
NAME

ots-build - bash library to build OTS packages

SYNTAX

```
# load library
if ots_build=$(pkg-config --libs ots-build); then
    . "$ots_build"
else
    echo >&2 "unable to load ots-build library"
    exit 1
fi

# override some default variables
otsb_set var1 val1
otsb_set var2 val2

# initialize the library
otsb_init

# parse the command line
otsb_options "$@"

# create build variables
otsb_set_build_variables

otsb_run_cmds
```

DESCRIPTION

Sometimes the Off-The-Shelf (OTS) software you need has a complicated build workflow. If you want to manage that in a scripted (patch)-config-build-test-install environment, that usually means building a script to perform all of the work. If you are working with dozens of OTS package, that can be a lot of redundant work.

ots-build is a **bash** library which provides support for common tasks in building OTS software. You'll still have to write a driver script to perform the build, but it should be much simpler.

It provides a framework for

- parsing command line arguments
- running partial or complete builds and installation
- unarchiving packages and patching the source prior to configuration.
- a rudimentary logging mechanism to keep things tidy.

ots-build tries to figure out some things about the package to be built, such as name and version, which is useful when installing into staging areas (such as used by the **graft** or **stow** commands). It does this using the name of the source directory (see the **--srcdir** option).

USING THE LIBRARY

ots-build consists of a single file containing **bash** functions.

driver scripts use those functions to implement the build process.

Typically a site would create a wrapper around **ots-build** to customize defaults or add extra functionality required for the site.

ots-build requires **bash** 4.3 or newer.

Assumptions

ots-build makes a few assumptions about where package archives and patches are located. These can be overridden with command line arguments or hooks. Two directories are special:

The directory containing the build script

By default **ots-build** assumes that the build script, the package archive, and an optional patch are located in the same directory. These can be overridden with the `--archive` and `--patchfile` options

The directory containing the unarchive package source.

ots-build deduces the package name and version from the name of the directory containing the unarchived source (which is usually something like *package-x.y.z*). It uses this to create the default patch file name.

This information can be fairly useful to the driver scripts, for instance in specifying version specific compiler or installation flags.

The example autoconf driver script uses it to offer support for installing packages into staged areas for use by the **graft** or **stow** commands.

Note: By default **ots-build** assumes that the *current directory* contains the source. If this is not the case, the `--srcdir` option must be specified.

Sharing information with the library

In order to simplify the interface, much of the information needed and provided by **ots-build** is shared via **bash** variables. Variables associated with particular tasks (for instance parsing the command line) are described in the documentation for those tasks.

ots-build is designed so that it does not override variables which have already been set. This is accomplished by using the **otsb_set** and **otsb_setarr** functions to modify variable values. These functions will only set a value if the variable has not yet been set; variables with values will not be changed. If you need to discard an existing value so that a default can be recomputed, use **otsb_reset** or **otsbv_reset** first.

The only deviation from this behavior is when parsing command line options; these always override preset variables.

Variables which are special to **ots-build** use the **otsbv_set** command.

Driver Scripts

Driver scripts are **bash** scripts which use **ots-build** to perform build operations. They must be invoked with at least one argument specifying which command to perform. See *Commands* for the commands provided by **ots-build**. The driver script specifies which commands are legal and the code to perform them.

The typical sequence of invocation of **ots-build** functions is

- 1 Load the library. This should be done at *the very start* of the build script. **ots-build** provides a **pkg-config** metadata file:

```
if ots_build=$(pkg-config --libs ots-build); then
. $ots_build
else
echo >&2 "unable to load ots-build library"
exit 1
fi
```
- 2 Initialize the library via

```
otsb_init
```

This sets up the internal globals, hook queues, and built-in option metadata.
- 3 Indicate the available required commands and their order via the `otsb_commands` and `otsb_opt_commands` arrays.
- 4 Define actions for commands by creating functions for them, e.g. `otsb_cmd_build`.
- 5 Add command line options to be recognized with `otsb_add_option`.
- 6 Parse the command line with `otsb_options`
- 7 Set up the variables which control the build by running

```
otsb_set_build_variables
```
- 8 Perform requested actions using **otsb_run_cmds**

Workflow Customization

Customization is done by providing code which will be run by **ots-build** functions at particular points (hooks) in their execution sequences. The code can be used to specify default values for parameters, add command line options, create new build variables, or whatever might be needed to support local build requirements.

ots-build maintains an execution queue for each hook location. The **otsb_hook_push_fn_front** and **otsb_hook_push_fn_back** functions add function callbacks to the front or back of the queue, while **otsb_hook_push_front** and **otsb_hook_push_back** add shell snippets. Function callbacks are invoked directly; shell snippets are evaluated as code when the hook runs. The latter approach is deprecated. For `*_before` hooks, function callbacks are passed the arguments given to the function which invoked the hook. Workflow command hooks are created automatically when commands are registered.

Hooks for the administrative commands are:

`init_before`

The code is run at the start of **otsb_init**.

`init_after`

The code is run at the end of **otsb_init**.

`options_before`

The code is run at the start of **otsb_options**.

`options_after`

The code is run at the end of **otsb_options**.

build_variables_before

The code is run at the start of **otsb_set_build_variables**.

build_variables_after

The code is run at the end of **otsb_set_build_variables**.

*_before and *_after hooks for each workflow command (e.g. patch, build, etc) are automatically generated.

Wrapping ots-build

Rather than encode site-specific options or defaults into driver scripts, it's easier to centralize the information in a wrapper around **ots-build**. The driver scripts should load the wrapper library instead of **ots-build**.

For example, let's say that a site wants to specify a default value for the **--prefix** option, add an additional command line option, and set some site-specific build variables. Here's one way of doing this:

```
#!/bin/bash

# don't load this multiple times
[[ ${_mst_otsb_loaded:=0} ]] && return 0
_mst_otsb_loaded=1

# load ots-build
if ots_build=$(pkg-config --libs ots-build); then
    . $ots_build
else
    echo >&2 "unable to load ots-build library"
    exit 1
fi

create_new_options() {

    # On Solaris, prefer GNU patch & tar
    # create defaults here so that help information can show the
    # default values

    case `uname -s` in

        SunOS )
            otsb_set make $(command -v gmake)
            otsb_set tar $(command -v gtar)
            otsbv_set patch-cmd /usr/local/bin/patch
            ;;

        * )
            otsb_set make $(command -v make)
            otsb_set tar $(command -v tar)
            ;;
```

```

esac

otsb_set ots_root /soft/ots

otsb_add_option scalar ots-root \
"top-level directory containing installed OTS [$ots_root]"

otsb_add_option scalar ots-pkgs \
    "directory into which OTS packages will be installed
[$ots-root/pkgs]"

otsb_add_option scalar make      "name of the make command [$make]"
otsb_add_option scalar tar       "name of the tar  command [$tar]"

otsb_add_option array  config-opts 'extra options for configure; may be
specified multiple times'
}

default_option_values () {
    # can't set this until after otsb_options is run,
    # as it depends upon the value of the ots-root option
    otsb_set ots_pkgs ${ots_root}/pkgs
}

otsb_hook_push_fn_back options_before create_new_options
otsb_hook_push_fn_back options_after  default_option_values

```

Command Line Options

ots-build provides an extensible parser for command line options, with an initial set of options, such as *--help*. The driver script can create additional options using *otsb_add_option* prior to the call to *otsb_options*.

The *otsb_options* function is parses command line options and arguments and sets up bash variables with their values. It takes a list of tokens to parse, which is typically just those passed on the command line to the script:

```
otsb_options "$@"
```

Option format

Only long options (preceded by *--*) are recognized. Options which take values may be separated from their values by a *=* character or white space. For example:

```
--exec-prefix=a --prefix b --flag
```

Hyphens in option names may be specified as underscores.

Command line options are either boolean (presence signifies true or false), scalars (take a single value) or arrays (may be specified multiple times, appending each value to an array). Boolean options may be specified either in the affirmative (*--flag*) or in the negative (*--no-flag*).

Provided options

The following options are provided out-of-the-box:

`--prefix string`

`--exec-prefix string`

These are the standard prefixes used by most software installers

`--debug`

`--log`

`--logconsole`

`--logpfx string`

`--logsingle`

See *Logging*

`--help`

Outputs help information and exits

`--version`

Outputs the version of **ots-build** and exits.

`--srcdir srcdir`

This specifies the directory containing the source. **ots-build** will change to this directory before performing any actions. It defaults to the current directory.

`--archive filename`

The file name of the archive containing the source. If the **unpack** command is specified the archive is unpacked and **ots-build** will change into the archive directory. Currently only compressed tar files are handled. **ots-build** can handle archives which don't unpack into a single top-level directory. See *Commands* for more information on what happens after an unpack operation.

If the filename is specified with a directory, it is assumed to be in the directory containing the driver script.

`--patch-cmd string`

The patch command to use. It defaults to the **patch** command in the users path.

`--patch-opts string`

Options to pass to the patch command in addition to the patch file. This is an array option, and it may be repeated. It defaults to `-p0 -N -s`.

Default values

Default values for options created via *otsb_add_option* may be set with *otsb_set* before calling *otsb_options*. Option parsing is the only context where values specified via *otsb_set* are overridden.

Retrieving option values

After parsing, the values of the options are stored in bash variables with the same name as the option (with any hyphens converted to underscores). Boolean values are represented as the string `true` for true and `false` for false. Options which are not specified on the command line are not set to any default value.

In addition to the options that you provide, **ots-build** provides some of its own. These are stored in

variables with a special prefix, `otsbv_` to prevent confusion. To specify defaults, use the `otsbv_set` command prior to calling `otsb_init`. For example, the `--logsingle` option value will be stored in `otsbv_logsingle`. To set its default value:

```
otsbv_set logsingle true
```

Option help

ots-build supports a `--help` option.

Options added via calls to `otsb_add_option` will use the help string provided in those calls.

Options provided by **ots-build** have pre-defined help strings which can be overridden by calling the `otsb_help` function after calling `otsb_init` and before calling `otsb_options`.

Arguments

The only arguments accepted on the command line are the names of commands. **ots-build** provides some built-in commands (see *Commands* for more information).

The names of commands that the driver script provides should be placed in one of two arrays:

`otsb_commands`

This array should contain all of the commands which are part of the normal build workflow. **ots-build** provides an `all` command which will run all of these in the order specified in `otsb_commands`.

This array defaults to

```
otsb_commands=( patch configure build test install )
```

`otsb_opt_commands`

This array should contain any commands which are not part of the normal build workflow. They are *not* run by the `all` command.

The actual requested commands are made available in the `otsb_req_cmds` array after the command line is parsed. See *Commands* for more information on how to interface build commands to the auto-invocation code.

Setting build variables

After invoking `otsb_options` and before running commands, build variables must be set by calling `otsb_set_build_variables`. **otsb_set_build_variables** assumes that the current directory is the source directory. This is automatically performed by **otsb_options** if the `--srcdir` option is specified on the command line.

The `prefix` variable must already be defined, which is typically set via the `--prefix` command line option.

The following variables are set; they may be overridden prior to invoking `otsb_set_build_variables` using `otsb_set`.

`otsbv_import_root`

This is a directory which may contain things associated with the package, such as a patch file. By default this is extracted from the name of the build script (under the assumption that all package related things are in the same directory as the build script). It may also be set with the **--import-root** option.

package

This is the name of the package (including version information). This is by default extracted from the name of the *current directory* using the `otsb_pkg` function (use the `--srcdir` to specify an alternative directory).

package_name

The name of the package (without any version information). This is by default derived from the `package` variable using the `otsb_pkg_name` function. An alternative to overriding the variable is to redefine the `otsb_pkg_name` function.

package_version

The version of the package. This is by default derived from the `package` variable using the `otsb_pkg_version` function. An alternative to overriding the variable is to redefine the `otsb_pkg_name` function.

patchfile

The name of the patch file (if any). This is by default set to
`${package}.patch`

exec_prefix

This will default to the value of `prefix` if it not already set.

bindir

Defaults to `${exec_prefix}/bin`

libdir

Defaults to `${exec_prefix}/lib`

incdir

Defaults to `${prefix}/include/${package}`

mandir

Defaults to `${prefix}/man`

docdir

Defaults to `${prefix}/share/doc/${package}`

Commands

The main purpose of **ots-build** is to run commands. The commands to run are specified on the command line (see *Arguments*). The commands which will always be available are:

all

Run all of the commands in the `otsb_commands` array, in order specified.

dump

Output the command line arguments.

unpack

Unpack the source archive and change into the containing directory. The `--archive` option must have been specified.

Because the unpacking is done after the `otsb_set_build_variables` function has executed, the

values of variables which depend upon the current directory being the build directory will be incorrect.

Therefore, after unpacking the driver script is automatically executed again with the **--srcdir** option set to the directory containing the unpacked source.

show-cmds

Output a list of the non-administrative (i.e. actual build workflow) commands to stdout.

ots-build provides the `otsb_run_cmds` function which will execute the commands specified on the command line. A command is implemented as a bash function of the same name, prefixed with `otsb_cmd_`. For example, the *install* command might be written as:

```
otsb_cmd_install () {  
    ... commands to perform the actual install ...  
}
```

It is important that these functions either exit or return with non-zero values if an error occurred (see *Errors* for other means of signalling an error). These functions are invoked in subshells, to isolate them from the rest of **ots-build**.

Stub functions are provided for the `configure`, `build`, `test`, and `install` commands. These will print an error message and exit with a non-zero status (to remind you that you haven't implemented them properly). You must provide replacement versions of these.

The `patch` command is more fully implemented. See *Patching* for more information.

Patching

ots-build provides a default **patch** command for patching a source distribution before it is configured. This can be changed by redefining the `otsb_cmd_patch` function. Patching is dependent upon the existence of the patch file specified via the `patchfile` global variable.

A working version of **patch** is required (Solaris' version is by definition broken). By default the version of **patch** in the user's path is used; an alternative may be specified with the **--patch-cmd** option.

The **patch** command is passed the options specified via the **--patch-opts** option.

If the **applypatch** command is available it is used to drive **patch**, e.g.

```
applypatch -patch "patch -p0 -N -s" $patchfile
```

Logging

If logging is turned on via the **--log** or **--logsingle** options, the output of each command is logged.

--log logs each command to a separate file, *pfxcmd.log*, where *pfx* is the prefix specified by **--logpfx**. This defaults to `otsb_`, so that it does not interfere with any log files created by the package (such as *config.log*, etc.).

--logsingle sends the output from all commands to a single file, *pfx.log*, where *pfx* is the value of the **--logpfx** option, with any trailing underscore removed.

When logging is turned on, by default no output is sent to the terminal. To log to the terminal as well as to files, specify **--logconsole**.

The **--debug**, **--log**, **--logconsole**, **--logsingle**, and **--logpfx** options may also be set via the environment variables `OTS_BUILD_DEBUG`, `OTS_BUILD_LOG`, `OTS_BUILD_LOGCONSOLE`,

OTS_BUILD_LOGSINGLE, and OTS_BUILD_LOGPFX. Command line settings override the environment.

Defaults for the variables associated with these options may set via the `otsbv_set` function *before* calling `otsb_init`.

Errors

If a command function returns a non-zero value **ots-build** will exit with an error. Typically it is up to the writer of that function to ensure that an error will cause a non-zero return. That can be a pain. **ots-build** provides a simple interface to the **bash** error trapping system which will ensure that if any command executed within a command function exits with an error it will be caught and **ots-build** will be notified.

The `otsb_trap_err` command is used to manage the trap.

To engage or disengage the error trap for all command functions, use

```
otsb_trap_err functions on
otsb_trap_err functions off
```

This sets the default state when a command function is invoked. If this is invoked within a command function it will change the state immediately.

To engage or disengage the error trap immediately

```
otsb_trap_err on
otsb_trap_err off
```

These turn the system on and off immediately. If invoked within a command function, the trap state will return to *off* after the function returns.

To reset the state to that specified for command functions

```
otsb_trap_err functions reset
```

This is only valid within a command function and resets the state to the default state specified by the last `otsb_trap_err functions` setting.

FUNCTIONS

`otsb_add_option`

```
otsb_add_option $type $name "$help"
```

Add an option of the given type (scalar, bool, array), name, and help string.

`otsb_add_dumpvar`

```
otsb_add_dumpvar $var_name
```

Add the *named* variable to the list of variables output by the dump command. Option variables are automatically added to the list.

`otsb_append_path`

```
otsb_append_path $path_var_name $path1 $path2 ...
```

Append the specified path(s) to the path variable with the given name. A path variable's value is a colon separated list of paths. For example,

```
foo=a:b:c:d
otsb_append_path foo e f g h
```

results in

```
foo=a:b:c:d:e:f:g:h
```

`otsb_assert_has_command`

```
otsb_assert_has_command $cmd1 $cmd2
```

Checks if the passed commands are in the user's path. If not it exits with an error message via **otsb_die**.

`otsb_has_command`

```
otsb_has_command $cmd
```

Returns true if the named command is available in the user's path.

`otsb_check_logfile`

```
otsb_check_logfile logfile
```

Ensures that the named logfile can be written. If the file does not exist it is created and removed again; if it already exists, write permission is checked.

`otsb_cleanup`

This function is called when the script exits and performs any necessary cleanup. By default it does nothing. You should redefine it if you need it.

`otsb_die`

```
otsb_die $message
```

Print the message to the standard error stream using a standard format and exit with a non-zero status code.

`otsb_warn`

```
otsb_warn $message
```

Print the message to the standard error stream using the standard library message format and continue.

`otsb_error`

```
otsb_error $message
```

Output the message to the standard error stream using a standard format.

`otsb_source`

```
otsb_source file1 [file2 ...]
```

Source the named files once and remember that they have been loaded. Repeated calls for the same file are ignored.

`otsb_help`

```
otsb_help cmd      cmd_name "cmd help string"
otsb_help option  option_name "option helpstring"
```

Change the help string of an existing option or command.

The second argument is either `cmd` or `option` indicating what element the help string is for. For example, here's how the default help string for the `--exec-prefix` option is specified:

```
otsb_help option exec-prefix \
    "install architecture-dependent files here"
```

`otsb_hook_push_back`

```
otsb_hook_push_back hook_name code
```

Push a shell snippet onto the back of the execution queue of the named hook. The snippet is evaluated when the hook runs.

`otsb_hook_push_front`

```
otsb_hook_push_front hook_name code
```

Push a shell snippet onto the front of the execution queue of the named hook. The snippet is evaluated when the hook runs.

`otsb_hook_push_fn_back`

```
otsb_hook_push_fn_back hook_name function_name
```

Push a function callback onto the back of the execution queue of the named hook. The named function is invoked directly when the hook runs.

`otsb_hook_push_fn_front`

```
otsb_hook_push_fn_front hook_name function_name
```

Push a function callback onto the front of the execution queue of the named hook. The named function is invoked directly when the hook runs.

`otsb_init`

```
otsb_init
```

Initializes the **ots-build** library.

`otsb_is_absolute`

```
otsb_is_absolute path
```

Returns true if the supplied path is absolute.

`otsb_is_boolean`

```
otsb_is_boolean $value
```

Returns true if the value is a boolean recognized by either `otsb_is_true` or `otsb_is_false`.

`otsb_is_set`

```
otsb_is_set variable_name
```

Returns true if the variable has been set.

`otsb_join`

```
otsb_join delimiter value1 value2 ...
```

Join the supplied values using the delimiter and print the result.

otsb_mesh

```
otsb_mesh left_array right_array start_index pair_format
sequence_format
```

Join corresponding elements from the two named arrays, starting at the given index. Each pair is formatted with the first `printf`-style format using the corresponding left and right elements as its two arguments. Pairs are then joined with the second `printf`-style format, which receives the accumulated result and the next pair as its two arguments. The two arrays must have the same number of elements.

otsb_argv

```
otsb_argv array_name
```

Set the named array variable to the original arguments most recently passed to `otsb_options`.

otsb_make_path_absolute

```
otsb_make_path_absolute path [base_dir]
```

Print an absolute path. Relative paths are resolved against the optional base directory, which defaults to the current directory.

otsb_is_true**otsb_is_false**

```
otsb_is_true $value
otsb_is_false $value
```

These return true if the passed value matches what **ots-build** understands as true or false:

```
true:  the strings 'true' or 'on', or a non-zero number
false: the strings 'false' or 'off', or zero
```

Any other values will return false.

otsb_options

```
otsb_options "$@"
otsb_options "$my_options_from_somewhere_else"
```

`otsb_options` parses the passed tokens for options and commands. It places values for passed options in variables of the same name and queues the requested commands internally. See *Parsing the Command Line* for more information.

Default values for options may be specified by setting the appropriate option variables before or after calling `otsb_options`.

otsb_internal_options

```
otsb_internal_options "$@"
```

Parse only the built-in "internal" options, such as `--help`, `--version`, `--srcdir`, `--import-root`, and the logging and patch options. The results are stored in variables with the `otsbi_` prefix, such as `otsbi_help` and `otsbi_srcdir`.

This parser ignores public options and commands. It is useful when a driver needs to inspect

built-in options before calling `otsb_options`. Calling `otsb_options` after `otsb_internal_options` has the same effect as if `otsb_internal_options` had never been called.

`otsb_split`

```
otsb_split string delimiter
```

Split the string on a single delimiter character and print the non-empty fields one per line.

`otsb_pkg`

This function returns the package name and version (as a single string). It defaults to

```
otsb_pkg () { echo $(basename $(pwd)); }
```

`otsb_pkg_name`

This function determines the name of the package and prints it to the standard output stream. It is used to generate the value of the `package_name` variable. The default definition is

```
otsb_pkg_name () { echo ${package%%-*} ; }
```

It may be overridden (before calling `otsb_set_build_variables`). When called, the package variable will have been set.

`otsb_pkg_version`

This function determines the version of the package and prints it to the standard output stream. It is used to generate the value of the `package_version` variable. The default definition is

```
otsb_pkg_version () { echo ${package##*-} ; }
```

It may be overridden (before calling `otsb_set_build_variables`). When called, the package variable will have been set.

`otsb_prepend_path`

```
otsb_prepend_path $path_var_name $path1 $path2 ...
```

Prepend the specified path(s) to the path variable with the given name. A path variable's value is a colon separated list of paths. For example,

```
foo=a:b:c:d
otsb_prepend_path foo e f g h
```

results in

```
foo=e:f:g:h:a:b:c:d
```

`otsb_default_value`

```
otsb_default_value variable [prefix]
```

Print the default value of the named variable. For array defaults, the value is printed as a space-separated string. If a prefix is supplied, it is used when looking up the underlying option variable.

`otsb_run_cmds`

This executes the build commands listed in the `otsb_req_cmds` variable, which is filled in by `otsb_options`. The commands are run in the order in which they were specified, *not* the order

listed in `otsb_commands`. Do *not* invoke this command more than once.

If a command does not return a successful exit status code, the subsequent commands are not run and `otsb_run_cmds` returns the exit status code of the failing command.

`otsb_set`

```
otsb_set variable value
```

If the named (scalar!) variable has not been set, assign it the specified value:

```
otsb_set prefix "a prefix is here"
```

In most cases this is the preferred way of setting a variable's value, as it allows using environment variables to specify default values.

For example, if the build script is run in an environment with the environment variable `prefix` set:

```
prefix=/my/prefix /imports/package/build all
```

then setting the default value of `prefix` via

```
[...]
prefix=/a/default/value
otsb_options "$@"
```

will ignore the environment, while

```
[...]
otsb_set prefix /a/default/value
otsb_options "$@"
```

will not.

`otsb_setarr`

```
otsb_setarr variable value1 value2 ...
```

If the named (array!) variable has not been set, assign it the specified list of values:

```
otsb_setarr config_opts --do-this --do-that
```

will do the equivalent of

```
config_opts=( --do-this --do-that )
```

if `config_opts` is empty.

`otsb_set-default-value`

```
otsb_set-default-value variable1 variable2 ...
```

Set the named variable or variables to their defaults if they are not already set. Array defaults are assigned with `otsb_setarr`; scalar defaults are assigned with `otsb_set`.

`otsb_reset`

```
otsb_reset variable1 variable2 ...
```

Unset the named variable or variables. This returns the variables to an unset state so that later calls to `otsb_set`, `otsb_setarr`, or the default-value helpers can repopulate them.

otsbv_set

otsbv_set variable value

Similar to `otsb_set`, but used only for option variables provided by **ots-build**.

otsbv_set log true

otsbv_set-default-value

otsbv_set-default-value variable1 variable2 ...

Set the named **ots-build** variable or variables to their defaults if they are not already set. Array defaults are assigned with `otsb_setarr`; scalar defaults are assigned with `otsbv_set`.

otsbv_reset

otsbv_reset variable1 variable2 ...

Unset the named *ots-build* variable or variables. This returns the variables to an unset state so that later calls to `otsbv_set` or the default-value helpers can repopulate them.

otsb_set_build_variables

This function is called after all invocations of `otsb_options`. It generates important build variables; see *Setting build variables*.

otsb_trap_err

See the *Errors* section.

VARIABLES

The following variables are important to **ots-build**. Be careful that you do not inadvertently use them in a different context.

otsb_commands

An array containing the available commands, in the order that they should be invoked.

otsb_req_cmds

An array containing the commands requested on the command line.

otsb_version

The version of the **ots-build** library.

otsbv_import_root

See *Setting build variables*

package

The name of the package, including the version number, derived from the directory that the build script is run in.

patchfile

The (optional) file containing the patch to apply to the source directory, derived from the package name and the directory containing the build script.

version

The package version, determined from the package name.

log

A boolean indicating whether or not logging is enabled.

logpfx

The prefix appended to the command name to create a name for the log file.

logsingle

A boolean indicating that all logs should go to a single file.

logconsole

A boolean indicating that logging should be sent to the console as well as to the log files.

EXAMPLES

* Drivers

Two example drivers are installed: `ots-build-autoconf` and `ots-build-cmake`.

* plugins

Drivers may want to allow per-package modifications to a more generic workflow, including changing the available commands and options.

This must be done prior to calling `otsb_options`, but also requires information that may be available only after parsing the command line, e.g. the value of `otsbv_import_root`. This can be accomplished using the `otsb_internal_options` command inside of a `otsb_options_before` hook, e.g.

```
load_plugins () {
    otsb_internal_options
    local plugin="$otsbv_import_root/plugins"
    if [ -d "$plugin" ]; then
        local -a plugins=()
        mapfile -d '' -t plugins < <( find "$plugin" -name '*.sh'
-print0 | sort )
        otsb_source "${plugins[@]}"
    fi
}
```

COPYRIGHT AND LICENSE

Copyright (C) 2010 Smithsonian Astrophysical Observatory

This file is part of `ots-build`.

`ots-build` is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

AUTHOR

Diab Jerius <djerius@cfa.harvard.edu>