# EcoNode Series Gateways

## C/C++ Development Guide

## Setting Up a PC Development Environment

Table of Contents

# 4 Steps to Set Up a Development Environment

Software Development Using PC-Based Coding, Cross-Compilation, and Remote Debugging

Software development uses PC-based coding, cross-compilation, and remote debugging. Before starting development, the following steps are needed to prepare the development environment:

1. Preparing the operating system
2. Choosing an IDE or editor
3. Installing cross-compilation tools
4. Porting common open source libraries

## 4.1 Operating System Requirements

The provided compilation and debugging toolchain runs on a 64-bit Linux operating system. Therefore, you need to prepare a 64-bit Linux OS. You can choose one of the following methods. It is recommended to use Ubuntu 20.04 or 22.04 for installation.

When installing on physical or virtual machines, download the installation image from the Ubuntu official website. You can download it from the following URL: https://cn.ubuntu.com/download.

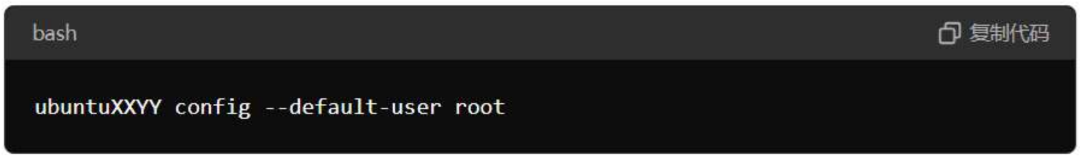
Choose either the desktop version or the server version based on your usage needs. The main difference is that the desktop version includes a GUI interface, while the server version does not.

### 4.1.1 WSL (Windows Subsystem for Linux)

First, ensure your Windows version is Windows 10 or later. If this condition is met, you can install WSL to provide a Linux environment. Currently, there are two versions of WSL: WSL 1 supports serial ports, but WSL 2 does not. If programming and debugging require validation on the PC, carefully consider whether to use this method. The installation process is as follows:

1. Enable Developer Mode:
   - In "Turn Windows features on or off," enable "Windows Subsystem for Linux" and "Virtual Machine Platform."
2. Install Linux:
   - Choose Ubuntu from the Microsoft Store to install. It is recommended to install the LTS version.
3. Initialize Ubuntu:
   - After installation, access and initialize Ubuntu through the Start Menu.
4. Access Ubuntu:
   - After initialization, you can access it via PowerShell, CMD, or the Start Menu. In PowerShell or CMD, type bash to enter the system. To exit, type exit.
5. Set Default User to Root:

For development, root access might be required frequently. Change the default user to root with the command:



```bash
ubuntuXXYY config --default-user root
```

Replace XXYY with your Ubuntu version (e.g., for Ubuntu 22.04, use ubuntu2204 config --default-user root).

6. View Other Parameters:
   - You can view other parameters with:

bash 复制代码
```bash
ubuntuXXYY --help
```

### 4.1.2 Virtual Machine

When using a virtual machine for installation, the physical machine must have sufficient hardware resources to ensure smooth operation of the virtual machine. VMware is recommended for virtual machine installation. This guide uses VMware as an example to explain the installation process and Ubuntu setup.

1. **Install VMware**:
   - Download and install VMware Workstation or VMware Player from the official VMware website.
2. **Create a New Virtual Machine**:
   - Open VMware and choose to create a new virtual machine. Follow the prompts to configure the virtual machine's settings.
3. **Configure Virtual Machine Settings**:
   - Allocate sufficient CPU, memory, and disk space to the virtual machine to ensure optimal performance.
4. **Install Ubuntu**:
   - Attach the Ubuntu ISO file to the virtual machine and boot from it. Follow the on-screen instructions to install Ubuntu.
5. **Complete Ubuntu Installation**:
   - After installation, configure Ubuntu as needed and ensure that VMware Tools or equivalent is installed for better integration between the host and the virtual machine.

By following these steps, you can successfully set up a virtual machine environment using VMware and install Ubuntu for development purposes.

### 4.1.3 Physical Machine

For installing the operating system on a physical machine, you need to create an installation CD or USB drive. Users can use tools like SoftDisc to create a bootable USB drive. The installation process is as follows:

1. **Open the Image File**:
   - Use SoftDisc or a similar tool to open the operating system image file.
2. **Write the Image to the USB Drive**:
   - Transfer the image file to the USB drive, making it bootable.
3. **Modify Boot Order**:
   - Adjust the boot order in the BIOS or UEFI settings of the physical machine to boot from the USB drive.
4. **Follow Installation Prompts**:
   - Boot from the USB drive and follow the on-screen instructions to complete the operating system installation.

By following these steps, you can prepare a physical machine for OS installation using a bootable USB drive.

### 4.1.4 Installing Basic Development Environment and Tools

During development, you might need auxiliary tools such as CMake, Autotools, Python, Perf, and Valgrind. In some cases, running programs in a PC environment for debugging purposes may also require appropriate tools.

**CMake**

CMake is a cross-platform build system generator that allows you to describe the build process using simple statements. It generates various types of makefiles or project files and can test the C++ features supported by the compiler, similar to Unix's automake. CMake's configuration file is named CMakeLists.txt.

Unlike directly building the final software, CMake produces standard build files (like Unix Makefiles or Windows Visual C++ projects/workspaces) which are then used in the usual build process. This enables developers familiar with a particular Integrated Development Environment (IDE) to build software in a standard way, distinguishing it from other systems like SCons.

CMake supports both in-place builds (where binaries and source code are in the same directory tree) and out-of-place builds (where binaries are in a different directory), making it easy to build multiple binaries from the same source tree. It also supports building static and dynamic libraries. Despite its name, which includes "make," CMake operates separately from the traditional Unix "make" system and offers a higher level of abstraction.

To install CMake, use the following command:

复制代码

```
apt install cmake
```

After installation, you can verify it with:

css 复制代码

```css
cmake --version
```

### Installing autoconf

Autoconf is part of the GNU build system, which uses scripts and make programs to build software on specific platforms. The typical process involves three steps: configure, make, and make install. This approach has become a standard practice and is widely used.

To simplify portable builds, the AutoTools suite was developed to assist programmers in constructing software. The configure, make, and make install steps are often based on AutoTools, which is the standard build system for GNU programs.

To install Autoconf, use the following command:

复制代码

```
apt install libtool autoconf
```

After installation, you can verify it with:

css 复制代码

```css
autoconf --version
```

### Installing build-essential

build-essential is a package in Ubuntu that includes a collection of compilers and tools for C, C++, Objective-C, Fortran, Ada, Go, and D programming languages. It is not installed by default in Ubuntu.

To install the build-essential package, use the following command:

```
apt install build-essential
```

After installation, you can verify it with:

```
gcc -v
```

**rockBulk Overview**

**rockBulk** is a suite of PC tools developed by our company, designed to streamline device management, maintenance, and software deployment. The suite includes the following tools:

1. **deviceScanner**: A local network discovery tool that periodically broadcasts a device's network and operational status. The PC receives and parses these broadcasts to identify devices within the local network, allowing direct access to device configuration pages or remote login for operations.
2. **smartTunnel**: A remote maintenance software with business data tunneling capabilities. It uses OpenVPN for P2P communication and direct network card access to facilitate cross-segment communication with gateway and subordinate devices. It's useful for remote PLC programming and debugging. In VPN ONLY mode, users can access remote gateway web pages and use rockTerminal for remote gateway login.
3. **rockTerminal**: A remote login software supporting telnet and SSH protocols, with additional support for the Zmodem protocol for file transfers. It integrates with deviceScanner and smartTunnel, allowing seamless remote logins. Key features include:
   - SSH and telnet remote login
   - Zmodem file transfer
   - Address book for frequently used server addresses
   - Text copy and search engine queries
   - Command history browsing with mouse scroll
   - English and Chinese interfaces
4. **fotaMan**: A separate tool for differential package management, essential for over-the-air (OTA) upgrades via our device management platform. It handles software packaging, verification, release, and upgrade control.
5. **Virtual Serial Port Tool**: Uses virtual serial port drivers to create virtual serial port pairs on the operating system. One end (with a lower number) is exposed to user programs, while the other (with a higher number) is used for network communication. Features include:
   - Adding, deleting, and modifying virtual serial ports
   - Virtual serial port service control (start, stop)
   - TCP and UDP client/server modes
   - Heartbeat packet functionality
   - Registration packet functionality with various sending options
6. **rockFtpD**: A simple FTP server program supporting both active and passive modes, easily configurable and capable of interacting with our device management platform using a public account.

Software can be obtained from our official website or by contacting our sales team.

## 4.2 Coding Tools

Below is an overview of several coding tools, along with their features and installation methods.

### Vim

Vim is a text editor that evolved from vi. It's well-known among programmers for its extensive features such as code completion, compilation, and error jumping. It stands alongside Emacs as one of the most popular text editors among Unix-like system users. Vim's design philosophy revolves around combining commands. By learning and flexibly using various movement and editing commands, Vim can be more efficient than editors without modes. It also shares many shortcuts and regular expressions that aid memory. Vim is optimized for programmers.

**Installation:**

- **Windows:** The distribution version, gVim, can be easily obtained from Tencent Computer Manager, 360 Software Market, or the official Vim website.
- **Ubuntu:** Use the following command to install:

arduino 复制代码

```arduino
apt-get install vim-gnome
```

If the machine doesn't run a GUI, install the non-GUI version with:

arduino 复制代码

```arduino
apt-get install vim
```

**Basic Commands:**

- :w - Save changes
- :wq - Save changes and exit
- :q - Exit (use :q! to force quit)
- h, j, k, l - Move the cursor (left, down, up, right)

Vim is powerful but has a steep learning curve, requiring memorization of many shortcuts and frequent mode switching between command and edit modes. However, learning basic commands is useful for editing documents in gateway environments during embedded development.

**Visual Studio Code (VSCode)**

VSCode is a cross-platform source code editor developed by Microsoft, announced at the 2015 Build developer conference. It runs on Windows, macOS, and Linux and is designed for modern web and cloud applications. VSCode supports JavaScript, TypeScript, Node.js, and many other languages and runtimes through an extensive extension ecosystem.

**Installation:**

- **Windows:** VSCode can be installed via Tencent Computer Manager, 360 Software Market, or downloaded from the official website.

VSCode is an excellent coding tool, but it has a significant drawback of high resource consumption. For programmers who do not wish to memorize shortcuts and have ample computer resources, VSCode is very convenient.

**Emacs**

Emacs, another powerful text editor and IDE, is highly regarded among professional programmers. It was initially developed by Richard Stallman in 1975 at MIT and has evolved into numerous branches, with GNU Emacs and XEmacs being the most widespread.

**Features:**

- Emacs Lisp, a highly extensible programming language.
- Full-featured environment, often considered an integrated development environment (IDE).
- Built-in features include email, FTP editing, remote login, calendar management, task management, and more.

**Installation:**

- **Windows/Linux:** Download from GNU's official website or install from software markets.

Although Emacs requires memorizing a large number of shortcuts, it doesn't differentiate between command and edit modes like Vim. For those willing to invest time, Emacs offers a highly efficient and comprehensive coding environment.

**QtCreator**

QtCreator is a cross-platform integrated development environment (IDE) designed to make development with the Qt application framework faster and easier. It integrates with Clang to check syntax during editing and supports CMake project management, making it convenient for non-Qt development as well.

**Installation:**

- Download QtCreator from Tsinghua University's mirror.
- For Windows users who prefer gcc, set up a gcc development environment with msys2, downloadable from msys2.org.

Compared to VSCode, QtCreator uses less memory and offers better project management. It is also lighter and has better cross-platform capabilities than Visual Studio. If you prefer using an IDE for development and project management, QtCreator is highly recommended.

**Summary**

While Vim, Notepad++, Emacs, and VSCode can manage projects, they lack the convenience of an IDE. For those accustomed to using an IDE, QtCreator or Codelite are excellent alternatives. ISG Gateways come with Vim installed, so it's beneficial to become familiar with using Vim for editing.

## 4.3 Engineering Management Tools

This section focuses on the tools and methods for engineering management in software development. Engineering management can be conducted using IDEs or more traditional methods, such as managing with makefiles. However, in the development of gateway embedded programs, traditional methods are more recommended. Considering factors such as the compilation environment and code reuse, it is advisable to use tools like CMake and Autoconf to facilitate code reuse and cross-platform compatibility.

While managing projects with an IDE can meet the requirements of embedded development, IDE project files are often not easily transferable. Therefore, the choice of management method should be based on personal preference or specific circumstances. Here, we use QtCreator as an example for IDE-based project management.

### 4.3.1 IDE Project Management

QtCreator is used as an example of IDE project management because it is smaller in size compared to Visual Studio and easier to configure. Developers working on Linux often find it more convenient. It is recommended to use CMake whenever possible, though qmake can also be used. However, if qmake is chosen, it can only be used within the Qt build environment.

### 4.3.2 Makefile Management of Projects

Using Makefile is a traditional method for managing projects in Linux. It allows for the management of the toolchain, header files, directory information, source files, and output methods required for the compilation process. Makefile enables flexible control over preprocessing, compiling, and linking processes through the use of variables and scripts.

#### 4.3.2.1 Makefile Structure

- **Variables**: Define strings, similar to macros in C, that expand to their referenced locations upon execution.
- **Explicit Rules**: Specify how to generate target files, including dependencies and commands.
- **Implicit Rules**: Leverage make's automatic derivation capabilities for simplified Makefile writing.
- **File Directives**: Include other Makefiles, similar to C's #include.
- **Comments**: Use # for comments, akin to C/C++'s //.

#### 4.3.2.2 Variable Definitions

- **Predefined Variables**: Useful shortcuts like $@ (target name), $^ (all dependencies), etc.
- **Custom Variables**: Defined using =, :=, ?=, += for different assignment behaviors.

#### 4.3.2.3 Targets and Dependencies

Defines how targets depend on each other, specifying commands to be executed.

#### 4.3.2.4 Phony Targets

Used for labels that are not actual files, marked with .PHONY to avoid conflicts with files of the same name.

#### 4.3.2.5 Functions

- **String Functions**: Manipulate strings, e.g., $(subst from,to,text).
- **Filename Functions**: Handle file paths and names, e.g., $(basename names…).

#### 4.3.2.6 Shell Commands in Makefile

Shell commands can be executed within Makefile using $(shell command).

#### 4.3.2.7 Conditional Statements

Makefile supports basic conditionals using ifeq, else, and endif.

#### 4.3.2.8 Handling Multi-level Directories

Makefile can manage projects with source code spread across multiple directories by including Makefiles from subdirectories.

### 4.3.3 Basic Process of Using CMake

The process of using CMake to generate a Makefile and compile a project is as follows:

1. **Write the CMake Configuration File**: Create a CMakeLists.txt file to define the build configuration.
2. **Generate the Makefile**: Run the command cmake PATH or ccmake PATH to generate the Makefile, where PATH is the directory containing CMakeLists.txt. The

difference between ccmake and cmake is that ccmake provides an interactive interface.
3. **Compile the Project**: Use the make command to compile the project using the generated Makefile.

#### 4.3.3.1 Structure of CMakeLists.txt

The CMakeLists.txt file is composed of commands, comments, and whitespace. Commands are case-insensitive, and comments are initiated with the # symbol. Commands typically consist of a command name, parentheses, and parameters separated by spaces.

For example:

- **Comments**: Start with # and are limited to a single line.
- **CMake Commands**: e.g., cmake_minimum_required(VERSION 3.9) specifies the minimum required CMake version.

#### 4.3.3.2 Version Control in CMake

You can control the minimum required version of CMake for a project using the cmake_minimum_required command. If the CMake version running is lower than the specified version, processing stops with an error.

- **VERSION**: Specifies the required version number.
- **min**: Specifies the minimum required CMake version.
- **max**: (For versions before 3.12) Specifies the maximum version. For CMake 3.12 and later, this is ignored.

#### 4.3.3.3 Project Name with project

Define the project name with the project command. When you specify a project name, CMake assigns the following variables:

- PROJECT_NAME: The name of the current project.
- PROJECT_SOURCE_DIR: The source directory of the project.
- PROJECT_BINARY_DIR: The binary directory of the project.

#### 4.3.3.4 Building Executable and Library Targets

CMake can build executable files, shared libraries, and static libraries:

- **Executable Files**: Use add_executable to specify the name of the executable file.
- **Static Libraries**: Use STATIC with add_library.
- **Shared Libraries**: Use SHARED with add_library.

Variables in CMake can be defined and accessed as follows:

- **Defining Variables**: Use the set command.
- **Accessing Variables**: Use ${<variable>} syntax.

CMake also provides various internal and environment variables for customization, including CMAKE_C_COMPILER, CMAKE_CXX_COMPILER, CMAKE_C_FLAGS, and CMAKE_CXX_FLAGS.

#### 4.3.3.5 Example of CMake Configuration

Below is a simplified example of a CMake configuration file for a project named rockTerminal:

```
project(rockTerminal)
```

```
cmake_minimum_required(VERSION 3.9)

set(CMAKE_VERBOSE_MAKEFILE ON)

set(CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake)

if(("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU") OR
("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Clang"))

    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Werror -Wno-error=deprecated-
declarations")

    set(CMAKE_CXX_FLAGS "${CMAKE_C_FLAGS} -Werror -Wno-error=deprecated-
declarations")

endif()

add_subdirectory(src)
```

This configuration sets up the project, specifies the minimum required version of CMake, and includes necessary build flags and subdirectories.

#### 4.3.3.6 Multi-Level Directory Support

CMake supports multi-level directory compilation. Each directory should contain a CMakeLists.txt file, and subdirectories can be added using the add_subdirectory command.

#### 4.3.3.7 To generate a Makefile

Once the entire project is written, you can create a new folder named build within the project directory. Then, execute cmake to convert the CMakeLists.txt file into a Makefile that can be used with the make command. cmake can also generate other types of project files, depending on the specified generator using the -G parameter, which may vary slightly depending on the platform. Currently, cmake supports generating the following types of project files:

- Visual Studio 17 2022
- Visual Studio 16 2019
- Visual Studio 15 2017
- Visual Studio 14 2015
- Visual Studio 12 2013
- Visual Studio 11 2012
- Visual Studio 9 2008
- Borland Makefiles
- NMake Makefiles
- NMake Makefiles JOM
- MSYS Makefiles
- MinGW Makefiles
- Green Hills MULTI
- Unix Makefiles
    - 
    - Ninja
- Ninja Multi-Config
- Watcom WMake
- CodeBlocks - MinGW Makefiles
- CodeBlocks - NMake Makefiles
- CodeBlocks - NMake Makefiles JOM
- CodeBlocks - Ninja
- CodeBlocks - Unix Makefiles
- CodeLite - MinGW Makefiles
- CodeLite - NMake Makefiles
- CodeLite - Ninja
- CodeLite - Unix Makefiles
- Eclipse CDT4 - NMake Makefiles

- Eclipse CDT4 - MinGW Makefiles
- Eclipse CDT4 - Ninja
- Eclipse CDT4 - Unix Makefiles
- Kate - MinGW Makefiles
- Kate - NMake Makefiles
- Kate - Ninja
- Kate - Unix Makefiles
- Sublime Text 2 - MinGW Makefiles
- Sublime Text 2 - NMake Makefiles
- Sublime Text 2 - Ninja
- Sublime Text 2 - Unix Makefiles

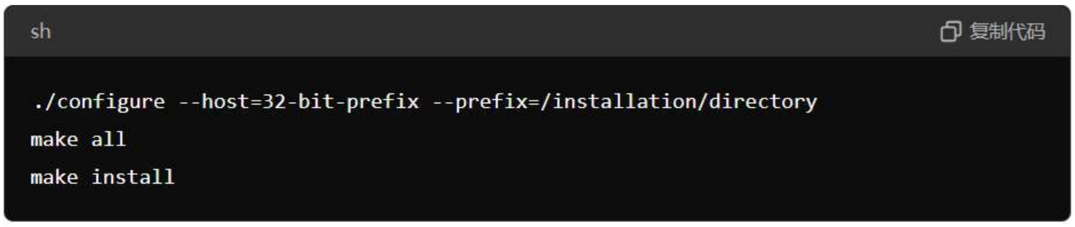
The basic command to generate project files for Visual Studio 2019 is as follows:

### 4.3.4 Managing Projects with autoconf

autoconf can be used in msys2, cygwin, and *nix development environments. Compared to cmake, autoconf has slightly less cross-platform capability but still offers powerful project management features. It is recommended to use cmake and Makefile for project management. However, since some open-source resources use autoconf to manage their projects, it's important to understand how to perform cross-compilation.

- If the project provides configure, you can automatically configure and cross-compile the project by executing the following commands:

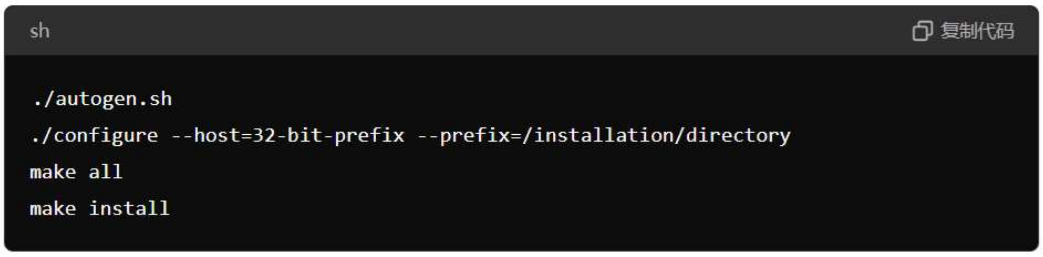
  The --host parameter in the command specifies the prefix for the cross-compilation tools. In this example, the parameter is set for a 32-bit compiler prefix. The --prefix parameter specifies the installation directory after compilation.

```sh
./configure --host=32-bit-prefix --prefix=/installation/directory
make all
make install
```

  The configure command checks and configures the project's compilation environment. make all executes the compilation process, which sometimes includes generating documentation and other tasks. Finally, make install performs the installation.

- If the project provides autogen.sh, some projects include a script like autogen.sh that converts configure.ac to configure. The compilation process for projects with this script is as follows:

```sh
./autogen.sh
./configure --host=32-bit-prefix --prefix=/installation/directory
make all
make install
```

- If the project provides configure.ac, in this case, you need to manually convert configure.ac to configure as follows:

```sh
autoreconf -i
./configure --host=32-bit-prefix --prefix=/installation/directory
make all
make install
```

## 4.4 Compilation Tools

We provide a GCC 4 toolchain that supports C++14 syntax. This toolchain requires a 64-bit Linux operating system. The toolchain is provided as a compressed package, including both 32-bit and 64-bit versions.

Before using the toolchain, you need to copy the software package to a Linux system and then extract it using the tar command. After extraction, add the environment variables accordingly.

To verify that the 32-bit compilation is working correctly, run:

```sh
# Verify 32-bit compilation
<run verification command here>
```

If the output matches the expected screenshot, the 32-bit compilation is functioning properly.

Then run:

```sh
# Verify 64-bit compilation
<run verification command here>
```

### 4.4.1 Compilation

The following command compiles a source file into an object file. To include debugging information, add the -g parameter:

```sh
gcc -c source_file.c -o object_file.o -g
```

In practical projects, manually compiling files one by one is impractical, so these commands should be integrated into the project management module.

### 4.4.2 Linking

- **Executable Files:** Use gcc to link and generate an executable file. For example:

```sh
gcc a1.o a2.o -o prog -pthread -lm -lrt -lstdc++ -O2 -s
```

  In this command:

  - a1.o and a2.o are the compiled object files.

    - -o prog specifies the output executable file.
    - -pthread, -lm, -lrt, and -lstdc++ are external libraries.
    - -O2 indicates level 2 optimization.
    - -s removes unnecessary symbol tables to reduce file size.
    - -l specifies library files, usually named libxxx.so or libxxx.a.

  To compile with debugging information, add the -g parameter:

```sh
gcc a1.o a2.o -o prog -pthread -lm -lrt -lstdc++ -g
```

- **Dynamic Libraries:** Add the -shared parameter to create a dynamic library. If you are working on Windows, you need to adjust your code accordingly.

```sh
gcc -shared -o libxxx.so source_file1.o source_file2.o
```

- **Static Libraries:** Use ar to create a static library from object files. Static libraries generally have a .a extension:

```sh
ar rcs libxxx.a source_file1.o source_file2.o
```

#### 4.4.3 Dependency Checking

After development, you may need to check program dependencies to ensure that all required dynamic libraries are included in the deployment package. Use the ldd command to check dynamic library dependencies:

```sh
ldd program_or_library
```

ldd will list the external dynamic libraries required by the program or library. Note that ldd is a script that relies on ld-linux.so for dependency checking. This functionality is platform-dependent and cannot be used in cross-compilation environments. You need to copy the program to the target machine and run the command there.

#### 4.4.4 Exporting Symbols

For symbols that need to be exported, add the __attribute__((dllexport)) declaration before the function or variable:

```cpp
__attribute__((dllexport)) void exported_function();
```