable. Local variables disappear when the function they belong to exits.

```
func $get_data()
{
  $t =ref [1, 2, 3];
  $global.$d =ref $t;
  return $t;
}

$v =ref $get_data();

// At this point, the array is referred to by both
// $v and the global $d.

$v =ref [];
$global.$d =ref [];

// Array [1, 2, 3] is unreferenced now and will be deleted.
```

2.15. Exceptions

Errors, such as attempting to read a non-existing array element or to divide by zero, cause exceptions, in a way similar to Java. In your own script, you can also throw an exception deliberately with the keyword `throw`.

An exception is represented by an instance of the predefined class `$exception`. It has the following member variables.

- `$type`: A string. If the exception was thrown by the system, this will be one of the strings listed in the table below. For your own exceptions you can define any string.

- `$description`: String describing the exception, for example `"Exception: Global variable not found, $a"`.

- `$user`: User-caused exceptions can assign any object of any type to this, for example for parametrized exceptions. For exceptions reported by the system, this will always be *nil*.

- `$stack_trace`: Stack trace (in the form of a string) indicating the context of the location where the exception was thrown.

You can create an instance of the `$exception` class by calling its constructor. It has the following form:

`func $exception(`*type, description [, user object]* `)`

Try-catch

A `try-catch` statement has the following form:

```
try
{
statement(s)
}
catch ( variable [, regular expression (string)] )
{
statement(s)
}
```

If there is a relevant `try-catch` statement in the function where the error occurred or one of its callers, the stack will be unwound to that point and the exception will be delivered there. Execution will continue at the first statement of the `catch` clause. The `$exception` object will be passed to the variable that is the first argument of `catch`.

The second, optional argument of the `catch` statement is a regular expression or string that is compared to member `$type` of the caught exception and determines whether the caught exception should be passed to this clause or not. This way it is possible to handle several types of exceptions that may occur in the try clause differently.

If no matching `catch` clause is found, an error message is shown and the script is terminated.

The next list shows the built-in exception types.

Exception type (literal string)	Cause
`"#INVALID_OPERAND"`	Type or value of operand is invalid.
`"#INVALID_INDEX"`	Occurs, for example, when trying to access element -1 of an array.
`"#DIV_BY_ZERO"`	Occurs when attempting to divide by zero using the operator `/ /=`, `%` or `%=`.
`"#OUT_OF_MEMORY"`	The host computer has too little memory available to perform the operation.
`"#FATAL_INTERNAL_ERROR"`	An internal error has occurred.
`"#FUNCTION_RETURNED_NO_VALUE"`	The calling function is attempting to access its callee's return value, but the callee did not return anything.
`"#IO_ERROR"`	See Section 2.18, *File I/O*
`"#KEY_NOT_FOUND"`	It was attempted to read a non-existent element of an associative array.
`"#USER_DEFINED"`	Can be used for custom purposes. Exceptions of this type are never thrown by the system itself.
`"#NIL_OBJECT"`	See Section 2.6, *Nil, $defined(...) and $delete(...)*.
`"#THREAD_NOT_STARTED"`	See Section 2.19, *Multithreading*.
`"#OBJ_NOT_HASHABLE"`	An object that is not a number or a string was used as an associative array key.
`"#OPERATION_NOT_ON_THIS_TYPE"`	The requested operation cannot be performed on instances of the given type.
`"#MEMBER_NOT_FOUND"`	It was attempted to access a specific member of a class, but it has no member with that name.
`"#INVALID_KEY"`	An error related to an associative array key occurred.
`"#TOO_MANY_PARAMETERS"`	The number of arguments specified in a function call is larger than the number expected by the function.
`"#TOO_FEW_PARAMETERS"`	The number of arguments specified in a function call is smaller than the number expected by the function.
`"#MODIFIYING_CONSTANT"`	Section 2.9.2, *Assignment of Literals*.
`"#EMPTY_ARRAY"`	An empty array was unused in an inappropriate situation.

Here is an example of how to catch a system generated exception:

```
func $divide($x, $y)
{
  return $x / $y;
}
```

```
func $foo()
{
  $x = 5;
  $y = 0;

  try
  {
    // This division by zero error will be caught.
    $printnl($divide($x, $y));
  }
  catch ($e, "#DIV_BY_ZERO")
  {
    $printnl("Caught exception: " + $e.$type);
  }

  // This division by zero error will not be caught,
  // assuming none of the callers of $foo() catch it.
  // The script thread will be terminated.
  $printnl($divide($x, $y));
}

$foo();
```

2.15.1. Throwing Exceptions Explicitly: throw($e)

You can use the `throw` statement in your script to explicitly throw an exception. It takes a single argument, which must be an instance of class `$exception`.

2.16. Built-in Functions

This section provides an overview of the built-in functions that are available in the debugger script language. The functions are grouped per data type for which you can use them. Debugger specific functions are described separately.

2.16.1. Functions Applicable to All Types

Function	Description
`$defined(`*expression*`)`	See Section 2.6, *Nil, $defined(...) and $delete(...)*.

`$print(`*[expression, [expression ...]]*`)`
 	
`$printnl(`*[expression, [expression ...]]*`)`
	Prints the provided expressions, with no whitespace in between. The variant `$printnl(…)` appends a newline character.

`$delete(`*variable*`)`	See Section 2.6, *Nil, $defined(...) and $delete(...)*.
`$copy(`*expression*`)`	Returns a reference to a "deep copy" of the expression, i.e. `$b =ref $copy($a)` is equivalent to `$b = $a`.

Function	Description
`$type(`*expression*`)`	Returns the type of the specified expression as a string. See Section 2.7, *Types, $type(...)*.
`$instance_type(`*expression*`)`	Returns the name of the class that *expression* is an instance of, e.g. `"$myclass"`. If it does not evaluate to a class instance, `""` is returned.

2.16.2. Functions Applicable to Numbers

Function	Description
`$chr(`*number*`)`	Returns a string containing the character associated with the specified character code. For example, `$chr(65)` returns `"A"` and `$chr(0x20ac)` returns `"€"`. Numbers outside the allowed range yield `"?"`.
`$string(`*number [,format [,precision]]*`)`	Converts a number to a string with the specified formatting. *format* must be one of the following one-character strings: • `"d"`: signed decimal • `"e"`: exponential floating-point format • `"f"`: non-exponential floating-point format • `"g"`: as `"f"` for numbers with a small exponent and as `"e"` otherwise • `"o"`: octal • `"u"`: unsigned decimal (32-bit wrap-around) • `"x"`: lowercase hexadecimal • `"X"`: uppercase hexadecimal The optional argument *precision* applies only to `"e"`, `"f"` and `"g"` and specifies the number of digits following the decimal point, the default being 6. All other formats are integral. The input number will be automatically rounded if necessary.
`$isinf(`*number*`)`	Returns 1 when the number is -INF or +INF and 0 otherwise.
`$isnan(`*number*`)`	Returns 1 when the number is -NaN or +NaN and 0 otherwise.
`$isfinite(`*number*`)`	Returns 1 if the value is finite, 0 if it is +INF, -INF or NaN.
`$modf(`*number*`)`	Extracts signed integral and fractional values from number. The results are returned in an indexed array: [*integral part*, *fraction*]. For example, `$modf(-1.7)` returns `[-1.0, -0.7]`.
`$ceil(`*number*`)`	Rounds *number* up to the nearest integer.

Function	Description
`$floor(`*number*`)`	Rounds *num*ber down to the nearest integer.
`$abs(`*number*`)`	Returns the absolute value of *number*.
`$pow10(`*number*, *exp*`)`	Returns *number* times (10 to the power *exp)*
`$frexp(`*number*`)`	Splits the number in its normalized fraction in the range [0.5, 1.0> and its power of 2 exponent. The result is an indexed array: [*mantissa*, *exponent*].
	If number is zero, *mantissa* and *exp* are zero. If *number* is NaN or INF, *mantiss* will assume *number*, *exp* will be zero.
`$ldexp(`*number*, *exp*`)`	Returns *number* times (2 to the power *exp)*
`$copysign(`*number*, *signvalue*`)`	Returns a value that has the sign of *signvalue* and the absolute value of *number*.
`$random_bigflt()`	Returns a random floating-point value, possibly +Inf, -Inf or 0, but not NaN.

2.16.3. Functions Applicable to Strings

Function	Description
`$match(`*string, pattern [, start [, all]]*`)`	Searches for a substring - from left to right - that matches the regular expression.
	The substring is returned when a match is found, otherwise the empty string is returned.
	Argument *start* indicates the start index of the substring search. With argument *all* set to 1, all matching strings will be found. In this case the return value is an array containing all matching strings.
	The compiler also interprets the '\' character in strings, so matching a backslash character requires using `"\\"`.
`$concat(`*string*, *string1)*	Concatenates *string1* to *string*. The return value is not a new object, but the same as *string*, i.e. *string* itself will be modified.
`$length(`*string*`)`	Returns the number of characters in *string*.

Function	Description
$number(*string [,strict [, radix] [, last]]*)	Converts the string to a number. If *radix* is not provided, the prefix of the string determines the radix. 0x and 0X cause hexadecimal interpretation, 0 causes octal interpretation and a decimal interpretation is used otherwise. If *radix* is provided, it must equal 8, 10 or 16. INF, INFINITY and NaN are recognized case-insensitively. More information on the allowed notation can be found in Section 2.7.1, *Numbers*. By default, characters that do not belong to the number string are filtered from the string, but with *strict* set to 1 all characters are used in the conversion. When provided, argument *last* is set to the index of the last character read by $number.
$ord(*string*)	Returns a number representing the character code corresponding to the first letter in *string*. For example, $ord("A") returns 65. $ord("") returns zero.
$pad(*string*, *newlength* [, *padding*])	Prefixes or suffixes a string *string* to reach the desired string length *newlength*. Prepending occurs when the second argument is positive, appending occurs when it is negative. If the last argument is not provided, a single space character is used. Note that the padding string can contain more than one character. When prefixing a multi-character padding string, the prefixing is done such that the last char of the prefix string is always the last char being prefixed at all. If the new length is smaller than the input string, truncation occurs.
$slice(*string*, *start* [, *length*])	Returns a slice (substring) of *string* starting at the zero-based index *start* (inclusive). If *length* is provided, the slice will have that length, otherwise it will run from *start* to the end of the string.
$slice(*string*, *range*)	As above, but here the start index (inclusive) and end index (exclusive) are specified via the two-element array *range*. For example, $slice("abcde", [2, 4]) returns "cd".
$search(*string, pattern [, last, [start]]*)	Returns the start index in the string where regular expression *pattern* is first found, or -1 if the pattern is not found. If argument *last* is provided the index of the end of the first occurrence of the pattern in the string is returned. Argument *start* indicates the start index of the search. The search is from left to right
$rsearch(*string, pattern[, last, [start]]*)	As $search(…), except that the search is performed from right to left.

Function	Description
`$replace(`*string, pattern, replacement [, all [, start [, end]]]*`)`	Returns a copy of *string* in which the first (if *all* is absent or equal to 0) or all occurrences of regular expression *pattern* has / have been replaced by the string *replacement*. The search can be restricted to a part of *string* via the optional index arguments *start* (inclusive) and *end* (exclusive).
`$rreplace(`*string, pattern, replacement [, start [, end]]*`)`	As `$replace(…)`, but the search is performed from right to left.
`$at(`*string*, *number*`)`	Returns a string of length 1 that contains the character at zero-based index *number* of the *string*. For example, `$at("abcdef", 2)` returns `"c"`.
`$set_at(`*string, index, char*`)`	Replaces the character at the specified zero-based index by *char*, which must be a string of length 1. Note that this function operates directly on *string*; it does not return a value.
`$separate(`*string, pattern include_pattern [, return_assoc]* `)`	Partitions *string* using the regular expression separator pattern, returning an indexed array of strings. If the optional argument *include_pattern* is equal to 1, the separator characters are included in the strings in the output arrays. If the optional argument *return_assoc* is equal to 1, the result is an associative array. Examples: ```\n$indxd =ref $separate("abc#def", "#");\n // Returns ["abc", "def"].\n$indxd =ref $separate("abc#def", "#", 1);\n // Returns ["abc#", "def"].\n$assoc =ref $separate("abc#def", "#", 0, 1);\n // Returns {"abc" : "abc", "def" : "def"}.\n```

`$upper(`*string*`)`
 	
`$lower(`*string*`)`
	Returns a copy of *string* in which all lowercase / uppercase characters have been substituted by the corresponding uppercase / lowercase characters.

`$strip(`*string*`)`
 	
`$lstrip(`*string*`)`
 	
`$rstrip(`*string*`)`
	`$strip(`*string*`)` returns a copy of *string* from which all leading and trailing whitespace characters have been removed.
	
	The variants `$lstrip(…)` and `$rstrip(…)` only remove leading and trailing whitespace, respectively.

`$strip1(`*string*`)`	Returns a copy of *string* in which all sequences of multiple consecutive whitespace characters have been reduced to a single whitespace character.

2.16.4. Functions Applicable to Indexed Arrays

Function	Description
`$slice(`*array, start [, length]*`)`	Returns a slice of *array* starting at the zero-based index *start* (inclusive). If *length* is provided, the slice will have that length, otherwise it will run from *start* to the end of the array.
`$slice(`*array, range*`)`	As above, but here the start index (inclusive) and end index (exclusive) are specified via the two-element array *range*. For example, `$array([51, 52, 53, 54], [1, 3])` returns `[52, 53]`.
`$insert(`*array, pos, elem*`)`	Inserts the specified element (by reference) into *array*. The return value is *array* itself. Argument *pos* may be any non-negative integer. If there already is an element at this index, that and any following elements are shifted one place.
`$append(`*array, elem*`)`	Appends the specified element (by reference) into *array*. The return value is *array* itself. Equivalent to *array*`[$length(`*array*`)] =ref` *elem*.
`$append(`*array, array1*`)`	Appends *array1* to *array*. All elements are copied by reference. The return value is *array* itself.

`$length(`*array*`)`	Returns the index of the highest ever defined element plus 1. This is not necessarily equal to the number of currently defined elements, as explained by the following example.
	
	`$a[2] = 3.5;`
	`$printnl($length($a)); // Prints 3.`
	`$a[4] = "hello";`
	`$printnl($length($a)); // Prints 5.`
	`$delete($a[4]); // Deletes element 4.`
	`$printnl($length($a)); // Still prints 5.`

`$lbound(`*array*`)`
 	
`$ubound(`*array*`)`
	Returns the lowest / highest index ever used in *array*, returning -1 if the array has always been empty. Like `$length(`*array*`)`, deleting elements does not affect these values.

`$string(`*array [, separator [, format [, precision]]]*`)`	Returns a string representation of the array. If present, the string *separator* will be used between each element. The individual elements will be printed as appropriate for their types, with *format* and *precision* applying for numbers, as described for `$string(…)`. Undefined elements will be skipped.

2.16.5. Functions Applicable to Associative Arrays

Function	Description
`$length(`*array*`)`	Returns the number of elements in *array*.

2.16.6. Debugger Specific Functions

This section describes the debugger specific functions that you can use to access and control the target.

Many of the functions listed in the following subsections have an optional argument called `options`. This should always be an associative array. For example, a valid argument for `$bp_code_add(...)` is `{"enabled" : 0, "expression" : "a->b == 0"}`.

2.16.6.1. Data Related Functions

Expression evaluation

Function	Description
`$evaluate(`*expression*[`,` *options* [`,` *error_var*]]`)`	Tries to evaluates a target expression. *expression* must be a string. Returns the value of the expression, always in the form of a string, or `""` if failed. If the argument *error_var* is present, `""` is assigned to it if evaluation succeeds, or an explanatory string otherwise.

Option	Value	Description
`stack_level`	Non-negative integer (default zero).	Specifies the stack level at which evaluation should take place, with 0 being the current function, 1 its caller, etc.

Example of accessing registers:

```
$val = $number($evaluate("#DPP0 & 0x1234"));
```

You can modify the target state by evaluating an expression that contains an assignment, for example:

```
$evaluate("#DPP0 = 0x" + $string($val, "x"));
```

2.16.6.2. Breakpoint Related Functions

Set a code breakpoint

Function	Description
`$bp_code_add(`*address*[`,` *options* [`,` *error_var*]]`)`	Tries to set a code breakpoint at the specified *address*. *address* must be an instance of class `$addr`. Returns the new breakpoint's ID (positive integer), or 0 if failed. If 0, an explanatory string is assigned to *error_var*, if present.
`$bp_code_add_src(`*source*`,` *line* [`,` *options* [`,` *error_var*]]`)`	Tries to set a code breakpoint at the specified source line. *source* must be a string. *line* must be an integer in the range [0, 2^{32} - 1] (but note that line numbers are one-based). Returns the new breakpoint's ID (positive integer), or 0 if failed. If 0, an explanatory string is assigned to *error_var*, if present.

Option	Value	Description
`enabled`	0 or 1 (default)	Defines whether or not the breakpoint should initially be enabled.
`expression`	string (default `""`)	If not equal to `""`, the breakpoint hit is reported if this expression is true (and certain other conditions are true).
`method`	`"hardware"`, `"software"` or `"any"` (default)	Defines the intended form of the breakpoint.
`skip`	integer in the range [0, 2^{64} - 1] (default 0)	Defines how many times the breakpoint hit must be ignored before it is reported.
`temporary`	0 (default) or 1	If 1, the breakpoint is deleted once it has been reported hit.

Remove, enable or disable breakpoints

Function	Description
`$bp_remove(`*id*`)` `$bp_enable(`*id*`)` `$bp_diable(`*id*`)`	Tries to remove, enable or disable a breakpoint. Enabling / disabling a breakpoint that already is enabled or disabled is allowed. *id* must be a positive integer (as returned by e.g. `$bp_code_add()`). Returns `""` if successful, otherwise an explanatory string.

Example:

```
$a =ref $addr("", 0x100);

$bp_id=$bp_code_add($a, {"enabled" : 0, "expression" : "a->b == 0"});

$err = $bp_remove($bp_id);
```

2.16.6.3. Execution Control Related Functions

Function	Description
`$continue([`*bp_var*`])`	Resumes execution.
`$halt([`*bp_var*`])`	Halts the target.
`$step_into_src([`*bp_var*`])`	Steps through a program one source line at a time. Steps into functions.
`$step_over_src([`*bp_var*`])`	Steps through a program. Functions are seen as one source line.
`$step_out_src([`*bp_var*`])`	Steps out of a function. Execution stops at the source line following a function call.
`$step_into_instr([`*bp_var*`])`	Steps through a program one assembly instruction at a time.
`$step_over_instr([`*bp_var*`])`	Steps through a program using instruction steps. Functions are seen as one instruction.
`$step_out_instr([`*bp_var*`])`	Steps out of a function using instruction steps. For example, execution stops at the instruction following a function call.

Function	Description
`$run_to_src(`*source*, *line* `[,`*bp_var*`])`	Tries to perform a source-level run to the specified source line. *source* must be a string. *line* must be an integer in the range [0, 2^{32} - 1] (but note that line numbers are one-based).
`$continue_from_src(`*source*, *line* `[,`*bp_var*`])`	Tries to perform a source-level continue from the specified source line. *source* must be a string. *line* must be an integer in the range [0, 2^{32} - 1] (but note that line numbers are one-based).
`$run_to_instr(`*address* `[,`*bp_var*`])`	Tries to perform an instruction-level run to the specified address. *address* must be an instance of class `$addr`.
`$continue_from_instr(`*address* `[,`*bp_var*`])`	Tries to perform an instruction-level continue from the specified address. *address* must be an instance of class `$addr`.

All commands above return "" if the command succeeds or a non-empty error string otherwise. If a *bp_var* argument is provided, the IDs of the breakpoints causing the target to halt are assigned to this (in the form of an array) if "" is returned.

In general it is good practice to begin your script with an invocation of `$halt()` to ensure that the target is properly halted so that the debugger can access it.

Example:

```
$a =ref $addr("", 0x100);
$bp_ids = 0;

$bp_id = $bp_code_add($a, {"enabled" : 0, "expression" : "a->b == 0"});

$step_into_src($bp_ids);
// at this point, $bp_ids equals [$bp_id]
```

2.16.6.4. Miscellaneous Debugger Functions

Function	Description
`$download(`*filename* `[,` *options* `])`	Tries to perform a download. Returns `""` if successful, otherwise a description of the failure.
	Note that there is no way for a user script to prematurely stop the downloading. Also, progress is not reported in any way.

Options of $download	Value	Description
`image_only`	0 (default) or 1	If 1, code and data sections are downloaded into the target, but no symbolic debug info is loaded.
`symbols_only`	0 (default) or 1	If 1, symbolic debug info is loaded, but nothing is downloaded into the target. Must not be 1 if `image_only` is 1.

Options of $download	Value	Description
main_arguments	Indexed array of zero or more strings	If present, these strings are passed to the target program's main() function via its arguments argc and argv. Note that this normally requires building the program in a special way; defining main(...) with said arguments is not enough.
run_to_main	0 (default) or 1	If 1, the target will run to the start of function main() after downloading.
verify	0 (default) or 1	If 1, the writing (and possibly flashing) into target memory is verified after downloading. Must not be 1 if symbols_only is 1.
map_file	string (default "")	If not equal to "", the specified file (normally with extension .mdf) will be used to configure target memory.
flash	0 (default) or 1	If 1, downloading will involve flashing if appropriate. Must not be 1 if symbols_only is 1.
flash_direct_access	0 (default) or 1	If 1, flashing will be done using direct access, instead of a monitor program. Must not be 1 if flash is not equal to 1.
reset	0 or 1 (default)	If 1, a reset will be done as part of the downloading.

Example:

```
$failed = $download("myprogram.elf", {"run_to_main" : 1});
```

Function	Description
$getargs()	Returns an array containing the strings passed via the **--arg** option. For example, after `dbg166 --arg=hello --arg=world myscript.dscr` $getargs() will return the array ["hello", "world"]. This array is a copy. Modifying it will not affect later invocations.

Function	Description
`$set_target_state_polling(` *interval* `)`	Changes the rate at which the debugger polls the target execution state, e.g. when waiting for it to hit a breakpoint. The function returns "" if successful, otherwise a description of the failure. The argument must be a non-negative integer. If it equals zero, the debugger will poll the target at its own discretion. If the argument is positive, it will poll the target at most once per *interval* microseconds. This functionality is intended for performance testing, where polling by the debugger could affect the measured cycle count or time, giving unexpected variations. Note that specifying a very large value for the argument will reduce the responsiveness of the debugger. If a test run is expected to last, say, approximately 5 seconds, a good value to call this function with would be 6000000 (6 seconds). Note also that the accuracy is subject to that of the host PC. Therefore, using arguments below about 500000 (0.5 second) is not recommended. If this function is called while the debugger is polling, i.e. when the target is running, the new polling behavior may not take effect immediately. It is therefore recommended to only call this function while the target is halted.

2.16.7. Miscellaneous Functions

Function	Description
`$get_current_thread_id()`	Returns the ID (a number) of the invoking thread.
`$getenv(` *name* `[,` *found* `] )`	Returns the value (as a string) of the specified environment variable, or an empty string if the variable does not exist. The optional argument *found* is set to 0 if the variable does not exist, 1 if is does.
`$get_millitime()`	Returns the current time in milliseconds.
`$unique_id()`	Returns a unique whole number. The first invocation returns 0, the second 1 and so on.
`$dump_stack_trace()`	Returns a string representation of the calling thread's current call stack.
`$thread_start(…)`	See Section 2.19, *Multithreading*.
`$mutex_create(…)`	
`$mutex_destroy(…)`	
`$mutex_lock(…)`	
`$mutex_trylock(…)`	
`$mutex_unlock(…)`	

2.17. Built-in Classes

The following table lists the built-in classes.

Name	Description
$addr	Represents a target address. See Section 2.17.1, *Class $addr*.
$exception	Represents an exception object. See Section 2.15, *Exceptions*.
$stream	See Section 2.18, *File I/O*.

2.17.1. Class $addr

Target addresses are represented as instances of class $addr. Member *space* is a string that, for the C166, should always equal " " and member *offset* is a non-negative integer indicating the absolute address, in bytes.

Example:

```
$a =ref $addr("", 0x100);
```

2.18. File I/O

All file I/O must be done via an instance of built-in class $stream

This class has the following class variables:

Class variable	Description
$EOF	end of file marker
$R_FILE	open mode: read file
$W_FILE	open mode: write to file
$A_FILE	open mode: append to file

The class constructor has the following form.

```
func $stream( stream name (string), open mode )
```

When you call the this constructor, the file specified by the first argument is, depending on the second argument, opened or created.

The member functions of the $stream class are listed below.

Function	Description
$getc()	Reads one character, which is returned as a string.

Function	Description
`$gets([`*number*`])`	Reads characters until a newline or the end of the file is encountered. If the argument is provided, at most that many characters are read. If a newline was encountered, the returned string will include it. Returns a string equal to `$stream.$EOF` if all characters have been read already.
`$puts([`*string*`])`	Writes the specified string to the stream.
`$flush()`	Forces writing any pending data to disk.
`$readlines()`	Returns the content of the entire file as an array, with each array element containing a single line (including newline character).
`$readfile()`	Returns the content of the entire file as a string.
`$close()`	Closes the stream.

This example reads all lines in a file and prints them unless an `"#IO_ERROR"` exception is thrown.

```
try
{
  $s =ref $stream("myfile.txt", $stream.$R_FILE);

  while (1)
  {
    $t =ref $s.$gets();
    if ($t == $stream.$EOF)
    {
      break;
    }
    $printnl("Read " + $t);
  }
  $s.$close();
}
catch ($e, "#IO_ERROR")
{
  $printnl("Error reading file.");
}
```

2.19. Multithreading

A script thread can start a new thread via the function `$thread_start(`*function* `,` *argument array*`)`, as shown in the following example.

```
func $ts($a, $b, $c)
{
  …
}
```

```
$thread_start($ts, [1, 2, 3]);
```

An exception of type `"#THREAD_NOT_STARTED"` is thrown upon failure.

Protecting data from simultaneous access by multiple threads can be done using a *mutex*. The following table lists the mutex support functions.

Function	Description
`$mutex_create()`	Creates a mutex. The return value is a *mutexhandle*, which is effectively a (unique) number.
`$mutex_destroy(`*mutexhandle*`)`	Deletes the specified mutex (*).
`$mutex_lock(`*mutexhandle*`)`	Locks the specified mutex (*). If the mutex is already locked, the calling thread is suspended until that is no longer the case. Note that a mutex is not recursive, meaning that a thread will deadlock itself by trying to lock a mutex it has locked already.
`$mutex_trylock(`*mutexhandle*`)`	As `$mutex_lock(…)` (*), but always returns immediately. The return value equals 1 if locking succeeded and 0 otherwise.
`$mutex_unlock(`*mutexhandle*`)`	Unlocks the specified mutex (*). If the mutex is not currently locked by the calling thread, an exception is thrown.

(*) If the argument does not correspond to an existing mutex, an exception is thrown.