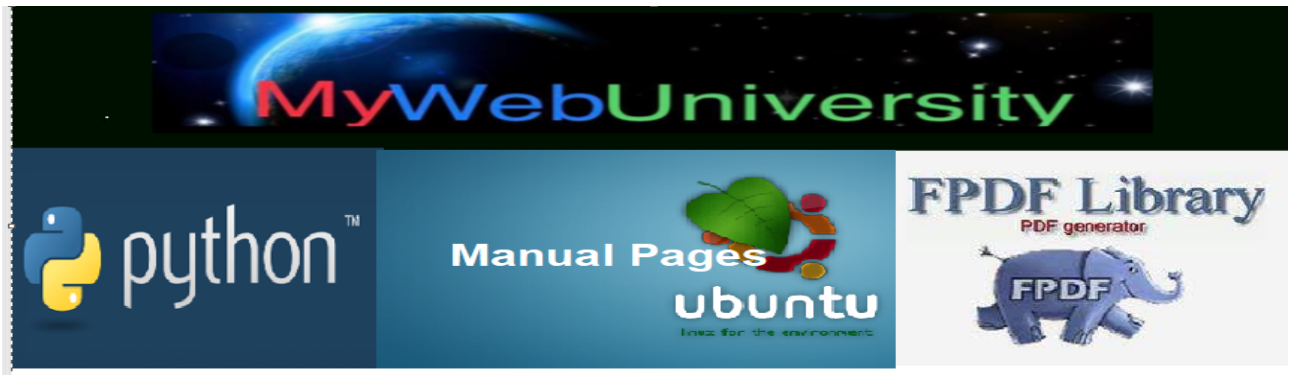

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'dpkg-buildflags.1'

$ man dpkg-buildflags.1

dpkg-buildflags(1) dpkg suite dpkg-buildflags(1)

NAME

dpkg-buildflags - returns build flags to use during package build

SYNOPSIS

dpkg-buildflags [option...] [command]

DESCRIPTION

dpkg-buildflags is a tool to retrieve compilation flags to use during build of Debian packages.

The default flags are defined by the vendor but they can be extended/overridden in several ways:

1. system-wide with /etc/dpkg/buildflags.conf;
2. for the current user with $XDG_CONFIG_HOME/dpkg/buildflags.conf where $XDG_CONFIG_HOME defaults to $HOME/.config;
3. temporarily by the user with environment variables (see section "ENVIRONMENT");
4. dynamically by the package maintainer with environment variables set via debian/rules (see section "ENVIRONMENT").

The configuration files can contain four types of directives:

SET flag value

Override the flag named flag to have the value value.

STRIP flag value

Strip from the flag named flag all the build flags listed in value. Since dpkg 1.16.1.

APPEND flag value

Extend the flag named flag by appending the options given in value. A space is

prepended to the appended value if the flag's current value is non-empty.

PREPEND flag value

Extend the flag named flag by prepending the options given in value. A space is appended to the prepended value if the flag's current value is non-empty. Since dpkg 1.16.1.

The configuration files can contain comments on lines starting with a hash (#). Empty lines are also ignored.

This program was introduced in dpkg 1.15.7.

COMMANDS

--dump

Print to standard output all compilation flags and their values. It prints one flag per line separated from its value by an equal sign (?flag=value?). This is the default action.

--list

Print the list of flags supported by the current vendor (one per line). See the "SUPPORTED FLAGS" section for more information about them.

--status

Display any information that can be useful to explain the behavior of dpkg-buildflags (since dpkg 1.16.5): relevant environment variables, current vendor, state of all feature flags. Also print the resulting compiler flags with their origin.

This is intended to be run from debian/rules, so that the build log keeps a clear trace of the build flags used. This can be useful to diagnose problems related to them.

--export=format

Print to standard output commands that can be used to export all the compilation flags for some particular tool. If the format value is not given, sh is assumed. Only compilation flags starting with an upper case character are included, others are assumed to not be suitable for the environment. Supported formats:

sh Shell commands to set and export all the compilation flags in the environment. The flag values are quoted so the output is ready for evaluation by a shell.

cmdline

Arguments to pass to a build program's command line to use all the compilation flags (since dpkg 1.17.0). The flag values are quoted in shell syntax.

configure

This is a legacy alias for cmdline.

make

Make directives to set and export all the compilation flags in the environment. Output can be written to a Makefile fragment and evaluated using an include directive.

--get flag

Print the value of the flag on standard output. Exits with 0 if the flag is known otherwise exits with 1.

--origin flag

Print the origin of the value that is returned by --get. Exits with 0 if the flag is known otherwise exits with 1. The origin can be one of the following values:

vendor

the original flag set by the vendor is returned;

system

the flag is set/modified by a system-wide configuration;

user

the flag is set/modified by a user-specific configuration;

env the flag is set/modified by an environment-specific configuration.

--query

Print any information that can be useful to explain the behavior of the program: current vendor, relevant environment variables, feature areas, state of all feature flags, whether a feature is handled as a builtin default by the compiler (since dpkg 1.21.14), and the compiler flags with their origin (since dpkg 1.19.0).

For example:

Vendor: Debian

Environment:

DEB_CFLAGS_SET=-O0 -Wall

Area: qa

Features:

bug=no

canary=no

Builtins:

Area: hardening

Features:

pie=no

Builtins:

pie=yes

Area: reproducible

Features:

timeless=no

Builtins:

Flag: CFLAGS

Value: -O0 -Wall

Origin: env

Flag: CPPFLAGS

Value: -D_FORTIFY_SOURCE=3

Origin: vendor

--query-features area

Print the features enabled for a given area (since dpkg 1.16.2). If the feature is handled (even if only on some architectures) as a builtin default by the compiler, then a Builtin field is printed (since dpkg 1.21.14). See the "FEATURE AREAS" section for more details about the currently recognized areas. Exits with 0 if the area is known otherwise exits with 1.

The output is in RFC822 format, with one section per feature. For example:

Feature: pie

Enabled: yes

Builtin: yes

Feature: stackprotector

Enabled: yes

--help

Show the usage message and exit.

--version

Show the version and exit.

SUPPORTED FLAGS

ASFLAGS

Options for the host assembler. Default value: empty. Since dpkg 1.21.0.

CFLAGS

Options for the host C compiler. The default value set by the vendor includes -g and the default optimization level (-O2 usually, or -O0 if the DEB_BUILD_OPTIONS environment variable defines noopt).

CPPFLAGS

Options for the host C preprocessor. Default value: empty.

CXXFLAGS

Options for the host C++ compiler. Same as CFLAGS.

OBJCFLAGS

Options for the host Objective C compiler. Same as CFLAGS. Since dpkg 1.17.7.

OBJCXXFLAGS

Options for the host Objective C++ compiler. Same as CXXFLAGS. Since dpkg 1.17.7.

DFLAGS

Options for the host D compiler (ldc or gdc). Since dpkg 1.20.6.

FFLAGS

Options for the host Fortran 77 compiler. A subset of CFLAGS.

FCFLAGS

Options for the host Fortran 9x compiler. Same as FFLAGS. Since dpkg 1.17.7.

RUSTFLAGS

Options for the host Rust compiler (rustc). Since dpkg 1.22.6ubuntu12.

LDFLAGS

Options passed to the host compiler when linking executables or shared objects.

Note: It is not safe to pass this variable directly to ld(1), but if the variable needs to be passed anyway, then these options should be sanitized to allow only options known to work. In general -Wl, -l and -L options should be safe, with -Wl getting stripped and , in its arguments replaced with spaces. But whether any sanitized option is safe to be passed through, will depend on the intended semantics of the ld(1) invocation.

Default value: empty.

ASFLAGS_FOR_BUILD

Options for the build assembler. Default value: empty. Since dpkg 1.22.1.

CFLAGS_FOR_BUILD

Options for the build C compiler. The default value set by the vendor includes -g and the default optimization level (-O2 usually, or -O0 if the DEB_BUILD_OPTIONS environment

variable defines noopt). Since dpkg 1.22.1.

CPPFLAGS_FOR_BUILD

Options for the build C preprocessor. Default value: empty. Since dpkg 1.22.1.

CXXFLAGS_FOR_BUILD

Options for the build C++ compiler. Same as CFLAGS_FOR_BUILD. Since dpkg 1.22.1.

OBJCFLAGS_FOR_BUILD

Options for the build Objective C compiler. Same as CFLAGS_FOR_BUILD. Since dpkg 1.22.1.

OBJCXXFLAGS_FOR_BUILD

Options for the build Objective C++ compiler. Same as CXXFLAGS_FOR_BUILD. Since dpkg 1.22.1.

DFLAGS_FOR_BUILD

Options for the build D compiler (ldc or gdc). Since dpkg 1.22.1.

FFLAGS_FOR_BUILD

Options for the build Fortran 77 compiler. A subset of CFLAGS_FOR_BUILD. Since dpkg 1.22.1.

FCFLAGS_FOR_BUILD

Options for the build Fortran 9x compiler. Same as FFLAGS_FOR_BUILD. Since dpkg 1.22.1.

LDFLAGS_FOR_BUILD

Options passed to the build compiler when linking executables or shared objects.

Note: It is not safe to pass this variable directly to ld(1), but if the variable needs to be passed anyway, then these options should be sanitized to allow only options known to work. In general -Wl, -l and -L options should be safe, with -Wl getting stripped and , in its arguments replaced with spaces. But whether any sanitized option is safe to be passed through, will depend on the intended semantics of the ld(1) invocation.

Default value: empty.

Since dpkg 1.22.1.

RUSTFLAGS_FOR_BUILD

Options for the build Rust compiler (rustc). Since dpkg 1.22.6ubuntu12.

New flags might be added in the future if the need arises (for example to support other languages).

Feature areas are currently vendor specific, and the ones described below are only recognized on Debian and derivatives.

Each area feature can be enabled and disabled in the DEB_BUILD_OPTIONS and DEB_BUILD_MAINT_OPTIONS environment variable's area value with the ?+? and ?-? modifier. Following the general syntax of these variables (described in dpkg-buildpackage(1)), multiple feature areas can be specified separated by spaces, where each get feature specifiers as mandatory parameters after an equal sign (?=?). The feature specifiers are comma-separated and parsed from left to right, where the settings within the same feature specifier override previous ones, even if the feature specifiers are split across multiple space-separated feature area settings for the same area.

For example, to enable the hardening ?pie? feature and disable the ?fortify? feature you can do this in debian/rules:

```
export DEB_BUILD_MAINT_OPTIONS = hardening=+pie,-fortify
```

The special feature all (valid in any area) can be used to enable or disable all area features at the same time. Thus disabling everything in the hardening area and enabling only ?format? and ?fortify? can be achieved with:

```
export DEB_BUILD_MAINT_OPTIONS = hardening=-all,+format,+fortify
```

Multiple feature areas can be set:

```
export DEB_BUILD_MAINT_OPTIONS = hardening=+pie abi=+lfs
```

The override behavior applies as much to the all special feature, as to specific features, which should allow for composition. Thus to enable ?lfs? in the abi area, and only ?pie? and ?fortify? in the hardening area, but ?format? only when CONDITION is defined, this could be done with:

```
export DEB_BUILD_MAINT_OPTIONS = hardening=-all,+pie,+format abi=+lfs
?
DEB_BUILD_MAINT_OPTIONS += hardening=+fortify
ifdef CONDITION
DEB_BUILD_MAINT_OPTIONS += hardening=-format
endif
```

abi

Several compile-time options (detailed below) can be used to enable features that can change the ABI of a package, but cannot be enabled by default due to backwards compatibility reasons unless coordinated or checked individually.

lfs This setting (since dpkg 1.22.0; disabled by default) enables Large File Support on 32-bit architectures where their ABI does not include LFS by default, by adding -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64 to CPPFLAGS.

When this feature is enabled it will override the value from the same feature in the future feature area.

time64

This setting (since dpkg 1.22.0; enabled by default except for i386 and hurd-i386 since dpkg 1.22.5) enables 64-bit time_t support on 32-bit architectures where their ABI does not include it by default, by adding -D_TIME_BITS=64 to CPPFLAGS. This setting automatically enables the lfs feature from the abi feature area.

If the setting is enabled explicitly then it gets enabled on all architectures including i386 but not hurd-i386 (where the kernel does not have time64 interfaces), ignoring the binary backwards compatibility default.

It is also enabled by default by gcc on the armel, armhf, hppa, m68k, mips, mipsel, powerpc and sh4 Debian architectures, where disabling the feature will add instead -U_LARGEFILE_SOURCE -U_FILE_OFFSET_BITS -U_TIME_BITS to CPPFLAGS.

future

Several compile-time options (detailed below) can be used to enable features that should be enabled by default, but cannot due to backwards compatibility reasons.

lfs This setting (since dpkg 1.19.0; disabled by default) is now an alias for the lfs feature in the abi area, use that instead. The feature from the abi area overrides this setting.

qa

Several compile-time options (detailed below) can be used to help detect problems in the source code or build system.

bug-implicit-func

This setting (since dpkg 1.22.3; enabled by default since dpkg 1.22.6) adds -Werror=implicit-function-declaration to CFLAGS.

bug This setting (since dpkg 1.17.4; disabled by default) adds any warning option that reliably detects problematic source code. The warnings are fatal. The only currently supported flags are CFLAGS and CXXFLAGS with flags set to -Werror=array-bounds, -Werror=clobbered, -Werror=implicit-function-declaration and -Werror=volatile-register-var.

This feature handles -Werror=implicit-function-declaration via the bug-implicit-func feature, if that has not been specified.

canary

This setting (since dpkg 1.17.14; disabled by default) adds dummy canary options to the build flags, so that the build logs can be checked for how the build flags propagate and to allow finding any omission of normal build flag settings. The only currently supported flags are CPPFLAGS, CFLAGS, OBJCFLAGS, CXXFLAGS and OBJCXXFLAGS with flags set to -D__DEB_CANARY_flag_random-id__, and LDFLAGS set to -Wl,-z,deb-canary-random-id.

elfpackagemetadata

This setting (since dpkg 1.22.6ubuntu11; enabled by default) adds an ELF note containing package metadata to every ELF binary being built. If the $DEB_BUILD_DEBUG_INFO_URL environment variable is defined, it will be used to include the value as a debuginfod URL. This follows the specification defined at: https://systemd.io/ELF_PACKAGE_METADATA/

optimize

Several compile-time options (detailed below) can be used to help optimize a resulting binary (since dpkg 1.21.0). Note: Enabling all these options can result in unreproducible binary artifacts.

lto This setting (since dpkg 1.21.0) enables Link Time Optimization by adding -flto=auto -ffat-lto-objects to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS, FCFLAGS and LDFLAGS.

Note: this option is enabled by default in Ubuntu on all 64bit architectures, except for riscv64. A list of packages with LTO disabled by default is maintained in /usr/share/lto-disabled-list/lto-disabled-list, provided by the lto-disabled-list package.

sanitize

Several compile-time options (detailed below) can be used to help sanitize a resulting binary against memory corruptions, memory leaks, use after free, threading data races and undefined behavior bugs. Note: These options should not be used for production builds as they can reduce reliability for conformant code, reduce security or even functionality.

address

This setting (since dpkg 1.18.0; disabled by default) adds -fsanitize=address to LDFLAGS and -fsanitize=address -fno-omit-frame-pointer to CFLAGS and CXXFLAGS.

thread

This setting (since dpkg 1.18.0; disabled by default) adds -fsanitize=thread to CFLAGS, CXXFLAGS and LDFLAGS.

leak

This setting (since dpkg 1.18.0; disabled by default) adds -fsanitize=leak to LDFLAGS. It gets automatically disabled if either the address or the thread features are enabled, as they imply it.

undefined

This setting (since dpkg 1.18.0; disabled by default) adds -fsanitize=undefined to CFLAGS, CXXFLAGS and LDFLAGS.

hardening

Several compile-time options (detailed below) can be used to help harden a resulting binary against memory corruption attacks, or provide additional warning messages during compilation. Except as noted below, these are enabled by default for architectures that support them.

format

This setting (since dpkg 1.16.1; enabled by default) adds -Wformat -Werror=format-security to CFLAGS, CXXFLAGS, OBJCFLAGS and OBJCXXFLAGS. This will warn about improper format string uses, and will fail when format functions are used in a way that represent possible security problems. At present, this warns about calls to printf and scanf functions where the format string is not a string literal and there are no format arguments, as in printf(foo); instead of printf("%s", foo); This may be a security hole if the format string came from untrusted input and contains ?%n?.

fortify

This setting (since dpkg 1.16.1; enabled by default) adds -D_FORTIFY_SOURCE=3 to CPPFLAGS. During code generation the compiler knows a great deal of information about buffer sizes (where possible), and attempts to replace insecure unlimited length buffer function calls with length-limited ones. This is especially useful for old, crufty code. Additionally, format strings in writable memory that contain ?%n? are blocked. If an application depends on such a format string, it will need to be worked around. Note that for this option to have any effect, the source must also be compiled with -O1 or higher. If the environment variable DEB_BUILD_OPTIONS contains noopt, then fortify support will be disabled, due to new warnings being issued by glibc 2.16 and later.

stackprotector

This setting (since dpkg 1.16.1; enabled by default if stackprotectorstrong is not in use) adds -fstack-protector --param=ssp-buffer-size=4 to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS. This adds safety checks against stack overwrites. This renders many potential code injection attacks into aborting situations. In the best case this turns code injection vulnerabilities into denial of service or into non-issues (depending on the application).

This feature requires linking against glibc (or another provider of __stack_chk_fail), so needs to be disabled when building with -nostdlib or -ffreestanding or similar.

stackprotectorstrong

This setting (since dpkg 1.17.11; enabled by default) adds -fstack-protector-strong to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS. This is a stronger variant of stackprotector, but without significant performance penalties.

Disabling stackprotector will also disable this setting.

This feature has the same requirements as stackprotector, and in addition also requires gcc 4.9 and later.

stackclash

This setting (since dpkg 1.22.0; enabled by default) adds -fstack-clash-protection on amd64, arm64, armhf and armel to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS. This adds code to prevent stack clash style attacks.

branch

This setting (since dpkg 1.22.0; enabled by default) adds -fcf-protection on amd64 and -mbranch-protection=standard on arm64 to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS. This adds branch protection to indirect calls, jumps and returns to check whether these are valid at run-time.

relro

This setting (since dpkg 1.16.1; enabled by default) adds -Wl,-z,relro to LDFLAGS. During program load, several ELF memory sections need to be written to by the linker. This flags the loader to turn these sections read-only before turning over control to the program. Most notably this prevents GOT overwrite attacks. If this option is disabled, bindnow will become disabled as well.

bindnow

This setting (since dpkg 1.16.1; disabled by default) adds -Wl,-z,now to LDFLAGS. During program load, all dynamic symbols are resolved, allowing for the entire PLT to be

marked read-only (due to relro above). The option cannot become enabled if relro is not enabled.

pie This setting (since dpkg 1.16.1; with no global default since dpkg 1.18.23, as it is enabled by default now by gcc on the amd64, arm64, armel, armhf, hurd-i386, i386, mips, mipsel, mips64el, powerpc, ppc64, ppc64el, riscv64, s390x, sparc and sparc64 Debian architectures) adds the required options to enable or disable PIE via gcc specs files, if needed, depending on whether gcc injects on that architecture the flags by itself or not. When the setting is enabled and gcc injects the flags, it adds nothing. When the setting is enabled and gcc does not inject the flags, it adds -fPIE (via /usr/share/dpkg/pie-compiler.specs) to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS, and -fPIE -pie (via /usr/share/dpkg/pie-link.specs) to LDFLAGS. When the setting is disabled and gcc injects the flags, it adds -fno-PIE (via /usr/share/dpkg/no-pie-compile.specs) to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS, and -fno-PIE -no-pie (via /usr/share/dpkg/no-pie-link.specs) to LDFLAGS.

Position Independent Executable (PIE) is needed to take advantage of Address Space Layout Randomization (ASLR), supported by some kernel versions. While ASLR can already be enforced for data areas in the stack and heap (brk and mmap), the code areas must be compiled as position-independent. Shared libraries already do this (-fPIC), so they gain ASLR automatically, but binary .text regions need to be built as PIE to gain ASLR. When this happens, ROP (Return Oriented Programming) attacks are much harder since there are no static locations to bounce off of during a memory corruption attack.

PIE is not compatible with -fPIC, so in general care must be taken when building shared objects. But because the PIE flags emitted get injected via gcc specs files, it should always be safe to unconditionally set them regardless of the object type being compiled or linked.

Static libraries can be used by programs or other shared libraries. Depending on the flags used to compile all the objects within a static library, these libraries will be usable by different sets of objects:

none

Cannot be linked into a PIE program, nor a shared library.

-fPIE

Can be linked into any program, but not a shared library (recommended).

-fPIC

Can be linked into any program and shared library.

If there is a need to set these flags manually, bypassing the gcc specs injection, there are several things to take into account. Unconditionally and explicitly passing -fPIE, -fpie or -pie to a build-system using libtool is safe as these flags will get stripped when building shared libraries. Otherwise on projects that build both programs and shared libraries you might need to make sure that when building the shared libraries -fPIC is always passed last (so that it overrides any previous -PIE) to compilation flags such as CFLAGS, and -shared is passed last (so that it overrides any previous -pie) to linking flags such as LDFLAGS. Note: This should not be needed with the default gcc specs machinery.

Additionally, since PIE is implemented via a general register, some register starved architectures (but not including i386 anymore since optimizations implemented in gcc >= 5) can see performance losses of up to 15% in very text-segment-heavy application workloads; most workloads see less than 1%. Architectures with more general registers (e.g. amd64) do not see as high a worst-case penalty.

reproducible

The compile-time options detailed below can be used to help improve build reproducibility or provide additional warning messages during compilation. Except as noted below, these are enabled by default for architectures that support them.

timeless

This setting (since dpkg 1.17.14; enabled by default) adds -Wdate-time to CPPFLAGS.

This will cause warnings when the __TIME__, __DATE__ and __TIMESTAMP__ macros are used.

fixfilepath

This setting (since dpkg 1.19.1; enabled by default) adds -ffile-prefix-map=BUILDPATH=. to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS where BUILDPATH is set to the top-level directory of the package being built. This has the effect of removing the build path from any generated file.

If both fixdebugpath and fixfilepath are set, this option takes precedence, because it is a superset of the former.

Note: If the build process captures the build flags into the resulting built objects, that will make the package unreproducible. And while disabling this option might make some of the objects reproducible again this would also require disabling fixdebugpath,

which might make any generated debug symbols objects unreproducible. The ideal fix is to stop capturing build flags.

fixdebugpath

This setting (since dpkg 1.18.5; enabled by default) adds -fdebug-prefix-map=BUILDPATH=/usr/src/PKGNAME-PKGVER to CFLAGS, CXXFLAGS, OBJCFLAGS, OBJCXXFLAGS, FFLAGS and FCFLAGS where BUILDPATH is set to the top-level directory of the package being built. This has the effect of removing the build path from any generated debug symbols and replacing it with /usr/src/PKGNAME-PKGVER, where PKGNAME is the source package name and PKGVER is the source package version.

Note: This feature has similar reproducible properties as fixfilepath.

ENVIRONMENT

There are 2 sets of environment variables doing the same operations, the first one (DEB_flag_op) should never be used within debian/rules. It's meant for any user that wants to rebuild the source package with different build flags. The second set (DEB_flag_MAINT_op) should only be used in debian/rules by package maintainers to change the resulting build flags.

DEB_flag_SET

DEB_flag_MAINT_SET (since dpkg 1.16.1)

This variable can be used to force the value returned for the given flag.

DEB_flag_STRIP (since dpkg 1.16.1)

DEB_flag_MAINT_STRIP (since dpkg 1.16.1)

This variable can be used to provide a space separated list of options that will be stripped from the set of flags returned for the given flag.

DEB_flag_APPEND

DEB_flag_MAINT_APPEND (since dpkg 1.16.1)

This variable can be used to append supplementary options to the value returned for the given flag.

DEB_flag_PREPEND (since dpkg 1.16.1)

DEB_flag_MAINT_PREPEND (since dpkg 1.16.1)

This variable can be used to prepend supplementary options to the value returned for the given flag.

DEB_BUILD_OPTIONS

DEB_BUILD_MAINT_OPTIONS (since dpkg 1.16.1)

These variables can be used by a user or maintainer to disable/enable various area features that affect build flags. The DEB_BUILD_MAINT_OPTIONS variable overrides any setting in the DEB_BUILD_OPTIONS feature areas. See the "FEATURE AREAS" section for details.

Some build profiles might currently require their counterpart to also be included in the build options, the list of such build profiles is vendor specific. For more information about build profiles, see dpkg-buildpackage(1).

DEB_VENDOR

This setting defines the current vendor. If not set, it will discover the current vendor by reading /etc/dpkg/origins/default.

DEB_BUILD_PATH

This variable sets the build path (since dpkg 1.18.8) to use in features such as fixfilepath so that they can be controlled by the caller. This variable is currently Debian and derivatives-specific.

DEB_BUILD_DEBUGPATH

This variable sets the debug build path (since dpkg 1.21.9ubuntu2) to use in features such as fixdebugpath so that they can be controlled by the caller. This variable is currently Ubuntu-specific.

DEB_HOST_ARCH

Sets the host architecture. This affects the build flags that are emitted, which is typically relevant when cross-compiling, where DEB_HOST_ARCH is different to DEB_BUILD_ARCH.

DPKG_COLORS

Sets the color mode (since dpkg 1.18.5). The currently accepted values are: auto (default), always and never.

DPKG_NLS

If set, it will be used to decide whether to activate Native Language Support, also known as internationalization (or i18n) support (since dpkg 1.19.0). The accepted values are: 0 and 1 (default).

FILES

Configuration files

/etc/dpkg/buildflags.conf

System wide configuration file.

$XDG_CONFIG_HOME/dpkg/buildflags.conf or

$HOME/.config/dpkg/buildflags.conf

User configuration file.

Packaging support

/usr/share/dpkg/buildflags.mk

Makefile snippet that loads (and optionally exports) all flags supported by dpkg-buildflags into variables (since dpkg 1.16.1).

/usr/share/dpkg/buildtools.mk

Makefile snippet that sets suitable host and build tools (and optionally exports them) into variables (since dpkg 1.19.0).

LTO exclusion list

/usr/share/lto-disabled-list/lto-disabled-list

A text file, listing source packages and architectures where LTO should not be enabled, allowing the package to build without modifying it.

EXAMPLES

To pass build flags to a build command in a Makefile:

```
$(MAKE) $(shell dpkg-buildflags --export=cmdline)
./configure $(shell dpkg-buildflags --export=cmdline)
```

To set build flags in a shell script or shell fragment, eval can be used to interpret the output and to export the flags in the environment:

```
eval "$(dpkg-buildflags --export=sh)" && make
```

or to set the positional parameters to pass to a command:

```
eval "set -- $(dpkg-buildflags --export=cmdline)"
for dir in a b c; do (cd $dir && ./configure "$@" && make); done
```

Usage in debian/rules

You should call dpkg-buildflags or include buildflags.mk from the debian/rules file to obtain the needed build flags to pass to the build system. Note that older versions of dpkg-buildpackage (before dpkg 1.16.1) exported these flags automatically. However, you should not rely on this, since this breaks manual invocation of debian/rules.

For packages with autoconf-like build systems, you can pass the relevant options to configure or make(1) directly, as shown above.

For other build systems, or when you need more fine-grained control about which flags are passed where, you can use --get. Or you can include buildflags.mk instead, which takes care

of calling dpkg-buildflags and storing the build flags in make variables.

If you want to export all buildflags into the environment (where they can be picked up by your build system):

```
DPKG_EXPORT_BUILDFLAGS = 1
include /usr/share/dpkg/buildflags.mk
```

For some extra control over what is exported, you can manually export the variables (as none are exported by default):

```
include /usr/share/dpkg/buildflags.mk
export CPPFLAGS CFLAGS LDFLAGS
```

And you can of course pass the flags to commands manually:

```
include /usr/share/dpkg/buildflags.mk
build-arch:
	$(CC) -o hello hello.c $(CPPFLAGS) $(CFLAGS) $(LDFLAGS)
```