HelixCore

Helix Core P4Perl Developer Guide

2020.1

October 2020

PERFORCE

www.perforce.com

Contents

How to use this guide **5**

Syntax conventions 5

Feedback 5

Other documentation 6

P4Perl **7**

System Requirements and Release Notes 7

Installing P4Perl 7

Programming with P4Perl 8

Connect to Helix Core server 8

Connect to Helix server over SSL 9

Converting forms between formats 9

P4Perl Classes 10

P4 11

P4::DepotFile 15

P4::Revision 15

P4::Integration 16

P4::Map 16

P4::MergeData 17

P4::Message 17

P4::OutputHandler 18

P4::Progress 18

P4::Resolver 18

P4::Spec 19

Class P4 19

Class P4::DepotFile 32

Class P4::Revision 32

Class P4::Integration 34

Class P4::Map 34

Class P4::MergeData 37

Class P4::Message 39

Class P4::OutputHandler 40

Class P4::Progress 41

Class P4::Resolver 42

Class P4::Spec 43

Glossary 45
License Statements 64

How to use this guide

This guide contains details about using the derived API for Perl to create scripts that interact with Helix Core server. You can download the API from the Perforce web site. The derived API depends on the Helix C/C++ API. For details, see the *Helix Core C/C++ Developer Guide*.

This section provides information on typographical conventions, feedback options, and additional documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.
`...`	Previous argument can be repeated. ■ `p4 [g-opts] streamlog [ -l -L -t -m max ]` ***`stream1`*** `...` means `1` or more stream arguments separated by a space ■ See also the use on `...` in Command alias syntax in the *Helix Core P4 Command Reference* **Tip** `...` has a different meaning for directories. See Wildcards in the *Helix Core P4 Command Reference*.
element1 | *element2*	Either *element1* or *element2* is required.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

> **Tip**
> You can also search for Support articles in the Perforce Knowledgebase.

Earlier versions of this guide: 2018.2

To find even earlier versions of this guide, use the following URL and replace *v16.1* with the version number you are looking for: https://www.perforce.com/manuals/*v16.1*/p4perl/index.html

P4Perl

P4Perl is a Perl module that provides an object-oriented API to Helix Core server. Using P4Perl is faster than using the command-line interface in scripts, because multiple command can be executed on a single connection, and because it returns Helix server responses as Perl hashes and arrays.

The main features are:

- Get Helix server data and forms in hashes and arrays.
- Edit Helix server forms by modifying hashes.
- Run as many commands on a connection as required.
- The output of commands is returned as a Perl array.
- The elements of the array returned are strings or, where appropriate, hash references.

System Requirements and Release Notes

P4Perl is supported on Windows, Linux, and OS X.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4perlnotes.txt.

> **Note**
> When passing arguments, make sure to omit the space between the argument and its value, such as in the value pair `-u` and ***`username`*** in the following example:
>
> ```
> change = p4.run_changes("-uusername", "-m1")[0]
> ```
>
> If you include a space (`"-u username"`), the command fails.

Installing P4Perl

You can download P4Perl from the Perforce web site at https://www.perforce.com/downloads/helix-core-api-perl.

After downloading, you can either run the installer or build the interface from source, as described in the Release notes.

Programming with P4Perl

Connect to Helix Core server

The following example shows how to connect to a Helix Core server, run a **`p4 info`** command, and open a file for edit:

```
#!/opt/local/bin/perl -w
use strict;
use P4;

my $p4 = new P4;
$p4->SetClient('bruno_ws');
$p4->SetUser('smoon');
$p4->SetPort('localhost:20081');
$p4->SetVersion("EnvTest 1.0");
$p4->Connect() or die("Was not able to connect\n");

my $info = $p4->Run("info"); #passing array ref
print "\n\nP4 Info Output:\n\n";
foreach my $akey (@{$info}) {
  my @infos = keys %$akey; # $akey is hash ref
  foreach my $hkey (@infos) {
    print "$hkey => $akey->{$hkey}\n";
  }
}

my $client_name = $p4->FetchClient($p4->GetClient());
print "\n\nClient Specification:\n\n";
foreach my $chkey (keys %{$client_name}) {
  if ($client_name->{$chkey} =~ /^ARRAY(.+)$/) {
    my $avals = $client_name->{$chkey};
    foreach my $achkey (@{$avals}) {
      print "$chkey => $achkey\n";
    }
  } elsif($client_name->{$chkey} =~ /^HASH(.+)$/) {
    my $hvals = $client_name->{$chkey};
    foreach my $hchkey (keys %{$hvals}) {
```

```
      print "$chkey => $hvals->{$hchkey}\n";
    }
  } else {
    print "$chkey => $client_name->{$chkey}\n";
  }
}

my $changes = $p4->Run("changes","-m2");
print "\n\nTwo Most Recent Changes:\n\n";
foreach my $each_chg (@{$changes}) {
  my @chg_key = keys %$each_chg; # $each_chg is hash ref
  foreach my $hchg (@chg_key) {
    print "$hchg => $each_chg->{$hchg}\n";
  }
  print "\n";
}
print "\n" . $p4->GetVersion() . "\n";
$p4->Disconnect();
```

Connect to Helix server over SSL

Scripts written with P4Perl use any existing `P4TRUST` file present in their operating environment (by default, `.p4trust` in the home directory of the user that runs the script).

If the fingerprint returned by the server fails to match the one installed in the `P4TRUST` file associated with the script's run-time environment, your script will (and should!) fail to connect to the server.

Converting forms between formats

Sometimes you have a form in a hash format, and want it in a string. Sometimes you have a string, and want a hash. In these situations, the following methods will help.

`FormatSpec( $type, $hash )`

Convert Perforce form hash to string

Converts a Perforce form of the specified type (client/label, and so on) held in the supplied hash into its string representation.

> **Note**
> Shortcut methods are available that obviate the need to supply the type argument.

The following two examples are equivalent:

```
$string = $p4->FormatSpec( "client", $hash );
$string = $p4->FormatClient( $hash );
```

See below for more information on the abbreviated form.

`Format<type>( $hash )` is shorthand for `$p4->FormatSpec( <type>, $hash )`

For example:

```
$change = $p4->FetchChange();
$change->{ 'Description' } = 'Some description';
$form   = $p4->FormatChange( $change );
printf( "Submitting this change:\n\n%s\n", $form );
$p4->RunSubmit( $change );
```

Convert Perforce form string to hash

Converts a Perforce form of the specified type (client/label, and so on) in the supplied string into a hash and returns a reference to that hash.

> **Note**
> Shortcut methods are available to avoid the need to supply the type argument.

The following two examples are equivalent:

```
$hash = $p4->ParseSpec( "client", $string );
$hash = $p4->ParseClient( $clientspec );
```

See below for more information on the abbreviated form.

`Parse<type>( $string )` is shorthand for `$p4->ParseSpec( <type>, $string )`

For example:

```
$hash = $p4->ParseClient( $string );
$hash = $p4->ParseLabel( $string );
$hash = $p4->ParseBranch( $string );
$hash = $p4->ParseProtect( $string );
```

P4Perl Classes

The P4 module consists of several public classes:

- "P4" below
- "P4::DepotFile" on page 15
- "P4::Revision" on page 15
- "P4::Integration" on page 16
- "P4::Map" on page 16
- "P4::MergeData" on page 17
- "P4::Message" on page 17
- "P4::OutputHandler" on page 18
- "P4::Progress" on page 18
- "P4::Spec" on page 19

The following tables provide brief details about each public class.

P4

The main class used for executing Helix server commands. Almost everything you do with P4Perl will involve this class.

Method	Description
`new()`	Construct a new **`P4`** object.
`Identify()`	Print build information including P4Perl version and Helix C/C++ API version.
`ClearHandler()`	Clear the output handler.
`Connect()`	Initialize the Helix server client and connect to the Server.
`Disconnect()`	Disconnect from the Helix Core server.
`ErrorCount()`	Returns the number of errors encountered during execution of the last command.
`Errors()`	Returns a list of the error strings received during execution of the last command.
`Fetch_<Spectype>()`	Shorthand for running: `$p4->Run( "<spectype>", "-o" );`

Method	Description
Format_<Spectype>_()	Shorthand for running: `$p4->FormatSpec( "<spectype>", hash );`
FormatSpec()	Converts a Helix server form of the specified type (client/label etc.) held in the supplied hash into its string representation.
GetApiLevel()	Get current API compatibility level.
GetCharset()	Get character set when connecting to Unicode servers.
GetClient()	Get current client workspace (**P4CLIENT**).
GetCwd()	Get current working directory.
GetEnv()	Get the value of a Helix server environment variable, taking into account **P4CONFIG** files and (on Windows or OS X) the registry or user preferences.
GetHandler()	Get the output handler.
GetHost()	Get the current hostname.
GetMaxLockTime()	Get **MaxLockTime** used for all following commands.
GetMaxResults()	Get **MaxResults** used for all following commands.
GetMaxScanRows()	Get **MaxScanRows** used for all following commands.
GetPassword()	Get the current password or ticket.
GetPort()	Get host and port (**P4PORT**).
GetProg()	Get the program name as shown by the **p4 monitor show -e** command.
GetProgress()	Get the progress indicator.
GetTicketFile()	Get the location of the **P4TICKETS** file.
GetUser()	Get the current username (**P4USER**).
GetVersion()	Get the version of your script, as reported to the Helix Core server.

Method	Description
IsConnected()	Test whether or not session has been connected and/or has been dropped.
IsStreams()	Test whether or not streams are enabled.
IsTagged()	Test whether or not tagged output is enabled.
IsTrack()	Test whether or not server performance tracking is enabled.
Iterate_<*Spectype*>()	Iterate through spec results.
Messages()	Return an array of **P4::Message** objects, one for each message sent by the server.
P4ConfigFile()	Get the location of the configuration file used (**P4CONFIG**).
Parse_<*Spectype*>()	Shorthand for running: `$p4->ParseSpec( "<spectype>", buffer );`
ParseSpec()	Converts a Helix server form of the specified type (**client**, **label**, etc.) held in the supplied string into a hash and returns a reference to that hash.
Run*Cmd*()	Shorthand for running: `$p4->Run( "cmd", arg, ... );`
Run()	Run a Helix server command and return its results. Check for errors with **P4::ErrorCount()**.
RunFilelog()	Runs a **p4 filelog** on the **fileSpec** provided and returns an array of **P4::DepotFile** objects.
RunLogin()	Runs **p4 login** using a password or ticket set by the user.
RunPassword()	A thin wrapper for changing your password.
RunResolve()	Interface to **p4 resolve**.
RunSubmit()	Submit a changelist to the server.
RunTickets()	Get a list of tickets from the local tickets file.

Method	Description

Save_<*Spectype*>()	Shorthand for running:
	`$p4->SetInput( $spectype );`
	`$p4->Run( "<spectype>", "-i" );`

ServerCaseSensitive()	Returns an integer specifying whether or not the server is case-sensitive.
ServerLevel()	Returns an integer specifying the server protocol level.
ServerUnicode()	Returns an integer specifying whether or not the server is in Unicode mode.
SetApiLevel()	Specify the API compatibility level to use for this script.
SetCharset()	Set character set when connecting to Unicode servers.
SetClient()	Set current client workspace (**P4CLIENT**).
SetCwd()	Set current working directory.
SetEnv()	On Windows or OS X, set an environment variable in the registry or user preferences.
SetHandler()	Set the output handler.
SetHost()	Set the name of the current host (**P4HOST**).
SetInput()	Save the supplied argument as input to be supplied to a subsequent command.
SetMaxLockTime()	Set **MaxLockTime** used for all following commands.
SetMaxResults()	Set **MaxResults** used for all following commands.
SetMaxScanRows()	Set **MaxScanRows** used for all following commands.
SetPassword()	Set Helix server password (**P4PASSWD**).
SetPort()	Set host and port (**P4PORT**).
SetProg()	Set the program name as shown by the **p4 monitor show -e** command.
SetProgress()	Set the progress indicator.
SetStreams()	Enable or disable streams support.

Method	Description
`SetTicketFile()`	Set the location of the `P4TICKETS` file.
`SetTrack()`	Activate or deactivate server performance tracking. By default, tracking is off (0).
`SetUser()`	Set the Helix server username (`P4USER`).
`SetVersion()`	Set the version of your script, as reported to the Helix Core server.
`Tagged()`	Toggles tagged output (1 or 0). By default, tagged output is on (1).
`TrackOutput()`	If performance tracking is enabled with `SetTrack()` returns an array of strings with tracking output.
`WarningCount()`	Returns the number of warnings issued by the last command.
`Warnings()`	Returns a list of the warning strings received during execution of the last command.

P4::DepotFile

Utility class allowing access to the attributes of a file in the depot. Returned by `P4::RunFilelog()`.

Method	Description
`DepotFile()`	Name of the depot file to which this object refers.
`Revisions()`	Returns an array of revision objects for the depot file.

P4::Revision

Utility class allowing access to the attributes of a revision of a file in the depot. Returned by `P4::RunFilelog()`.

Method	Description
`Action()`	Returns the action that created the revision.
`Change()`	Returns the changelist number that gave rise to this revision of the file.
`Client()`	Returns the name of the client from which this revision was submitted.
`DepotFile()`	Returns the name of the depot file to which this object refers.

Method	Description
`Desc()`	Returns the description of the change which created this revision.
`Digest()`	Returns the MD5 digest for this revision.
`FileSize()`	Returns the size of this revision.
`Integrations()`	Returns an array of `P4::Integration` objects representing all integration records for this revision.
`Rev()`	Returns the number of this revision.
`Time()`	Returns date/time this revision was created.
`Type()`	Returns the Helix server filetype of this revision.
`User()`	Returns the name of the user who created this revision.

P4::Integration

Utility class allowing access to the attributes of an integration record for a revision of a file in the depot. Returned by `P4::RunFilelog()`.

Method	Description
`How()`	Integration method (merge/branch/copy/ignored).
`File()`	Integrated file.
`SRev()`	Start revision.
`ERev()`	End revision.

P4::Map

A class that allows users to create and work with Helix server mappings without requiring a connection to the Helix Core server.

Method	Description
`New()`	Construct a new Map object (class method).
`Join()`	Joins two maps to create a third (class method).
`Clear()`	Empties a map.
`Count()`	Returns the number of entries in a map.

Method	Description
`IsEmpty()`	Tests whether or not a map object is empty.
`Insert()`	Inserts an entry into the map.
`Translate()`	Translate a string through a map.
`Includes()`	Tests whether a path is mapped.
`Reverse()`	Returns a new mapping with the left and right sides reversed.
`Lhs()`	Returns the left side as an array.
`Rhs()`	Returns the right side as an array.
`AsArray()`	Returns the map as an array.

P4::MergeData

Class encapsulating the context of an individual merge during execution of a **`p4 resolve`** command. Passed to `P4::RunResolve`.

Method	Description
`YourName()`	Returns the name of "your" file in the merge. (file in workspace)
`TheirName()`	Returns the name of "their" file in the merge. (file in the depot)
`BaseName()`	Returns the name of "base" file in the merge. (file in the depot)
`YourPath()`	Returns the path of "your" file in the merge. (file in workspace)
`TheirPath()`	Returns the path of "their" file in the merge. (temporary file on workstation into which `TheirName()` has been loaded)
`BasePath()`	Returns the path of the base file in the merge. (temporary file on workstation into which `BaseName()` has been loaded)
`ResultPath()`	Returns the path to the merge result. (temporary file on workstation into which the automatic merge performed by the server has been loaded)
`MergeHint()`	Returns hint from server as to how user might best resolve merge.
`RunMergeTool()`	If the environment variable `P4MERGE` is defined, run it and indicate whether or not the merge tool successfully executed.

P4::Message

Class encapsulating the context of an individual error during execution of Helix server commands. Passed to `P4::Messages()`.

Method	Description
`GetSeverity()`	Returns the severity class of the error.
`GetGeneric()`	Returns the generic class of the error message.
`GetId()`	Returns the unique ID of the error message.
`GetText()`	Get the text of the error message.

P4::OutputHandler

Handler class that provides access to streaming output from the server; call `P4::SetHandler()` with an implementation of `P4::OutputHandler` to enable callbacks:

Method	Description
`OutputBinary()`	Process binary data.
`OutputInfo()`	Process tabular data.
`OutputMessage()`	Process information or errors.
`OutputStat()`	Process tagged output.
`OutputText()`	Process text data.

P4::Progress

Handler class that provides access to progress indicators from the server; call `P4::SetProgress()` with an implementation of `P4::Progress` to enable callbacks:

Method	Description
`Init()`	Initialize progress indicator as designated type.
`Total()`	Total number of units (if known).
`Description()`	Description and type of units to be used for progress reporting.
`Update()`	If non-zero, user has requested a cancellation of the operation.
`Done()`	If non-zero, operation has failed.

P4::Resolver

Class for handling resolves in Helix server.

Method	Description
`Resolve()`	Perform a resolve and return the resolve decision as a string.

P4::Spec

Utility class allowing access to the attributes of the fields in a Helix server form.

Method	Description
`_fieldname()`	Return the value associated with the field named *fieldname*.
`_fieldname()`	Set the value associated with the field named *fieldname*.
`PermittedFields()`	Lists the fields that are permitted for specs of this type.

Class P4

Description

Main interface to the Helix server client API.

This module provides an object-oriented interface to Helix server, the Perforce version control system. Data is returned in Perl arrays and hashes and input can also be supplied in these formats.

Each **`P4`** object represents a connection to the Helix Core server, also called Helix server, and multiple commands may be executed (serially) over a single connection.

The basic model is to:

1. Instantiate your **`P4`** object.
2. Specify your Helix server client environment.
   - `SetClient()`
   - `SetHost()`
   - `SetPassword()`
   - `SetPort()`
   - `SetUser()`
3. Connect to the Perforce service.

   The Helix server protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4Perl) should ensure that a separate connection is used for each thread, or that only one thread may use a shared connection at a time.

4. Run your Helix server commands.
5. Disconnect from the Perforce service.

Class methods

P4::new() -> P4

Construct a new **P4** object. For example:

```
my $p4 = new P4;
```

P4::Identify() -> string

Print build information including P4Perl version and Helix C/C++ API version.

```
print P4::Identify();
```

The constants **OS**, **PATCHLEVEL** and **VERSION** are also available to test an installation of P4Perl without having to parse the output of **P4::Identify()**. Also reports the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4Perl was built.

P4::ClearHandler() -> undef

Clear any configured output handler.

P4::Connect() -> bool

Initializes the Helix server client and connects to the server. Returns **false** on failure and **true** on success.

P4::Disconnect() -> undef

Terminate the connection and clean up. Should be called before exiting.

P4::ErrorCount() -> integer

Returns the number of errors encountered during execution of the last command.

P4::Errors() -> list

Returns a list of the error strings received during execution of the last command.

P4::Fetch<Spectype>([name]) -> hashref

Shorthand for running:

```
$p4->Run( "<spectype>", "-o" );
```

and returning the first element of the result array. For example:

```
$label      = $p4->FetchLabel( $labelname );
$change     = $p4->FetchChange( $changeno );
$clientspec = $p4->FetchClient( $clientname );
```

P4::Format<Spectype>(hash) -> string

Shorthand for running:

```
$p4->FormatSpec( "<spectype>", hash);
```

and returning the results. For example:

```
$change     = $p4->FetchChange();
$change->{ 'Description' } = 'Some description';
$form       = $p4->FormatChange( $change );
printf( "Submitting this change:\n\n%s\n", $form );
$p4->RunSubmit( $change );
```

P4::FormatSpec($spectype, $string) -> string

Converts a Helix server form of the specified type (**`client`**, **`label`**, etc.) held in the supplied hash into its string representation. Shortcut methods are available that obviate the need to supply the type argument. The following two examples are equivalent:

```
my $client = $p4->FormatSpec( "client", $hash );
```

```
my $client = $p4->FormatClient( $hash );
```

P4::GetApiLevel() -> integer

Returns the current API compatibility level. Each iteration of the Helix Core server is given a level number. As part of the initial communication, the client protocol level is passed between client application and the Helix Core server. This value, defined in the Helix C/C++ API, determines the communication protocol level that the Helix server client will understand. All subsequent responses from the Helix Core server can be tailored to meet the requirements of that client protocol level.

For more information about the client protocol levels, see the Support Knowledgebase article, "Helix Client Protocol Levels".

P4::GetCharset() -> string

Return the name of the current charset in use. Applicable only when used withHelix servers running in unicode mode.

P4::GetClient() -> string

Returns the current Helix server client name. This may have previously been set by `P4::SetClient()`, or may be taken from the environment or `P4CONFIG` file if any. If all that fails, it will be your hostname.

P4::GetCwd() -> string

Returns the current working directory as your Helix server client sees it.

P4::GetEnv($var) -> string

Returns the value of a Helix server environment variable, taking into account the settings of Helix server variables in `P4CONFIG` files, and, on Windows or OS X, in the registry or user preferences.

P4::GetHandler() -> Handler

Returns the output handler.

P4::GetHost() -> string

Returns the client hostname. Defaults to your hostname, but can be overridden with `P4::SetHost()`

P4::GetMaxLockTime($value) -> integer

Get the current `maxlocktime` setting.

P4::GetMaxResults($value) -> integer

Get the current `maxresults` setting.

P4::GetMaxScanRows($value) -> integer

Get the current `maxscanrows` setting.

P4::GetPassword() -> string

Returns your Helix server password. Taken from a previous call to `P4::SetPassword()` or extracted from the environment (`$ENV{P4PASSWD}`), or a `P4CONFIG` file.

P4::GetPort() -> string

Returns the current address for your Helix server server. Taken from a previous call to `P4::SetPort()`, or from `$ENV{P4PORT}` or a `P4CONFIG` file.

P4::GetProg() -> string

Get the name of the program as reported to the Helix Core server.

P4::GetProgress() -> Progress

Returns the progress indicator.

P4::GetTicketFile([$string]) -> string

Return the path of the current **`P4TICKETS`** file.

P4::GetUser() -> String

Get the current user name. Taken from a previous call to **`P4::SetUser()`**, or from **`$ENV{P4USER}`** or a **`P4CONFIG`** file.

P4::GetVersion($string) -> string

Get the version of your script, as reported to the Helix Core server.

P4::IsConnected() -> bool

Returns true if the session has been connected, and has not been dropped.

P4::IsStreams() -> bool

Returns true if streams support is enabled on this server.

P4::IsTagged() -> bool

Returns true if Tagged mode is enabled on this client.

P4::IsTrack() -> bool

Returns true if server performance tracking is enabled for this connection.

P4::Iterate<Spectype>(arguments) -> object

Iterate over spec results. Returns an iterable object with **`next()`** and **`hasNext()`** methods.

Valid <*spectype*>s are **`clients`**, **`labels`**, **`branches`**, **`changes`**, **`streams`**, **`jobs`**, **`users`**, **`groups`**, **`depots`** and **`servers`**. Valid arguments are any arguments that would be valid for the corresponding **`P4::Run`*`Cmd`*`()`** command.

Arguments can be passed to the iterator to filter the results, for example, to iterate over only the first two client workspace specifications:

```
$p4->IterateClients( "-m2" );
```

You can also pass the spec type as an argument:

```
$p4->Iterate( "changes" );
```

For example, to iterate through client specs:

```
use P4;

my $p4 = P4->new;
$p4->Connect or die "Couldn't connect";
my $i = $p4->IterateClients();
while($i->hasNext) {
  my $spec = $i->next;
  print( "Client: " . ($spec->{Client} or "<undef>") . "\n" );
}
```

P4::Messages() -> list

Returns an array of **`P4::Message()`** objects, one for each message (info, warning or error) sent by the server.

P4::P4ConfigFile() -> string

Get the path to the current **`P4CONFIG`** file.

P4::Parse<Spectype>($string) -> hashref

Shorthand for running:

```
$p4-ParseSpec( "<spectype>", buffer);
```

and returning the results. For example:

```
$p4 = new P4;
$p4->Connect() or die( "Failed to connect to server" );
$client = $p4->FetchClient();

# Returns a string (equivalent to running 'p4 client -o' from the command-
line)
$client = $p4->FormatClient( $client );

# Convert to a hashref
$client = $p4->ParseClient( $client );
# Convert back to string
$client = $p4->FormatClient( $client );
```

Comments in forms are preserved. Comments are stored as a **`comment`** key in the spec hash and are accessible. For example:

```
my $spec = $pc->ParseGroup( 'my_group' );
print $spec->{'comment'};
```

P4::ParseSpec($spectype, $string) -> hashref

Converts a Helix server form of the specified type (client/label etc.) held in the supplied string into a hash and returns a reference to that hash. Shortcut methods are available to avoid the need to supply the type argument. The following two examples are equivalent:

```
my $hash = $p4->ParseSpec( "client", $clientspec );
```

```
my $hash = $p4->ParseClient( $clientspec );
```

> **Important**
> Custom specifications require that you call **`Fetch`** first so that the **`specDefs`** can be determined by the API and later used by **`ParseSpec`**.

P4::Run<Cmd>([$arg...]) -> list | arrayref

Shorthand for running:

```
$p4-Run( "cmd", arg, ...);
```

and returning the results.

P4::Run("<cmd>", [$arg...]) -> list | arrayref

Run a Helix server command and return its results. Because Helix server commands can partially succeed and partially fail, it is good practice to check for errors using **`P4::ErrorCount()`**.

Results are returned as follows:

- A list of results in array context
- An array reference in scalar context

The AutoLoader enables you to treat Helix server commands as methods:

```
p4->RunEdit( "filename.txt" );
```

is equivalent to:

```
$p4->Run( "edit", "filename.txt" );
```

Note that the content of the array of results you get depends on (a) whether you're using tagged mode, (b) the command you've executed, (c) the arguments you supplied, and (d) your Helix server version.

Tagged mode and form parsing mode are turned on by default; each result element is a hashref, but this is dependent on the command you ran and your server version.

In non-tagged mode, each result element is a string. In this case, because the Helix server sometimes asks the client to write a blank line between result elements, some of these result elements can be empty.

Note that the return values of individual Helix server commands are not documented because they may vary between server releases.

To correlate the results returned by the P4 interface with those sent to the command line client, try running your command with RPC tracing enabled. For example:

Tagged mode: **`p4 -Ztag -vrpc=1 describe -s 4321`**

Non-Tagged mode: **`p4 -vrpc=1 describe -s 4321`**

Pay attention to the calls to **`client-FstatInfo()`**, **`client-OutputText()`**, **`client-OutputData()`** and **`client-HandleError()`**. Each call to one of these functions results in either a result element, or an error element.

P4::RunFilelog([$args ...], $fileSpec ...) -> list | arrayref

Runs a **`p4 filelog`** on the **`fileSpec`** provided and returns an array of **`P4::DepotFile`** objects when executed in tagged mode.

P4::RunLogin(...) -> list | arrayref

Runs **`p4 login`** using a password or ticket set by the user.

P4::RunPassword($oldpass, $newpass) -> list | arrayref

A thin wrapper for changing your password from **`$oldpass`** to **`$newpass`**. Not to be confused with **`P4::SetPassword()`**.

P4::RunResolve([$resolver], [$args ...]) -> string

Run a **`p4 resolve`** command. Interactive resolves require the **`$resolver`** parameter to be an object of a class derived from **`P4::Resolver`**. In these cases, the **`P4::Resolve()`** method of this class is called to handle the resolve. For example:

```
$resolver = new MyResolver;
$p4->RunResolve( $resolver );
```

To perform an automated merge that skips whenever conflicts are detected:

```
use P4;


package MyResolver;
our @ISA = qw( P4::Resolver );
```

```
sub Resolve( $ ) {
  my $self      = shift;
  my $mergeData = shift;

  # "s"kip if server-recommended hint is to "e"dit the file,
  # because such a recommendation implies the existence of a conflict
  return "s" if ( $mergeData->MergeHint() eq "e" );
  return $mergeData->MergeHint();
}
1;

package main;

$p4 = new P4;
$resolver = new MyResolver;

$p4->Connect() or die( "Failed to connect to Perforce" );
$p4->RunResolve( $resolver, ... );
```

In non-interactive resolves, no **`P4::Resolver`** object is required. For example:

```
$p4->RunResolve( "at" );
```

P4::RunSubmit($arg | $hashref, ...) -> list | arrayref

Submit a changelist to the server. To submit a changelist, set the fields of the changelist as required and supply any flags:

```
$change = $p4->FetchChange();
$change->{ 'Description' } = "Some description";
$p4->RunSubmit( "-r", $change );
```

You can also submit a changelist by supplying the arguments as you would on the command line:

```
$p4->RunSubmit( "-d", "Some description", "somedir/..." );
```

P4::RunTickets() -> list

Get a list of tickets from the local tickets file. Each ticket is a hash object with fields for **`Host`**, **`User`**, and **`Ticket`**.

P4::Save<Spectype>() -> list | arrayref

Shorthand for running:

```
$p4->SetInput( $spectype );
$p4->Run( "<spectype>", "-i");
```

For example:

```
$p4->SaveLabel( $label );
$p4->SaveChange( $changeno );
$p4->SaveClient( $clientspec );
```

P4::ServerCaseSensitive() -> integer

Returns an integer specifying whether or not the server is case-sensitive.

P4::ServerLevel() -> integer

Returns an integer specifying the server protocol level. This is not the same as, but is closely aligned to, the server version. To find out your server's protocol level, run **`p4 -vrpc=5 info`** and look for the **`server2`** protocol variable in the output.

For more information about the Helix server version levels, see the Support Knowledgebase article, "Helix server Version Levels".

P4::ServerUnicode() -> integer

Returns an integer specifying whether or not the server is in Unicode mode.

P4::SetApiLevel($integer) -> undef

Specify the API compatibility level to use for this script. This is useful when you want your script to continue to work on newer server versions, even if the new server adds tagged output to previously unsupported commands.

The additional tagged output support can change the server's output, and confound your scripts. Setting the API level to a specific value allows you to lock the output to an older format, thus increasing the compatibility of your script.

Must be called before calling **`P4::Connect()`**. For example:

```
$p4->SetApiLevel( 67 ); # Lock to 2010.1 format
$p4->Connect() or die( "Failed to connect to Perforce" );
# etc.
```

P4::SetCharset($charset) -> undef

Specify the character set to use for local files when used with aHelix server running in unicode mode. Do not use unless yourHelix serveris in unicode mode. Must be called before calling **`P4::Connect()`**. For example:

```
$p4->SetCharset( "winansi" );
$p4->SetCharset( "iso8859-1" );
$p4->SetCharset( "utf8" );
# etc.
```

P4::SetClient($client) -> undef

Sets the name of your Helix server client workspace. If you don't call this method, then the client workspace name will default according to the normal Helix server conventions:

1. Value from file specified by **`P4CONFIG`**
2. Value from **`$ENV{P4CLIENT}`**
3. Hostname

P4::SetCwd($path) -> undef

Sets the current working directory for the client.

P4::SetEnv($var, $value) -> undef

On Windows or OS X, set a variable in the registry or user preferences. To unset a variable, pass an empty string as the second argument. On other platforms, an exception is raised.

```
$p4->SetEnv( "P4CLIENT", "my_workspace" );
$P4->SetEnv( "P4CLIENT", "");
```

P4::SetHandler(Handler) -> Handler

Sets the output handler.

P4::SetHost($hostname) -> undef

Sets the name of the client host, overriding the actual hostname. This is equivalent to **`p4 -H hostname`**, and only useful when you want to run commands as if you were on another machine.

P4::SetInput($string | $hashref | $arrayref) -> undef

Save the supplied argument as input to be supplied to a subsequent command. The input may be a hashref, a scalar string, or an array of hashrefs or scalar strings. If you pass an array, the array will be shifted once each time the Helix server command being executed asks for user input.

P4::SetMaxLockTime($integer) -> undef

Limit the amount of time (in milliseconds) spent during data scans to prevent the server from locking tables for too long. Commands that take longer than the limit will be aborted. The limit remains in force until you disable it by setting it to zero. See **`p4 help maxresults`** for information on the commands that support this limit.

P4::SetMaxResults($integer) -> undef

Limit the number of results for subsequent commands to the value specified. Helix server will abort the command if continuing would produce more than this number of results. Once set, this limit remains in force unless you remove the restriction by setting it to a value of 0.

P4::SetMaxScanRows($integer) -> undef

Limit the number of records Helix server will scan when processing subsequent commands to the value specified. Helix server will abort the command once this number of records has been scanned. Once set, this limit remains in force unless you remove the restriction by setting it to a value of 0.

P4::SetPassword($password) -> undef

Specify the password to use when authenticating this user against the Helix Core server - overrides all defaults. Not to be confused with **`P4::Password()`**.

P4::SetPort($port) -> undef

Set the port on which your Helix server is listening. Defaults to:

1. Value from file specified by **`P4CONFIG`**
2. Value from **`$ENV{P4PORT}`**
3. **`perforce:1666`**

P4::SetProg($program_name) -> undef

Set the name of your script. This value is displayed in the server log on 2004.2 or later servers.

P4::SetProgress(Progress) -> Progress

Sets the progress indicator.

P4::SetStreams(0 | 1) -> undef

Enable or disable support for streams. By default, streams support is enabled at 2011.1 or higher (`P4::GetApiLevel()` >= 70). Streams support requires a server at 2011.1 or higher. You can enable or disable support for streams both before and after connecting to the server.

P4::SetTicketFile([$string]) -> string

Set the path to the current `P4TICKETS` file (and return it).

P4::SetTrack(0 | 1) -> undef

Enable (1) or disable (0) server performance tracking for this connection. By default, performance tracking is disabled.

P4::SetUser($username) -> undef

Set your Helix server username. Defaults to:

1. Value from file specified by `P4CONFIG`
2. Value from `C<$ENV{P4USER}>`
3. OS username

P4::SetVersion($version) -> undef

Specify the version of your script, as recorded in the Helix server log file.

P4::Tagged(0 | 1 | $coderef) -> undef

Enable (1) or disable (0) tagged output from the server, or temporarily toggle it.

By default, tagged output is enabled, but can be disabled (or re-enabled) by calling this method. If you provide a code reference, you can run a subroutine with the tagged status toggled for the duration of that reference. For example:

```
my $GetChangeCounter = sub{ $p4->RunCounter('change')->[ 0 ] };
my $changeno = $p4->Tagged( 0, $GetChangeCounter );
```

When running in tagged mode, responses from commands that support tagged output will be returned in the form of a hashref. When running in non-tagged mode, responses from commands are returned in the form of strings (that is, in plain text).

P4::TrackOutput() -> list

If performance tracking is enabled with `P4::SetTrack()`, returns a list of strings corresponding to the performance tracking output of the most recently-executed command.

P4::WarningCount() -> integer

Returns the number of warnings issued by the last command.

```
$p4->WarningCount();
```

P4::Warnings() -> list

Returns a list of warning strings from the last command

```
$p4->Warnings();
```

Class P4::DepotFile

Description

`P4::DepotFile` objects are used to present information about files in the Helix server repository. They are returned by **`P4::RunFilelog()`**.

Class Methods

None.

Instance Methods

$df->DepotFile() -> string

Returns the name of the depot file to which this object refers.

$df->Revisions() -> array

Returns an array of **`P4::Revision`** objects, one for each revision of the depot file.

Class P4::Revision

Description

`P4::Revision` objects are represent individual revisions of files in the Helix server repository. They are returned as part of the output of **`P4::RunFilelog()`**.

Class Methods

$rev->Integrations() -> array

Returns an array of **`P4::Integration`** objects representing all integration records for this revision.

Instance Methods

$rev->Action() -> string

Returns the name of the action which gave rise to this revision of the file.

$rev->Change() -> integer

Returns the changelist number that gave rise to this revision of the file.

$rev->Client() -> string

Returns the name of the client from which this revision was submitted.

$rev->DepotFile() -> string

Returns the name of the depot file to which this object refers.

$rev->Desc() -> string

Returns the description of the change which created this revision. Note that only the first 31 characters are returned unless you use **`p4 filelog -L`** for the first 250 characters, or **`p4 filelog -l`** for the full text.

$rev->Digest() -> string

Returns the MD5 digest for this revision.

$rev->FileSize() -> string

Returns the size of this revision.

$rev->Rev() -> integer

Returns the number of this revision of the file.

$rev->Time() -> string

Returns the date/time that this revision was created.

$rev->Type() -> string

Returns this revision's Helix server filetype.

$rev->User() -> string

Returns the name of the user who created this revision.

Class P4::Integration

Description

`P4::Integration` objects represent Helix server integration records. They are returned as part of the output of **`P4::RunFilelog()`**.

Class Methods

None.

Instance Methods

$integ->How() -> string

Returns the type of the integration record - how that record was created.

$integ->File() -> string

Returns the path to the file being integrated to/from.

$integ->SRev() -> integer

Returns the start revision number used for this integration.

$integ->ERev() -> integer

Returns the end revision number used for this integration.

Class P4::Map

Description

The **`P4::Map`** class allows users to create and work with Helix server mappings, without requiring a connection to a Helix server.

Class Methods

$map = new P4::Map([array]) -> aMap

Constructs a new **P4::Map** object.

$map->Join(map1, map2) -> aMap

Join two **P4::Map** objects and create a third.

The new map is composed of the left-hand side of the first mapping, as joined to the right-hand side of the second mapping. For example:

```
# Map depot syntax to client syntax
$client_map = new P4::Map;
$client_map->Insert( "//depot/main/...", "//client/..." );

# Map client syntax to local syntax
$client_root = new P4::Map;
$client_root->Insert( "//client/...", "/home/bruno/workspace/..." );

# Join the previous mappings to map depot syntax to local syntax
$local_map = P4::Map::Join( $client_map, $client_root );
$local_path = $local_map->Translate( "//depot/main/www/index.html" );

# $local_path is now /home/bruno/workspace/www/index.html
```

Instance Methods

$map->Clear() -> undef

Empty a map.

$map->Count() -> integer

Return the number of entries in a map.

$map->IsEmpty() -> bool

Test whether a map object is empty.

$map->Insert(string ...) -> undef

Inserts an entry into the map.

May be called with one or two arguments. If called with one argument, the string is assumed to be a string containing either a half-map, or a string containing both halves of the mapping. In this form, mappings with embedded spaces must be quoted. If called with two arguments, each argument is assumed to be half of the mapping, and quotes are optional.

```
# called with two arguments:
$map->Insert( "//depot/main/...", "//client/..." );

# called with one argument containing both halves of the mapping:
$map->Insert( "//depot/live/... //client/live/..." );

# called with one argument containing a half-map:
# This call produces the mapping "depot/... depot/..."
$map->Insert( "depot/..." );
```

$map->Translate(string, [bool]) -> string

Translate a string through a map, and return the result. If the optional second argument is 1, translate forward, and if it is 0, translate in the reverse direction. By default, translation is in the forward direction.

$map->Includes(string) -> bool

Tests whether a path is mapped or not.

```
if ( $map->Includes( "//depot/main/..." ) ) {
  ...
}
```

$map->Reverse() -> aMap

Return a new **P4::Map** object with the left and right sides of the mapping swapped. The original object is unchanged.

$map->Lhs() -> array

Returns the left side of a mapping as an array.

$map->Rhs() -> array

Returns the right side of a mapping as an array.

$map->AsArray() -> array

Returns the map as an array.

Class P4::MergeData

Description

Class containing the context for an individual merge during execution of a **`p4 resolve`**. Users may not create objects of this class; they are created internally during **`P4::RunResolve()`**, and passed down to the **`Resolve()`** method of a **`P4::Resolver`** subclass.

Class Methods

None.

Instance Methods

$md.YourName() -> string

Returns the name of "your" file in the merge, in client syntax.

$md.TheirName() -> string

Returns the name of "their" file in the merge, in client syntax, including the revision number.

$md.BaseName() -> string

Returns the name of the "base" file in the merge, in depot syntax, including the revision number.

$md.YourPath() -> string

Returns the path of "your" file in the merge. This is typically a path to a file in the client workspace.

$md.TheirPath() -> string

Returns the path of "their" file in the merge. This is typically a path to a temporary file on your local machine in which the contents of **`P4::MergeData::TheirName()`** have been loaded.

$md.BasePath() -> string

Returns the path of the base file in the merge. This is typically a path to a temporary file on your local machine in which the contents of **`P4::MergeData::BaseName()`** have been loaded.

$md.ResultPath() -> string

Returns the path to the merge result. This is typically a path to a temporary file on your local machine in which the contents of the automatic merge performed by the server have been loaded.

$md.MergeHint() -> string

Returns a string containing the hint from Helix server's merge algorithm, indicating the recommended action for performing the resolve.

$md.RunMergeTool() -> integer

If the environment variable **`P4MERGE`** is defined, **`P4::MergeData::RunMergeTool()`** invokes the specified program and returns true if the merge tool was successfully executed, otherwise returns false.

$md.MergeType() -> string

Returns a string describing the merge type, such as **`Branch resolve`**.

$md.YourAction() -> string

Returns the name of "your" action, such as **`ignore`**.

$md.TheirAction() -> string

Returns the name of "their" action, such as **`branch`**.

$md.MergeAction() -> string

Returns the name of the action used in the merge. For example, if **`TheirAction`** is **`branch`** and **`YourAction`** is **`ignore`**, then if you choose yours, you get an ignore, and if you choose theirs, you get a branch.

$md.MergeInfo() -> string

Returns an object containing details about the resolve. For example:

```
'clientFile' => '/Users/jdoe/Workspaces/main.p4-
perl/test/resolve/action/file-88.txt',
'fromFile' => '//depot/projA/src/file-88.txt',
'startFromRev' => 'none',
'resolveType' => 'branch',
'resolveFlag' => 'b',
'endFromRev' => '2'
```

Class P4::Message

Description

`P4::Message` objects contain error or other diagnostic messages from the Helix Core server; they are returned by `P4::Messages()`.

Script writers can test the severity of the messages in order to determine if the server message consisted of command output (`E_INFO`), warnings, (`E_WARN`), or errors (`E_FAILED`/`E_FATAL`).

Class methods

None.

Instance methods

$message.GetSeverity() -> int

Severity of the message, which is one of the following values:

Value	Meaning
`E_EMPTY`	No error.
`E_INFO`	Informational message only.
`E_WARN`	Warning message only.
`E_FAILED`	Command failed.
`E_FATAL`	Severe error; cannot continue.

$message.GetGeneric() -> int

Returns the generic class of the error.

$message.GetId() -> int

Returns the unique ID of the message.

$message.GetText() -> int

Converts the message into a string.

Class P4::OutputHandler

Description

The `P4::OutputHandler` class provides access to streaming output from the server. After defining the output handler, call `P4::SetHandler()` with your implementation of `P4::OutputHandler`.

Because P4Perl does not provide a template or superclass, your output handler must implement all five of the following methods: `OutputMessage()`, `OutputText()`, `OutputInfo()`, `OutputBinary()`, and `OutputStat()`, even if the implementation consists of trivially returning `0` (report only: don't handle output, don't cancel operation).

These methods must return one of the following four values:

Value	Meaning
`0`	Messages added to output (don't handle, don't cancel).
`1`	Output is handled by class (don't add message to output).
`2`	Operation is marked for cancel, message is added to output.
`3`	Operation is marked for cancel, message not added to output.

Class Methods

None.

Instance Methods

$handler.OutputBinary() -> int

Process binary data.

$handler.OutputInfo() -> int

Process tabular data.

$handler.OutputMessage() -> int

Process informational or error messages.

$handler.OutputStat()-> int

Process tagged data.

$handler.OutputText() -> int

Process text data.

Class P4::Progress

Description

The `P4::Progress` provides access to progress indicators from the server. After defining the progress class, call `P4::SetProgress()` with your implementation of `P4::Progress`.

Because P4Perl does not provide a template or superclass, you must implement all five of the following methods: `Init()`, `Description()`, `Update()`, `Total()`, and `Done()`, even if the implementation consists of trivially returning `0`.

Class Methods

None.

Instance Methods

$progress.Init() -> int

Initialize progress indicator.

$progress.Description(string, int) -> int

Description and type of units to be used for progress reporting.

$progress.Update() -> int

If non-zero, user has requested a cancellation of the operation.

$progress.Total()-> int

Total number of units expected (if known).

$progress.Done() -> int

If non-zero, operation has failed.

Class P4::Resolver

Description

`P4::Resolver` is a class for handling resolves in Helix server. It is intended to be subclassed, and for subclasses to override the **`Resolve()`** method. When **`P4::RunResolve()`** is called with a **`P4::Resolver`** object, it calls the **`P4::Resolver::Resolve()`** method of the object once for each scheduled resolve.

Class Methods

$actionResolve() -> string

Enables support for resolves of branches, deletes, and file types. This method is invoked if an action resolve is required. It lets you add a callback in your Resolver implementation to determine what the resolve action should be after the automatic resolver has evaluated it. This is similar to resolves in P4V, when you are prompted to select what you want to do, given what the automatic resolver suggested. The **`$resolver->ActionResolve()`** method receives an argument (**`mergeData`**) and lets you return a string that specifies what to do. See $resolver.Resolve() for available strings.

The following example counts the number of times it has been called (line 5), stores the **`mergeData`** from the autoresolver (**`type`**, **`hint`**, and **`info`**) and returns what the automatic resolver suggested (**`hint`**) on line 10 as the answer.

```
sub ActionResolve( $ ) {
    my $self      = shift;
    my $mergeData = shift;

    $self->{'ActionResolve'} += 1;
    $self->{'type'} = $mergeData->Type();
    $self->{'hint'} = $mergeData->MergeHint();
    $self->{'info'} = $mergeData->MergeInfo();

    return $mergeData->MergeHint();
}
```

Instance Methods

$resolver.Resolve() -> string

Returns the resolve decision as a string. The standard Helix server resolve strings apply:

String	Meaning
`ay`	Accept Yours.
`at`	Accept Theirs.
`am`	Accept Merge result.
`ae`	Accept Edited result.
`s`	Skip this merge.
`q`	Abort the merge.

By default, all automatic merges are accepted, and all merges with conflicts are skipped. The `P4::Resolver::Resolve()` method is called with a single parameter, which is a reference to a `P4::MergeData` object.

Class P4::Spec

Description

`P4::Spec` objects provide easy access to the attributes of the fields in a Helix server form.

The `P4::Spec` class uses Perl's AutoLoader to simplify form manipulation. Form fields can be accessed by calling a method with the same name as the field prefixed by an underscore (_).

Class Methods

$spec = new P4::Spec($fieldMap) -> array

Constructs a new `P4::Spec` object for a form containing the specified fields. (The object also contains a ***`fields`*** member that stores a list of field names that are valid in forms of this type.)

Instance Methods

$spec->_<fieldname> -> string

Returns the value associated with the field named <*fieldname*>.

```
$client = $p4->FetchClient( $clientname );
$client->_Root();    # Get client root
```

$spec->_<fieldname>($string)-> string

Updates the value of the named field in the spec.

```
$client = $p4->FetchClient( $clientname );
$client->_Root( $newroot );    # Set client root
```

$spec->PermittedFields() -> array

Returns an array containing the names of fields that are valid in this spec object. This does not imply that values for all of these fields are actually set in this object, merely that you may choose to set values for any of these fields if you want to.

```
my $client = $p4->FetchClient( $clientname );
my @permitted = $client->PermittedFields();
foreach $field (@permitted) {
  print "$field\n";
}
```

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (versioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

For files: The file revision, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision. For checked out streams: The public have version from which the checked out version is derived.

binary file type

A Helix server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix server and is able to run scripts on the commands before sending them to the Helix server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction and changelist number.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix server depot.

check out

To designate one or more files, or a stream, for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix server files that is not streams-based. Uses the Perforce file revision model, not the graph model. The default depot name is depot. See also default depot, stream depot, and graph depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Core server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files, or stream versions, are compared. A conflict is a pair of unequal diffs between each of two files and a base, or between two versions of a stream. (verb) To compare the contents of files or file revisions, or of stream versions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

extension

Similar to a trigger, but more modern. See "Helix Core Server Administrator Guide" on "Extensions".

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicate servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix server. See also Helix4Git and classic depot.

group

A feature in Helix server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

heartbeat

A process that allows one server to monitor another server, such as a standby server monitoring the master server (see the p4 heartbeat command).

Helix server

The Helix server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core server.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix server depots.

hybrid workspace

A workspace that maps to files stored in a depot of the classic Perforce file revision model as well as to files stored in a repo of the graph model associated with git.

I

iconv

A PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix server. The job template determines what information is tracked. The template can be modified by the Helix server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job daemon

A program that checks the Helix server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

A form describing the fields and possible values for each job stored in the Helix server machine.

job view

A syntax used for searching Helix server jobs.

journal

A file containing a record of every change made to the Helix server's metadata since the time of the last checkpoint. This file grows as each Helix server transaction is logged. The file should be automatically truncated and renamed into a numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix server's metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix server from two conflicting file revisions.

metadata

The data stored by the Helix server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata is stored in the Perforce database and is separate from the archive files that users submit.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix server has assigned a number.

O

opened file

A file you have checked out in your client workspace as a result of a Helix Core server operation (such as an edit, add, delete, integrate). Opening a file from your operating system file browser is not tracked by Helix Core server.

owner

The Helix server user who created a particular client, branch, or label.

P

p4

1. The Helix Core server command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix server; p4d manages depot files and metadata.

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

Perforce

Perforce Software, Inc., a leading provider of enterprise-scale software solutions to technology developers and development operations (“DevOps”) teams requiring productivity, visibility, and scale during all phases of the development lifecycle.

project

In Helix Swarm, a group of Helix server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix server’s protections table.

proxy server

A Helix server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix server but not make any changes.

remote depot

A depot located on another Helix server accessed by the current Helix server.

replica

A Helix server that contains a full or partial copy of metadata from a master Helix server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file, or two versions of a stream. You can choose to resolve file conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes. To resolve stream conflicts, you can choose to accept the public source, accept the checked out target, manually accept changes, or combine path fields of the public and checked out version while accepting all other changes made in the checked out version.

reverse delta storage

The method that Helix server uses to store revisions of text files. Helix server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

review daemon

A program that periodically checks the Helix server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files included in those changelists, informing them of changes in files they are interested in.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

RPM

RPM Package Manager. A tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

The combination of server metadata (the Helix server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Core server, the shared versioning service that responds to requests from Helix server client applications. The Helix server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

storage record

An entry within the db.storage table to track references to an archive file.

stream

A "branch" with built-in rules that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients. Has structured branching, unlike the free-form branching of a "classic" depot. Uses the Perforce file revision model, not the graph model. See also classic depot and graph depot.

submit

To send a pending changelist into the Helix server depot for processing.

super access

An access level that gives the user permission to run every Helix server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script that is automatically invoked by Helix server when various conditions are met. (See "Helix Core Server Administrator Guide" on "Triggers".)

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix server in which you assign file types to files.

U

user

The identifier that Helix server uses to determine who is performing an operation. The three types of users are standard, service, and operator.

V

versioned file

Source files stored in the Helix server depot, including one or more revisions. Also known as an archive file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

Perforce programs include software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

Perforce Software, Inc. includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (http://zookeeper.apache.org/)

Perforce programs include software developed by the OpenLDAP Foundation (http://www.openldap.org/).

Perforce programs include software developed Computing Services at Carnegie Mellon University: Cyrus SASL (http://www.cmu.edu/computing/).