SEP Project
Development Environment

CodeCatalyst UG33

Team Documentation

Version 1.0

September 6, 2025

Contents

1 Quick Reference

Essential commands for immediate productivity.

1.1 One-Command Setup

Simple Environment Setup

```
# Install build dependencies (Ubuntu/Debian)
make install-deps

# Build everything and test
make test-all

# Generate IDE support (optional)
make compile-commands

# Script handles:
# - Essential build tools installation
# - Cross-compilation setup (MinGW)
# - Build verification and testing
```

1.2 Essential Build Commands

Main Build Commands

```
# Main development workflow
make all                          # Build main executable
make test-all                     # Build and run all tests
make clean                        # Clean build artifacts

# Windows submission
make windows-package              # Complete Windows .exe.zip creation

# Testing
make test-compression-unit        # Run algorithm unit tests
make test-integration             # Run end-to-end tests
make run-case1                    # Test with sample data
make run-case2                    # Test with sample data
```

1.3 Pre-Setup Checklist

Before Starting

Verify you have:

- Git installed and configured
- Admin/sudo access for package installation
- Internet connection for downloading dependencies
- At least 500MB free disk space
- Text editor or IDE of choice
- make utility (usually pre-installed on Linux)

2 Simple Setup (Recommended)

The fastest way to set up your development environment using Make.

2.1 Makefile-Based Setup

What the Makefile Provides

The comprehensive Makefile handles all development needs:

Dependency Management:

- Automatic dependency installation (Ubuntu/Debian)
- C++ toolchain setup (g++, make)
- Cross-compilation tools (MinGW for Windows)
- Code formatting tools (clang-format)

Build Automation:

- Linux native compilation
- Windows cross-compilation
- Comprehensive testing framework
- Clean build artifact management

IDE Support:

- compile_commands.json generation via bear
- IntelliSense and autocompletion support
- Integration with popular editors

2.2 Quick Setup Process

Simple Setup Process

```
# Navigate to project directory
cd SEP-UG-33

# Install dependencies (Ubuntu/Debian)
make install-deps

# Build and test everything
make test-all

# Generate IDE support (optional)
make compile-commands

# Expected output:
# - Dependency installation progress
# - Clean compilation
# - All tests passing
# - Success confirmation
```

Target	Description
install-deps	Install system dependencies (Ubuntu/Debian)
install-mingw	Install MinGW for Windows cross-compilation
compile-commands	Generate compile_commands.json for IDE
help	Show all available targets

Table 1: Setup-Related Makefile Targets

2.3 Makefile Targets

Setup Examples

```
# Full setup with all dependencies
make install-deps

# Just cross-compilation tools
make install-mingw

# IDE support only
make compile-commands

# See all available options
make help
```

3 Manual Setup by Platform

Step-by-step manual setup for each supported platform.

3.1 Ubuntu/Debian Setup

3.1.1 System Dependencies

Ubuntu/Debian Package Installation

```
# Automated installation via Makefile
make install-deps

# Or manual installation:
sudo apt update
sudo apt install -y build-essential mingw-w64

# Optional: Code formatting tools
sudo apt install -y clang-format bear

# Manual dependency list:
# - build-essential (g++, make)
# - mingw-w64 (Windows cross-compilation)
# - bear (compile_commands.json generation)
# - clang-format (code formatting)
```

3.1.2 Verify Installation

Ubuntu/Debian Verification

```
# Check installed versions
gcc --version            # Should be 7.0+
make --version           # Any recent version
clang-format --version   # Any recent version (optional)
bear --version           # Any recent version (optional)

# Verify cross-compilation
x86_64-w64-mingw32-g++ --version  # MinGW cross-compiler

# Test build
make test-all            # Comprehensive build and test
```

3.2 macOS Setup

3.2.1 Prerequisites

macOS Prerequisites

Required first steps:

- Install Xcode Command Line Tools: `xcode-select --install`
- Install Homebrew if not present: `https://brew.sh/`
- Ensure adequate disk space (Xcode tools are large)

3.2.2 Homebrew Package Installation

macOS Package Installation

```
# Verify Homebrew installation
brew --version

# Install development tools
brew install cmake ninja git curl

# Install code quality tools
brew install clang-format

# Note: clang-tidy comes with Xcode Command Line Tools
# Note: Cross-compilation to Windows not directly supported on macOS
```

3.2.3 Verify Installation

macOS Verification

```
# Check installed versions
clang --version         # Should be recent Xcode version
cmake --version         # Should be 3.16+
ninja --version         # Any recent version
clang-format --version  # Any recent version
```

3.3 Windows Setup

3.3.1 Option 1: Visual Studio (Recommended)

Visual Studio Installation

Install Visual Studio 2019 or later:

- Download from `https://visualstudio.microsoft.com/`
- Select "Desktop development with C++" workload
- Include CMake tools component
- Include Git for Windows component

Additional tools:

- Install CMake separately: `https://cmake.org/download/`
- Add CMake to system PATH
- Install Git for Windows if not included: `https://git-scm.com/`

3.3.2 Option 2: Command Line Tools

Windows Command Line Setup

```
# Install Chocolatey package manager (as Administrator)
Set-ExecutionPolicy Bypass -Scope Process -Force
[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.
    ServicePointManager]::SecurityProtocol -bor 3072
iex ((New-Object System.Net.WebClient).DownloadString('https://community.
    chocolatey.org/install.ps1'))

# Install development tools via Chocolatey
choco install cmake ninja git

# Install Build Tools for Visual Studio
choco install visualstudio2019buildtools --package-parameters "--add
    Microsoft.VisualStudio.Workload.VCTools"
```

3.3.3 Windows Subsystem for Linux (WSL)

WSL Alternative

For Linux-like development on Windows:

- Install WSL2 with Ubuntu distribution
- Follow Ubuntu setup instructions within WSL
- Use Windows IDE with WSL backend
- Cross-compilation to Windows works from WSL

Benefits:

- Native Linux toolchain
- Better package management
- Consistent with CI/CD environment

4 Build System

Simple and effective Makefile-based build system.

4.1 Makefile Overview

Why Makefile for This Project?

Makefile advantages for our needs:

- Simple, direct compilation control
- No external build system dependencies
- Easy cross-platform compilation
- Clear, readable build process

Perfect for this project because:

- Small codebase (4 source files)
- No external library dependencies
- Standard C++17 only
- Clear compilation requirements

4.2 Makefile Features

Comprehensive Makefile Capabilities

```
# Build targets
make all                    # Build main executable
make test                   # Build test programs
make clean                  # Clean build artifacts

# Testing targets
make test-all               # Run all tests
make test-integration       # Run end-to-end tests
make run-case1              # Test with sample data

# Cross-compilation
make windows                # Build Windows executable
make windows-package        # Complete Windows packaging

# IDE support
make compile-commands       # Generate compile_commands.json
```

4.3 IDE Integration

IDE Support via compile_commands.json

The Makefile provides IDE integration through bear:

Setup process:

- Install bear: `sudo apt install bear`
- Generate compile commands: `make compile-commands`
- Your IDE automatically detects `compile_commands.json`
- Full IntelliSense and error detection available

Supported editors:

- Visual Studio Code (with C++ extension)
- CLion (automatic detection)
- Vim/Neovim (with clangd LSP)
- Any editor supporting Language Server Protocol

5 Makefile Build Targets

Complete reference of all available build targets.

5.1 Build Target Categories

Organized Build Targets

Main categories:
- Build targets (compilation)
- Test targets (verification)
- Run targets (execution)
- Utility targets (maintenance)
- Platform targets (cross-compilation)

Benefits of comprehensive targets:
- Consistent workflow across team
- Self-documenting build process
- Automated quality assurance
- Easy CI/CD integration

5.2 Build Presets

Preset	Description
default	Basic build using Ninja generator
debug	Debug build with debugging symbols
release	Optimized release build (Linux)
vcpkg	Build with vcpkg dependency management
vcpkg-debug	Debug build with vcpkg
vcpkg-release	Release build with vcpkg
windows-mingw	Cross-compile for Windows using MinGW

Table 2: Available CMake Build Presets

5.3 Build Commands

CMake Build Workflow

```
# Configure builds (choose appropriate preset)
cmake --preset debug            # Debug configuration
cmake --preset release          # Release configuration (Linux)
cmake --preset windows-mingw    # Windows cross-compilation

# Build project
cmake --build build/release
cmake --build build/debug
cmake --build build/windows-mingw

# Clean builds
cmake --build build/release --target clean
```

5.4 Testing Integration

Testing with CTest

```
# Run all tests
ctest --test-dir build/release --output-on-failure

# Run specific test types
ctest --test-dir build/release -R "Compression"
ctest --test-dir build/release -R "Integration"

# Run tests with verbose output
ctest --test-dir build/release --verbose

# Custom test targets
cmake --build build/release --target test-all
cmake --build build/release --target run-case1
cmake --build build/release --target run-case2
```

6 IDE Configuration

Setting up popular IDEs for optimal C++ development experience.

6.1 Visual Studio Code (Recommended)

6.1.1 Required Extensions

VSCode Extension Setup

Essential extensions:
- **C/C++ Extension Pack** – Microsoft's official C++ support
- **CMake Tools** – CMake integration and IntelliSense
- **clangd** – Language server (recommended over C/C++)

Installation:
- Open VSCode Extensions (Ctrl+Shift+X)
- Search and install each extension
- Reload VSCode after installation

6.1.2 Workspace Configuration

Automatic Configuration

The setup script creates `.vscode/settings.json` with:

CMake integration:
- Configure on open enabled
- Ninja generator preference
- Build directory configuration

IntelliSense:
- clangd configuration with compile commands
- Header file associations
- Include path resolution

Code quality:
- Format on save enabled
- clang-format integration
- C++ standard configuration

6.1.3 Manual VSCode Setup

Manual VSCode Configuration

```
# Open project in VSCode
code .

# Generate compile commands for IntelliSense
cmake --preset release -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
cp build/release/compile_commands.json .

# Configure CMake Tools extension
# 1. Open Command Palette (Ctrl+Shift+P)
# 2. Run "CMake: Select a Kit"
# 3. Choose your preferred compiler
# 4. Run "CMake: Select Variant" -> Release
```

6.2 CLion

6.2.1 Project Import

CLion Setup Process

Import CMake project:

- File → Open → Select `CMakeLists.txt`
- Choose "Open as Project"
- CLion will automatically configure CMake

Toolchain configuration:

- File → Settings → Build → Toolchains
- Verify CMake and compiler paths
- Configure vcpkg toolchain if using

Code style:

- File → Settings → Editor → Code Style → C/C++
- Scheme → Import → Select project `.clang-format`
- Enable "Format code on save"

6.3 Vim/Neovim

6.3.1 LSP Configuration

Vim/Neovim Setup

```
-- For Neovim with nvim-lspconfig
require'lspconfig'.clangd.setup{
    cmd = {"clangd", "--compile-commands-dir=build"},
    filetypes = {"c", "cpp", "objc", "objcpp"},
    root_dir = require'lspconfig.util'.root_pattern(
        '.clangd',
        '.clang-tidy',
        '.clang-format',
        'compile_commands.json',
        'compile_flags.txt',
        'configure.ac',
        '.git'
    ),
}
```

Vim/Neovim Plugins

Recommended plugins:

- **nvim-lspconfig** – LSP configuration
- **nvim-cmp** – Autocompletion
- **telescope.nvim** – Fuzzy finder
- **nvim-treesitter** – Syntax highlighting

7 Verification and Testing

Ensuring your development environment is properly configured.

7.1 Environment Verification

Complete Environment Test

```
# 1. Verify all tools are installed
cmake --version     # Should be 3.16+
ninja --version     # Any recent version
clang-format --version  # Any recent version

# 2. Test CMake configuration
cmake --preset release

# 3. Test build process
cmake --build build/release

# 4. Test executable creation
ls -la build/release/block_model  # Should exist

# 5. Test with sample data
cmake --build build/release --target run-case1

# 6. Run all tests
ctest --test-dir build/release --output-on-failure

# 7. Test cross-compilation (if on Linux)
cmake --preset windows-mingw
cmake --build build/windows-mingw
ls -la build/windows-mingw/block_model.exe  # Should exist
```

7.2 Troubleshooting

7.2.1 Common Issues

Build Environment Problems

Problem: CMake not found

- Install CMake 3.16+ from official website
- Add to system PATH environment variable
- Restart terminal/IDE after installation

Problem: Ninja not found

- Install: `sudo apt install ninja-build` (Ubuntu)
- Alternative: Use Make: `cmake -G "Unix Makefiles"`
- Windows: Install via Visual Studio or Chocolatey

Problem: vcpkg integration fails

- Ensure `VCPKG_ROOT` environment variable is set
- Run: `$VCPKG_ROOT/vcpkg integrate install`
- Restart terminal to pick up environment changes

7.2.2 IDE Issues

IDE Configuration Problems

Problem: IntelliSense not working

- Generate: `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON`
- Copy: `cp build/compile_commands.json .`
- Reload IDE/window

Problem: Headers not found

- Verify `include/` directory exists
- Check CMakeLists.txt target_include_directories
- Ensure compile commands are up to date

Problem: Build configuration errors

- Delete build directory: `rm -rf build`
- Reconfigure: `cmake --preset release`
- Check for conflicting IDE configurations

7.3 Performance Optimization

Build Performance Tips

Faster builds:

- Use Ninja generator (default in presets)
- Enable parallel builds: `cmake --build build --parallel $(nproc)`
- Use ccache: `export CMAKE_CXX_COMPILER_LAUNCHER=ccache`
- Incremental builds: only rebuild changed files

Development workflow optimization:

- Use IDE's built-in build commands
- Configure editor to format on save
- Set up automatic test running on file changes
- Use build caching for faster CI/CD

8 Next Steps

8.1 After Environment Setup

Ready to Develop!

You're now ready to:

Build the project with modern CMake

Run comprehensive tests

Cross-compile for Windows submission

Use professional development tools

Follow consistent coding standards

Next recommended reading:

- `docs/coding-standards.tex` – Code style and quality guidelines
- `README.md` – Daily development command reference
- `docs/Git & Github Workflow.pdf` – Version control workflow