



# Using HP Code Advisor in Application Builds: [HP Code Advisor] White Paper

Abstract.....	2
Intended audience .....	2
Prerequisites.....	2
Terms and definitions .....	2
Related information .....	2
Introduction.....	3
Using cadvise with the builds .....	3
Method 1: Controlling use of cadvise through makefile variables.....	4
Method 2: Introducing a cadvise-specific target.....	6
Method 3: Using compiler wrappers .....	9
For more information.....	9

## Abstract

This document describes methods for using HP Code Advisor in the build environment of your application.

## Intended audience

This document is intended for C/C++ developers who want to integrate HP Code Advisor in their application build process. The application is assumed to span over multiple source files and organized to manage the build by one or more makefiles.

## Prerequisites

Following are the prerequisites for integrating HP Code Advisor in application build process:

- Knowledge of makefiles
- Knowledge of compilation and linking process of an executable/shared library
- Basic knowledge of HP Code Advisor

## Terms and definitions

The following table lists the terms and definitions used in this document.

<b>Term</b>	<b>Definition</b>
Build	The process of compiling and linking source files to build an application.
Application	The end result of building. In this case, a single or a collection of executables and libraries built.
Build line	A compilation or link command line.

## Related information

The following documents provide additional information about HP Code Advisor:

- *HP Code Advisor Release Notes*
- *HP Code Advisor User's Guide*
- *HP Code Advisor Diagnostics*
- *HP C/aC++ Features to Improve Developer Productivity - Whitepaper*

See the HP Code Advisor web page for a list of product documentation:

[www.hp.com/go/cadvise](http://www.hp.com/go/cadvise)

## Introduction

HP Code Advisor (cadvise), is a static analysis tool for C and C++ programs. It reports various coding errors in the source code. Cadvise analyses the source code using the same language mode, defines, header files etc. as used by your compiler. Therefore, cadvise requires an entire build line as the input. This is achieved by prefixing cadvise to the regular build line. Cadvise also does extensive cross-file analysis to find defects across files, in order to do this cadvise must be used with the link line also.

Cadvise first executes the build line to which it has been prefixed and then performs its own code analysis. A build that is performed using cadvise does not change anything in the actual compilation or link process. Cadvise performs the code analysis only after the regular build passes, thereby providing an interface to facilitate integration of cadvise with an existing build environment. This document describes techniques to incorporate cadvise into the build environment of an application.

## Using cadvise with the builds

Figure 1 shows a sample makefile that captures the common elements of a makefile used to build an application.

Figure 1: A Sample makefile

```
# This is a sample Makefile
# This is used to illustrate how Cadvise can be #incorporated into
application builds

# List of source code files
SOURCES = source1.c source2.c

# Create a list of the object files using macro #substitutions
OBJECTS = $(SOURCES:.c=.o)

# Build tools and options

# Use aCC as the default C compiler
CC      = aCC

# Flags
CFLAGS  = -c -Ae -O -I.

# Use aCC as the linker
LD      = aCC
LDFLAGS = -o

# Application target and dependencies
YourApplication: $(OBJECTS)
             $(LD) $(LDFLAGS) @$ $(OBJECTS) $(LIBS)

# .o files depend on .c files
$(OBJECTS): includes.h
             $(CC) $(CFLAGS) $(SOURCES)

# Clean built files
clean:
             rm -rf *.o *~ YourApplication core
```

The elements of the sample makefile are as follows:

- The application in consideration has two `.c` files and a `.h` file.
- The makefile uses conventionally named variables, `CC` and `LD`, to define the compiler and linker.
- The makefile has two target definitions as listed below:
  - A default target called *YourApplication*, which is used to build the application.
  - A clean target, which is used to remove objects generated during the build.
- Building using this makefile results in the following output:

```

$ make
          aCC -c -Ae -O -I. source1.c source2.c
source1.c:
source2.c:
          aCC -o YourApplication source1.o source2.o
$

```

In the methods described below, the default kind of build for the application never changes, that is, executing a `make` command always builds the application without `cadvise`.

## Method 1: Controlling use of `cadvise` through makefile variables

Change the `CC` and `LD` variables to prefix `cadvise` specific variables, which can be defined in the command line, to the compiler and linker. An example is shown in Figure 2 below.

NOTE: If a build has autogenerated makefiles, changes to the makefiles are lost every time it is regenerated. In such circumstances, `cadvise`-specific changes must be made in template files, which generate makefiles.

Figure 2: Invoking `CC` and `LD` variables using `cadvise` options

```

# This is a sample Makefile
# This is used to illustrate how Cadvise can be incorporated into
# application builds

# List of source code files
SOURCES = source1.c source2.c

# Create a list of the object files using macro substitutions
OBJECTS = $(SOURCES:.c=.o)

# Introducing cadvise options in the build tools and options
# Use aCC as the default C compiler
CC      = $(CADVISE) $(CADVISE_OPTS) aCC
# Flags
CFLAGS  = -c -Ae -O -I.
# Use aCC as the linker
LD      = $(CADVISE) $(CADVISE_OPTS) aCC
LDFLAGS = -o

# Application target and dependencies
YourApplication: $(OBJECTS)
              $(LD) $(LDFLAGS) @$ $(OBJECTS) $(LIBS)

# .o files depend on .c files
$(OBJECTS): includes.h
           $(CC) $(CFLAGS) $(SOURCES)

# Clean built files
clean:
      rm -rf *.o *~ YourApplication core

```

---

You can build your application using `cadvise` by overriding the `cadvise` corresponding variables from the command line, as shown below:

```
$ CADVISE="cadvise" CADVISE_OPTS="-pdb YourApplication.pdb" make
    cadvise -pdb YourApplication.pdb aCC -c -Ae -O -I. source1.c
source2.c
source1.c:
source2.c:
    cadvise -pdb YourApplication.pdb aCC -o YourApplication
source1.o source2.o
$
```

Alternatively, you can choose not make any changes to the sample makefile (figure 1) and instead override the `CC` and `LD` variables on the command line as follows:

```
$ CC="cadvise -pdb YourApplication.pdb aCC" LD=$CC make
    cadvise -pdb YourApplication.pdb aCC -c -Ae -O -I. source1.c
source2.c
source1.c:
source2.c:
    cadvise -pdb YourApplication.pdb aCC -o YourApplication
source1.o source2.o
```

If you are using a version of `make`, which supports the use of conditionals in makefiles, you can modify your makefile to create a switch for `cadvise` usage. This option can be toggled in the command line to build the application with or without `cadvise`. Figure 3 shows the makefile using a conditional.

Figure 3: Creating a command-line switch

```
# This is a sample Makefile
# This is used to illustrate how Cadvise can be incorporated into
# application builds

# List of source code files
SOURCES = source1.c source2.c

# Create a list of the object files using macro substitutions
OBJECTS = $(SOURCES:.c=.o)

# Conditional switch option
ifeq ($(USE_CADVISE),yes)
    CADVISE=cadvise
    CADVISE_OPTS= -pdb YourApplication.pdb
endif

# Build tools and options

# Use aCC as the default C compiler
CC      = $(CADVISE) $(CADVISE_OPTS) aCC

# Flags
CFLAGS  = -c -Ae -O -I.

# Use aCC as the linker
LD      = $(CADVISE) $(CADVISE_OPTS) aCC
LDFLAGS = -o

# Application target and dependencies
YourApplication: $(OBJECTS)
            $(LD) $(LDFLAGS) $@ $(OBJECTS) $(LIBS)

# .o files depend on .c files
$(OBJECTS): includes.h
            $(CC) $(CFLAGS) $(SOURCES)

# Clean built files
clean:
    rm -rf *.o *~ YourApplication core
```

Use the following command line to build the application with cadvise:

```
$ USE_CADVISE=yes gmake
    cadvise -pdb YourApplication.pdb aCC -c -Ae -O -I.
    source1.c source2.c
    source1.c:
    source2.c:
    cadvise -pdb YourApplication.pdb aCC -o YourApplication
    source1.o source2.o
```

## Method 2: Introducing a cadvise-specific target

You can introduce a new target for integrating cadvise into the build process. Figure 4 shows the introduction of a new target.

Figure 4: Introducing a target to invoke the build through cadvice

```
# This is a sample Makefile
# This is used to illustrate how Cadvice can be incorporated into
# application builds

# List of source code files
SOURCES = source1.c source2.c

# Create a list of the object files using macro substitutions
OBJECTS = $(SOURCES:.c=.o)

# Build tools and options

# Use aCC as the default C compiler
CC      = aCC

# Flags
CFLAGS  = -c -Ae -O -I.

# Use aCC as the linker
LD      = aCC
LDFLAGS = -o

# Application target and dependencies
YourApplication: $(OBJECTS)
             $(LD) $(LDFLAGS) $@ $(OBJECTS) $(LIBS)

# Introducing Cadvice specific target
Cadvice-YourApplication:
    make CC="cadvice -pdb YourApplication.pdb $(CC)" LD=$(CC) YourApplication

# .o files depend on .c files
$(OBJECTS): includes.h
           $(CC) $(CFLAGS) $(SOURCES)

# Clean built files
clean:
    rm -rf *.o *~ YourApplication core
```

The advantage of introducing a separate target is that the customized actions to suit projects and teams can be implemented along with a build using cadvice.

For example, you can check number of diagnostics generated in the build before and after a code change and check if the code quality has improved or degraded.

If the number of diagnostics increases, the build can fail, thereby forcing the developer to ensure that the change is of bad quality. Figure 5 shows a basic implementation of this functionality in the cadvice specific target, described above.

Figure 5: Implementing custom actions to check code quality

```
# This is a sample Makefile
# This is used to illustrate how Cadvise can be incorporated into
# application builds

# List of source code files
SOURCES = source1.c source2.c

# Create a list of the object files using macro substitutions
OBJECTS = $(SOURCES:.c=.o)

# Build tools and options

# Use aCC as the default C compiler
CC      = aCC

# Flags
CFLAGS  = -c -Ae -O -I.

# Use aCC as the linker
LD      = aCC
LDFLAGS = -o

# Application target and dependencies
YourApplication: $(OBJECTS)
            $(LD) $(LDFLAGS) $@ $(OBJECTS) $(LIBS)

# Basic implementation of customized actions
cadvise-YourApplication:
    make CADVISE=cadvise CADVISE_OPTS="-pdb YourApplication.pdb"
YourApplication
    cadvise report -severity all -pdb YourApplication.pdb -
basepdb:new > new_diagnostics
    if [ -f new_diagnostics ] ; \
    then \
    echo "New diagnostics introduced. Please verify them.\n" ; \
    cat new_diagnostics ; \
    exit 1 ; \
    fi ;

# .o files depend on .c files
$(OBJECTS): includes.h
            $(CC) $(CFLAGS) $(SOURCES)

# Clean built files
clean:
    rm -rf *.o *~ YourApplication core
```

## Method 3: Using compiler wrappers

It is common in several build environments for compilers to be invoked through wrappers. Since `cadvise` will not be able to recognize this wrapper as a supported compiler, you need to use the `-compiler` option to specify the name of the actual compiler being used from within the wrapper.

For example:

If the compiler wrapper being used is a shell script named `my_aCC`, which performs other operations and then invokes `aCC` for the actual compilation, then `cadvise` should be prefixed to `myaCC` as follows

```
$cadvise -pdb YourApplication.pdb -compiler aCC my_aCC <compiler-  
options>
```

- In case your compiler wrapper also sets some compiler options, then instead of modifying the make file modify the wrapper, such that `cadvise` is invoked from within the wrapper where it has visibility to all compiler options being used.

For example:

If the compiler wrapper appends the 64-bit mode compilation option while it invokes the compiler within itself as shown below:

```
$ cat my_aCC_bit_64bit  
#!/bin/ksh  
aCC +DD64 $@
```

The `+DD64` option is then visible to `cadvise` from outside the wrapper. Therefore, you can modify the wrapper as:

```
$ cat my_aCC_bit_64bit  
#!/bin/ksh  
cadvise -pdb YourApplication.pdb aCC +DD64 $@
```

## For more information

[www.hp.com/go/cadvise](http://www.hp.com/go/cadvise)

© Copyright 2009 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Linux is a U.S. registered trademark of Linus Torvalds. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group.

September 2009

