System Inspection

This chapter will describe how you can use CMake to inspect the environment of the system on which the software is being built. This is a critical factor in creating cross-platform applications or libraries. It covers how to find and use system and user installed header files and libraries. It also covers some of the more advanced features of CMake including the `try_compile` and `try_run` commands. These commands are extremely powerful tools for determining the capabilities of the system and compiler that is hosting your software. This chapter also describes how to generate configured files and how to cross compile with CMake. Finally, the steps required to enable a project for the `find_package` command are covered, explaining how to create a `<Package>Config.cmake` file and other required files.

5.1 Using Header Files and Libraries

Many C and C++ programs depend on external libraries. However, when it comes to the practical aspects of compiling and linking a project, taking advantage of existing libraries can be difficult for both developers and users. Problems usually show up as soon as the software is built on a system other than that on which it was developed. Assumptions regarding where libraries and header files are located become obvious when they are not installed in the same place on the new computer and the build system is unable to find them. CMake has many features to aid developers in the integration of external software libraries into a project.

The CMake commands that are most relevant to this type of integration are the `find_file`, `find_library`, `find_path`, `find_program`, and `find_package` commands. For most C and C++ libraries, a combination of `find_library` and `find_path` will be enough to compile and link with an installed library. `find_library` can be used to locate, or allow a

user to locate a library, and `find_path` can be used to find the path to a representative include file from the project. For example, if you wanted to link to the tiff library, you could use the following commands in your CMakeLists.txt file:

```
# find libtiff, looking in some standard places
find_library (TIFF_LIBRARY
              NAMES tiff tiff2
              PATHS /usr/local/lib /usr/lib
              )

# find tiff.h looking in some standard places
find_path (TIFF_INCLUDES tiff.h
          /usr/local/include
          /usr/include
          )

include_directories (${TIFF_INCLUDES})

add_executable (mytiff mytiff.c )

target_link_libraries (myprogram ${TIFF_LIBRARY})
```

The first command used is `find_library` which in this case will look for a library with the name tiff or tiff2. The `find_library` command only requires the library's base name without any platform specific prefixes or suffixes, such as lib and .dll. The appropriate prefixes and suffixes for the system running CMake will be added to the library name automatically when CMake attempts to find it. All the `FIND_*` commands will look in the `PATH` environment variable. In addition, the commands allow the specification of additional search paths as arguments listed after the `PATHS` marker argument. As well as supporting standard paths, windows registry entries and environment variables can be used to construct search paths. The syntax for registry entries is the following:

```
[HKEY_CURRENT_USER\\Software\\Kitware\\Path;Build1]
```

Because software can be installed in many different places, it is impossible for CMake to find the library every time, but most standard installations should be covered. The `find_*` commands automatically create a cache variable so that users can override or specify the location from the CMake GUI. This way if CMake is unable to locate the files it is looking for users will still have an opportunity to specify them. If CMake does not find a file, the value is set to `VAR-NOTFOUND`. This value tells CMake that it should continue looking each time CMake's configure step is run. Note that in `if` statements, values of `VAR-NOTFOUND` will evaluate as false.

The next command used is `find_path`. This is a general purpose command that, in this example, is used to locate a header file from the library. Header files and libraries are often installed in different locations, and both locations are required to compile and link programs that use them. `find_path` is similar to `find_library`, although it only supports one name. It supports a list of search paths.

The next part of the CMakeLists file uses the variables created by the `find_*` commands. The variables can be used without checking for valid values as CMake will print an error message notifying the user if any of the required variables have not been set. The user can then set the cache values and reconfigure until the message goes away. Optionally, a CMakeLists file could use the `if` command to use alternative libraries or options to build the project without the library if it cannot be found.

From the above example you should be able to see how using the `find_*` commands can help your software to compile on a wide variety of systems. It is worth noting that the `find_*` commands search for a match starting with the first argument and first path. So when listing paths and library names you should list your preferred paths and names first. If there are multiple versions of a library, and you would prefer tiff over tiff2, make sure you list them in that order.

5.2 System Properties

Although it is a common practice in C and C++ code to add platform-specific code inside preprocessor `ifdef` directives, for maximum portability this should be avoided. Software should not be tuned to specific platforms with `ifdefs`, but rather to a canonical system consisting of a set of features. Coding to specific systems makes the software less portable, because systems and the features they support change with time, and even from system to system. A feature that may not have worked on a platform in the past may be a required feature for the platform in the future. The following code fragments illustrate the difference between coding to a canonical system and a specific system:

```
// coding to a feature
#ifdef HAS_FOOBAR_CALL
  foobar();
#else
  myfoobar();
#endif

// coding to specific platforms
#if defined(SUN) && defined(HPUX) && !defined(GNUC)
  foobar();
#else
  myfoobar();
```

```
#endif
```

The problem with the second approach is that the code will have to be modified for each new platform on which the software is compiled. For example, a future version of SUN may no longer have the foobar call. Using the HAS_FOOBAR_CALL approach, the software will work as long as HAS_FOOBAR_CALL is defined correctly, and this is where CMake can help. CMake can be used to define HAS_FOOBAR_CALL correctly and automatically by making use of the try_compile and try_run commands. These commands can be used to compile and run small test programs during the CMake configure step. The test programs will be sent to the compiler that will be used to build the project, and if errors occur the feature can be disabled. These commands require that you write a small C or C++ program to test the feature. For example, to test if the foobar call is provided on the system, try compiling a simple program that uses foobar. First write the simple test program (testNeedFoobar.c in this example) and then add the CMake calls to the CMakeLists file to try compiling that code. If the compilation works then HAS_FOOBAR_CALL will be set to true.

```
--- testNeedFoobar.c -----

#include <foobar.h>
main()
{
  foobar();
}
```

```
--- testNeedFoobar.cmake ---

try_compile (HAS_FOOBAR_CALL
  ${CMAKE_BINARY_DIR}
  ${PROJECT_SOURCE_DIR}/testNeedFoobar.c
  )
```

Now that HAS_FOOBAR_CALL is set correctly in CMake you can use it in your source code through either the add_definitions command or by configuring a header file. We recommend configuring a header file as that file can be used by other projects that depend on your library. This is discussed further in section 5.6.

Sometimes, just compiling a test program is not enough. In some cases, you may actually want to compile and run a program to get its output. A good example of this is testing the byte order of a machine. The following example shows how you can write a small program that CMake will compile and then run to determine the byte order of a machine.

```
---- TestByteOrder.c ------

int main () {
  /* Are we most significant byte first or last */
  union
  {
    long l;
    char c[sizeof (long)];
  } u;
  u.l = 1;
  exit (u.c[sizeof (long) - 1] == 1);
}
```

```
----- TestByteOrder.cmake-----

try_run (RUN_RESULT_VAR
  COMPILE_RESULT_VAR
  ${CMAKE_BINARY_DIR}
  ${PROJECT_SOURCE_DIR}/Modules/TestByteOrder.c
  OUTPUT_VARIABLE OUTPUT
  )
```

The return result of the run will go into `RUN_RESULT_VAR` and the result of the compile will go into `COMPILE_RESULT_VAR`, and any output from the run will go into `OUTPUT`. You can use these variables to report debug information to the users of your project.

For small test programs the `FILE` command with the `WRITE` option can be used to create the source file from the CMakeLists file. The following example tests the C compiler to verify that it can be run.

```
file (WRITE
  ${CMAKE_BINARY_DIR}/CMakeTmp/testCCompiler.c
  "int main(){return 0;}"
  )

try_compile (CMAKE_C_COMPILER_WORKS
  ${CMAKE_BINARY_DIR}
  ${CMAKE_BINARY_DIR}/CMakeTmp/testCCompiler.c
  OUTPUT_VARIABLE OUTPUT
  )
```

There are several predefined try-run and try-compile macros in the `CMake/Modules` directory, some of which are listed below. These macros allow some common checks to be performed without having to create a source file for each test. For detailed documentation or to see how these macros work look at the implementation files for them in the `CMake/Modules` directory of your CMake installation. Many of these macros will look at the current value of the `CMAKE_REQUIRED_FLAGS` and `CMAKE_REQUIRED_LIBRARIES` variables to add additional compile flags or link libraries to the test.

CheckFunctionExists.cmake

Checks to see if a C function is on a system. This macro takes two arguments, the first is the name of the function to check for. The second is the variable to store the result into. This macro does use `CMAKE_REQUIRED_FLAGS` and `CMAKE_REQUIRED_LIBRARIES` if they are set.

CheckIncludeFile.cmake:

Checks for an include file on a system. This macro takes two arguments. The first is the include file to look for and the second is the variable to store the result into. Additional CFlags can be passed in as a third argument or by setting `CMAKE_REQUIRED_FLAGS`.

CheckIncludeFileCXX.cmake

Check for an include file in a C++ program. This macro takes two arguments. The first is the include file to look for and the second is the variable to store the result into. Additional CFlags can be passed in as a third argument.

CheckIncludeFiles.cmake

Check for a group of include files. This macro takes two arguments. The first is the include files to look for and the second is the variable to store the result into. This macro does use `CMAKE_REQUIRED_FLAGS` if it is set. This macro is useful when a header file you are interested in checking for is dependent on including another header file first.

CheckLibraryExists.cmake

Check to see if a library exists. This macro takes four arguments; the first is the name of the library to check for. The second is the name of a function that should be in that library. The third argument is the location of where the library should be found. The fourth argument is a variable to store the result into. This macro uses `CMAKE_REQUIRED_FLAGS` and `CMAKE_REQUIRED_LIBRARIES` if they are set.

CheckSymbolExists.cmake

Check to see if a symbol is defined in a header file. This macro takes three arguments. The first argument is the symbol to look for. The second argument is a list of header files to try including. The third argument is where the result is stored.

This macro uses CMAKE_REQUIRED_FLAGS and CMAKE_REQUIRED_LIBRARIES if they are set.

CheckTypeSize.cmake

Determines the size in bytes of a variable type. This macro takes two arguments. The first argument is the type to evaluate. The second argument is where the result is stored. Both CMAKE_REQUIRED_FLAGS and CMAKE_REQUIRED_LIBRARIES are used if they are set.

CheckVariableExists.cmake

Checks to see if a global variable exists. This macro takes two arguments. The first argument is the variable to look for. The second argument is the variable to store the result in. This macro will prototype the named variable and then try to use it. If the test program compiles then the variable exists. This will only work for C variables. This macro uses CMAKE_REQUIRED_FLAGS and CMAKE_REQUIRED_LIBRARIES if they are set.

Consider the following example that shows a variety of these modules being used to compute properties of the platform. At the beginning of the example four modules are loaded from CMake. The remainder of the example uses the macros defined in those modules to test for header files, libraries, symbols, and type sizes respectively.

```
# Include all the necessary files for macros
include (CheckIncludeFiles)
include (CheckLibraryExists)
include (CheckSymbolExists)
include (CheckTypeSize)

# Check for header files
set (INCLUDES "")
CHECK_INCLUDE_FILES ("${INCLUDES};winsock.h" HAVE_WINSOCK_H)

if (HAVE_WINSOCK_H)
  set (INCLUDES ${INCLUDES} winsock.h)
endif (HAVE_WINSOCK_H)

CHECK_INCLUDE_FILES ("${INCLUDES};io.h" HAVE_IO_H)
if (HAVE_IO_H)
  set (INCLUDES ${INCLUDES} io.h)
endif (HAVE_IO_H)

# Check for all needed libraries
set (LIBS "")
CHECK_LIBRARY_EXISTS ("dl;${LIBS}" dlopen "" HAVE_LIBDL)
```

```
if (HAVE_LIBDL)
  set (LIBS ${LIBS} dl)
endif (HAVE_LIBDL)

CHECK_LIBRARY_EXISTS ("ucb;${LIBS}" gethostname "" HAVE_LIBUCB)
if (HAVE_LIBUCB)
  set (LIBS ${LIBS} ucb)
endif (HAVE_LIBUCB)

# Add the libraries we found to the libraries to use when
# looking for symbols with the CHECK_SYMBOL_EXISTS macro
set (CMAKE_REQUIRED_LIBRARIES ${LIBS})

# Check for some functions that are used
CHECK_SYMBOL_EXISTS (socket   "${INCLUDES}" HAVE_SOCKET)
CHECK_SYMBOL_EXISTS (poll     "${INCLUDES}" HAVE_POLL)

# Various type sizes
CHECK_TYPE_SIZE (int      SIZEOF_INT)
CHECK_TYPE_SIZE (size_t   SIZEOF_SIZE_T)
```

For more advanced `try_compile` and `try_run` operations, it may be desirable to pass flags to the compiler, or to CMake. Both commands support the optional arguments `CMAKE_FLAGS` and `COMPILE_DEFINITIONS`. `CMAKE_FLAGS` can be used to pass `-DVAR:TYPE=VALUE` flags to CMake. The value of `COMPILE_DEFINITIONS` is passed directly to the compiler command line.

5.3 Finding Packages

Many software projects provide tools and libraries meant as building blocks for other projects and applications. CMake projects that depend on outside packages locate their dependencies using the `find_package` command. A typical invocation is of the form

```
find_package(<Package> [version])
```

where "`<Package>`" is the name of the package to be found, and "`[version]`" is an optional version request (of the form `major[.minor[.patch]]`). See Appendix C – Listfile Commands for the full command documentation. The command's notion of a package is distinct from that of CPack, which is meant for creating source and binary distributions and installers.

The command operates in two modes: `Module` mode and `Config` mode. In Module mode the command searches for a `find-module`: a file named "`Find<Package>.cmake`". It looks first in the `CMAKE_MODULE_PATH` and then in the CMake installation. If a find-module is found, it is loaded to search for individual components of the package. Find-modules contain package-specific knowledge of the libraries and other files they expect to find, and internally use commands like `find_library` to locate them. CMake provides find-modules for many common packages; see Appendix D – Selected Modules. Find-modules are tedious and difficult to write and maintain because they need very specific knowledge of every version of the package to be found.

The Config mode of `find_package` provides a powerful alternative through cooperation with the package to be found. It enters this mode after failing to locate a find-module or when explicitly requested by the caller. In Config mode the command searches for a `package configuration file`: a file named "`<Package>Config.cmake`" or "`<package>-config.cmake`" that is provided by the package to be found. Given the name of a package, the `find_package` command knows how to search deep inside installation prefixes for locations like

```
<prefix>/lib/<package>/<package>-config.cmake
```

(see documentation of `find_package` in Appendix C – Listfile Commands for a complete list of locations). CMake creates a cache entry called "`<Package>_DIR`" to store the location found or allow the user to set it. Since a package configuration file comes with an installation of its package, it knows exactly where to find everything provided by the installation. Once the `find_package` command locates the file it provides the locations of package components without any additional searching.

The "`[version]`" option asks `find_package` to locate a particular version of the package. In Module mode, the command passes the request on to the find-module. In Config mode the command looks next to each candidate package configuration file for a `package version file`: a file named "`<Package>ConfigVersion.cmake`" or "`<package>-config-<version>.cmake`". The version file is loaded to test whether the package version is an acceptable match for the version requested (see documentation of `find_package` for the version file API specification). If the version file claims compatibility the configuration file is accepted, otherwise it is ignored. This approach allows each project to define its own rules for version compatibility.

5.4 Built-in Find Modules

CMake has many predefined modules that can be found in the `Modules` subdirectory of CMake. The modules can find many common software packages. See Appendix D – Selected Modules for a detailed list.

Each `Find<XX>.cmake` module defines a set of variables that will allow a project to use the software package once it is found. Those variables all start with the name of the software being found <XX>. With CMake we have tried to establish a convention for naming these variables, but you should read the comments at the top of the module for a more definitive answer. The following variables are used by convention when needed:

<XX>_INCLUDE_DIRS

Where to find the package's header files, typically <XX>.h, etc.

<XX>_LIBRARIES

The libraries to link against to use <XX>. These include full paths.

<XX>_DEFINITIONS

Preprocessor definitions to use when compiling code that uses <XX>.

<XX>_EXECUTABLE

Where to find the <XX> tool that is part of the package.

<XX>_<YY>_EXECUTABLE

Where to find the <YY> tool that comes with <XX>.

<XX>_ROOT_DIR

Where to find the base directory of the installation of <XX>. This is useful for large packages where you want to reference many files relative to a common base (or root) directory.

<XX>_VERSION_<YY>

Version <YY> of the package was found if true. Authors of find modules should make sure at most one of these is ever true. For example TCL_VERSION_84

<XX>_<YY>_FOUND

If false, then the optional <YY> part of <XX> package is not available.

<XX>_FOUND

Set to false, or undefined, if we haven't found or don't want to use <XX>.

Not all of the variables are present in each of the `FindXX.cmake` files. However, the `<XX>_FOUND` should exist under most circumstances. If `<XX>` is a library, then `<XX>_LIBRARIES` should also be defined, and `<XX>_INCLUDE_DIR` should usually be defined.

Modules can be included in a project either with the `include` command or the `find_package` command.

```
find_package(OpenGL)
```

is equivalent to

```
include(${CMAKE_ROOT}/Modules/FindOpenGL.cmake)
```

and

```
include(FindOpenGL)
```

If the project converts over to CMake for its build system, then the `find_package` will still work if the package provides a <XX>`Config.cmake` file. How to create a CMake package is described in section 5.7.

5.5 How to Pass Parameters to a Compilation?

Once you have determined all features of the system in which you are interested, it is time to configure the software based on what has been found. There are two common ways to pass this information to the compiler: on the compile line, and using a preconfigured header. The first way is to pass definitions on the compile line. A preprocessor definition can be passed to the compiler from a CMakeLists file with the `add_definitions` command. For example, a common practice in C code is to have the ability to selectively compile in/out debug statements.

```
#ifdef DEBUG_BUILD
  printf("the value of v is %d", v);
#endif
```

A CMake variable could be used to turn on or off debug builds using the `OPTION` command:

```
option (DEBUG_BUILD
        "Build with extra debug print messages.")

if (DEBUG_BUILD)
  add_definitions (-DDEBUG_BUILD)
endif (DEBUG_BUILD)
```

Another example would be to tell the compiler the result of the previous HAS_FOOBAR_CALL test that was discussed earlier in this chapter. You could do this with the following:

```
if (HAS_FOOBAR_CALL)
  add_definitions (-DHAS_FOOBAR_CALL)
endif (HAS_FOOBAR_CALL)
```

If you want to pass preprocessor definitions at a finer level of granularity, you can use the COMPILE_DEFINITIONS property that is defined for directories, targets, and source files. For example, the code

```
add_library (mylib src1.c src2.c)
add_executable (myexe main1.c)
set_property (
   DIRECTORY
   PROPERTY COMPILE_DEFINITIONS A AV=1
   )
set_property (
   TARGET mylib
   PROPERTY COMPILE_DEFINITIONS B BV=2
   )
set_property (
   SOURCE src1.c
   PROPERTY COMPILE_DEFINITIONS C CV=3
   )
```

will build the source files with these definitions:

```
src1.c:   -DA -DAV=1 -DB -DBV=2 -DC -DCV=3
src2.c:   -DA -DAV=1 -DB -DBV=2
main2.c:  -DA -DAV=1
```

When the add_definitions command is called with flags like "-DX" the definitions are extracted and added to the current directory's COMPILE_DEFINITIONS property. When a new subdirectory is created with add_subdirectory the current state of the directory-level property is used to initialize the same property in the subdirectory.

Note in the above example that the set_property command will actually set the property and replace any existing value. The command provides the APPEND option to add more definitions without removing existing ones. For example, the code

```
set_property (
   SOURCE src1.c
   APPEND PROPERTY COMPILE_DEFINITIONS D DV=4
   )
```

will add the definitions "-DD -DDV=4" when building `src1.c`. Definitions may also be added on a per-configuration basis using the `COMPILE_DEFINITIONS_<CONFIG>` property. For example, the code

```
set_property (
   TARGET mylib
   PROPERTY COMPILE_DEFINITIONS_DEBUG MYLIB_DEBUG_MODE
   )
```

will build sources in mylib with `-DMYLIB_DEBUG_MODE` only when compiling in a `Debug` configuration.

The second approach for passing definitions to source code is to configure a header file. For maximum portability of a toolkit, it is recommended that `-D` options are not required for the compiler command line. Instead of command line options, CMake can be used to configure a header file that applications can include. The header file will include all of the `#define` macros needed to build the project. The problem with using compile line definitions can be seen when building an application that in turn uses a library. If building the library correctly relies on compile line definitions, then chances are that an application that uses the library will also require the exact same set of compile line definitions. This puts a large burden on the application writer to make sure they add the correct flags to match the library. If instead the library's build process configures a header file with all of the required definitions, then any application that uses the library will automatically get the correct definitions when that header file is included. A definition can often change the size of a structure or class, and if the macros are not exactly the same during the build process of the library and the application linking to the library, the application may reference the "wrong part" of a class or struct and crash unexpectedly.

5.6 How to Configure a Header File

Hopefully we have convinced you that configured header files are the right choice for most software projects. To configure a file with CMake the `configure_file` command is used. This command requires an input file that is parsed by CMake which then produces an output file with all variables expanded or replaced. There are three ways to specify a variable in an input file for `configure_file`.

```
#cmakedefine VARIABLE
```

If VARIABLE is true, then the result will be:

```
#define VARIABLE
```

If VARIABLE is false, then the result will be:

```
/* #undef VARIABLE */
```

${VARIABLE}

This is simply replaced by the value of VARIABLE.

@VARIABLE@

This is simply replaced by the value of VARIABLE.

Since the `${}` syntax is commonly used by other languages, there is a way to tell the `configure_file` command to only expand variables using the `@var@` syntax. This is done by passing the `@ONLY` option to the command. This is useful if you are configuring a script that may contain `${var}` strings that you want to preserve. This is important because CMake will replace all occurrences of `${var}` with the empty string if `var` is not defined in CMake.

The following example configures a .h file for a project that contains preprocessor variables. The first definition indicates if the `FOOBAR` call exists in the library, and the next one contains the path to the build tree.

```
---- CMakeLists.txt file-----

# Configure a file from the source tree
# called projectConfigure.h.in and put
# the resulting configured file in the build
# tree and call it projectConfigure.h
configure_file (
  ${PROJECT_SOURCE_DIR}/projectConfigure.h.in
  ${PROJECT_BINARY_DIR}/projectConfigure.h)
```

```
-----projectConfigure.h.in file------
/* define a variable to tell the code if the */
/* foobar call is available on this system */
#cmakedefine HAS_FOOBAR_CALL

/* define a variable with the path to the */
/* build directory  */
#define PROJECT_BINARY_DIR "${PROJECT_BINARY_DIR}"
```

It is important to configure files into the binary tree, not the source tree. A single source tree may be shared by multiple build trees or platforms. By configuring files into the binary tree the differences between builds or platforms will be kept isolated in the build tree, where it will not corrupt other builds. This means that you will need to include the directory of the build tree where you configured the header file into the project's list of include directories using the `include_directories` command.

5.7 Creating CMake Package Configuration Files

Projects must provide package configuration files so that outside applications can find them. Consider a simple project "Gromit" providing an executable to generate source code and a library against which the generated code must link. The `CMakeLists.txt` file might start with:

```
cmake_minimum_required (VERSION 2.6.3)
project (Gromit C)
set (version 1.0)

# Create library and executable.
add_library (gromit STATIC gromit.c gromit.h)
add_executable (gromit-gen gromit-gen.c)
```

In order to install Gromit and export its targets for use by outside projects, one adds the code

```
# Install and export the targets.
install (FILES gromit.h DESTINATION include/gromit-${version})
install (TARGETS gromit gromit-gen
         DESTINATION lib/gromit-${version}
         EXPORT gromit-targets)
install (EXPORT gromit-targets
         DESTINATION lib/gromit-${version})
```

as described in Section 0. Finally, Gromit must provide a package configuration file in its installation tree so that outside projects can locate it with `find_package`:

```
# Create and install package configuration and version files.
configure_file (
   ${Gromit_SOURCE_DIR}/pkg/gromit-config.cmake.in
   ${Gromit_BINARY_DIR}/pkg/gromit-config.cmake @ONLY)

configure_file (
   ${Gromit_SOURCE_DIR}/gromit-config-version.cmake.in
   ${Gromit_BINARY_DIR}/gromit-config-version.cmake @ONLY)

install (FILES ${Gromit_BINARY_DIR}/pkg/gromit-config.cmake
         ${Gromit_BINARY_DIR}/gromit-config-version.cmake
         DESTINATION lib/gromit-${version})
```

This code configures and installs the package configuration file and a corresponding package version file. The package configuration input file `gromit-config.cmake.in` has the code,

```
# Compute installation prefix relative to this file.
get_filename_component (_dir "${CMAKE_CURRENT_LIST_FILE}" PATH)
get_filename_component (_prefix "${_dir}/../.." ABSOLUTE)

# Import the targets.
include ("${_prefix}/lib/gromit-@version@/gromit-targets.cmake")

# Report other information.
set (gromit_INCLUDE_DIRS "${_prefix}/include/gromit-@version@")
```

After installation the configured package configuration file `gromit-config.cmake` knows the locations of other installed files relative to itself. The corresponding package version file is configured from its input file `gromit-config-version.cmake.in` which contains code such as,

```
set (PACKAGE_VERSION "@version@")
if (NOT "${PACKAGE_FIND_VERSION}" VERSION_GREATER "@version@")
  set (PACKAGE_VERSION_COMPATIBLE 1) # compatible with older
  if ("${PACKAGE_FIND_VERSION}" VERSION_EQUAL "@version@")
    set (PACKAGE_VERSION_EXACT 1) # exact match for this version
  endif ()
endif ()
```

An application that uses the Gromit package might create a CMake file that looks like this:

```
cmake_minimum_required (VERSION 2.6.3)
project (MyProject C)

find_package (gromit 1.0 REQUIRED)
include_directories (${gromit_INCLUDE_DIRS})
# run imported executable
add_custom_command (OUTPUT generated.c
                    COMMAND gromit-gen generated.c)
add_executable (myexe generated.c)
target_link_libraries (myexe gromit) # link to imported library
```

The call to `find_package` locates an installation of Gromit or terminates with an error message if none can be found (due to `REQUIRED`). After the command succeeds, the Gromit package configuration file `gromit-config.cmake` has been loaded, so Gromit targets have been imported and variables like `gromit_INCLUDE_DIRS` have been defined.

The above example creates a package configuration file and places it in the `install` tree. One may also create a package configuration file in the `build` tree to allow applications to use the project without installation. In order to do this, one extends Gromit's CMake file with the code:

```
# Make project useable from build tree.
export (TARGETS gromit gromit-gen FILE gromit-targets.cmake)
configure_file (${Gromit_SOURCE_DIR}/gromit-config.cmake.in
                ${Gromit_BINARY_DIR}/gromit-config.cmake @ONLY)
```

This `configure_file` call uses a different input file, `gromit-config.cmake.in`, containing

```
# Import the targets.
include("@Gromit_BINARY_DIR@/gromit-targets.cmake")

# Report other information.
set(gromit_INCLUDE_DIRS "@Gromit_SOURCE_DIR@")
```

The package configuration file `gromit-config.cmake` placed in the build tree provides the same information to an outside project as that in the install tree, but refers to files in the source and build trees. It shares an identical package version file `gromit-config-version.cmake` with that placed in the install tree.