© 2011 Check Point Software Technologies Ltd.

All rights reserved. This product and related documentation are protected by copyright and distributed under licensing restricting their use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form or by any means without prior written authorization of Check Point. While every precaution has been taken in the preparation of this book, Check Point assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

TRADEMARKS:

Refer to the Copyright page (http://www.checkpoint.com/copyright.html) for a list of our trademarks.

Refer to the Third Party copyright notices (http://www.checkpoint.com/3rd_party_copyright.html) for a list of relevant copyrights and third-party licenses.
Important Information

Latest Documentation
The latest version of this document is at:
http://supportcontent.checkpoint.com/documentation_download?ID=12381
For additional technical information, visit the Check Point Support Center
(http://supportcenter.checkpoint.com).

Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 July 2011</td>
<td>Initial version</td>
</tr>
</tbody>
</table>

Feedback
Check Point is engaged in a continuous effort to improve its documentation.
Please help us by sending your comments
(mailto:cp_techpub_feedback@checkpoint.com?subject=Feedback on CPcode for Check Point DLP R75.20 Reference Guide).
Logical Operators ........................................... 35
&& (logical AND) ........................................... 35
|| (logical OR) ........................................... 35
!(logical NOT) ........................................... 36
Bitwise Operators ........................................... 36
& (bitwise AND) ........................................... 37
| (bitwise OR) ........................................... 37
~ (bitwise NOT) ........................................... 38
^ (bitwise XOR) ........................................... 38
<< (left shift) ........................................... 39
>> (right shift) ........................................... 39
Other Operators ........................................... 39
= (assignment) ........................................... 39
inside ....................................................... 40
?: (conditional) ........................................... 41
CPcode Statements ........................................... 42
= (assignment) ........................................... 42
{} (block) ................................................. 42
break ......................................................... 43
foreach ....................................................... 43
func ........................................................ 44
if .............................................................. 45
requires ...................................................... 46
return ......................................................... 46
while ......................................................... 46
CPcode Built-in Variables ................................... 48
System Variables .......................................... 48
Feature Variables .......................................... 48
Global Variables .......................................... 49
CPcode Built-in Functions ................................... 52
align() ......................................................... 54
arrayindices() ............................................. 54
atoi() ......................................................... 55
blobbytes() ............................................... 55
byte() ......................................................... 56
csv_open() ................................................ 56
csv_close() ................................................ 57
csv_getcols() ............................................ 57
csv_getrows() ............................................ 58
csv_getval() .............................................. 58
caseprefix() ............................................... 59
casesuffix() ............................................... 59
cat() ......................................................... 60
debase64() ............................................... 60
dehex() ....................................................... 61
deftutf8() .................................................. 61
dns_expand() .............................................. 62
dns_next() .................................................. 62
elem() ......................................................... 63
ermm() ......................................................... 63
ethmac() ..................................................... 64
filename() ................................................... 64
formattedtime() .......................................... 65
hatoi() ......................................................... 66
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>host()</td>
<td>66</td>
</tr>
<tr>
<td>host6()</td>
<td>67</td>
</tr>
<tr>
<td>incksum()</td>
<td>67</td>
</tr>
<tr>
<td>index()</td>
<td>68</td>
</tr>
<tr>
<td>io_close()</td>
<td>68</td>
</tr>
<tr>
<td>io_delete()</td>
<td>69</td>
</tr>
<tr>
<td>io_open()</td>
<td>69</td>
</tr>
<tr>
<td>io_read()</td>
<td>69</td>
</tr>
<tr>
<td>io_readlines()</td>
<td>70</td>
</tr>
<tr>
<td>io_write()</td>
<td>70</td>
</tr>
<tr>
<td>kiss_compile()</td>
<td>71</td>
</tr>
<tr>
<td>kiss_pattern()</td>
<td>71</td>
</tr>
<tr>
<td>kiss_search()</td>
<td>71</td>
</tr>
<tr>
<td>le_long()</td>
<td>72</td>
</tr>
<tr>
<td>le_short()</td>
<td>72</td>
</tr>
<tr>
<td>le_ulong()</td>
<td>72</td>
</tr>
<tr>
<td>le_ushort()</td>
<td>72</td>
</tr>
<tr>
<td>listadd()</td>
<td>73</td>
</tr>
<tr>
<td>listcombine()</td>
<td>73</td>
</tr>
<tr>
<td>listglom()</td>
<td>74</td>
</tr>
<tr>
<td>listlen()</td>
<td>74</td>
</tr>
<tr>
<td>long()</td>
<td>75</td>
</tr>
<tr>
<td>lsort()</td>
<td>75</td>
</tr>
<tr>
<td>luniq()</td>
<td>76</td>
</tr>
<tr>
<td>match()</td>
<td>76</td>
</tr>
<tr>
<td>multi_compile()</td>
<td>76</td>
</tr>
<tr>
<td>multi_escape()</td>
<td>76</td>
</tr>
<tr>
<td>multi_pattern()</td>
<td>77</td>
</tr>
<tr>
<td>multi_search()</td>
<td>78</td>
</tr>
<tr>
<td>net()</td>
<td>79</td>
</tr>
<tr>
<td>net6()</td>
<td>80</td>
</tr>
<tr>
<td>pattern()</td>
<td>80</td>
</tr>
<tr>
<td>pred()</td>
<td>80</td>
</tr>
<tr>
<td>prefix()</td>
<td>81</td>
</tr>
<tr>
<td>prunearray()</td>
<td>81</td>
</tr>
<tr>
<td>regcomp()</td>
<td>82</td>
</tr>
<tr>
<td>regexec()</td>
<td>83</td>
</tr>
<tr>
<td>scope()</td>
<td>84</td>
</tr>
<tr>
<td>sed()</td>
<td>84</td>
</tr>
<tr>
<td>short()</td>
<td>85</td>
</tr>
<tr>
<td>split()</td>
<td>85</td>
</tr>
<tr>
<td>str()</td>
<td>86</td>
</tr>
<tr>
<td>strcasecmp()</td>
<td>86</td>
</tr>
<tr>
<td>strerror()</td>
<td>86</td>
</tr>
<tr>
<td>strlen()</td>
<td>87</td>
</tr>
<tr>
<td>strstr()</td>
<td>87</td>
</tr>
<tr>
<td>strtype()</td>
<td>88</td>
</tr>
<tr>
<td>sublhist()</td>
<td>88</td>
</tr>
<tr>
<td>substr()</td>
<td>89</td>
</tr>
<tr>
<td>succ()</td>
<td>90</td>
</tr>
<tr>
<td>suffix()</td>
<td>90</td>
</tr>
<tr>
<td>tolower()</td>
<td>91</td>
</tr>
<tr>
<td>toupper()</td>
<td>91</td>
</tr>
<tr>
<td>typeof()</td>
<td>92</td>
</tr>
<tr>
<td>ubyte()</td>
<td>92</td>
</tr>
<tr>
<td>ulong()</td>
<td>93</td>
</tr>
<tr>
<td>unpack()</td>
<td>93</td>
</tr>
<tr>
<td>url_flush()</td>
<td>94</td>
</tr>
<tr>
<td>url_needs_refresh()</td>
<td>95</td>
</tr>
</tbody>
</table>
Chapter 1

Developing Custom CPcode Functions

In the DLP environment, CPcode can be used for creating new custom DLP Data Types.

**Note** - Throughout this guide, “data types” has two meanings: CPcode data types are the supported data objects that can be used in scripts. For example: a CPcode data type is “integer”.

DLP data types are the representation of data sent by users, to identify messages that should be matched to the DLP policy. For example: a DLP Data Type is “Social Security Numbers”.

In This Chapter

<table>
<thead>
<tr>
<th>DLP CPcode Basics</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>The func Statement</td>
<td>8</td>
</tr>
<tr>
<td>Restriction on Executed CPcode Statements</td>
<td>9</td>
</tr>
<tr>
<td>CPcode Scope</td>
<td>9</td>
</tr>
<tr>
<td>Function Calls</td>
<td>10</td>
</tr>
<tr>
<td>Using Built-in CPcode Functions</td>
<td>11</td>
</tr>
<tr>
<td>Fastpaths</td>
<td>11</td>
</tr>
</tbody>
</table>

DLP CPcode Basics

Each file should contain one filter function: `filter run dip_dt` which does not receive any variables. This is the main function that will be called each time the Data Type is checked. This function should return 1 if the Data Type is matched, and 0 if not.

The func Statement

The **func** statement declares the name and statements that form the body of a function.

In most cases, the function declaration contains multiple statements. Curly braces ({} ) are used to make the function a block statement. Functions may take an arbitrary number of arguments (accessed using the $1, $2, $3, ... notation). Each argument may be any data type.

<table>
<thead>
<tr>
<th>func</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>func name {{statement}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name of the function</td>
</tr>
<tr>
<td>statement</td>
<td>one or more statements to be executed within the function</td>
</tr>
</tbody>
</table>
Restriction on Executed CPcode Statements

The CPcode execution engine keeps a heuristic count of approximate CPcode statements executed. At the beginning of every function call, during the prolog and at the head of all loops, the count of the number of CPcode statements executed is increased and compared against the limit.

Functions that exceed the statement limit are interrupted. The system simulates an immediate return at the point of interruption. Interrupted functions are eligible to run again when their conditions are matched.

The current limits are approximately 5,000 "ticks" from the point of execution (the run() function).

CPcode Scope

Scope refers to the availability of symbols and values within an execution context. Some languages (such as C) support three scopes: function local, file and global scope. CPcode supports multiple scopes. The variables and symbols from one scope are available to another scope by referencing the variable's containing scope. This is similar to providing a fully qualified path name in a file system, and in fact, directly applies to a hierarchical CPcode type/subtype. The scope name is determined when the sensor execution engine loads the CPcode. This name is derived from the CPcode file names.

For example, assume that:
- The protocol configuration file is dlprules.cfg
- The dlprules CPcode contains a CPcode file named commands
- The CPcode for this Protection Group is commands.cpc

The directory structure would look something like this:

```
Root dir
   +---- dlprules
         +---- commands.cpc
         +---- othercommands.cpc
```
Therefore, the scope name for variables and functions in this CPcode is dlprules_commands (which is the file name of the master CPcode, followed by an underscore, and the file name of the subfunction CPcode). The scope name for protocol-wide is simply the protocol file name. For example, the scope for the protocol-wide $WORKSPACE/protocols/dlprules.cpc file is dlprules. Multiple Protection Groups can have global variables and functions with the same name. These functions and variables will not interfere with each other because they are in different scopes. For example, both the commands and othercommands files contain a function named process_request. However, these functions exist within different scopes, dlprules_commands and dlprules_othercommands, respectively. Their fully-scoped names are dlprules_commands:process_request and dlprules_othercommands:process_request.

When you refer to global variables or functions by name, you are implicitly using variables and functions within the same scope as the reference. You can call functions and access variables in different scopes by using the fully qualified function or variable name.

You can declare values and assign them only to variables and functions in the same scope. By default, all CPcode functions and variables you define are global and exist within a specific scope. If you need local variables, you must explicitly create them.

You can use the `scope()` built-in function (see "scope()" on page 84) to obtain scope name.

### Local Variables

CPcode local variables are defined by how they are named. To create a local variable, prefix the variable name with a dollar sign ($). The initial value of a local variable is NULL.

A local variable has a local scope. It exists only inside the function where it is declared. For example, the variable `$count` could hold the number of iterations inside a function named `read_data`. Another function, named `get_dataStream`, could also use a variable named `$count`. The `$count` variables in different functions hold different values.

Local variables are stack-allocated and are not persistent between calls.

For example:

This example function declares the `display_write` function, which calls the `our_display` and `our_write` functions. The values of the parameters `$1` and `$2` are available as local variables inside the function, but they cannot be modified.

```cpcode
func display_write
{
    if ($1 == 1)
        $a = our_display($2);
    else
        $a = our_write($2);
    return $a * 2 + 1;
}
```

### Function Calls

To invoke a declared function, create a statement that includes the function name, followed by the parameters enclosed in parentheses.

For example:

```cpcode
show_service (80, "http");
```

To call a function and ignore the return value: `func_name();`

Example:
Using Built-in CPcode Functions

Built-in CPcode functions use syntax conventions.

<table>
<thead>
<tr>
<th>Built-in CPcode Type</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Data Types</td>
<td>Functions that can take multiple data types for one parameter.</td>
<td>{array, int, list} where any of the data types in the curly braces is valid</td>
</tr>
<tr>
<td>Optional Parameters</td>
<td>Functions that use optional parameters.</td>
<td>[int a, int b, string str] where the parameters in the brackets are optional</td>
</tr>
<tr>
<td>Ellipsis</td>
<td>Functions that can take an indeterminate number of parameters.</td>
<td>listadd(list li, any item, [...]) where the ellipsis (...) can arbitrarily match many arguments</td>
</tr>
<tr>
<td>Default Parameter Values</td>
<td>Functions that use default parameter values. If a specific argument is not passed to one of these functions, the default value is assumed.</td>
<td>align(int num, [int hash :=4]); where the default value appears after the := symbol (for example, 4)</td>
</tr>
</tbody>
</table>

Fastpaths

Several built-in CPcode functions have pre-optimized fastpaths they can follow if their most relevant parameter is the same variable as their return variable.

For example:

```cpcode
string = dehex(string);
```

A performance boost is gained by working on the parameter in place without allocating a temporary variable as the return variable. This gain can be significant, especially for the complex data types (such as array, list, and string). Functions that do not return variables of a complex data type would not gain anything by using a fastpath, because their return variables are cheap to allocate. Functions that have pre-optimized fastpaths have that ability noted in their description.
Chapter 2

Optimizing CPcode

The CPcode Compiler optimizes the performance of the CPcode system. You can increase CPcode performance by taking advantage of certain features of the language and avoiding others. On the other hand, CPcode may have an adverse performance impact if you are not careful.

In general: avoid using too many string operations in one Data Type script, and avoid creating long loops.

In This Chapter

- Optimizing Variables
- Optimizing Arrays
- Optimizing Function Calls

Optimizing Variables

The CPcode Compiler is optimized to move as much of the run-time type checking into a compile-time type checker. All CPcode variables are cross-referenced with each statement that might assign its value to the variable in question. After all CPcode packages are compiled into an intermediate form, several passes of type inference are performed to determine the types of all variables concretely.

Polymorphic Variables

If the type of a variable cannot be resolved by the type inference process, that variable is called a polymorph. With polymorphs, the engine is forced to use runtime type checking, which is more expensive. To avoid this, assign only one type of data to a variable.

For example, if you initially assign a string to the variable and later assign an integer to the same variable, the variable will be tagged as polymorphic. If one variable is tagged as polymorphic, variables that depend on it will likely also be tagged polymorphic.

Network Variables

Do not mix IP addresses and networks inside a single variable. This will cause the variable to become a polymorph.

The CPCode Compiler tracks IPv4 addresses, IPv4 networks, IPv6 addresses, and IPv6 networks as separate types. However, it condenses them during code generation, because the execution engine considers them to be a unified type.

Deleting Variables

Each variable is either global or local. You may delete a variable by assigning a NULL to it (such as foo = NULL). This reclaims the memory that the variable occupies. You must nullify global variables when they become irrelevant to reclaim their memory.

You do not need to explicitly nullify local variables, unless you intend to reuse or reassign them. The CPCode Compiler will automatically nullify local variables when they fall out of scope, and will handle their destruction more efficiently than could be done with CPcode statements.
Variables Or Literals

In CPcode, there is no performance penalty in using a variable over a constant literal.

In this example, both statements take the same code path, and with the exception of cache and paging, they take the same amount of time:

```
$foo = 1; $bar = $foo;
$foo = ($bar = 1);
```

Two exceptions to this rule are constant folding and conditionals.

- **Constant folding** - The CPcode Compiler will attempt to evaluate operations on integer literals at compile time. Therefore, '1 + 1' translates to '2', and in the example:
  ```
  FOO = 1;
  BAR = 1;
  BLETCH = FOO + BAR;
  ```

- **Conditionals** - The literal will be evaluated during the compilation process, so the condition will be removed and only one code block will be compiled. For example:

```
func absindex {
    if (feature.offsetindex) {
        return offsetindex($1,$2,$3);
    }
    else {
        $i = index(substr($1,$3),$2);
        if ($i == -1 || $i == NULL) {
            return -1;
        }
        else {
            return $i + $3;
        }
    }
}
```

Optimizing Arrays

These guidelines will help you write CPcode scripts that use arrays and are optimized for performance.

Array Copies

Avoid writing CPcode that could result in a duplicate array. Copying an array can consume unnecessary CPU time and memory.

To avoid duplicate arrays:

- Do not use arrays or lists as a key to an array.
- Do not use arrays or lists as a value in an array.
- Do not use a polymorphic variable to contain an array.

Index Usage

Choose a simple data type to use as an array index. The index governs the speed of array operations.

- Use a simple data type, such as an integer, for best performance.
- Using blobs of the same size gives the worst performance. Blob comparison first checks the lengths of the blobs. If they are of different lengths, the comparison will be faster than if the blobs are of identical length, which means that their buffers must be walked.
Complex Indexes

It is common practice to track the number of times a TCP or UDP port-tuple has caused an alert, while sending the alert only once. If IP addresses are not of interest, it is better to combine the two 16-bit ports into a single integer than to use the blobbytes() function to create a blob. For example:

```$key = ($src_port << 16) | $dst_port;
```

is better than:

```$key = blobbytes($src_port, $dst_port);
```

Optimizing Function Calls

If used carefully, CPcode functions will simplify the writing of quality CPcode. If used carelessly, performance will suffer or the CPcode may fail to compile.

To ensure that functions do not adversely affect performance:

- Do not write recursive CPcode functions unless necessary.
- Do not use polymorphic variables in a function call.
Character Sets

<table>
<thead>
<tr>
<th>Character Set</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII Alphanumeric</td>
<td>CPcode supports all ASCII alphanumeric characters.</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>CPcode is not a line-oriented language. It does not have line continuation characters. Where line breaks occur, they must use UNIX-style newline characters, not the Windows-style carriage return and linefeed characters.</td>
</tr>
</tbody>
</table>
| White Space            | You can use spaces and horizontal tabs around elements to increase readability and to follow formatting conventions. Comments are also processed as white space.  
Note: You must use spaces on either side of the "inside" operator. |
| Comment                | A hash mark (#) indicates the beginning of a comment. The comment includes everything between the hash mark and the end of the line.        |

Simple Operators

Simple operators are one character.

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
</tbody>
</table>
### Compound Operators

Compound operators are two or more characters.

The "inside" operator searches a list for specified values and returns a Boolean.

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
</tbody>
</table>

### Naming Variables

- Variable names can contain only ASCII alphanumeric characters and underscores.
- Variable names must begin with an alphabetic character.
• Variable names are not case sensitive.
• Variable names are limited to 128 characters in length.
• Local variables must be defined using the dollar sign character ($) as the first character of their name.
• You cannot use reserved words as variable names. However, variable names can contain reserved words. For example, if_available is a valid variable name even though if is a reserved word.

**Special Local Variables**

• $# - the number of parameters passed to a function or filter
• $n - nth parameter ($1 is the first parameter; $2 is the second parameter, and so on)

You cannot assign values to these special local variables, and they do not contain the same value across invocations.

**For example:**

```bash
# This example makes use of the $# variable
func foo
{
    sum = bar(1, 2, 3);
    # "There were three parameters to bar 1 2 3"
    # sum == 6
    sum = bar(4, 5);
    # "There were two parameters to bar 4 5"
    # sum == 9
    sum = bar();
    # "Incorrect call to bar()"
    # sum == NULL
}

func bar
{
    if ($# == 3) {
        print("There were three parameters to bar", $1, $2, $3);
        return $1 + $2 + $3;
    } else if ($# == 2) {
        print("There were two parameters to bar", $1, $2);
        return $1 + $2;
    } else {
        print("Incorrect call to bar()");
        return NULL;
    }
}
```

**NULL**

NULL is a special variable that is a member of every data type.

You can compare expressions against NULL.

**For example:**

```bash
# Is $foo an error? (Is it invalid?)
if ($foo == NULL)
```
# Is $foo not an error? (Is it valid?)
if ($foo != NULL)
 # Is this division in error
if ( ($foo = $a / $b) == NULL )

NULL is also used to clear variables and free their memory.

**For example:**

```
func clean_up {
    COUNT = NULL;
}
func count_connections {
    if (COUNT[$1] == NULL) {
        COUNT[$1] = 1;
    } else {
        if (COUNT[$1] > CONNECTION_THRESHOLD) {
            complain($1)
        }
    }
}
```

Most built-in functions will return a NULL when there is a problem with an input parameter. In some instances, the CPcode Compiler will detect the error and reject the statement.

**For example:**

```
# Most Built-in functions can return a NULL if they
don't like the parameters. The compiler will actually
# reject this statement because it knows it is an error.
if (($length = listlen(42)) == NULL)
```

If you set a variable or any array element to NULL, that variable or array element will be discarded.

**For example:**

```
# Make the variable $foo invalid
$foo = NULL;
# Make the variable in array at index '1' invalid
$array[1] = NULL;
```

NULLS are also commutative.

**For example:**

```
$a = NULL;
# $b becomes invalid (and the equivalent of NULL)
$b = $a + 1;
```

## Reserved Words

Do not use CPcode statement names or built-in function names as variable names.

In addition, you cannot use these words as variable names:

- call
- continue
- filter
Constants

The CPcode Compiler evaluates operations on integer constants at compile time. Therefore, '1 + 1' translates to '2', and this example:

```c
if (1) {
    foo();
} else {
    bar();
}
```

translates to `foo();`.

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>+ 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>Prefix 0x and: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f</td>
</tr>
<tr>
<td>IPv4host</td>
<td>Dotted quad format, 0 to 255. Example: 10.0.0.1</td>
</tr>
<tr>
<td>IPv4net</td>
<td>IPv4net constants in dotted quad (may use netmask notation) or Classless Inter-Domain Routing (CIDR) format. The CIDR element numbers are 0 to 32. Example: 10.0.0.0:255.255.255.0 (dotted quad with netmask) 10.0.0.0/24 (CIDR)</td>
</tr>
<tr>
<td>IPv6net</td>
<td>IPv6/NUM notation, where NUM is 1 to 128. Example: ::/64 ::1/128</td>
</tr>
<tr>
<td>MAC</td>
<td>MAC address, hexadecimal, 00 to ff, each octet must contain two digits. Example: 00:80:00:EA:8C:3A</td>
</tr>
</tbody>
</table>
Configuring Constants

You can create a values file to configure specified values for a constant. The values file name must match the CPcode source file name, except for the file extension.

For example:

- CPcode file - time_travel.cpc
- Values file - time_travel.values

Content of time_travel.cpc:

```c
class run dlp dt() {
    echo("value of DELOREAN_SPEED is ", DELOREAN_SPEED);
}
```

Content of time_travel.values:

```plaintext
name DELOREAN_SPEED
text Speed
desc The speed the car needs to get to,
desc in order to time travel.
mode scalar
  88
```
Note - When you upload a CPcode file to SmartDashboard you must also upload the values file.
Chapter 4

CPcode Data Types

This chapter details the types of data that CPcode supports - not to be confused with the Data Types that you are creating for DLP policies.

Parameters passed to, or returned from, CPcode functions and filters must be of a valid CPcode data type. Most built-in CPcode functions return NULL when there is a problem with an input parameter. NULL is a special variable that is a member of every data type.

In This Chapter

array
ethmac
int
ipv4host and ipv6host
ipv4net and ipv6net
list
pattern
str
tag

array

<table>
<thead>
<tr>
<th>Description</th>
<th>A one-dimensional array of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>array_name[element]</td>
</tr>
<tr>
<td>Range</td>
<td>The range for each element depends on the data type of that element. For example: \texttt{b = a[&quot;hello&quot;]};</td>
</tr>
<tr>
<td>Comments</td>
<td>• The index into the array can be any valid CPcode data type.</td>
</tr>
<tr>
<td></td>
<td>• Multiple data types can be used as an index into the same array.</td>
</tr>
<tr>
<td></td>
<td>• Each value in an array can be any valid CPcode data type.</td>
</tr>
<tr>
<td></td>
<td>• Data types within an array can be mixed.</td>
</tr>
<tr>
<td></td>
<td>To delete an element from an array, set the element to NULL.</td>
</tr>
<tr>
<td></td>
<td>To delete an array, set the array to NULL.</td>
</tr>
<tr>
<td></td>
<td>For example: \texttt{$array = NULL;}</td>
</tr>
</tbody>
</table>
# define constants to use in emulating a structure
# with an array:
USERNAME = 1;
AUTHENTICATED = 2;
STATE = 3;
LINE = 4;

# build up the structure:
...
$fake_structure[USERNAME] = $username;
$fake_structure[AUTHENTICATED] = 0;
$fake_structure[STATE] = STATE_PENDING_AUTH;
...
if ($fake_structure[STATE] < STATE_MSG_DONE) {
    # Buffer up the current message for something
    # XXX Should be careful about the length
    $fake_structure[MSG] = cat($fake_structure[MSG], "new data")
} else {
    # Try each virus signature
    foreach $virus_sig inside (VIRUS_REGEXES) {
        if (regexec($virus_sig, fake_structure[MSG])) {
            do_alert($fake_structure);
            break;
        }
    }
    # Free the current message.
    $fake_structure[MSG] = NULL;
}

### ethmac

**Description**
An Ethernet MAC address.

**Syntax**
Six octets of hexadecimal values separated by colons.

**Range**
00:00:00:00:00:00 to ff:ff:ff:ff:ff:ff

**Example**
- **Correct usage**
  
  ```
  our_mac_address = 00:60:08:31:14:b7;
  ```

  ```
  # Incorrect usage
  our_mac_address = 0:60:8:31:14:b7;
  ```

### int

**Description**
An integer.

**Syntax**
A decimal or hexadecimal representation written with a prefix of 0x.

**Range**
(-231) to (231 -1) or -2,147,483,648 to +2,147,483,647
<table>
<thead>
<tr>
<th><strong>int</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comments</strong></td>
<td></td>
</tr>
</tbody>
</table>
- Octets with a value less than 0x10 must start with a leading zero.  
- **int** is a signed data type, even when taken as a result of the **ulong** built-in function.  
- **Example**  
  `2001`  
  `0xAA` |

### ipv4host and ipv6host

#### ipv4host

| **Description** | A host IP address in IPv4 format. |
| **Syntax** | Four octets of decimal values separated by period (a dotted quad) |
| **Range** | 0.0.0.0 - 255.255.255.255 |
| **Example** | `our_router = 208.244.85.1;` |

#### ipv6host

| **Description** | A host IP address in IPv6 format.  
Refer to RFCs 2460 and 2373 for details about ipv6 address formats. |

### ipv4net and ipv6net

#### ipv4net

| **Description** | A network IP address in IPv4 or CIDR format. |
| **Syntax** | A dotted quad defining the network number, followed by a colon,  
followed by a dotted quad describing the netmask;  
**Or:**  
A dotted quad defining the network number, followed by a slash,  
followed by the number of 1 bits in the netmask (CIDR syntax). |
| **Range** | 0.0.0.0:0.0.0.0 - 255.255.255.255:255.255.255.255  
or  
0.0.0.0/0 - 255.255.255.255/32 |
| **Example** | `our_network = 208.244.85.0:255.255.255.0;`  
`our_network = 208.244.85.0/24;` |
<table>
<thead>
<tr>
<th>ipv4net</th>
<th>ipv6net</th>
</tr>
</thead>
</table>

Description
A network IP address in IPv6 format.
Refer to RFCs 2460 and 2373 for details about ipv6 address formats.

**list**

<table>
<thead>
<tr>
<th>list</th>
</tr>
</thead>
</table>

Description
An ordered list of values.

Syntax
[element,element]

Range
The range for each element depends on the data type of that element.
Data types may be mixed within a list.
While a list can contain any number of elements, it is best to keep the list short.

Comments
Spaces are not allowed.

Example
```
interesting_ports = [21, 23, 80, 443, 2001];
a = [1, 2, 3];
$b=1;
a = [$b, $b+$b, 4-$b];
```

**pattern**

<table>
<thead>
<tr>
<th>pattern</th>
</tr>
</thead>
</table>

Description
A regular expression returned by the `regcomp()` or `pattern()` functions, that can be either passed to the `regexec()` function or directly compared against a string.

**str**

<table>
<thead>
<tr>
<th>str</th>
</tr>
</thead>
</table>

Description
An arbitrary sequence of bytes.

Syntax
- Text, enclosed within single or double quotes.
- Hex numbers for each byte, separated by a backslash, enclosed within single or double quotes.
str

<table>
<thead>
<tr>
<th>Range</th>
<th>&quot;&quot; (empty string) to no explicit limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The str data type is not a C-style NULL-terminated string. It may contain binary data including any number of NULL characters.</td>
</tr>
<tr>
<td></td>
<td>The following C-style special characters can be used when creating a str data type:</td>
</tr>
<tr>
<td></td>
<td>• \n - newline</td>
</tr>
<tr>
<td></td>
<td>• \r - carriage return</td>
</tr>
<tr>
<td></td>
<td>• \t - tab</td>
</tr>
<tr>
<td></td>
<td>• \ - backslash</td>
</tr>
<tr>
<td></td>
<td>• \xnn - an arbitrary byte value for this character, where nn is any hex digit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comments</th>
<th>Only the CPcode application engine can generate a pattern data type. You cannot define it as a constant literal when developing CPcode or build it directly out of literals.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operations on large strings can be memory intensive.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>foo = &quot;string&quot;;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>foo = &quot;\x73\x74\x72\x69\x6e\x67&quot;;</td>
</tr>
</tbody>
</table>

---
tag

tag

<table>
<thead>
<tr>
<th>Description</th>
<th>Tags are descriptors for triggered filters, which you can use to turn off the filter during later execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>There is an implicit parameter to filters - $1 - which is the filter tag for the filter (&quot;Special Local Variables&quot; on page 17).</td>
</tr>
</tbody>
</table>
Chapter 5

CPcode Operators

CPcode operators determine how CPcode operates on one or more expressions within a statement.

In This Chapter

- Arithmetic Precedence  27
- Arithmetic Operators  28
- Relational Operators  30
- Logical Operators  35
- Bitwise Operators  36
- Other Operators  39

Arithmetic Precedence

<table>
<thead>
<tr>
<th>Order</th>
<th>Oper.</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ ]</td>
<td></td>
</tr>
<tr>
<td>2nd</td>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>3rd</td>
<td>inside</td>
<td>Boolean, dependent upon whether an item is inside a list</td>
</tr>
<tr>
<td>4th</td>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>5th</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>6th</td>
<td>&lt;&lt;</td>
<td>left bit shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>right bit shift</td>
</tr>
<tr>
<td>7th</td>
<td>&lt;</td>
<td>relational less than</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>relational less than or equal to</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>relational greater than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>relational greater than or equal to</td>
</tr>
<tr>
<td>8th</td>
<td>==</td>
<td>relational equal to</td>
</tr>
</tbody>
</table>
### Arithmetic Operators

In this section

<table>
<thead>
<tr>
<th>Order</th>
<th>Oper.</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>9th</td>
<td>!=</td>
<td>relational greater than or equal to</td>
</tr>
<tr>
<td>10th</td>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>10th</td>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>11th</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12th</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>13th</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14th</td>
<td>?:</td>
<td>conditional</td>
</tr>
<tr>
<td>15th</td>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>16th</td>
<td>,</td>
<td>parameter or list delimiter</td>
</tr>
</tbody>
</table>

#### + (addition)

<table>
<thead>
<tr>
<th>+</th>
<th>addition</th>
</tr>
</thead>
</table>

**Description**
The addition operator adds expression2 to expression1.

**Syntax**
```
int expression1 + int expression2
```

**Operands**
- `int expression1` - an expression
- `int expression2` - an expression

**Return Value**
`int` - the sum of the two expressions

**Example**
```
# This example adds 1 to the variable $count
$count = $count + 1;
```

#### - (subtraction)

<table>
<thead>
<tr>
<th>-</th>
<th>subtraction</th>
</tr>
</thead>
</table>

**Description**
The subtraction operator subtracts expression2 from expression1.
### Subtraction

**Syntax**
```
int expression1 - int expression2
```

**Operands**
- `int expression1` - an expression
- `int expression2` - an expression

**Return Value**
`int` - the difference between the two expressions

**Example**
```cpp
# Subtract 1 from the $time_left variable
$time_left = $time_left - 1;
```

### Multiplication

**Description**
The multiplication operator multiplies `expression1` by `expression2`.

**Syntax**
```
int expression1 * int expression2
```

**Operands**
- `int expression1` - an expression
- `int expression2` - an expression

**Return Value**
`int` - the product of the two expressions

**Example**
```cpp
# Multiply number of minutes by 60
# to determine number of seconds
seconds = minutes * 60;
```

### Division

**Description**
The division operator divides `expression1` by `expression2`.

**Syntax**
```
int expression1 / int expression2
```

**Operands**
- `int expression1` - an expression
- `int expression2` - an expression

**Return Value**
`int` - the quotient

**Example**
```cpp
# Divide number of seconds by 60
# to determine the # number of minutes
minutes = seconds / 60;
```

### Modulus

**Description**
The modulus operator divides `expression1` by `expression2` and returns the remainder.

**Syntax**
```
int expression1 % int expression2
```

**Operands**
- `int expression1` - an expression
- `int expression2` - an expression
% | modulus
---|---
**Return Value** | int - the remainder

**Example** | # Use division and modulus by 60
| # to convert seconds to minutes and seconds
| seconds = 318;
| minutes = seconds / 60;
| seconds = seconds % 60;

- (negation)

| - | negation
---|---
**Description** | The negation operator displays the negative value of an expression.

**Syntax** | - int expression

**Operands** | int expression - an expression

**Return Value** | int - the negation of the expression

**Example** | # Assign the negative of variable $b$ to variable $a$
| $a = - b;$

**Relational Operators**

In this section

== (equal to) | 31
!= (not equal to) | 32
> (greater than) | 33
>= (greater than or equal to) | 33
< (less than) | 34
<= (less than or equal to) | 34
### == (equal to)

<table>
<thead>
<tr>
<th>==</th>
<th>equal to</th>
</tr>
</thead>
</table>

**Description**  
The equal to operator compares expression2 to expression1 for equality, using different evaluation criteria for each data type.

- **array** - compares each index and the corresponding item
- **ethmac** - compares as a set of six octets. Each octet in expression1 must be equal to the matching octet in expression2
- **int** - compares by value
- **ipv4host** - compares as a set of four octets. Each octet in expression1 must be equal to the matching octet in expression2.
- **ipv4net** - compares as a set of eight octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- **ipv6host** - compares as a set of 16 octets. Each octet in expression1 must be equal to the matching octet in expression2.
- **ipv6net** - compares as a set of 32 octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- **list** - compares by each element of each list. Each element in expression1 must be equal to the matching element in expression2
- **pattern** - compares by pattern
- **str** - compares by value, with regard to case

You can compare expressions that have different data types, for example:

- **ipv4host** and **ipv4net** - if the host is in the network, the expressions are considered equal
- **str** and **pattern** - if the string is the same as the pattern, the expressions are considered equal

```expression1 == expression2```

**Syntax**

**Operands**

- **expression1** - an expression
- **expression2** - an expression

The following data types are valid for these operands:

- **array**
- **ethmac**
- **int**
- **ipv4host**
- **ipv4net**
- **ipv6host**
- **ipv6net**
- **list**
- **pattern**
- **str**

**Return Value**

- `1` - expression1 is equal to expression2
- `0` - expression1 is not equal to expression2
== | equal to
---|---
**Example** | # This example compares the subject to NULL, and if it is not NULL, then other processing occurs
if ($subject == NULL)
    $subject = $1;
else
    $subject = cat( $subject, $1);
return;

!= (not equal to)

| != | not equal to |
---|---|
**Description** | The not equal operator compares expression2 to expression1 for inequality, using different evaluation criteria for each data type.
- array - compares each index and the corresponding item
- ethmac - compares as a set of six octets. Each octet in expression1 must be equal to the matching octet in expression2
- int - compares by value
- ipv4host - compares as a set of four octets. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv4net - compares as a set of eight octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv6host - compares as a set of 16 octets. Each octet in expression1 must be equal to the matching octet in expression2.
- ipv6net - compares as a set of 32 octets or a set of four octets and the netmask. Each octet in expression1 must be equal to the matching octet in expression2.
- list - compares by each element of each list. Each element in expression1 must be equal to the matching element in expression2
- pattern - compares by pattern
- str - compares by value, with regard to case
You can compare expressions that have different data types, for example:
- ipv4host and ipv4net - if the host is not in the network, the expressions are not equal.
- str and pattern - if the string is not the same as the pattern, the expressions are not equal.
**Syntax** | expression1 != expression2
### != (not equal to)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

**Operands**
- expression1 - an expression
- expression2 - an expression

The following data types are valid for these expressions:
- array
- ethmac
- int
- ipv4host
- ipv4net
- ipv6host
- ipv4net
- list
- pattern
- str

**Return Value**
- 1 - expression1 is not equal to expression2
- 0 - expression1 is equal to expression2

**Example**
```bash
# Compare value in $s_nevermind to 1
if ($s_nevermind != 1)
    webdet_server_process($v);
```

### > (greater than)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
</tbody>
</table>

**Syntax**
```bash
int expression1 > int expression2
```

**Operands**
- int expression1 - an integer expression
- int expression2 - an integer expression

**Return Value**
- 0 - expression1 is not greater than expression2
- 1 - expression1 is greater than expression2

**Example**
```bash
# Compare string length of variable $blob to 2048
if (strlen($blob) > 2048)
    $blob = "";
```

### >= (greater than or equal to)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt;</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

**Syntax**
```bash
int expression1 => int expression2
```

**Description**
The greater than or equal to operator compares expression2 to expression1 and determines whether expression1 is greater than or equal to expression2.
### Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
<th>Operands</th>
<th>Return Value</th>
<th>Example</th>
</tr>
</thead>
</table>
| `>=`     | **greater than or equal to** | int expression1 - an integer expression  
int expression2 - an integer expression | 0 - expression1 is not greater than or equal to expression2  
1 - expression1 is greater than or equal to expression2 | # Compare return value of the byte function to 0  
if (byte ($sblob, bloblimit) >= 0)  
$state = -1; ""; | |
| `<`      | **less than** | int expression1 < int expression2 | 0 - expression1 is not less than expression2  
1 - expression1 is less than expression2 | # Compare value in $state variable to 0  
if ($state < 0) return; | |
| `<=`     | **less than or equal to** | int expression1 - an integer expression  
int expression2 - an integer expression | 0 - expression1 is not less than or equal to expression2  
1 - expression1 is less than or equal to expression2 | # Compare the string length of the variable $cmd to 0  
if (strlen ($cmd) <= 0) | |
Logical Operators

In this section

&& (logical AND) 35
|| (logical OR) 35
! (logical NOT) 36

&& (logical AND)

<table>
<thead>
<tr>
<th>&amp;&amp;</th>
<th>logical AND</th>
</tr>
</thead>
</table>
| Description | The logical AND operator evaluates both sides of the expression. If both expressions evaluate to true (they are non-zero and non-NULL), the result is true. If either, or both, expression evaluates to false, the result is false.

The CPcode logical AND operator, like the C language && operator, is a short-circuit operator. If the first expression evaluates to false, the second expression is not evaluated.

Syntax
expression1 && expression2

Operands
- expression1 - an expression
- expression2 - an expression

Return Value
- 0 - the expression is false
- 1 - the expression is true

Example
if (ALERT_ON_ROOT && index($data, "root") >= 0) {
    alert(...);
}

# If ALERT_ON_ROOT is set to non-zero, index() will be called.
# If ALERT_ON_ROOT is set to zero, the value of the conditional will be false and index() will not be called.
# Compiler generates code similar to:
# if (ALERT_ON_ROOT) {
#     if (index($data, "root") >= 0) {
#         alert(...);
#     }
# }

|| (logical OR)

| || | logical OR |
|----|-------------|
| Description | The logical OR operator evaluates both sides of the expression. If either or both expressions evaluates to true (non-zero and non-NULL), the result is true.

This operator is a short circuit operator, like the || operator in the C language. If the first expression evaluates as true, the second expression is not evaluated.

Syntax
expression1 || expression2
|| logical OR

**Operands**
- expression1 - an expression
- expression2 - an expression

**Return Value**
- 0 - the expression is false
- 1 - the expression is true

**Example**
```
# because of short circuiting, strlen() will not be called in this
# example
$string = "foo";
if ($string || strlen($strlen) < 20)
```

---

! (logical NOT)

| ! | logical NOT |

**Description**
The logical NOT operator computes the logical negation of the expression. True indicates that the expression is zero or NULL.
- If the expression is true, the result is false.
- If the expression is false, the result is true.

**Syntax**
```
! expression
```

**Operands**
- expression - an expression

**Return Value**
- 0 - the expression is not zero
- 1 - the expression is zero

**Example**
```
$string = "foo";
if (! $string || strlen(string) < 20)
```

---

**Bitwise Operators**

In this section
- & (bitwise AND) 37
- | (bitwise OR) 37
- ~ (bitwise NOT) 38
- ^ (bitwise XOR) 38
- << (left shift) 39
- >> (right shift) 39
& (bitwise AND)

<table>
<thead>
<tr>
<th>&amp;</th>
<th>bitwise AND</th>
</tr>
</thead>
</table>

**Description**
The bitwise AND operator evaluates identically positioned bits in both expressions. If both bits are 1, the operator sets the corresponding bit in the result to 1. If either bit is 0, the operator sets the corresponding bit in the result to 0. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression1 is</th>
<th>and bit in expression2 is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Syntax**
int expression1 & int expression2

**Operands**
- int expression1 - an expression
- int expression2 - an expression

**Return Value**
int - the bitwise evaluation of both expressions

**Example**
```
# Bitwise AND on the value in the variable foo
# and the number 1
if ((foo & 1) == 1)
    echo ("foo is odd\n");
```

<table>
<thead>
<tr>
<th></th>
<th>(bitwise OR)</th>
</tr>
</thead>
</table>

**Description**
The bitwise OR operator evaluates identically positioned bits in both expressions. If both bits are 0, the operator sets the corresponding bit in the result to 0. If either bit is 1, the operator sets the corresponding bit in the result to 1.

**Syntax**
int expression1 | int expression2

**Operands**
- int expression1 - an expression
- int expression2 - an expression

**Return Value**
int - the bitwise evaluation of the two expressions

**Example**
```
# This example performs a bitwise OR on the values in the
# variables a and b, and displays the result of 7
a = 5;
b = 3;
c = a | b;
echo ("c is ", c, "\n");
```
~ (bitwise NOT)

~ (bitwise NOT)

Description
The bitwise NOT operator inverts the bit values in an expression. If the bit is 0, the operator sets the corresponding bit in the result to 1. If the bit is 1, the operator sets the corresponding bit in the result to 0. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Syntax
~ int expression

Operands
int expression - an expression

Return Value
int - the bitwise evaluation of the expression

Example
# Do a bitwise NOT on the variable a where a = ~ a;
# this example clears the least significant bit in the variable b
b = b & ~ 1;

^ (bitwise XOR)

^ (bitwise XOR)

Description
The bitwise XOR operator evaluates identically positioned bits in both expressions. If both bits are the same, the operator sets the corresponding bit in the result to 0. If the bits are different, the operator sets the corresponding bit in the result to 1. The following table summarizes this behavior:

<table>
<thead>
<tr>
<th>If bit in expression1 is</th>
<th>and bit in expression2 is</th>
<th>then bit in result will be</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Syntax
int expression1 ^ int expression2

Operands
- int expression1 - an expression
- int expression2 - an expression

Return Value
int - the bitwise evaluation of the two expressions

Example
# Do a bitwise XOR on the low 4 bits of the value in
# the variable $a
$a = $a ^ 0xf;
<< (left shift)

**Description**
The left shift operator shifts all of the bits in expression1 to the left by expression2 bits.

**Syntax**
```
int expression1 << int expression2
```

**Operands**
- `int expression1` - the integer to be shifted
- `int expression2` - an integer representing the number of bits by which expression1 should be shifted

**Return Value**
`int` - the result of the bitwise shift

**Example**
```
# This example determines whether the Nth bit is set
$tmp = 1 << Nth;
if ($1 & $tmp)
    echo ("Nth bit set
n");
```

>> (right shift)

**Description**
The right shift operator shifts all of the bits in expression1 to the right by expression2 bits. This is a logical shift, (for example, 0x80000000 >> 1 == 0x40000000).

**Syntax**
```
int expression1 >> int expression2
```

**Operands**
- `int expression1` - the integer to be shifted
- `int expression2` - an integer representing the number of bits by which expression1 should be shifted

**Return Value**
`int` - the result of the bitwise shift

**Example**
```
# Find the version of the IP packet payload contained in the
# variable $1
$version = ubyte($1,0) >> 4;
```

Other Operators

In this section

= (assignment)

**Description**
The assignment operator sets variable equal to expression. You can use assignments within expressions (often referred to as built-in assignments).

**Syntax**
```
variable = expression
```
### Assignment

<table>
<thead>
<tr>
<th>=</th>
<th>assignment</th>
</tr>
</thead>
</table>

**Operands**
- variable - a global variable, local variable, or array element
- expression - an expression. All data types are valid.

**Return Value**
variable - the value assigned by the expression

**Note** - Built-in assignments must be enclosed inside parentheses, for example:

```sh
# this is correct:
$foo = ($bar = NULL);
```

**Example**

```sh
# Use the assignment operator to set variables to
# different data types
$services[ "telnet" ] = 1;
router_mac_address = 00:c0:7b:6b:ee:2b;
$counter = 318830;
router_IP_address = 208.244.85.1;
our_network = 208.244.85.0:255.255.255.0;
our_network = 208.244.85.0/24;
web_ports = [ 80, 8000, 8080, 2001 ];
company_name = "Check Point";

# The value of the Built-in assignment expression will
# be the left hand side of the assignment after it is assigned.
$foo = 1;
$bar = ($foo = $foo + 1);
# The value of $bar is 2
# The value of $foo is also obviously 2
```

### Inside

<table>
<thead>
<tr>
<th>inside</th>
<th>special operator</th>
</tr>
</thead>
</table>

**Description**
The inside operator searches for expression1 in the list represented by expression2. The result is true if expression1 is an element of the list represented by expression2.

**Syntax**

```sh
expression1 inside expression2
```

There must be a space on either side of the inside operator.

Do not confuse the inside operator, whose role is to return a Boolean value, with "inside" used in the syntax of a foreach or declare statement.

**Operands**
- expression1 - an expression. Valid data types are any that can be added to a list.
- expression2 - a list

**Return Value**
- 0 - expression1 is not found inside expression2
- 1 - expression1 is found inside expression2
### Example

# Assume $path contains extracted path element
# from a data source to test against.
# This example looks at $path
# and tests it against elements in a list variable,
# and if a match is found,
# sets a local variable to 1.

SPOOLER_WINDOWS_ROOT_PATHS=['c:\winnt','c:\windows'];
$bad = 0;

foreach $path_var inside (SPOOLER_WINDOWS_ROOT_PATHS) {
    if (cat($path_var,'\system32') == $path) {
        $bad = 1;
        break;
    }
}

?: (conditional)

#### Description
The conditional operator evaluates an expression for a condition. It functions as an IF-THEN-ELSE shortcut.

#### Syntax
variable = expression ? value1 : value2

#### Operands
- expression - the expression to be evaluated
- value1 - the expression that will be assigned to ret_value if expression is true
- value2 - the expression that will be assigned to ret_value if expression is false

#### Return Value
variable - the value returned from the evaluation, which is value1 if expression is true or value2 if expression is false

Note - CPcode does not support the use of the conditional operator on the left-hand side of an expression. For example: ((bar > 50) ? bar : $foo) = $counter + 1; is not allowed.

#### Example
# A counter that will forgive you for starting at NULL
# 'scounter = $counter + 1;'
# will not work unless $counter initialized to be
# something other than NULL at some point
$scounter = $counter ? $counter + 1 : 1;
# To limit $foo to numbers less than or equal to 100
$foo = $bar < 100 ? $bar : 100;
Chapter 6

CPcode Statements

This section lists valid CPcode statements.

If you are familiar with the C language, you should exercise caution when writing CPcode. While CPcode statements appear to be C-like, there are differences. CPcode may not support some C language statements.

In This Chapter

= (assignment) 42
{} (block) 42
break 43
foreach 43
func 44
if 45
requires 46
return 46
while 46

= (assignment)

=  

| assignment |

Description  An assignment statement assigns the value on the right-hand side of the equal sign (=) to the variable on the left-hand side. You can use the assignment statement to set the value of a variable.

Syntax  

variable = expression;

Parameters  

- variable - the variable to which the expression will be assigned
- expression - the expression to be assigned to the variable

Example  

# Use the assignment statement to set the value of our_router to 192.168.10.1.
our_router = 192.168.10.1;

{} (block)

{}  

| block |

Description  Block notation groups together multiple CPcode statements. The block starts with the left curly bracket and ends with the right curly bracket. All statements inside these brackets are part of the block.

Syntax  

{statement_list}
### block

**Parameters**  
statement_list - a list of CPcode statements

**Example**  
# The block statement groups together several statements into a user-defined function  
func counter  
{}  
  return ($count = $count + 1);

### break

**Description**  
The break statement ends execution of the innermost foreach or while statement in which it appears. You can use the break statement to exit from a loop.

**Syntax**  
break;

**Parameters**  
N/A

**Example**  
# Go through the buffer line-by-line.  
# If check_line finds what it is looking for, # it will return 1, and we should exit  
while (($eol = index($buf, "\n")) > -1) {  
  $line = substr($buf, 0, $eol);  
  $buf = substr($buf, $eol + 1);  
  if (check_line($line)) {  
    break;
  }
  }

### foreach

**Description**  
The foreach statement executes the same statements for each element of a list or indices of an array.

When used with a list, the foreach statement sets the variable to the elements of a list and executes the statements once for each element. The sensor execution engine processes the elements in the order in which they appear in the list.

When used with an array, the foreach statement sets a variable to the index value of the array and executes the statements once for each index value. The order in which the index values appear is not defined.

**Syntax**  
foreach $variable inside [list|array] { statements }
### foreach

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$variable</td>
<td>- a local variable</td>
</tr>
<tr>
<td>list</td>
<td>- a variable, constant, or expression that evaluates to a list</td>
</tr>
<tr>
<td>array</td>
<td>- a variable, constant, or expression that evaluates to an array</td>
</tr>
<tr>
<td>statements</td>
<td>- statements to be executed for each element in the list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not modify a list or array inside the loop. While the CPcode Compiler normally prevents this, it could cause stability issues with the gateway or appliance. For example, do not code:</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
define $foo
foreach $foo inside array {
    if (array[$foo] > 1)
        array[$foo] = NULL;
}
define $foo
foreach $foo inside list {
    list = listadd(list, $foo * 2);
}
# Do a set of actions for each recipient listed in a mail message
foreach $tmp inside ($to_list)
{
    if (mail_list:present)
        mail_list:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject );
    if (mail_od:present)
        mail_od:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject );
    if (mail_tricks:present)
        mail_tricks:rec($start_time, $src_addr, $fromaddr, $dst_addr, $tmp, $subject );
}
# Do a set of actions for each element in an array.
$services["telnet"] = 1;
$services["http"] = 2;
$services["ssl"] = 3;
define $tmp
foreach $tmp inside ($services){
    echo( $tmp, " is element ", $services[$tmp], \\
        "\n" );
}
```

This function prints:

telnet is element 1
http is element 2
ssl is element 3

### func

<table>
<thead>
<tr>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The func statement declares the name and statements that form the body of a function. In most cases, the function declaration contains multiple statements.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>func name {{[statements]}}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>- the name of the function</td>
</tr>
<tr>
<td>statement</td>
<td>- one or more statements to be executed within the function</td>
</tr>
</tbody>
</table>
### if

#### Description
The if statement evaluates an expression, and if the expression evaluates to true, executes a statement. If the expression evaluates to false, the statement is not executed. If the value of the expression evaluates to false and there is an else clause, the else clause statements are executed.

#### Syntax
- `if (expression) statement`
- `if (expression) statement else statement`

#### Parameters
- `expression` - the expression to be evaluated
- `statement` - the statement to be executed if the expression is true

Parentheses are required around the expression parameter.

If the expression being evaluated is complex (for example, contains `&&` or `||`), the evaluation will terminate at the earliest opportunity.

#### Notes
In the following example, the if statement is used to determine if there is any data in the `$blob` variable. If the `$blob` variable does not contain any data, the statement assigns data to it. If the `$blob` variable does contain data, the statement after the else clause adds to the data already in the variable.

```python
if ($blob == NULL)
    $blob = $my_data;
else
    $blob = cat ($blob, $my_data);
```

#### Example
# some_function will never be called because its results will not affect the value of the expression.

```python
$false = 0;
$true = 1;
if ($false && some_function() != NULL {
}
if ($true || some_function() != NULL {
}
```
### requires

<table>
<thead>
<tr>
<th>Description</th>
<th>The &quot;requires&quot; statement specifies variables that must be set. If any variables are NULL, the rest of the CPcode is not loaded. You can use the &quot;requires&quot; statement to control whether a protocol or Protection Group is run. If a required variable is not set, the Protection Group or protocol is not loaded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>requires variable1 variable2 variable3;</td>
</tr>
<tr>
<td>Parameters</td>
<td>variable - one or more variables that must contain non-NULL values before the remaining CPcode is loaded.</td>
</tr>
<tr>
<td>Example</td>
<td># The requires statement specifies that the value for ourdomainname must be set for the protocol to load.</td>
</tr>
</tbody>
</table>

```plaintext
requires ourdomainname;
```

### return

<table>
<thead>
<tr>
<th>Description</th>
<th>The return statement exits a function and, optionally, returns a value.</th>
</tr>
</thead>
</table>
| Syntax | return;
return [expression]; |
| Parameters | expression - any expression |
| Example | # The return statement exits the if statement and returns the value in the $line variable. |

```
$xxx = index($line, ">");
if ($xxx < 0)
    return $line;
```

### while

<table>
<thead>
<tr>
<th>Description</th>
<th>The while statement evaluates an expression and repeatedly executes the statement as long as the expression remains true, as long as it is non-zero and non-NULL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>while (expression) statement;</td>
</tr>
</tbody>
</table>
| Parameters | • expression - expression to be evaluated
• statement - the statement to be executed |
### while

<table>
<thead>
<tr>
<th>While</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The while statement walks along the contents of $cblob. It starts at whatever value is in $n when the while statement is entered. As long as the $nth byte of $cblob is greater than zero, it keeps going. When it sees a byte that is less than or equal to zero, it stops.</td>
<td></td>
</tr>
</tbody>
</table>
| while (byte($cblob,$n) > 0) {  
  $n = $n + 1;
} |
Chapter 7

CPcode Built-in Variables

This section lists the packet variables of interest to CPcode developers. Built-in variables gather data from the packet just received and provide easy access to information about the connection stream.

In This Chapter

- System Variables
- Feature Variables
- Global Variables

System Variables

System variables extract data from the system running a Check Point appliance or gateway.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>system.maxticks</td>
<td>Returns the maximum number of ticks that can be used.</td>
<td>int</td>
</tr>
<tr>
<td>system.ticks</td>
<td>Returns the current number of ticks that the CPcode has used out of its allotted amount.</td>
<td>int</td>
</tr>
<tr>
<td>system.time</td>
<td>Returns the system time, in seconds, since the beginning of the epoch (Jan 1 1970 UTC).</td>
<td>int</td>
</tr>
</tbody>
</table>

Feature Variables

Feature variables provide access to supported execution engine features and revision information (such as when a built-in CPcode function was changed).

Feature variables are evaluated at compile time. For example:

```c
if (feature.filename >= 20030412) {
    #Supports new Microsoft OS's filename scheme
    $file = filename($line);
} else {
    #Do it in CPcode
    file = filename_ms($line);
}
```

translates at compile to either:

```c
$file = filename($line);
```

or
Global Variables

DLP functions in CPcode can use any of the CPcode functions and variables that are not traffic related (notice that UDP and TCP variables are not supported).

In addition, there are DLP specific variables.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy.internal_domains</td>
<td>List of domains (strings), that come from &quot;My Organization&quot; object in the policy.</td>
<td>[&quot;domain1.com&quot;,&quot;domain2.com&quot;]</td>
</tr>
<tr>
<td>message.protocol</td>
<td>IP protocol.</td>
<td>&quot;SMTP&quot; &quot;FTP&quot; &quot;HTTP&quot;</td>
</tr>
<tr>
<td>message.source</td>
<td>The source of the message.</td>
<td>&quot;<a href="mailto:dan@domain.com">dan@domain.com</a>&quot; &quot;190.1.1.1&quot;</td>
</tr>
<tr>
<td>message.source_username</td>
<td>The username of the message sender or an empty string if the sender’s email is not mapped to a username.</td>
<td>&quot;dan&quot;</td>
</tr>
<tr>
<td>message.dests</td>
<td>List of destinations.</td>
<td>In SMTP: [&quot;<a href="mailto:sam@domain.com">sam@domain.com</a>&quot;,&quot;<a href="mailto:joe@domain.com">joe@domain.com</a>&quot;] In FTP: [&quot;154.1.0.0&quot;, &quot;154.1.1.0&quot;]</td>
</tr>
<tr>
<td>message.dests_username</td>
<td>The username of the message's destination or an empty string if the recipient's email is not mapped to a username.</td>
<td>[&quot;sam&quot;, &quot;joe&quot;]</td>
</tr>
<tr>
<td>message.mail_subject</td>
<td>For SMTP - contains the mail subject (in the recipient's part). For HTTP/FTP - empty.</td>
<td>[&quot;RE: New List&quot;]</td>
</tr>
<tr>
<td>message.header_names</td>
<td>Contains protocol header fields.</td>
<td>MIME headers</td>
</tr>
<tr>
<td>message.header_values</td>
<td>Contains protocol header values.</td>
<td>MIME headers</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>part.content</td>
<td>A buffer containing the text of a part. It is either the text of a mail's body, or text of a file (attached to mail, uploaded in http or ftp).</td>
<td>&quot;This is the body.... .... Regards &quot;</td>
</tr>
<tr>
<td>part.type</td>
<td>Describes the type of the part.</td>
<td>&quot;Recipients&quot; &quot;Content&quot;</td>
</tr>
<tr>
<td>part.name</td>
<td>The name of the attachment, or the name of the file uploaded.</td>
<td>&quot;body&quot; &quot;file.doc&quot;</td>
</tr>
<tr>
<td>part.is_file</td>
<td>This variable equals 1 when the scanned part is an attachment, otherwise it equals 0.</td>
<td>Relevant only for SMTP.</td>
</tr>
<tr>
<td>verify.str</td>
<td>A string that was matched to a pattern ( The pattern was defined in a data type).</td>
<td>&quot;0404787875&quot;</td>
</tr>
<tr>
<td>verify.index</td>
<td>An index in CONTENT variable to the end of the matched string.</td>
<td>50</td>
</tr>
</tbody>
</table>

**Example**

This script shows a CPcode Data Type that is turned on when there are more than 2 external domains in an Email message.

```c
filter run dip_dt() {
    # This data type is active for SMTP only, for the recipient part
    if( message.protocol != "SMTP" || part.type != "Recipients" )
        return 0;
    $num_of_external_domains = 0;
    $external_domain_list = [];
    foreach $recipient inside message.dests {
        # get the domain of the mail address
        $domain = global:get_domain_from_address( $recipient );
        # check if the domain is external
        if( !global:is_internal_domain($domain) ) {
            if( !global:is_member( $external_domain_list , $domain ) ) {
                # Add it to the external domains list
                $external_domain_list = listadd( $external_domain_list , $domain );
                # Check if we reached the max of external domains
                $num_of_external_domains = $num_of_external_domains + 1;
                if( $num_of_external_domains >= MAX_EXTERNAL_DOMAINS )
                    return 1;
            }
        }
    }
    return 0;
}
```
Recommendations:

CPcode Data Types, just like any other Data Type, run on every part of a message. If an email is scanned, the CPcode function will be called several times: once for the Recipients part, then for each attachment, and the body of the email.

Include the following code at the beginning of your filter. If you want the CPcode Data Type to run only once for each email, allow it to run only in the "recipients part":

```cpp
if( message.protocol != "SMTP" || part.type != "Recipients" )
    return 0;
```
Chapter 8

CPcode Built-in Functions

In This Chapter

align() 54
arrayindices() 54
atoi() 55
blobbytes() 55
byte() 56
csv_open() 56
csv_close() 57
csv_getcols() 57
csv_getrows() 58
csv_getval() 58
caseprefix() 59
casesuffix() 59
cat() 60
debase64() 60
dehex() 61
deutf8() 61
dns_expand() 62
dns_next() 62
elem() 63
errno() 63
ethmac() 64
filename() 64
formattedtime() 65
hatoi() 66
host() 66
host6() 67
incksum() 67
index() 68
io_close() 68
io_delete() 69
io_open() 69
io_read() 69
io_readlines() 70
io_write() 70
kiss_compile() 71
kiss_pattern() 71
kiss_search() 71
le_long() 72
le_short() 72
le_ulong() 72
le_ushort() 73
listadd()  73
listcombine()  74
listglom()  74
listlen()  74
long()  75
lsort()  75
luniq()  76
match()  76
multi_compile()  76
multi_escape()  76
multi_pattern()  77
multi_search()  78
net()  79
net6()  80
pattern()  80
pred()  80
prefix()  81
prunearray()  81
regcomp()  82
regexec()  83
scope()  84
sed()  84
short()  85
split()  85
str()  86
strcasecmp()  86
strlen()  87
strerror()  87
strstr()  88
strtype()  88
sublist()  89
substr()  90
succ()  90
suffix()  91
tolower()  91
toupper()  92
typeof()  92
ubyte()  93
ulong()  93
unpack()  94
url_flush()  95
url_needs_refresh()  95
url_open()  95
url_readall()  96
url_readlines()  96
ushort()  97
**align()**

The `align()` function bit aligns data items that are larger than one byte (for example, a record where a four-byte integer only occurs at an address that is a multiple of four bytes).

<table>
<thead>
<tr>
<th>align()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>align(int num, [int hash :=4]);</code></td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td></td>
<td><code>int num</code></td>
</tr>
<tr>
<td></td>
<td><code>int hash</code></td>
</tr>
<tr>
<td>Return Value</td>
<td>int - The aligned integer.</td>
</tr>
<tr>
<td>Example</td>
<td><code>align(0) == 0</code></td>
</tr>
<tr>
<td></td>
<td><code>align(1) == 4</code></td>
</tr>
<tr>
<td></td>
<td><code>align(2) == 4</code></td>
</tr>
<tr>
<td></td>
<td><code>align(3) == 4</code></td>
</tr>
<tr>
<td></td>
<td><code>align(4) == 4</code></td>
</tr>
<tr>
<td></td>
<td><code>align(5) == 8</code></td>
</tr>
<tr>
<td></td>
<td><code>align(0,2) == 0</code></td>
</tr>
<tr>
<td></td>
<td><code>align(1,2) == 2</code></td>
</tr>
<tr>
<td></td>
<td><code>align(2,2) == 2</code></td>
</tr>
<tr>
<td></td>
<td><code>align(3,2) == 4</code></td>
</tr>
<tr>
<td></td>
<td><code>align(4,2) == 4</code></td>
</tr>
</tbody>
</table>

**arrayindices()**

The `arrayindices()` function creates a sorted list of an array's associative indices.

<table>
<thead>
<tr>
<th>arrayindices()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>arrayindices(array arr);</code></td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td></td>
<td><code>array arr</code></td>
</tr>
<tr>
<td>Return Value</td>
<td>- list - a list of all the array's indices</td>
</tr>
<tr>
<td></td>
<td>It is illegal to modify an array being iterated through using a foreach statement. Therefore, it is necessary to first create a list of all the array's indices, and then use foreach to iterate through that list.</td>
</tr>
</tbody>
</table>
arrayindices()

Example

array[0] = "zero";
array[1] = "one";
foreach $iterator inside (arrayindices(array)) {
    echo("array[", $iterator, "] = ", array[$iterator], "\n");
    # Output:
    # array[0] = zero
    # array[1] = one
    # The next statement would be illegal if iterating
    # over the array instead of a copy of its indices.
    array[$iterator] = NULL;
}

atoi()

The atoi() function converts a string to an integer representation. If you specify an index, the conversion
begins at that location inside the string.

The atoi() function ignores any white space leading up to the number.

Syntax
atoi(string str, [int index := 0]);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str</td>
<td>The string that is to be converted to an integer.</td>
</tr>
<tr>
<td>int index</td>
<td>The point in the string at which to begin the conversion.</td>
</tr>
</tbody>
</table>

Return Value

- int - the integer value of the string, str
- 0 - the string is empty, does not contain digits, or contains non-digits before the first digit in the string

Example

atoi("192") == 192
atoi(" 192 ") == 192
atoi(" 1 9 2 ", 4) == 2
atoi("192ABC") == 192
atoi("ABC") == 0
atoi("ABC192", 3) == 192
atoi(""") == 0
atoi(42) == error

blobbytes()

The blobbytes() function creates a blob from the byte values of an expression. Integers are an exception -
their lowest significant bytes are used. For example, if the integer is 0x23, only one byte is used; if the
integer is 0x1234, two bytes are used; etc.

The cat() function should, preferably, be used when concatenating strings.
The blobbytes() function has been optimized to generate a blob from non-strings, and is usually used to generate a hash index for an array.

### blobbytes()

**Syntax**

blobbytes(...);

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>The values to be added to the blob/string.</td>
</tr>
</tbody>
</table>

**Return Value**

- **string** - a blob/string
  - The overall return string for the blobbytes() function cannot be larger than 8KB.

**Example**

```plaintext
# Get first 4 bytes of a data packet
# but swap them around
$1 = udp.blob;
$2 = blobbytes(substr( $1, 3, 1), substr($1, 2, 1),
substr($1, 1, 1), substr($1, 0, 1);
```

### byte()

The byte() function returns a byte from a specified offset in a string. The returned value is signed.

You can use the ubyte() function to return an unsigned value.

The byte() function is useful for determining if all of the desired fields have been gathered (for example, to determine if the byte at offset 40 in a particular string contains the expected value).

**Syntax**

byte(string str, int offset);

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str</td>
<td>The string to be examined. The string begins at byte 0.</td>
</tr>
<tr>
<td>int offset</td>
<td>The offset location in the string for the byte to be returned.</td>
</tr>
</tbody>
</table>

**Return Value**

- **int** - the offset in the string as a signed value
- **error** - the offset was out of bounds

**Example**

```plaintext
# the eighth byte is the signed type code
$code = byte(udp.blob, 7);
```

### csv_open()

The csv_open() function opens a CSV file.

**Syntax**

int csv_open(int fd);

**Parameters**

int fd – the file descriptor returned from io_open/url_fetch
### csv_open()

<table>
<thead>
<tr>
<th><strong>Return Value</strong></th>
<th>int – file handle to parsed CSV file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
</tbody>
</table>
|                  | ```
$fd = url_fetch("http://www.mysite.com/myfile.csv");
$csvp = csv_open($fd);
...
$numrows = csv_getrows($csvp,1);
$current_row = 0;
while ($current_row < $numrows) {
   echo("first value in row ", $current_row, ":",
   csv_getval($csvp,$current_row,1));
   $current_row = $current_row + 1;
}
csv_close($csvp);
io_close($fd)
``` |

### csv_close()

The csv_close() function closes a CSV file previously opened with csv_open().

<table>
<thead>
<tr>
<th><strong>Syntax</strong></th>
<th>csv_close(int csvp);</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
<td>Int csvp – the file descriptor returned from csv_open.</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>None</td>
</tr>
</tbody>
</table>
| **Example** | ```
$fd = io_open("./myfile.csv");
$csvp = csv_open($fd);
...
csv_close($csvp);
``` |

### csv_getcols()

The csv_getcols() function returns the number of columns in a particular row in a CSV file previously opened with csv_open().

<table>
<thead>
<tr>
<th><strong>Syntax</strong></th>
<th>int csv_getcols(int csvp, int row);</th>
</tr>
</thead>
</table>
| **Parameters** | - Int csvp – the file descriptor returned from csv_open.  
                - Int row – which row to get columnar count |
| **Return Value** | Int – number of columns in the specified row |
### csv_getcols()

**Example**

```php
$fd = io_open("./myfile.csv");
$csvp = csv_open($fd);
...
$numcols = csv_getcols($csvp, 1);
echo("numcols",

```
The `caseprefix()` function compares two strings and returns true if the second string is a prefix of the first string, without regard to case.

Unlike the `index()` and `prefix()` functions, the `caseprefix()` function performs a case-insensitive comparison of the second argument against the beginning of the first argument.

You can use the `prefix()` function for a case-sensitive comparison.

### `caseprefix()`

**Syntax**

```php
caseprefix(string big, string little);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string big</td>
<td>The larger string, which is the one compared against.</td>
</tr>
<tr>
<td>string little</td>
<td>The smaller string, which is compared to the beginning of big.</td>
</tr>
</tbody>
</table>

**Return Value**

- 1 - little is the same as the beginning of big, regardless of case
- 0 - little is not the same as the beginning of big

**Example**

```php
caseprefix("ABCDEFG", "ABC") == 1
caseprefix("ABCDEFG", "abc") == 1
caseprefix("ABCDEFG", "DEF") == 0
```

### `casesuffix()`

The `casesuffix()` function compares two strings and returns true if the second string is a suffix of the first string, without regard to case.

You can use the `suffix()` function for a case-sensitive comparison.

**Syntax**

```php
casesuffix(string big, string little);
```
**casesuffix()**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>string big</td>
<td>The larger string, which is the one compared against.</td>
</tr>
<tr>
<td></td>
<td>string little</td>
<td>The smaller string, which is compared to the end of big.</td>
</tr>
</tbody>
</table>

**Return Value**

- 1 - little is the same as the end of big, regardless of case
- 0 - little is not the same as the end of big

Unlike the suffix() function, the casesuffix() function performs a case-insensitive comparison of the second argument against the end of the first argument.

**Example**

casesuffix("ABCDEFG", "EFG") == 1  
casesuffix("ABCDEFG", "efg") == 1  
casesuffix("ABCDEFG", "ABC") == 0

---

**cat()**

The cat() function concatenates arguments by appending them into a single string. Non-string arguments are converted to their textual representation before concatenation. For list arguments, each element in the list is appended to the new string.

<table>
<thead>
<tr>
<th>cat()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
</tbody>
</table>
| Return Value | string - a blob/string  
The cat() function will ignore NULL parameters. Therefore, a statement such as the following would be safe when $blob has not yet been initialized:  
$blob = cat($blob, $data);  
When using cat() to concatenate an array or a list with an array member, the textual representation of the array must not exceed 8KB. |
| Example | # Keep data around in a single blob  
if ($blob == null)  
    $blob = $newdata;  
else  
    $blob = cat($blob, $newdata); |

---

**debase64()**

The debase64() function decodes a base64-encoded string.

The debase64() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.
The debase64() function may be impacted by the change in the behavior of string variables. See str for additional information about this behavior.

debase64()

Syntax
debase64(string str);

Parameters
- string str - the string to decode

Return Value
- string - the decoded input string

The overall return string for the debase64() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

Example

```
# Decode email attachment
$email_attachment = debase64($email_attachment);
# IMAP4 authentication challenges and responses
# are also base64 encoded
$challenge = debase64($challenge);
```

defhex()

The dehex() function undoes the ASCII translation performed for HTTP requests. When sending special characters in a URL, the HTTP specification says to encode them as their hexadecimal values preceded by a percent sign (%). The dehex() function looks for such metacharacters and returns them to their original values.

The dehex() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The dehex() function may be impacted by the change in the behavior of string variables ("str" on page 25).

defhex()

Syntax
defhex(string str, [string delimiter]);

Parameters
- string str - a string that contains hex characters
- string delimiter - a single-character string used as the delimiter, with a default value of %.

Return Value
- string - a string that has had its characters changed back to the original values

The overall return string for the dehex() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

Example

```
dehex("abcd%41%42%43%44") == "abcdABCD"
dehex("abcd%4a%4b%4c%4d") == "abcdJKLM"
dehex("abcd%zz") = "abcd%zz"
```

deutf8()

The deutf8() function decodes UTF-8 encoding, first passing the input string through the dehex() function.

The deutf8() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The deutf8() function may be impacted by the change in the behavior of string variables ("str" on page 25).
deutf8()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>deutf8(string str);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string str - the string to decode</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - the decoded input string</td>
</tr>
</tbody>
</table>

The overall return string for the debase64() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

Example

```
# Decode email attachment
$email_attachment = debase64($email_attachment);
# IMAP4 authentication challenges and responses
# are also base64 encoded
$challenge = debase64($challenge);
```

dns_expand()

The dns_expand() function decodes a DNS-encoded string. It operates safely on recursive names and operates correctly on heavily nested names.

DNS names may be compressed and some needed information may reside at the beginning or end of the string. Therefore, it is important to pass the entire string into dns_expand(), not just the relevant substring.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>string dns_expand(string str, int offset);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string str - the string containing the encoded name</td>
</tr>
<tr>
<td></td>
<td>int offset - the offset at which to start decoding the encoded name</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - the lowercase decoded name, or an error string (for example, &quot;Recursive pointer&quot; and &quot;Bad pointer&quot;).</td>
</tr>
</tbody>
</table>

Example

```
# Decode a SOA record with 2 contiguous strings
# (name and admin)
$name = dns_expand($dns_blob, $offset);
if (!$next = dns_next($dns_blob, $offset)) return -1;
$admin = dns_expand($dns_blob, $next);
if (!$next = dns_next($dns_blob, $next)) return -1;
$offset = $next;
# $offset will point to the next resource record
```

dns_next()

The dns_next() function finds the next DNS resource record in a string, after a decoded name. It returns the offset into the string of the next DNS RR or a NULL error indication if it could not be fully or correctly decoded.
### dns_next()

**Syntax**

```c
int dns_next(string str, int offset);
```

**Parameters**

- `string str` - the string containing multiple DNS resource records
- `int offset` - the offset at which to start decoding the current encoded name

**Return Value**

- `int` - the offset into the string of the next resource record
- `NULL` - the encoded name could not be fully or correctly decoded

**Example**

```c
# Decode SOA record which contains
# two contiguous strings (name and admin)
$name = dns_expand($dns_blob, $offset);
if (!$next = dns_next($dns_blob, $offset))
    return -1;
$admin = dns_expand($dns_blob, $next);
if (!$next = dns_next($dns_blob, $next)) return -1;
$offset = $next;
# And $offset points to next resource record
```

---

### elem()

The `elem()` function extracts a single element from a list. If you do not specify an index, the function returns the first element in the list.

**Syntax**

```c
elem(list li, [int index := 0]);
```

**Parameters**

- `list li` - the list from which to extract the element
- `int index` - the index to the element you want to extract. The index value starts at zero (that is, an index value of 0 extracts the first element).

**Return Value**

- `any` - an element from the list.

Full run-time type checking is performed on the return value and any variables that depend on it.

**Example**

```c
# Extract the third element in the list
portlist = [21, 23, 53, 80];
third = elem(portlist, 2);
third == 53;
```

---

### errno()

The `errno()` function returns the Boolean value associated with an error.
**errno()**

Syntax

```plaintext
errno(any argument);
```

Parameters

- any argument - the expression you want evaluated for error status

Return Value

int - a Boolean value corresponding to the error status of the argument, where:

- 1 - the argument evaluated to an error
- 0 - the argument did not evaluate to an error

The `errno()` function has changed significantly beginning with CPcode v3. It now returns a Boolean integer instead of a numeric value. Unlike past CPcode versions, the CPcode execution engine will stop if a catastrophic error is encountered.

Example

```plaintext
Print("Error status ", errno($foo + $bar));
```

**ethmac()**

The `ethmac()` function constructs an Ethernet MAC address from the supplied arguments.

Syntax

```plaintext
ethmac({string, int} a, [int b, int c, int d, int e, int f]);
```

Parameters

- `{string, int} a` - either the first octet or a string with the textual representation of the MAC address
- int b - the second octet
- int c - the third octet
- int d - the fourth octet
- int e - the fifth octet
- int f - the sixth octet

Return Value

`ethmac` - the MAC address

Example

```plaintext
ethmac("00:00:ba:de:ca:fe") == 00:00:BA:DE:CA:FE
ethmac(0, 0, 0xba, 0xde, 0xca, 0xfe) == 00:00:BA:DE:CA:FE
```

**filename()**

The `filename()` function will deobfuscate ("cleanse") a filename. It contracts ../ and . directory hiding and removes superfluous //.

The `filename()` function has a pre-optimized fastpath, where `str` is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The `filename()` function may obfuscate the actual occurrences when symbolic links are present. For example:

```plaintext
mkdir /tmp/link
cd /tmp/link
ln -s /usr foo
```
The filename() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

Depending on the protocol being decoded, it may be necessary to run the input filename through the deutf8() or dehex() functions before passing it through filename().

The overall return string for the filename() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

### filename()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>filename(string str, [int BackSlashFlag := 0]);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>• string str - the original filename&lt;br&gt;• int BackSlashFlag - Boolean used to treat the filename as a Microsoft Windows filename</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - a cleansed filename</td>
</tr>
<tr>
<td>Example</td>
<td>filename(&quot;/usr/..../etc/passwd&quot;) == &quot;/etc/passwd&quot; &lt;br&gt;filename(&quot;../etc//passwd&quot;) == &quot;../etc/passwd&quot; &lt;br&gt;filename(&quot;/etc/foo..bar&quot;) == &quot;/etc/foo../bar&quot; &lt;br&gt;filename(&quot;/etc/././foo&quot;) == &quot;/etc/foo&quot; &lt;br&gt;filename(&quot;/etc/.../foo&quot;) == &quot;/etc/.../foo&quot;</td>
</tr>
</tbody>
</table>

### formattedtime()

The formattedtime() function returns the time of day as a formatted string.

The formattedtime() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

The formattedtime() function is extremely slow and should be used judiciously.

The overall return string for the formattedtime() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>formattedtime(int time,[string format]);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>• int time - the time to be formatted&lt;br&gt;• string format - the format to use for displaying the time. Refer to the strftime in UNIX man pages for formatting options. If a format is not specified, the following default format is used:&lt;br&gt;• [%m%d%y %H:%M:%S]</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - a string containing the time in the format specified</td>
</tr>
</tbody>
</table>
hatoi()

The hatoi() function converts a two’s complement hex string to an integer. If you specify an index, the conversion begins at that location inside the string.

If the leading bit is set to (1<<31), then the return value will be a negative number.

The hatoi() function ignores any white space leading up to the number. However, unlike the atoi() function, hatoi() will return a NULL if there are any characters after the hex number or if the hex value overflows the size of the CPcode integer.

Syntax

```
hatoi(string str, [int index := 0]);
```

Parameters

- `string str` - the hex string that is to be converted to an integer
- `int index` - the point in the string at which to begin the conversion

Return Value

- `int` - the integer value of the hex string, `str`
- `NULL` - the string is empty, contains characters that are not valid hex digits, or is too large to fit inside an CPcode integer.

Example

```
{ hatoi("10") == 16
  hatoi("ff") == 255
  hatoi("ffXX") == NULL
  hatoi(" 80000000") == -2147483648
  hatoi(" ffffffff") == NULL
  hatoi("monkeys: 41", 8) == 65
  hatoi(""") == NULL
  hatoi(42) == NULL
}
```

host()

The host() function constructs an IPv4host address from other parameters (for example, information in a data stream).
For example, the PORT command in FTP sends the octets of an IP address as a comma-separated, printable list. The `host()` function could be used to create an IP address from the list. The `long()` function could be used to gather information from a packet, using the single-integer syntax of the `host()` function to create an IP address. This can be easier than extracting each octet individually.

### `host()`

**Syntax**

```
host((ip, string, list, int) a, [int b, int c, int d]);
```

**Parameters**

- `{ip, string, list, int}` - a 32-bit, host-endian integer containing:
  - a list of four integers representing each octet in the address
  - the full IPv4 address
  - a string representation of an IP address
  - the first of four integers representing each IP address octet
  - `int b` - in the case of four octets, the second octet
  - `int c` - in the case of four octets, the third octet
  - `int d` - in the case of four octets, the fourth octet

**Return Value**

- `ip` - an IPv4 address in network byte order

**Example**

```cpp
$foo = host(208, 217, 179, 1);
$foo = host(0xD0D9b301);
$foo = host([208, 217, 179, 1]);
$foo = host("208.217.179.1");
$foo = host(208.217.179.1);
```

### `host6()`

The `host6()` function constructs an IPv6 host address from the supplied parameter.

**Syntax**

```
host6(string address, [int binary]);
```

**Parameters**

- `string address` - an textual or binary IPv6 address
- `int binary` - tells `host6()` whether to parse the address parameter as a binary 16-byte address, where:
  - `0` - indicates that the address parameter is textual
  - `1` - indicates that the address parameter is binary

**Return Value**

- `ip` - an IPv6 address in network byte order

**Example**

```cpp
$foo = host6("::1");
$foo = host6("abcd:0123::5432:1000", 0);
```

### `incksum()`

The `incksum()` function computes a 16-bit checksum, summing two characters at a time, which corresponds to the IP header checksum.

`incksum()` can be used to either verify or compute checksums.
index()

The index() function searches for the first occurrence of the little string in the big string. The index count starts with zero.

Using a list of little strings may not always be an ideal solution. If the list is static, it may be significantly faster to use a regular expression before calling the index() function.

The prefix() function is a faster case of the index() function.

<table>
<thead>
<tr>
<th>incksum()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>incksum({string, list} in);</td>
</tr>
<tr>
<td>Parameters</td>
<td>• {string, list} in - the string or list of strings for which the checksum should be computed</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - an integer checksum</td>
</tr>
<tr>
<td>Example</td>
<td>incksum(&quot;foo bar&quot;) == 61865</td>
</tr>
<tr>
<td></td>
<td>incksum([&quot;foo&quot;, &quot;bar&quot;] ) == 53673</td>
</tr>
<tr>
<td></td>
<td>incksum([&quot;fo&quot;, &quot;o &quot;, &quot;ba&quot;, &quot;r&quot;]) == 61865</td>
</tr>
</tbody>
</table>

io_close()

The io_close() function closes a previously opened file descriptor.

Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

<table>
<thead>
<tr>
<th>io_close()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>io_close(int fd);</td>
</tr>
<tr>
<td>Parameters</td>
<td>Int fd – file descriptor to be closed</td>
</tr>
<tr>
<td>Return Value</td>
<td>None</td>
</tr>
<tr>
<td>Example</td>
<td>io_close($fd);</td>
</tr>
</tbody>
</table>
**io_delete()**

The `io_delete()` function deletes a file from the file system.

`io_*` functions do not integrate with the caching implementation provided with the `url_*` functions. If caching is desired or network access is desired, use the `url_*` functions instead.

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>io_delete (string filename);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td><code>Int fd</code> – file descriptor to be closed</td>
</tr>
<tr>
<td>Return Value</td>
<td>0 on success Error code otherwise</td>
</tr>
<tr>
<td>Example</td>
<td><code>$filename = &quot;/test.txt&quot;;</code> <code>io_delete($filename);</code></td>
</tr>
</tbody>
</table>

**io_open()**

The `io_open()` function returns an I/O descriptor for reading or writing. The file will be automatically closed when the calling CPCode filter exits.

`io_*` functions do not integrate with the caching implementation provided with the `url_*` functions. If caching is desired or network access is desired, use the `url_*` functions instead.

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>int io_open (string filename);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td><code>string filename</code> – filename to be opened</td>
</tr>
<tr>
<td>Return Value</td>
<td><code>int</code> – returns a valid file descriptor, NULL if the file does not exist</td>
</tr>
<tr>
<td>Example</td>
<td><code>$fd = io_open(&quot;./test.txt&quot;);</code></td>
</tr>
</tbody>
</table>

**io_read()**

The `io_read()` function reads data from an I/O descriptor.

`io_*` functions do not integrate with the caching implementation provided with the `url_*` functions. If caching is desired or network access is desired, use the `url_*` functions instead.

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>list io_read (int fd);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td><code>int</code> – file descriptor to be read</td>
</tr>
<tr>
<td>Return Value</td>
<td><code>list</code> – list of strings representing the contents of the file. NULL if file descriptor is invalid.</td>
</tr>
</tbody>
</table>
### io_readlines()

The `io_readlines()` function reads data from an I/O descriptor.

Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

#### Syntax

```
list io_readlines(int fd, <int chomp>);
```

#### Parameters

- `int fd` – file descriptor to be read
- `int chomp` – optional flag indicating whether or not to remove newline characters from the lines returned. Will remove "\n" as well as "\r\n."

#### Return Value

- `list` – list of strings representing the contents of the file.
- `NULL` if file descriptor is invalid.

#### Example

```php
$fd = io_open("./test.txt");
...
$contents = io_readlines($fd);
io_close($fd);
```

---

### io_write()

The `io_write()` function writes data to an I/O descriptor. Writes do not append lines to a file, rather a single io() call will replace the file in question.

Io_* functions do not integrate with the caching implementation provided with the url_* functions. If caching is desired or network access is desired, use the url_* functions instead.

#### Syntax

```
list io_write(int fd, list data);
```

#### Parameters

- `int fd` – file descriptor to be written to.
- `list data` – A list of strings to write into a file.

#### Return Value

`None`

#### Example

```php
$fd = io_open(cat(scope(),"-known-ports");
...
io_write($fd,["80","8080","9090"]);
io_close($fd);
```
kiss_compile()

A different name for the kiss_pattern() function. See kiss_pattern() (on page 71).

kiss_pattern()

The kiss_pattern() function creates a list of patterns into a single multi-pattern search. It behaves and is used in a manner similar to the multi_pattern() function.

<table>
<thead>
<tr>
<th>kiss_pattern()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
<tr>
<td>Example</td>
</tr>
</tbody>
</table>

kiss_search()

The kiss_search() function compares a kiss_pattern expression to a given string. It behaves and is used in a manner similar to multi_search();

<table>
<thead>
<tr>
<th>kiss_search()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
| Example       | MY_PATTERN = kiss_compile([ ["x0",1], ["x0x0",2], ["y0y0",3], ["y0y0dyne",4] ])

    func my_search {
        if ($matches = kiss_search(MY_PATTERN,$1)) {
            foreach $elm inside ($matches) {
                if ($elm == 1) echo ("x0");
                else if ($elm == 2) echo ("x0x0");
                else if ($elm == 3) echo ("y0y0");
                else if ($elm == 4)
                    echo ("Is your name John Smith?");
            }
        }
    }
The `le_long()` function returns a four-byte signed integer from the specified offset within a string. It extracts the long integer in host order without converting it.

### `le_long()`

**Syntax**

`le_long(string str, int offset);`

**Parameters**
- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**
- `int` - the offset integer, as a signed value
- `error` - the offset was out of bounds

**Example**

```
le_long("^A^@^@^@", 0) == 0x00000001;
le_long("^@^A^@^@", 0) == 0x00000100;
le_long("^@^@^A^@", 0) == 0x00010000;
```

The `le_short()` function returns a two-byte signed integer from the specified offset within a string. It extracts the short integer in host order without converting it.

### `le_short()`

**Syntax**

`le_short(string str, int offset);`

**Parameters**
- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**
- `int` - the offset integer, as a signed value
- `error` - the offset was out of bounds

**Example**

```
le_short("^A^@", 0) == 0x00000001;
le_short("^@^A", 0) == 0x00000100;
```

The `le_ulong()` function returns a four-byte unsigned integer from the specified. It extracts the long integer in host order without converting it.

The `le_ulong()` function returns the same result as the `le_long()` function. It is provided for backwards compatibility. The internal representation of all CPcode integers is signed. Therefore, the `le_ulong()` function yields a signed integer.

### `le_ulong()`

**Syntax**

`le_ulong(string str, int offset);`

**Parameters**
- `string str` - the string to be examined, beginning at byte 0
- `int offset` - the offset in the string for the first byte to be returned

**Return Value**
- `int` - the offset integer, as an unsigned value
- `error` - the offset was out of bounds
The le_ushort() function returns a two-byte unsigned integer from a specified offset within a string. It extracts the short integer in host order without converting it.

**le_ushort()**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>le_ushort(&quot;^A^@^@^@&quot;, 0) == 0x00000001;</td>
</tr>
<tr>
<td>le_ushort(&quot;^@^A^@^@&quot;, 0) == 0x00000100;</td>
</tr>
<tr>
<td>le_ushort(&quot;^@^@^A^@&quot;, 0) == 0x00010000;</td>
</tr>
<tr>
<td>le_ushort(&quot;\x00\xff\xff\xff&quot;, 0) == -256</td>
</tr>
</tbody>
</table>

**le_ulong()**

The le_ulong() function returns a two-byte unsigned integer from a specified offset within a string. It extracts the short integer in host order without converting it.

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>le_ulong(string str, int offset);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>string str - the string to be examined, beginning at byte 0</td>
</tr>
<tr>
<td>int offset - the offset in the string for the first byte to be returned</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int - the offset integer, as an unsigned value</td>
</tr>
<tr>
<td>error - the offset was out of bounds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>le_ulong(&quot;^A^@^@^@&quot;, 0) == 0x00000001;</td>
</tr>
<tr>
<td>le_ulong(&quot;^@^A^@^@&quot;, 0) == 0x00000100;</td>
</tr>
</tbody>
</table>

The listadd() function constructs a list at runtime, or adds items to an existing list.

NULL items are added to the list.

The listadd() function has a pre-optimized fastpath, where li, ... is its most relevant parameter. It operates more efficiently if its input list is the same CPcode variable as its output list.

However, duplicating the list may be an expensive operation, as in the example:

```perl
$listB = listadd($listA, 1, 2, 2);  
```

<table>
<thead>
<tr>
<th>listadd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>listadd(list li, any item, [...]);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>list li - the existing list. If this value is NULL, the function creates an empty list and adds to that list.</td>
</tr>
<tr>
<td>any item - the first item to be added to the tail of the list</td>
</tr>
<tr>
<td>... - a series of items to be added to the tail of the list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The original list, or the newly constructed list, with the items added to it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| badIPs = [ 208.217.179.1, 208.217.179.3 ];  
| badIPs = listadd(badIPs, ip.src, ip.dst);  
| A = [1, 2, 3];  
| B = [4, 5, 6];  
| C = listadd(A, B); # C == [1, 2, 3, [4, 5, 6]] |
listcombine()

The listcombine() function combines two lists. It appends each element from the second list to the first list. The listcombine() function differs from the listadd() function, which adds the entire second list to the first list instead of adding each individual element of the second list.

The listcombine() function has a pre-optimized fastpath, where list is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

<table>
<thead>
<tr>
<th>listcombine()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>list listcombine(list a, list b);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>list a - the first list to process</td>
</tr>
<tr>
<td>list b - the second list to process</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>list - a list containing all the elements of both lists</td>
</tr>
<tr>
<td>The original list, or the newly constructed list, with the items added to it.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>a = [1, 2, 3];</td>
</tr>
<tr>
<td>b = [4, 5, 6];</td>
</tr>
<tr>
<td>c = listcombine(a, b);</td>
</tr>
<tr>
<td>c == [1, 2, 3, 4, 5, 6]</td>
</tr>
</tbody>
</table>

listglom()

See listcombine() (on page 74)

listlen()

The listlen() function returns a count of the number of elements in a list.

<table>
<thead>
<tr>
<th>listlen()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>listlen(list li);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>list li - the list whose elements are to be counted</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>int - the number of elements in the list</td>
</tr>
<tr>
<td>error - the parameter is not a list</td>
</tr>
<tr>
<td>The listlen() function does not count the elements of sublists as separate elements.</td>
</tr>
<tr>
<td>The listlen() function is faster than the description implies. Lists keep an internal count of their members, which this function returns.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>portlist = [21, 23, 80];</td>
</tr>
<tr>
<td>listlen(portlist) == 3;</td>
</tr>
<tr>
<td>superlist = [1, 2, 3, [4, 5, 6]];</td>
</tr>
<tr>
<td>listlen(superlist) == 4;</td>
</tr>
</tbody>
</table>
**long()**

The `long()` function returns a four-byte signed integer from a specified offset within a string. It converts the value from network byte order in the string to the host’s byte order. Network byte order is big endian, therefore the most significant byte is stored at the lowest address in memory.

The `long()` function is useful for extracting values from packets.

<table>
<thead>
<tr>
<th>long()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><code>long(string str, int offset);</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>- string str - the string to be examined, beginning at byte 0</td>
</tr>
<tr>
<td>- int offset - the offset in the string for the first byte to be returned</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>- int - the offset integer, as a signed value</td>
</tr>
<tr>
<td>- error - the offset was out of bounds</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
| `# the structure starts with a 4-byte
# integer in network byte order.
# It will be converted to host byte
# order before placing it in $data
$data = long($struct_data, 0);` |

**lsort()**

The `lsort()` function sorts the members of a list according to type, grouping elements of the same type together. Within each data type, the `lsort()` function sorts the elements in ascending order.

The order of the elements within the group is not guaranteed.

The `lsort()` function has a pre-optimized fastpath, where `li` is its most relevant parameter. It operates more efficiently if its input list is the same CPcode variable as its output list.

<table>
<thead>
<tr>
<th>lsort()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><code>lsort(list li);</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>- list li - the list to be sorted</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>- list - the sorted list</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
| `foo = [ 8, 4, 2, 9 10, 6, 4 ];
echo(lsort(foo));     # Slow
# [2, 4, 4, 6, 8, 9, 10]
foo = [ 8, 4, 2, 9 10, 6, 4 ];
foo = lsort(foo);     # Fast
echo(foo);
# [2, 4, 4, 6, 8, 9, 10]` |
The luniq() function creates a list of the unique elements from another list (that is, removes duplicates). It compares adjacent elements of a list. If they are equal, the luniq() function adds only one of the elements to the new list. If they are not equal, the luniq() function adds both elements to the list.

The luniq() function determines if elements are equal using the criteria for the == operator.

To guarantee that the new list contains unique elements, ensure that the list passed to the luniq() function is sorted first. It is much faster to assign the sorted result to the list being sorted than it is to use a temporary list, either explicitly or implicitly.

### luniq()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>luniq(list li);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>list li - the list from which duplicate elements will be removed</td>
</tr>
<tr>
<td>Return Value</td>
<td>list - a new list with duplicate adjacent elements removed</td>
</tr>
<tr>
<td></td>
<td>error - the parameter is not a list</td>
</tr>
</tbody>
</table>

#### Example

```c
foo = [ 8, 4, 2, 8 10, 6, 4 ];
echo(luniq(foo));    # Slow
# [8, 4, 2, 8, 10, 6, 4]
foo = [ 8, 8, 9, 9 10, 10, 4 ];
foo = luniq(foo);
echo(foo);
# [8, 9, 10, 4]
```

### match()

See regexec() (on page 83).

### multi_compile()

See multi_pattern() (on page 77).

### multi_escape()

The multi_escape() function escapes any reserved special characters in a string. If used in conjunction with the multi_pattern() function, the multi_pattern() function can then accept the string literally.

<table>
<thead>
<tr>
<th>multi_escape()</th>
<th>multi_escape(string str);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string str - the string to escape</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - the escaped string</td>
</tr>
</tbody>
</table>
multi_pattern()

The `multi_pattern()` function creates a complex regular expression from a list of multiple patterns. Unlike normal regular expressions, a multi-pattern search can perform the match function in a single iteration through the search string. A typical multi-pattern search is sub 0(n).

The allowable regular expression characters are:

- +
- *
- ?
- ^
- $
- .
- [

Multi-pattern search compilation is resource intensive and should be done only once and then assigned to a global variable.

The `multi_pattern()` function should never be called from a filter.

### Syntax

```c
multi_pattern(list pattern_list, [int case_insensitive]);
```

### Parameters

- **list pattern_list** - the list of patterns and their return tokens
- **int case_insensitive** - tells `multi_pattern()` whether the pattern is case insensitive, where:
  - 0 - indicates that the pattern is case sensitive
  - a non-zero integer - indicates that the pattern is case insensitive

### Return Value

- **string** - a complex regular expression
**multi_pattern()**

<table>
<thead>
<tr>
<th>Example</th>
<th>multi_pattern( [ [&quot;one&quot;, 1], [&quot;two&quot;, 2], [&quot;three&quot;, 3] ] );</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>multi_search(example, &quot;one three&quot;) == [1,3]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;two&quot;) == [2]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;three&quot;) == [3]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;THREE&quot;) == NULL</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;foo&quot;) == NULL</td>
</tr>
<tr>
<td></td>
<td>example = multi_pattern( [ [&quot;[oO][nN][eE]&quot;, 1] ] );</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;one&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;ONE&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;oNe&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx oNe xxx&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;foo&quot;) == NULL</td>
</tr>
<tr>
<td></td>
<td>example = multi_pattern( [ [&quot;^[oO][nN][eE]&quot;, 1] ] );</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;one&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;ONExxx&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;oNexxx&quot;) == [1]</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx oNe xxx&quot;) == NULL</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;foo&quot;) == NULL</td>
</tr>
<tr>
<td></td>
<td>example = multi_pattern( [ [&quot;\x90\x90&quot;, &quot;NOP string&quot;], [&quot;root&quot;, &quot;root login&quot;] ] );</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx \x90\x90\x90\x90 xxx&quot;) == &quot;NOP string&quot;</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx root xxx&quot;) == &quot;root login&quot;</td>
</tr>
<tr>
<td></td>
<td>example = multi_pattern( [ [&quot;foo&quot;, &quot;bar&quot;, &quot;zah&quot;] ] );</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx foo xxx&quot;) == &quot;foo&quot;</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx zah xxx&quot;) == &quot;zah&quot;</td>
</tr>
<tr>
<td></td>
<td>multi_search(example, &quot;xxx dog xxx&quot;) == NULL</td>
</tr>
</tbody>
</table>

**multi_search()**

The multi_search() function compares a multi-pattern expression to a string and returns the tokens associated with all matches.

| multi_search() | multi_search(string multi_pattern, string str); |
**multi_search()**

<table>
<thead>
<tr>
<th>Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string multi_pattern - the multi-pattern expression to be compared</td>
<td></td>
</tr>
<tr>
<td>string str - the string to be compared</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>list - the matched patterns' tokens</td>
<td></td>
</tr>
<tr>
<td>NULL - no string match was found within the multi-pattern expression</td>
<td></td>
</tr>
</tbody>
</table>

**Example**

See the multi_pattern() function Examples (see "multi_pattern()" on page 77).

---

**net()**

The net() function uses the supplied arguments to construct an IPv4 address from several different data types.

### net()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>net({string, list, ip, int} a, {[ip, int] b, int c, int d, int e, int f, int g, int h});</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{string, list, ip, int} a - a 32-bit host-endian integer containing:</td>
<td></td>
</tr>
<tr>
<td>a list of eight integers representing each octet in the address/mask</td>
<td></td>
</tr>
<tr>
<td>a full IPv4 address</td>
<td></td>
</tr>
<tr>
<td>a string representation of an IP address and netmask</td>
<td></td>
</tr>
<tr>
<td>the first of four integers representing each IP address octet</td>
<td></td>
</tr>
<tr>
<td>{ip, int} b - one of the following:</td>
<td></td>
</tr>
<tr>
<td>in the case of eight octets, the second octet</td>
<td></td>
</tr>
<tr>
<td>in the case of two 32-bit integers, the netmask</td>
<td></td>
</tr>
<tr>
<td>in the case of two network variables, the netmask</td>
<td></td>
</tr>
<tr>
<td>int c - the third octet of the network address</td>
<td></td>
</tr>
<tr>
<td>int d - the fourth octet of the network address</td>
<td></td>
</tr>
<tr>
<td>int e - the first octet of a netmask, or the netmask in CIDR notation</td>
<td></td>
</tr>
<tr>
<td>int f - the second octet of a netmask</td>
<td></td>
</tr>
<tr>
<td>int g - the third octet of a netmask</td>
<td></td>
</tr>
<tr>
<td>int h - the fourth octet of a netmask</td>
<td></td>
</tr>
</tbody>
</table>

| Return Value     | ip - an IPv4 address/network in network byte order |
The net6() function constructs an IPv6 network from the supplied parameter.

<table>
<thead>
<tr>
<th>net6()</th>
<th>net6(string network);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>string network - a textual IPv6 network</td>
</tr>
<tr>
<td>Return Value</td>
<td>ip - an IPv6 network in network byte order</td>
</tr>
<tr>
<td>Example</td>
<td>$foo = net6(&quot;::1/128&quot;);</td>
</tr>
<tr>
<td></td>
<td>$foo = net6(&quot;abcd:0123::/32&quot;);</td>
</tr>
</tbody>
</table>

pattern()

See regcomp() (on page 82)

pred()

The pred() function finds and returns the logical predecessor of the indexed item in an array. The pred() function allows reverse foreach-like behavior, but at a larger performance cost.

<table>
<thead>
<tr>
<th>pred()</th>
<th>pred(array arr, any index);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td></td>
</tr>
</tbody>
</table>

Example

$foo = net(208, 217, 179, 0, 24);    # 208.217.179.0/24
$foo = net(0xD0D9b300, 0xffffff0);   # 208.217.179.0/24
$foo = net([208, 217, 179, 0, 255, 255, 255, 0]);  # 208.217.179.0/24
$foo = net([208, 217, 179, 0, 24]);    # 208.217.179.0/24
$foo = net("208.217.179.0/255.255.255.0");  # 208.217.179.0/24
$foo = net("208.217.179.0/24");        # 208.217.179.0/24
$foo = net([208, 217, 179, 0, 24]);     # 208.217.179.0/24
$foo = net(208.217.179.0/24);           # 208.217.179.0/24
$foo = net(208.217.179.0, 24);          # 208.217.179.0/24
$foo = net(208.217.179.0/24);           # 208.217.179.0/24
$foo = net(208.217.179.0, 24);          # 208.217.179.0/24

### pred()

**Parameters**
- array arr - the array from which to extract the predecessor
- any index - the index to the item whose predecessor will be extracted

**Return Value**
- any - the predecessor of the indexed item in the array (that is, the item that will be extracted)

**Example**
```c
array[0] = "zero";
array[1] = "one";
array[2] = "two";
pred(array, -1) == NULL
pred(array, 0) == NULL
pred(array, 1) == 0
pred(array, 2) == 1
pred(array, 500) == NULL
```

### prefix()

The prefix() function compares two strings and returns true if the second string is a prefix of the first string.

The prefix() function is a faster case of the index() function.

Use the caseprefix() function for a case-insensitive comparison.

**Syntax**
```
prefix(string big, string little);
```

**Parameters**
- string big - the string against which little will be compared
- string little - the smaller string, which will be compared to the beginning of big

**Return Value**
- int, where:
  - 1 - little is the same as the beginning of big
  - 0 - little is not the same as the beginning of big

**Example**
```c
prefix("ABCDEFG", "ABC") == 1
prefix("ABCDEFG", "aBc") == 0
prefix("ABCDEFG", "DEF") == 0
```

### prunearray()

The prunearray() function randomly removes a specified percentage of elements in an array, and is intended to be used in conjunction with the overflow memory exception and the system.pressure packet variable.

If CPcode uses a global state table in an array, the system stability can be put at risk if the array grows arbitrarily large and uses all of the system's memory. The prunearray() function should be used to randomly (and fairly) free part of those global arrays.

**Syntax**
```
prunearray(array ar, int percent);
```
**prunearray()**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array ar</td>
<td>the array to prune</td>
</tr>
<tr>
<td>int percent</td>
<td>the percentage of the array to prune</td>
</tr>
</tbody>
</table>

| Return Value        | array - the pruned array                         |

| Example             | STATE_TABLE = prunearray(STATE_TABLE, system.pressure) |

---

**regcomp()**

The `regcomp()` function creates and compiles a regular expression from a string. The compiled regular expression can be compared against other strings using the `regex()` function.

The `regcomp()` function is not a fast operation. Therefore, it should be used only on global variables, and only once for each variable.

To mitigate the performance impact of regular expression compilation from a filter, there is a 256-entry, fully associative `regcomp` cache. However, it is still better to execute a single `regcomp()` on a global variable.

The `regcomp()` function recognizes those regular expressions that are recognized by `grep` or the `regcomp()` function on UNIX systems.

The `regcomp()` function does not work on strings that contain Nulls.

<table>
<thead>
<tr>
<th>regcomp()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>regcomp(string pattern);</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string pattern</td>
<td>the string to be converted into a regular expression</td>
</tr>
</tbody>
</table>

| Return Value        | string - a regular expression pattern           |
The `regexec()` function compares a regular expression to a string, and returns an integer result indicating whether the patterns matched. This function utilizes regular expressions compiled by the `regcomp()` function.

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>regexec(string pattern, string str);</code></th>
</tr>
</thead>
</table>
| Parameters   | • string pattern - the regular expression to be compared  
               • string str - the string to be compared |
| Return Value | int, where:  
               • 0 - patterns did not match  
               • 1 - patterns matched |
| Example      | example = regcomp("abc");  
               regexec(example, "defabc") == 1  
               example = "^abc";  
               regexec(example, "abcdef") == 1  
               regexec(example, "defabc") == 0 |
scope()

The scope() function returns the name of the scope in which the CPcode is currently executing. Calls to scope() are evaluated at compile time and are the functional equivalent of using a string literal with a scope name. For example:

```cpcode
$my_scope = scope();
```

translates to:

```
$my_scope = "ftp_commands";
```

<table>
<thead>
<tr>
<th>scope()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>scope();</td>
</tr>
<tr>
<td>Parameters</td>
<td>None</td>
</tr>
</tbody>
</table>
| Return Value | • string - a string showing the name of the scope in which the CPcode is currently executing. The string displays as:
| | • the protocol, an underscore, and the Protection Group name if this CPcode is for a Protection Group
| | • the protocol if this CPcode is protocol-wide CPcode |

Example

```cpcode
func foo {
    echo("debug:: foo called from", $1);
    ...
}
...
foo(scope());
```

sed()

The sed() function rewrites strings. It corresponds to the UNIX sed command (Stream EDitor), which simplifies later processing by replacing or removing items from a string.

The sed() function must compile the pattern into a regular expression, which is extremely slow. To alleviate the compilation bottleneck, the sed() function uses a 64-entry, fully associative regex cache. Using the same pattern is normally much faster than using many patterns. To this end, the cache is not LRU (least recently used); its replacement scheme is to remove the closest match to the new entry on the assumption that they were both dynamically generated from the same Protection Group. Dynamic patterns are discouraged and are penalized by low cache precedence.

<table>
<thead>
<tr>
<th>sed()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>sed(string pattern, string replace, string input, [int matchOnce :=0]);</td>
</tr>
</tbody>
</table>
| Parameters | • string pattern - the string pattern to be matched and replaced
| | • string replace - the replacement string
| | • string input - the input string
| | • int matchOnce - an optional Boolean where:
| | • 0 - replace all occurrences of the pattern
| | • 1 - replace the first occurrence of the pattern |
| Return Value | string - the rewritten string |
The short() function returns a two-byte signed integer from a specified offset within a string. It converts the value from network byte order in the string to the host’s byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The short() function is useful for extracting values from packets.

```
short(string str, int offset);
```

**Parameters**
- string str - the string to be examined, beginning at byte 0
- int offset - the offset in the string for the first byte to be returned

**Return Value**
- int - the offset integer, as a signed value
- error - the offset was out of bounds

**Example**
```
# Structure starts with 2-byte int
# in network byte order.
# It will be converted to host byte order
# before placing it in $data
$data = short($struct_data, 0);
```

The split() function breaks apart and returns a list of strings derived from another string. Characters before the specified delimiter become the first element in the list, and characters after the delimiter become the second element in the list, and so forth.
It is permissible to split on the NULL character or other binary characters by entering hex characters (for example, \x00 or \xff) in the string.

The split() function will ignore any leading delimiters in the string.

If no delimiter is specified, white space characters (a space, \t, \r, and \n) will become the default.

### split()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>split(string str,[string delimiter]);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>• string str - the string to be split</td>
</tr>
<tr>
<td></td>
<td>• string delimiter - the parameter on which the string should be split</td>
</tr>
<tr>
<td>Return Value</td>
<td>list - a list of strings</td>
</tr>
<tr>
<td>Example</td>
<td>split(&quot;www.company.com&quot;, &quot;.&quot;) == [&quot;www&quot;, &quot;company&quot;, &quot;com&quot;]</td>
</tr>
<tr>
<td></td>
<td>split(&quot;Foo and his pet rock&quot;) == [&quot;Foo&quot;, &quot;and&quot;, &quot;his&quot;, &quot;pet&quot;, &quot;rock&quot;]</td>
</tr>
<tr>
<td></td>
<td>split(&quot;One,Two Three;four&quot;, &quot; ,;&quot;) == [&quot;One&quot;, &quot;Two&quot;, &quot;Three&quot;, &quot;four&quot;]</td>
</tr>
<tr>
<td></td>
<td>split(&quot; ONE   TWO &quot;) == [&quot;ONE&quot;, &quot;TWO&quot;]</td>
</tr>
</tbody>
</table>

### str()

The str() function converts each parameter to its textual representation and returns a list of those strings.

The str() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>str(…);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>• … - parameters to convert to textual representation and put in a list</td>
</tr>
<tr>
<td>Return Value</td>
<td>list - a list of strings</td>
</tr>
<tr>
<td>Example</td>
<td>str(1, 2, &quot;three&quot;, 4.5.6.7) == [&quot;1&quot;, &quot;2&quot;, &quot;three&quot;, &quot;4.5.6.7&quot;]</td>
</tr>
</tbody>
</table>

### strcasecmp()

The strcasecmp() function compares two strings, character by character, ignoring case. Two characters are considered equal if they have the same byte value, or if they are alphabetically the same character without regard to case (for example, D is equal to d).

The strcasecmp() function is useful for comparing case-insensitive strings. Many protocols specify that strings must be case insensitive.

The strcasecmp() function has a legacy optimization to first compare string lengths. If one string is shorter than the other is, the character comparison is not done and <0 is returned.

The first occurrence of an unequal character causes the function to stop the comparison. If the characters are equal, comparison continues until the end of the string is reached.
strcasecmp()

<table>
<thead>
<tr>
<th>Syntax</th>
<th>strcasecmp(string s1, string s2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string s1 - the first string to be compared</td>
</tr>
<tr>
<td></td>
<td>string s2 - the second string, which will be compared against the first</td>
</tr>
<tr>
<td>Return Value</td>
<td>int &lt;0 - the first string is less than the second string</td>
</tr>
<tr>
<td></td>
<td>int 0 - the two strings are equal</td>
</tr>
<tr>
<td></td>
<td>int &gt;0 - the first string is greater than the second string</td>
</tr>
<tr>
<td>Example</td>
<td>if (strcasecmp($ftp_user, &quot;anonymous&quot;) == 0</td>
</tr>
</tbody>
</table>

strerror()

The `strerror()` function returns a Boolean string indicating whether an error occurred.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>strerror(any arg);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>any arg - the expression being evaluated</td>
</tr>
<tr>
<td>Return Value</td>
<td>string - a string indicating whether the argument evaluated as &quot;no error&quot; or &quot;unknown error&quot;</td>
</tr>
<tr>
<td>Example</td>
<td>$one = 1; $zero = 0; strerror($one + $zero) == &quot;no error&quot; strerror($one / $zero) == &quot;unknown error&quot; $div = $one / $zero; strerror($div) == &quot;unknown error&quot;</td>
</tr>
</tbody>
</table>

strlen()

The `strlen()` function returns the length, in bytes, of a string or blob.
Unlike other programming languages, CPcode does not use a trailing NULL character.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>strlen(string str);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>string str - the string whose length is to be determined</td>
</tr>
<tr>
<td>Return Value</td>
<td>int - the length of str</td>
</tr>
<tr>
<td>Example</td>
<td>strlen(&quot;123&quot;) == 3</td>
</tr>
</tbody>
</table>
**strstr()**

See index() (on page 68).

---

**strtype()**

The `strtype()` function returns an integer representing the type of data that is in a string.

The `strtype()` function is useful for performing one type of processing on data that contains only ASCII characters and another type of processing on binary data.

<table>
<thead>
<tr>
<th><strong>strtype()</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>strtype(string str);</code></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>• string str - the string whose data is to be evaluated</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
<td>int, where:</td>
</tr>
<tr>
<td></td>
<td>• 1 - NULL</td>
</tr>
<tr>
<td></td>
<td>• 2 - the string contains only digits</td>
</tr>
<tr>
<td></td>
<td>• 3 - the string contains only alphabetic characters, including ISO 8859-1 Latin 1 characters 192 through 255, except 215 and 247</td>
</tr>
<tr>
<td></td>
<td>• 4 - the string contains only spaces</td>
</tr>
<tr>
<td></td>
<td>• 5 - the string contains only white space, including ISO 8859-1 Latin-1 characters 10, 11, 12, 13, 32, and 160</td>
</tr>
<tr>
<td></td>
<td>• 6 - the string contains only alphanumeric characters</td>
</tr>
<tr>
<td></td>
<td>• 7 - the string contains only printable characters, including ISO 8859-1 Latin-1 characters 32 through 126 and 160 through 255</td>
</tr>
<tr>
<td></td>
<td>• 8 - the string contains only ASCII characters between 0 and 127</td>
</tr>
<tr>
<td></td>
<td>• 9 - the string contains only binary data</td>
</tr>
</tbody>
</table>
strtype()

Example

```cpp
func foo {
    type = strtype($1);
    if (type == 9)
        echo("The data has a string type of ", type, "\n");
    else
        echo("The data ", $1, " has a string type of ", type, "\n");
}
foo("21167");
foo("Company");
foo("Company123");
foo(""");
foo("Generic Company, Inc.");
foo("www.company.com\x00");
foo("\xff \x01");
The output is:
The data 21167 has a string type of 2
The data Company has a string type of 3
The data Company123 has a string type of 6
The data has a string type of 1
The data Generic Company, Inc. has a string type of 7
The data www.company.com has a string type of 9
The data has a string type of 9
```

sublist()

The sublist() function returns a list that is a subset of another list. It creates a new list, starting with the element specified as the start, and continuing for the number of elements specified for the length.

When using the sublist() function, remember that the first character in the string is always at zero.

If a length is not included, the new list starts at the offset and uses the remaining elements in the old list. If the starting offset is past the end of the list, the new list is empty. If the length reaches past the end of the list, the new list uses the remaining elements in the list.

```cpp
code

sublist()

Syntax
sublist(list li, int start, [int length]);

Parameters
- list li - the list to be searched
- int start - the offset in the list for the beginning of the new list
- int length - the number of elements of the first list to use in the second list

Return Value
list - the sublist
```
substr() function

The substr() function creates a new string from another string, starting with the character specified at the offset start, and continuing for the number of characters specified for length.

When using the substr() function, remember that the first element in the list is always zero.

If a length is not included, the new string starts at the offset and uses the remaining characters in the old string.

<table>
<thead>
<tr>
<th>substr()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>string <code>src</code> - the string to be examined</td>
</tr>
<tr>
<td>int <code>start</code> - the offset in the string for the beginning of the new string</td>
</tr>
<tr>
<td>int <code>length</code> - the number of characters of the first string to use in the second string</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

succe() function

The succ() function finds and returns the logical successor of the indexed item in a passed array.

The succ() function allows foreach-like behavior, but at a larger performance cost.

<table>
<thead>
<tr>
<th>succ()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>array <code>arr</code> - the array from which to extract the successor</td>
</tr>
<tr>
<td>any <code>index</code> - the index to the item whose successor will be extracted</td>
</tr>
</tbody>
</table>
### succ()

<table>
<thead>
<tr>
<th>Return Value</th>
<th>any - the successor of the indexed item in the array (that is, the item that will be extracted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>array[0] = &quot;zero&quot;; array[1] = &quot;one&quot;; array[2] = &quot;two&quot;; succ(array, 0) == 1 succ(array, 1) == 2 succ(array, 2) == NULL succ(array, 500) == NULL</td>
</tr>
</tbody>
</table>

#### suffix()

The suffix() function compares two strings and returns true if the second string is a suffix of the first string. Use the casesuffix() function for a case-insensitive comparison.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>suffix(string big, string little);</th>
</tr>
</thead>
</table>
| Parameters   | • string big - the string against which little will be compared  
              • string little - the smaller string, which will be compared to the end of big |
| Return Value | int, where:  
              • 1 - little is the same as the end of big  
              • 0 - little is not the same as the end of big |
| Example      | suffix("ABCDEFG", "EFG") == 1  
              suffix("ABCDEFG", "eFg") == 0  
              suffix("ABCDEFG", "ABC") == 0 |

#### tolower()

The tolower() function converts all English alphabetic characters from uppercase to lowercase. Other characters, including alphabetic characters in other character sets, are not converted.

The tolower() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The tolower() function may be impacted by the change in the behavior of string variables. See str for additional information about this behavior.

The overall return string for the tolower() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>tolower(string in);</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>• string in - the string to be converted to lowercase</td>
</tr>
</tbody>
</table>
### tolower()

**Return Value**
string - the same string that was passed in, with all uppercase characters (A-Z) converted to lowercase characters (a-z).

**Example**
```
tolower("AB12fj4") == "ab12fj4"
tolower("ABCdefG") == "abcdefg"
$foo = tolower($foo); # FAST
```

### toupper()

The toupper() function converts all English alphabetic characters from lowercase to uppercase. Other characters, including alphabetic characters in other character sets, are not converted.

The toupper() function has a pre-optimized fastpath, where str is its most relevant parameter. It operates more efficiently if its input string is the same CPcode variable as its output string.

The toupper() function may be impacted by the change in the behavior of string variables. Refer to the description of the string data type on page B-5 for additional information about this behavior.

The overall return string for the toupper() function cannot be larger than 8KB. You should avoid passing in an input string larger than 8KB.

**Syntax**
toupper(string in);

**Parameters**
- string in - the string to be converted to uppercase

**Return Value**
string - the same string that was passed in, with all lowercase characters (a-z) converted to uppercase characters (A-Z).

**Example**
```
toupper("AB12fj4") == "AB12FJ4"
toupper("ABCdefG") == "ABCDEFG"
$foo = toupper($foo); # FAST
```

### typeof()

The typeof() function returns the data type of an expression.

You can use the typeof() function for error checking or for conditional operations. However, there is a performance penalty associated with this usage.

**Syntax**
typeof(any argument);

**Parameters**
- any argument - the variable whose data type will be returned
typeof()

**Return Value**
string - a string indicating the data type of the argument, where:

- alert - the argument is an alert
- array - the argument is an array
- ethmac - the argument is an ethmac address
- filter tag - the argument is filter tag
- function pointer - the argument is a function pointer
- error - the argument is an error code
- int - the argument is an integer
- ipv4host - the argument is an IPv4 host address
- ipv4net - the argument is an IPv4 network address
- ipv6host - the argument is an IPv6 host address
- ipv6net - the argument is an IPv6 network address
- list - the argument is a list
- recorder - the argument is a recorder
- str - the argument is a string, a blob, or a pattern
- unknown - a processing error occurred

**Example**
```javascript
if (typeof($1) != "str")
    return 0;
```

ubyte()
The ubyte() function returns a one-byte unsigned integer from the specified offset within a string.

<table>
<thead>
<tr>
<th>ubyte()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>ubyte(string str, int offset);</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>- string str - the string to be examined, beginning at byte 0</td>
</tr>
<tr>
<td>- int offset - the offset in the string for the byte to be returned</td>
</tr>
<tr>
<td><strong>Return Value</strong></td>
</tr>
<tr>
<td>- int - the offset integer, as an unsigned value</td>
</tr>
<tr>
<td>- error - the offset is out of bounds</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>#the eighth byte is the unsigned type code</td>
</tr>
<tr>
<td>$code = ubyte( $struct_data, 7 );</td>
</tr>
</tbody>
</table>

ulong()
The ulong() function returns a signed four-byte integer from a specified offset within a string. It converts the value from network byte order in the string to the host’s byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The ulong() function is useful for extracting values from packets.

The ulong() function returns the same result as the long() function. It is provided for backwards compatibility. The internal representation of all CPcode integers is signed. Therefore, the ulong() function yields a signed integer.
ulong()

Syntax
ulong(string str, int offset);

Parameters
- string str - the string to be examined, beginning at byte 0
- int offset - the offset in the string for the first byte to be returned

Return Value
- int - the offset integer, as a signed value
- error - the offset is out of bounds

Example
# Structure starts with a 4-byte integer
# in network byte order.
# Converted to host byte order
# before placing it in $data
$data = ulong($struct_data, 0);
# Caveat
$blob = cat(-1);
echo(ulong($blob, 0));        # -1
echo(ulong($blob, 0) >> 2);   # -1
$blob = cat(-4);
echo(ulong($blob, 0) >> 1);   # -2
# Be _very_ careful with
# shifting signed integers.
# Their sign bit is extended.

unpack()

The unpack() function extracts the individual variables from a structure that was packed using the pack() function.

Syntax
int unpack(list schema, string packed, var1, var2, ...

Parameters
- list schema - the result of a call to the packing_schema() function
- string packed - a packed structure
- var1, var2, varn - the variables in which to store data elements from the unpacked structure

Return Value
int - !NULL if the unpacking was successful
**unpack()**

**Example**

```plaintext
{
    schema = packing_schema("ip", "ip", "port");
    ...
    $struct = pack(schema, $src_ip, $dst_ip, $dst_port);
    if (ALERTS[$struct] == NULL)
        ALERTS[$struct] = 1;
    else
        ALERTS[$struct] = ALERTS[$struct] + 1;
    ...
    unpack(schema, $struct, $src, $dst, $dport);
    msg = cat(ALERTS[$struct], " attacks from ", $src, " against ", $dst, ":", $dport);
    echo( $msg);
}
```

**url_flush()**

The `url_flush()` function removes a specified URL from cache.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the `url_*` functions.

**Syntax**

```plaintext
url_flush( URL )
```

**Parameters**


**Return Value**

None

**url_needs_refresh()**

The `url_needs_refresh()` function indicates whether or not the specified URL needs to be updated because it has been modified and/or the cached representation is stale.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the `url_*` functions.

**Syntax**

```plaintext
int url_needs_refresh ( URL )
```

**Parameters**


**Return Value**

Int – an integer with one of the following possible values:

- 0 – the cached URL is up to date.
- 1 – the cached URL is stale.
- 2 – the URL is not cached.
url_open()

The url_open() function returns a handle to the resource which can be used with the I/O builtins (io_*) to retrieve data. The user must call io_close on the handle to release it.

If the cached representation is stale and the resource has been modified, the cached representation will be refreshed before the handle is returned.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.

```
url_open()
Syntax
int url_open ( URL, <username>, <password> )
Parameters
• <username> - optional parameter supporting basic HTTP authentication,
• <password> - optional parameter supporting basic HTTP authentication.
Return Value
int – Returns a handle to the cached resource if successful, otherwise NULL if the not successful.
```

url_readall()

The url_readall() function returns all of the contents of the specified URL.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.

```
url_readall()
Syntax
string url_readall ( URL, <username>, <password> )
Parameters
• <username> - optional parameter supporting basic HTTP authentication,
• <password> - optional parameter supporting basic HTTP authentication.
Return Value
string - A string containing the contents of the specified URL.
```

url_readlines()

The url_readlines() function returns the contents of the specified URL as a list of strings with each entry containing a single line. Line terminators are not removed.

Cache control parameters are inherited by the CPcode runtime from the calling application for use with the url_* functions.
url_readlines()

Syntax
list url_readlines ( URL, <username>, <password> )

Parameters
- <username> - optional parameter supporting basic HTTP authentication,
- <password> - optional parameter supporting basic HTTP authentication.

Return Value
list - A list of strings. Each entry is a single line from the URL.

ushort()

The ushort() function returns a two-byte value as an unsigned integer from a specified offset within a string. It converts the value from network byte order in the string to the host's byte order. Network byte order is big endian; therefore, the most significant byte is stored at the lowest address in memory.

The ushort() function is useful for extracting values from packets.

ushort()

Syntax
ushort(string str, int offset);

Parameters
- string str - the string to be examined, beginning byte 0
- int offset - the offset in the string for the first byte to be returned

Return Value
- int - the offset integer, as an unsigned value
- error - the offset is out of bounds

Example
# Structure starts with a 2-byte integer
# in network byte order.
# Converted to host byte order
# before placing it in $data
$data = ushort($struct_data, 0);