

# SNiFF+™

**Version 3.2 for Unix and Windows**

## **User's Guide**



**TakeFive Software, Inc.**  
Cupertino, CA  
E-mail: [info@takefive.com](mailto:info@takefive.com)

**TakeFive Software GmbH**  
5020 Salzburg, Austria  
E-mail: [info@takefive.co.at](mailto:info@takefive.co.at)

## **Copyright**

Copyright © 1992–1999 TakeFive Software Inc.

All rights reserved. TakeFive products contain trade secrets and confidential and proprietary information of TakeFive Software Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure.

## **Parts of SNIFF+:**

Copyright 1991, 1992, 1993, 1994 by Stichting Mathematisch Centrum,  
Amsterdam, The Netherlands.

Portions copyright 1991-1997 Compuware Corporation.

## **Trademarks**

SNIFF+ is a trademark of TakeFive Software Inc.

Other brand or product names are trademarks or registered trademarks of their respective holders.

## **Credits**

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

# Table of Contents

---

## Part I Guidelines

<b>About this Manual</b>	<b>3</b>
Conventions. . . . .	3
Tool elements . . . . .	4
Typography . . . . .	5
Feedback and useful links. . . . .	5
<b>SNiFF+J for Java</b>	<b>7</b>
<b>SNiFF + Basic Concepts</b>	<b>9</b>
SNiFF+ Architecture . . . . .	10
Projects . . . . .	11
Working environments. . . . .	12
Make Support . . . . .	13
Versions and configurations . . . . .	14
Documentation building. . . . .	15
Source code parsing and symbol information . . . . .	16
Cross reference subsystems. . . . .	17
Mix-and-match tool and control integration . . . . .	18

## Part II SNiFF+ Projects and Working Environments

<b>Projects</b>	<b>23</b>
Project directories and SNiFF+ generated files. . . . .	23
The contents of a project. . . . .	24
Tracking dependencies in a project. . . . .	24
Project structures. . . . .	24
Project types . . . . .	25
Organizing project structures. . . . .	26
How you would create this SNiFF+ project structure. . . . .	27
What to do next . . . . .	28
<b>Working Environments</b>	<b>29</b>
What are working environments? . . . . .	29
What types of working environments are there? . . . . .	30
Make Support and working environments. . . . .	30
Working environments and teams. . . . .	30
Shared access to your team Repository . . . . .	30
Shared and transparent access to team source code. . . . .	30
Directories for platform-specific object code . . . . .	31

Isolating individual work from the team. . . . .	31
Working on selected configurations of a team project . . . . .	32
Avoiding unnecessary builds in the PWE . . . . .	32
How file sharing works . . . . .	33
A closer look at file sharing. . . . .	34
Examples of using working environments . . . . .	36

## **Part III Setting Up SNIFF+ Projects**

### **Project Setup Overview 43**

SNIFF+ Project Setup Wizard. . . . .	43
Project setup overview — procedures . . . . .	44
Typical development situations. . . . .	46
Working with new project templates. . . . .	47
Creating a template . . . . .	47
Creating new projects using an existing template . . . . .	49
Specifying a Working Environment Configuration Directory. . . . .	51
Specifying a default working environment . . . . .	52
Initializing team working environments. . . . .	53
Initializing your team's Repository . . . . .	54
Initializing your team's SOWE . . . . .	55
Initializing a PWE . . . . .	56

### **Setting Up Team Working Environments 57**

Overview. . . . .	58
Step 1: Create root directories . . . . .	58
Step 2: Set permissions for working environment files. . . . .	58
Step 3: Create and set up team working environments . . . . .	59

### **Creating Team Projects 63**

Overview. . . . .	64
Step 1: Creating shared projects in the SSWE. . . . .	64
Step 2: Initializing your team's Repository . . . . .	65
Step 3: Initializing your team's SOWE . . . . .	66
Step 4: Initializing your PWE . . . . .	67
What you should do next . . . . .	68
Method 1 — Working in your PWE. . . . .	68
Method 2 — Working in your team's SOWE. . . . .	69

## **Part IV Setting Up the Build Process**

### **Build and Make Support 73**

Technical overview. . . . .	74
SNIFF+ Makefiles and Make Support Files . . . . .	77

Project Makefile . . . . .	77
General Makefile . . . . .	80
Make Support Files . . . . .	80
Updating Make Support Files . . . . .	80
Language Makefiles . . . . .	81
Platform Makefile . . . . .	82
Specifying the targets of a project . . . . .	85
Exporting targets of a project . . . . .	85
Building targets recursively . . . . .	88
Setting up Make Support . . . . .	88
Building purify and quantify targets (Unix only) . . . . .	100
Specifying platform-specific Make information . . . . .	101
Language Makefiles — details . . . . .	103

## **Using Your Own Makefiles 105**

Specifying Make attributes . . . . .	106
Make commands you can execute in SNIFF+ . . . . .	109

## **Make Support changes from 3.0.x to 3.1 111**

No support for VPATH . . . . .	112
Updating project Makefiles . . . . .	113
Reworked SNIFF+ Make-support files . . . . .	115
Use of pattern rules instead of suffix rules . . . . .	116
MAKE_TARGET macro . . . . .	117

# **Part V Maintaining SNIFF+ Projects**

## **Modifying SNIFF+ Projects 121**

Opening Projects . . . . .	122
Saving projects . . . . .	124
Closing projects . . . . .	124
Deleting projects . . . . .	125
General procedures for modifying projects . . . . .	126
Modifying multiple project attributes . . . . .	127
Project properties you can modify . . . . .	127
Adding and removing subprojects . . . . .	127
Adding and removing files . . . . .	129
Using the Group Project Attributes dialog . . . . .	131

## **Version Control 135**

Technical overview . . . . .	136
Locking files during check-out . . . . .	137
Notation used when referring to file versions . . . . .	137
Configurations . . . . .	138

Change sets . . . . .	138
Branches . . . . .	138
Situations for using SNIFF+'s branch support . . . . .	139
Default Configuration . . . . .	140
Executing version control commands in SNIFF+ . . . . .	141
Looking at file version history . . . . .	144
Creating your own CMVC adaptor . . . . .	146
Working with configurations . . . . .	147
Looking at and merging differences between two file versions . . . . .	151
Showing the differences between change sets . . . . .	153
Showing and merging three way differences . . . . .	153
Specifying Default Configurations. . . . .	154

## **Updating Working Environments 157**

Technical overview . . . . .	158
The Working Environments Administrator . . . . .	159
General guidelines for updating SSWEs and PWEs . . . . .	160
Updating within SNIFF+ . . . . .	163
Updating outside of SNIFF+ . . . . .	166
Unattended updates . . . . .	167

## **Part VI Compiling and debugging**

### **Preprocessing C/C++ Code in SNIFF+ 173**

Preprocessing source code . . . . .	174
Enabling full preprocessing. . . . .	174
Configuring the Parser with a configuration file . . . . .	178

### **Compiling and Debugging in SNIFF+ 185**

Building a project's targets . . . . .	185
Running a project's executable. . . . .	187
Debugging targets . . . . .	187
SNIFF+ help targets . . . . .	190

### **Introduction to Cross-Platform Development 193**

Introduction . . . . .	193
How SNIFF+ supports cross-platform development. . . . .	193
Limitations. . . . .	193
Cross-platform development vs. remote compile & debug. . . . .	195
Basic differences between Windows NT and Unix. . . . .	196

### **Setting Up Cross-Platform Development 199**

Cross-platform setup — Unix side . . . . .	199
Cross-platform setup — Windows side. . . . .	201

Setting up the shared project on Windows . . . . .	204
<b>Remote Compile and Debug</b>	<b>207</b>
Overview . . . . .	208
Requirements . . . . .	208
Scenarios. . . . .	209
Preparation . . . . .	210
Setting up remote compile and debug. . . . .	212
<b>Part VII Cross Reference Subsystems</b>	
<b>Cross Reference Information</b>	<b>219</b>
Overview . . . . .	220
Extracting symbol information . . . . .	221
How the X-Ref subsystems work. . . . .	221
Location of generated X-Ref information. . . . .	225
Working Environments and cross referencing. . . . .	226
Selecting your preferred X-Ref technology . . . . .	229
<b>Part VIII Editor Integrations</b>	
<b>Emacs Integration</b>	<b>233</b>
Integration features . . . . .	233
How the Emacs integration works . . . . .	234
Integrating Emacs . . . . .	235
Working with Emacs and SNIFF+ . . . . .	237
Command Reference . . . . .	239
<b>Vim Integration</b>	<b>243</b>
Integration Features . . . . .	243
How the Vim integration works . . . . .	244
Integrating Vim . . . . .	244
Configuring the Vim integration . . . . .	245
Working with Vim and SNIFF+. . . . .	245
Command Reference . . . . .	247
<b>Codewright Integration (Windows only)</b>	<b>251</b>
Integration Features . . . . .	251
Integrating Codewright . . . . .	252
Working with Codewright and SNIFF+. . . . .	254
Command Reference . . . . .	255
<b>MS Developer Studio Integration (Windows)</b>	<b>257</b>
Integration Features . . . . .	257

Integrating MS Developer Studio . . . . .	258
Working with MS Developer Studio and SNIFF+ . . . . .	259
Menus . . . . .	261

## **Part IX ClearCase Integration**

<b>Integrating SNIFF+ with ClearCase</b>	<b>267</b>
Integration overview . . . . .	268
Setting up a SNIFF+ project with ClearCase . . . . .	268
Working with ClearCase in SNIFF+ . . . . .	271
Advanced features . . . . .	272

## **Part X Documenting Source Code**

<b>Documenting Your Source Code</b>	<b>279</b>
Documentation Editor modes . . . . .	280
Writing source code documentation . . . . .	280
Jumping between the source code and documentation . . . . .	284
Changing the documentation status of a symbol . . . . .	285
Looking at the status of a symbols documentation. . . . .	285
Updating documentation . . . . .	286
Browsing documentation . . . . .	289
Managing documentation together with source code . . . . .	289
Exporting documentation . . . . .	289
Changing the layout of source documentation . . . . .	290
Creating documentation templates files . . . . .	290

## **Part XI Glossary and Index**

<b>Glossary</b>	<b>303</b>
<b>Index</b>	<b>309</b>



# **Part I**

## **Guidelines**



# About this Manual

---

## What this manual is

This manual is part of the SNIFF+ documentation set, which consists of:

- User's Guide
- Reference Guide
- C++ Tutorial
- C Tutorial
- Java Tutorial
- Fortran Tutorial
- Quick Reference Guide
- Release Notes, Installation Guide and Application Papers
- Online documentation of the above in HTML, PostScript and PDF formats

## Conventions

### One basic term

- **Symbol** — any programming language construct such as a class, method, etc.

### Two conventions: menu references

For clarity and to avoid undue verbosity, the phrase:

"Choose the MenuCommand from the MenuName" is presented as follows:

- Choose **MenuName > MenuCommand**.

A context menu that appears when you click the right mouse button is referred to as:

**Context menu**, and consequently:

"Choose a menu command from the context menu that appears when you click the right mouse button" is presented as follows:

- Choose **Context menu > MenuCommand**

## A note on Unix/Windows

The screenshots in this manual are all done on Windows NT. If you are working on Unix, what you see on your screen may look slightly different.

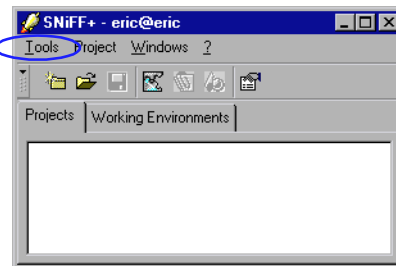
When you start SNIFF+, the first tool that appears is the Launch Pad. In this and other SNIFF+ tools, the first item in the menu bar is for launching tools.

- On **Windows**, it is called **Tools**.
- On **Unix**, it is depicted by an **Icon**.

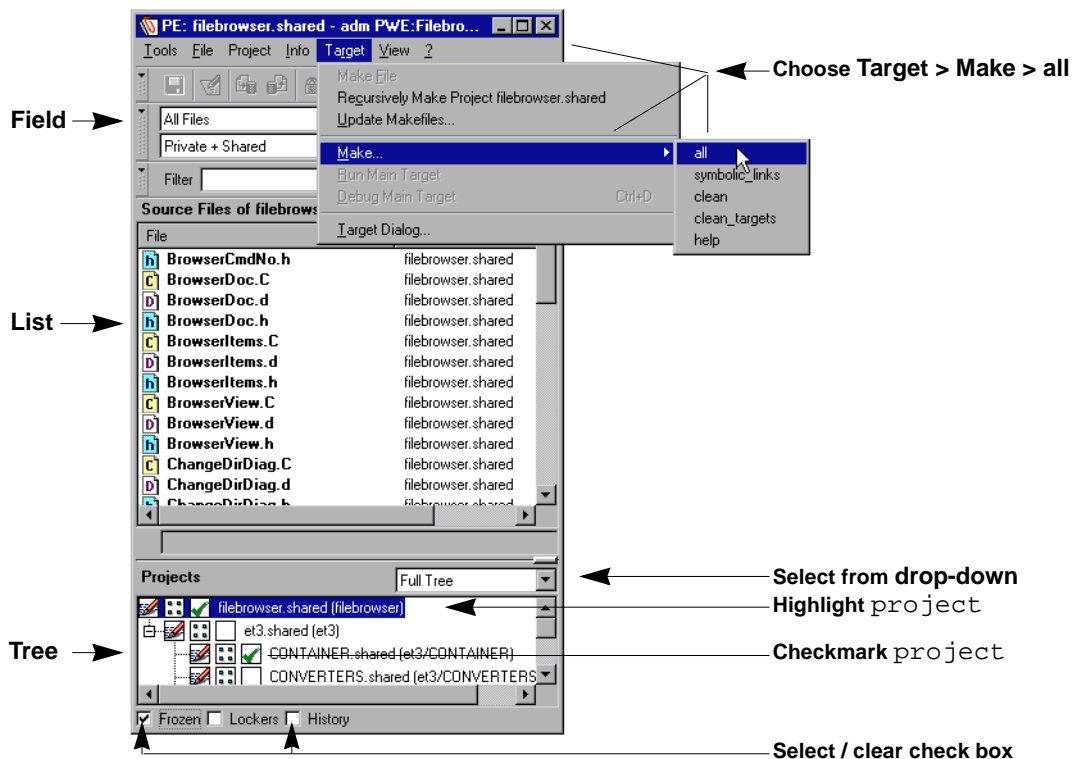
When we refer to this menu in order to launch a tool from the Launch Pad, or any other open SNIFF+ tool, we will use the notation:

Choose **Tools > ToolName**.

- On Unix a “check box” looks like a “button” (Motif Look), and a “drop-down” looks like a “pop-up”.



## Tool elements



# Typography

Capitalized Words	Names of tools, windows, dialogs and menus start with capital letters. Examples: Symbol Browser, <b>Tools</b> menu, File dialog.
<i>Italics</i>	Names of manuals and newly introduced terms are in italics. Examples: <i>User's Guide</i> , the <i>workspace</i> concept.
<b>Boldface</b> and <b><i>Bold italics</i></b>	Menu, field and button names and menu entries are printed in bold-face. Placeholders for symbols, selections or other strings in menus are in bold italics. Example: From the menu, choose <b>Show &gt; Symbol(s) <i>selection...</i></b>
Monospace	Code examples and symbol, file and directory names, as well as user entries are printed in monospace type. Examples: .login, \$PATH, class VObject. Type abc.
<Keys>	Special keys are printed in monospace type with enclosing '< >'. Examples: <CTRL>, <Return>, <Meta>.

## Feedback and useful links

Your feedback is always very welcome. Please send feedback to one of our support e-mail addresses.

### Europe:

[sniff-support@takefive.co.at](mailto:sniff-support@takefive.co.at)

### USA:

[sniff-support@takefive.com](mailto:sniff-support@takefive.com)

## Useful links

SNiFF+ web pages:

- SNiFF+ Users Mailing List  
<http://www.takefive.com/support/sniff-list.html>
- SNiFF+ Users Mailing List Archive  
<http://www.takefive.com/sniff-list>
- Frequently Asked Questions  
<http://www.takefive.com/support/faq.html>
- Customer Newsletter  
[http://www.takefive.com/news/customer\\_newsletter.html](http://www.takefive.com/news/customer_newsletter.html)



This manual relates to SNiFF+ in general.

- For Java-specific issues, please refer to the *SNiFF+ Java Tutorial*. To open the *Java Tutorial* online, choose **Help(?) > Tutorials > Java** from the Launch Pad's menu.
- Please also refer to the *Java Tutorial* to find out how to get started with the **Visaj GUI Builder** integration. SNiFF+ integration features are incorporated in the *Visaj User's Guide* under the Visaj Class Editor's **Help** menu.





## SNiFF + Basic Concepts

---

### What is SNiFF+?

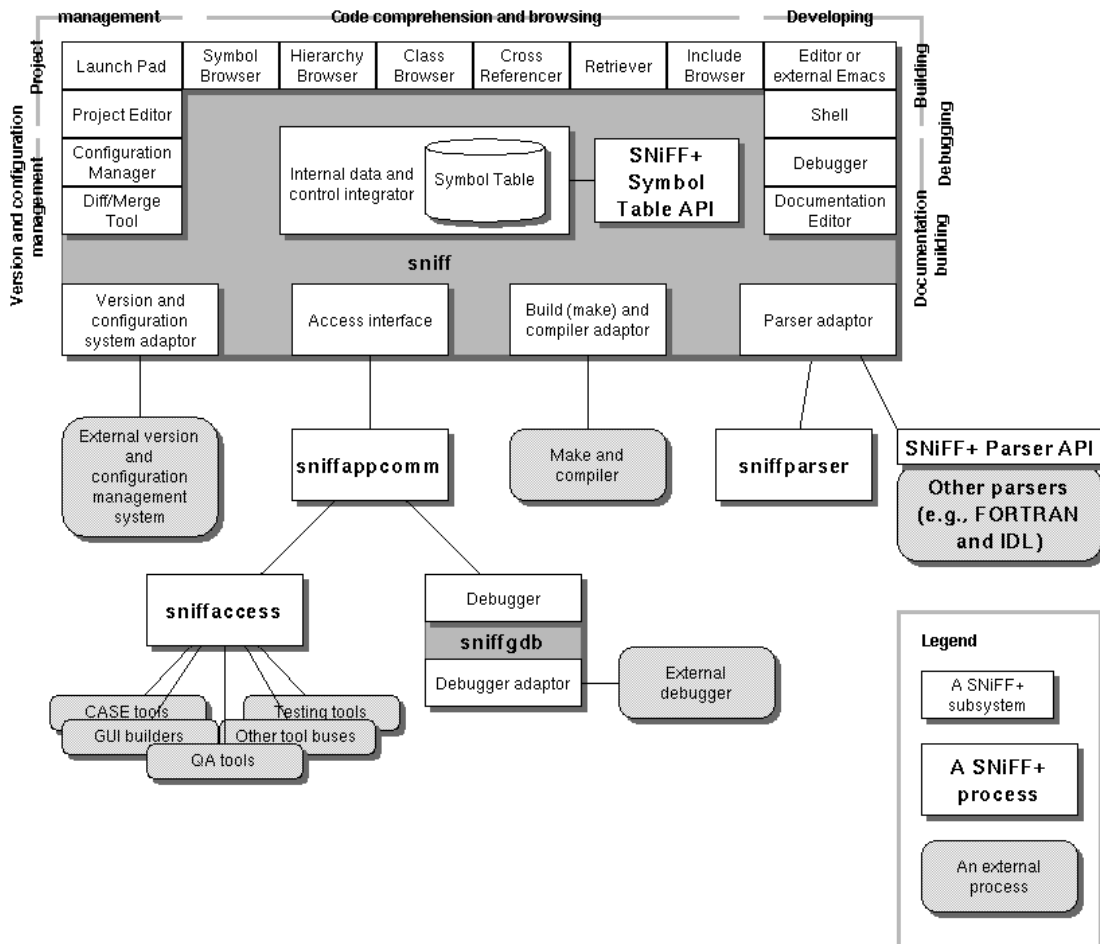
SNiFF+ is an open and scalable programming environment for C, C++, Java, Fortran and IDL. SNiFF+ supports the following development tasks:

- **Code comprehension and browsing**—SNiFF+ provides very powerful browsing and cross referencing features. Powerful filtering and visualization techniques work even with very large projects with many thousands of files, tens of thousands of symbols, and millions of lines of code. No compilation is necessary to extract the symbol information, as SNiFF+ has its own fast and fault-tolerant parsers.
- **Development**—SNiFF+ provides strong development support. All changes in the code are immediately reflected in all the browsing tools—no recompilation is necessary.
- **Documentation building**—SNiFF+ offers an integrated Documentation Editor that supports incremental and iterative documentation building. The hypertext-like linking provides quick navigation between source code and documentation. The generated documentation can be converted to other document publishing formats.
- **Project and code management for teams**—SNiFF+'s project and working environment concepts allow for the organization of large projects and effective cooperation in teams.
- **Version and configuration management**—All files of any file type can be managed with SNiFF+. Advanced tools facilitate the management of versions of files and configurations of projects.
- **Build support**—SNiFF+'s Make Support allows you to set up and activate your build processes using SNiFF+'s GUI. It provides automatic support for multi-platform development and works with compilers, linkers, archivers and other build tools of your choice. Furthermore, it provides (recursive) make rules for C/C++, Java, IDL and FORTRAN. Using these rules, you can completely regulate the make process (e.g., by choosing the order in which your targets are built).
- **Debugging support**—SNiFF+ provides interfaces to a wide range of debuggers under a consistent user interface that tightly integrates debugging tasks with browsing tasks.
- **Tool and control integration**—Together with adaptors to external tools like compilers, debuggers and version and configuration tools, the `sniffaccess` interface provides a flexible means for integrating SNiFF+ with third-party tools.

# SNIFF+ Architecture

## The SNIFF+ environment

The SNIFF+ environment consists of several tools and processes. The common data source for all tools is the Symbol table, which is held in memory but is persistent between sessions.



## Projects

### What is a project?

A *project* is the main structuring element in SNIFF+ for grouping together files and directories on your file system that logically belong together. You create projects in SNIFF+ to:

- browse, edit and compile a group of files and then run and debug their end products (e.g., executables and libraries)
- browse external libraries that another SNIFF+ project uses
- make a set of files, such as `#include` files, available to other SNIFF+ projects

### Tracking dependencies in a project

If you use SNIFF+'s Make Support, SNIFF+ tracks dependencies among source files in the project. During a build, only those source files that need recompiling are recompiled. As a result, you can add or remove include files in a project without having to worry about which files need to be recompiled. Before each build, just tell SNIFF+ to update a project's dependency information and other Make-related information.

### Project types

SNIFF+ distinguishes between two different project types: *shared* and *absolute*. The following table outlines the differences between these two project types:

Project Type	PDF Default extension	Can project files be shared among developers?	Project attributes refer to files and subprojects using:
Shared	*.shared	yes	paths relative to a root directory
Absolute (Browsing-Only)	*.proj	no	absolute path names

### For detailed information

Please refer to [Projects — page 23](#).

## Working environments

### What are working environments?

*Working environments* are physical directories on your file system in which SNIFF+ shared projects reside. In SNIFF+, you open shared projects by first specifying in which working environment you work.

### Working environments and teams

Working environments enable:

- shared access to your team data Repository
- shared and transparent access to team source code
- shared access to platform-specific object code
- individual team members to work in isolation from the rest of the team
- individual team members to work on selected configurations of a team project

### Working environments and single users

Single users can also benefit from using working environments:

- Working environments are easily movable.
- Working environments enable you to always know which projects you are working on.
- By using a Repository Working Environment, you can maintain one directory for your data Repository and another for your workspace.
- Just like with teams, single users can use working environments for single-platform or multi-platform development.

### For detailed information

Please refer to [Working Environments — page 29](#).

## Make Support

SNiFF+'s Make Support allows you to set up and activate your build processes using SNiFF+'s GUI. It provides automatic support for multi-platform development and works with compilers, linkers, archivers and other build tools of your choice. Furthermore, it provides (recursive) make rules for C/C++, Java, IDL and FORTRAN. Using these rules, you can completely regulate the make process (e.g., by choosing the order in which your targets are built).

SNiFF+'s Make Support uses data generated by SNiFF+, such as include path and dependencies information, to generate Make Support Files. Make Support Files, together with the different types of Makefiles provided with SNiFF+, act as an interface between SNiFF+ and Make.

### Features

SNiFF+'s Make Support:

- is based on standard Unix make tools
- comes with its own Makefiles
- is fully integrated with working environments to build targets across multiple shared working environments
- automatically generates make support files that contain data about include paths and dependencies lists for shared projects
- automatically provides make rules for recursively building a project's target
- maintains your build system by automatically updating make support files
- can be used with your choice of compilers, debuggers, linkers and archivers

### Make Support and working environments

SNiFF+'s Make Support maintains information about dependencies and include directives across working environment boundaries and supplies this information to your Make utility and compiler. Although you could maintain this information in your own makefiles, we strongly recommend that you use SNiFF+'s Make Support instead.

### Using your own makefiles

In SNiFF+ you can use your own Makefiles, however we strongly recommend that you use the Makefiles provided with SNiFF+'s Make Support for building the targets of your projects. It is also possible for each subproject to use your own Makefiles or the Makefiles provided by SNiFF+. For more information, see [Using Your Own Makefiles — page 105](#).

### For detailed information

For details about SNiFF+'s Make Support, please refer to [Build and Make Support — page 73](#).

## Versions and configurations

Support of configuration management and version control (CMVC) is an integral part of SNIFF+. The following features for handling versions and configurations are available:

### Features

SNIFF+'s CMVC support comes with the following features:

- Checking out files from your Repository with either *exclusive lock*, *concurrent lock*, or *no lock*.
- Looking at a file's history and seeing which files in a project are locked by which people.
- Working with *configurations* - selected file versions grouped together under the same symbolic name.
- Working with *change sets* - a set of files checked in at the same time under the same symbolic name to the files.
- Working in *branches* of a file's version tree.
- Displaying two-way and three-way differences and merging versions, branches, change sets and configurations.
- Associating comments, dates and modifier information with versions, change sets, and configurations.
- Choosing a *Default Configuration* for a working environment.

#### Note

SNIFF+'s CMVC features provide the functionality available in the RCS version control system. If you use a tool other than RCS, please be aware that your tool may not support all of the functionality available in SNIFF+.

### Available CMVC interfaces in SNIFF+

SNIFF+ provides an open and customizable interface to file and command-based version and configuration management tools. Please read the *Release Notes* to find out which interfaces are available.

### For detailed information

For details about SNIFF+'s Version Control and Configuration Management features, please refer to [Maintaining SNIFF+ Projects — page 119](#).

## Documentation building

SNiFF+ comes with a set of tools that allow you to incrementally and iteratively document your source code.

SNiFF+ uses your source code's symbol information to generate documentation frames out of a set of configurable templates. You can then fill out these frames during the process of documenting your source code.

### What can I document?

You can document any or all of the symbols in your source code. You can select the types of symbols that you want to document in your Preferences.

### Browsing and editing documentation

You can browse and edit documentation using SNiFF+'s Documentation Editor. The Documentation Editor behaves much like a hypertext browser that you can use for navigating between source code and its documentation.

### Customizing and creating documentation templates

When you document symbols in your source code, SNiFF+ uses documentation template files to generate documentation frames for each symbol. There is one documentation template file for each type of symbol. You can customize the template files that come with SNiFF+, or you can create new ones.

There are two ways in which you can customize documentation template files:

- by using HTML tags for layout
- by using macros to extract information from SNiFF+'s Symbol Table and comments from your source files

### Version-controlling documentation

Documentation files can be version-controlled in the same way as the other files in your software system.

### Exporting documentation

You can export your documentation in two different formats: MIF (Maker Interchange Format of FrameMaker™) and HTML. When you export your documentation as MIF files, you also have the option of creating a book file for the individual MIF files.

### For detailed information

For details about documenting your source code in SNiFF+, please refer to [Documenting Source Code — page 277](#).

# Source code parsing and symbol information

## SNiFF+ parsing technology

SNiFF+ uses its own parsers for parsing C/C++, Java, Fortran, CORBA IDL, Tcl and Python source code. No compilation is necessary in order to extract symbol information. The parser is highly configurable and can optionally preprocess the source code.

The SNiFF+ parsers are independent operating system processes that send a stream of information about the symbols defined and declared in the source code to SNiFF+'s Symbol Table. The symbol information is kept persistent on disk, so that parsing is done only once for each file or again after a change. When you modify a source file and save it, the file is immediately reparsed, and its symbol information is sent to the Symbol Table. All browsing tools are also updated.

You can edit project files outside of SNiFF+ and still browse the latest symbols. SNiFF+ knows when a project file has been modified externally and reparses it the next time you access the file.

## Preprocessing C/C++ source code

By default, SNiFF+ does not preprocess source code. However you can preprocess source code by doing the following:

- In the Project Editor, double-click on your project in the Project Tree to open the Project Attributes dialog.
- In the Project Attributes dialog, select the **Parser** node.
- In the Parser view, select the **Preprocess Source Code before Parsing** checkbox.
- Press **OK**.

You can configure preprocessing in the Parser configuration file. For more information, see [Parser configuration file — page 178](#).



## Cross reference subsystems

In addition to parsing symbol information, SNIFF+ also generates symbol cross reference information from your source code.

SNIFF+ provides two alternative cross reference (X-Ref) subsystems for managing cross reference information, either *RAM-based cross referencing*, or *database-driven cross referencing* (available as of SNIFF+ 3.2).

In a nutshell, the RAM-based solution loads a set of indexed files to memory to resolve X-Ref queries, whereas the DB-driven solution directly accesses a database.

However, because X-Ref information is stored at Working Environment level under DB-driven cross referencing, this technology has a number of implications for Working Environments administration in team projects.

Which cross reference engine is used therefore determines not only how X-Ref information is generated, stored, and subsequently accessed to resolve queries, but also influences Project and Working Environment administration. As well as performance and scalability.

### For detailed information

For details about SNIFF+ cross reference subsystems and their implications, please refer to [Cross Reference Information — page 219](#).

# Mix-and-match tool and control integration

## Introduction

SNiFF+ can be used with a variety of third-party tools. You can use SNiFF+ with your choice of:

- editors
- compilers
- debuggers
- version and configuration management systems

## Editing systems

SNiFF+ is integrated with a number of popular 3rd-party editors (see [Editor Integrations — page 231](#)), as well as also the Visaj Java GUI builder.

## Compilers and debuggers

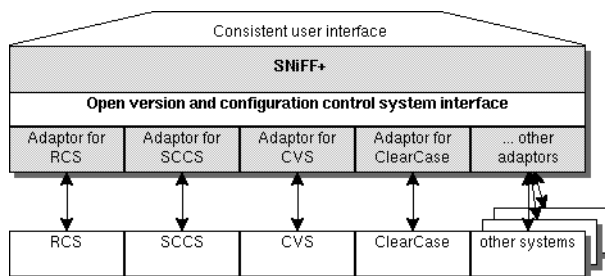
Due to its compiler-independent architecture, SNiFF+ can interface to any compiler. Any C debugger can debug code of any C compiler on the same platform as long as the memory layout is the same. Because of name mangling and other issues, C++ debugging is compiler-dependent.

Debugging systems that are not yet supported in SNiFF+ can be integrated with minimal effort. The design of SNiFF+ factors out the commonalities of the debugging systems and provides a flexible adaptor architecture. This guarantees that new debuggers can be incorporated relatively easy.

For a list of debuggers supported by SNiFF+, please refer to the Release Notes.

## Version and configuration control systems

SNiFF+ has an open and adaptable command interface to any file- and command-based third-party version and configuration control tool. The interface consists of approximately forty commands and options to be set for a specific tool. Please refer to the *Release Notes* for information on supplied tool adaptors.



## Control integration with sniffaccess

Besides having tool integration features, SNiFF+ can be controlled externally.

The `sniffaccess` subsystem provides bidirectional control integration between SNiFF+ and external tools.

For example, SNiFF+ can be told to open a project, display a class in the inheritance hierarchy, load a file into the Editor, or send a notification when a file is saved. Third-party tools like CASE tools, GUI builders, and testing tools can be integrated via that interface. A generic bridge to the HP Softbench BMS is provided with the package.

Please refer to the *Release Notes* for information on specific tools.



# Part II

## **SNiFF+ Projects and Working Environments**



# Projects

---

## What is a project?

A *project* is the main structuring element in SNIFF+ for grouping together files and directories on your file system that logically belong together. You create projects in SNIFF+ to:

- browse, edit and compile a group of files and then run and debug their end products (e.g., executables and libraries)
- browse external libraries that another SNIFF+ project uses
- make a set of files, such as `#include` files, available to other SNIFF+ projects

## Project directories and SNIFF+ generated files

Generally, you create a project for existing source files. When you create the project, you must specify the directory containing these source files. The directory you specify is referred to as the *project directory*. Each project in SNIFF+ corresponds to a project directory on your file system.

During project creation, SNIFF+ generates the following files and directories in a project directory:

- **Makefile**—The project makefile. This file is generated when you choose to build your targets using SNIFF+'s Make Support.
- **The Project Description File (PDF)**—Each SNIFF+ project is described by a Project Description File (PDF) that stores the structure, the list of files, and the attributes of the project. SNIFF+ maintains a project's PDF for you. These files have the extension `.proj` and `.shared`.
- **Project Generate Directory**—This directory contains a number of files generated for the project and maintained by SNIFF+. By default, this directory is called `.sniffdir` and is located in the project directory.

## The contents of a project

Each SNiFF+ project contains the following:

- **Your source files**—You can include any type and number of source files in a project. For example, a typical SNiFF+ project might have C++ implementation and header files, yacc sources, documentation files and files of a third-party documentation tool like FrameMaker.
- **Makefile**—Either your own or SNiFF+'s, depending on whether you use SNiFF+'s Make Support or not.
- **The Project Description File (PDF)**—When you open a project, you are really telling SNiFF+ to load the project's PDF. When you modify a project's structure in any way (e.g., by adding or removing files to the project), its PDF will be changed accordingly.

## Tracking dependencies in a project

If you use SNiFF+'s Make Support, SNiFF+ tracks dependencies among source files in the project. During a build, only those source files that need recompiling are recompiled. As a result, you can add or remove include files in a project without having to worry about which files need to be recompiled. Before each build, just tell SNiFF+ to update a project's dependency information and other *make*-related information.

If you don't use SNiFF+'s Make Support, you must update your own makefiles to reflect any changes in dependencies.

## Project structures

You can include projects in other projects to create a hierarchical project structure. The process of including a project is referred to in SNiFF+ as *adding a subproject*.

Some typical uses of hierarchical project structures are:

- Your source files are already in a hierarchical directory structure and you want to use this same structure in SNiFF+.
- You have a project that builds a library. You want to make this library accessible to other projects. To do so, you would *add the library subproject to your other projects*. If you use SNiFF+'s Make Support, you can then set an attribute that tells SNiFF+ to automatically link the library to the targets of your projects.
- Your project uses an external shared library to build its targets. Source code for the shared library is also available, and you would like to browse it. To do so, you would create a project for the external source code and include it in your main project. So, whenever you open the main project, you can immediately browse the external source code as well.

Note that SNiFF+'s Make Support also handles dependency tracking in project structures.



## Project types

SNiFF+ distinguishes between two different project types: *shared* and *absolute*. The following table outlines the differences between these two project types:

Project Type	PDF Default extension	Can files be shared among developers?	Project attributes refer to files and subprojects using:
<b>Shared</b>	*.shared	yes	paths relative to a root directory
<b>Absolute (Browsing-Only)</b>	*.proj	no	absolute path names

### Shared projects

As the name suggests, shared projects are suitable for team development. Each team member has access to a shared project and can make changes to its files and/or structure. Shared projects are always used in conjunction with a configuration management and version control (CMVC) tool of your choice.

Shared projects offer a great deal of flexibility. Since all references to files and subprojects are relative to a root directory, you can easily move a shared project to another location on a file system.

#### Note

We strongly recommend that you work with shared projects, even if you don't work in a team. In our experience, most single-user development work is incorporated into a team development environment sooner or later. With shared projects, the transition from a single-user to a team environment is much smoother than with absolute projects.

### Absolute projects (Browsing-Only)

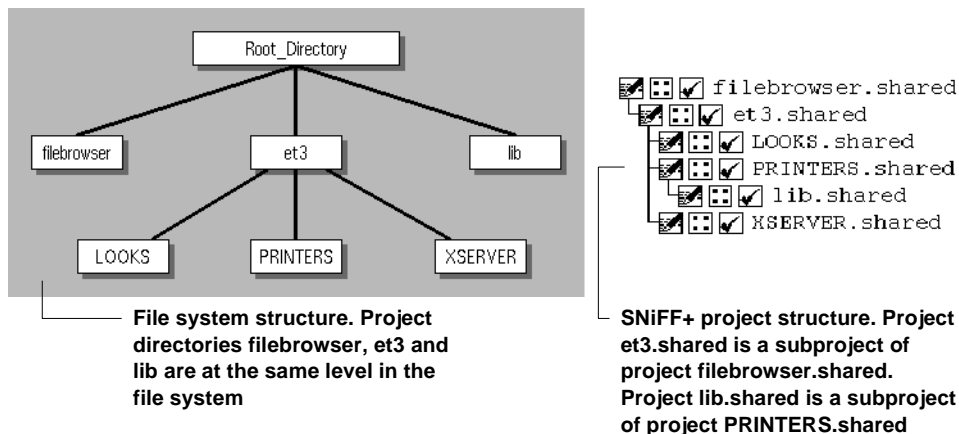
Absolute projects are most suitable for browsing code. It's easy to set up an absolute project, so it makes sense to use them if you just need to get your source code "into SNiFF+" for browsing purposes. For any serious development work, use shared projects.

## Organizing project structures

Project structures in SNIFF+ **need not** map directly to file system structures. The following figure illustrates this idea.

### Note

Please note that the following figure gives just one example of how project structures can be organized in SNIFF+. We'll use this figure again in the next chapter to illustrate other key SNIFF+ concepts.



## The file system structure

The left-hand side of the diagram shows an example file system structure. The directory `Root_Directory` contains directories: `filebrowser`, `et3` and `lib`. Each directory corresponds to a SNIFF+ project of the same name. Project `et3` contains the framework for the GUI used by project `filebrowser`.

Notice that, in the file system, the `et3` project directory contains three subdirectories: `LOOKS`, `PRINTERS` and `XSERVER`. Each of these subdirectories also corresponds to a SNIFF+ project of the same name.

Finally, the `lib` directory in the file system also corresponds to a SNIFF+ project of the same name.

## The SNIFF+ project structure

The right-hand side of the diagram shows how this file system structure is mapped to a SNIFF+ project structure. This structure has been chosen according to the following criteria:

1. Project `PRINTERS` uses the library target of project `lib.shared` to build its target. As a result, project `lib.shared` needs to be a subproject of `PRINTERS.shared`.
2. Project `et3` uses the targets of projects `LOOKS.shared`, `PRINTERS.shared`, and `XSERVER.shared` to build its target. As a result, these three projects need to be subprojects of `et3.shared`.
3. Project `filebrowser` uses the targets of project directory `et3.shared` to build its target. Therefore, `et3.shared` needs to be a subproject of `filebrowser.shared`.

As you can see, the project tree structure that you work with in SNIFF+ need not match the project directory structure on your file system.

## How you would create this SNIFF+ project structure

You may be wondering how you would create this project structure. Basically, the steps involved are:

1. Create a SNIFF+ project for directory `filebrowser`.
2. Create SNIFF+ projects for the directory `et3` and its subdirectories `LOOKS`, `PRINTERS`, and `XSERVER`.

Note that, when you create a SNIFF+ project for the directory `et3`, you can have SNIFF+ automatically create subprojects for its subdirectories as well.

3. Create a SNIFF+ project for directory `lib`.

There are now six SNIFF+ projects, each one corresponding to one of the six directories under `Root_Directory`. The final steps are to add subprojects to `PRINTERS.shared` and `filebrowser.shared`.

1. Add `lib.shared` to `PRINTERS.shared`.
2. Add `et3.shared` to `filebrowser.shared`.

## Choosing which project to open

As we've already mentioned, each project in a SNIFF+ project structure is a complete project. To work on a project, you first have to open it. Suppose you've got a project structure like the one on [page 26](#). You then have the following options:

- If you plan to modify and rebuild a single project, open only that project.
- If you plan to modify and then rebuild the `et3` framework, open `et3.shared`. SNIFF+ will automatically open all its subprojects. You can then work on `et3.shared` and all its subprojects.
- If you plan to modify and then rebuild the `filebrowser` project, open `filebrowser.shared`. SNIFF+ will automatically open all its subprojects. You can then work on `filebrowser.shared` and all the subprojects it includes.

## What to do next

If any of the following points apply to you, you will be working with SNIFF+ shared projects:

- You work in a team.
- You use a data Repository for version control purposes.
- You develop for multiple platforms.
- You work alone but will likely be sharing your work with others in the future.

Before learning how to set up shared projects, please read the next chapter ([Working Environments — page 29](#)). The chapter introduces SNIFF+ Working Environments and discusses its uses in both team and single-user development environments.

If you just need to get your source code “into SNIFF+” for browsing it, you will be working with SNIFF+ absolute projects. Please read [Project Setup Overview — page 43](#) next.

# Working Environments

---

## Introduction

This chapter discusses working environments and their role in day-to-day development work.

### This chapter covers the following topics

- Working environments and team development
- Working environments and single-user development

### Assumptions made in this chapter

- You have already read chapter [Projects — page 23](#)

### Related SNiFF+ topics

- Setting up working environments — [Setting Up Team Working Environments — page 57](#)
- Creating shared projects — [Creating Team Projects — page 63](#)

## What are working environments?

*Working environments* are physical directories on your file system in which SNiFF+ shared projects reside. In SNiFF+, you open shared projects by first specifying in which working environment you work.

You **must** use working environments:

- If you are a member of a development team that works on the same set of files, and you do not use a third-party configuration management tool that furnishes a workspace model of its own (such as ClearCase™).
- If you develop software for multiple platforms (as a member of a development team or alone).
- If you work alone on projects and plan to share them in the future.

You **need not** use working environments:

- If you are working with absolute projects.
- If you work alone on a project and don't need to share your project with others, now or in the future.
- If you already use a third-party configuration management tool such as ClearCase™.

## What types of working environments are there?

There are four types of working environments:

- Repository Working Environment
- Shared Source Working Environment
- Shared Object Working Environment
- Private Working Environment

These four types are discussed in detail in the rest of this chapter.

## Make Support and working environments

SNiFF+'s Make Support maintains information about dependencies and include directives across working environment boundaries and supplies this information to your *make* utility and compiler. Although you could maintain this information in your own makefiles, we strongly recommend that you use SNiFF+'s Make Support instead.

For details about Make Support, please refer to [Build and Make Support — page 73](#).

## Working environments and teams

Working environments are designed to be used in teams. In this section, you will learn how working environments support team development. At the same time, you will also learn about each working environment type and how the four types interact with each other.

## Shared access to your team Repository

Your team members access and modify a permanent shared data Repository using commands provided by your underlying configuration management and version-control (CMVC) tool. SNiFF+ provides an interface to your CMVC tool. This interface needs to know the location of your Repository. You provide this information by defining a *Repository Working Environment* (RWE), which specifies the root directory of your Repository.

## Shared and transparent access to team source code

SNiFF+ requires you to specify the root directory under which your team's shared source code is located. The files and directories under this root directory access your team's Repository. At regular intervals, all these files and directories need to be updated to reflect the most current state of your team's software system. How this updating occurs is discussed later on. When creating software systems from scratch, your team's first job is to populate this root directory with source code. For existing software systems, your team will already have such a central location. In either case, once you have such a root directory, you have to tell SNiFF+ where it is. You do this by defining a *Shared Source Working Environment* (SSWE).

All team members see, or *share*, the latest version of your software system as reflected by the source files in the SSWE. When browsing the source files, this view is read-only. When editing source files, team members work on *local* copies of the shared source files they want to modify—they never directly modify the shared source files in the SSWE. The view to all other source files (those not being modified) remains read-only.

SNiFF+ also supports more complicated models of source code storage. You might, for example, have multiple locations containing shared source code, with different team members making modifications to the different shared source pools. A central location might “pool together” the most current versions of the source code from the different shared source pools. In this case, you can define multiple SSWEs. We’ll discuss different strategies for using SSWEs later on.

## Directories for platform-specific object code

Just like with shared source code, SNiFF+ also requires you to specify a central location for your team’s shared object code. In SNiFF+, you define a *Shared Object Working Environment* (SOWE), which specifies the root directory containing your shared object files.

SOWEs serve two purposes:

- To be shared repositories for your team’s most current and stable platform-specific object code. During an update of a SOWE, source files in the SSWE are compiled and the resulting object code is stored in the SOWE.
- To speed up the build process in the PWEs that access them (see [Isolating individual work from the team — page 31](#)).

### Multi-platform development

Complex software projects often involve maintaining several configurations for several platforms at the same time. SNiFF+ allows you to have several platform-specific directories for organizing your team’s shared object code. For each platform-specific directory, you define a SOWE. As is the case for a single SOWE, the object code in platform-specific SOWEs is also derived from your team’s shared source code.

## Isolating individual work from the team

Developers must be able to work in isolation from other team members. They need their own workspaces in which they can edit, compile and debug projects without interfering with the work of their team members. Furthermore, they continually need to have access to their software system’s most current source code and object code base.

SNiFF+ supports this by allowing each member of a team to work in an isolated workspace. In SNiFF+, you define a *Private Working Environment* (PWE) to specify the root directory of each team member’s workspace.

You can go through the entire edit-compile-debug cycle in your PWE. When working in your PWE, you have a read-only view to the shared source files located in your team’s SSWE. When you need to modify shared source files, you check out the necessary files from your team’s Repository. When you’re satisfied that the changes you’ve made are error-free, you

check the modified files back into your team's Repository. The next time your team's SSWE is updated, these changes are incorporated, and the shared source files in the SSWE once again reflect the most current state of your software system.

## Working on selected configurations of a team project

Development teams are usually split up into groups of programmers who are responsible for a particular aspect of a project. For example, after releasing a version of your software product, you might choose to split your team into two sub-teams: one responsible for developing the product further (Team A), and another responsible for fixing the bugs found in the product (Team B). In terms of the files maintained in your Repository, Team A would work on the main (trunk) configuration of your software system, and Team B would work on a branch configuration. At some time in the future, the most current versions of the two configurations are merged with each other.

To assist project leaders in making sure that team members are working on the correct configurations of a software system, working environments can specify a *Default Configuration*. This is the configuration of your software system that your team members work on, and is used as the default value for version control operations such as check-in and check-out.

Default Configurations and SNIFF+'s branch support are discussed in the Chapter [Version Control — page 135](#).

## Avoiding unnecessary builds in the PWE

An essential aspect of shared object working environments (SOWEs) is avoiding unnecessary builds in PWEs that access them. When you build the targets of a project opened in your PWE, only checked-out source files newer than their dependencies in the SOWE are compiled. All other objects are taken from the SOWE. This can speed up builds in PWEs considerably.

For example, suppose you're working in your PWE on a project called `Views.shared`, whose target is an executable. The SOWE accessed by your PWE already has targets for the project. Let's further suppose that you've checked out and modified a single header file, `View.h`, in your PWE.

Now, when you build the project's executable in your PWE, `View.o` will be locally built in your PWE, and SNIFF+ creates symbolic links to all other objects in the SOWE. As a result, your local `View.o` will be linked with other shared object files to produce your executable.

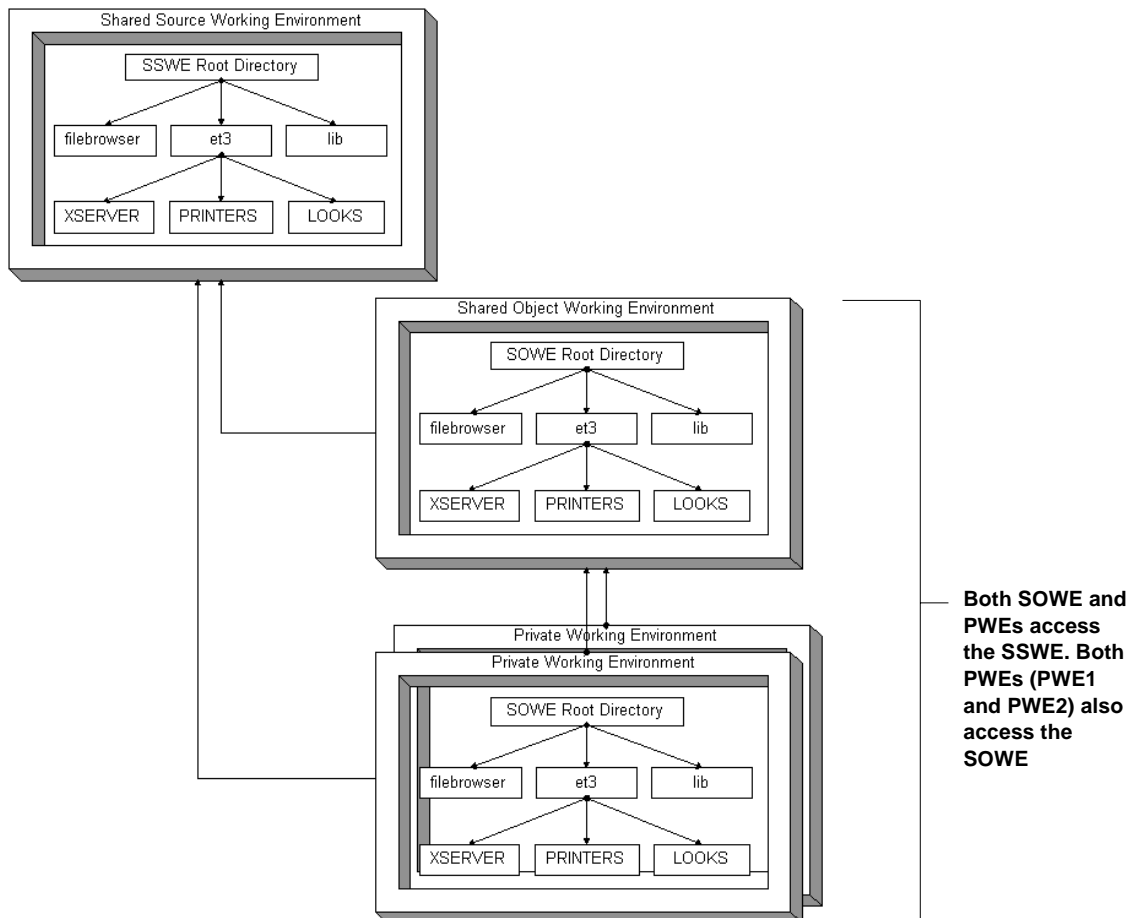


## How file sharing works

SNiFF+ handles file sharing among working environments by requiring that all affected working environments have the same project directory structure. This is not a restriction, but rather the easiest way for file sharing to work. A SNiFF+ project's Project Description File stores, among other things, structural information about the project — information such as the names of project files and their location relative to the project directory, and the names and locations of any subprojects. When all working environments that share files have the same project directory structure, SNiFF+ can easily find any project files or subprojects.

The project directory structure of the Shared Source Working Environment (SSWE) is the basis for all other working environment project directory structures. SNiFF+ automatically copies the SSWE's project directory structure into your other working environments for you. Note that SNiFF+ only copies the SSWE's directory structure — and not contents.

The following diagram illustrates the idea of equivalent project directory structures:



The SOWE and two PWEs have the same project directory structure as the SSWE. When the SOWE is updated, object code in its project directories is derived from source code in the corresponding project directories of the SSWE. For example, object code in the SOWE's `filebrowser` project directory is generated from source code in the SSWE's `filebrowser` project directory.

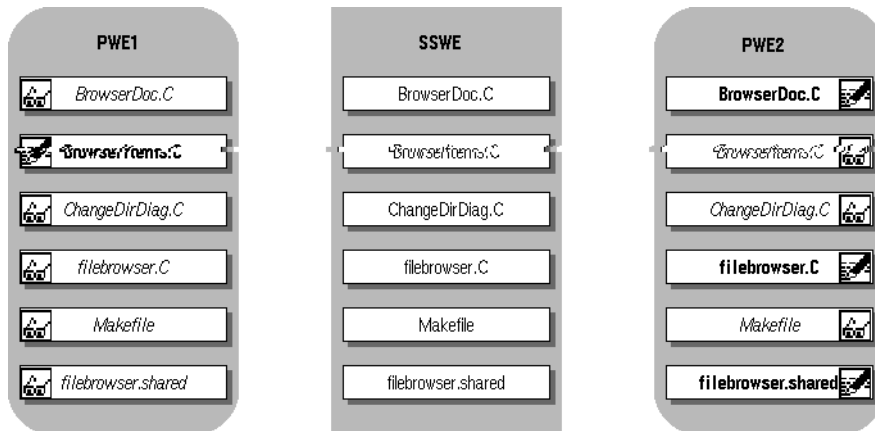
The two team members working in PWE1 and PWE2, respectively, share the source files in the SSWE. When browsing source files, their view to the files is read-only. When editing source files, they work on local, writable copies of the source files they've checked out from the Repository. When compiling in their PWEs, object code is created locally from both shared (read-only) source files and local (writable) source files.


## A closer look at file sharing

Let's look more closely at the SSWE, PWE1 and PWE2. Let's suppose that the `filebrowser` project directory in the SSWE contains the following:

- The Project Description File `filebrowser.shared`
- The project makefile `Makefile`
- The following source files: `BrowserDoc.C`, `BrowserItems.C`, `ChangeDirDialog.C` and `filebrowser.C`
- The Project Generate Directory, which contains Make support files and other generated files

The following figure shows the contents of the filebrowser project directory in the SSWE, PWE1 and PWE2. Let's assume that two developers named Jill and Peter own the PWEs. Jill owns and works in PWE1. Peter owns and works in PWE2. Both Jill and Peter share common source files located in the SSWE:



 Indicates a read-only file located in the SSWE

 Indicates a local, writable file checked out in a PWE

As the figure above shows, Jill has checked out only one file to the filebrowser project directory in her PWE: BrowserItems.C. She has a read-only view to all other files.

Peter has checked out three files to the filebrowser project directory in his PWE: two source files (BrowserDoc.C and filebrowser.C) and the Project Description File (filebrowser.shared). He has a read-only view to all other files.

### Note

To make structural changes to a SNIFF+ project, you must check out the project's Project Description File! Examples of structural changes are adding/removing project files and subprojects, and changing project attributes (e.g., names of project targets).

Note carefully in the figure that Jill has a read-only view to files checked out by Peter, and Peter has a read-only view to files checked out by Jill. That is, while Jill is making changes to her local copy of BrowserItems.C in her PWE, Peter can only browse the original copy of the file located in the SSWE. And while Peter is making changes to his copies of BrowserDoc.C and filebrowser.C and filebrowser.shared, Jill can only

browse the original copies of these files. This is an example of the *exclusive file locking*. When one team member has checked out a file in his/her PWE, all other team members can only browse this file.

---

**Note**

SNiFF+'s configuration management and version control (CMVC) interfaces provide for other file locking mechanisms as well. Your underlying CMVC tool determines which mechanisms are available for you to use.

## Examples of using working environments

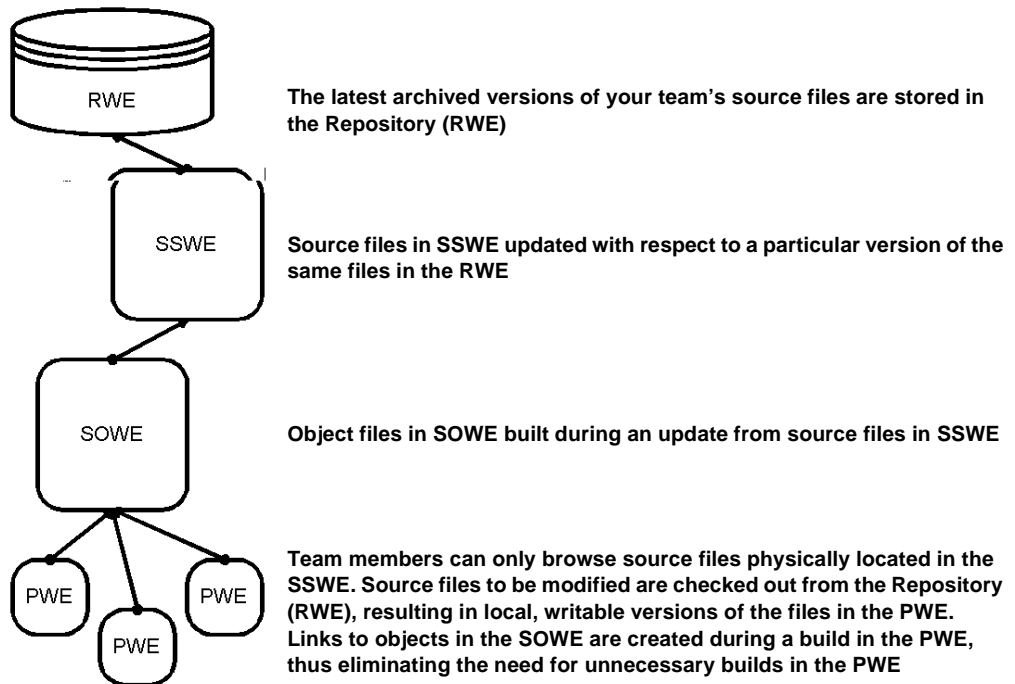
This section presents two different scenarios in which working environments are required. Each scenario is followed by a description of how it can be realized using working environments. All scenarios assume that a Repository is used.

### Team development with single SSWE and SOWE

**The scenario** — Your development team shares a common shared source and shared object base. You check out files from the Repository using exclusive lock.

**The solution** — This is the “classic” scenario for using working environments. To realize this scenario in SNiFF+, use one RWE, one SSWE, one SOWE and one PWE for each team member.

The following diagram illustrates this:

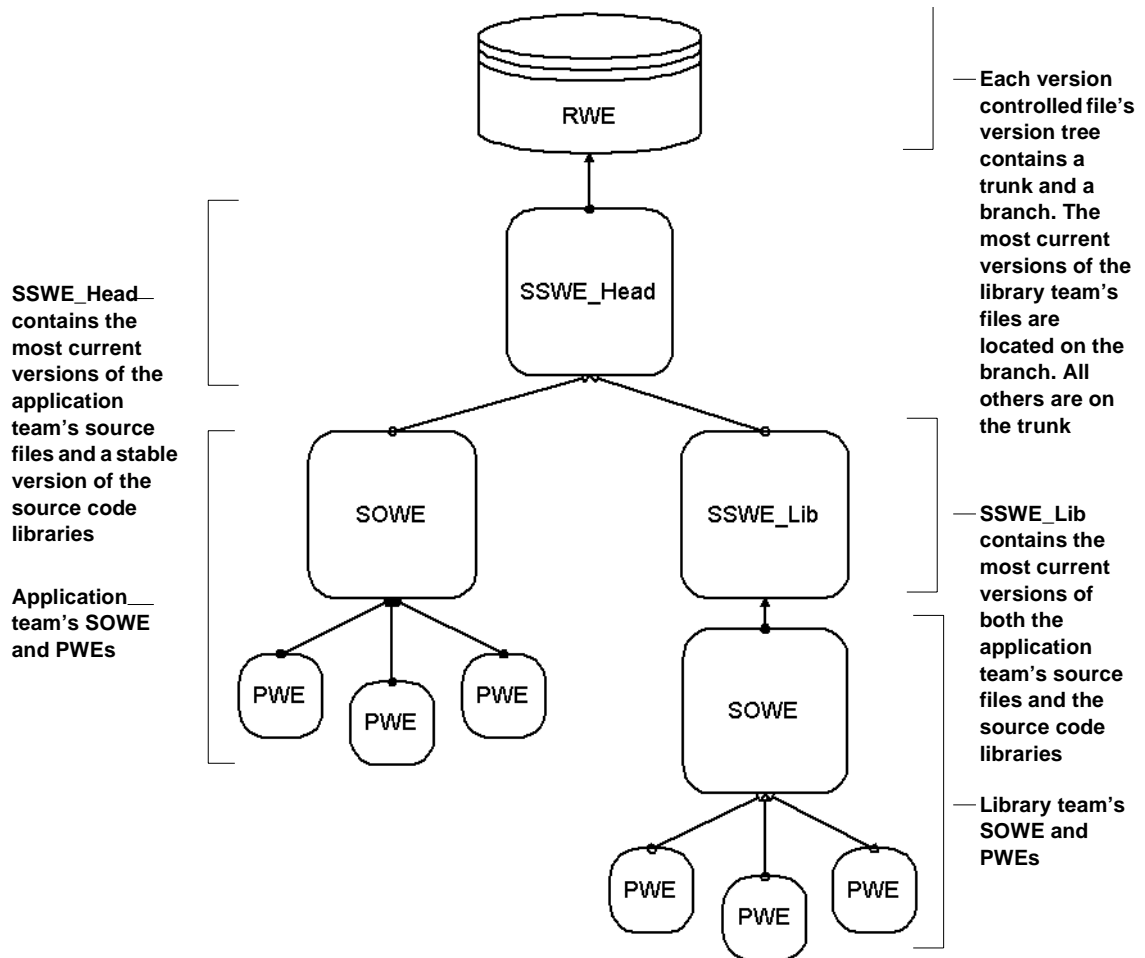


## Team development with multiple SSWEs and SOWEs

**The scenario** — A situation in which you might want to use multiple SSWEs and SOWEs is when your development team is split into two teams—for example, into a *library team* that owns and develops a library and an *application team* that uses and occasionally makes changes to this library.

The library team's SSWE (SSWE\_Lib) would contain the latest configuration of the library. The SSWE accessed by the application team (SSWE\_Head) contains the most current versions of the application team's source files and less often updated, but more stable configurations of the library.

**The solution** — This scenario can best be realized using one RWE, two SSWEs, two SOWEs and one PWE for each team member in combination with SNIFF+'s branch support. The following diagram illustrates how this works:



To implement this solution in SNIFF+, you must do two things:

- Specify the Default Configurations for the required working environments. Remember that a working environment's Default Configuration is used as the default value for version control operations such as check-in and check-out.
- Use SNIFF+'s branch support.

For details about SNIFF+'s version control features including branch support, please refer to [Version Control — page 135](#). Here, we use branch support as it relates to Default Configurations.

## For SSWE\_Head

For the SSWE labelled `SSWE_Head` in the diagram, you would specify its Default Configuration to be `HEAD`. The symbolic name `HEAD` is used by many version control systems (e.g., `RCS`) to refer to most current version of a file in the trunk (main branch) of a file's revision tree. You would also specify `HEAD` as the Default Configuration for the application team's SOWE and PWEs.

## For SSWE\_Lib

For the SSWE labelled `SSWE_Lib` in the diagram, you would specify two Default Configurations. The first one is the name of the version tree branch on which the library is developed, say `Lib`. The second one is `HEAD`.

## How SNIFF+ uses the Default Configurations

When the library team updates its working environments, all source files are updated to the latest version on branch `Lib`, if this branch exists. Of course, this branch exists for the library team's source files, but not for the application team's source files. So, in the library team's working environments, library source files are updated to `Lib`, and application source files are updated to `HEAD`.

When the latest versions of the files on branch `Lib` have reached a pre-defined level of stability, they are checked into the Repository as `HEAD`. This is normally done by a selected member of the team, and usually from the SSWE (`SSWE_Lib`). Consequently, the next time members of the application team update their working environments, they will have access to the latest versions of both their files and those of the library team.

In summary, the application team works most of the time with stable, but not necessarily current, versions of the source libraries maintained by the library team. It has access to the latest version only after library source files have been checked into the Repository as the `HEAD` versions of their respective version trees.

## Working environments and single users

Single users can also benefit from using working environments:

- Working environments are easily movable.
- By using a Repository Working Environment, you can maintain one directory for your data Repository and another for your workspace.
- Just like with teams, single users can use working environments for single-platform or multi-platform development.

## Single-platform development

For single-platform development, the following working environments are needed:

- Repository Working Environment for data Repository
- Private Working Environment for source code

## Multi-platform development

For multiple-platform development, the following working environments are needed:

- Repository Working Environment for data Repository
- Shared Source Working Environment for source code
- one Shared Object Working Environment for each target platform
- one Private Working Environment for each developer



# Part III

## Setting Up SNiFF+ Projects



# Project Setup Overview

---

## Introduction

Setting up projects in SNIFF+ can be as simple or as complicated as you want it to be. This is because SNIFF+ offers you a great deal of flexibility during the setup process.

When you create a new project, you can start off by specifying a minimum number of project attributes and then specify the rest after setup, or you can specify as much as you want right away.

In the rest of this chapter, and for that matter of this Guide, we assume that you always want to start with the basics and move onto the complicated stuff afterwards. Therefore, we don't consider setting up project attributes such as Make and preprocessor directives to be part of the regular project setup process. Instead, we first describe how to set up projects without these attributes and then point you to other Parts of this Guide that cover them.

## Abbreviations and shortcuts used in this chapter

RWE — Repository Working Environment  
SSWE — Shared Source Working Environment  
SOWE — Shared Object Working Environment  
PWE — Private Working Environment

## SNIFF+ Project Setup Wizard

You can set up projects and working environment using SNIFF+'s Project Setup Wizard. The Project Setup Wizard can be used to set up projects for the following development situations:

- single-platform development in teams
- multi-platform development in teams
- single-user, single-platform development
- single-user, multi-platform development
- single-user, browsing-only

The Project Setup Wizard can also be used for the following tasks:

- adding new working environments
- creating new projects in an existing working environment.

## Project setup overview — procedures

### Java Projects

Note that the project setup described here does not apply to Java projects. Please refer to the SNIFF+ *Java Tutorial* for details about setting up Java projects.

For team projects, the following steps should be completed by the Working Environments Administrator (described on [page 159](#)).

Here is our recommended procedure for starting out with a new project:

1. Determine your development situation.
2. If you need working environments, create and specify a *Working Environment Configuration Directory*.
3. Create the project.
4. Specify a *default working environment* if necessary.
5. Initialize new team working environments if necessary.
6. Set up Make Support.

### Step 1: Your development situation

- Determine your development situation. For a quick overview of the type of project and working environments you will need, please refer to [Typical development situations — page 46](#).

### Step 2: The Working Environment Configuration Directory

SNIFF+ stores and maintains working environment files in a *Working Environment Configuration Directory*. By default, this directory is:

```
$SNIFF_DIR/workingenvs
```

- If you intend to use this directory, you will first have to set permissions for the working environment files stored in it. To do so, please refer to [Using the default Working Environment Configuration Directory — page 51](#).
- If you intend to use another directory, please refer to [Using a different Working Environment Configuration Directory — page 51](#).

### Step 3: Create the project

- Set up the appropriate SNIFF+ project and/or working environments for your development situation. Then, in the Launch Pad:
  - To create the project using the Project Setup Wizard, choose **Project > New Project... > with Wizard...**
  - To create the project using default values taken from your Preferences, choose

**Project > New Project... > with Defaults....** For a detailed example of how to use this option for multi-user development projects, refer to [Creating Team Projects — page 63](#).

- To create the project using templates, refer to [Working with new project templates — page 47](#).

We recommend that novice and intermediate SNIFF+ users use the Project Setup Wizard. More advanced users, and particularly Working Environment Administrators, may be interested in creating projects using either defaults or templates.

#### Note

In a real world situation, it may not matter to you whether your source code is initially compilable. However, before creating a new SNIFF+ project from scratch, we recommend that you verify that your source files are compilable. Then, when you set up Make Support for the project, you will know that any compile-time errors must be a result of improperly set Make attributes.

## Step 4: Specify a default working environment

Your *default working environment* is the working environment in which you normally work. For individual developers working in teams or alone, this will generally be a PWE. For a Working Environments Administrator, this will usually be either the team's SSWE or SOWE. Setting a default working environment has two advantages:

- You can open projects more quickly, since you won't have to first select the working environment in which you work. SNIFF+ will use the default working environment.
- SNIFF+ warns you if you try to open projects in a working environment that isn't your default working environment.

To specify a default working environment, please refer to [Specifying a default working environment — page 52](#).

## Step 5: Initialize team working environments

The goal of initializing team working environments is to be able to share files between all the working environments your team uses.

- To initialize working environments, please refer to [Initializing team working environments — page 53](#).
- To first learn how file sharing between working environments works, please read [How file sharing works — page 33](#).

## Step 6: Set up Make Support

You can build targets in SNIFF+ using either SNIFF+'s Make Support or your own Makefiles. In a team development environment, we strongly recommend using SNIFF+'s Make Support.

- For details about Make Support, please refer to [Build and Make Support — page 73](#).
- For details about using your own Makefiles in SNIFF+, please refer to [Using Your Own Makefiles — page 105](#).

## Typical development situations

Note that we recommend that your Working Environments Administrator sets up all team projects and working environments.

### Single-platform development in teams

Type of setup in Project Setup Wizard: Standard

Required SNIFF+ project type: Shared

Required working environments: RWE, SSWE, one SOWE, and one PWE for each team member

### Multi-platform development in teams

Type of setup in Project Setup Wizard: Standard

Required SNIFF+ project type: Shared

Required working environments: RWE, SSWE, one SOWE for each target platform, and one PWE for each team member

### Single-user, single-platform development with external data Repository

Type of setup in Project Setup Wizard: Standard

Required SNIFF+ project type: Absolute or Shared (Shared recommended)

Required working environments (for Shared projects only): RWE for Repository, PWE

### Single-user, multi-platform development

Type of setup in Project Setup Wizard: Standard

Required SNIFF+ project type: Shared

Required working environments: RWE, SSWE for source code, one SOWE for each target platform

### Single-user, browsing-only

Type of setup in Project Setup Wizard: Browsing-only

Required SNIFF+ project type: Absolute

Required working environments: None

## Working with new project templates

New project templates allow you to quickly create a new SNIFF+ project using the attributes of an existing project stored in a special template file. Template files have the extension `.ptmpl` and behave in many ways like a project's Project Description File (PDF).

When you tell SNIFF+ that you want to create a new project using a template, SNIFF+ first asks you to select the template you want to use, then the project directory containing the source files of the new project. An Attributes of a New Project dialog then appears, and the attributes stored in the selected template file are loaded into the various fields and boxes of the dialog. You can change any of these attributes or leave them as they are.

## Creating a template

There are two ways of creating templates:

- By saving an existing SNIFF+ project as a template file
- By creating a new project using the Attributes of a New Project dialog and default values from your Preferences

### Creating a template using an existing SNIFF+ project

1. Open the project for which you want to create a template.
2. In the Project Editor, double-click on the root project in the Project Tree.  
The Project Attributes dialog appears.
3. In the General view, select the **As Template** radio button and press **OK**.  
The Project Template dialog appears. You will now name the template file and store it onto disk.
4. In the **Template File** field, enter a name for the new template file. By default, its extension will be `.ptmpl`.
5. In the **Description** field, enter a description for the new template file.  
Entering a description lets you easily distinguish between a list of template files later on.
6. By default, template files are stored in your `<sniff_installation_dir>/config/project` directory. To select a different template file directory, press the **Change Directory** button at the top of the dialog and use the Directory dialog that appears.  
When you are done, press the **Save** button to save the template file to disk.

#### Note

A template file is created for the current project only. No template files are created for any subprojects that the current project may have.

## Creating a template using the Project Attributes dialog

1. In the Launch Pad, choose **Project > New Project... > with Defaults...**.  
The Directory dialog appears.
2. Press the **Select** button. The directory that you choose in this dialog is unimportant.  
The Attributes of a New Project dialog appears.
3. Navigate to the **General** view and, under **Save Current Project Attribute Settings**, choose the **As Template** radio button.
4. Set all other attributes in the Attributes of a New Project dialog according to your needs.  
When you are done, press the **Ok** button.  
The Project Template dialog appears. You will now name the new template file and store it onto disk.
5. In the **Template File** field, enter a name for the new template file. By default, its extension will be `.ptmpl`.
6. In the **Description** field, enter a description for the new template file.  
Entering a description lets you easily distinguish between a list of template files later on.
7. By default, template files are stored in your `<sniff_installation_dir>/config/project` directory. To select a different template file directory, press the **Change Directory** button at the top of the dialog and use the Directory dialog that appears.
8. When you are done, press the **Save** button to save the template file to disk.



## Creating new projects using an existing template

To create a new SNIFF+ project using an existing template:

1. In the Launch Pad, choose **Project > New Project... > with Template...**

The Project Template dialog appears.

2. The default template file directory is your `<sniff_installation_dir>/config/project` directory. To select a different template file directory, press the **Change Directory** button and use the Directory dialog that appears.

3. Choose a template from the list of available template files.

The name of the selected template appears in the **Template File** field. Its description appears in the **Description** field.

4. In the **Project Directory** field, enter the root directory of the new project. To navigate to the correct directory, press the **Change Directory** button and use the Directory dialog that appears.

5. Press the **Ok** button.

The Attributes of a New Project dialog appears. The attributes stored in the selected template are used for the various fields and boxes in the dialog. Note that some of the attributes in the dialog have no corresponding attributes in templates. In such cases, default values are taken from your Preferences.

### Note

If the root directory of the new project is physically located in a SNIFF+ working environment, the new project will be a shared project by default. Then, to open this project, you must open it in this working environment, or in another working environment that accesses this working environment.

You may, however, choose to make the new project an absolute project, meaning that you can open it independently of a working environment. To do so, in the **General** view, **File Type** drop-down, choose **Absolute**. Please note that the **Relative** option is for backward compatibility only.

6. When you have set all the attributes in the Attributes of a New Project dialog to your satisfaction, press the **Ok** button.

SNIFF+ will now create the new project and all its subprojects (if it has any). When SNIFF+ is finished, it opens the new project and displays its structure and contents in the Project Editor. The attributes of the project and its subprojects will correspond to those stored in the template.

## Editing templates

You can edit template files in the same way that you edit a SNIFF+ project's attributes. For information on editing project attributes, please refer to *Reference Guide — Project Attributes*. To edit a template file:

- In the Launch Pad, choose **Project > New Project... > with Template...**

The Project Template Dialog appears.

### In the Project Template Dialog

1. Press the **Change Directory...** button to the right of the **Directory** field.
2. In the **Directory Name** dialog that appears, navigate to the directory in which the template file is stored and press the **Select** button.
3. Highlight the template file in the main view.  
The name and description of the template file is automatically entered in the **Template File** and **Description** fields.
4. Enter any path in the **Project Directory** field and press **Ok**.  
The Attributes of New Project dialog appears.

### In the Attributes of New Project dialog

1. Modify the attributes according to your needs.
2. In the General view, under **Save Current Project Attribute Settings**, select **As Template**.
3. Press the **Ok** button to apply your changes.  
The Project Template Dialog appears.

### In the Project Template Dialog

Saving the template file using the same name:

- Follow the first 3 steps in the Project Template Dialog above and press **Save**.

The template file is now saved under the same name as the template file you modified.

Saving the template file using a different name:

1. Press the **Change Directory...** button to the right of the **Directory** field.
2. In the **Directory Name** dialog that appears, navigate to the directory in which you want to save the template file and press **Select**.
3. In the **Template File** field, enter a name for the template file.
4. In the **Description** field, enter a description of the template file.
5. Press **Save**.

The template file is now saved under a different name.

## Specifying a Working Environment Configuration Directory

The `$SNIFF_DIR/workingenvs` directory is the default Working Environment Configuration Directory. There are three reasons for not using this directory:

- Normally, not everyone in your development team may have write permission for the `$SNIFF_DIR/workingenvs` directory.
- To make it easier to manage working environments and the projects that you open in them, we recommend that you store the Working Environment Configuration Directory directory under the same root directory as the corresponding working environments.
- You may want to keep your SNIFF+ installation in its original state without ever modifying it.

## Using the default Working Environment Configuration Directory

In order for your team members to work with shared projects in working environments, read and write permissions must be properly set for a number of files and directories maintained by SNIFF+.

1. Make sure that all members have read and write permissions for the following files and directories:
  - `$SNIFF_DIR/workingenvs/`
  - `$SNIFF_DIR/workingenvs/.WEProjectCache/`
  - `$SNIFF_DIR/workingenvs/WorkingEnvData.sniff`
2. Make sure that only the Working Environments Administrator has write permissions for the following files:
  - `$SNIFF_DIR/workingenvs/WorkingEnvUser.sniff`
  - `$SNIFF_DIR/SitePrefs.sniff`

## Using a different Working Environment Configuration Directory

To specify the Working Environment Configuration Directory:

### In the Shell or Command Prompt

1. Create a directory called e.g. `workingenvs`. From now on, we will refer to the full path to this directory as:
 

```
<workingenvs>
```
2. Start SNIFF+.
 

The Launch Pad appears.

### In SNIFF+

1. Open your Preferences by choosing **Tools > Preferences...** in any open tool.
2. Under the **Tools** node, select **Working Environments**.

3. In the **Working Environment Config. Directory** field, enter the path to the `workingenvs` directory you just created. Press **Ok** to save and apply the changes.  
SNIFF+ will now create a number of files in `<workingenvs>`. These files will hold your working environment information.

## In the Shell or Command Prompt

In order for your team members to work with projects in working environments, read and write permissions must be properly set for the files in the Working Environment Configuration Directory.

1. Make sure that all members have read and write permissions for the following files and directories:
  - `<workingenvs>`
  - `workingenvs>/WEProjectCache/`
  - `workingenvs>/WorkingEnvData.sniff`
2. Make sure that only the Working Environments Administrator has write permissions for the following files:
  - `workingenvs>/WorkingEnvUser.sniff`
  - `$SNIFF_DIR/SitePrefs.sniff`

## Specifying a default working environment

You can specify your default working environment either in your Preferences or directly in the Working Environment tool. Here, we show you how to do so in your Preferences.

1. Ask your Working Environments Administrator the name and location of your Working Environment Configuration Directory. You will specify this directory in the **Working Environment Config. Directory** field of your Preferences.

See also [Specifying a Working Environment Configuration Directory — page 51](#).

2. Choose **Tools > Preferences...** from any open SNIFF+ tool.  
The Preferences dialog appears.
3. Under the **Tools** node, select **Working Environments**.
4. In the **Working Environment Config. Directory** field, specify the Working Environment Configuration Directory.
5. In the **Default Working Environments State** field, enter the name of the working environment in which you normally work.
6. Press **Ok** to save and apply the changes.

You can now open projects directly from the Launch Pad without first specifying your working environment, since SNIFF+ will use your default working environment.

## Initializing team working environments

In a team development situation, your Working Environments Administrator should initialize the RWE and all shared working environments. Each team member can then initialize his/her own PWE.

If you followed the project setup instructions, your team's shared source code is located in your SSWE, and you created a SNiFF+ project (and possibly subprojects) for the source code. To review the setup instructions, please refer to [Project setup overview — procedures — page 44](#).

Now, in order for file sharing between your team's working environments to work, all working environments need to have the same project directory structure. That is, assuming that the SSWE contains the new project directory structure, the

- RWE,
- SOWEs and PWEs

must also have this structure.

The process of copying the SSWE project directory structure is referred to as *initializing working environments*.

Initializing a working environment is an easy process:

- To initialize an RWE, simply check in files for the first time from the SSWE.
- To initialize SOWEs and PWEs, simply open a project in them for the first time. SNiFF+ then alerts you that it can't find the necessary project directories and waits for your confirmation before creating them.

For details about version control in SNiFF+, please refer to [Version Control — page 135](#).

### Important

When you open a project structure (projects that include subprojects) for the first time, you can tell SNiFF+ to copy all necessary project directories at the same time during initialization. We therefore recommend that you open the root project of all the projects in the SSWE.

If you don't open the root project, you will have to repeat the initialization procedure for all projects you haven't yet opened in your working environment.

We also recommend using the root project when updating your working environments later on. In this case, they are referred to as *workspace projects*. To learn about workspace projects, please refer to [Workspace projects — page 162](#).

## Initializing your team's Repository

To initialize your team's Repository, you check in your team's source files for the first time from the SSWE. As a result of this initial check-in, your team's Repository will have the same directory structure as the SSWE. For each project directory, SNiFF+ creates a subdirectory named after your underlying version control tool (e.g. RCS). Version control information for the files in the project are then stored in this directory.

Checking in project files for the first time is the first step in version-controlling your SNiFF+ projects. We recommend that you version control at least the following types of files:

- Project Description Files (PDFs)
- source and source code documentation files
- Makefiles (only if you don't use SNiFF+'s Make Support)

### Initializing your team's Repository — procedures

1. Open the root project in your team's SSWE. Then, complete the remaining steps in the Project Editor.

(To open the root project, please refer to [Opening the root project in a working environment — page 56](#).)

2. In the Project Tree, checkmark all projects by right-clicking anywhere in the Project Tree, and then choosing **Context menu > Select From All Projects**.
3. Press the **Filters...** button.

The Filters dialog appears. You will now filter out SNiFF+'s Makefiles from the Project Editor's File List.

4. In the **FileTypes** tab, clear the **Make** check box and press **Ok**.

SNiFF+'s Makefiles are generated and maintained by SNiFF+, so there's no reason to version control them.

5. Choose **File > Select All**.

6. Choose **File > Check In....**

An alert dialog appears informing you that SNiFF+ could not find a Repository directory for the Project in the Repository Working Environment (RWE). This dialog will reappear for each new Repository directory, unless you enable **Repeat**.

7. Enable **Repeat** and press **Yes** to create the necessary Repository directories for the project.

The Check In dialog appears.

8. In this dialog, press the **Ok** button to check in an initial version of the shared project.

SNiFF+ will now create the SSWE's directory structure in the RWE. Your team's files will be checked in to your Repository during the process. This may take some time, depending on the number of files you're checking in. After the check-in process is over, all files shown in your Project Editor's File List will be read-only.

9. Close the shared project in the SSWE.

## Initializing your team's SOWE

For single-platform development, complete the following steps once. For multi-platform development, complete the following steps for each target platform.

1. Open the root project in the SOWE for the target platform. Then, complete the remaining steps in the Project Editor.

(To open the root project, please refer to [Opening the root project in a working environment — page 56.](#))

SNiFF+ informs you that it cannot find the directories of the shared project in the SOWE root directory (they haven't been created yet). You will now have SNiFF+ copy the SSWE project directory structure in the SOWE.

2. Enable **Repeat** and press **Create Directory**. This will save you from having to press **Create Directory** for each new project directory.

SNiFF+ will now incrementally create the project directories. When done, the SOWE root directory contains the same project directory structure as the SSWE.

SNiFF+ also creates a Project Makefile in each project directory. On Unix, the Project Makefile will be a symbolic link to the corresponding Project Makefile in the SSWE. On Windows NT/95, a local copy is made instead.

When the generation process is over, SNiFF+ automatically opens the project in the SOWE.

3. Close the shared project in the SOWE.

Your SOWE is now initialized for the project.

4. Complete the above steps for all additional target platforms.

When you are done, the next step is to initialize each team member's PWE. Basically, you will go through the same procedure for each PWE as you just did for your team's SOWEs.

## Initializing a PWE

The goal of this step is create the SSWE project directory structure in your PWE. SNIFF+ does this for you when you open the shared project in your PWE for the first time.

1. Open the root project in your team's PWE. Then, complete the remaining steps in the Project Editor.

(To open the root project, please refer to [Opening the root project in a working environment — page 56.](#))

SNIFF+ informs you that it cannot find the directories of the shared project in the PWE root directory (they haven't been created yet). You will now have SNIFF+ copy the SSWE project directory structure into the PWE.

2. Enable **Repeat** and press **Create Directory**. This will save you from having to press **Create Directory** for each new project directory.

SNIFF+ will now incrementally create the project directories. When done, the PWE root directory contains the same project directory structure as the SSWE.

SNIFF+ also creates a Project Makefile in each project directory. On Unix, the Project Makefile will be a symbolic link to the corresponding Project Makefile in the SSWE. On Windows NT/95, a local copy is made instead.

When the generation process is over, SNIFF+ automatically opens the project in the PWE.

3. Close the shared project in the PWE.

Your PWE is now initialized for the project.

## Opening the root project in a working environment

1. Open the Working Environments tool.
2. In the WorkingEnvs Tree, double-click on the working environment in which you want to open the root project. For multi-platform development, double-click on the SOWE for the target platform.

The Open Project dialog appears.

3. If the Project List is initially empty, press the **Update List** button to display all the projects that can be opened in the SOWE.

A dialog appears asking you whether SNIFF+ should also look in any accessed working environments for projects that can be opened.

4. Press **Yes**.
5. Double-click on your shared project to open it.



# Setting Up Team Working Environments

---

# 7

## Introduction

In this chapter, you will learn how to manually set up working environments for your team projects. This chapter covers the setup procedure for all languages **except for** Java. To learn how to set up working environments for your team Java projects, please refer to the *SNiFF+ Java Tutorial*.

## Assumptions made in this chapter

- You are not using SNiFF+'s Project Setup Wizard
- You have read [Project Setup Overview — page 43](#)
- Your team's shared source code is compilable
- You are the Working Environments Administrator for your team
- You want to create and use a Repository for version control purposes
- You are creating working environments from scratch and are not adding or modifying existing working environments
- You develop either for a single platform or for multiple platforms

## Related SNiFF+ topics

- Creating shared projects in working environments — [Creating Team Projects — page 63](#)
- Configuration management and version control in SNiFF+ — [Version Control — page 135](#)
- How to use the Working Environments tool — [Working Environments — page 239](#)

## Abbreviations and shortcuts used in this chapter

RWE — Repository Working Environment  
SSWE — Shared Source Working Environment  
SOWE — Shared Object Working Environment  
PWE — Private Working Environment  
\$SNiFF\_DIR — path to your SNiFF+ installation directory

## Overview

Setting up working environments consists of the following numbered steps:

1. Creating working environment root directories on your file system and setting permissions for them
2. Setting permissions for working environment files created and maintained by SNIFF+
3. Creating and setting up your team's working environments

## Step 1: Create root directories

1. On your file system, create root directories for the RWE and SOWE. Also create a root directory for each PWE needed by your team.
2. If you want SNIFF+ to create shared projects in your team's current shared source file directory, use this directory as the root directory of your team's SSWE.
3. If you first want to create a new directory and copy the current set of your source files and directories into it, do so now. Use this new directory as the root directory of your team's SSWE.

In the rest of this chapter, we will often use the following notation when referring to your working environment root directories:

- `<RWE_root_directory>` — RWE root directory
  - `<SSWE_root_directory>` — SSWE root directory
  - `<SOWE_root_directory>` — SOWE root directory
  - `<PWE_root_directory>` — PWE root directory
4. Give all members of your team read-only permissions for the SSWE and SOWE root directories.
  5. Give all members of your development team write permissions for the RWE root directory.  
Write permission for the RWE is necessary for performing version-control operations on files in your Repository.

## Step 2: Set permissions for working environment files

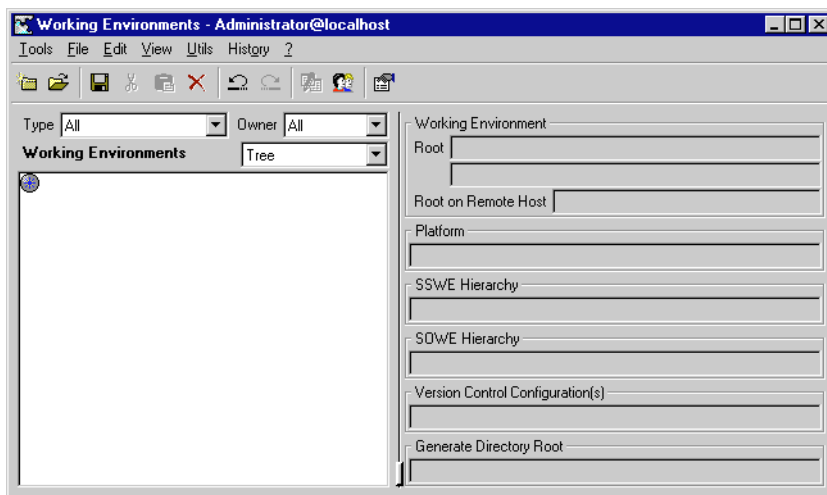
For your team members to work with shared projects in working environments, read and write permissions must be properly set for a number of files and directories maintained by SNIFF+.

1. Make sure that all members have read and write permissions for the following files and directories:
  - `$SNIFF_DIR/workingenvs/`
  - `$SNIFF_DIR/workingenvs/.WEProjectCache/`
  - `$SNIFF_DIR/workingenvs/WorkingEnvData.sniff`

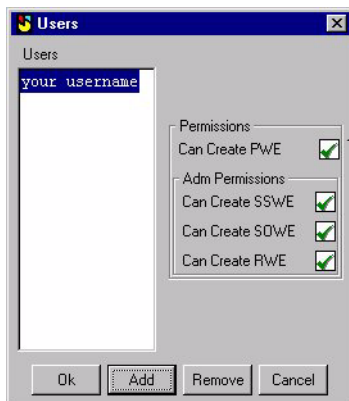
2. Make sure that only the Working Environments Administrator has write permissions for the following files:
  - \$SNIFF\_DIR/workingenvs/WorkingEnvUser.sniff
  - \$SNIFF\_DIR/SitePrefs.sniff

## Step 3: Create and set up team working environments

1. Launch SNIFF+.
2. In SNIFF+'s Launch Pad, choose **Tools > Working Environments**.
3. The main view of the Working Environments tool appears.



4. In the Working Environments tool, choose **Utils... > User Permissions...**.  
The Users dialog appears.



Permissions field

Select all check boxes to give yourself permission to create all four kinds of working environments

5. Check whether your `username` appears in the Users List. If it doesn't, add it to the list by pressing the **Add** command. In the New User dialog that appears, enter your `username` and press **Ok**.

Your `username` should appear highlighted in Users List.

#### Note

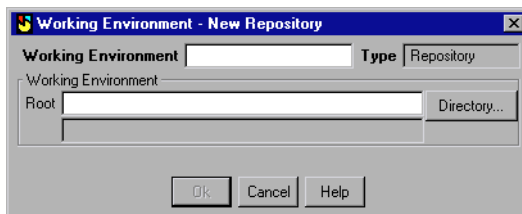
On Unix and Windows NT/95, `username` refers to the name that you use to log onto your machine. SNIFF+ needs it for handling permissions correctly.

6. Under **Permissions**, select all check boxes to give yourself permission to create all four kinds of working environments.
7. Now, add the `username` of your team members in the New User dialog.
8. For each team member, select the **Can Create PWE** check box to give the team member permission to create his/her own PWE.
9. Press **OK** to return to the main view of the Working Environments tool.

## Create and set up the RWE

1. In the WorkingEnvs Tree, select the asterisk (\*).
2. Choose **Edit > New Repository**.

The Working Environment - New Repository dialog appears.



3. In the **Working Environment** field, enter a name for the RWE.
4. Enter the root directory of your Repository in the **Root** field. If you want to use the Directory dialog to select the root directory, press the **Directory...** button.
5. Press **OK** to return to the main view of the Working Environments tool.

You have just finished defining your team's RWE.

## Create and set up the SSWE

1. In the WorkingEnvs Tree, select the RWE you just defined.
2. Choose **Edit > New Shared Source based on Repository**.  
The Working Environment - New Shared Source based on Repository dialog appears.
3. In the **Working Environment** field, enter a name for the SSWE.
4. Enter the root directory of the SSWE in the **Root** field. If you want to use the Directory dialog to select the root directory, press the **Directory...** button.
5. Press **OK** to return to the main view of the Working Environments tool.

## Create and set up the SOWE

For single-platform development, complete the following steps once. For multi-platform development, complete the following steps for each target platform.

1. In the WorkingEnvs Tree, select the SSWE you just defined.
2. Choose the **Edit > New Shared Object based on SSWE**.  
The Working Environment - New Shared Object based on SSWE dialog appears.
3. In the **Working Environment** field, enter a name for the SOWE.
4. Enter the root directory of the SOWE in the **Root** field. If you want to use the Directory dialog to select the root directory, press the **Directory...** button.
5. If you have multi-platform shared projects, you will need subdirectories under the SOWE root directory for each target platform. To create platform specific subdirectories (if necessary), enter `$PLATFORM` at the end of the SOWE root directory in the **Root** field.

### On Unix

The `PLATFORM` environment variable is set automatically each time you start SNIFF+.

### On Windows

The `PLATFORM` environment variable is not set automatically. You must set the `PLATFORM` environment variable manually before using it. To do so, please refer to the Windows online help.

This will enable SNIFF+'s working environment update mechanism to find the appropriate subdirectory of the SOWE root directory for each target platform.

6. Press **OK** to return to the main view of the Working Environments tool.

## Create and set up PWEs

When a new team member joins your development team, a Private Working Environment must be created and set up for him/her. Although each member may complete the following steps, we recommend that the Working Environments Administrator do so. In more complicated development environments, only the Working Environments Administrator will know enough about a team's projects to decide where a new PWE fits into the overall scheme of things.

1. In the WorkingEnvs Tree, select the SOWE defined in the previous section.

2. Choose **Edit > New Private based on SOWE**.

The Working Environment - New Private based on SOWE dialog appears.

3. In the **Working Environment** field, enter a name for the PWE.

4. Enter the root directory of the PWE in the **Root** field. If you want to use the Directory dialog to select the root directory, press the **Directory...** button.

5. Press **OK** to return to the main view of the Working Environments tool.

6. Save your working environments information by choosing **File > Save**.

## Introduction

In this chapter, you will learn how to create shared SNIFF+ projects for a development team situation. This chapter covers the setup procedure for all languages **except for** Java. To learn how to set up shared Java projects, please refer to the *SNIFF+ Java Tutorial*.

### This chapter covers the following topics

- Create shared projects for each directory in your team's Shared Source Working Environment
- Initialize a new team Repository
- Initialize your team's Shared Object Working Environment
- Initialize each team member's Private Working Environment

### Assumptions made in this chapter

- You are not using SNIFF+'s Project Setup Wizard
- You have read [Project Setup Overview — page 43](#)
- Your team's shared source code is compilable
- You have already set up your team's working environments
- You are the Working Environments Administrator for your team
- You have set up a Private Working Environment for yourself

### Related SNIFF+ topics

- Setting up Make Support for team development — [Build and Make Support — page 73](#)
- Configuration management and version control in SNIFF+ — [Version Control — page 135](#)

### Abbreviations used in this chapter

RWE — Repository Working Environment  
SSWE — Shared Source Working Environment  
SOWE — Shared Object Working Environment  
PWE — Private Working Environment  
PDF — Project Description File

## Overview

Creating shared team projects consists of the following steps:

1. Creating shared projects for each directory in your team's SSWE
2. Initializing your Repository
3. Initializing your team's SOWE and then performing an initial build in it
4. Initializing each team member's PWE

## Step 1: Creating shared projects in the SSWE

1. In the Working Environments tool, select your team's SSWE and choose **File > New Project... >With Defaults...**

The Directory dialog appears.

2. In the Directory dialog, select the root directory of the SSWE and press the **Select** button.

The Attributes of a New Project dialog appears. In the dialog, you will set the most important project attributes needed for creating a new project.

### Note

The setup procedure described here does not cover setting up SNIFF+'s Make Support. Before building your team projects, we strongly recommend that you read chapter [Build and Make Support — page 73](#).

3. Select the **General** node and look at the **File Name** field. By default, the project name is the same as the project directory name. You can change the project name at this point if you want.
4. If you use a Make System other than the SNIFF+ Make System, select the **Build Options** node and in the **Make Command** field enter the command that you use to call Make.
5. Select the **Version Control System** node and select your underlying CMVC tool from the **VCS Tool** drop-down menu.
6. Select the **File Types** node and make sure the file types you need are loaded. Create new file types if necessary. For details on how to create new file types, please refer to [page 133](#).
7. Press the **Ok** button to begin creating the shared project.

SNIFF+ will now create the new project and all its subprojects. When SNIFF+ is finished, it opens the new project in the SSWE (where you set up the project) and displays its structure and contents in the Project Editor.



## Step 2: Initializing your team's Repository

The next step is to initialize your team's Repository. You do this by checking in your source files for the first time from the SSWE. As a result of this initial check-in, your team's Repository will have the same directory structure as the SSWE. For each project directory, SNIFF+ creates a subdirectory named after your underlying version control tool (e.g. RCS). Version control information for the files in the project are then stored in this directory.

To check in the entire project, do the following in the Project Editor:

1. Checkmark all the projects in the Project Tree.

2. Press the **Filters...** button.

The **Filters...** dialog appears.

3. In the File Types tab, clear the **Make** check box to filter out SNIFF+'s Makefiles from the Project Editor's File List (we assume you are using SNIFF+'s Make Support).

SNIFF+'s Make Support files are generated and maintained by SNIFF+, so there's no reason to version control them.

4. Press the **OK** button to apply changes and to close the **Filters...** dialog.

5. Choose **File > Select all**.

All the files in the File List are now selected.

6. Choose **File > Check In....**

SNIFF+ informs you that it cannot find the directory structure of the SSWE in the Repository (it hasn't been created yet). You will now tell SNIFF+ to copy the SSWE project directory structure into the Repository.

7. Enable **Repeat** and press **Yes**. This will save you from having to press **Yes** for each new project directory.

The Check In dialog appears.

8. In this dialog, press the **Ok** button to check in an initial version of the shared project.

SNIFF+ will now create the SSWE's directory structure in the RWE. Your team's files will be checked in to your Repository during the process. This may take some time, depending on the number of files you're checking in. After the check-in process is over, all files shown in your Project Editor's File List will be read-only.

9. In the Launch Pad, select the project and choose **Project > Close Project** to close the shared project in the SSWE.

## Step 3: Initializing your team's SOWE

The goal of this step is create the SSWE directory structure in the SOWE. SNIFF+ does this for you when you open the shared project in the SOWE for the first time.

For single-platform development, complete the following steps once. For multi-platform development, complete the following steps for each target platform.

1. Open the Working Environments tool.
2. Double-click on your team's SOWE in the hierarchy of available working environments. For multi-platform development, double-click on the SOWE for the target platform.

The Open Project dialog appears.

3. If the Project List is initially empty, press the **Update List** button to display all the projects that can be opened in the SOWE.

A dialog appears asking you whether SNIFF+ should also look in any accessed working environments for projects that can be opened in the SOWE. Here, the SOWE accesses the SSWE, so pressing **Yes** will display also the projects in the SSWE.

4. Press **Yes**.
5. Select the root shared project (all other projects are its subprojects) and press **Open**.

SNIFF+ informs you that it cannot find the directories of the shared project in the SOWE root directory (they haven't been created yet). You will now have SNIFF+ copy the SSWE project directory structure into the SOWE.

6. Enable **Repeat** and press **Create Directory**. This will save you from having to press **Create Directory** for each new project directory.

SNIFF+ will now incrementally create the project directories. When done, the SOWE root directory contains the same project directory structure as the SSWE.

SNIFF+ also creates a Project Makefile in each project directory. On Unix, the Project Makefile will be a symbolic link to the corresponding Project Makefile in the SSWE. On Windows NT/95, a local copy is made instead.

When SNIFF+ is finished, it opens the new project in the SOWE and displays its structure and contents in the Project Editor.

7. In the Launch Pad, select the shared project and then choose **Project > Close Project** to close the shared project in the SOWE.

Your SOWE is now set up. The next step is to initialize each team member's PWE. Basically, you will go through the same procedure for each PWE as you just did for the SOWE.

## Step 4: Initializing your PWE

The goal of this step is create the SSWE directory structure in your PWE. SNiFF+ does this for you when you open the shared project in the PWE for the first time.

### Note

For now, you should complete the following steps only for your PWE. After reading and following the steps outlined in the next and last section ([What you should do next — page 68](#)), go through test steps again for your team members' PWEs.

1. Open the Working Environments tool.
2. Double-click on your PWE in the hierarchy of available working environments.  
The Open Project dialog appears.
3. If the Project List is initially empty, press the **Update List** button to display all the projects that can be opened in the PWE.  
A dialog appears asking you whether SNiFF+ should also look in any accessed working environments for projects that can be opened in the PWE. Here, the PWE accesses the SOWE and the SSWE, so pressing **Yes** will display also the projects in both of these working environments.
4. Press **Yes**.
5. Select the root shared project (all other projects are its subprojects) and press **Open**.  
SNiFF+ informs you that it cannot find the directories of the shared project in the PWE root directory (they haven't been created yet). You will now have SNiFF+ copy the SSWE project directory structure into the PWE.
6. Enable **Repeat** and press **Create Directory**. This will save you from having to press **Create Directory** for each new project directory.  
SNiFF+ will now incrementally create the project directories. When done, the PWE root directory contains the same project directory structure as the SSWE.  
SNiFF+ also creates a Project Makefile in each project directory. On Unix, the Project Makefile will be a symbolic link to the corresponding Project Makefile in the SSWE. On Windows NT/95, a local copy is made instead.  
When SNiFF+ is finished, it opens the new project in the PWE and displays its structure and contents in the Project Editor.
7. In the Launch Pad, select the shared project and then choose **Project > Close Project** to close the shared project in the PWE.  
Your PWE is now set up. This completes the setup procedure for your shared team projects.

## What you should do next

The next tasks that you, the Working Environments Administrator for your team, should complete are:

1. Set up SNIFF+'s Make Support for your shared project (covered in [Build and Make Support — page 73](#)).
2. Build the targets of your shared project to check whether you set everything up properly. Note that we assume that your shared source files are compilable, so any compiler errors that might occur during the build process are a result of improperly set Make attributes.

In the rest of this chapter, we suggest two ways of doing this.

## Method 1 — Working in your PWE

You can set up Make Support in your PWE and then locally build the targets of your project.

### Advantage

You would use SNIFF+'s team support as it is intended to be used:

### Procedure

1. Check out files that you want to modify in your PWE—Setting up Make Support for a project means setting the project's Make attributes. These attributes, like all other project attributes, are stored in the project's PDF. As part of the shared project setup process, you should check in PDFs along with your other files, so you need to check them out before modifying them.
2. Set up Make Support for the shared project in your PWE.
3. Test the modifications by building the targets of your project in your PWE.
4. If the build is successful, check in the modified PDFs. If the build is unsuccessful, correct any mistakes in your Make attributes and then rebuild the targets.
5. Update your team's SSWE and SOWE. By updating the SSWE, your changes will be available to the rest of your team. By updating the SOWE, the targets of your project will be successfully build in it.

## Method 2 — Working in your team's SOWE

You can set up Make Support in your SOWE and then build the targets of your project in it.

### Advantage

It's generally faster than Method 1, since you have to update only one working environment (your team's SSWE).

### Disadvantage (On Windows NT/95 only)

You must first check in your shared source files from the SSWE before opening the shared project in the SOWE. If you don't do this and later open the shared project in a PWE, the permissions for your Project Makefiles will be improperly set, and local builds in the PWE will fail.

### Procedure

You can work in the SOWE just like you would in a PWE:

1. Check out your PDFs in the SOWE.
2. Set up Make Support and build the targets of your project in the SOWE.
3. If the build is successful, check in the modified PDFs. If the build is unsuccessful, correct any mistakes in your Make attributes and then rebuild the targets.
4. Update your team's SSWE. By updating the SSWE, your changes will be available to the rest of your team.



# Part IV

## Setting Up the Build Process





# Build and Make Support

---

## Introduction

In this chapter, you will learn how to set up SNIFF+'s Make Support for your projects. This chapter covers the setup procedure for all languages **except for** Java. To learn how to set up Make Support for your Java projects, please refer to the *SNIFF+ Java Tutorial*.

### Note

You must use SNIFF+'s Make Support if you work in a team environment and use working environments.

## This chapter covers the following topics

- How SNIFF+'s Make Support works
- How to set up SNIFF+'s Make Support for your projects

## Assumptions made in this chapter

- You intend to use SNIFF+'s Makefiles and Make Support Files for regulating builds
- You know how to set up working environments and shared projects
- You use a version control tool

## Related SNIFF+ topics

- Executing SNIFF+ commands for building, running and debugging targets — [Compiling and Debugging in SNIFF+ — page 185](#).
- Using your own Makefiles in SNIFF+ — [Using Your Own Makefiles — page 105](#)

## Abbreviations and shortcuts used in this chapter

SSWE — Shared Source Working Environment

SOWE — Shared Object Working Environment

PWE — Private Working Environment

\$SNIFF\_DIR — path to your SNIFF+ installation directory

## Technical overview

The primary reason for the interdicting an automated Makefile generation feature in SNiFF+ was to "make" projects more manageable and easier to maintain from a build perspective. When new files or targets are added to a project, developers typically don't want to be involved in the modification of Makefiles because Makefiles tend to be very complex and changing Makefiles could have wide ranging side-effects on the overall system.

SNiFF+ knows which source files belong to a project and extracts include dependencies for browsing purposes. Hence, SNiFF+ could easily use its symbol database together with generic rules about how to build executables, libraries, etc. to generate Makefiles. SNiFF+ can derive file dependencies of source and header files, generate include paths for header files that are part of the project structure, and automatically update Makefiles when new files are added to a project.

Each time a SNiFF+ project is created, a project Makefile is created from a Makefile template, `template.Makefile`, located in the `$SNIFF_DIR/config` directory. This Makefile includes Make Support files, e.g. `macros.incl`. As a matter of fact, SNiFF+ never modifies the project Makefile itself, SNiFF+ only updates the Make Support files. These Make Support files located in a subdirectory of each project called `.sniffdir`, should not be modified by the user. The general makefiles, `general.mk`, `general.<language>.mk` and `<platform>.mk` are located in the `$SNIFF_DIR/make_support` directory.

### Note for Windows users

In the following section, there are several references to *symbolic links*. Windows does not, however, support symbolic links. So, wherever symbolic links are created by SNiFF+ on Unix, local copies are made on Windows. Therefore, if you are working on Windows, please read all references to "symbolic links" as "local copies" in the following.

## Features

SNiFF+'s Make Support:

- comes with its own Makefiles
- is based on standard Unix Make tools
- is fully integrated with working environments to build targets across multiple shared working environments
- automatically generates Make Support Files that contain data about include paths and dependencies lists for shared projects
- automatically provides Make rules for recursively building a project's target
- provides automatic support for multi-platform development and works with compilers, linkers, archivers and other build tools of your choice
- maintains your build system by automatically updating Make Support Files

- supports multiple programming languages, e.g., C/C++, IDL, Java, Fortran, etc.
- supports multiple targets, e.g., executables, static and shared libraries, DLL's, applets, etc.

## Specifying your compiler

SNiFF+ does not have its own compiler. To compile in SNiFF+, you must have a compiler installed on your computer. By default, the following compiler is specified in your Platform Makefile:

- **On Unix**

The gnu compiler

- **On Windows**

Microsoft Developer Studio

If you use any other compiler to compile SNiFF+ projects, please specify it in your Platform Makefile. For information about Platform Makefiles, see [Platform Makefile — page 82](#). The Platform Makefile then uses dependency and include path information from your Project Makefile. For information about Project Makefiles, see [Project Makefile — page 77](#). In addition, make sure that your `PATH` environment variable points to the compiler you are using.

## Building targets when using team working environments

If you use SNiFF+'s working environments with your team software development projects, you must use SNiFF+'s Make Support in its entirety (i.e., including Makefiles and Make Support Files) for building your object files and targets.

SNiFF+'s Make Support allows you to take full advantage of working environments by providing a mechanism for automatically sharing source and object files between members of a team. As a result, it is not possible to use "normal" Makefiles with shared working environments.

## Sharing source files

SNiFF+ uses an internal sharing mechanism for source files. This mechanism allows SNiFF+ to know exactly which project source files are shared (only found in shared working environments) and which are checked out to a PWE (contained in local project directories). However, this internal sharing mechanism is not supported by standard Make implementations, which expect that all source files needed to build the object files for the local target are also located in the local directory. If this is not the case, SNiFF+ creates symbolic links in your PWE to shared source files in your team's SSWE where appropriate. These symbolic links are automatically created when you open a project in your PWE or when needed.

When you check out a shared source file to your PWE, SNiFF+ removes the symbolic link and then makes a local copy of the file.

**Note**

Project Makefiles in a PWE are always symbolic links to those in the accessed SSWE, since Make assumes that Makefiles are stored in local directories.

## Sharing object files

SNiFF+ also creates symbolic links in your PWE to shared object files and targets in your team's SOWE. This is done by means of a special help target, `symbolic_links`.

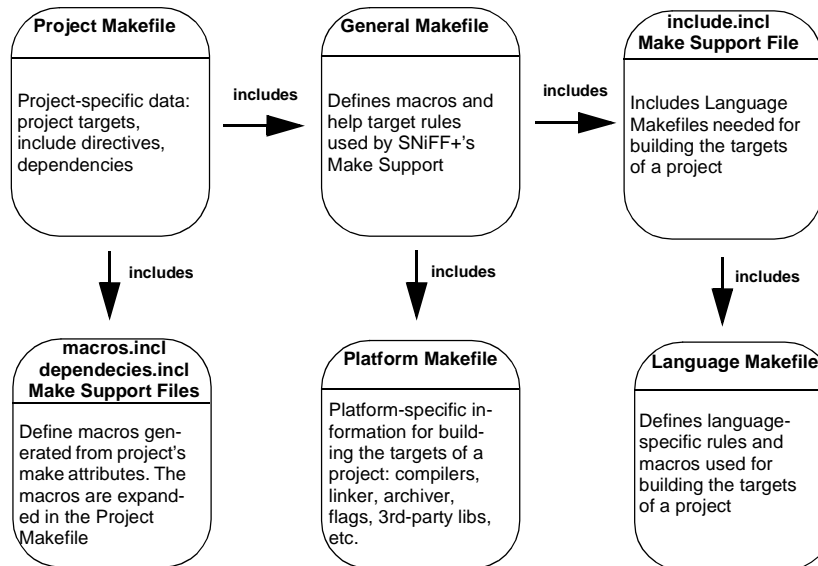
Before compiling source files and building targets locally in your PWE, we recommend that you first build (run) `symbolic_links`. This will reduce disk space and speed up compilation time by eliminating unnecessary recompilation and relinking.

To find out how to build `symbolic_links`, please refer to [Building the help targets — page 191](#).

When you build targets in your PWE, SNiFF+'s Make Support first checks whether any object files need to be rebuilt or whether any targets need to be relinked. If so, SNiFF+ removes it and then rebuilds it (same principle applies to a target that needs to be relinked).

## SNiFF+ Makefiles and Make Support Files

Your SNIFF+ installation comes with a number of standard Makefiles that are used by SNIFF+'s Make Support to communicate with Make. Basically, there are four types of Makefiles: General, Language, Platform and Project. These Makefiles, as well as how they interact with each other, are described in the following diagram:



## Project Makefile

A Project Makefile (from now on: Makefile) is a generic Makefile template that SNIFF+ automatically generates and adds to a newly created project. This Makefile is located in the project directory. Makefiles define macros which are used by SNIFF+'s Make Support for building project targets. These macros, in turn, are expanded to the values of other macros defined in Make Support Files. There are macro definitions, e.g., for:

- the targets built in the project
- the components needed to build the targets
- dependency information
- include path information

Macro definitions are automatically maintained by SNIFF+. The values of the macros used in the Makefile are taken from the **Build Options** view of the Project Attributes dialog.

For details about the macros defined in a Makefile, please refer directly to a project's Makefile. Make Support Files are discussed on [page 80](#).

## Configuring the Project Makefile (excluding Java)

For configuring Make Support for Java, please refer to the Technical Reference in the Java tutorial. In general, it is unnecessary for you to modify a project's Makefile. By modifying the Project Makefile, you can extend or override all settings that you make in the SNIFF+ Project Attributes. However, you should try to avoid such modifications wherever possible because they are usually hard to migrate to new versions of SNIFF+.

All macros defined in a Platform Makefile can be overridden in the Project Makefile, but note that this makes the current project platform dependent. Therefore, we discourage changing any settings in the Project Makefile apart from the examples shown below.

In a project's Makefile, you can also redefine the platform-independent flags and macros that are set in the General Makefile. If you want to incorporate additional Make rules in a project's Makefile, you must do so after the code line that includes the General Makefile:

```
include $(SNIFF_DIR)/make_support/general.mk
```

Additional flags can be defined using macros containing the prefix `OTHER_`. Multiple additional flags are separated from each other by spaces.

Example	Description
<code>Include .sniffmake/macros.incl</code>	If you modified the <b>Directory for Make Support Files</b> in your Project Attributes, then you need to modify the line which includes <code>macros.incl</code> , all future lines will be set correctly through the <code>\$(SNIFF_MAKEDIR)</code> macro. Note that if you generally want to use a different directory for your SNIFF generated Make Support files, you should change the Makefile Template in <code>\$(SNIFF_DIR)/config/template.Makefile</code> .
<code>SUB_LIBS = lib1 lib2 \$(SNIFF_SUB_LIBS)</code>	If your executable depends on linking sub libraries in a particular order, you may add the libraries which should be linked first to the <code>SUB_LIBS</code> macro. This can also be used to force linking to shared libraries which are produced by subprojects. By default, SNIFF+'s Make Support doesn't pass shared libraries to superprojects.
<code>SUBDIRS = my_subdir \$(SNIFF_SUBDIRS)</code>	<b>Additional Recursive Make Subdirectories</b> generated in the SNIFF Project Attributes will be overwritten when you press the <b>Generate</b> button there. To make sure that such subdirectories are persistent, you may add them in your project Makefile.

Example	Description
OTHER_CLEANUP = "*.bak" "*~"	Additional patterns that you would like to remove if a <code>make clean</code> is issued
PURIFY_TARGET = \\$(LINK_TARGET).purify	By defining this macro, you activate the <code>purify</code> rule of the global Make Support for the current project.
OTHER_CXXFLAGS = \\$(CXX_DEBUGFLAG)	Produce debugging information in this project, even if the global options turn it off. Using <code>CXXFLAGS</code> in the project Makefile would overwrite the global settings from the Platform Makefile.
OTHER_IDLFLAGS = -DmyLocalDef	Use a specific Preprocessor definition also for IDL. In general, all <code>OTHER_XXX</code> macros are meant to hold such additional switches which should be valid only in the local project.
IDL_CFILE_TYPE_SPEC = C	Build an IDL Client here. For details, see IDL Setup in the User's Guide.
PRE_TARGETS = my_pretarget \\$(SNIFF_PRETARGETS)	If one target needs to be compiled before all other files in this project, add it to the <code>PRE_TARGETS</code> macro.
MAKEFILE = Makefile.server	If your local Makefile has a different name, you must set this macro in order for <i>make</i> running in a sub-process to work correctly. Note, that in order for <b>Recursive Make</b> to work correctly, you also need a file named <code>Makefile</code> in addition to your renamed one (for details, see IDL Setup in the User's Guide).
all :: additional_stuff additional_stuff : x1.gen x2.gen \\$(GENERATOR) \$*	You can add local rules for additional stuff to the <b>all</b> target if you use the double colon. Your <code>additional_stuff</code> should be defined as other target in the SNIFF Project Attributes. Note: if you define this rule before including <code>general.mk</code> , it will be executed before your SNIFF+ main target, if you add the <b>all</b> rule after including <code>general.mk</code> , it will be executed after the main target and the subdirectories have been built.

## General Makefile

The General Makefile is the `$SNIFF_DIR/make_support/general.mk` file. Each SNIFF+ installation comes with a the General Makefile, which:

- defines a number of macros used by SNIFF+'s Make Support,
- specifies rules for building SNIFF+'s help targets,
- specifies rules for recursively building targets,
- includes the correct Platform Makefile for your target platform, and
- includes the `include.incl` Make Support File.

For details about the macros and rules defined in the General Makefile, please refer directly to the file. For details about help targets, please refer to [SNIFF+ help targets — page 190](#)

## Make Support Files

By default, Make Support Files are stored in the `.sniffdir` subdirectory of each SNIFF+ project directory. You can specify another directory in which SNIFF+ should store generated Make Support files in your **Project Attributes > Build Options > Advanced** view, **Other directory** field. Please note that this path must be relative since absolute paths aren't currently supported. Once you've specified another directory for Make Support files, you must manually enter the path to this directory in your project Makefiles.

SNIFF+ uses a project's Make information, such as include path and dependencies information, to generate Make Support Files. The macros defined in Make Support Files are then used by the different types of SNIFF+ Makefiles to build a project's targets.

There are four Make Support Files:

- **Dependencies file (dependencies.incl)** — Lists all dependencies between source files in the project.
- **Macros file (macros.incl)** — Contains e.g. target names, include path information and linked libraries information for the project.
- **Include file (include.incl)** — Includes those Language Makefiles needed for building the targets of a project.
- **Platform Make Support file (\$PLATFORM.incl)** — Contains information about the path to the redirection directories.

All Make Support Files except for **Include file** are automatically included by Project Makefiles. The **Include file** Make Support File is included by the General Makefile.

For details about the macros defined in Make Support Files, please refer directly to the files.

## Updating Make Support Files

You should update a project's Make Support Files after you've made any structural changes to the project, or whenever you include additional files in its source files.

Note that SNIFF+ automatically updates the **Macros** Make Support File of a project when you do one of the following:



- add or remove files to or from the project
- add or remove subprojects to or from the project
- Modify any Make attribute of the project.

## Language Makefiles

SNiFF+'s Language Makefiles are located in your `$SNiFF_DIR/make_support` directory. For details about Language Makefiles, please refer to [Language Makefiles — details — page 103](#).

Language Makefiles define the language-specific macros and rules needed to build the targets of a project. For details about the macros and rules, please refer directly to the Language Makefiles.

Each language file type has an attribute called **General Makefile**, which specifies the Language Makefile associated with the file type. SNiFF+'s Make Support uses this information to automatically include the correct Language Makefile in the **Include file** Make Support File.

You can use SNiFF+'s default settings for the Language Makefiles it provides, or you can specify your own:

- You can associate more than one Language Makefile to a file type. For example, SNiFF+ associates both yacc and C Language Makefiles with the **Yacc Source** file type. So, when you build targets from yacc source files, rules for compiling both yacc and C sources are made available to your Make utility.
- Since all the rules and macros for a given language are contained in a single Language Makefile, finding and adapting these rules to your exact environment is easy.
- You can write your own Language Makefiles and use them for all your SNiFF+ projects or only for selected projects.

To learn how to specify your own Language Makefiles, please refer to [Specifying Language Makefiles: — page 103](#).

---

### Note

SNiFF+'s Language Makefiles extensively use macros defined by SNiFF+'s Make Support. Therefore, if you intend to create your own language-specific Make rules, please use SNiFF+'s Language Makefiles as templates.

## Platform Makefile

SNiFF+ comes with a set of pre-configured Platform Makefiles for all platforms on which it runs. These Makefiles are stored in your `$SNiFF_DIR/make_support` directory.

Platform Makefiles are included by the General Makefile and define macros for the various build tools needed for each specific platform (e.g., compilers, linker, archiver, preprocessor). The flags corresponding to these tools can also be defined in Platform Makefiles.

For details about which macros and flags can be set in Platform Makefiles, please refer directly to the files.

## Configuring the Platform Makefile

In your Platform Makefiles, you can redefine the macros and rules that are set in the General Makefile for your particular platform, see below:

Example	Description
<code>CXX = CC</code>	Use the SparcWorks native compiler on Solaris
<code>SYSYPE = -DOS_SYSV -DOS_Solaris</code>	Preprocessor Defines that help your sources to identify a platform -- you are free to choose these, they will just be evaluated by the Preprocessor in your Source Code
<code>CXXFLAGS = -O -pte.cc</code>	Global C++ Compiler flags: Optimize and use .cc as extension for templates
<code>CXX_DEBUGFLAG = -g</code>	Define the debugging information flag which is used on this particular platform (for multi-platform use in the Project Makefile)
<code>LD_SHAREDFLAGS = -shared</code>	While GNU compilers use -G for producing shared libraries, the Irix native compiler uses -shared. Producing shared libraries is generally rather system dependent, so you might need to change the corresponding flags in your Platform Makefile to make it work.
<code>SOCKET_LIBS = -lsocket -lnsl</code>	These libs are typically only needed on Solaris. By defining SOCKET_LIBS in the Platform Makefile, you can use \$(SOCKET_LIBS) as a platform-dependent library to be linked to your executables (+libraries linked field in the Project Attributes).
<code>MY_GLOBAL_LIB = -L/lib/mine -lglob</code>	By defining a macro for external additional libraries, you can use it as \$(MY_GLOBAL_LIB) in the Project Attributes (similar to the SOCKET_LIBS, see above)

Example	Description
<pre>YACC_CFILE_SUFFIX = c LEX_CFILE_SUFFIX = c IDL_CFILE_SUFFIX = cpp</pre>	Change the suffix that is used for files generated from the Yacc, Lex or IDL precompiler. Note that you should also change the SNIFF+ filetype for ... <i>generated Implementation</i> files to use the suffix that you have changed.

## Specifying the targets of a project

SNiFF+'s Make Support provides rules in the General Makefile for building the following types of targets in a project:

- executable
- relinkable object
- library
- object file (for the purposes of this section, all object files built in a project are collectively considered to be a single target)

The following rules apply when specifying targets:

- Any number of object files may be built in a project.
- Of the other three target types—executable, relinkable object, library—*only one of each* may be built in a project.

Each project has only one *default target*. If only one of the targets mentioned above is built in a project, this target is the default target of the project. If several targets are built in a project, the first target in the order—executable, relinkable object, library, object file—is the default target of the project.

The default target of a project fulfills the following purposes:

- It is used as the default target for many of the commands available in the **Target** menu.
- Make builds the default target of the current project when you start a build without explicitly entering a target name, or when you execute `make all` in the Shell.
- By properly exporting default targets (see following pages), the structure of your projects in SNiFF+ will be optimally suited for making full use of SNiFF+'s Make Support.

## Exporting targets of a project

SNiFF+'s Make Support provides a mechanism for automatically linking the targets of a project to its superproject. When you use this mechanism, SNiFF+ automatically enters and maintains the necessary link commands in Project Makefiles, thus saving you from having to do this yourself.

A project can *export* its target to its superproject. The superproject can then either link these targets to its own targets during a build, or export them to its own superproject.

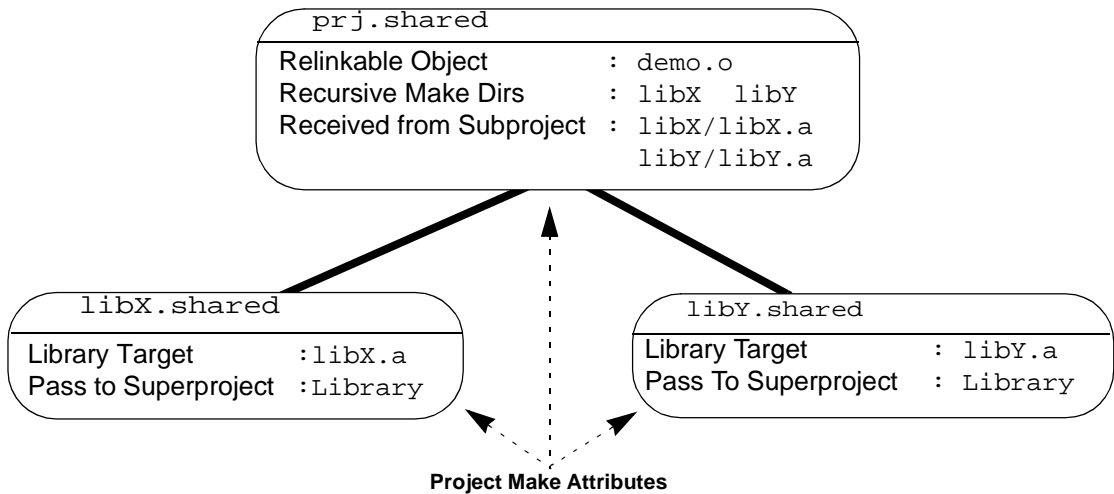
The following types of targets can be exported:

- relinkable object
- library
- object file (for the purposes of this section, all object files built in a project are collectively considered to be a single target)

Although an exported target need not be the default project of a target, it is good practice that it is.

The following examples show how exporting targets works:

## Example 1

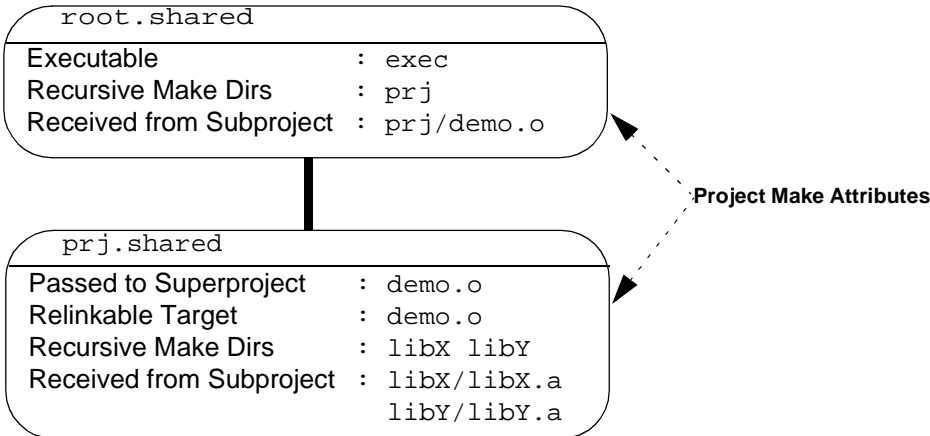


The project `prj.shared` in the above illustration has two subprojects: `libX.shared` and `libY.shared`. Library targets are built in both subprojects and are then exported (passed) to `prj.shared`.

`prj.shared` imports (receives) the two library targets from its subprojects. These library targets are linked to `prj.shared`'s own target, `demo.o` (a relinkable object).

## Example 2

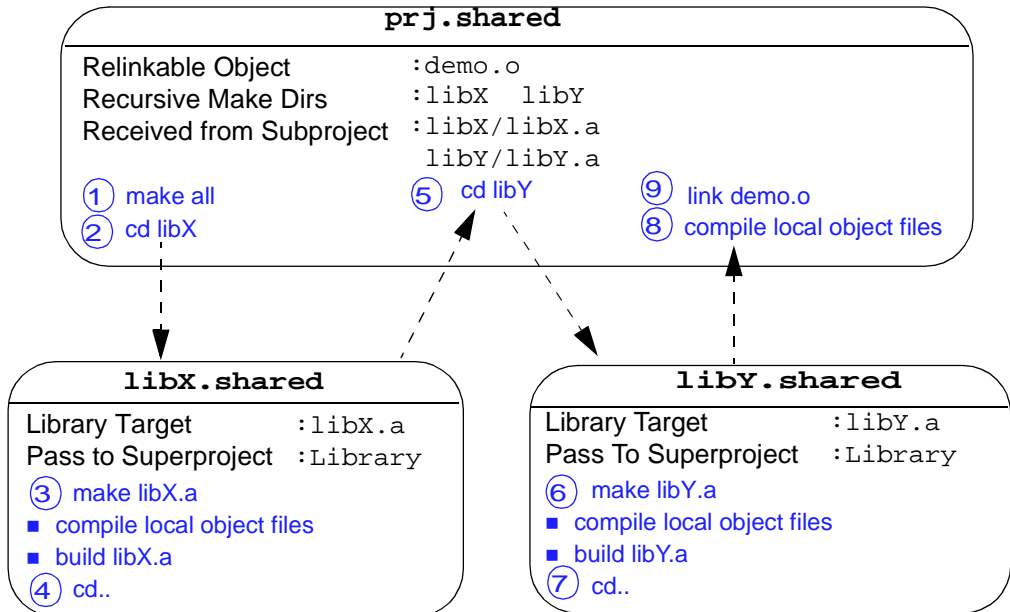
Now suppose `prj.shared` from Example 1 is itself a subproject of another project, `root.shared`, in which the executable `exec` is built. `prj.shared` exports its relinkable object and the two libraries exported from its subprojects to `root.shared`. This is shown in the following illustration:



Several more export scenarios are possible in SNIFF+. These are explained in detail in the section that covers Make Support setup later on in this chapter on [page 88](#).

## Building targets recursively

SNiFF+'s Make Support supports the use of recursive Make rules. Let's use the first illustration from the previous section to show how SNiFF+ handles a recursive Make:



To recursively build a project's target, choose the **Make all** command in the **Target** menu of the Project Editor. SNiFF+ then opens a Shell tool and executes `make all` on the command line. The targets of three projects are then built according to the steps in the above illustration.

## Setting up Make Support

In this section, you will learn how to set up Make Support for your projects. To set up Make Support:

1. Start SNiFF+ and open the project for which you want to set up Make Support.
2. Check out the Project Description Files (PDFs) of all the projects for which you will be building targets.
3. In the Project Tree, checkmark all the projects for which you will be building targets.
4. Choose **Project > Attributes of Checkmarked Projects....**  
The Group Project Attributes dialog appears.
5. Select the **Build Options** node.



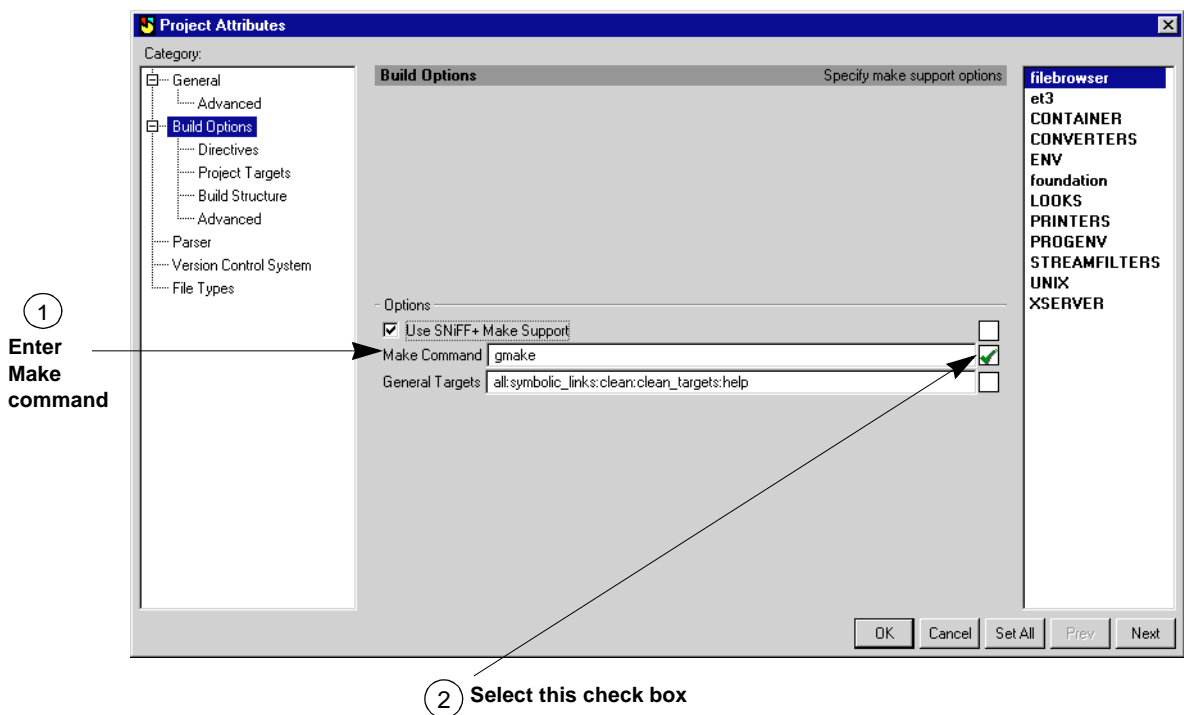
## Setting common Make attributes in the Group Project Attributes dialog

You will now set those Make attributes which are the same for all the projects for which you will be building targets. Basically, to set the attributes, you will perform the following tasks:

- Specify the command you want to use for calling your Make utility in SNIFF+.
- Generate include and dependencies information for the project and its subprojects.

Make attributes in the Group Project Attributes dialog are grouped into 5 main views. In the rest of this section, you will set the attributes, one view at a time.

### In the Build Options view



1. If you use SNIFF+'s Make Support, select the **Use SNIFF+ Make Support** checkbox.
2. If you use a Make System other than the SNIFF+ Make System, in the **Make Command** field enter the command that you use to call Make.

This Make command will then be executed in SNIFF+'s Shell tool when you launch Make in SNIFF+. If you use the SNIFF+ Make System, the default Make command is used.

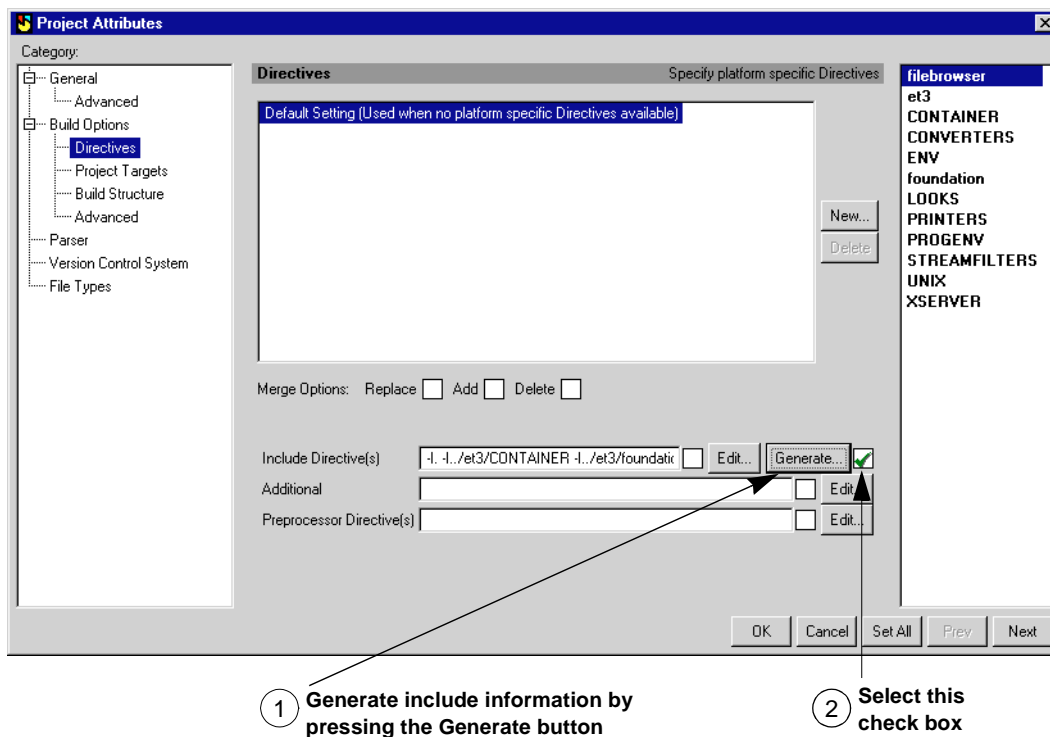
3. Select the check box to the right of the **Make Command** field.

This attribute will now also apply to all projects checkmarked in the Project Tree.

4. Select the **Directives** node.

## In the Directives view

### Setting up the Include Directives



1. Generate include information by pressing the **Generate** button. Then, manually edit the **Include Directive(s)** field by removing the include directives for all subdirectories that your compiler can ignore during file searches.

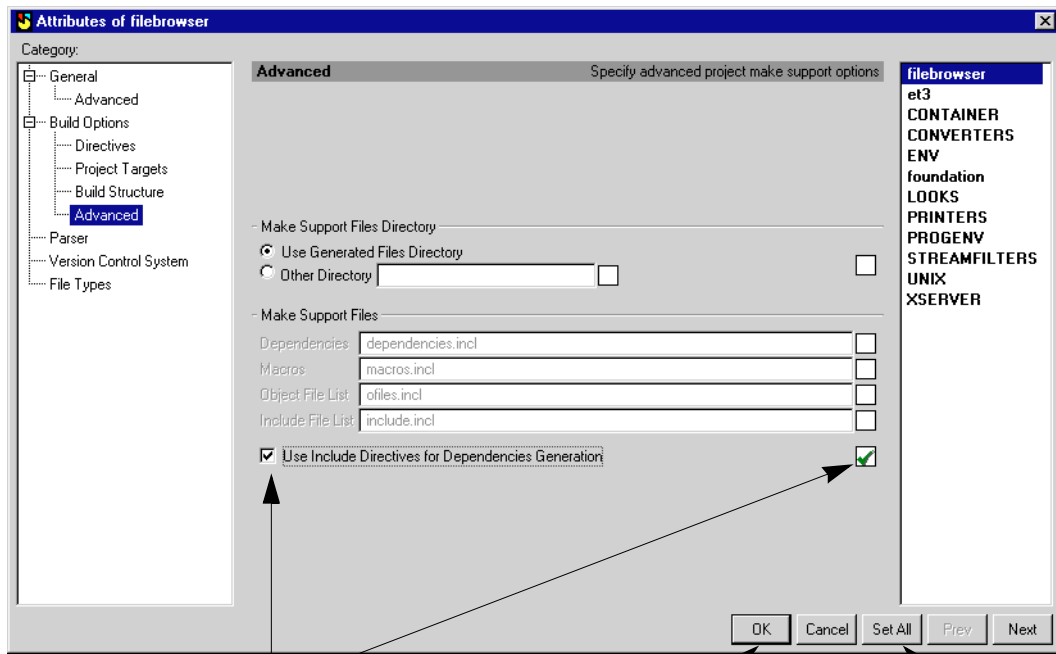
As a rule, you should update your project's include directives whenever you include new files that aren't in the path shown in the **Include Directive(s)** field.

**Note**

- When you press the Generate button SNiFF+ enters include directives for the project directory and all subproject directories in the Include Directive(s) field. SNiFF+ does not check whether all the directories listed in the field actually contain include files or not.
- Include files must be located in the project structure; otherwise, SNiFF+ won't take them into account when updating its Make Support Files. Paths to include files outside of the project structure should be added in the **Additional** field.
- For a large project, we suggest that you make a backup of its include directives. You can find a project's include directives in the `macros.incl` Make Support File in the `.sniffdir` subdirectory of your project directory.

2. Select the check boxes to the right of the **Generate** button.
3. Select the **Advanced** node.

## In the Advanced view



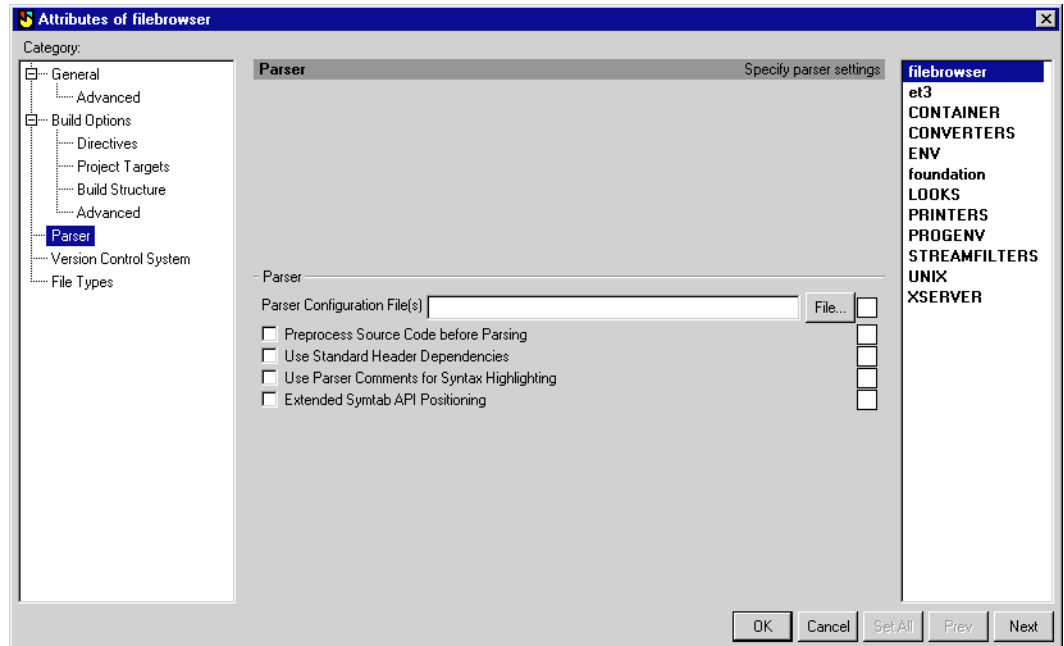
① To optimize how SNIFF+ generates the project's dependencies information, select these two check boxes

③ Press Ok

② Press Set For All to propagate settings to all projects

1. To optimize how SNIFF+ generates the project's dependencies information, select the **Use Include Directives for Dependencies Generation** check box (see [Optimizing how SNIFF+ determines dependencies — page 94](#) for details). Also select the check box to its right.
2. Press the **Set For All** button to apply to all the projects checkmarked in the Project Tree.
3. Select the **Parser** node.

## In the Parser view



1. If you use `<>` instead of `" "` to include header files located in the project structure, select the **Use Standard Header Dependencies** check box. Then, select the check box to its right.

For example, select the check box if you use the following syntax:

```
#include <header_file.h>
```

SNiFF+ will then be able to generate the project's include path information correctly. It does so by treating the above include statement like the following:

```
#include "header_file.h"
```

2. Press the **Set For All** button to apply to all the projects checkmarked in the Project Tree.
3. Press the **Ok** button to apply the settings and to close the Group Project Attributes dialog.

Notice that the Project Tree indicates that all checkmarked projects are modified. We recommend that you save all modified projects at this time.

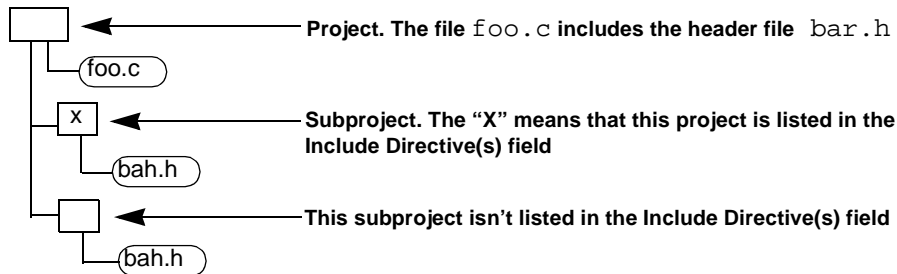
## Optimizing how SNIFF+ determines dependencies

To optimize SNIFF+'s include file search process during the generation of the project's `dependencies.incl` Make Support File, enable the **Use Include Directives for Dependencies Generation** checkbox.

The following diagrams show how SNIFF+ finds include files based on the status of the **Use Include Directives for Dependencies Generation** checkbox.

### Case 1

Suppose your project has a project structure like this one:

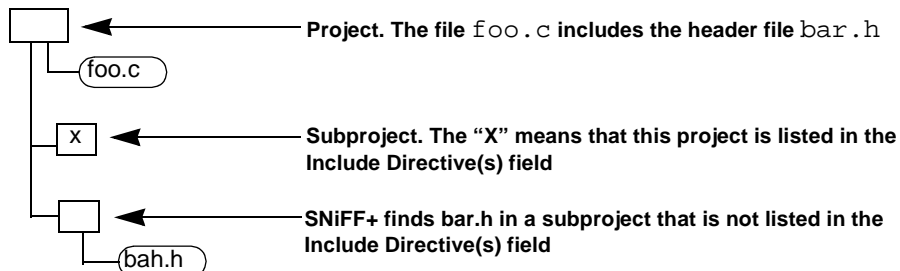


The header file `bar.h` is in two subprojects: in one that is listed in the **Include Directive(s)** field and in one that isn't.

- When the **Use Include Directives for Dependencies Generation** check box is selected, SNIFF+ will search for `bar.h` in the subprojects that are listed in the **Include Directive(s)** field.
- When the **Use Include Directives for Dependencies Generation** check box is not selected, SNIFF+ will search for `bar.h` in the entire Project Tree. SNIFF+ will end the search as soon as it (randomly) finds the first `bar.h` file.

### Case 2

Another possible situation in which this checkbox plays an important role is described in the following illustration.



The header file `bar.h` is in a subproject that isn't listed in the **Include Directive(s)** field.

- When the **Use Include Directives for Dependencies Generation** check box is selected, SNiFF+ will search for `bar.h` in the subprojects that are listed in the **Include Directive(s)** field. Since it won't be able to find `bar.h` in one of these subprojects, SNiFF+ will continue searching in the rest of the Project Tree until it finds the header file. When SNiFF+ finds the header file, it will give you a warning message to inform you that the **Include Directive(s)** field may no longer be up-to-date.
- When the **Use Include Directives for Dependencies Generation** check box is not selected, SNiFF+ will search for `bar.h` in the entire Project Tree and will end the search as soon as it finds the file.

## Project-specific attributes in the Project Attributes dialog

You will now set those Make attributes which are specific to each project for which you will be building targets.

Please complete the following steps **for each project**:

- In the Project Tree, double-click on the project for which you will be setting project-specific Make attributes.

The Project Attributes dialog appears.

### In the Project Attributes dialog

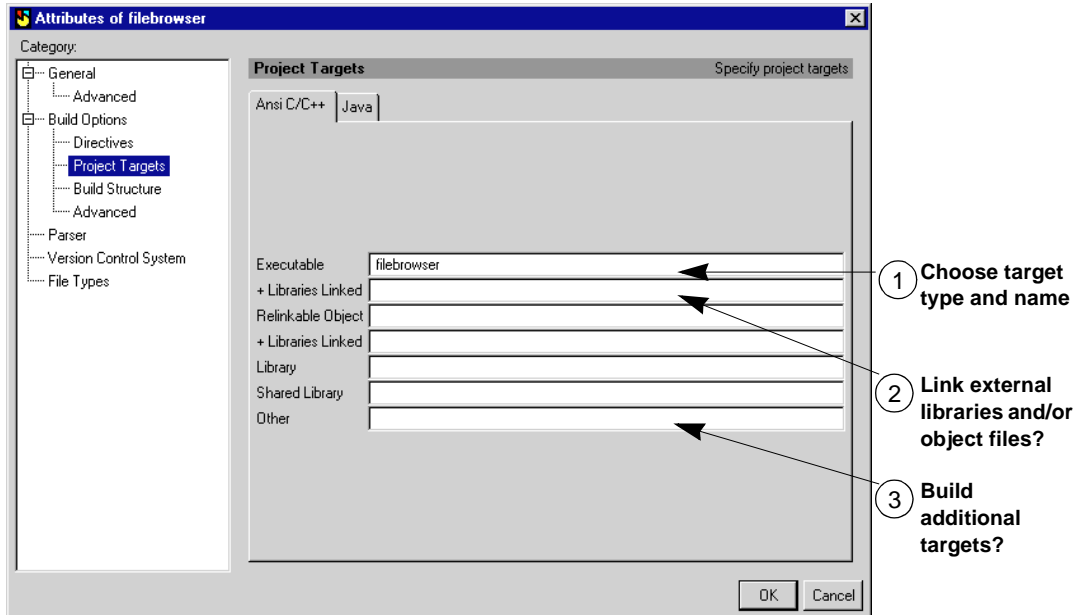
- Select the **Project Targets** node.

You will now set those Make attributes which are unique for each project. Basically, to set the attributes, you will do the following:

- Enter the name(s) of the target(s) to be built in the project.
- Enter any external libraries or object files needed for building the project's target.
- Generate directories for a recursive build (if used).
- Tell SNiFF+ whether targets built for the project are needed by the superproject.

In the rest of this section, you will set the attributes, one view at a time.

## In the Project Targets view



1. Enter the name of the project's default target.
  - If the target is an executable, enter its name in the **Executable** field.
  - If the target is an relinkable object, enter its name in the **Relinkable Object** field.
  - If the target is a library object, enter its name in the **Library** field.
2. If external libraries or object files are required by the target (executable or relinkable object), enter these in the **Libraries Linked** field. Use the exact command-line syntax as required by the linker (e.g. `-lm`) on your platform.
3. If you want to build additional targets, enter their names in the **Other** field. Use a colon (:) to separate target names.

Note that SNIFF+'s Make Support does not provide Make rules for building targets listed in the **Other** field. You can write the rules for building these targets in the General Makefile or in the project's Makefile. However, if the same target is to be built in more than one project, you should write the rule for building it in the General Makefile.

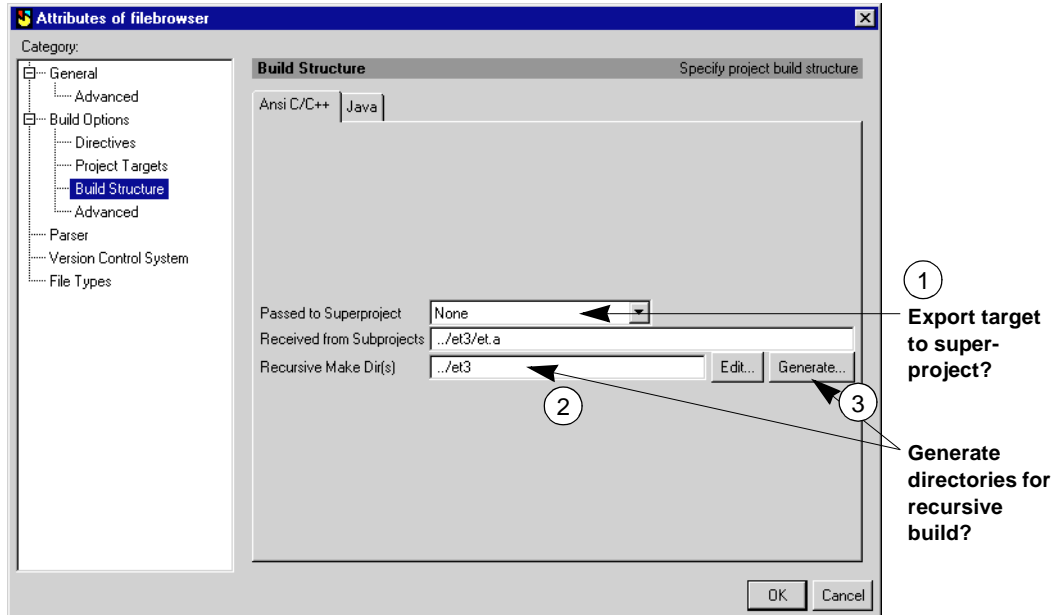
### For purify and quantify targets

You can enter purify and quantify targets in the **Other** field. For these two special targets, SNIFF+ **does** provide Make rules. Please refer to [Building purify and quantify targets \(Unix only\) — page 100](#) for details.

4. Select the **Build Structure** node.



## In the Build Structure view



1. From the **Passed to Superproject** drop-down menu, choose the type of target to be exported to the superproject. The object of this type is then displayed in the **Received from Subprojects** field in the superproject's Make attributes.

For details about using the **Passed to Superproject** drop-down menu, please refer to [Using the Passed to Superproject drop-down — page 98](#).

### Note

All targets imported to a project are displayed in the **Received from Subprojects** field.

2. To build your project's target recursively, make sure that the order of subproject directories listed in the **Recursive Make Dir(s)** field is correct.

SNiFF+ will build — in the order of listed subproject directories — the default target of each of the subprojects during a recursive Make.

3. If there are any discrepancies, press the **Generate...** button next to the field.

SNiFF+ will regenerate the order of subproject directories.

**IMPORTANT:** SNiFF+ considers only those subprojects that use SNiFF+ Make Support when generating the order of subprojects.

Also, once you have pressed the **Generate...** button for a project, the **Recursive Make Dir(s)** field will be automatically updated whenever you add/remove subprojects.

4. Press the **Ok** button to apply the Make attributes to the project.
5. Save the modified project.

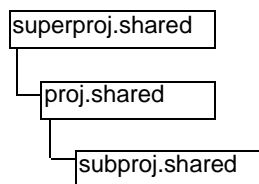
## Using the Passed to Superproject drop-down

There are five entries in the **Passed to Superproject** drop-down. Note that the selection in the drop-down also determines whether any, some, or all of the objects of the project's subprojects are exported to its superproject.

### Relinkable object

The relinkable object built in the project is exported to the superproject. Any targets imported to the project from its subprojects are used to build the relinkable object.

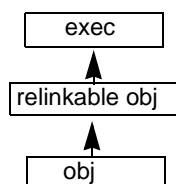
#### Project view



Relinkable object built in proj.shared and exported to superproject.shared

Target built in subproj.shared exported to proj.shared

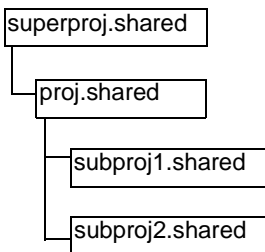
#### Target view



### Library

The library built in the project is exported to the superproject. Object files and relinkable objects imported to the project from its subprojects are used to build the library. Any libraries exported from subprojects are directly imported to the superproject.

#### Project view

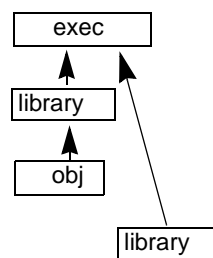


Library built in proj.shared exported to superproj.shared

Target (not a library) built in subproj1.shared exported to proj.shared

Library built in subproj2.shared and exported to superproj.shared

#### Target view

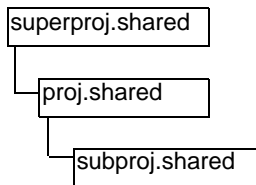


## Object files + Received

Any library, relinkable object and/or executable targets built in the project are not exported to the superproject.

The object files in the project (built from the source files in the project), plus the targets exported from subprojects, are directly exported to the superproject.

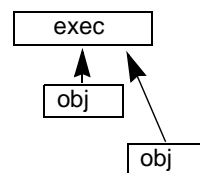
### Project view



Only object files built in proj.shared  
exported to superproj.shared

Target built in subproj.shared exported  
to superproj.shared

### Target view

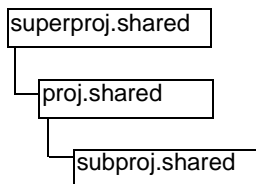


## Received targets

Targets imported to the project from its subprojects are used to build the project's target. Targets and object files built in the project are not exported to the superproject.

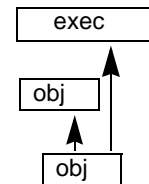
The targets imported to the project are also exported to the superproject.

### Project view



Target built in subproj.shared exported  
to proj.shared and to superproj.shared

### Target view

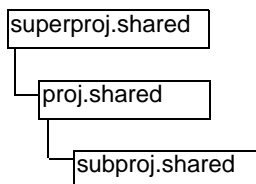


## None

Targets imported to the project from its subprojects are used to build the project's target. Targets and object files built in the project are not exportable.

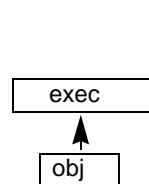
This setting is appropriate when the project's target is an executable.

### Project view



Target built in subproj.shared exported  
to proj.shared

### Target view



## Building purify and quantify targets (Unix only)

SNiFF+ provides rules in the General Makefile for preparing targets for further analysis with Purify™ and Quantify™ from Rational Software. You can specify targets to be "purified" and "quantified" in the **Other** field of the Project Targets view.

### To build a purify target

1. Enter the target name, followed by `.purify`, in the **Other** field of the Project Targets view. For example, to build a purify target for the executable `filebrowser`, you would enter `filebrowser.purify`.
2. Remove the hash (#) in the following line in the project's Makefile:  

```
#PURIFY_TARGET=$(LINK_TARGET).purify
```

This activates SNiFF+'s Make rules for purifying targets.
3. Update the project's Make Support Files.
4. Build the target. The project's Make command (entered in the **Make Command** field of the Build Options view) will be used for building the target.

### To build a quantify target

1. Enter the target name, followed by `.quantify`, in the **Other** field of the Project Targets view. For example, to build a quantify target for `filebrowser`, you would enter `filebrowser.quantify`.
2. Remove the hash (#) in the following line in the project's Makefile:  

```
#QUANTIFY_TARGET=$(LINK_TARGET).quantify
```

This activates SNiFF+'s Make rules for quantifying targets.
3. Update the project's Make Support Files.
4. Build the target. The project's Make command (entered in the **Make Command** field of the Build Options view) will be used for building the target.

Once you've built your purify and quantify targets in SNiFF+, you can load them into Purify™ and Quantify™.

## Specifying platform-specific Make information

SNiFF+ comes with a set of pre-configured Platform Makefiles. The correct Platform Makefile for your platform is automatically included by the General Makefile.

Your Platform Makefile contains macro definitions for the compilers, linkers, and archivers used by your Make utility during builds.

As an example, here are the C, C++ and Fortran macro definitions with their default values specified in the Platform Makefile for Solaris 2.x:

Language	Macro definition	Description
C/C++	CXX = gcc	C/C++ compiler specifications
	CXXFLAGS = -g	
	OVERALL_OPTION_CXX = -c	program for linking an executable
	LINK = \$(CXX)	
	LD = ld -r	programs for linking relocatable object files
	LDFLAGS =	
	AR = ar	yacc compiler
	ARFLAGS = rv	
	YACC = bison	lexical analysis generator
	YACCFLAGS = -yd	
Fortran	LEX = flex	Fortran compiler specifications
	LEXFLAGS =	
	FC = f77	
	FFLAGS =	
	OVERALL_OPTION_FC = -c	

## Procedures for specifying platform-specific information

1. Determine your Platform Makefile using the following table:

(All Platform Makefiles are located in your `$SNIFF_DIR/make_support` directory.)

Platform	Platform Makefile
AIX 3.2 or newer	rs6000-ibm-aix3.2.mk
AIX 4.2 or newer	rs6000-ibm-aix4.2.mk
DEC-Unix 3.2C	alpha-dec-osf3.0.mk
HP-UX 9.x	pa_risc-hp-hpux9.0.mk
HP-UX 10.x or newer	pa_risc-hp-hpux10.mk
Irix 5.3	mips-sgi-irix5.3.mk
Linux 2.x (SuSE 6.x or newer, RedHat 5.x, Debian 2.x)	i586-linux-glibc.mk
Linux 2.x (SuSE 5.3 or older, RedHat 4.x, Debian 1.x)	i586-linux-libc.mk
SCO 3.2	i386-unknown-sco3.2v4.2.mk
Sinix 5.42	mips-sni-sinix5.42.mk
Solaris 2.x	sparc-sun-solaris2.3.mk
SunOS 4.1.3	sparc-sun-solaris4.1.mk
Unixware 2.1	i386-unknown-sysv4.2MP.mk
Unixware 7.x	i386-unknown-unixware7.mk
Windows NT 4.0	i386-unknown-winnt4.mk
Windows 95/98	i386-unknown-win95.mk

2. Make a backup of your Platform Makefile.
3. Edit the Platform Makefile by changing the values of the macro definitions for your language (default values for C/C++/Fortran on Solaris 2.x given on [Specifying platform-specific Make information — page 101](#)).
4. Save the Platform Makefile.

The new values will be used the next time you build targets in SNIFF+.

## Language Makefiles — details

SNiFF+'s Language Makefiles are located in your `$SNIFF_DIR/make_support` directory. In it, you'll find the following Makefiles:

Language Makefile	Description
<code>general.link.mk</code>	Rules for executable, relinkable object and library targets
<code>general.c.mk</code>	Rules for C, C++, pro*C/C++ compilers
<code>general.fortran.mk</code>	Rules for Fortran compiler
<code>general.idl.mk</code>	Rules for IDL compiler
<code>general.java.mk</code>	Rules for Java compiler
<code>general.ada.mk</code>	Rules for Ada compiler
<code>general.yl.mk</code>	Rules for yacc and lex
<code>general.ilog.mk</code>	Rules for ILog Broker preprocessor

### Specifying Language Makefiles:

1. To specify Language Makefiles for all new projects, choose **Tools > Preferences....** To specify Language Makefiles for an existing project, open the project's Project Attributes dialog.
2. Select the **File Types** node.
3. Select the appropriate file type in the File Types List. If you need to first create one:
  - Select a file type in the File Types List whose attributes most closely match the new file type's attributes and press the **New...** button.
  - In the dialog that appears, give the new file type a name and press the **Ok** button.
  - Set the new file type's attributes.
4. Select the new file type.
5. Select the **Build System** tab.
6. In the **General Makefile** field, enter the full path name of the Language Makefile for the file type, for example:
 

```
$SNIFF_DIR/make_support/general.c.mk
```

 or  
 Press the **File...** button and in the dialog that appears, navigate to the Language Makefile and press **Open**.  
 When specifying multiple Makefiles, use spaces as delimiters.
7. Press the **Ok** button.





# Using Your Own Makefiles

---

## Introduction

Although we strongly encourage you to use SNIFF+'s Make Support and the makefiles that are part of it, you may choose to use and maintain your own makefiles for building your SNIFF+ projects.

### Note

You must use SNIFF+'s Make Support if you work in a team environment and use working environments.

## This chapter covers the following topics

- Using your own makefiles in SNIFF+
- Setting Make attributes for building and running targets in SNIFF+
- Which Make commands you can execute in SNIFF+ when using your own makefiles

## Assumptions made in this chapter

- You know how to work with SNIFF+ projects

## Related SNIFF+ topics

- Executing SNIFF+ commands for building, running and debugging targets — [Compiling and Debugging in SNIFF+ — page 185](#)

## Specifying Make attributes

You can set a project's Build attributes under the **Build Options** node of the Project Attributes dialog. The default values used in this dialog are specified in your Preferences. See also [Build System — page 147](#).

Basically, you need to do the following things to use your own makefiles in SNIFF+:

- Tell SNIFF+ what your Make command is
- Tell SNIFF+ the names of your targets

To use your own makefiles:

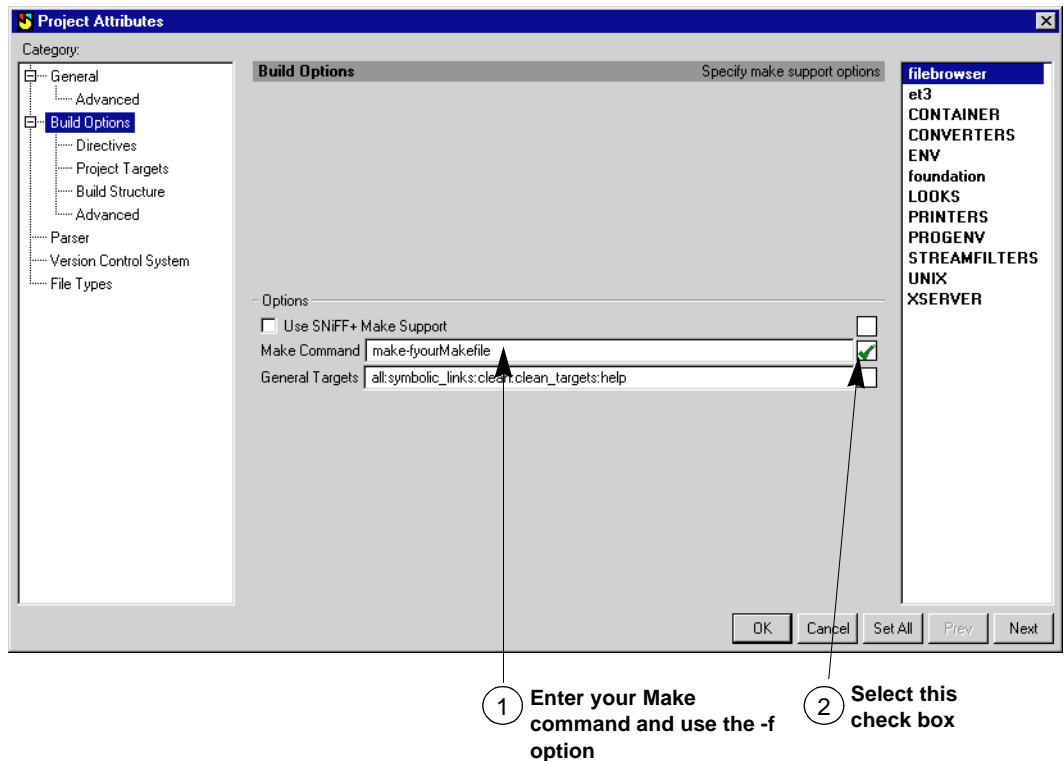
1. Start SNIFF+ and open the project for which you want to set up Make Support.
2. Check out the Project Description Files (PDFs) of all the projects for which you will be building targets.
3. In the Project Tree, checkmark all the projects for which you will be building targets.
4. Choose **Project > Attributes of Checkmarked Projects....**

The Group Project Attributes dialog appears.

5. Select the **Build Options** node.

## Project Attributes — Build Options

You will now set the Make attributes that are the same for all projects.



1. Enter your Make command in the **Make Command** field. Use the `-f` option to specify your makefile's name. For example:

```
make -f yourMakefile
```

This Make command will then be submitted to the Shell when you build targets in SNIFF+.

2. Select the check box to the right of the **Make Command** field.

This attribute will now also apply to all projects checkmarked in the Project Tree.

3. Press the **Set All** button to apply to all the projects checkmarked in the Project Tree.

4. Press the **Ok** button to apply the settings and to close the Group Project Attributes dialog.

Notice that the Project Tree indicates that all checkmarked projects are modified. We recommend that you save all modified projects at this time.

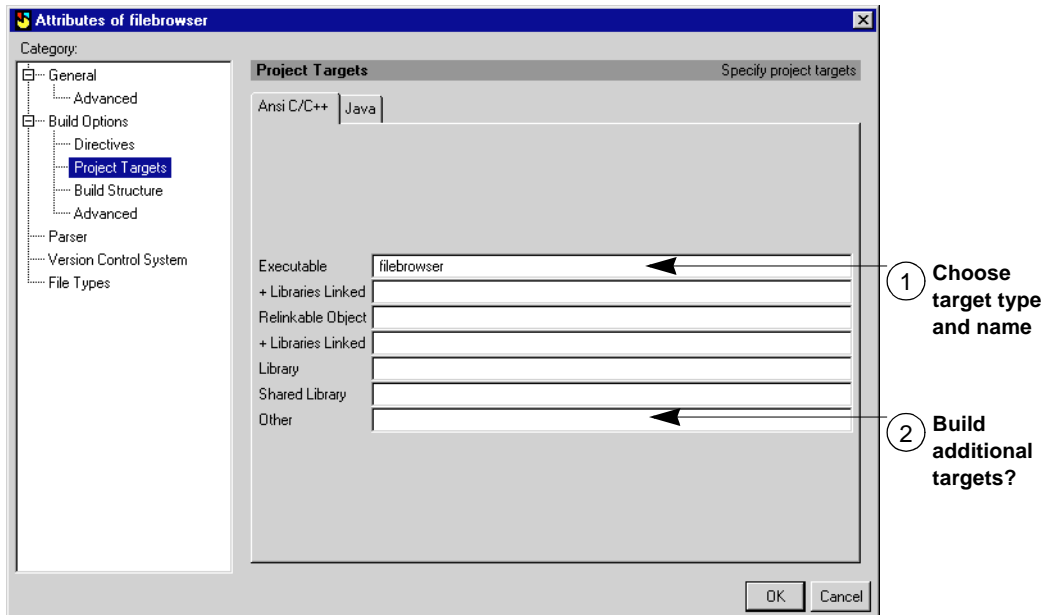
## Project-specific attributes in the Project Attributes dialog

You will now set those Make attributes which are specific to each project for which you will be building targets. Please complete the following steps **for each project**:

1. In the Project Tree, double-click on the project for which you will be setting project-specific Make attributes.

The Project Attributes dialog appears.

2. Select the **Project Targets** node.



3. Enter the name of the project's default target (described below).
  - If the target is an executable, enter its name in the **Executable** field.
  - If the target is an relinkable object, enter its name in the **Relinkable Object** field.
  - If the target is a library object, enter its name in the **Library** field.
4. If you want to build additional targets, enter their names in the **Other** field. Use a colon (:) to separate target names.
5. Press the **Ok** button to apply the Make attributes to the project.
6. Save the modified project.

## Specifying the targets of a project

The following types of targets may be specified in a project:

- executable
- relinkable object
- library

Each project has only one *default target*. SNIFF+ uses this target name as the default target for the **Make...** command in the **Target** menu. If only one of the targets mentioned above is specified in a project, this target is the default target of the project. If several targets are specified in a project, the first target in the order—executable, relinkable object, library—is the default target of the project. When no target is specified, object files are built.

## Make commands you can execute in SNIFF+

When using your own makefiles, you can execute the following Make commands from SNIFF+'s **Target** menu:

- **Recursively Make *default target***  
(The default target of the project is built using the recursive Make rules defined in your makefile.)
- **Make *default target***  
(The default target of the project is built using the Make rules defined in your makefile.)
- **Run *target***
- **Debug *target***



# Make Support changes from 3.0.x to 3.1

---

## Introduction

This chapter describes the differences between Make Support for SNiFF+ 3.1 and Make Support for SNiFF+ 3.0. With the latest SNiFF+ version 3.1, we provide a new Make Support concept, which has some fundamental changes in comparison to the former Make Support concept. Because of this, the new Make Support is faster and more user friendly.

### Note

Especially customers who upgrade from 3.0x (or earlier versions) to 3.1 should read this chapter carefully.

## This chapter covers the following topics

- [No support for VPATH](#)
- [Updating project Makefiles](#)
- [Reworked SNiFF+ Make-support files](#)
- [Use of pattern rules instead of suffix rules](#)
- [MAKE\\_TARGET macro](#)

## Assumptions made in this chapter

- You intend to use SNiFF+'s Makefiles and Make Support Files for regulating builds.

## Related SNiFF+ topics

- Setting up SNiFF+'s Make Support for your projects — [Build and Make Support — page 73](#).
- Using your own Makefiles in SNiFF+ — [Using Your Own Makefiles — page 105](#).

## Abbreviations and shortcuts used in this chapter

SSWE — Shared Source Working Environment

SOWE — Shared Object Working Environment

PWE — Private Working Environment

\$SNiFF\_DIR — path to your SNiFF+ installation directory

## No support for VPATH

In 3.0x VPATH was used to search for the source files in a given list of directories.

### Consequences

- no `gmake check_vpath` anymore
- no VPATH search during the Make run. Make looks for the dependencies of files in VPATH, checks the PWE and the SSWE. This is not necessary since source files are linked or copied on Windows.
- all VPATH related flags and macros e.g.,  
`VPATH`, `VINCLUDE`, `INHIBIT_LOCAL_INCLUDES`, etc.,  
are removed from the SNIFF+ Makefiles.

### Reasons why VPATH is not supported anymore

The VPATH macro is not used in the new Make Support because of its limitations listed below.

- not all compilers support the `INHIBIT_LOCAL_INCLUDES` flag
- inconsistencies because dependencies may be incorrectly resolved
- builds take longer

### Dummy rule for `check_vpath`

- Since all customers who use 3.0x or an earlier version have the default make command  
`gmake check_vpath;gmake`  
stored either in the Project Attributes or in the Platform definitions, each time Make is called the following error message will appear:  
`check_vpath: rule not found`
- We have added a "dummy `check_vpath` rule" to avoid that error. Now,  
`check_vpath not required anymore`  
is printed out (the message is printed out only for the project root directory).



## Updating project Makefiles

The SNIFF+ Make Support has been modified and is no longer compatible with the Project Makefiles of earlier versions so you will need to update the Makefiles. To update project Makefiles, do the following:

### Note

For upgrading **Java** projects, please refer to the technical reference in the Java tutorial.

### Note for Windows users

In the following section, there are several references to *symbolic links*. Windows does not, however, support symbolic links. So, wherever symbolic links are created by SNIFF+ on Unix, local copies are made on Windows. Therefore, if you are working on Windows, please read all references to "symbolic links" as "local copies" in the following.

### In the SiteMenus.sniff file

The SiteMenus.sniff file is in the SNIFF\_DIR/config/ directory.

- Open SiteMenus.sniff in an editor.

- Under

```
# Patch Makefiles for New MakeSupport
```

```
uncomment the following lines:
```

```
# >Makefiles
```

```
# shell "Update Makefile(s) for New Makesupport" "echo
Updating File %f; sh $SNIFF_DIR/make_support/
UpdateMakefile.sh %f"
```

or

### In the UserMenus.sniff file

The UserMenus.sniff file is in the %SNIFF\_DIR%\Profiles\*<Username>*\ directory on Windows, and in your \$HOME/.sniffrc/ directory on Unix.

- Open UserMenus.sniff in an editor.

- Copy the following lines from `SiteMenus.sniff` to `UserMenus.sniff`:
 

```
#Patch Makefiles for New MakeSupport
#>Makefiles
# shell "Update Makefile(s) for New Makesupport" "echo
Updating File %f; sh $SNIFF_DIR/make_support/
UpdateMakefile.sh %f"
```
- In `UserMenus.sniff`, under
 

```
# Patch Makefiles for New MakeSupport
```

 uncomment the following lines:
 

```
# >Makefiles
# shell "Update Makefile(s) for New Makesupport" "echo
Updating File %f; sh $SNIFF_DIR/make_support/
UpdateMakefile.sh %f"
```

## In the Project Editor

1. Load the project created with an earlier SNIFF+ version.
2. In the Filter dialog, **File Types** view, make sure that Makefiles are selected.
3. In the File List, select all Makefiles.
4. If the Makefiles are read-only, check them out by choosing **File > Check Out**.
5. Choose **Makefiles > Update Makefiles for New Makesupport**.
 

This command runs a script which removes the following lines from the selected Makefiles:

```
include $(SNIFF_MAKEDIR)/$(SNIFF_VPATH_INCL)
include $(SNIFF_MAKEDIR)/$(SNIFF_OFILES_INCL)
include $(SNIFF_MAKEDIR)/vpath.incl
include $(SNIFF_MAKEDIR)/ofiles.incl
```

Also replaces:

```
INCLUDE = $(SNIFF_INCLUDE) <other includes> with
SNIFF_INCLUDE += <other includes>
```

and inserts

```
SHARED LIB_TARGET
```
6. It is necessary to remove these lines since the `vpath.incl` and `ofiles.incl` files are no longer generated by the new Make Support, so trying to include them would result in an error.

## Reworked SNIFF+ Make-support files

1. `vpath.incl` is not needed anymore
2. `ofiles.incl` not needed anymore -> `OFILES` macro is now in `macros.incl`
3. Drop of unnecessary macros from `macros.incl`

### List of dropped macros

Dropped macros	Reason why dropped
<code>SNIFF_OFILES_INCL</code>	<code>ofiles.incl</code> not generated anymore
<code>SNIFF_VPATH_INCL</code>	<code>vpath.incl</code> not generated anymore
<code>SNIFF_ShSWS_2</code>	only needed for <code>check_vpath</code>
<code>SNIFF_OBJ_VPATH</code>	only needed for <code>check_vpath</code>
<code>SNIFF_PrOBJD</code>	only for Java
<code>SNIFF_ShOBJD1</code>	---"---
<code>SNIFF_ShOBJD2</code>	---"---
<code>SNIFF_ShOBJD3</code>	---"---
<code>SNIFF_ShOBJD4</code>	---"---
<code>SNIFF_ShOBJD5</code>	---"---
<code>SNIFF_ShOBJD6</code>	---"---
<code>SNIFF_VCS</code>	Repository rules are dropped
<code>SNIFF_REPOSITORY_DIR</code>	---"---
<code>SNIFF_FILES</code> + all corresponding types ( <code>SNIFF_Header_DIR</code> , <code>SNIFF_Header_FILES</code> , ...)	only needed for "link rules", but that is exactly what SNIFF+ does now. For your own <i>make</i> rules, <code>SNIFF_FILES</code> can be generated by setting the environment variable <code>SNIFF_FILES_NEEDED</code> to 1
<code>SNIFF_LIBS</code>	Backward compatible macro for SNIFF+ 2.1 or older
<code>SUB_RELINK_OFILES</code>	Backward compatible macro for SNIFF+ 2.1 or older
<code>IMPLEMENTATION_DIR</code>	
<code>DVPATH_DELIMITER</code>	Backward compatible macro for SNIFF+ 2.1 or older

### Why we dropped all macros for Java

All macros for Java are dropped, because the Make Support for Java and the "default Make Support" are split up. Now, within a Java project only "Java macros" are generated.

## Use of pattern rules instead of suffix rules

There is no need for Suffixes any more. This speeds up *make* since no implicit rules are called.

### Advantages of pattern rules

- More powerful and flexible than suffix rules
- Possible to add dependencies. For instance, ofiles depend on the Makefiles so ofiles are automatically generated when Makefiles are modified.

### Example of a Suffix rule

- `.c.o`: matches a file `test.o` to `test.c`

The disadvantage of suffix rules is that you can't specify any prefixes or letters at the end of a filename. You also can't specify directories.

### Example of Pattern rules

Use of suffixes:

- `%.o : %win.c` matches a file `test.o` to `testwin.c`

Matches object files in a subdirectory:

- `$(OBJECT_DIR)/%.o : %win.c` matches a object file `test.o` in a **subdir** to `testwin.c`

Redirection of object files to the **subdir** can be done without `VPATH`.

#### Note

Pattern rules and suffix rules cannot be used together to build inheritance chains! So if you have self-defined suffix rules, you will need to rewrite these suffix rules to pattern rules.

## MAKE\_TARGET macro

The script `sniffMakeTarget.sh` has been removed from the `$SNIFF_DIR/bin` directory. This script checked for a link to the SOWE and removed it. This is now done by the `MAKE_TARGET` macro.

You have only to add the following line before your compiler call within the *make* rule:

```
TARGET=$@; TARGET_TYPE="C++ object"; $(MAKE_TARGET); \
```

<b>TARGET</b>	is the current target to be built (usual this is \$@)
<b>TARGET_TYPE</b>	is the string that is echoed when the target is built
<b>MAKE_TARGET</b>	removes the target or its symbolic link before it is built



# Part V

## Maintaining SNIFF+ Projects





# Modifying SNIFF+ Projects

---

## Introduction

The main focus of this chapter is modifying projects. However, basic tasks such as opening, saving, closing and deleting projects are also described.

### This chapter covers the following topics

- How to open, save, close and delete projects
- General procedures for modifying projects
- General procedures for modifying multiple projects
- How to add and remove subprojects to and from existing projects
- How to add and remove files to and from existing projects

### SNIFF+ concepts you should already know

- SNIFF+ projects

### Related SNIFF+ topics not discussed in this chapter

- Version control and configuration management — [Version Control — page 135](#)
- Working environments
- Project Attributes dialog — [Reference Guide — Project Attributes — page 163](#)

### Abbreviations and shortcuts used in this chapter

- PWE — Private Working Environment
- PDF — Project Description File

## Opening Projects

If you are opening a shared team project for the first time in your working environment, please read [Initializing team working environments — page 53](#).

You can open shared projects and absolute (browsing-only) projects from the Launch Pad. You can also open shared projects from the Working Environments tool.

For shared projects:

- To open a shared project in your own Private Working Environment(s), use the Launch Pad's Working Environments tab.
- To open a shared project in the default working environment, use the Launch Pad's Working Environments tab or the **Project** menu.
- To open a shared project in any other working environment use the Working Environments tool.

### Note

In the Working Environments Tool (and the Launch Pad's Working Environments tab), you can choose from a list of available projects. In the Launch Pad, you have to know the exact location and the name of the project you want to open.

## Opening a shared project in a Private Working Environment

To open a shared project in your own Private Working Environment (PWE):

1. In the Launch Pad, select the **Working Environments** tab.

The Launch Pad displays the PWEs owned by you.

2. Double-click on the PWE in which you want to open the project.

The Projects dialog appears.

3. When you open the Projects dialog for the first time, the Project List is empty. Press the **Update List** button.

SNIFF+ scans the current working environment's directory structure and lists all the project description files (PDFs) in it.

4. Select the PDF of the project that you want to open.
5. If you do not want to parse the project when you open it, disable the **With Symbols** button.

You would disable the **With Symbols** button if:

- you only want to view or modify the project's structure and aren't interested in browsing its source files
- you only want to execute version control operations on the project's files

Loading a project with symbol information takes longer, since all the source files in the project will be parsed.

6. If you do not want to use cached project information when opening the project, clear the **Use Cache** check box.

You would clear the **Use Cache** check box if someone has made modifications to the project (including repository files) outside SNIFF+, and you want to get the latest project information or if your previous SNIFF+ was unexpectedly terminated.

Loading a project without the cache takes longer, since SNIFF+ will create the cache information while loading the project.

7. Press the **Open** button to open the selected project.

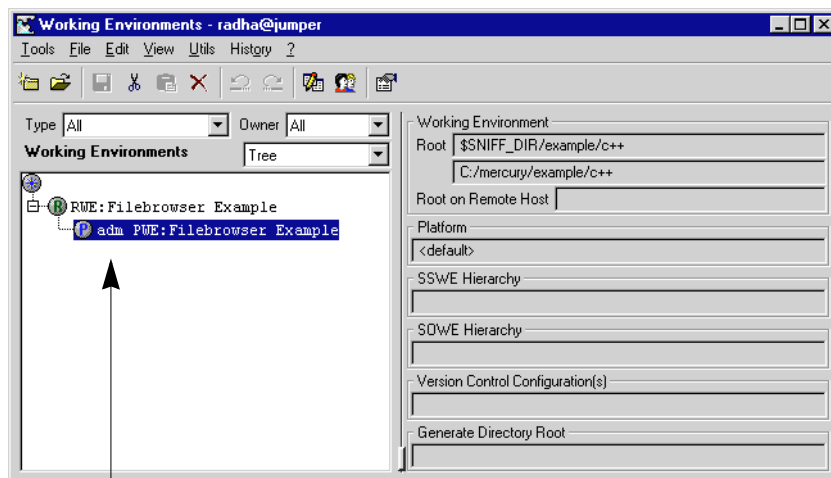
A new **Project Editor** appears, in which the loaded project is the root project in the Project Tree.

## Opening shared projects from the Working Environments tool

To open a project from the Working Environments tool:

1. Choose **Tools > Working Environments** in any open SNIFF+ tool.

The Working Environments tool appears.



Selected working environment  
Double-click a working environment  
to open projects in it.

2. Double-click on the working environment in which you want to open the project.  
The Projects dialog appears.
3. Open the Project. For details about the Projects dialog, please refer to [page 122](#).

## Opening projects from the Project Editor

When opening projects from the Project Editor, you can only open those that are displayed in the Project Tree of any open Project Editor on your screen.

To open projects from the Project Editor:

1. Make sure that the **Project Description** file type is visible in the Project List of the Project Editor. If it isn't, press **Filters...** and, in the **FileTypes** tab, select **Project Description** and press **Ok**.
2. In the File List, double-click the PDF of the project that you want to open.

A new Project Editor appears, in which the newly opened project is the root project in the Project Tree.

## Saving projects

You can save a project either in the Project Editor or in the Launch Pad. To save a project:

1. Select the Project.
2. Choose **Project > Save Project**.

## Saving projects with subprojects

If the project contains subprojects that have been modified, a dialog appears asking you whether you want to save the subprojects as well.

## Closing projects

You can close a project either in the Launch Pad or in the Project Editor. If the project has been modified, you will be asked if you'd like to save the changes.

To close a project:

1. Select the project.
2. Choose either:
  - In the Launch Pad  
**Project > Close Project <Project>**
  - or In the Project Editor  
**Project > Close Project.**

Upon closing a project, SNIFF+:

- saves the current symbol information of the project's source files to disk
- removes all project related information like the project structure, the attributes of a project and symbol information from memory
- saves SNIFF+'s current window state for future SNIFF+ sessions

## Deleting projects

To delete a project, you will first have to open it in SNIFF+. Then delete it in the Launch Pad. In the Launch Pad:

1. Select the project.
2. Choose **Project > Delete Project <Project>**.

A dialog appears asking you if you want to delete SNIFF+ related files, directories and the project description file of the project.

3. Select the **Repeat** check box to delete all subprojects at the same time.
4. Press the **Delete** button.

The following types of SNIFF+ files are deleted:

- project description files
- window state files
- symbol information files
- all other SNIFF+ generated files and directories

The following types of files are **not** deleted:

- all source files
- makefiles
- all other non-SNIFF+ files

## General procedures for modifying projects

You can modify projects in the Project Editor. Note that the project's PDF must be writable. To modify a project:

1. Check out the project's PDF by selecting it in the File List of the Project Editor and then choose **File > Check Out...**

The Check out dialog appears, in which you can select the file version to check out. See also [Checking out a version of a file — page 141](#).

2. If you want to modify the last version of the project's PDF, press the **Exclusive Lock** button. If you want to modify another version of the PDF, select that version and then press the **Exclusive Lock** button.

An alert dialog appears, in which you are asked whether the project structure should be reloaded or not.

3. Press the **Yes** button to reload the project structure.
4. In the Project Editor's Project Tree, double-click on the project you want to modify.

The Project Attributes dialog appears.

5. Modify the project according to your needs. As a result of modifying the project's attributes, you may have to add or remove files or subprojects to or from the project. In the next section, you will learn how to do so.

6. Save the project.

If you've modified any of the project's Make attributes, the project's Make Support Files will be regenerated.

7. If you've included new SNIFF+ file types in the project, you must manually add these files to the project. To do so complete the following steps in the Project Editor:

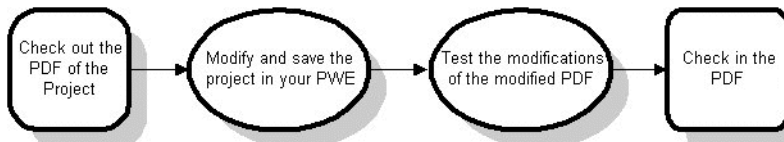
(i) make sure that the project is selected in the Project Tree

(ii) choose the **Add/Remove Files to/from <Project>...** command from the **Project** menu and then add the new files. For details about adding files, please refer to [Adding and removing files — page 129](#).

8. We suggest that you test the modifications to the project by building the project in your Private Working Environment before you check in the changes.
9. Once you have tested your changes, you can check in the project's PDF and any new files added to the project.

Your modifications will be seen by your team members after the next update of your team's working environments.

The above mentioned process is described in the following illustration:



## Modifying multiple project attributes

You can modify the attributes of multiple projects in the Project Editor. Note that the PDFs of the projects you want to modify must be writable.

To modify the attributes of multiple projects at the same time:

1. In the Project Tree, checkmark all the projects whose attributes you want to modify.
2. Check out the PDFs of the checkmarked projects.
3. Choose **Project > Attributes of Checkmarked Projects....**

The Group Project Attributes dialog appears. See also [Using the Group Project Attributes dialog — page 131](#)

## Project properties you can modify

You can do any of the following to modify a project:

- add and remove files to and from the project — for a detailed description, see [Adding and removing files — page 129](#)
- add and remove subprojects to and from the project — for a detailed description, see [Adding and removing subprojects — page 127](#)
- modify project attributes using the Project Attributes dialog

## Adding and removing subprojects

You can add and remove subprojects in the Project Editor. Note that the Project Description File (PDF) of the project must be writable (checked out). Please note that for shared projects, the projects must be in the same working environment.

### Adding a subproject to a project

1. Choose **Tools > Project Editor** in any open SNIFF+ tool.
2. In the Project Tree, select the project that you want to add a subproject to.
3. Choose **Project > Add Subproject to <Project>....**

This command is only enabled if the PDF is writable (checked out).

A Subproject File dialog appears. This dialog is similar to the Project File dialog.

4. Select the PDF of the subproject to be added.
5. Press the **Open** button.

The selected project in the Project Tree is now the superproject of the subproject you just added.

6. Save the project.

When you add a subproject to a project:

- the subproject is opened and its project structure and symbol information are loaded into memory

- a reference to the subproject is made in the project's project description file (PDF)

**Note**

You cannot add a subproject more than once to the same superproject.

## Removing a subproject from a project

1. In the Project Tree, select the subproject that you want to remove.
2. Choose **Project > Remove Subproject <Subproject>...**
3. Save the project.

When you remove a subproject from a project:

- the subproject is closed and the project structure and symbol information are removed from memory
- the reference to the subproject is removed from the project's PDF and the project becomes modified

**Note**

You cannot delete a project's PDF or any of the files of the subproject by issuing the **Remove Subproject <Subproject>...** command.



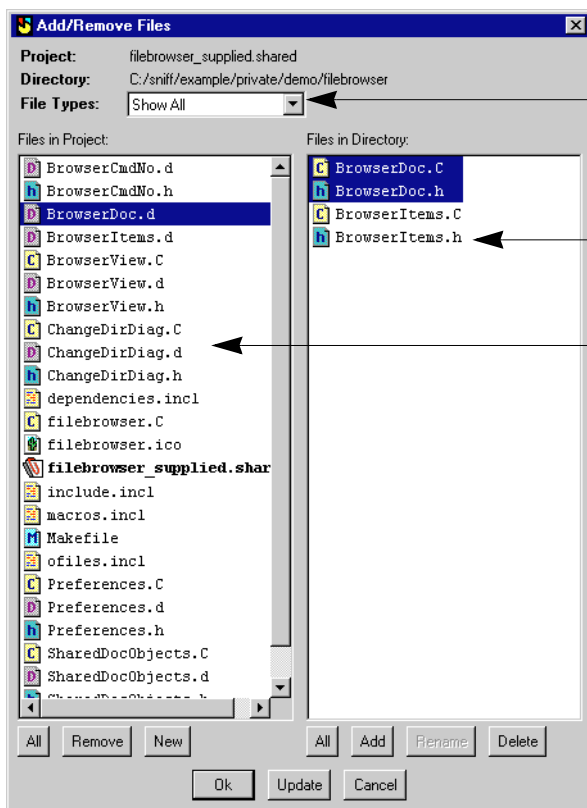
## Adding and removing files

**IMPORTANT:** If you have created new source files that are needed for building a project's targets, you **must** add them to the project. You must also add new source files to the project in order to browse their symbols.

You can add and remove files to and from projects in the Project Editor. Note that the PDF of the project that you add (remove) files to (from) must be writable.

1. Choose **Tools > Project Editor** in any open SNIFF+ tool.
2. To add or remove files, choose **Project > Add/Remove Files to/from <Project>....**

The Add/Remove Files dialog appears.



**File types drop down list:**  
allows the filtering of the lists  
to display only one file type

**File List:**  
multiple selections possible.  
A selection can be extended  
by pressing <SHIFT> and  
selecting entries. All entries  
of a list can be selected by  
pressing the All button

3. Select the file(s) to be added/removed.
4. To add selected file(s) to the project, press the **Add** button.

When you add a file to a project:

- the file is parsed and is added to the list of files in the project's PDF
- the file's symbol information is sent to the Symbol Table and can be browsed in SNIFF+

- the file is available for Make Support
  - the project has been modified
5. To remove the selected file(s) from the current project, press the **Remove** button.

When you remove a file from a project:

- the file is removed from the list of files in the project's PDF
- symbol information for this file is removed from memory
- the project has been modified

#### Note

The file is only removed from the SNIFF+ project, but not physically deleted.

6. Save the project.

## Adding Make Support Files to a project

To add Make Support files to a group of projects:

### In the Project Editor

1. Check out the PDFs of all the projects you want to modify.
2. In the Project Tree, checkmark these projects.
3. Choose **Project > Attributes of Checkmarked Projects....**

The Group Project Attributes dialog appears. To learn how to use this dialog, please refer to [Using the Group Project Attributes dialog — page 131](#).

### In the Group Project Attributes dialog

1. Select the **File Types** node.
2. Press the **Show All** button.

The File Types List now shows the complete list of file types defined in your Preferences.
3. Select **Make Support** in the File Types List and press the **Add File Type** button.
4. Under Merge Options, select the **Add** checkbox.
5. Press the **Set All** button.

The attributes you've modified now apply to all the projects checkmarked in the Project Tree. SNIFF+ now indicates that these projects have been modified.
6. Press **OK** to close the Group Project Attributes dialog.

### In the Project Editor

- Save all modified projects.

## Using the Group Project Attributes dialog

The Group Attributes dialog is an expanded Project Attributes dialog that lets you set the project attributes of multiple projects listed in the Project Editor's Project Tree. In this section, you will learn how to use the Group Attributes dialog.

1. Start SNIFF+ and open the root project of all the projects whose attributes you want to modify.

After the project is loaded, you should see the project and its subprojects in the Project Tree of the Project Editor.

2. Check out the Project Description Files (PDFs) of all the projects whose attributes you want to modify.
3. In the Project Tree, checkmark these projects.
4. Choose **Project > Attributes of Checkmarked Projects...**

The Group Project Attributes dialog appears. The projects checkmarked in the Project Editor are listed in hierarchical order in the dialog's Project List.

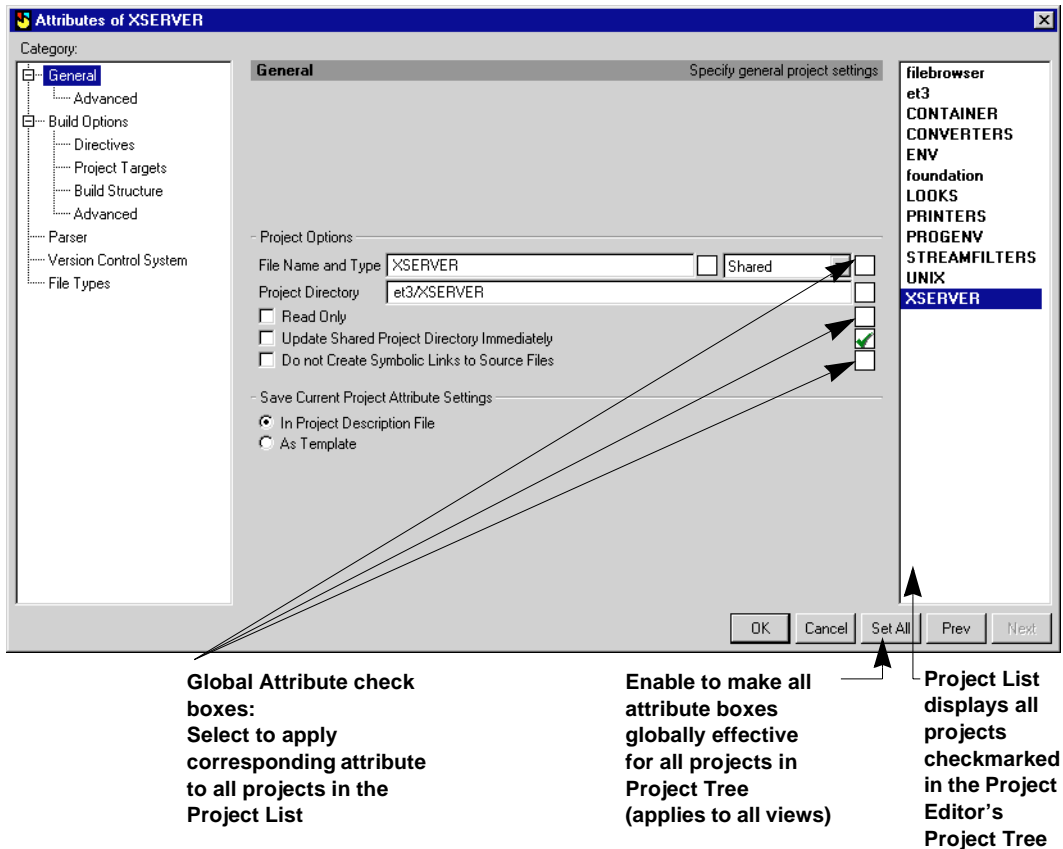
5. Notice that the topmost project in the Project List is selected. In the Group Attributes dialog, you modify the attributes of the project selected in the Project List and apply these attributes to the other projects.
6. If you want to select another project in the Project List, do so now.

### Setting attributes in the Group Project Attributes dialog

Like the Project Attributes dialog, the Group Project Attributes dialog groups project attributes into 5 main views:

- General
- Build Options
- Parser
- Version Control System
- File Types

Here's what the **General** view looks like:

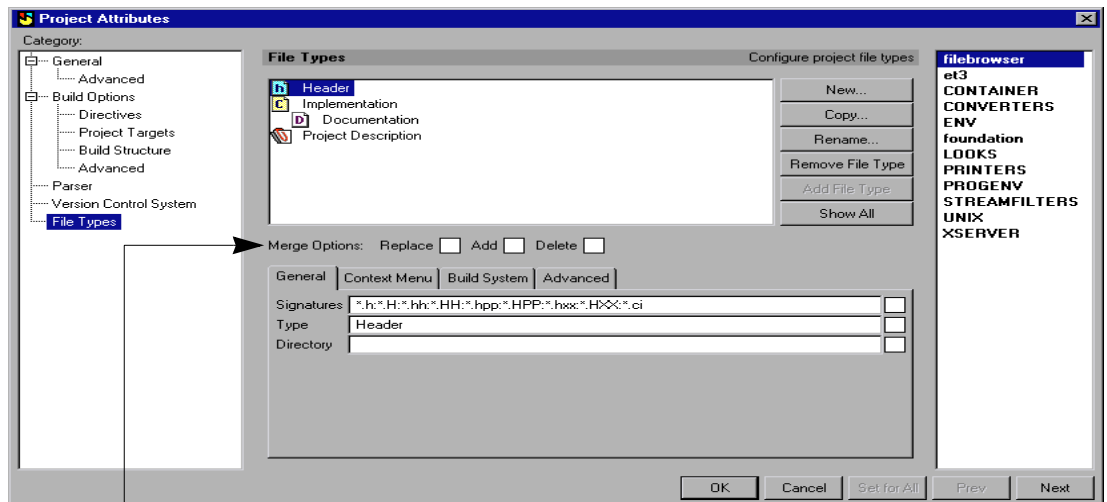


## To set all project attributes except for File Types attributes

1. Select the appropriate node.
2. Decide which attributes you want to set and apply to all the projects in the Project List.
3. Set these attributes. To apply the settings to all the projects, for each attribute you set, select the **Global Attribute** check box to its right.
4. Press the **Set All** button.  
The attributes now apply to all the projects checkmarked in the Project Tree. SNIFF+ now indicates that these projects have been modified.
5. To modify other project attributes, select the appropriate node and complete the same steps as above in that view.
6. Press **Ok** to close the Group Project Attributes dialog.
7. Save all modified projects.

## To set File Types attributes

1. Select the **File Types** node.



Merge options (Replace, Add, Delete) apply the attributes of the current file type to all projects in the Project List

2. If you want to add an existing file type to all projects in the Project List:
  - Press the **Show All** button.
  - In the File Types List, select the file type that you want to add. File types that aren't already part of the project appear in italics.
  - Press the **Add File Type** button.
  - Select the **Add** check box.
3. If you want to create a new file type and add it to all the projects in the Project List:
  - Select a file type in the File Types List whose attributes most closely match the new file type's attributes and press the **New...** button.
  - In the dialog that appears, give the new file type a name and press the **Ok** button.
  - Set the new file type's attributes.
  - Select the **Add** check box.
4. If you want to delete an existing file type from all the projects in the Project List:
  - Select the file type in the File Types List.
  - Select the **Delete** check box.
5. If you want to modify the attributes of the same file type in all the projects in the Project List:
  - Select the file type in the File Types List.

- Modify the file type's attributes.
  - Select the **Replace** check box to apply the file type's attribute to all the projects in the Project List.
6. Press the **Set All** button.

The attributes you've modified now apply to all the projects checkmarked in the Project Tree. SNIFF+ now indicates that these projects have been modified.
  7. Press **Ok** to close the Group Project Attributes dialog.
  8. Save all modified projects.

# Version Control

---

## Introduction

Support of configuration management and version control (CMVC) is an integral part of SNIFF+. In this chapter, you will learn how to use SNIFF+'s CMVC support for version controlling your projects.

### This chapter covers the following topics

- Execute version control commands in SNIFF+
- Look at a file's history and locking information
- Work with configurations
- Look at and merge differences between files

### Assumptions made in this chapter

- You know how to work with working environments and shared projects
- You understand the purpose of version controlling software
- You use a Repository and have the necessary write permissions for accessing it

### Abbreviations and shortcuts used in this chapter

SSWE — Shared Source Working Environment  
SOWE — Shared Object Working Environment  
PWE — Private Working Environment

## Technical overview

SNiFF+'s CMVC support provides the functionality available in the RCS version control system. If you use a tool other than RCS, please be aware that your tool may not support all of the functionality available in SNiFF+.

### Features

SNiFF+'s CMVC support comes with the following features:

- Checking out files from your Repository with either *exclusive lock*, *concurrent lock*, or *no lock*.
- Looking at a file's history and seeing which files in a project are locked by which people.
- Working with *configurations* - selected file versions grouped together under the same symbolic name.
- Working with *change sets* - a set of files checked in at the same time under the same symbolic name to the files.
- Working in *branches* of a file's version tree.
- Displaying two-way and three-way differences and merging versions, branches, change sets and configurations.
- Associating comments, dates and modifier information with versions, change sets, and configurations.
- Choosing a *Default Configuration* for a working environment.

### Your Repository

SNiFF+ only manages the structure of your Repository, but does not access the Repository's files directly — the access is delegated to your underlying version control tool.

### Differences between SCCS and RCS support

SNiFF+'s support for the creation and management of branches differs for SCCS and RCS. Please note the following differences in SNiFF+'s functionality when using SCCS for version control:

- With SCCS, you cannot associate a symbolic name to the latest version of a file on a branch.
- With RCS, you create branches when you check in a file version. With SCCS, you create branches during check-out.

### Working environments and version control

In team projects, each team member works in a Private Working Environment (PWE). All version control commands performed by team members are then naturally executed in the PWE. Your Working Environments Administrator, however, is also the owner of your team's Shared Source Working Environment (SSWE). While maintaining your team projects, he/she may also perform version control commands in the SSWE.



In the rest of this chapter, you will read phrases like:

- "the working environment you are currently in", or
- "in your working environment", or
- "in the working environment".

If you are a team member, these phrases refer to your PWE. If you are your team's Working Environments Administrator, these phrases refer to your team's SSWE.

## Locking files during check-out

SNiFF+ supports three mechanisms for locking files during a check-out:

- **Exclusive Lock**—Creates a local, writable copy of the file version in the working environment. Your version control tool puts an exclusive lock on the selected version. No other team member can check out the file while you have an exclusive lock on it. When you are done making modifications to the file and testing the changes, you check in the file, thereby removing your exclusive lock on it.
- **Concurrent Lock**—Creates a local, writable copy of the file version in the working environment. Your version control tool puts a lock on the version in such a way that others can also lock the same version. You and any number of your team members can check out and then modify a file with concurrent lock. After modification and testing, a merge of all the concurrently locked files must be performed, and the merged version must be locked and then checked in.

For systems like RCS that do not support concurrent locking, no locking mechanism is used.

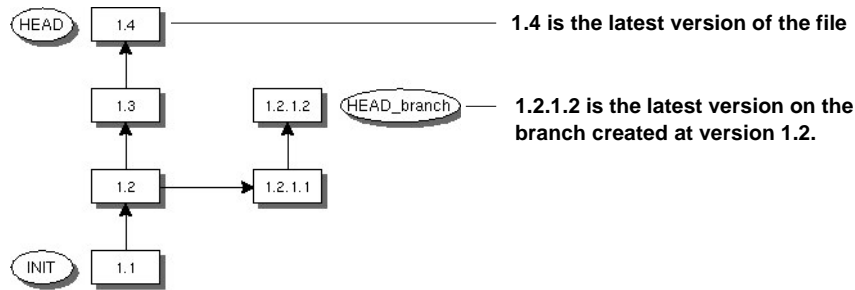
- **No Lock**—Creates a local, read-only copy of the file version in the working environment. Your version control tool does not put a lock on the version.

## Notation used when referring to file versions

SNiFF+ uses the following notation when referring to different versions of a file:

- **version tree** — A graphical presentation of a file's "evolution" from one version to another.
- **INIT** — The symbolic name given to the initial checked-in version of a version controlled file. The version number of the initial version is 1 . 1.
- **HEAD** — The symbolic name given to the latest checked-in version of a version controlled file. On a branch, the latest version is given the symbolic name HEAD\_<branch\_name>, where <branch\_name> is the symbolic name of the branch.

Here's an example of a file's version tree containing both `INIT` and `HEAD` versions.



## Configurations

At special times during the software development process, you might want to create a “virtual snapshot” of your software system. You do this in SNIFF+ by assigning a selected version of all the files in your software system the same symbolic name. In SNIFF+, the set of all file versions having the same symbolic name is referred to as a *configuration*, and the symbolic name assigned to the set is its *configuration name*. The process of creating a single configuration and associating it with a symbolic name is called *freezing a configuration*.

Note that `INIT` and `HEAD` are two special configurations used by SNIFF+.

- `INIT` is the configuration name of the initial checked-in version of **all** version controlled files in your Repository.
- `HEAD` is the configuration name of the latest checked-in version of **all** version controlled files in your Repository.

## Change sets

You can check in a set of files at the same time and assign this set a symbolic name. The set of files is referred to as a *change set*, and the symbolic name assigned to the set is its *change set name*.

## Branches

SNIFF+ supports the use of branches in a version controlled file's version tree. Branches occur in a version tree when you create new versions of a file from the middle instead of the end of the tree. Basically, SNIFF+ allows you to perform the same operations on branches that you can perform on the main trunk of a version tree. These include:

- Creating branches at any point in a file's version tree. This includes creating a branch at a file version already on a branch.
- Freezing the latest version of all files in a particular branch and assigning a configuration name to the set.

- Comparing branch configurations with other configurations (either on the main trunk or on a branch).

You can also merge a branch version of a file back into the main trunk of the file's version tree.

#### Note

In SNIFF+, you can create branches:

- during check-out when using SCCS
- during check-in when using all other version control systems

## Situations for using SNIFF+'s branch support

We strongly recommend that your Working Environments Administrator create and implement policies for working in branches. While branch support offers great opportunities, there is an inherent danger in allowing team members to create branches on their own.

You might choose to use SNIFF+'s branch support for one of the following situations:

- **Parallel development** — Stable versions of your software system are maintained in the main trunk of your files' version tree. Your team may also be working on alternative or experimental development approaches — which could be carried out on branches.
- **Temporary fixes and/or customization** — You may be asked by a customer to develop a site-specific version of your software system. You don't want this work to affect the main development work on the software system. You could create a temporary branch for the customization, and then reintegrate this branch with the main trunk at a later, opportune time.
- **Conflicting updates** — A member of your team might have an exclusive lock on a file that you also need to modify. You could do one of the following:
  - Break his/her exclusive lock. This really isn't a good choice.
  - Check out the same file version with a concurrent lock. This results in a local, writable copy of the file in your PWE. After making and testing your modifications, you can create a branch and check the file in on the branch. Your work can then be merged with the work of your team member at a later time.

## Default Configuration

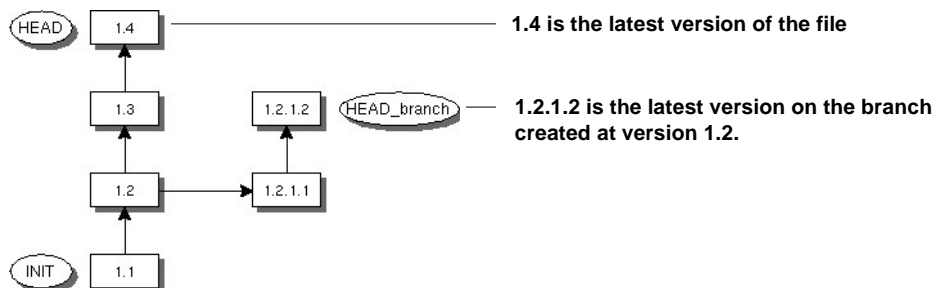
Each working environment specifies its own Default Configuration. When you open a SNIFF+ project in a working environment, SNIFF+ uses its Default Configuration:

- for setting the default value when you choose one of the various version control commands (e.g., check-out and check-in).
- when updating files in the working environment to the most current versions available in the Repository. A file is updated with respect to the Default Configuration in the file's version tree.

### Note

SNIFF+ automatically sets the Default Configuration for a newly created working environment to `HEAD`. You can change this default behavior either while creating a new working environment or afterwards.

To better understand the idea of a Default Configuration, let's suppose you work on a software system with files having the same version tree we looked at on [page 137](#):



You work on the main “trunk” of the tree and check out and check in files on the trunk. A member in your team works on the branch of the tree created at version 1.2. As a result, the Default Configuration for your PWE would be `HEAD`, and the Default Configuration for your team member's PWE would be `HEAD_branch`.

**IMPORTANT:** Actually, in this example, your team member should specify **two** Default Configurations for his/her PWE. The next section discusses why.

## Using multiple Default Configurations

Not all version controlled files in your Repository will have the same version tree. During normal development, files will often be created and checked into the Repository. Now, if the Default Configuration for your PWE is on a branch and you try to perform version control operations on a file that doesn't have this branch, your version control tool will complain.

To get around this problem, SNIFF+ allows you to specify multiple Default Configurations for a single working environment. If SNIFF+ can't perform a version control operation using the first specified Default Configuration, it will try to perform the operation using the next specified Default Configuration, and so on, until it is successful.

Going back to the example in the last section, your team member could specify two Default Configurations for his/her PWE. The first one would be HEAD\_branch, and the second could be HEAD.

### Note

To learn how to specify the Default Configuration(s) for a working environment, please refer to [Specifying Default Configurations — page 154](#).

## Executing version control commands in SNIFF+

The following version control operations are described in this section:

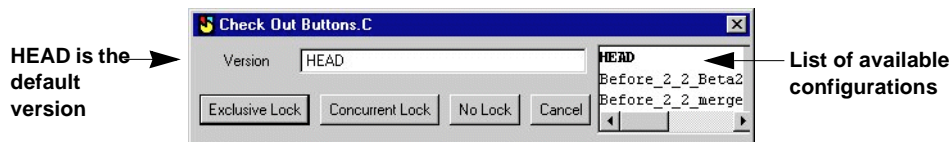
- [Checking out a version of a file](#)
- [Locking a file version](#)
- [Checking in files](#)
- [Unlocking a version of a file](#)
- [Deleting a version of a file](#)
- [Replacing the comments of a file version](#)

### Checking out a version of a file

You can check out a version of a file in the Project Editor, the Source Editor, Documentation Editor, the Configuration Manager, or the Diff/Merge tool. Here, we will use the Project Editor.

1. In the Project Editor, select the file in the File List and choose **File > Check Out...**

The Check Out dialog appears.



2. To check out the file, enter its version number in the **Version** field or, if the version is part of a configuration, select the configuration name from the list of available configurations.

### Note

You can check out more than one file at a time by selecting them (by holding down <SHIFT> and clicking on the file names with the left mouse button) and then choosing the **File > Check Out...** command in the Project Editor.

3. Choose a locking mechanism. See also [Locking files during check-out — page 137](#).

### SCCS Version Control

If you want to create a new branch and check in the new file version(s) on this branch, select the **New Branch** check box.

A field appears where you can enter the name of the new branch. Enter the name of the new branch. Note that SNIFF+ automatically prefixes the new branch's name with HEAD\_.

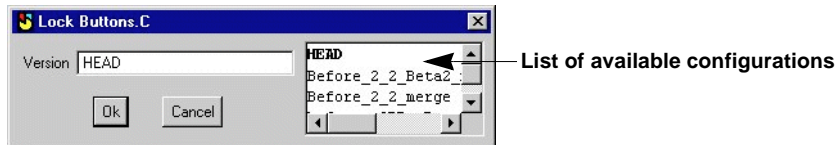
## Locking a file version

You can lock a version of a file in the Project Editor, the Source Editor, Documentation Editor, or the Diff/Merge tool. Note that you can only check in files that are locked by you.

To lock a file version:

1. In the Project Editor, select the file and then choose **File > Lock...** command.

The Lock dialog appears.



2. Enter the version you want to lock in the **Version** field or, if the version is part of a configuration, select the configuration name from the list of available configurations.
3. Press **Ok**.

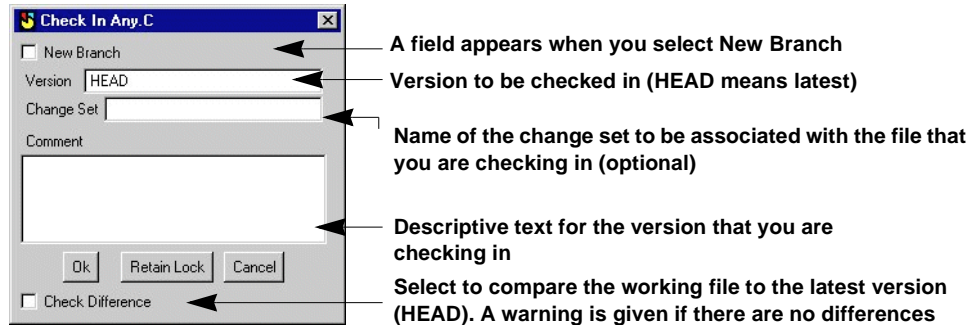
You now have a local, writable copy of the file version in your working environment. Your version control tools puts an exclusive lock on this version.

## Checking in files

You can check in a version of a file in the Project Editor, the Source Editor, Documentation Editor, or the Diff/Merge tool. Here, we will use the Project Editor.

1. In the Project Editor, select the file(s) in the File List and then choose **File > Check In....**

A Check In dialog appears.



2. If you want to create a new branch and check in the new file version(s) on this branch, select the **New Branch** check box.

A field appears in which you can enter the name of the new branch.

3. Enter the name of the new branch. Note that SNIFF+ automatically prefixes the new branch's name with HEAD\_.
4. Enter a version number or configuration name in the **Version** field. The new branch starts at this point in each checked-in file's version tree. The checked-in file will be the first version on the new branch.

For the SCCS version control system, you have to create branches in the Check Out dialog.

5. If you want to check in multiple files using a change set, enter the name of the change set in the **Change Set** field.
6. If you want to see whether the checked-in file is different from its latest version, select the **Check Difference** check box.
7. Check in the file(s):
  - To unlock the file(s) after the check-in, press **Ok**.
  - To retain the lock on the file(s) after the check-in, press **Retain Lock**.

## Unlocking a version of a file

You can unlock a version of a file in the Project Editor, the Source Editor, Documentation Editor, or the Diff/Merge tool. Here, we will use the Project Editor.

To unlock a file version:

1. In the Project Editor, select the file and then choose **File > Unlock....**
2. In the Unlock dialog that appears, enter the version you want to unlock in the **Version** field and press **Ok**.

Your version control tool removes your exclusive lock on the file version.

## Deleting a version of a file

You can delete a version of a file in the Project Editor or in the Configuration Manager. Here, we will use the Project Editor.

To delete a file version:

1. In the Project Editor, select the file and then choose **File > Delete Version....**
2. In the Delete Version dialog that appears, enter the version you want to delete in the **Version** field and press **Ok**.

The version you entered is deleted from the file's version tree. Your version control tool readjusts version numbers in the version tree accordingly.

## Replacing the comments of a file version

You can replace the comments of version of a file in the Project Editor or in the Configuration Manager. To do so:

1. Select the file version in either the Project's Editor History View or the Configuration Manager's File List.
2. Choose **File > Replace Comment....**
3. In the Replace Comment dialog that appears, change the file version's comments and then press the **Replace** button.

## Looking at file version history

To look at the history of a file:

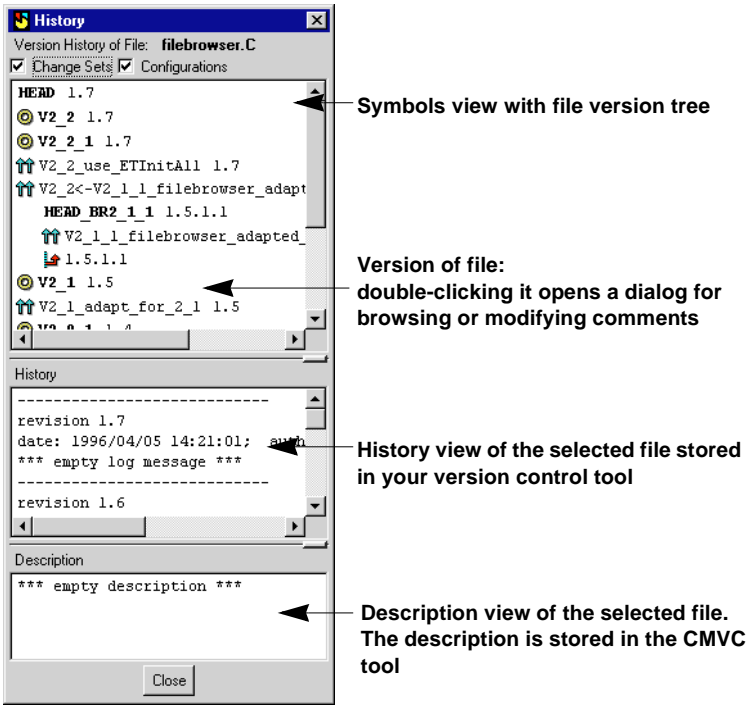
1. Use the Project Editor's filters and Project Tree to select the files whose history information you want to view.
2. Select the **History** check box in the Project Editor.

Three new views are added to the Project Editor:

- **Symbols View** — Displays all versions of the file as stored and maintained by your version control tool in the file's version tree. All file versions are displayed in the Version Tree of the Symbols View, along with detailed information about each version.
- **History View** — Displays the history of the file version selected in the Symbols View.
- **Description View** — Description about the file selected in the File List. To enter or



modify a file's description, the file must be writable.



**Symbols View**

The symbols and text in the Version Tree of the Symbols view give you details about the versions of the selected file.

Five different symbols are possible in the Version Tree of the Symbols view:

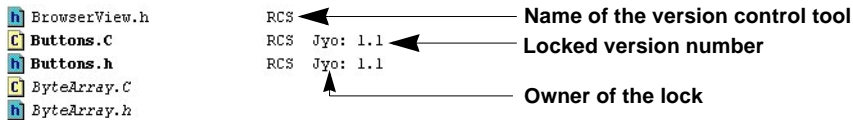
Symbol	This symbol means that the entry is...
↑	a single unnamed version
=	a new file version without a change
↑	a new branch of a version
↑↑	a change set
⦿	a configuration name

## Displaying locking information

To load and display locking information:

1. Use the Project Editor's filters and Project Tree to select the files whose locking information you want to view.
2. Select the **Lockers** check box in the Project Editor.

The File List is expanded to show file locking information.



## Filtering the File List according to locking state

You can filter the File List according to a file's locking state. To do so, choose one of the following entries from the **File Status** pop-up menu:

- **All Files** — Displays all files regardless of their locking state.
- **Modified** — A Files Compared to dialog appears. Files different from the version you enter in this dialog are displayed.
- **Unchanged** — A Files Compared to dialog appears. Only those files that have not changed from the version you enter in this dialog are displayed.
- **Own** — Displays only those files locked by you.
- **Own modified** — A Files Compared to dialog appears. Files locked by you that are different from the version you enter in this dialog are displayed.
- **Locked** — Displays only locked files.
- **Not Locked** — Displays only unlocked files in the File List.
- **Filtered...** — Multiple entries are selected in the Filters dialog. Selecting the **Filtered...** entry itself opens the Filters dialog.

## Creating your own CMVC adaptor

SNIFF+'s version control and configuration management (CMVC) functionality is provided via a consistent user interface that sits on top of an abstract CMVC interface. In this sense, SNIFF+ does not implement any version storage functionality itself; it delegates all actions via this CMVC interface to a CMVC tool like RCS or CVS, which is responsible for the actual repository management. The interface consists of about 40 commands that can be easily mapped to any specific CMVC.

CMVC adaptors should only be created by experienced users of the version tool and Unix. However, it is a one-time task since SNIFF+ only needs to be adapted once to a new CMVC. You can use any scripting language e.g., Bourne Shell, Python, TCL, etc. to create your own adaptor. For information on how to do so, please get in touch with your TakeFive Sales contact.

## Working with configurations

You use SNIFF+'s Configuration Manager to work with configurations of your software system. This section covers

- [Looking at configurations](#)
- [Comparing two configurations](#)
- [Checking out a configuration](#)
- [Freezing configurations](#)
- [Renaming configurations](#)
- [Deleting configurations](#)

For details about the Configuration Manager's user interface, please refer to the *Reference Guide*.

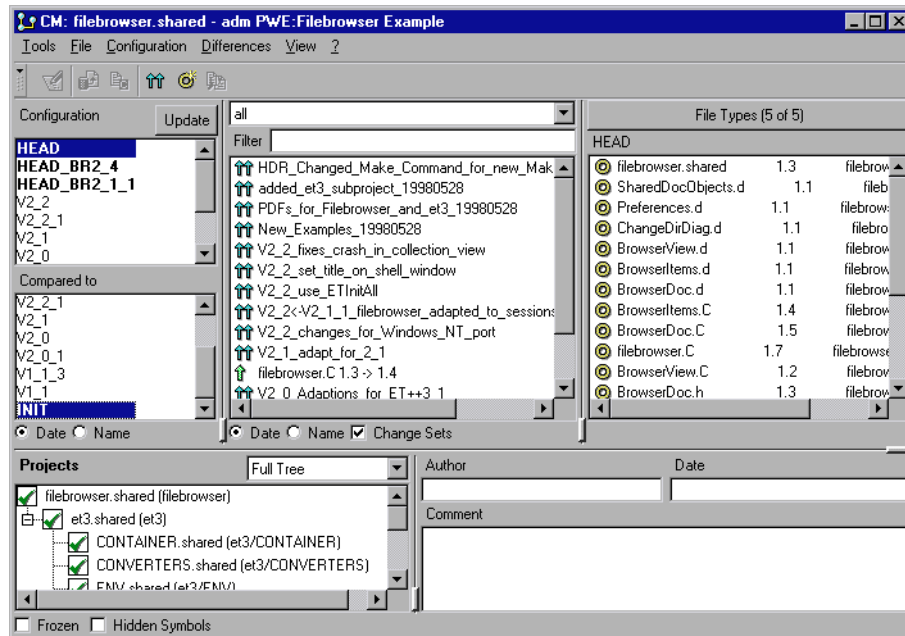
### Looking at configurations

1. Open the SNIFF+ project whose configuration information you want to view.
2. Choose **Tools > Configuration Manager**.  
The Configuration Manager appears. Complete the remaining steps in this section in this tool.
3. In the Configuration Manager's Project Tree, checkmark the projects that you want to view.
4. Press the **Update** button to load the configuration information for the checkmarked projects.

When the update process is over, you should see a list of configurations for the checkmarked projects in the Configuration List.

5. To look at the details of a particular configuration, select it in the Configuration List.

The Configuration Manager now displays all the files that are part of the selected configuration.



## Comparing two configurations

To see what differences exist between two configurations of your software system:

1. Select one of the two configurations in the Configuration List.
2. Select the other configuration in the Compared To List.

SNiFF+ displays the differences between the two configurations in the Change List. Icons in the Change List indicate the nature of the difference. Online, use the Configuration Managers **Help(?) > Quick** menu for a description of the various icons, or see the *Reference Guide*.

3. You can look more closely at any change sets in the Change List. To do so, select the change set. SNiFF+ displays the files in the selected change set in the File List.

## Checking out a configuration

You can check out a complete configuration of your software system in your working environment. To so do:

1. Select the configuration you want to check out in the Configuration List.
2. Choose **Configuration > Check Out <Configuration>...**

A Check Out Configuration dialog appears, in which you can specify how you want your version control tool to lock the configuration.

3. Choose a locking mechanism (see page [Locking files during check-out — page 137](#) for a description).

## Freezing configurations

You can freeze either the latest version of selected files in your software system or the Default Configuration for the working environment you are currently in.

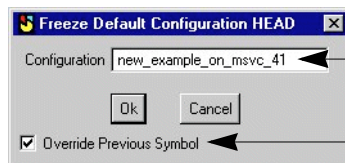
### Freezing the latest version of your software system

When you freeze the latest version of your software system, you create a new configuration that consists of the latest (HEAD) version of all your version controlled files.

To freeze the latest version:

1. If you have a single root SNIFF+ project for your software system, open this project.
2. Open the Configuration Manager.
3. Select **HEAD** in the Configuration list.
4. Choose **Configuration > Freeze Head...**

A Freeze Configuration dialog appears.



← Name of new frozen configuration

← Force a freeze even if the configuration name is already used

5. Enter the name of the new configuration in the **Configuration** field and press **Ok**. Note that if you enter an existing configuration name, the original configuration will be deleted.

The new frozen configuration will appear in the Configuration List as the newest configuration below the HEAD configuration.

## Freezing a Default Configuration

You can freeze the Default Configuration in the working environment you are currently in. When you freeze the Default Configuration, you create a new configuration that consists of all the files that are part of the Default Configuration.

To freeze the Default Configuration:

1. If you have a single root SNIFF+ project for your software system, open this project.
2. Open the Configuration Manager.
3. Choose **Configuration > Freeze Default Configuration....**  
A Freeze Default Configuration dialog appears.
4. Enter the name of the new configuration in the **Configuration** field and press **Ok**. Note that if you enter an existing configuration name, the original configuration will be deleted.

The new frozen configuration appears in the Configuration List.

## Renaming configurations

To rename a configuration:

1. Select the configuration in the Configuration List.
2. Choose **Configuration > Rename Configuration Name....**  
The Rename Configuration dialog appears.
3. Choose the new name for the configuration and press **Ok**.  
SNIFF+ renames the configuration in all the projects checkmarked in the Configuration Manager's Project Tree.

## Deleting configurations

To delete a configuration:

1. Select the configuration in the Configuration List.
2. Choose **Configuration > Delete Configuration Name....**  
The Delete Configuration dialog appears and asks you to confirm the deletion.
3. Press **Ok**.  
SNIFF+ requests your version control tool to delete the configuration name from your Repository. Note that your version control tool will not delete any files that are associated with the deleted configuration.

## Looking at and merging differences between two file versions

In this section, you will learn how you can use the Diff/Merge tool to show and merge differences between files and versions of files.

You can open a Diff/Merge tool in the Project Editor, the Source Editor, or the Configuration Manager. For a detailed description of the Diff/Merge tool, please refer to Reference Guide —Diff/Merge tool.

### Comparing file versions

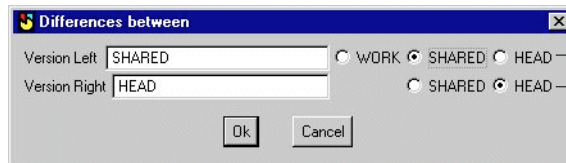
You can look at the differences between two versions of a file in the Project Editor, the Source Editor, Documentation Editor, the Diff/Merge tool, or the Configuration Manager.

Here, we will use the Project Editor.

To look at the differences between two versions of a file:

1. Choose **File > Show Differences....**

The Differences dialog appears



—These two radio buttons show either the default configuration or the file version selected in the Project Editor's History View

2. In the **Version Left** field, enter one of the file versions to compare. Use the radio buttons if applicable (see below).
3. In the **Version Right** field, enter the other file version to compare. Use the radio buttons if applicable (see below).
4. Press **Ok**.

SNiFF+ opens a Diff/Merge tool, in which the differences between both file versions are displayed. The Diff/Merge tool is shown on [page 152](#).

### Radio buttons

**WORK**—Refers to the version of the file you "see" in the working environment. This can either be a writable, local copy or the read-only original located in the shared working environment (usually the SSWE).

**SHARED**—Refers to the version of the file currently located in the shared working environment accessed by the working environment you are currently in. Usually, you are in your PWE, so the accessed shared working environment is your team's SSWE.

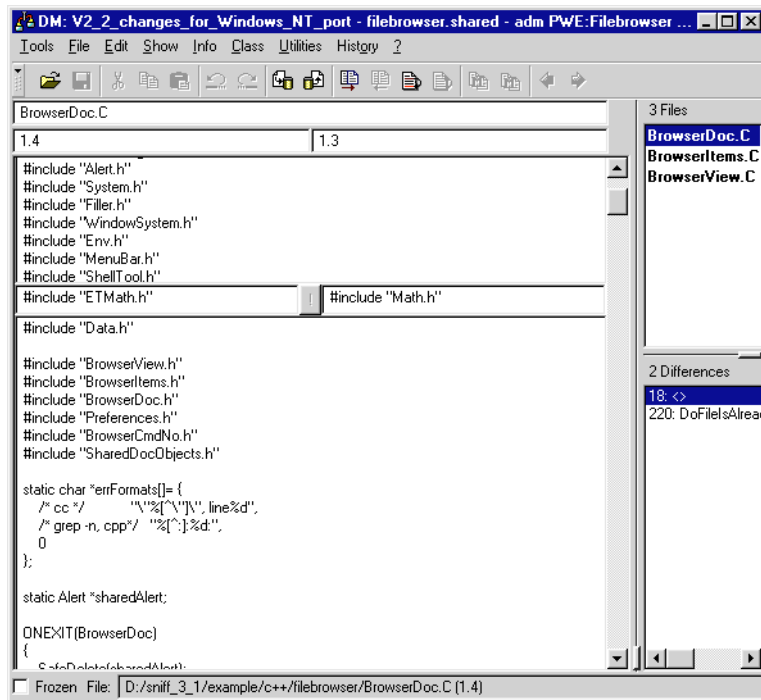
**Default Configuration or File Version** (here HEAD)—If a file version is selected in the Project's Editor History View, this radio button displays the version number. Otherwise, it displays the Default Configuration of the working environment you are currently in. Default Configurations are discussed on [Default Configuration — page 140](#).

#### Note

If you select the **SHARED** radio button for the **Version Right** field, you must select **WORK** for the **Version Left** field.

## Comparing file versions in the Diff/Merge tool

A Diff/Merge tool appears when you compare two file versions using the **Show Differences...** command from the **File** menu.



You can also use the Diff/Merge tool for editing files. In the figure above, the working file (file version WORK) is writable, so you can edit it and merge differences into it.



## Showing the differences between change sets

You can use the Diff/Merge tool to view differences between versions of files in a change set and previous versions of the same files. To do so:

1. Select a change set in the Configuration Manager and choose **Differences > Show Differences....**

A Show Differences dialog appears, in which you can select the type of differences (two-way or three-way) that you want to see.

2. Press the **2-Way** button. (For three-way differences please refer to [Showing and merging three way differences — page 153.](#))

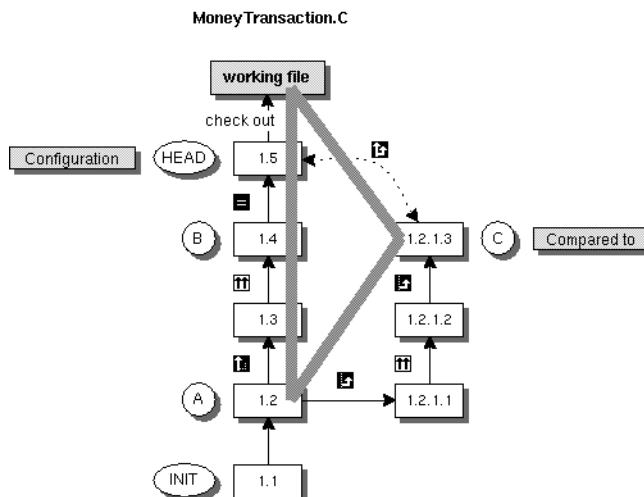
A Diff/Merge tool appears. Differences between files in the change set and previous versions are shown in the tool.

3. The File List displays the files in the change set. By clicking on a file in the list, you can look at the changes between it and its previous version.

## Showing and merging three way differences

You can also use the Diff/Merge tool to look at three-way differences between files and file versions. One situation where this might come in handy is when you want to merge branch file versions and configurations back into the "main trunk" of a file's version tree.

As an example of viewing/merging three-way differences between versions that are on two different branches, let us assume that a source file called `MoneyTransaction.C` has the following version tree:



The latest version on the main trunk is called HEAD. The latest version on the branch is called HEAD\_C. Version 1.2 is the common ancestor of both versions.

Now, you can view the three-way differences between the latest version on the main trunk (1.5), the latest version on the branch (1.2.1.3) and the ancestor (1.2). To do so:

1. Open the project that contains the configurations you are interested in.
2. Open the Configuration Manager and press the **Update** button to load your project's configuration information into the Configuration Manager.
3. Select **HEAD** in the Configuration List and **HEAD\_C** in the Compared To List.  
The Change List displays all the changes made on the main trunk and the branch since the ancestor version.
4. From the **Change type** drop-down menu above the Change List, choose **different branches**.  
The Change List displays only those version controlled files that have a branch at version 1 . 2 in their version trees.
5. From the Change List, select the file whose three-way differences you are interested in.
6. If you just want to look at the three-way differences, choose **Configuration > Show 3-Way Differences**.  
A Diff/Merge tool appears, in which the three-way differences are displayed.
7. If you want to merge differences, choose **Differences > Merge Differences...** command and press the **3-Way Merge** button.  
A dialog appears and asks you whether you want to check out the file selected in the Change List.
8. If you have already checked out the file, press **No**. Otherwise, press **Yes**.  
A Diff/Merge tool appears, in which the three-way differences are displayed.

## Specifying Default Configurations

### Note

We strongly recommend that your Working Environments Administrator specify the Default Configurations for your team's working environments.

Default Configurations may be specified for SSWEs and PWEs, either during their creation or afterwards. Note that an SOWE inherits the Default Configurations of the SSWE that it accesses.

To specify Default Configurations for a working environment:

1. Make sure you have the appropriate write permissions for modifying working environments.
2. Start SNIFF+ and open the Working Environments tool.

## In the Working Environments tool

1. In the Working Environments tool, select the working environment for which you want to specify Default Configurations.
2. Choose **Edit > Modify...**  
The Working Environment - Modify dialog appears.
3. In the **Version Control Configuration(s)** field, specify the Default Configurations for the working environment:
  - To specify HEAD as the default, either leave the field blank or enter HEAD.
  - To specify a non-HEAD configuration as the default, enter the configuration name.
  - To specify the HEAD version of a branch as the default, enter HEAD\_<branch\_name>.
  - Use a colon (:) to separate multiple Default Configurations. For example, to specify both HEAD and HEAD\_branch01 as Default Configurations, you would enter:  
HEAD\_branch01:HEAD
4. When you are done, press **Ok** to return to the main window of the Working Environments tool.
5. Save the modifications to your working environments.



# Updating Working Environments

---

## Introduction

SNiFF+ provides a powerful updating mechanism for your working environments that ensures that all members in your development team are working on the latest, most stable version of your software system.

### This chapter covers the following topics

- Why working environments need to be updated
- What happens to projects and files in working environments during an update
- How to update working environments in SNiFF+
- How to update working environments from the Shell
- How to run unattended updates from the Shell

### Assumptions made in this chapter

- Your team's Working Environments Administrator has already set up Make Support for your projects

### Related SNiFF+ topics

- Configuration management and version control in SNiFF+ — [Version Control — page 135](#)
- `sniffaccess` (used to drive unattended working environment updates, see [Reference Guide — Sniffaccess — page 259.](#))

### Abbreviations and shortcuts used in this chapter

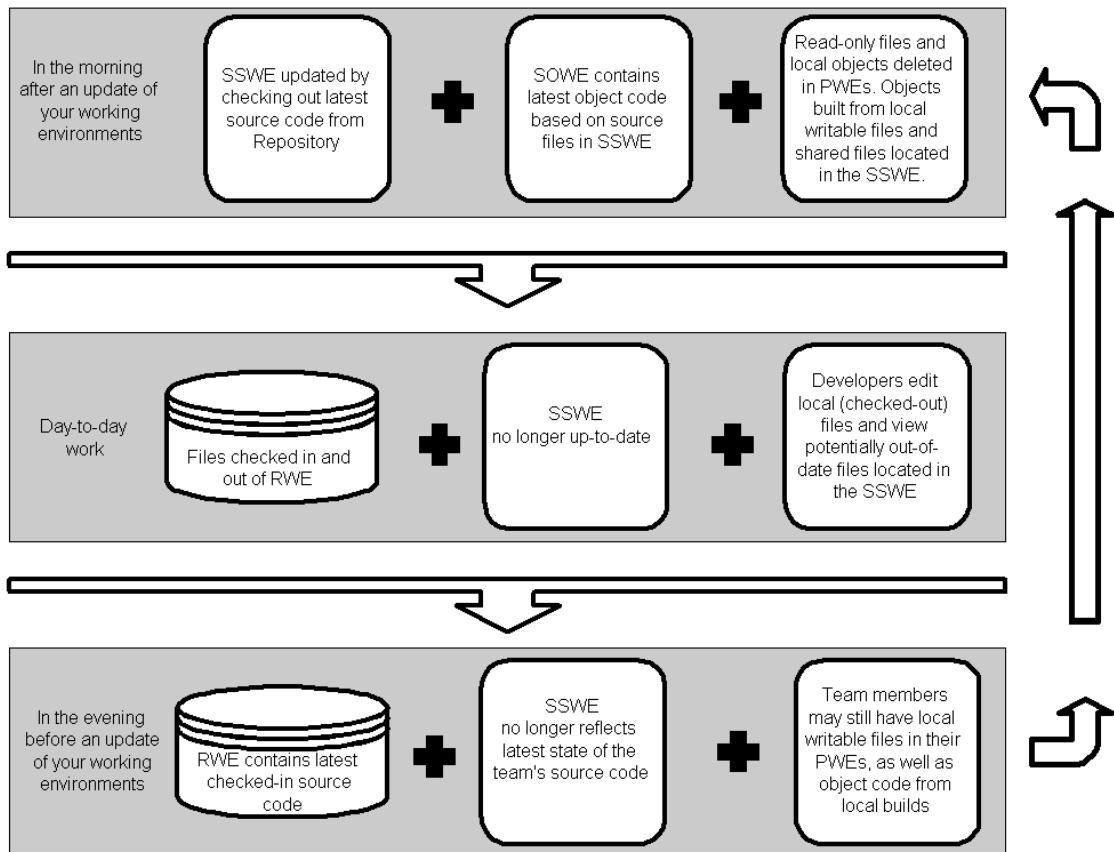
RWE — Repository Working Environment  
SSWE — Shared Source Working Environment  
SOWE — Shared Object Working Environment  
PWE — Private Working Environment  
\$SNiFF\_DIR — SNiFF+ installation directory

## Technical overview

### Why working environments need to be updated

During a day's work, you and your team members check out files and go through the edit/compile/debug cycle in your PWEs. Some of the modifications that you make to checked-out files and other files are checked in during the day. Consequently, the latest versions of your source files will be spread over several working environments, and your team won't be working with a consistent set of up-to-date files the next day unless something is done about it. This is where SNIFF+'s updating mechanism for working environments comes into the picture.

SNIFF+ provides a mechanism for updating both shared working environments (SSWEs and SOWEs) and PWEs. The following diagram summarizes how working environments slowly "lose sync" with each other and how updating them corrects this situation:



As the diagram suggests, updates and builds of shared working environments should be done at times when no developers are working. This usually means either overnight or over the weekend. Your team members can update their PWEs at any time. However, the best time to do so is immediately after the shared working environments you access have been updated.

## Unattended updates of working environments

Many development organizations use scripts for unattended software updates and builds each night. Such *nightly updates and builds* verify that your team's object code is still buildable and consistent the next time team members open projects in their PWEs.

SNiFF+ comes with an administrative script that help you manage and integrate the work that your team members perform in their respective PWEs. This scripts, as well as both manual and unattended updates, are discussed later on in this chapter.

## The Working Environments Administrator

Working environment updates do not always work according to plan. A number of things can go wrong, such as:

- builds during updates don't deliver the desired results because team members have checked in buggy source files
- updates terminate suddenly because of hardware problems
- unattended updates don't take place because of a bug in the update script

To handle these and any other update-related problems that may occur, we strongly recommend that you appoint a *Working Environments Administrator*. In addition to making sure that updates function properly, he/she could also be responsible for:

- creating new working environments
- creating and enforcing guidelines regarding how shared files should be checked out and checked in
- keeping shared working environments clean of obsolete files and directories
- maintaining your team's Repository

## General guidelines for updating SSWEs and PWEs

The updating procedure for both SSWEs and PWEs is very similar. When updating both types of working environments, we strongly suggest that you follow these general guidelines: (The updating instructions later on in the chapter are based on these guidelines.)

- Whenever you update a working environment, also update any other working environment that accesses it. First update the topmost working environment (the one that doesn't access any other one and is accessed by all others) and work "downwards". For example, suppose your team uses one SSWE, one SOWE and several PWEs. You would first update the SSWE (both SOWE and PWEs access it), then the SOWE (the PWEs access it) and finally the PWEs.
- When updating a working environment, compare all local, read-only files in it, *or* in any accessed working environment, to the latest version in the Repository. If the latest version in the Repository is newer, check it out in the working environment being updated.
- When updating a Private Working Environment, remove all read-only files in it. Do nothing to local, writable files (checked-out usually).

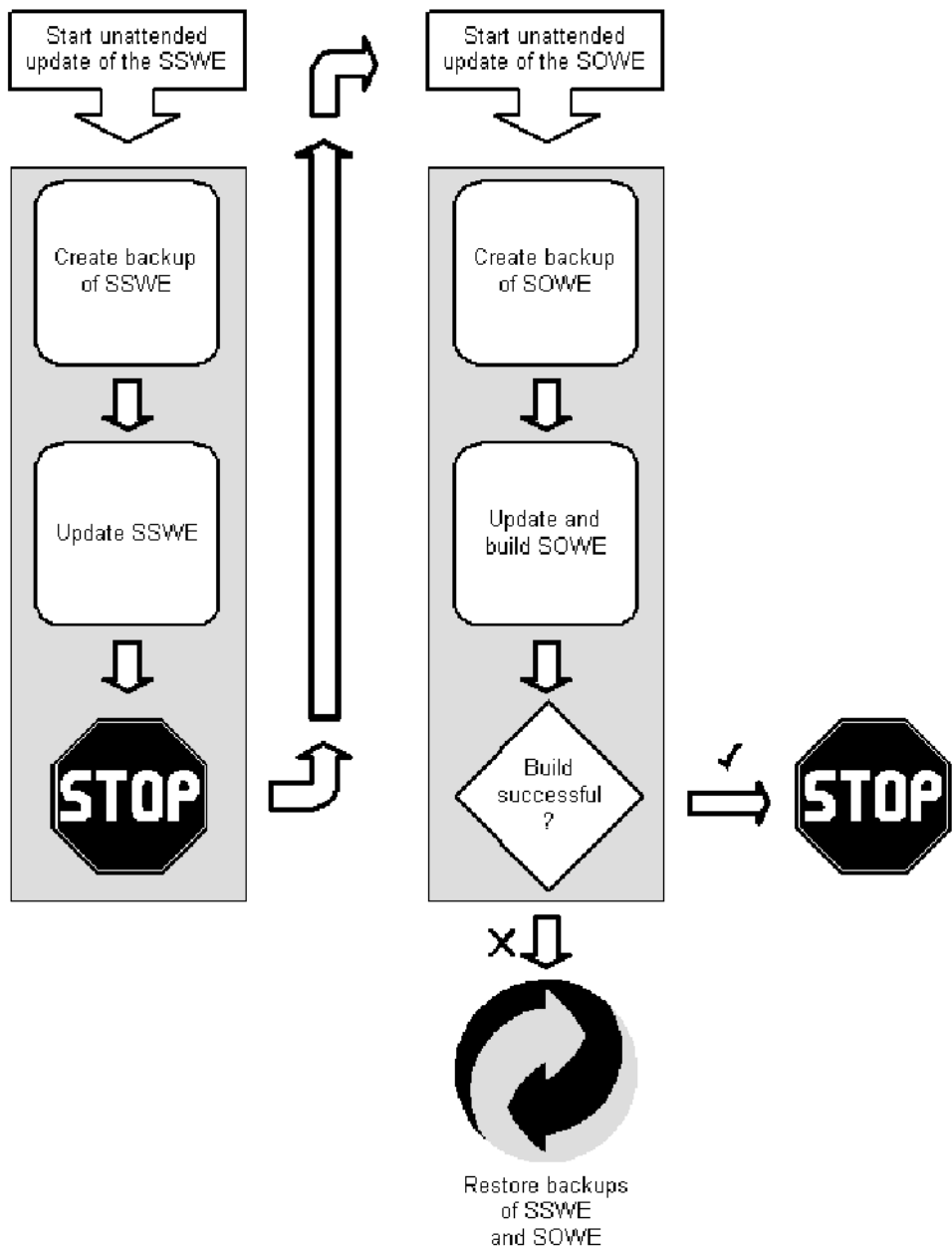
### Rollback in case of unsuccessful builds in the SOWE

Builds in a SOWE are not always successful, especially if one of your team members has checked in files without first testing them locally. To avoid such a situation, we recommend that you implement a rollback mechanism that restores your SSWEs and SOWE to their status before the update. The only implication for the team in such a case is that they will continue to work with the same version as they did before the unsuccessful update. Your Working Environments Administrator can then investigate the problem and fix it without disrupting your team's development work. Of course, implementing a rollback mechanism in your updating procedure means that you will have to make a backup of your SSWEs and SOWE before starting the update.



# Model for updating shared working environments

Here's how an updating and build process with a rollback mechanism might look like for your team's shared working environments:



## Workspace projects

To make it easier to update working environments, we recommend that you set up a *workspace project* as a root project of all other projects in a working environment. Workspace projects have two main advantages:

- To update all the projects in a working environment, you just have to update the workspace project, since SNIFF+ automatically updates all its subprojects for you.
- Unattended automatic nightly updates and builds are easier if there is a single workspace project for a working environment.

You should create workspace projects before you begin updating working environments for the first time. To create a workspace project for a working environment that already contains SNIFF+ projects:

1. Open the Working Environments tool and select the working environment for which you want to create the workspace project.
2. Choose **File > New Project... > with Defaults....**
3. Select the root directory of the working environment as the project directory.

A Project Attributes dialog for a new project is opened.

4. Select the **General** node.
5. Give the project a name that clearly describes it (e.g., `workspace.shared`).
6. Clear the **Generate Subproject Tree** check box, since you do not want to create SNIFF+ projects for the entire project tree again.
7. Press the **Ok** button to create the workspace project.

SNIFF+ will now generate the workspace project and its associated files. A Progress dialog appears to inform you of the generation process. When the generation process is over, SNIFF+ automatically opens the new project and displays its structure in a Project Editor.

8. Manually add all root projects of the working environment as subprojects by choosing **Project > Add subproject...** in the Project Editor for each project you want to add.

## Updating within SNIFF+

In this section, you will learn how to update your working environments. Working environments are to be updated in the following order:

- SSWEs
- SOWEs
- PWEs

Your Working Environment Administrator should be responsible for updating the shared working environments. PWEs may either be updated by the Working Environment Administrator, or by their respective owners.

Please complete the steps outlined in this section in a SNIFF+ session.

### Updating your team's SSWE

- Open the workspace project of the SSWE. If there is no workspace project for the working environment, create one. See also [Workspace projects — page 162](#)

#### In the Project Editor

1. Checkmark all the projects in the Project Tree.
2. Choose **Project > Synchronize Checkmarked Projects...**  
All files in the working environment are synchronized to the newest version.
3. Choose **Target > Update Makefiles...**  
Make support files are regenerated for all projects in the working environment.
4. To update configuration management information (optional), choose **Tools > Configuration Manager**.  
Configuration management information for all modified projects in the PWE is updated and stored on disk. For unmodified projects, shared information is used.
5. Close the Configuration Manager. To update cross-reference information (optional), open any tool where you can see symbols. Now choose **Info > Referred-By**.  
Cross referencing tables for all modified projects in the PWE are updated and stored on disk. For unmodified projects, shared information is used.

## Updating your team's SOWE

Please complete the following steps for each target platform for which you want to update the SOWE.

### In a Shell or Command Prompt

1. Change to the SOWE root directory and execute the following command on the command line:

```
gmake -i sniffclean
```

All symbol table, cross-reference files etc. are deleted in the working environment.

### In the Launch Pad or Working Environments tool

- Open the root project of the SOWE. We suggest that you open the project without symbols since this is faster.

If there is no root project for the working environment, create one. See also [Workspace projects — page 162](#)

### In the Project Editor

1. Checkmark all the projects in the Project Tree.
2. Choose **Project > Synchronize Checkmarked Projects....**  
All files in the working environment are synchronized to the newest version.
3. In the Project Editor, choose **Target > Update Makefiles....**
4. In the Update Makefiles dialog that appears, clear the **Generate Dependencies File** check box and press **Yes**.

Make support files are regenerated for all projects in the working environment.

### In a Shell or Command Prompt

In the SOWE root directory, execute the following commands on the command line:

- `gmake -i symbolic_link_to_dependencies_file`

Creates symbolic links to the dependencies file in the Shared Source Working Environment.

- `gmake -i CHECK_UPDATE=0`

Make is started for each project in the SOWE. The targets of the project are built recursively.

## Updating your team's PWEs

Complete the following steps to update a PWE:

- Open the workspace project of the PWE. If there is no workspace project for the working environment, create one. See also [Workspace projects — page 162](#)

### In a Shell or Command Prompt

- In the PWE root directory, execute the following command on the command line:

```
gmake -i clean
```

Object files, targets and core files are removed from the working environment.

### In the Project Editor

1. Checkmark all the projects in the Project Tree.
2. Choose **Project > Synchronize Checkmarked Projects...**  
All files in the working environment are synchronized to the newest version.
3. In the Project Editor, choose **Target > Update Makefiles...**  
Make support files are regenerated for all projects in the working environment.
4. To update configuration management information (optional), choose **Tools > Configuration Manager**.  
Configuration management information for all modified projects in the PWE is updated and stored on disk. For unmodified projects, shared information is used.
5. Close the Configuration Manager. To update cross-reference information (optional), open any tool where you can see symbols. Now choose **Info > Referred-By**.  
Cross referencing tables for all modified projects in the PWE are updated and stored on disk. For unmodified projects, shared information is used.

### In a Shell or Command Prompt

1. In the PWE root directory, execute the following command on the command line:

```
gmake -i symbolic_links
```

Symbolic links are created to object files in the SOWE and Makefiles in the SSWE. On Windows NT/95, local copies are made instead.

2. Execute the following command on the command line:

```
gmake -i CHECK_UPDATE=0
```

Make is started for each project in the PWE. The targets of the project are built recursively.

## Updating outside of SNIFF+

In this section, you will learn how to update your working environments outside of an active SNIFF+ session on both Unix and Windows using an update and build script (`updateWS.sh`) provided with your SNIFF+ installation. This script starts SNIFF+ in batch mode (driven via `sniffaccess`) and calls other tools like Make.

The update and build script is located in your `$SNIFF_DIR/ws_support` directory. Note that it can also be used for unattended updates of your working environments.

- On Unix, you can perform external updates from any Shell.
- On Windows, you can perform unattended updates from the Command Prompt.

### Update procedures

Working environments are to be updated in the following order:

- SSWEs
- SOWEs
- PWEs

Your Working Environment Administrator should be responsible for updating the shared working environments. PWEs may be updated either by the Working Environment Administrator, or by their respective owners.

After an update is over, SNIFF+ creates a log file with a summary report in the updated working environment's root directory. This summary report will also be automatically sent to you by email.

#### On Windows

To automatically receive summary reports, a mail program with a command-line interface is necessary. An example of such a tool is Postmail, a shareware program available in the public domain.

### How to

1. Change to the Working Environment root directory you want to update.
2. Call the script (all in one line)

- On Unix:

```
sh $SNIFF_DIR/ws_support/updateWS.sh
<WorkingEnvironmentName> <project> [SSWE | SOWE | PWE]
```

- In the Windows Command Prompt

```
sh %SNIFF_DIR%\ws_support\updateWS.sh
<WorkingEnvironmentName> <project> [SSWE | SOWE | PWE]
```

## Unattended updates

In this section, you will learn how to run unattended updates of your working environments using the same script discussed under [Updating outside of SNIFF+ — page 166](#).

- On Unix, unattended updates are run using `cron`.
- On Windows NT, unattended updates are performed using `at`. On Windows 95, you will need a utility for delayed command execution.

Unattended updates should be performed in the same order as regular updates:

- first your SSWEs
- then your SOWEs
- and finally your PWEs

Your Working Environment Administrator should be responsible for unattended updates.

## How SNIFF+ handles project structure changes during an unattended update

Your team members can modify the structure of a project in a PWE by checking out and modifying the project's project description file (PDF). When the modified PDF is checked in again, the project in the other working environments used by your team are structurally out of date. Its structure must be updated during an update of the working environments. The following is an overview of the possible changes and how SNIFF+ in batch-mode reacts to these structural project changes while updating a working environment:

- **A PDF has been modified** — Any structural change to a project requires a modification to its PDF. When you open a project, SNIFF+ checks whether its PDF is up to date, checks out the latest version if necessary, and then opens it.
- **Files have been removed from a project** — If files have been removed from a project, SNIFF+ does not delete them. It is the responsibility of the owner of the working environment to remove obsolete files. For a listing of potentially obsolete files in a working environment, you can use the **Check Obsolete Files** command in the Project Editor or the `sniffaccess` command.
- **Files have been added to a project** — If files have been added to a project, these files need to be checked out in your team's SSWE. When SNIFF+ opens a project, it checks out those files specified in the project's PDF that aren't present in the working environment.
- **New subprojects have been added to a project** — If a new subproject has been added to a project, the directory for this subproject needs to be created, and its PDF and the other files in the project need to be checked out. When SNIFF+ opens a project and it cannot find the PDF of a subproject, it creates the project directory and tries to check out the PDF and then to open it. Then the files of the new subproject are checked out if necessary.

- **Subprojects have been removed from a project** — If a subproject has been removed from a project, SNiFF+ does not delete it. It is the responsibility of the owner of the working environment to remove obsolete files. For a listing of potentially obsolete files in a working environment, you can use the **Check Obsolete Files** command in the Project Editor or the `sniffaccess` command.

## Examples of executing the update script on Unix

Here are some examples of how you would execute the update script:

- **Shared Source Working Environment**

```
00 02 * * * cd /shared_src; $SNIFF_DIR/ws_support/  
updateWS.sh <WorkingEnvironmentName> <project> SSWE
```

- **Shared Object Working Environment**

```
00 03 * * * cd /shared_obj; $SNIFF_DIR/ws_support/  
updateWS.sh <WorkingEnvironmentName> <project> SOWE
```

- **Private Working Environment**

```
00 04 * * * cd /private; $SNIFF_DIR/ws_support/updateWS.sh  
<WorkingEnvironmentName> <project> PWE
```

In the examples above, the commands are executed at 2, 3 and 4 o'clock in the morning, respectively.

### Note

For a description of the parameters, refer to [Parameters used with the update script — page 169](#).



## Parameters used with the update script

Parameter	Description
<WorkingEnvironmentName>	Refers to the working environment whose projects are to be updated. Enter the full name of the working environment, (including the PWE owner's username if there is one), as it appears in the Working Environments Tree. Examples: "SSWE:GA_teamSSWE" "SOWE:GA_teamSOWE" "Bob PWE:BobPWE" (username included) "PWE:BobPWE" (username not included)
<project>	Refers to the specific project that is to be updated. A project is specified by the project directory (as it appears in the <b>Project Directory</b> field in the Project Attributes dialog), followed by the project's PDF, e.g., COMPLEX/complex/complex.shared
[ SSWE   SOWE   PWE ]	Enter one of the three options to specify the type of working environment to be updated. SSWE — Shared Source Working Environment SOWE — Shared Object Working Environment PWE — Private Working Environment

## Running SNIFF+ without display (batch mode)

You can start a SNIFF+ session without a display.

SNIFF+ runs without a display when the `SNIFF_BATCH` environment variable is set to 1. In the SNIFF+ update script, this is set by default. To manually set the variable:

### On Unix:

1. Open a shell.
2. In a shell, set the `SNIFF_BATCH` environment variable to 1:

```
setenv SNIFF_BATCH 1
```

### On Windows:

1. Open a Command Prompt.
2. In the Command Prompt, set the `SNIFF_BATCH` environment variable to 1:

```
set SNIFF_BATCH=1
```



# Part VI

## Compiling and debugging



# Preprocessing C/C++ Code in SNIFF+

---

## Introduction

This chapter covers preprocessing C/C++ code in SNIFF+.

### For browsing WinAPI and /or MFC code

Please refer to the `readme.wri` file in your `%SNIFF_DIR%` directory.

## This chapter covers the following topics

- Enable full preprocessing
- Configure SNIFF+'s C/C++ Parser with a configuration file

## SNIFF+ concepts you should already know

- SNIFF+ projects

## Abbreviations used in this chapter

- Parser—SNIFF+'s C/C++ Parser
- PDF—Project Description File

## Preprocessing source code

By default, SNIFF+'s C/C++ Parser does not expand preprocessor macros when it parses source files. This approach has the advantage of speed, but occasionally some preprocessor macros confuse the Parser.

For example, macros that make the non-preprocessed source code syntactically incorrect confuse the Parser and may result in incomplete symbolic information. Such macros are called *non-syntactic macros*.

SNIFF+ provides several mechanisms to solve these kinds of problems:

- **Full preprocessing of the project** — If your code heavily uses non-syntactic macros, we recommend that you preprocess it. You can enable preprocessing for a single SNIFF+ project, or for a project structures.
- **Configuring the Parser** — You can configure the Parser to selectively preprocess your source code.

### Note

SNIFF+'s C/C++ Parser handles preprocessor options (e.g., `-Idir`, `-I-`, `-nostdinc`) in accordance with ANSI standards.

## Enabling full preprocessing

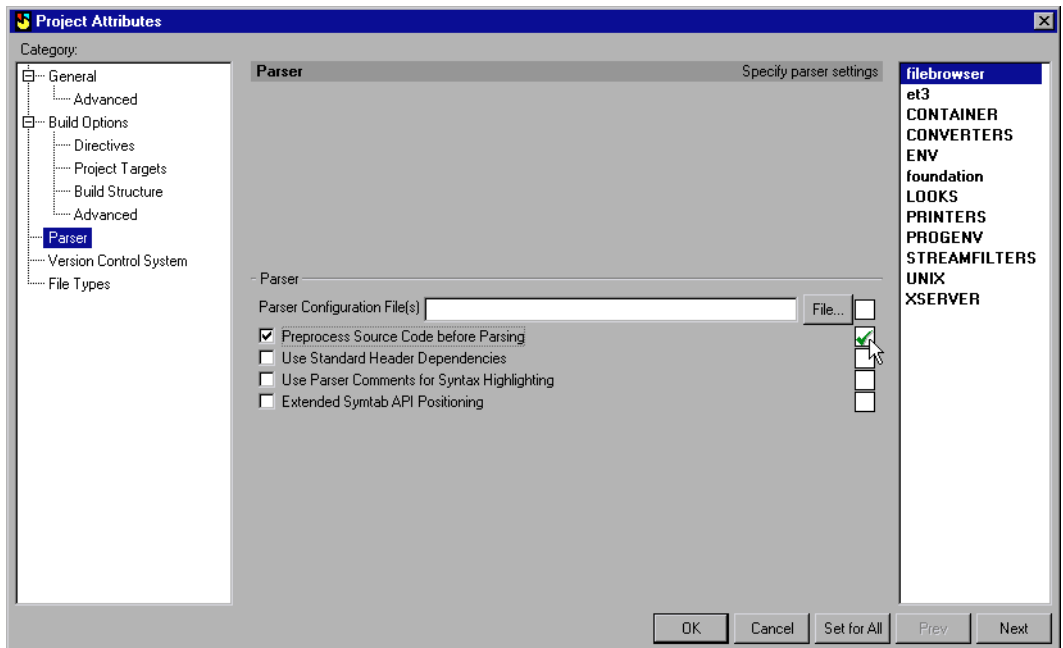
You should enable full preprocessing if your code heavily uses non-syntactic macros and you don't want to configure the Parser.

To enable full preprocessing:

1. In any open SNIFF+ tool, choose **Tools > Project Editor** to open the Project Editor.  
In the Project Tree, checkmark the projects for which you want to enable full preprocessing.
2. Check out the PDFs of the checkmarked projects.
3. Choose **Project > Attributes of Checkmarked Projects....**
4. Select the **Parser** node.

## In the Group Project Attributes dialog

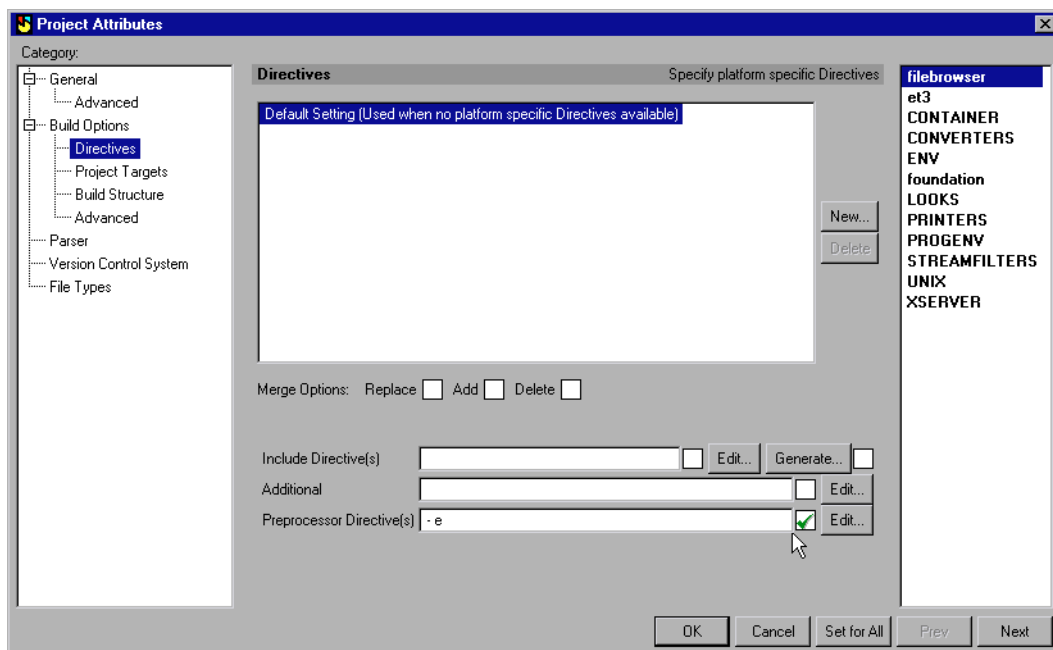
- Select the Parser node.



## In the Parser View

1. Select the **Preprocess Source Code before Parsing** check box.
2. Select the check box to the right of the above, this will make the attribute applicable to all Projects in the list on the right.
3. Press the **Set for All** button.

## 4. Select the Directives node.



## In the Directives view

## 1. Specify preprocessor directives.

**Note**

The syntax used by `cpp` applies to this field (`-Dmacro-spec` or `-Umacro-spec` directives separated by blanks).

**Specifying the "-L" option for selective preprocessing**

If you have an include file that defines problematic macros, you can just process this file using the "-L" option.

- Enter `-L<include_file>` in the **Preprocessor Directive(s)** field.

The preprocessor loads and expands `include_file` only. No other include files are processed.



### Specifying multiple "-L" options

1. Enter `-L"foo.h" -L<include/bar.h>` in the **Preprocessor Directive(s)** field.

This includes and expands the macros defined in the two files `foo.h` and `include/bar.h` in all files of the project as if the files were included with the normal `#include` statement. No other include files are processed, even if the source files contain `#include` statements.

2. To apply the settings in the **Preprocessor Directive(s)** field to all checkmarked projects, select the check box to its right.
3. Press the **Set For All** button.  
The attributes now apply to all the projects checkmarked in the Project Tree.
4. Press the **OK** button to close the Group Project Attributes dialog.

### In the Project Editor

1. Save all modified projects.
2. In order for full processing to take effect, you must also reparse the projects that you just modified. In the Project Tree, make sure the projects for which you've enabled full preprocessing are still checkmarked.
3. Choose **Project > Force Reparse**.

### Include path used by preprocessor

SNiff+ supplies the same include directives used by your compiler to your preprocessor.

### Speeding up preprocessing by caching

When preprocessing is enabled, parsing takes longer since all include files are parsed and macros are expanded. To speed up this process, SNIFF+ has an intelligent include file caching mechanism that only loads and parses an include file once.

### Advantages of the caching mechanism

- Whenever an include file is referenced a second time, the symbols and macros are taken directly from the cache.
- After a file has been successfully preprocessed and parsed, its symbol information is stored persistently.
- Like non-preprocessed files, preprocessed files are only reparsed when they are modified.

#### Note

Once a file has been preprocessed and its symbolic information stored to disk, SNIFF+ loads it just as quickly as a non-preprocessed file's symbolic information.

## Configuring the Parser with a configuration file

For every project, you can specify a file containing directives for the Parser. See also [Parser configuration file — page 178](#) Here, we assume that you are using the same configuration file for multiple projects.

### Specifying the location of your Parser configuration file

1. In any open SNIFF+ tool, choose **Tools > Project Editor** to open the Project Editor.  
In the Project Tree, checkmark the projects for which you will be using the Parser configuration file.
2. Check out the PDFs of the checkmarked projects.
3. Choose **Project > Attributes of Checkmarked Projects....**
4. Select the **Parser** node.

#### In the Parser view

1. Specify the location of the Parser configuration file in the **Parser Configuration File(s)** field.
2. Select check box to the right of the **Parser Configuration File(s)** field, this will make the attribute applicable to all Projects in the list on the right..
3. Press the **Set for All** button.
4. Press the **OK** button to close the Group Project Attributes dialog.

#### In the Project Editor

1. Save all modified projects.
2. In order for your changes to take effect, you must also reparse the projects that you just modified. In the Project Tree, make sure the projects for which you've enabled full preprocessing are still checkmarked.
3. Choose **Project > Force Reparse**.

### Parser configuration file

The Parser configuration file contains special configuration instructions for the Parser. The location of the file is specified by the **Parser Configuration File(s)** attribute in the **Parser** view of the Project Attributes and the Preferences.

The Parser considers the configuration file both when preprocessing is enabled and when it is disabled. If preprocessing is enabled, the directives in the configuration file are evaluated and executed after preprocessing.

## Parser directives and modifiers

The configuration file can contain the following directives: (The expressions in square brackets ([ ]) are modifiers you can use with directives, see [Modifiers — page 180](#) for details.)

Directives	Description
<code>ignore string <i>string</i> [leading] [trailing] [anywhere] [whole]</code>	The Parser ignores words that match <i>string</i> in the source code. If no option is specified only whole words are matched.
<code>ignore from <i>string1</i> to <i>string2</i> [exclusive] [instring] [incomment] [leading] [trailing] [anywhere]</code>	The Parser ignores anything between the two <i>strings</i> .
<code>ignore line <i>string</i> [begin] [instring] [incomment] [leading] [trailing] [anywhere]</code>	The Parser ignores lines that contain <i>string</i> .
<code>define <i>symbol</i></code>	The Parser resolves <code>#ifdef</code> and <code>#if</code> directives containing <i>symbol</i> .
<code>undefine <i>symbol</i></code>	The Parser resolves <code>#ifndef</code> , <code>#if ! defined</code> and <code>else</code> branches of <code>#if</code> directives containing <i>symbol</i> .

### Note

All preprocessor directives (e.g., `-I`, `-L`, `-U`) except for `-D` can also be used in the Parser configuration file. `cpp` syntax applies to all directives.

In addition to any ASCII character, *string* can contain `\n`, `\t` and `\nnn`, where `\nnn` is an octal number. The following table describes the modifiers you can use:

Modifiers	Description
[leading]	Means that <i>string</i> identifies a leading part of a word.
[trailing]	Means that <i>string</i> identifies a trailing part of a word.
[anywhere]	Means that <i>string</i> is matched anywhere in a word.
[whole]	Causes the Parser to ignore the whole word, even if just a part of the word is matched by <i>string</i> . Note: In order to use this modifier, you also have to use the [anywhere] modifier.
[exclusive]	Means that <i>string2</i> is not ignored. If the option is not present, the Parser ignores everything between <i>string1</i> and <i>string2</i> , inclusively.
[instring]	Means that the Parser also looks for <i>string</i> in strings delimited by quotes (" or ').
[incomment]	Means that the Parser also looks for <i>string</i> in comments delimited by // or /*...*/.
[begin]	Causes the Parser to ignore only lines beginning with <i>string</i> . If this option is not present, the location of <i>string</i> is not important.

```
# Example ignore strings file

#
ignore string VIRTUAL          # for NIHCL
ignore string _C_ARG1         # for the License project
ignore from EXEC to ;         # ignore all embedded SQL state-
                              # ments
define UNIX                   # resolve ifdefs for UNIX
```

## Examples — Parser configuration

### Strings, lines and constructs to be ignored by the Parser

The Parser can be configured to ignore strings, anything between two strings and lines containing a certain string.

#### Example of ignoring strings

The `VIRTUAL` macro is used in the NIH class library in class definitions like this:

```
class A : public VIRTUAL B
{ ... };
```

The `VIRTUAL` string confuses the Parser since the C++ syntax requires that a class name must follow the keyword `public`.

To solve the above problem without doing full preprocessing:

- Tell the Parser to ignore the `VIRTUAL` string:

```
ignore string VIRTUAL
```

**IMPORTANT:** All affected projects must be reparsed after the configuration file has been changed.

### Resolving `#ifdef` and `#if` directives

Some class libraries use the preprocessor directives `#ifdef` or `#if` to modify the code in a way that confuses the Parser. In such cases the Parser configuration file allows you to selectively resolve `#ifdef` or `#if` directives without doing full preprocessing.

#### Scenario 1: resolving different class definitions for the same class

```
#ifdef UNIX
class someClass : unixBaseClass
#else
class someClass : otherBaseClass
#endif
{ ... };
```

Since `SNiFF+` normally parses the file without resolving `#ifdef` statements, it reads two class definition headers and just one actual definition.

To solve this problem, you have two possibilities:

- add the following line to the Parser configuration file:

```
define UNIX
```

This tells the Parser to ignore the line between the `#else` and the `#endif` directives.

- add the following line to the Parser configuration file:  
`undefine UNIX`  
 The Parser will ignore the line between `#ifdef` and `#else`.

## Scenario 2: resolving unbalanced braces

```
#ifdef HUGE_INT
for (int i=0; i<MAXVAL; i++) {
#else
for (long i=0; i<MAXVAL; i++) {
#endif
... }
```

Since SNIFF+ normally parses files without resolving `#ifdef` statements, it reads two opening and only one closing brace.

To solve this problem, you have two possibilities:

- add the following line to the Parser configuration file:  
`define HUGE_INT`
- remove the opening brace from the two `for` lines and put it after the `#endif` directive.

## Scenario 3: resolving complex `#if` directives

```
#if defined (UNIX) || defined (VMS)
class someClass : unixBaseClass
#else
class someClass : otherBaseClass
#endif
{ ... };
```

The expression after the `#if` directive will be evaluated only if it contains the `||`, `&&`, `!` (logical negation), `defined` operator and parentheses for grouping. If the expression contains other operators or a `defined` operator with an identifier that does not appear in the Parser configuration file, the `#if` is not resolved (i.e., both branches are parsed).

For example, assuming that your configuration file contains the following:

```
define AAA
undefine BBB
```

Then:

- AAA is defined
- BBB is not defined
- CCC is neither defined nor undefined

in the preprocessor call. Then, from the following source code, only a, d, e and f will appear in the Symbol Table.

```
#if defined(AAA)
int a;
#else
int b;
#endif
#if defined(AAA) && defined(BBB)
int c;
#else
int d;
#endif
#if defined(CCC)
int e;
#else
int f;
#endif
```





## Introduction

This chapter covers how to compile and debug in SNIFF+.

### This chapter covers the following topics

- Build a project's targets
- Run and debug executables
- Use special SNIFF+ help targets

### Assumptions made in this chapter

- If you use SNIFF+'s Make Support, your Working Environments Administrator has already set it up for your projects.
- If you don't use SNIFF+'s Make Support (you use your own Makefiles), you have already read [Using Your Own Makefiles — page 105](#)

### Related SNIFF+ topics

- Setting up Make Support — [Build and Make Support — page 73](#)

### Abbreviations and shortcuts used in this chapter

SSWE — Shared Source Working Environment

SOWE — Shared Object Working Environment

PWE — Private Working Environment

\$SNIFF\_DIR — path to your SNIFF+ installation directory

## Building a project's targets

This section only applies if you use SNIFF+'s Make Support. If you use your own Makefiles, please refer to [Using Your Own Makefiles — page 105](#).

#### Note

By default, debugging information is generated for targets compiled in SNIFF+. To turn off this default behavior, please refer to [Specifying platform-specific Make information — page 101](#).

You can build targets in the Project Editor, the Source Editor and the Shell. Here, the Project Editor is used. To build the project's targets:

1. Open the project whose target(s) you want to build.
2. Update Make Support Files:
  - Choose the **Update Makefiles...** command from the **Target** menu.
  - Press the **OK** button in the dialog that appears to update the project's dependencies information.

SNIFF+ generates or updates the Make Support Files of all projects checkmarked in the Project Tree.
3. If you work in a PWE that accesses an SOWE, choose the **Make > symbolic\_links** command from the **Target** menu.
 

SNIFF+ creates symbolic links in your PWE to object files and targets in the SOWE (for details, please refer to [Sharing object files — page 76](#)).
4. In the Project Tree, select the project whose target(s) you want to build.
5. Choose the appropriate **Make** command from the **Target** menu:
  - To build the default target of the project, choose the **Make default target** command.
  - To recursively build the default target of the project and each of its subprojects, choose the **Make all** command.

## Results

You should see the following results, depending on which command you executed.

- For both commands:
 

When your Make utility is called in a project directory, it checks the project's default target against its dependencies. Any dependencies that need recompiling are recompiled, and the default target is then built.
- If you chose the **Make default target** command:
 

SNIFF+ opens a Shell tool and executes the project's Make command on the command line.
- If you chose the **Make all** command:
 

SNIFF+ opens a Shell tool and executes `make all` on the command line.

SNIFF+'s General Makefile defines rules for building the `all` target. Basically, SNIFF+ recursively executes the **Make default target** command in the project directory and each subproject directory listed in the **Recursive Make Dirs** field of the project's Make attributes. To see an example of how this works, please refer to [Building targets recursively — page 88](#).

## Trouble Shooting

The following trouble shooting tips may come in handy:

- If the **Make Target** command is disabled:

You have forgotten to enter the name of the project's default target while setting up Make Support. For details, please refer to [Setting up Make Support — page 88](#).

- Your Make utility outputs an error message similar to the following:

```
$SNIFF_DIR/make_support/.mk no such file or directory
```

where \$SNIFF\_DIR is your sniff installation directory.

The PLATFORM environment variable is not set. Set this variable to the value returned by the sniff\_arch program, restart SNIFF+ and try again. Here's an example of how you would set the variable in the C-shell:

```
setenv PLATFORM `sniff_arch`
```

## Running a project's executable

To run an executable:

1. Choose the **Run target** command in the **Target** menu.  
A Program Arguments dialog appears.
2. Enter any arguments for the executable and press **OK**.  
SNIFF+ then opens a Shell tool and runs the target.

## Debugging targets

**On Windows NT/95**, you can use the Java Debugger and the MSDevStudio integration for debugging.

**On Unix**, do the following:

### Choosing a debugger adaptor

To choose a debugger adaptor:

1. Choose the **Preferences** command from any open SNIFF+ tool.  
The Preferences dialog appears.
2. Select the **Platform** node.

### In the Platform view

1. From the Platform list, select the platform on which you debug.
2. Select the **Debugger** tab.  
The Platform Settings dialog appears.
3. From the **Adaptor** drop-down menu, choose the debugger adaptor for your debugger.
4. Press **Ok** to apply and save your Preferences.

## Debugging

To debug targets:

1. Choose the **Debug target** command in the **Target** menu of the Project Editor, Source Editor or the Shell.

### Note

If the **Debug target** command is not enabled, either the target name is not specified in the project's Make attributes or the executable does not exist in the project directory.

You should notice two things:

- The Debugger dialog appears. For details, please refer to [Debugger \(Unix and Java\) — page 75](#).
  - The Source Editor is now in debugging mode and a button bar with the most common debugger commands appears in it.
2. Execute debug commands. Either type in debug commands on the Debugger's command prompt or use the button bar in the Source Editor or select the commands in the Debugger.

## Some useful debugging commands

### Setting a breakpoint

- Choose the line in the Source Editor where you want to set a breakpoint.
- Press the **Break At** button.

A small stop sign at the beginning of the line indicates the breakpoint.

### Displaying values

- Select a variable by double-clicking on it in the Source Editor.
- Press the **Print** button.

The Debugger displays the current value of the variable.

### Single-stepping

There are two possibilities for single-stepping:

- **Next** steps over functions and methods.
- **Step** steps into functions and methods.

## Showing the call hierarchy

- Press the **Stack** button in the Source Editor or select the **Callstack** tab in the Debugger. The call chain is displayed.

## Quitting the Debugger

- Choose **Close Tool** from the **Tools** menu. The Debugger closes. The button line with debugging commands is removed from the Source Editor after the Debugger quits.

## Source Editor in debugging mode

When the Source Editor is in debugging mode, all files loaded in it are read-only and a row of new buttons is added to the tool.



Command	Description
<b>Run</b>	Runs the application being debugged from scratch.
<b>Cont</b>	Continues interrupted execution.
<b>Step</b>	Single-steps into the next function/method.
<b>Next</b>	Single-steps over the next function/method.
<b>Break In</b>	Sets a break point at the first execution line of a selected function/method.
<b>Break At</b>	Sets a breakpoint at the current cursor position.
<b>Clear</b>	Clears the breakpoint in the current line. The cursor must be positioned to a line with a breakpoint.
<b>Print *</b>	Prints the value pointed to by the current selection. The selection must evaluate to a valid pointer.
<b>Print</b>	Prints the value of the current selection. The selection must evaluate to a valid variable.
<b>this</b>	Prints the value of the current object.
<b>Stack</b>	Displays the current call stack.
<b>Up</b>	Goes one stack frame up in the call hierarchy. A reusable Source Editor is automatically positioned at the source location of the new stack frame.
<b>Down</b>	Goes one stack frame down in the call hierarchy. A reusable Source Editor is automatically positioned at the source location of the new stack frame.

## SNIFF+ help targets

This section only applies if you use SNIFF+'s Make Support. If you use your own Makefiles, please refer to [Using Your Own Makefiles — page 105](#).

Your SNIFF+ installation's General Makefile defines rules for building a number of special targets known as *help targets*. These targets are used by SNIFF+ in a number of situations, such as when you build targets using Make Support and update your working environments. Note that you can view a complete listing of all help targets by:

- choosing **Target > Make > help** during a SNIFF+ session, or by
- browsing the General Makefile (`$SNIFF_DIR/make_support/general.mk`).

In this section, you will learn how to use some of SNIFF+'s help targets for cleaning up your working environments and making symbolic links (on Unix only).

### Note

All the help targets listed here work recursively. That is, when you build one of the help targets in a project, the target is also built in each of the recursive subprojects listed for the project in the **Recursive Make Dir(s)** field in the **Build Options > Build Structure** view of the Project Attributes dialog.

## Cleaning up working environments with help targets

Here's a list of the help targets you can use for cleaning up working environments:

- **clean\_targets**—Removes all (non-Java) targets in the current project directory and all recursive subproject directories.
- **clean\_objects**—Removes all (non-Java) objects in the current project directory and all recursive subproject directories.
- **clean**—Removes all object files, Java class files and targets in the current project directory and all recursive subproject directories.
- **sniffclean**—Removes all SNIFF+ symbolic information files (`*.symtab`) in the `.sniffdir` directory of the current project directory and all recursive subproject directories.
- **clean\_state**—Removes all SNIFF+ state files (`*.state*`) in the `.sniffdir` directory of the current project directory and all recursive subproject directories.
- **clean\_backup**—Removes all backup files created by SNIFF+ in the current project directory and all recursive subproject directories.

## Creating symbolic links (local copies) with help targets

### On Windows

Symbolic links are not supported. As a result, SNIFF+ creates local copies instead of symbolic links. If disk space is an issue, please use the above help targets with care.

Here's a list of the help targets you can use for creating symbolic links. Note that all these targets are built in the current project directory and all recursive subproject directories in the current working environment, so directories above it are not affected:

- **symbolic\_links**—Recursively creates symbolic links to object files and targets in the SOWE accessed by the current working environment. Also creates symbolic links to Makefiles in the SSWE accessed by the current working environment.
- **symbolic\_links\_to\_dependencies\_files**—Just for use in the SOWE. Use when you want to open projects in the SOWE without symbol information. Creates symbolic links to dependencies files in the SSWE.

## Building the help targets

To build help targets, please complete the following steps:

1. Open the project for which you want to build help targets.
2. Update Make Support Files:
3. Choose the **Update Makefiles...** command from the **Target** menu.
4. Press the **OK** button in the dialog that appears to update the project's dependencies information.

SNIFF+ generates or updates the Make Support Files of all projects checkmarked in the Project Tree.

5. Use SNIFF+'s **Target > Make...** walking menu to build the following targets:

- `symbolic_links`
- `clean_targets`
- `clean`
- `help`

6. To build the other targets, open a SNIFF+ Shell tool and execute the following:

- **On Unix:** `gmake <help_target_name>`
- **On Windows NT/95:** `sniffmake <help_target_name>`





## Introduction

This chapter covers how to use SNIFF+ for multi-platform development on Unix and Windows NT/95 systems. It is assumed that readers of this chapter are already familiar with creating and working with SNIFF+ Projects and Working Environments.

## How SNIFF+ supports cross-platform development

SNIFF+ supports cross-platform development by:

- supporting consistent working environments on Windows and Unix
- making it possible to administrate your projects on just one platform
- providing consistent source code administration using a central repository and the same CMVC tools on all platforms
- correctly processing carriage-return symbols (Unix LF, Windows CRLF) on the given platform. No conversion is made when saving a file on one platform and opening it on another.
- using a single make concept on both platforms
- differentiating between Unix `*.c` (C files) and `*.C` (C++ files) on Windows
- simulating Unix symbolic links by means of file copy on Windows
- separating platform-specific source code and objects (by using working environments)

## Limitations

Please be aware of the following limitations when working with multi-platform projects with SNIFF+:

- the same RCS version must be used on both platforms (Package includes RCS 5.7 for Windows/Unix)
- when naming files on Windows, be aware that Unix distinguishes between upper and lower case

- shared object working environments are not supported on Windows

**Note**

When developing software for 2 or more platforms, shared multi-platform source files located in a shared source working environment should not contain any platform-specific system calls (via an API)!

On Windows more so than on Unix, names of standard libraries (e.g., standard C++ classes) do not necessarily conform to a standard naming convention. Furthermore, naming conventions are strongly compiler-dependent (on Windows).

If you use standard libraries that have different names for different target platforms, adapt your platform-specific Makefiles accordingly.

- When using an adaptor other than the `RCS_CROSS` adaptor, all Window usernames must be mapped to a single Unix user (e.g. `rcsadmin`) in order to access an RCS repository located on Unix. Thus we suggest that you use the `RCS_CROSS` adaptor. The `RCS_CROSS` adaptor extends the functionality of the `RCS` adaptor and sets the affected repository files to writable for the user and the group when you execute commands that modify the Repository files.

## Cross-platform development vs. remote compile & debug

### Cross-platform development

- Setup for multiple target platforms is possible.
- All program sources can be shared.
- Cross-platform setup relates to setting up the working environment tree. For an example of a working environment tree, see [Example Working Environments Tree — page 204](#).

### Remote compile and debug

- Compiling and debugging on a different target platform is possible.
- Program sources can be shared.
- Remote compile and debug relates to setting up one working environment in which you can compile and debug remotely.
- You can combine the cross platform development concept and the remote compile and debug concept to use one or more of the working environments in the working environment hierarchy to compile and debug remotely.

See also [Remote Compile and Debug — page 207](#).

## Basic differences between Windows NT and Unix

Here's a list of basic differences you should be aware of during cross-platform development on Windows and Unix:

Difference	Windows	Unix
Datasystem: Pathnames <ul style="list-style-type: none"> <li>■ Directory separator</li> <li>■ Relative declaration</li> <li>■ Absolute declaration</li> </ul>	Concept of diskdrive conventions <ul style="list-style-type: none"> <li>■ \</li> <li>■ ..\dir1\src</li> <li>■ d:\dir1\dir2\src</li> </ul>	File systems would be mounted from '/' <ul style="list-style-type: none"> <li>■ /</li> <li>■ ../dir1/src</li> <li>■ /usr/lib/bin</li> </ul>
Datasystems: Symbolic Links	unknown	available
Datasystems: Networkaccess	<ul style="list-style-type: none"> <li>■ no standard NFS support</li> <li>■ uses LAN Manger for Networkaccess (de facto standard)</li> <li>■ Addressing of network resources via UNC</li> </ul>	<ul style="list-style-type: none"> <li>■ uses NFS for Networkaccess (de facto standard)</li> <li>■ no standard LAN Manager support</li> <li>■ no special addressing, all resources are added in the filesystem</li> </ul>
Datasystem: Case Sensitivity	no	yes
Datasystem: valid datanames	<ul style="list-style-type: none"> <li>■ FAT (Dos): no use of " &amp; * + , / : ; &lt; = &gt; ? [ ] \ ^  </li> <li>■ VFAT (Win95): no use of " &amp; * / : &lt; &gt; ? \ ^  </li> <li>■ NTFS (NT): no use of " * / : &lt; &gt; ? \  </li> </ul>	All symbols except of "/" can be used in the datanames
Datasystem: Data access	<ul style="list-style-type: none"> <li>■ NTFS: access list</li> <li>■ FAT: read – only bit</li> <li>■ VFAT: read – only bit</li> </ul>	Access control per UserID / GroupID
Order of data expansion	<ul style="list-style-type: none"> <li>■ Objectfiles: .obj</li> <li>■ Libraries: .lib</li> <li>■ C source files: .c</li> <li>■ C++ source files: .cpp</li> </ul>	<ul style="list-style-type: none"> <li>■ Objectfiles: .o</li> <li>■ Libraries: .a</li> <li>■ C source files: .c</li> <li>■ C++ source files: .C</li> </ul>

Difference	Windows	Unix
User ID's	<ul style="list-style-type: none"><li>■ Windows NT users "worldwide" unique User ID's</li><li>■ Windows NT: User ID's are given by the system.</li><li>■ Use of a User ID over the network is only possible with a Domain Controller (NT Server)</li><li>■ There exists no suitable User ID concept under Windows 95</li></ul>	<ul style="list-style-type: none"><li>■ Unix uses numerical values for User ID's and Group ID's</li><li>■ ID's can be provided by the administrator</li><li>■ User ID's can be used uniformly in the entire network</li></ul>

The Shell tool in SNIFF+ is a Unix shell which uses only Unix conventions.



## Setup assumptions

We assume the following scenario for describing how to configure SNIFF+ for cross-platform development:

- Files, shared working environments and the Repository are all located on a Unix machine. You develop your cross-platform projects on Windows machines that access this Unix machine.
- There are four developers: 2 work on Windows and the other 2 on Unix. All developers have their own Private Working Environments.
- The cross-platform project consists of the root directory `complex` and two subdirectories, `complexlib` and `iolib`.

## Cross-platform setup — Unix side

This section goes through the steps that you must complete on Unix for configuring SNIFF+ for cross-platform development.

### Initial remarks

As part of this step-by-step guide, we will first create the directory structure for the SNIFF+ cross-platform project on a Unix machine. Note that this directory must be made accessible to Windows machine(s) via NFS or LanManager.

### Working environment structure

In the steps that follow, we assume that the directory

```
/Projects/work/cross
```

contains the individual root directories of the repository working environment (RWE), shared source working environment (SSWE) and shared object working environment (SOWE)

```
/Projects/work/cross/rwe
```

```
/Projects/work/cross/sswe
```

```
/Projects/work/cross/sowe
```

The root directory of each individual private working environment (PWE) is assumed to be

```
$HOME/Project1/cross
```

The environment variable `$HOME` refers to the home directories of the individual users (two on Unix, the other two on Windows). Assuming that all user home directories are located under `/Users` on Unix and `D:\Users` on Windows:

```
HOME = /Users/user1
HOME = /Users/user2
HOME = D:\Users\user3
HOME = D:\Users\user4
```

## Project source files

The source files of the COMPLEX project are in the SSWE root directory.

## Summary of the directory structure

Here's a summary of the directory structure we'll be working with:

```
/Projects/work/cross
  /rwe
  /sowe
  /sswe
    /complex
      /iolib
      /complexlib
```

Each developer's private working environment root directory is:

```
$HOME/Project1/cross
```

`$HOME` is set to each individual's home directory.

## Setting necessary variables, links and permissions

We are now ready to set the variables, links and permissions necessary on the Unix side for cross-platform development.

- Start a shell (e.g, `bash`, `csh`)

### In the shell

1. Set an environment variable named `WS_CROSS` to `/Projects/work/cross`. Remember that `/Projects/work/cross` contains the individual root directories of the RWE, SSWE and SOWE.

By setting an environment variable to `/Projects/work/cross`, you can use it to refer to the directory on both Windows and Unix without having to change any project and working environment attributes.

2. Launch SNIFF+.



## In SNIFF+

1. Choose **Tools > Preferences...** in any open SNIFF+ tool.  
The Preferences dialog appears.
2. Select the **Tools > Working Environments** node.
3. Enter `$WS_CROSS` in the **Working Environment Config. Directory** field.  
Your working environment files will be stored in the directory specified by `$WS_CROSS`.
4. Select the **Platform** node.
5. Select the **Make Support** tab, make sure that the Make command in the **Make Command** field is set correctly.

### Note

For the Make command to be executed, the Make Command field in the **Build Options** view of the Project Attributes dialog must be empty.

6. Quit SNIFF+.

## Cross-platform setup — Windows side

This section goes through the steps that you must complete on Windows for configuring SNIFF+ for cross-platform development.

### Initial remarks

No team directories need to be created on the Windows side, since all project-relevant files are located on a Unix machine (in this example, the name of the machine is `BRUTUS`). The only required directory is

`Project1/cross`

This directory should be created in each team member's home directory.

Note that standard Windows UNC notation (e.g., `\\brutus\Projects`) cannot be used when specifying directories in SNIFF+. As a result, network resources must be mapped to a network drive. In this example, we assume that `\\brutus\Projects` is mapped to the `F` drive.

## Setting necessary variables and links

We are now ready to set the variables and links necessary on the Windows side for cross-platform development.

### In the shell

1. Set an environment variable called `WS_CROSS` to the root of your shared working environments on Unix. In this example, `WS_CROSS` would be set to:

```
F:\Projects\work\cross
```

(Remember that `/Projects/work/cross` contains the individual root directories of the RWE, SSWE and SOWE.)

An environment variable called `WS_CROSS` is now set on both Unix and Windows to the same directory (which is physically located on the Unix side).

2. For each user, set an environment variable called `HOME`. In this example `HOME` for `user1` would be set to `HOME=D:\Users\user1`. For `user2`, `HOME` would be set to `HOME=D:\Users\user2`.

(Remember that all home directories on Windows in this example are located in the `D:\Users\` directory.)

3. Launch **SNiFF+**.

### In **SNiFF+**

1. Choose **Tools > Preferences...** in any open **SNiFF+** tool.

The Preferences dialog appears.

2. Select the **Tools > Working Environments** node.

3. Enter `$WS_CROSS` in the **Working Environment Config. Directory** field.

Your working environment files will be stored in the directory specified by `$WS_CROSS`.

4. Quit **SNiFF+**.

The necessary attributes have now been set on both Unix and Windows. The next step is to set up the working environments necessary for cross-platform projects. To ensure that the project directories for the cross-platform project have the correct permissions, we recommend that you perform this setup on the Unix side.

## Setting up the working environments

How to set up working environments is covered in the language-specific tutorials supplied with your SNIFF+ package. This section only covers those steps needed for cross-platform development. For instructions on how to define working environments and specify their root directories, please refer to the tutorial for your language.

The following steps should be completed by your Working Environments Administrator (WEAdmin).

- Launch SNIFF+.

### In SNIFF+

1. Open the Working Environment tool by choosing **Tools > Working Environments**.

The Working Environment tool appears.

2. Choose **Utils > User Permissions....**

The Users dialog appears.

3. Give yourself (the WEAdmin) permission to create all four types of working environments.
4. Create your working environments.

### In the shell

- Set group read/write permissions for the Unix directory `$WS_CROSS/.WEProject-Cache`

```
chmod g+w $WS_CROSS/.WEProjectCache
```

This directory is generated when you press the **Update List** button in the Projects dialog of the Working Environments tool.

## Example Working Environments Tree

In this document, the working environment root directories are:

- On Unix, for the RWE, SSWE and SOWE:
  - /Projects/work/cross/rwe
  - /Projects/work/cross/sswe
  - /Projects/work/cross/sowe
- On Unix, for the PWEs of user1 and user2:
  - /Users/user1/project1/cross
  - /Users/user2/project1/cross
- On Windows, for the PWEs of user3 and user4:
  - D:\Users\user3\project1\cross
  - D:\Users\user4\project1\cross

Based on this information, the Working Environments Tree for this example would be:

```

RWE: Unix_REPOSITORY      (Directory $WS_CROSS/rwe)
├── SSWE: Unix_SSWE       (Directory $WS_CROSS/sswe)
│   └── SOWE: Unix_SOWE   (Directory $WS_CROSS/sowe)
│       ├── PWE: PWE_Unix_user1 (Directory $HOME/Project1/cross ; Owner user1)
│       ├── PWE: PWE_Unix_user2 (Directory $HOME/Project1/cross ; Owner user2)
│       ├── PWE: PWE_Windows_user3 (Directory $HOME/Project1/cross ; Owner user3)
│       └── PWE: PWE_Windows_user4 (Directory $HOME/Project1/cross ; Owner user4)

```

## Setting up the shared project on Windows

This section only covers those new project attributes needed for cross-platform development.

### In the Working Environment tool

1. Select the Shared Source Working Environment in which you want to create the cross-platform project.
2. Choose **File > New Project > with Defaults....**  
The Directory dialog appears.
3. In the Directory dialog, navigate to and select the directory  
\$WS\_CROSS/sswe  
The Attributes of a New Project dialog appears.

## In the Attributes of a New Project dialog

1. Select the **Version Control System** node.
2. From **VCS Tool** drop-down menu, choose **RCS\_CROSS**.

We suggest that you use the RCS\_CROSS adaptor because, unlike other adaptors, it sets the affected repository files to writable for the user and the group when you execute commands that modify the Repository.

3. Set all other new project attributes as needed and then press **Ok** to start generating the project.

If you use standard libraries that have different names for different target platforms, do **not** enter these libraries in the **Libraries Linked** field when specifying your project targets. Instead, adapt your platform-specific Makefiles. To do so:

## In your platform specific Makefile

Modify the following lines:

```
# platform specific library
OS_LIBS =

to

# platform specific library
OS_LIBS = -L <path to your library>
```



# Remote Compile and Debug

---

## Introduction

SNiFF+ supports remote compiling and debugging, allowing programmers in multi-platform environments to develop on the platform of their choice (Windows NT, Windows 95 and all major Unix platforms) and transparently compile, debug and execute commands on a remote Unix machine.

## This chapter covers the following topics

- How to compile and debug remotely

## Assumptions made in this chapter

- If you use SNiFF+'s Make Support, your Working Environments Administrator has already set it up for your projects.
- If you don't use SNiFF+'s Make Support (you use your own Makefiles), you have already read [Using Your Own Makefiles — page 105](#)

## Related SNiFF+ topics

- Compiling and debugging — [Compiling and Debugging in SNiFF+ — page 185](#)

## Overview

To remote compile and debug, you must select a target platform and specify the platform settings for it, i.e., target machine name, user name, sniff installation directory on the target platform (if applicable) in the Platform view of your Preferences. You can also specify other general platform settings if need be. These settings are stored in the following directory:

- On Unix:

```
$HOME/.sniffrc/Preferences/Platforms/<Platform_name>.sniff
```

- On Windows:

```
%SNIFF_DIR%\Profiles\<Username>\Preferences\Platforms\<Platform_name>
.sniff
```

So for each target platform, for which you define platform settings, a new file is created.

Next, in the Working Environments tool, you will assign a target platform to each Private Working Environment or Shared Object Working Environment. If you don't assign a target platform to a particular working environment, the default platform will be used. Information about which working environment is assigned to which platform is stored in:

- `<your_sniff_installation_directory>/workingenvs/WorkingEnv-Data.sniff`

Thus when you open a project in a working environment and compile it, SNIFF+ uses the information in the platform settings to compile and debug. This makes it possible for you to compile and debug locally or remotely as desired.

## Requirements

### Licensing

To compile and debug remotely, you need a SNIFF\_EVAL License or a SNIFF\_CROSS license. You can check to see if you have one of these licenses by choosing **Help(?) > Licenses...** in the Launch Pad.

### SNIFF+ Installation

There are two possibilities:

- If SNIFF+ is available for the target and host platform, we recommend that you install both SNIFF+ executables in a common directory that is mounted on the host platform. The version of SNIFF+ installed on the target platform must be platform specific, i.e., if your target platform is Solaris, SNIFF+ for Solaris must be installed. This is recommended when you use SNIFF+'s Make Support or when you want to make nightly updates or for any other administrative work on the target platform. These tasks are much faster when done on the target platform.
- If SNIFF+ is not available for the target platform, install it on the host platform. By doing so, you won't be able to use SNIFF+'s Make Support, make nightly updates, or do any other administrative work on the target platform.

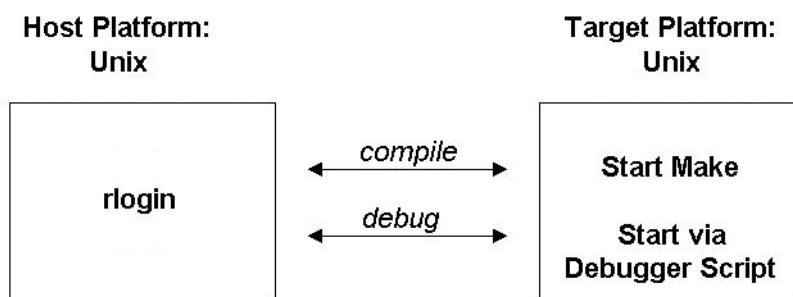


## Scenarios

### Unix - Unix

When developing on a host Unix machine and compiling and debugging on a remote Unix machine, please note:

- The `rlogin` tool, provided with the Unix operating system, is used.
- The debugger script **remote\_debug.sh**, is located in `$SNIFF_DIR/bin` directory. When you start the debugger remotely, this script is copied to the working environment root directory and is later used to start the debugger. You can set other debugger commands/parameters in the original script.



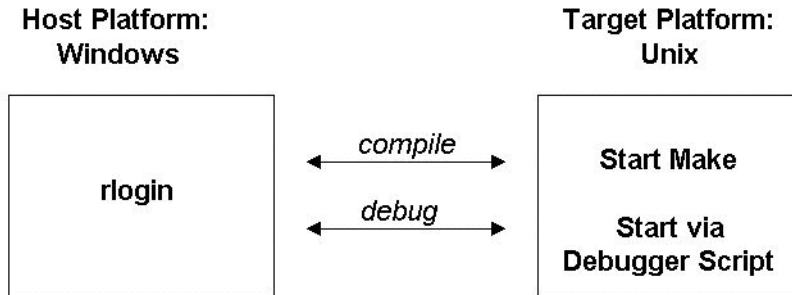
To access your working environments from both the host and the target machine, you can either mount directories or you can create symbolic links to access the working environment root directory so that it looks the same from both machines.

### Windows - Unix

When developing on a host Windows machine and compiling and debugging on a remote Unix machine, please note:

- The `rlogin` tool is delivered with `SNIFF_EVAL` and `SNIFF_CROSS` and has been adapted for remote compiling and debugging. Thus remote compile and debug is only supported by SNIFF+ if the supplied `rlogin` is used on Windows.

- The debugger script **remote\_debug.sh**, is located in %SNIFF\_DIR%\bin directory. When you start the debugger remotely, this script is copied to the working environment root directory and is later used to start the debugger. You can set other debugger commands/parameters in the original script.



We require that all working environments are located on the Unix machine and are accessible on Windows. For details, please refer to [Cross-platform setup — Windows side — page 201](#). By doing so, nightly updates, administrative work like updating, and using SNIFF+'s Make Support on the target machine are much faster.

## Preparation

### Make Command for Remote Compiling

#### For existing projects

Where you set your Make command depends on the following:

- Only if the Make command is the same for all target platforms, can you specify it in the Project Attributes dialog. The information is stored in the Project Description File (PDF), which overwrites the Make command specified in the Platform Settings.
- If you want to compile on target machines that require different Make commands, you specify these in your Platform Settings and therefore the Make command field in the Project Attributes dialog must be empty.

#### For new projects

- We recommend that you leave the **Make command** field empty in the Build Options view of the Project Attributes dialog. This allows you the flexibility of compiling on target machines that require different Make commands. The same conditions for existing projects apply for new projects. See also [For existing projects](#) above.

## Password and rhost settings

For remote compiling and debugging, make sure that the name of your host machine is specified in one of the following files:

- `.rhosts` file. This file is in your home directory.
- `/etc/hosts.equiv`. Since this file applies to the entire system and contains the names of all hosts that are allowed or denied access to a remote host, it is checked first.

### Caution

Generally avoid any setting that requires manual input when logging in to a remote host.

## When is the default platform setting used

The Default Platform setting specified in the Platform view of the Preferences is used when:

- the platform setting in the working environment where you will open your project is set to `<default>`.
- you open absolute projects.

### Note

You can only remote compile and debug absolute projects if both your host machine and target machine are Unix machines and when the path to the project is the same from both machines.

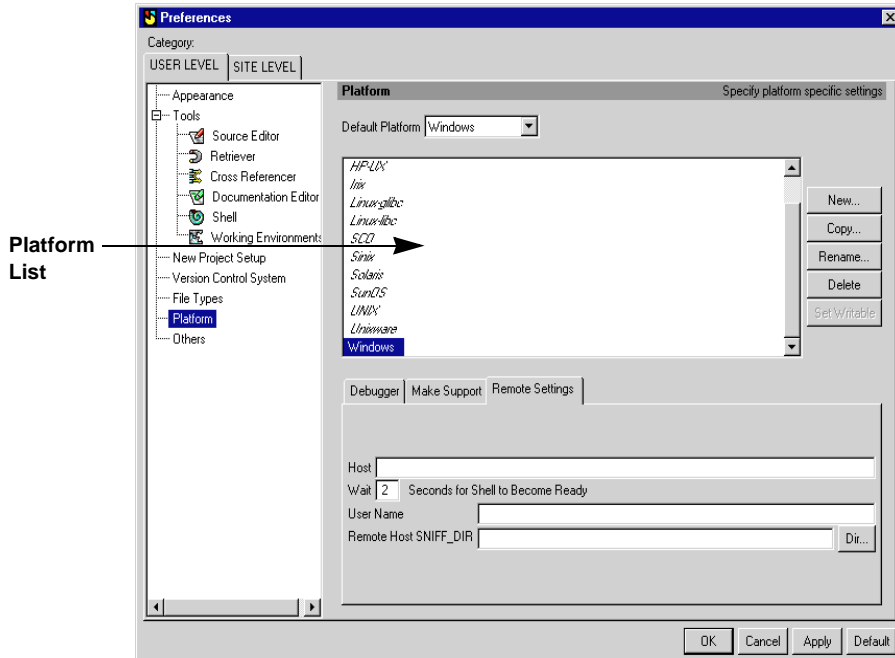
## Setting up remote compile and debug

### Specifying the platform settings

To compile and debug remotely you must specify the platform specific settings in the Preferences. To do so:

- Choose **Tools > Preferences....**

The Preferences tool opens.



### Specific Settings for remote compile and debug

1. In the Platform List, select a target platform or create a new one.
2. Select the Remote Settings tab.
3. In the **Host** field, enter the name of the target machine.
4. By default, the number of seconds reserved for Remote Shell Script Execution is specified. If the scripts need more time to execute, increase the number of seconds in the **Wait \_ Seconds for Shell to Become Ready** field or vice-versa.

5. By default, the local user name is used for remote connections. If you will login to the target machine using a different user name, enter this user name in the **User Name** field.

#### Note

Entries in the **User Name** field are case sensitive.

6. If you are using SNIFF+'s Make Support, in the **Remote Host SNIFF\_DIR** field, enter the directory where SNIFF+ is installed on the target machine.

## General settings for compiling and debugging

1. Select the Debugger tab.
2. In the Debugger view, choose a debugger adaptor from the **Adaptor** drop-down.  
SNIFF+ automatically fills in the **Executable** and **Prompt** fields. Modify these fields if necessary.
3. Select the Make Support tab.
4. The SNIFF+ platform settings, the **Make command** and the **Platform Makefile** are already defined. If you use a different Make system and/or Platform Makefile, enter the appropriate settings in the **Make Command** field and/or **Platform Makefile** field. For new platforms, enter the appropriate platform settings.
5. Press **OK** to apply the settings and to close the Preferences.

## Specifying a different remote Shell executable

By default, SNIFF+ is configured to use `rlogin` for remote compilation.

If you want to use a different remote Shell executable on Unix:

1. In the Preferences, under the Tools node select the Shell view.
2. In the **Local Executable for Remote Shell** field, enter the name of the Shell executable.
3. Press **OK** to close the Preferences.

#### On Windows

We do not recommend Windows users changing the `rlogin.exe` entry in the **Local Executable for Remote Shell** field because this `rlogin` entry has been especially adapted for remote compile and debug.

## Assigning platforms to SNIFF+ working environments

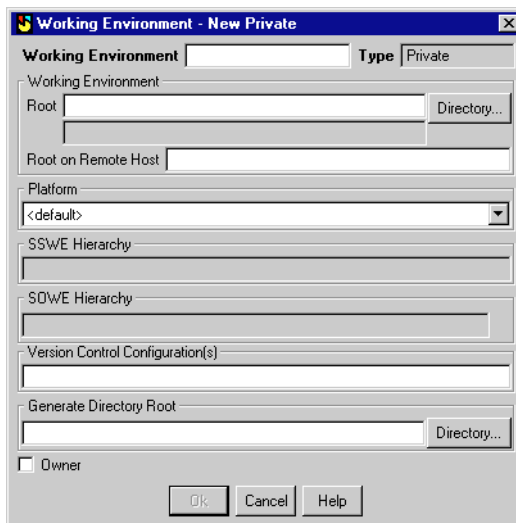
You can assign platforms to *Shared Object Working Environments (SOWEs)* and to *Private Working Environments (PWEs)*, because Make can only be started in these working environments.

- Choose **Tools > Working Environments**.

The Working Environments tool opens.

### Assigning a platform to a new working environment

1. In the Working Environments tool, select the *Shared Source Working Environment (SSWE)* or the root node if you want to create a PWE.
2. Choose **Edit > New Shared Object** or **Edit > New Private**. A dialog opens.



3. In the **Working Environment** field, enter the name of the working environment.
4. In the **Root** field, specify the root of the working environment as you would access it from the host machine.

5. In the **Root on Remote Host** field, specify the root of the working environment as you would access it from the target machine.

**Caution**

Do not use any environment variables or any shell metacharacters to specify the path name.

In a Unix — Unix work situation, you can create symbolic links to access the working environment root directory so that it looks the same from both machines. In this case, leave the **Root on Remote Host** field blank.

**Note**

**Root on Remote Host** is disabled when you don't have a `SNiFF_CROSS` or a `SNiFF_EVAL` license, please refer to [Licensing — page 208](#).

6. In the **Platform** drop down, select the remote platform.
7. Press **Ok** to close the New Private / New Shared Object dialog.
8. In the Working Environments tool, choose **File > Save** to save the changes to the Working Environments.

## Assigning a platform to an existing working environment

You can assign a platform to your Private Working Environment and to the Shared Object Working Environment.

1. Select a working environment and choose **Context menu > Modify...**

The Modify dialog appears.

2. In the **Root** field of the Modify dialog, specify the root of the working environment as you would access it from the machine that you are currently working on.
3. In the **Root on Remote Host** field, specify the root of the working environment as you would access it from the target machine.

### Caution

Don't use any environment variables or any shell metacharacters to specify the path name.

In a Unix — Unix work situation, you can create symbolic links to access the working environment root directory so that it looks the same from both machines. In this case, leave the **Root on Remote Host** field blank.

### Note

**Root on Remote Host** is disabled when you don't have a `SNiFF_CROSS` or a `SNiFF_EVAL` license, please refer to [Licensing — page 208](#)

4. In the **Platform** drop down, select the remote platform.
5. Press **Ok** to close the Modify dialog.
6. In the Working Environments tool, choose **File > Save** to save the changes to the Working Environments.

## Invoking Remote Compile and Debug

Now that you have set all the relevant settings for remote compile and debug, you can go ahead and open your project, either in your Private Working Environment or in the Shared Object Working Environment. Before compiling, remember to update Makefiles. When you compile and debug the project, SNiFF+ uses the information in the Platform view of the Preferences. This allows you to compile and debug remotely using the commands in the Project Editor's **Target** menu.



# Part VII

## Cross Reference Subsystems



## Cross Reference Information

---

### Introduction

Cross reference (X-Ref) information describes where a certain construct is used (or is referred to) and which other constructs it in turn uses (or refers to).

SNiFF+ provides two alternative X-Ref subsystems for managing cross reference information, either *RAM-based cross referencing*, or *database-driven cross referencing* (available as of SNiFF+ 3.2, and also known as Rapid Reference Technology™). This chapter aims at describing and contrasting these two systems (referred to in the following as *RAM-based* and *DB-driven*).

In a nutshell, the RAM-based solution loads a set of indexed files to memory to resolve X-Ref queries, whereas the DB-driven solution directly accesses a database.

However, because X-Ref information is stored at Working Environment level under DB-driven cross referencing, this technology has a number of implications for Working Environments administration in team projects.

Which cross reference engine is used therefore determines not only how X-Ref information is generated, stored, and subsequently accessed to resolve queries, but also influences Project and Working Environment administration. As well as performance and scalability.

Furthermore, the procedure for managing cross reference information differs between C/C++ and other languages, e.g. Java.

All these factors are outlined in the following.

## Overview

Each of the points introduced in this overview are discussed in more detail later on.

As mentioned already, the process of generating cross reference information in SNIFF+ differs depending on the programming language being parsed.

In the following, we distinguish between **C/C++** and **Java**, whereby the procedure for **Java** is analogous for all other **non-C/C++** languages supported in SNIFF+.

The following topics are discussed

- [Extracting symbol information — page 221](#)

How and when parsing is triggered for extracting symbol information is independent of programming language and X-Ref technology, but what happens during parsing is not.

- [How the X-Ref subsystems work — page 221](#)

The available X-Ref technologies (RAM-based and DB-driven) are contrasted in the context of C/C++ and of Java/other languages, respectively. This should give you an understanding of the internal procedures involved.

- [Location of generated X-Ref information — page 225](#)

Where and how X-Ref information is stored depends solely on the X-Ref technology used (not on the programming language).

- [Working Environments and cross referencing — page 226](#)

Because the X-Ref databases are maintained at Working Environment level and shared across Working Environment boundaries, there are a number of points to be aware of. These include database access control (locking) and database maintenance.

- [Selecting your preferred X-Ref technology — page 229](#)

How to select your preferred X-Ref technology is described at the end of the chapter — by which time you should have a fair understanding of the issues involved.

## Extracting symbol information

Initially, all symbol information is extracted from source code files by the appropriate language-specific parser. Parsing is triggered

- during project setup
- when modified source files are saved
- when files are checked in/out
- by the user (menu command: **Project > Force Reparse**)
- when projects are opened if the symbol table is not available

All extracted structural symbol information is written to the symbol table (\* .symtab files).

How cross reference information is generated depends both on the technology used to drive cross referencing, as well as on the language being parsed. This is described in the following section.

## How the X-Ref subsystems work

This section uses two tables to describe how the X-Ref subsystems used by SNIFF+ work.

RAM-based and DB-driven cross reference management is contrasted, first in the context of C/C++, then of Java/all other non-C/C++ languages, respectively.

Note that the tables **do not describe user interaction**, each column simply outlines what SNIFF+ does at each stage. Each table is followed by a summary of the immediate implications.

## RAM-based versus DB-driven cross referencing in C/C++

The following table is divided into three rows, each representing a stage in the cross reference management cycle. These stages are

### ■ Parsing

How parsing is triggered is described under [Extracting symbol information — page 221](#). The table includes only X-Ref related information.

### ■ Generating

While the first step in the X-Ref process takes place during parsing, the following steps can be triggered either on demand immediately after project setup, or using the menu command: **Project > Update Cross Ref Info**. If the procedure is not invoked by the user, it is automatically started when the first Referred-By query is issued after a (re-)parse.

### ■ Querying

Note that, if source files have been modified, or if structural changes have been made to projects, both the “Generating” and “Querying” stages are triggered to incrementally update X-Ref information when you issue your first subsequent Referred-By query.

Stage	RAM-Based	DB-Driven
<b>Parsing</b>	<ul style="list-style-type: none"> <li>■ Generate temporary lexical analysis files (*.lex files), one for each source file parsed. These files contain fuzzy descriptions of what is referenced where, and are saved to disk to be used as input for later generation of cross reference information.</li> </ul>	<ul style="list-style-type: none"> <li>■ Same as RAM-based.</li> </ul>
<b>Generating</b>	<ul style="list-style-type: none"> <li>■ Interpret the lex files in the context of the symbol table.</li> <li>■ Resolve the fuzzy descriptions in lex files and write information to actual cross reference files (*.ref files); these are needed for Referred-By queries.</li> <li>■ Create/update indexes (*.index files). A global index is written per project.</li> <li>■ Save ref and index files to disk.</li> <li>■ Delete lex files.</li> </ul>	<ul style="list-style-type: none"> <li>■ Interpret the lex files in the context of the symbol table.</li> <li>■ Resolve the fuzzy descriptions in lex files in memory.</li> <li>■ Interpret and save resolved X-Ref information to database.</li> <li>■ Delete lex files.</li> </ul>

Stage	RAM-Based	DB-Driven
<b>Querying</b>	<ul style="list-style-type: none"> <li>■ First query after file/project modifications: apply “Generating” stage (above) to increment.</li> <li>■ Load index and necessary <code>ref</code> files to memory.</li> <li>■ Resolve query and display results.</li> </ul>	<ul style="list-style-type: none"> <li>■ First query after file/project modifications: apply “Generating” stage (above) to increment.</li> <li>■ Query database and display results.</li> </ul>

## Summary

- For C/C++ files, there is no difference in the parsing stage.  
Differences are language-specific — compare following table.
- During the generation stage, procedures differ.  
Under the RAM-based system, `ref` files are written to disk and indexes have to be created and written to disk.  
Under the DB-driven system, no `ref` or `index` files need be written to disk, but the `lex` file information needs to be interpreted so that the database can handle it, and then this has to be written to the database.  
**Result:** When all the X-Ref information has to be generated for large software systems (project setup / forced reparse), the generation time is noticeably longer under the DB-driven system. For incremental updates (after file/project modifications, regular/nightly updates), the generation time differences will generally be negligible.
- Major differences are apparent at the querying stage.  
Under the RAM-based system, the index(es) and all necessary `ref` files have to be loaded. Depending on the size of the project and nature of the query (and the what is queried in which order), this can be time and memory consuming.  
Under the DB-driven system, the database is directly queried, resulting in constantly fast queries and negligible additional resource consumption — regardless of how complex the query. The first Referred-By query after file/project modifications takes longer under both technologies because incremental differences have to be processed before queries are resolved.

## RAM-based versus DB-driven cross referencing in Java

The following table is divided into two rows, each representing a stage in the cross reference cycle. These stages are

### ■ Parsing and Generating

How parsing is triggered is described under [Extracting symbol information — page 221](#). The table includes only X-Ref related information. For Java and other non-C/C++ languages, all X-Ref information is generated during parsing. This means that, to force an update of X-Ref information, you need to use the **Force Reparse** menu command (the **Update Cross Reference Info** command has **no** effect in non-C/C++ languages).

### ■ Querying

By the time a query is issued, all X-Ref information has already been generated.

Stage	RAM-Based	DB-Driven
<b>Parsing and Generating</b>	<ul style="list-style-type: none"> <li>■ Generate cross reference files (*.ref files).</li> <li>■ Create/update ref file indexes (*.index files). A global index is written per project.</li> <li>■ Save ref and index files to disk.</li> </ul>	<ul style="list-style-type: none"> <li>■ Interpret X-Ref information in memory.</li> <li>■ Write X-Ref information directly to database.</li> </ul>
<b>Querying</b>	<ul style="list-style-type: none"> <li>■ Load index(es) and necessary ref files to memory.</li> <li>■ Resolve query and display results.</li> </ul>	<ul style="list-style-type: none"> <li>■ Query database and display results.</li> </ul>

## Summary

- Under the RAM-based system, ref files are written to disk and indexes have to be created and written to disk during Parsing.

Under the DB-driven system, no ref or index files need be written to disk, but the X-Ref information extracted by the parser has to be interpreted so that the database can handle it, and then the information has to be written to the database.

**Result:** When all the X-Ref information has to be generated for large software systems (project setup / forced reparse), the generation time is noticeably longer under the DB-driven system. For incremental updates (after file/project modifications, regular/nightly updates), the generation time differences will generally be negligible.



- Major differences are apparent in the querying stage.

Under the RAM-based system, the index(es) and all necessary `ref` files have to be loaded. Depending on the size of the project and nature of the query (and the what is queried in which order), this can be time and memory consuming.

Under the DB-driven system, the database is directly queried, resulting in constantly fast queries and negligible additional resource consumption — regardless of how complex the query.

## Location of generated X-Ref information

Where and how X-Ref information is stored depends solely on the X-Ref technology used (not on the programming language).

Project Type	RAM-Based X-Ref	DB-Driven X-Ref
<b>Shared</b>	<p><b>Default:</b> In each Project root directory.</p> <p><b>Options:</b> Can be stored anywhere in your file system.</p>	<p>Always in a subdirectory (<code>.sniffdb</code>) at the root of each Working Environment.</p> <p>Note that only one database is created per Working Environment, and all database files are stored in <code>.sniffdb</code>.</p>
<b>Absolute (Browsing-Only)</b>	<p><b>Default:</b> In each Project root directory.</p> <p><b>Options:</b> Can be stored anywhere in your file system.</p>	<p>If absolute projects are opened in a Working Environment, the X-Ref info is generated to the (existing) database for that Working Environment (under <code>.sniffdb</code>).</p> <p>If opened outside of a Working Environment, the X-Ref information is written to a database (under <code>.sniffdb</code>) in a <i>Default Working Environment for Absolute Projects</i>, which can be set in your Preferences.</p>

- Notice that, under DB-driven cross referencing, X-Ref information is stored at Working Environment level. The implications of this are discussed in the following section.

## Working Environments and cross referencing

After a brief note on RAM-based cross referencing, this section concentrates on issues relating to [DB-driven cross referencing and Working Environments — page 226](#).

- Regardless of which X-Ref technology you use, **mixing** X-Ref technologies within a Working Environment hierarchy does **not** work. This is because different files are used by each system, as shown under [How the X-Ref subsystems work — page 221](#).

### RAM-based cross referencing

Under RAM-based cross referencing, there are no special implications as far as Working Environments are concerned. This is because the X-Ref information files are maintained at project level.

When files are checked out/in they are reparsed, and the procedure for generating X-Ref information as described for this technology under [How the X-Ref subsystems work — page 221](#) is set in motion when queries are issued. As a result, multiple access is not a problem under RAM-based cross referencing.

### DB-driven cross referencing and Working Environments

X-Ref databases are maintained at Working Environment level. Although only one X-Ref database is created per Working Environment, information sharing has to work across Working Environment boundaries. This implies that some sort of mechanism for controlling **Read** and **Write** access to database files has to be implemented.

#### X-Ref database access control

Note that “accessing an X-Ref database” effectively means “opening Projects (with symbol information) in a given Working Environment, and thereby accessing the X-Ref database for that Working Environment and higher-level (Shared) Working Environments”.

In the following, a number of points relating to database access control are listed, together with examples as they affect team development.

- **Write** access is **absolutely exclusive within a Working Environment**.

This means that if an X-Ref database is opened for writing, no other SNIFF+ session can access the database (neither for reading, nor for writing). In other words, Projects cannot be opened with symbol information in that Working Environment by **other** SNIFF+ sessions.

#### Example:

For day-to-day development work, you will generally open Projects (with symbol information) in your Private Working Environment. Because you will want any changes you make to be reflected also in the cross-reference information, you need **Write** access to the X-Ref database for that Working Environment. Which is not a problem. A problem will only arise if and when **another** SNIFF+ session tries to open Projects (with symbols) in that same Working Environment.

- **Write** access is **hierarchically absolutely exclusive**.

**Write** access in a higher-level Working Environment (e.g. SSWE) also blocks all (**Read** and **Write**) X-Ref database access in all lower-level Working Environments (e.g. PWEs). To put it another way, if Projects are opened in a (Shared) Working Environment with **Write** access, no lower-level (Private) Working Environment can then access the database system of the hierarchy in any way. This means that all Working Environments lower down in the hierarchy are blocked in terms of symbol information.

**Example:**

What happens when the SSWE needs to be updated? Updating a Working Environment means, among other things, **Write** access to its X-Ref database. And **Write** access is “hierarchically absolutely exclusive”, that is, all Projects open (with symbol information) in lower-level Working Environments have to be closed. Please refer also to [Write access to an X-Ref database that is already opened in Read mode. — page 227](#).

- If a higher-level Working Environment has a **Write** lock, it is only possible to open Projects **without** symbol information (that is, without any X-Ref database access) in lower-level Working Environments.
- Multiple **Read** access is possible.

Any number of SNIFF+ sessions can access a given Working Environment's X-Ref database in **Read** mode — as long as it is **not** already being accessed in **Write** mode (see earlier).

**Example:**

Generally, each user will have Projects opened (with symbols) in Private Working Environments, that is, with **Write** access to the PWE's X-Ref database. And all users will have **Read** access to the Shared Source Working Environment's X-Ref database.

- **Write** access to an X-Ref database that is already opened in **Read** mode.

Not only is **Write** access hierarchically absolutely exclusive (see earlier), it also allows you to take priority over existing **Read** access locks. That is, if an X-Ref database is open in **Read** mode by other SNIFF+ sessions, and you attempt a **Write** access, you are given two options, a **Remote Shutdown** option and a **Break Lock** option.

The **Remote Shutdown** option saves and closes all Projects in other SNIFF+ sessions with **Read** access to the database in question. Note that **Remote Shutdown** starts execution after a 30 second delay.

The **Break Lock** breaks existing locks. This option is **not** recommended as inconsistencies are inevitable.

**Example:**

An unattended nightly update of the SSWE is started, and **Write** access is therefore required for the SSWE's X-Ref database. But one or more team members still have active SNIFF+ sessions with Projects opened in Private Working Environments that access the SSWE. These open sessions therefore have **Read** access to the SSWE X-Ref database. Rather than abort the update, all open files and Projects can be saved and closed using the **Remote Shutdown** option. The update then proceeds normally.

- Accessing an X-Ref database that is already opened in **Write** mode?

You can't. Neither in **Read** mode, nor in **Write** mode. At least not without breaking the lock on the database. This is **not** to be recommended, and should only be used in exceptional circumstances.

**Example:**

None. Do not use the **Break Lock** option, unless you are certain that this lock is invalid. Even if SNIFF+ terminates unexpectedly, exit handlers should be able to remove existing locks.

## Where locking information is maintained

Locking information is maintained in a subdirectory of your **Working Environments Configuration Directory** (you set this in your Preferences) called `.snifflock`. A subdirectory is created for each lockable Working Environment, this subdirectory takes the name of the Working Environment for which the locking information is maintained. All current **Read** access locks are stored directly in this subdirectory. Each of these subdirectories can have another subdirectory called `lock`. Any existing **Write** access lock on the Working Environment is stored in the `lock` subdirectory.

Lock file names have the following syntax:

```
lock.machinename.PID
```

If an attempt is made to access locked databases, files are created called

```
try.machinename.PID
```

These files are used by SNIFF+ to identify requests for X-Ref database access.

## Maintenance of X-Ref databases

If, for some reason, databases become corrupt, SNIFF+ may be able to repair them. If the databases are irreparably damaged, they can be recreated from scratch by deleting the corrupt database files (see also [Location of generated X-Ref information — page 225](#)) and issuing a **Force Reparse** command.

The information in the X-Ref database is cumulative, that is X-Ref information for every project ever opened in a given Working Environment is stored in that Working Environment's database. The X-Ref information for any given project will only be removed from the database when the Project is deleted using the **Delete Project** command.

## Synchronizing Working Environments

The correct order for updating/synchronizing Working Environments should always be maintained to avoid inconsistencies. This is

- Shared Source Working Environment (SSWE)
- Shared Object Working Environment (SOWE)
- Private Working Environment (PWE)

Again: **Whenever** a higher-level Working Environment (e.g. SSWE) is updated, **all** lower-level Working Environments (e.g. PWEs) accessing it should (must) be subsequently updated.

This holds especially for **DB-driven** cross referencing. Apart from all other possible inconsistencies caused by out-of-sync Working Environments, cross reference information will also be inconsistent because, for performance reasons, internal database objects are shared among Working Environments.

## Selecting your preferred X-Ref technology

- You set your preferred cross reference system at user-level, that is, in your SNIFF+ Preferences.
- If you decide to use the **DB-driven** X-Ref technology, be aware of the implications as described under [Working Environments and cross referencing — page 226](#)
- If you decide to use the **RAM-based** X-Ref technology, there is not much else you have to take into consideration. The only thing you need to remember is that you can **not** mix different X-Ref technologies within a Working Environment hierarchy.



# Part VIII

## Editor Integrations





# Emacs Integration

---

## Introduction

You can use the Emacs editor for editing source code in SNIFF+. This chapter covers the basics of the integration.

---

**Note**

Emacs is not supplied as part of the product package.

## This chapter covers the following topics

- Integrate SNIFF+ with Emacs
- Work with the integration

## Assumptions made in this chapter

- You are an experienced Emacs user

## Integration features

The following integration features are available:

- Emacs can be used for all editing requests
- SNIFF+ recognizes and updates all browsers when a file is saved in Emacs
- SNIFF+ commands can be issued directly from Emacs

## How the Emacs integration works

Emacs need not be changed to work in the SNIFF+ environment. An Emacs-Lisp configuration file supplied with the SNIFF+ distribution tells Emacs how to:

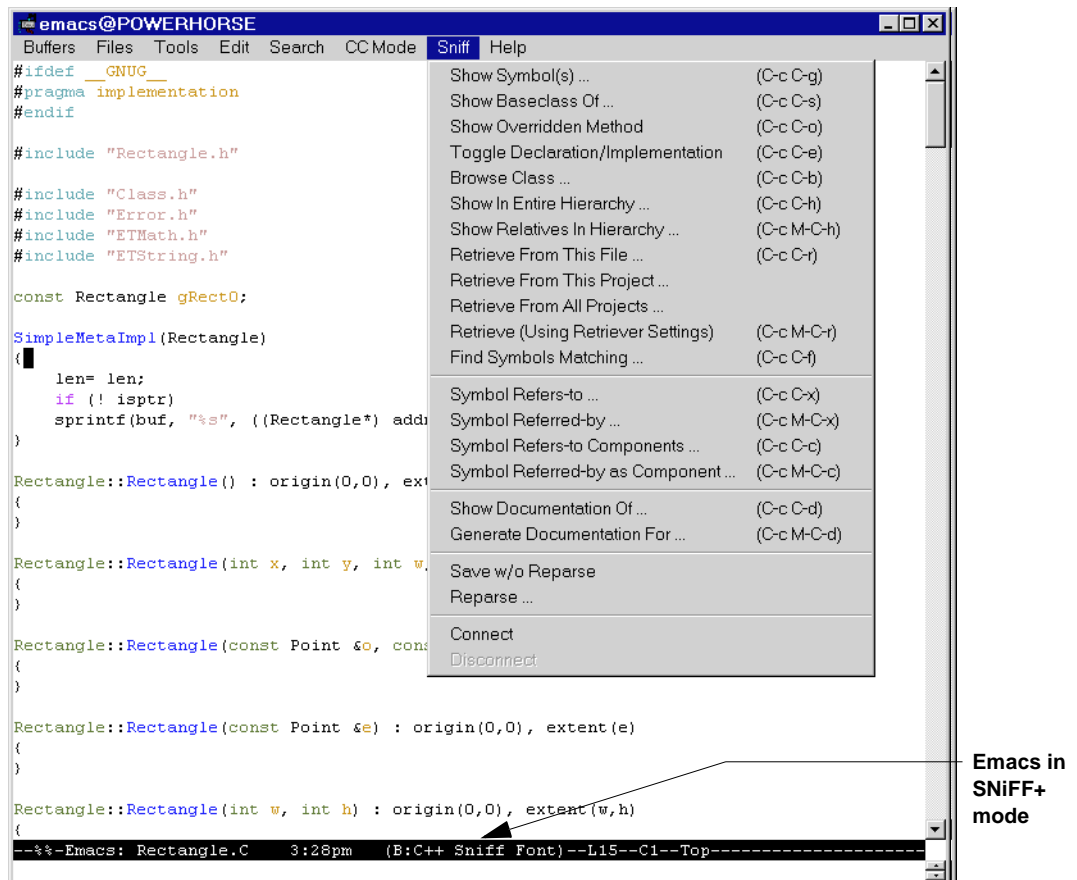
- define a SNIFF+ mode
- communicate with SNIFF+
- define keyboard bindings and a pull-down menu for available SNIFF+ commands

This file is called `sniff-mode.el` and is located in your `$SNIFF_DIR/config` directory. To use the file, simply load it into Emacs. Once the file is loaded, a new SNIFF+ mode is available in Emacs. Then, execute the function `sniff-connect` to connect Emacs to your current SNIFF+ session.

You also have to tell SNIFF+ to use Emacs as the main editor. When you do so, SNIFF+ uses Emacs for displaying and editing source code.

## User interface examples

The following figure shows Emacs connected to SNIFF+:



## Integrating Emacs

### Prerequisites

- SNIFF+ installed at your site
- GNU Emacs (version 19 or later) or XEmacs (formerly Lucid Emacs) installed at your site
- The `sniff-mode.el` file (part of the SNIFF+ package)

### Setting Emacs as your preferred editor

1. Start SNIFF+.
2. Choose **Tools > Preferences** in any open SNIFF+ tool.

## In the Preferences dialog

1. Select the **Tools > Source Editor** node.
2. From the **Current Editor** drop-down, choose **Emacs/Vim**.
3. Press **Ok**.

SNiFF+ will now use Emacs for all editing requests.

## Switching Emacs to SNiFF+ mode

- Add the following line to your `.emacs` file

```
(load "$SNIFF_DIR/config/sniff-mode")
```

This causes Emacs to load the `sniff-mode.el` configuration file at start-up.

### Note

You can avoid the path specification by copying `sniff-mode.el` to the directory for site-wide Emacs-Lisp files:

```
cp $SNIFF_DIR/config/sniff-mode.el /usr/local/  
lib/emacs/site-lisp
```

After you have done this, your `.emacs` file entry would look like this:

```
(load "sniff-mode")
```

## Connecting Emacs to SNiFF+

- To connect Emacs to a running SNiFF+ session, type:

```
M-x sniff-connect
```

### Note

You could also add this command to your `.emacs` file. Then, your `.emacs` would look like this:

```
(load "$SNIFF_DIR/config/sniff-mode")  
(sniff-connect)
```

Note that you can have only one Emacs-SNiFF+ connection active at a given time.

4. To force Emacs to disconnect from SNiFF+, type:

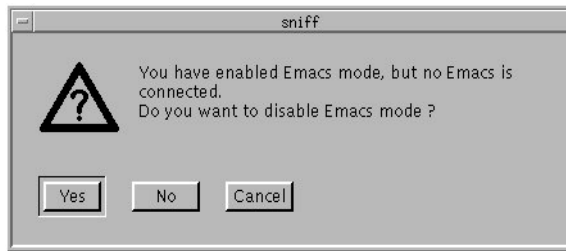
```
M-x sniff-disconnect
```

Note that Emacs automatically disconnects when you end a current SNiFF+ session.

## Working with Emacs and SNIFF+

Once a connection between SNIFF+ and Emacs is established, SNIFF+ uses Emacs for all requests to show or edit source code. Emacs also can send queries to SNIFF+.

When you issue an editing request in SNIFF+ and Emacs is not connected to SNIFF+, you are asked whether you want to switch off the Emacs mode and use the Source Editor.



## Positioning Emacs from SNIFF+

The Emacs integration offers many of the navigation features available in the Source Editor. For example, by double-clicking on a symbol in any SNIFF+ browser, Emacs loads the corresponding source file and positions the cursor at the appropriate location.

## Changing key bindings and Sniff menu entries

The Emacs-SNIFF+ key bindings and the **Sniff** menu are defined in:

```
$SNIFF_DIR/config/sniff-mode.el
```

You can change the SNIFF+ key bindings as for any other Emacs key bindings. The same is true for the **Sniff** menu.

## Configuring symbol highlighting

Emacs can use different fonts. SNIFF+ uses this feature to highlight symbols in source code. As a result, Emacs is able to mimic the Source Editor's symbol highlighting behavior.

- To enable symbol highlighting, enter the following line in your `.emacs` file:
 

```
(setq sniff-want-fonts 1)
```
- To switch off symbol highlighting, enter the following line in your `.emacs` file:
 

```
(setq sniff-want-fonts nil)
```

The default font table for the highlighting is defined in the `sniff-mode.el` file. You can change any value in the default font table by setting variables in your `.emacs` file after `sniff-mode.el` is loaded. For example, the following line in your `.emacs` file would tell Emacs to use bold typeface for constants:

```
(aset sniff-font-table 10 'bold)
```

Please see the `sniff-mode.el` file for a full description of table entries.

### Note

If you have enabled symbol highlighting, but the feature doesn't work, this means that your Emacs might use a font that does not supply the necessary typefaces. The courier font family normally supplies all necessary typefaces. To use this family, add the following X resource to your `.Xdefaults` file:

```
emacs.font: -*-courier-medium-r-normal--*-120-75-75-*-*-*-*
```

## Using the Sniff menu

Most of the commands described in [Command Reference — page 239](#) are available in the **Sniff** menu in Emacs. The following illustration shows the menu:

Show Symbol(s) ...	(C-c C-g)
Show Baseclass Of ...	(C-c C-s)
Show Overridden Method	(C-c C-o)
Toggle Declaration/Implementation	(C-c C-e)
Browse Class ...	(C-c C-b)
Show In Entire Hierarchy ...	(C-c C-h)
Show Relatives In Hierarchy ...	(C-c M-C-h)
Retrieve From This File ...	(C-c C-r)
Retrieve From This Project ...	
Retrieve From All Projects ...	
Retrieve (Using Retriever Settings)	(C-c M-C-r)
Find Symbols Matching ...	(C-c C-f)
Symbol Refers-to ...	(C-c C-x)
Symbol Referred-by ...	(C-c M-C-x)
Symbol Refers-to Components ...	(C-c C-c)
Symbol Referred-by as Component ...	(C-c M-C-c)
Show Documentation Of ...	(C-c C-d)
Generate Documentation For ...	(C-c M-C-d)
Save w/o Reparse	
Reparse ...	
Connect	
Disconnect	

## Switching a non-SNIFF+ buffer to SNIFF+ mode

When a file is loaded in Emacs from SNIFF+, this buffer is automatically in SNIFF+ mode. When you load a file manually (with the Emacs **Load file** command), you can switch the buffer to SNIFF+ mode with the following command:

```
M-x sniff-mode
```

After the command is executed, all SNIFF+ key bindings are available and symbols are highlighted.

## Command Reference

All of the SNIFF+ commands that are important when editing source code are also available in Emacs. To accomplish this, a few keys have been bound to functions that communicate with SNIFF+. The functions and the key bindings are defined in:

```
$SNIFF_DIR/config/sniff-mode.el
```

Generally all commands (except toggle declaration/definition, edit overridden method and find next match) read the argument string (or symbol) from the minibuffer, whereby the string around the point is inserted as a default. (In Emacs nomenclature, the cursor position is called *point*.)

The following commands and bindings are available:

SNIFF+ command	What happens	Emacs key binding
<b>Show Symbol(s)...</b>	Shows the declaration or implementation of <i>symbol</i> . If <i>symbol</i> is ambiguous, a dialog opens with a list of valid alternatives	C-c C-g sniff-goto-symbol
<b>Show Baseclass Of...</b>	Shows the declaration of the base class of the currently selected <i>class</i> . This entry is enabled when the cursor is positioned in the scope of a class that has a base class.	C-c C-s sniff-superclass
<b>Show Overridden Method</b>	Shows the overridden <i>method</i> of the closest base class that defines <i>method</i> into a Source Editor.	C-c C-o sniff-overridden
<b>Toggle Declaration/Implementation</b>	Shows the declaration/implementation of <i>method</i>	C-c C-e sniff-toggle
<b>Browse Class...</b>	Shows the members of <i>class</i> in the Class Browser	C-c C-b sniff-browse-class
<b>Show In Entire Hierarchy...</b>	Opens a Hierarchy Browser and loads the entire class graph. <i>class</i> is highlighted in the Hierarchy Browser	C-c C-h sniff-hierarchy
<b>Show Relatives In Hierarchy...</b>	Opens a Hierarchy Browser and loads the graph of the base and derived classes. <i>class</i> is highlighted in the Hierarchy Browser	C-c M-C-h sniff-restr-hier

SNiFF+ command (cont.)	What happens	Emacs key binding
<b>Retrieve From This File...</b>	Searches for a string in the current file using the Retriever.	C-c C-r sniff-retrieve
<b>Retrieve From This Project...</b>	Searches for a string in the current project using the Retriever.	sniff-retrieve-proj
<b>Retrieve From All Projects...</b>	Searches for a string in all projects using the Retriever.	sniff-retrieve-allprojs
<b>Retrieve (Using Retriever Settings)</b>	Searches for a string using the current Retriever settings.	C-c M-C-r sniff-retrieve-next
<b>Find Symbols Matching...</b>	Opens a Symbol Browser to search for symbols that match <i>selection as a whole word</i> .	C-c C-f sniff-find-symbol
<b>Symbol Refers-to...</b>	Opens a Cross Referencer and starts a Refers-To query on <i>symbol</i> . The settings of this Cross Referencer's Xref Filter are used for the query parameters.	C-c C-x sniff-xref-to
<b>Symbol Referred-by..</b>	Opens a Cross Referencer and starts a refers-by query on <i>symbol</i> . The settings of this Cross Referencer's Xref Filter are used for the query parameters.	C-c M-C-x sniff-xref-by
<b>Symbol Refers-to Components...</b>	Opens a Cross Referencer and starts a query for showing all symbols (classes and structures) that are components of <i>symbol</i> . If the current selection is a member of a class/structure, the class/structure is taken for this query.	C-c C-c sniff-xref-has



<b>SNiFF+ command (cont.)</b>	<b>What happens</b>	<b>Emacs key binding</b>
<b>Symbol Referred-by as Component...</b>	Opens a Cross Referencer and starts a query for showing all symbols that have <i>symbol</i> as a component. Note that you can also query primitive C data types with this command.	C-c M-C-c sniff-xref-used by
<b>Show Documentation Of...</b>	Opens a Documentation Editor and positions it to the documentation of <i>symbol</i> . An alert message appears if no documentation file exists for either the entire file or for <i>symbol</i> . You then have the option of creating a documentation file	C-c C-d sniff-show-docu
<b>Generate Documentation For...</b>	If the cursor is positioned on a symbol, you are asked if you would like to generate documentation for the symbol or for the file, if not documentation is generated for the file.	C-c M-C-d sniff-gen-docu
<b>Save w/o Reparse</b>	SNiFF+ saves the current file without reparsing it.	sniff-writebuffer
<b>Reparse...</b>	SNiFF+ reparses the current file and updates the Symbol Table.	sniff-reparse
<b>Connect</b>	Establishes connection with SNiFF+. Make sure that SNiFF+ is in Emacs mode.	sniff-connect
<b>Disconnect</b>	Disconnects from SNiFF+. You can reconnect at any time with :sniff connect.	sniff-disconnect



# Vim Integration

---

## Introduction

You can use the Vim editor for editing source code in SNIFF+. This chapter covers the basics of the integration.

### This chapter covers the following topics

- Integrate SNIFF+ with Vim
- Work with the integration

### Assumptions made in this chapter

- You are an experienced vi user

## Integration Features

This integration is based on Vim 5.x, which is almost fully compatible to vi but provides a variety of additional features (e.g., multiple buffers, syntax highlighting, etc.). The following integration features are available:

- Vim can be used for all editing requests.
- SNIFF+ recognizes and updates all browsers when a file is saved in Vim.
- SNIFF+ commands can be issued directly from Vim.

## How the Vim integration works

Vim must be compiled with the `sniff` option enabled to work in the SNIFF+ environment. The SNIFF+ product package includes a pre-compiled version with this option enabled.

Start Vim in an environment where the `SNIFF_DIR` environment variable is correctly set.

Then, enter `:sniff connect` to connect Vim to your current SNIFF+ session. You also need to set Vim as your preferred editor.

### Note

You can have only one Vim-SNIFF+ connection active at a given time.

## Integrating Vim

### Prerequisites

- SNIFF+ installed at your site.
- Vim 5.x with sniff support installed at your site.
- the `sniff.vim` file (part of the SNIFF+ package).

### Setting Vim as your preferred editor

1. Start SNIFF+.
2. Choose **Tools > Preferences** in any open SNIFF+ tool.

#### In the Preferences dialog

1. Select the **Tools > Source Editor** node.
2. From the **Current Editor** drop-down, choose **Emacs/Vim**.
3. Press **Ok**.

SNIFF+ will now use Vim for all editing requests.

### Connecting Vim to SNIFF+

- Set the environment variable `$VIM` to `$SNIFF_DIR/config/vim`.  
**IMPORTANT:** This environment variable must be set to use Vim's online help and syntax highlighting.
- Start Vim.

- Enter the following command in Vim:  
`:sniff connect`  
 The following message appears:  
 Connecting to SNIFF+ ... Done

#### Note

If the above message does not appear, check to see if the above steps (1 to 5) have been done properly.

## Configuring the Vim integration

### Connection at start-up

1. Check if the initialization file with the name `.vimrc` exists in your home directory (`~/`  
`.vimrc`), if not create it.
2. Insert the following line in the `.vimrc` file:  
`sniff connect`  
 Vim connects to SNIFF+ immediately at start-up and will execute the commands contained in the `.vimrc` file at start-up.

### Key mappings

- The key mappings are defined in the `sniff.vim` file. This file is sourced each time you connect to SNIFF+. You can change these mappings to suit your needs. By default Vim looks for this file in the `$SNIFF_DIR/config` directory. However it is possible to define another search path. Insert the following into the `.vimrc` file before the line that contains `sniff connect`:  
`let sniff_mappings = '<where-your-sniff-mappings-are>'`

### Merging from vi to Vim

- If you have a `.exrc` file (for use with your standard vi editor) that you would like to use with Vim, add the following command to the `.vimrc` file  
`source ~/.exrc`
- If you want Vim to have the same 'look and feel' as vi, enter the following command:  
`:set compatible`

## Working with Vim and SNIFF+

Once a connection between SNIFF+ and Vim is established, SNIFF+ uses Vim for all requests to display or edit source code.

## Positioning Vim from SNIFF+

By double-clicking on a symbol in any SNIFF+ browser, Vim loads the corresponding source file and positions the cursor at the appropriate location.

## Buffers in Vim

Vim can hold several buffers at once. To work with buffers, use the following commands:

Command	Description
<code>:ls</code>	Prints a list of currently loaded buffers
<code>:b[uffer] &lt;name&gt;</code> or <code>:b[uffer] &lt;number&gt;</code>	Switches to the buffer specified by the filename or the number (as listed by <code>ls</code> )

For details about buffers in Vim, refer to Vim's online help using the following command:

```
:help buffer
```

## Additional features

### Syntax highlighting

Vim is capable of syntax highlighting for a variety of languages and formats.

To enable syntax highlighting, issue the following command:

```
:syntax on
```

To disable syntax highlighting, issue the following command:

```
:syntax off
```

**IMPORTANT:** Set the environment variable `$VIM` to `$SNIFF_DIR/config/vim`. This environment variable must be set to use Vim's online help and syntax highlighting.

### Online help

Vim provides online documentation for all available commands and features. To view on-line help, enter the following command:

```
:help [keyword]
```

If you would like to know more about the differences between `vi` and Vim, enter the following command in Vim:

```
:help vi_diff
```

**IMPORTANT:** Set the environment variable `$VIM` to `$SNIFF_DIR/config/vim`. This environment variable must be set to use Vim's online help and syntax highlighting.

## Command Reference

All of the SNIFF+ commands that are important when editing source code are also available in Vim. To accomplish this, a new Vim command has been added to communicate with SNIFF+:

```
:sniff request [symbol]
```

For ease of use, key mappings (shortcuts) are predefined. These allow fast queries without typing long command names. Shortcut commands always act on the symbol under the current cursor position.

The key mappings are defined in:

```
$SNIFF_DIR/config/sniff.vim
```

The following commands are available:

Vim Command	Description	Shortcut
:sniff connect	Establishes connection with SNIFF+. Make sure that SNIFF+ is in Vim mode.	sc
:sniff disconnect	Disconnects from SNIFF+. You can reconnect at any time with :sniff connect.	sq
:sniff toggle	Toggles between the implementation and definition of the symbol.	st
:sniff superclass	If the symbol is a class, and the class has at least one superclass, Vim is positioned to the declaration of the superclass. If the class has more than one superclass, a Choose Symbol dialog appears, offering all possible choices.	ss
:sniff overridden	If the cursor is positioned inside a method, Vim positions to the overridden method if it exists.	so
:sniff retrieve-file	Searches for a string in the current file using the Retriever.	srf
:sniff retrieve-project	Searches for a string in the current project using the Retriever.	srp
:sniff retrieve-all-projects	Searches for a string in all projects using the Retriever.	srP

Vim Command	Description	Shortcut
<code>:sniff retrieve-next</code>	Searches for a string using the current Retriever settings.	sR
<code>:sniff goto-symbol</code>	Vim goes to the declaration or implementation of the symbol. If more than one symbol match, a Choose Symbol dialog opens offering all possible choices.	sg
<code>:sniff find-symbol</code>	Loads the symbol into a Symbol Browser.	sf
<code>:sniff browse-class</code>	Loads the class - its interface and hierarchy - into a Class Browser.	sb
<code>:sniff hierarchy</code>	Loads the class under the cursor into a Hierarchy Browser and positions to the class in the full hierarchy.	sh
<code>:sniff restr_hier</code>	Loads the class into a Hierarchy Browser and shows only related classes.	sH
<code>:sniff xref-to</code>	Opens a Cross Referencer and starts a refers-to query on the symbol. If more than one symbol match, a Choose Symbol dialog appears.	sxt
<code>:sniff xref-by</code>	Opens a Cross Referencer and starts a refers-by query on the symbol. If more than one symbol match, a Choose Symbol dialog is opened.	sxb
<code>:sniff xref-has</code>	Opens a Cross Referencer and starts a query for showing all classes and structures that are components of the symbol. The symbol must be a class or structure. If more than one symbol match, a Choose Symbol dialog is opened.	sxh
<code>:sniff xref-used-by</code>	Opens a Cross Referencer and starts a query for showing all symbols that have the current symbol as a component. The symbol must be a valid type. If more than one symbol match, a Choose Symbol dialog is opened.	sxu



Vim Command	Description	Shortcut
<code>:sniff show-docu</code>	Loads the documentation of the symbol into a Documentation Editor.	<code>sd</code>
<code>:sniff gen-docu</code>	Opens the Documentation Synchronization dialog.	<code>sD</code>



## Codewright Integration (Windows only)

---

### Introduction

You can use the Codewright editor for editing source code in SNIFF+. This chapter covers the basics of the integration.

#### This chapter covers the following topics

- Integrate SNIFF+ with Codewright
- Work with the integration

#### Assumptions made in this chapter

- You are an experienced Codewright user

### Integration Features

This integration is based on Codewright 5.0. The following integration features are available:

- Codewright can be used for all editing requests made in SNIFF+.
- SNIFF+ recognizes and updates all browsers when a file is saved in Codewright.
- SNIFF+ commands can be issued directly from Codewright.

A `dll` file supplied with the SNIFF+ distribution tells Codewright how to communicate with SNIFF+. This file is called `cwsniff.dll` and is located in your `%SNIFF_DIR%\integrations\codewright-5.0` directory.

# Integrating Codewright

## Prerequisites

- SNIFF+ installed at your site.
- Codewright 5.0 installed at your site.
- For SNIFF+ to communicate with Codewright and vice-versa, the following file is needed

File	Description
cwsniff.dll	Distributed together with SNIFF+ and implements functions that send commands from Codewright to SNIFF+. This file is in %SNIFF_DIR%\integrations\codewright-5.0

## Connecting Codewright to SNIFF+

To connect Codewright to SNIFF+, complete the following steps

### Note

We assume that you installed Codewright in the following directory:  
C:\codewright

1. Copy the cwsniff.dll file from  
%SNIFF\_DIR%\integrations\codewright-5.0 to C:\codewright
2. Make a backup of your old C:\codewright\cwright.ini file.
3. Copy the cwright.ini file from  
%SNIFF\_DIR%\integrations\codewright-5.0 to C:\codewright
4. If you want, merge your old cwright.ini file with the one supplied by us. See also [Merging the old cwright.ini file with the new cwright.ini file — page 253](#)
5. Make sure SNIFF+ can start Codewright by setting your PATH environment variable to:  
PATH=%PATH%;C:\codewright

### Note

You can have only one Codewright-SNIFF+ connection active at a given time.

## Setting Codewright as your preferred editor

1. Start SNIFF+.
2. Choose **Tools > Preferences** in any open SNIFF+ tool.

### In the Preferences dialog

1. Select the **Tools > Source Editor** node.
2. From the **Current Editor** drop-down, choose **Codewright**.
3. Press **Ok**.

SNIFF+ will now use Codewright for all editing requests.

## Merging the old `cwright.ini` file with the new `cwright.ini` file

If you have already customized Codewright and don't want to lose the changes, merge the old `cwright.ini` file with the new one. To do so:

- Copy the lines that differ and paste them into the appropriate sections.

To simplify the merging process, lines relevant to the Codewright integration with SNIFF+ are "blocked" by special comment lines. This makes it easier to find the lines that Codewright needs to be able to communicate with SNIFF+.

These lines are enclosed in the following comment lines:

```
BEGIN_SNIFF_SPECIFIC
END_SNIFF_SPECIFIC
```

### Note

Comments inside the SNIFF+ specific blocks explain problems which you might encounter when merging.

## Working with Codewright and SNIFF+

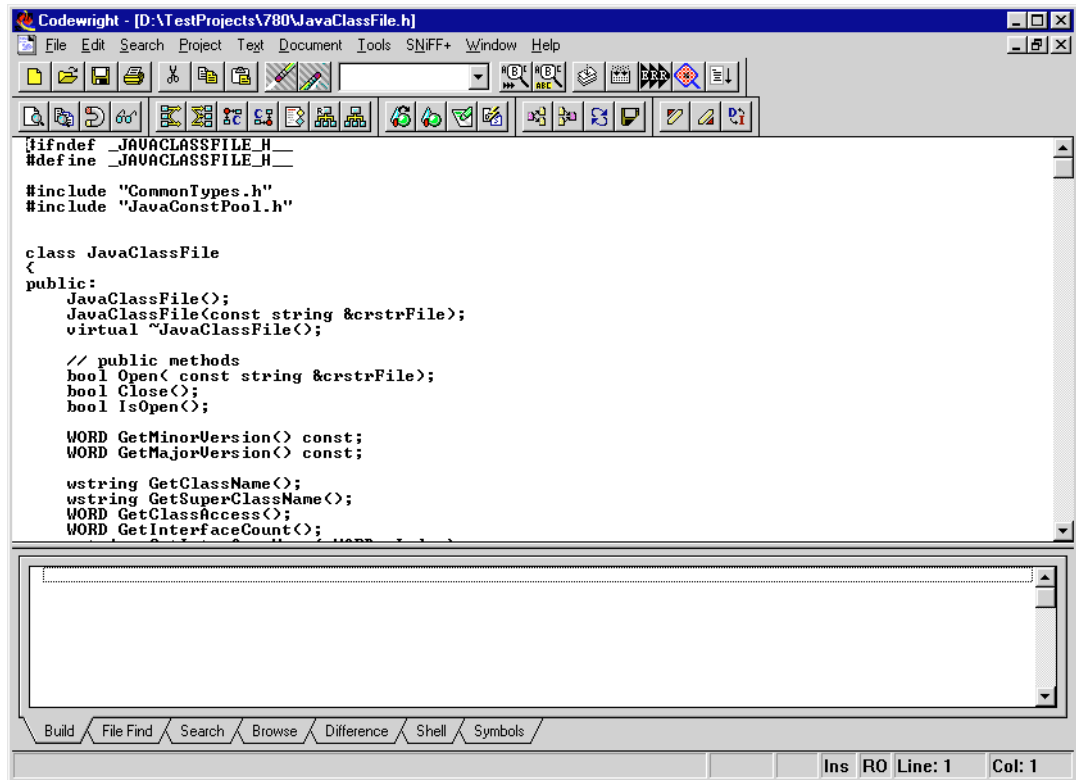
Once a connection between SNIFF+ and Codewright is established, SNIFF+ uses Codewright for all requests to display or edit source code.

### Starting the Codewright Editor from SNIFF+

There are two possibilities to start the Codewright Editor from SNIFF+:

- Select the **Editor** command from the **Tools** menu in any open SNIFF+ tool.
- Double-click on a file in the File List in the Project Editor.

The Codewright Editor appears.



### Positioning Codewright from SNIFF+

By double-clicking on a symbol in any SNIFF+ browser, Codewright loads the corresponding source file and positions the cursor at the appropriate location.

## Command Reference

Many menu commands correspond to the standard SNIFF+ menu items. (All commands act on the symbol under the current cursor position.) For a description of these menu items, please see the *Reference Guide*, [Common Menus — page 13](#). Additional commands are described in the following table:

Command	Description
<b>Reparse File</b>	SNIFF+ reparses the current file and updates the Symbol Table
<b>Save Without Reparse</b>	SNIFF+ saves the current file without reparsing it
<b>Toggle Declaration/Implementation</b>	Toggles between the implementation and definition of a symbol





# MS Developer Studio Integration (Windows) **24**

---

## Introduction

You can use MS Developer Studio (5.0 or 6.0) for editing and debugging source code in SNIFF+. This chapter covers the basics of the integration.

### This chapter covers the following topics

- Integrate SNIFF+ with MS Developer Studio
- Work with the integration

### Assumptions made in this chapter

- You are an experienced MS Developer Studio user

## Integration Features

The following integration features are available:

- MS Developer Studio can be used for all editing requests made in SNIFF+.
- SNIFF+ recognizes and updates all browsers when a file is saved in MS Developer Studio.
- SNIFF+ commands can be issued directly from MS Developer Studio.
- Custom menus can be maintained for both the SNIFF+ integrated mode and the native MS Developer Studio mode.
- The native MS Developer Studio debugger is used for debugging.

# Integrating MS Developer Studio

## Prerequisites

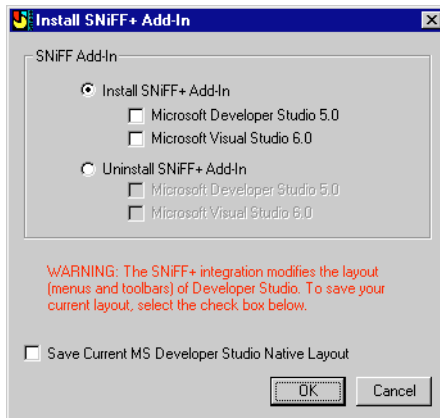
- SNIFF+ 3.0 or later is installed at your site.
- MS Developer Studio 5.0 or 6.0 is installed at your site.

## Connecting MS Developer Studio

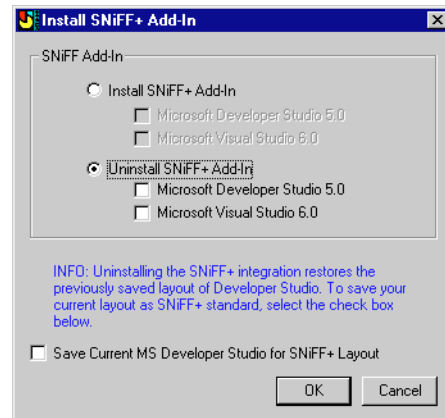
To use the MS Developer Studio editor and debugger in SNIFF+

1. Make sure MS Developer Studio is closed.
2. In your SNIFF+ for Windows program group, click **MSDev Connection** to open the Install SNIFF+ Add-In dialog.
3. Select the **Install SNIFF+ Add-In** option.
4. Select the **Microsoft Developer Studio 5.0** check box.
5. To save the current MS Developer Studio settings for your next native mode session, select the **Save Current MS Developer Studio Native Layout** check box.

### Connecting:



### Disconnecting:



## Disconnecting MS Developer Studio

To disconnect the MS Developer Studio editor and debugger from SNIFF+

1. Make sure MS Developer Studio is closed.
2. In your SNIFF+ for Windows program group, click **MSDev Connection** to open the Install SNIFF+ Add-In dialog.
3. Select the **Uninstall SNIFF+ Add-In** option.
4. Select the **Microsoft Developer Studio 5.0** check box.

5. To save the current MS Developer Studio settings for your next SNIFF+ mode session, select the **Save Current MS Developer Studio for SNIFF+ Layout** check box.

## Switching between modes

We recommend that you disconnect MS Developer Studio from SNIFF+ before working on non-SNIFF+ projects. Before you work on SNIFF+ projects, reconnect MS Developer Studio. Custom settings can be maintained for both modes, so that switching between modes is quick and painless (see above).

Note that (custom) layout and toolbar etc. settings are separately saved to the Registry, and not merged. When you first open DevStudio, default settings are used. You change the layout to suit your needs at any time, the new layout is saved for the SNIFF+ mode when you close MS Developer Studio.

## Working with MS Developer Studio and SNIFF+

- Make sure that you have connected MS Developer Studio to SNIFF+ as described under [Connecting MS Developer Studio — page 258](#).

## Setting MS Developer Studio as your preferred editor

1. Start SNIFF+.
2. Choose **Tools > Preferences** in any open SNIFF+ tool.

### In the Preferences dialog

1. Select the **Tools > Source Editor** node.
2. From the **Current Editor** drop-down, choose **MS DevStudio 5.0 (DDE)**.
3. Press **Ok**.

SNIFF+ will now use MS Developer Studio for all editing requests.

- If you disconnect MS Developer Studio, don't forget to reset your preferences.

## Opening the MS Developer Studio editor from SNIFF+

To start the MS Developer Studio editor from SNIFF+ you can

- Double-click on a file in the File List in the Project Editor. This opens the selected file positioned at the first line.
- Double-click on a symbol in any SNIFF+ browsing tool. This opens the file where the selected symbol is declared and positions to the declaration.
- Select the **Source Editor** command from the **Tools** menu in any open SNIFF+ tool. This opens an empty editor.

### Editing

The MS Developer Studio functionality is maintained unchanged under the familiar **File**, **Edit** and **View** menus. Please refer to your MS Developer Studio documentation for details.

### Building

Compile and build commands are under the **SNIFF+ Build/VCS** menu. Commands are executed in the SNIFF+ Shell tool. Please see the [SNIFF+ Build/VCS menu — page 261](#) for a description of the available commands.

### Debugging

**Debug** commands are under the **SNIFF+ Build/VCS** menu. The functionality is the same as in the native MS Developer Studio mode.

### SNIFF+ versioning

SNIFF+ version controlling commands are under the **SNIFF+ Build/VCS** menu. Please see the [SNIFF+ Build/VCS menu — page 261](#) for details.

### SNIFF+ browsing

SNIFF+ browsing and cross-file navigation commands are under the SNIFF+ Browsing menu.

## Menus

Note that native MS Developer Studio commands are not described here.

### SNiFF+ Build/VCS menu

All the commands in this menu apply to the currently opened file.

SNiFF+ Build/VCS command	Description
<b>Compile File</b>	Compiles the loaded file.
<b>Build Project Target</b>	Builds the project the loaded file is part of, unless otherwise set with the <b>Custom Build...</b> command.
<b>Custom Build...</b>	Calls up a dialog where you can select and order targets. Requires the executable as Workspace.
<b>Debug</b>	Starts debugging the currently loaded project. This entry is only enabled when you have entered the target name in the <b>Make</b> view of the Project Attributes dialog and the target is executable.
<b>Check Out...</b>	Checks current file out of the Repository.
<b>Check In...</b>	Checks current file in to the Repository.
<b>Lock File...</b>	Locks current file in the Repository.
<b>Unlock File...</b>	Unlocks current file in the Repository.
<b>Show History...</b>	Shows file history.
<b>Show Differences...</b>	Shows file version differences.

## SNiFF+ Info menu

Info menu command	Description
<b>Retrieve Selection from File</b>	Retrieves all occurrences of <i>Selection</i> from the current file.
<b>Retrieve Selection from Project</b>	Retrieves all occurrences of <i>Selection</i> from the project to which the current file belongs.
<b>Retrieve Selection from All Projects</b>	Retrieves all occurrences of <i>Selection</i> from all projects in the Project Tree.
<b>Symbol Refers To</b>	Opens a Cross Referencer and starts a Refers-To query on <i>Symbol</i> .
<b>Symbol Referred By</b>	Opens a Cross Referencer and starts a Referred-By query on <i>Symbol</i> .
<b>Symbol Refers To Components</b>	Opens a Cross Referencer to show all symbols (classes and structures) that are components of <i>symbol</i> . If the current selection is a member of a class/structure, the class/structure is taken for this query.
<b>Symbol Referred By As Component</b>	Opens a Cross Referencer to show all symbols that have <i>symbol</i> as a component. Note that you can also query primitive C data types with this command.
<b>Find Symbols Matching Selection</b>	Opens a Symbol Browser to display symbols that match the name of <i>Selection</i> .
<b>Find Symbols Containing Selection</b>	Opens a Symbol Browser to display symbols that contain <i>Selection</i> .
<b>File Includes</b>	Opens an Include Browser and shows the files that the current file includes.
<b>File Included By</b>	Opens an Include Browser and shows the files that the current file is included by.
<b>Reparse File</b>	SNiFF+ reparses the current file and updates the Symbol Table.
<b>Show Documentation</b>	Opens a Documentation Editor and positions it to the documentation of <i>Symbol</i> . An alert message appears if no documentation file exists for either the entire file or for <i>Symbol</i> .
<b>Documentation Synchronizer...</b>	Opens the Documentation Synchronizer.

## SNiFF+ Browsing menu

Class menu command	Description
<b>Browse Class</b>	Loads <i>class</i> into a Class Browser.
<b>Show Class Relatives</b>	Opens a Hierarchy Browser and loads the graph of the base and derived classes. <i>Class</i> is highlighted in the Hierarchy Browser
<b>Show Class in Entire Hierarchy</b>	Opens a Hierarchy Browser and loads the entire class graph. <i>Class</i> is highlighted in the Hierarchy Browser.
<b>Base Class</b>	Shows the declaration of the immediate base class of the class where the cursor is currently positioned.
<b>Overridden</b>	Shows the overridden <i>method</i> of the closest base class that uses the selected <i>method</i> .
<b>Toggles Header/Implementation File</b>	Shows the corresponding header/implementation file.





# Part IX

## ClearCase Integration



# Integrating SNIFF+ with ClearCase

---

## Introduction

ClearCase from Rational Software, is a high-end software configuration management tool that is used by software development teams working in either the Unix or Windows NT environment. ClearCase provides comprehensive configuration management features, including version control, workspace management, build management and process control. Both text-based and graphical interfaces are available.

With the ClearCase integration with SNIFF+, you can perform ClearCase operations on your SNIFF+ projects and their source files using SNIFF+'s CMVC interface.

### This chapter covers the following topics

- Set up a SNIFF+ project with ClearCase
- Work with ClearCase in SNIFF+
- Customize ClearCase update features
- Set your Make command in SNIFF+

### Assumptions made in this chapter

- You are an experienced ClearCase user
- Your ClearCase administrator has already set up at least one VOB and a View that accesses the VOB
- You are setting up a SNIFF+ project in your View

## Integration overview

### How ClearCase VOBs and Views are realized in SNIFF+

ClearCase VOBs (Versioned Object Bases) contain all public ClearCase data. This data can either be shared object or shared source data. ClearCase Views are private storage areas that access VOBs. The contents of a VOB can only be “seen” through a private View that accesses it.

In SNIFF+, ClearCase Views correspond to Private Working Environments (PWEs). There is no SNIFF+ equivalent to a VOB.

To view and work with the elements (source files and directories) of a VOB, you first have to create a PWE for the View with which you access the VOB. Then, you can open SNIFF+ projects in the PWE that contain the files you're interested in. If the projects don't exist, you first have to create them.

(Creating projects for ClearCase are discussed under [Setting up a SNIFF+ project with ClearCase — page 268.](#))

### How builds are performed in SNIFF+

You can use both standard Make and **clearmake** for building object files in your PWE. The appropriate Make command can be set in the Project Attributes dialog.

Shared Object Working Environments are also unnecessary when using **clearmake**, because **clearmake**'s wink-in feature handles how object files (ClearCase DOs) are shared between views.

## Setting up a SNIFF+ project with ClearCase

In this section, you will learn how to set up a SNIFF+ project in your ClearCase View. But before setting up a SNIFF+ project, you will have to prepare your ClearCase environment:

1. Make sure that all the files for which you want to create the project are already under the control of ClearCase. In other words, the files should already be elements of a VOB.
2. Make sure that there is a View that accesses the VOB. In the following, it is assumed that you are the owner of this View.
3. Set your View.
  - **On Unix** — In a shell, enter the command `cleartool setview your_view`.
  - **On Windows** — The View becomes active as soon as it is mapped to a drive.

4. **On Windows** — The SHELL environment variable must be modified to use forward slashes in the Make Support file. To do so, open the file

```
%SNIFF_DIR%\make_support\i386-unknown-win32.mk
```

and edit the line:

```
SHELL = sh
```

to read — using FORWARD slashes:

```
SHELL = path/to/directory/holding/sh.exe
```

(By default, sh.exe is in SNIFF\_DIR/bin)

## Launching the Project Setup Wizard

You're now ready to create the project. You'll do so in the Project Setup Wizard.

1. Start SNIFF+ if you haven't already done so.
2. To start the Project Setup Wizard, in the Launch Pad, choose **Project > New Project > with Wizard....**

### In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNIFF+ Project.

- Accept the default selection, **Standard Setup**, and press **Next**.

The “Select development task” page appears.

In the remaining steps, we will refer to the names of Wizard pages. You can find a page's name in the title bar of the Wizard.

### In the “Select development task” page

- Select **Create a new SNIFF+ Project from scratch** and press **Next**.

### In the “Your development organization” page

This tutorial is for single-user/single platform development using ClearCase as the underlying CMVC tool, so:

1. Accept the first two defaults (**No/No**) in the Page.
2. Select ClearCase as your version control tool.
3. Press **Next**.

### In the “Select file types” page

- Select the file types that you want to be included in the project and press **Next**.

Note that, after project setup, you can add new standard file types (like the ones in the “Additional File Types Column”), or create and add your own.

## In the “Specify Private Working Environment” page

You are asked to specify your Private Working Environment (PWE) root directory. This directory must be set to the mounting point of the VOB.

For example, suppose you have defined the VOB *first\_vob*. Furthermore, your View, called *your\_view*, accesses *first\_vob*. Then, the mounting point of the VOB is, for example:

- **On Unix** — `~your_home_dir/first_vob`
- **On Windows** — `F:\first_vob`, where *F* is the drive letter associated with *your\_view*

### Note

If you intend to change ClearCase Views during a single SNIFF+ session, please read [Changing Views in current SNIFF+ session — page 272](#) before continuing.

To specify your PWE root directory:

1. Press **Browse**, and in the Directory dialog, navigate to the mounting point of the VOB and then press **Select**.
2. In the **PWE name** field, enter a name for the PWE, e.g., `myClearcaseView`.

Notice that your username is entered next to the enabled **Owner** button. SNIFF+ needs your username to correctly handle permissions. Being the owner of the PWE means that you are the only one who is allowed to modify the working environment's attributes.

3. Press **Next**.

## In the “Create New SNIFF+ Project” page

SNIFF+ has set your **Project root directory** to the mounting point of the VOB. The project has the same name as the VOB, with the extension `.shared`.

1. If you intend to create the project in another directory, navigate to the **Project root directory** by pressing the **Browse** button.

The name of the project is entered in the **Project name** field with the extension `.shared`.

2. If SNIFF+ should automatically create subprojects in the subdirectories of the **Project root directory**, make sure that the **Create Subprojects** is selected.
3. Press **Next**.

## In the “Project Setup Summary” page

This page summarizes your specifications for the new SNIFF+ project and the PWE.

- Press **Finish**.

SNIFF+ will now create the new project and all its subprojects. When SNIFF+ is finished, it opens the new project and displays its structure and contents in the Project Editor.

## Working with ClearCase in SNIFF+

### Accessing ClearCase commands using the ClearCase custom menu

A custom menu called **ClearCase** is available in SNIFF+'s Project Editor. With it, you can invoke ClearCase commands to do such things as:

- listing versions
- listing checkouts
- listing history
- displaying version tree

### Activating the ClearCase custom menu

The **ClearCase** custom menu is predefined for the Project Editor in the site-wide custom menu file `SiteMenus.sniff`.

- **On Unix** — `SiteMenus.sniff` is located in `$SNIFF_DIR/config/`
- **On Windows NT** — `SiteMenus.sniff` is located in `%SNIFF_DIR%\config\`

By default, the **ClearCase** custom menu is disabled. To enable it:

1. Open `SiteMenus.sniff` in an editor.
2. Search for the following line in the PROJECT EDITOR section of the file:
 

```
# ClearCase Menus
```
3. Starting with the next line, remove all leading hash (#) characters from it and the following lines in the PROJECT EDITOR section of the file.
4. Save `SiteMenus.sniff`.

The **ClearCase** custom menu will be accessible the next time you start SNIFF+. If required, additional commands can easily be added on to the custom menu. For details, please refer to [Reference Guide — Custom menus — page 283](#).

### Changing View's config spec in current SNIFF+ session

You can change a View's config spec in ClearCase while working on projects opened in the corresponding PWE. You can then use the **ClearCase** custom menu in SNIFF+ for viewing the config specs and starting the appropriate ClearCase GUI for your platform (`xclear-case` or `cleardetails`).

Note that changing your View's config spec may result in a change in the project structure that you can "see" through the View. Therefore, in order for the View and its corresponding SNIFF+ PWE to reflect the most current information as defined in its config spec, reload all current projects opened in the PWE.

You can reload projects using the command **Project > Reload Project > In Current Working Environment** in either the Project Editor or the Launch Pad.

## Changing Views in current SNIFF+ session

In ClearCase, you may want to access data visible in other Views. SNIFF+ allows you to change your ClearCase View during a single session.

Changing a ClearCase View corresponds to reloading the current project in the working environment that corresponds to the new View.

To change Views during a single SNIFF+ session, we recommend that the root directories of the PWEs that you create for the Views point to the **view-extended full pathname** of the View. By using extended pathnames, you can use other Views active on your host without first having to “set” them.

For example, suppose three Views — `your_view`, `bill` and `john` — are active, and:

- `/vobs/design/src/msg.c` specifies the version of an element selected by your View
- `/view/bill/vobs/design/src/msg.c` specifies the version of the same element selected by the View `bill`
- `/view/john/vobs/design/src/msg.c` specifies the version of the same element selected by the View `john`

Furthermore, suppose the current project is open in the PWE corresponding to View `bill`, and you want to change the View to `john`. To do so, just reload the project in the PWE that corresponds to `john`.

To reload projects in another working environment, use the command **Project > Reload Project > In Other Working Environment** in either the Project Editor or the Launch Pad.

## Advanced features

### Notifying SNIFF+ of files checked-in with ClearCase

When a file is checked in with ClearCase, all Views which are configured to display the latest version of the file are immediately updated. If the file is part of a loaded SNIFF+ project, ClearCase and SNIFF+ can be configured so that the modification is notified to SNIFF+

To enable the automatic updating of the checked-in file's symbol information, the ClearCase **trigger** concept is used. Triggers can be defined to fire at specific events.

To notify SNIFF+ when files are checked-in with ClearCase, we need to define a trigger that fires on every **checkin** operation performed on any VOB that is used together with SNIFF+. In this section, you will learn how to define such a trigger.

The trigger that you define will execute a script that stores all **checkin** events in an Update Log File accessible to all running SNIFF+ processes. The path of the Update Log File is system dependent and will have to be set in the `SitePrefs.sniff` file.

Every SNIFF+ process that has at least one opened ClearCase project will have to check the Update Log File. When SNIFF+ finds a new entry that pertains to a file currently loaded in a Project Editor, it will reparse the file and thereby update its symbol information.



## Defining the trigger

The follow instructions are for defining the trigger on a Unix platform.

### Note

The Update Log File that stores the **checkin** events must be writable by all developers. This is because the trigger script will run with the `user id` of the user performing the checkin operation. Also, the Update Log File will grow line by line with every **checkin** operation, so we recommend that you delete it regularly (and preferably outside usual working times).

1. Choose the location of the Update Log File and make sure that the location is accessible from the whole network. Otherwise SNIFF+ cannot update symbol information automatically.
2. In SNIFF+, open the Preferences dialog and navigate to the **Version Control System** node.
3. Select **ClearCase** and press the **More Options...** button.
4. In the Advanced dialog that appears, select the **Enable Automatic Update** check box.
5. In the **Update Log File** field, specify the location of the Update Log File.
6. In the **Update Interval** field, enter a value (in seconds) for the update timer.  
SNIFF+'s update timer determines how often SNIFF+ checks the Update Log File.
7. If you want to be informed when SNIFF+ checks the Update Log File, make sure that the **Show Confirmation dialog** check box is selected.  
When SNIFF+ checks the Update Log File, a dialog will appear with a list of all the files checked-in since the last check of the log file.
8. Define a trigger for every VOB used together with SNIFF+. To look at a sample trigger, refer to [Sample trigger — page 273](#).

## Sample trigger

To define a trigger, you need to have the necessary permissions to modify a VOB. Generally, either only VOB owner or your ClearCase administrator will have the needed permissions. For detailed information about permissions, please refer to your ClearCase manuals.

To define a trigger:

1. Change to the VOB directory (here, we assume the VOB is already mounted):

```
cd /vobs/hello_prj
```

## 2. Define the trigger type.

### for ClearCase Release 3.x

```
cleartool mktrtype -element -all \
-postop checkin \
-eltype text_file \
-exec "$SNIFF_DIR/bin/sniff_ClearCase_notify.sh
<logfile_fullpath>" \
-c 'SNIFF+ checkin notification trigger' \
sniff_notify_trigger
```

### for ClearCase Release 2.x

```
cleartool mktrtype -element -global \
-postop checkin \
-eltype text_file \
-exec "$SNIFF_DIR/bin/sniff_ClearCase_notify.sh
<logfile_fullpath>" \
-c 'SNIFF+ checkin notification trigger' \
sniff_notify_trigger
```

This command defines a trigger for all elements of type *text\_file* in our example VOB. The type *text\_file* is predefined by ClearCase. If there are other element types that should fire a trigger at **checkin**, simply add them to the `-eltype` option above as a comma-separated list. More options to this command are described in the manual page of the **mktrtype** subcommand.

## 3. Repeat the above steps for all VOBs that are in use with SNIFF+

## Setting your Make command

Both standard Make and **clearmake** can be used for building object files. The appropriate Make command can be set in the Project Attributes dialog.

- To set **clearmake** as your Make command, please refer to [Setting clearmake as your Make command — page 275](#).
- To use SNIFF+'s Make Support with ClearCase and a standard Make utility, please refer to [Build and Make Support — page 73](#).
- To use your own Makefiles and a standard Make utility, please refer to [Using Your Own Makefiles — page 105](#).

SNIFF+'s shared object workspace feature is also unnecessary when using **clearmake**. **clearmake** can handle the sharing of object files (DO derived objects) between views. This is done by selecting the wink-in feature.

## Setting clearmake as your Make command

To use your own makefiles:

1. Start SNIFF+ and open the root project for which you want to set **clearmake** as your Make command.
2. Check out the Project Description Files (PDFs) of all the projects.
3. In the Project Tree, checkmark all the projects.
4. Choose **Project > Attributes of Checkmarked Projects...**  
The Group Project Attributes dialog appears.
5. Select the **Build Options** node.

You will now set the Make attributes that are the same for all projects.

### In the Group Project Attributes dialog

1. Enter your command that you use to run **clearmake** in the **Make Command** field. For example:  

```
clearmake <options>
```

This command will then be submitted to the Shell when you execute the **Target > Make default target** command.
2. Select the check box to the right of the **Make Command** field.
3. Press the **Set for All** button to apply this attribute to all the projects checkmarked in the Project Tree.
4. Press the **Ok** button to apply the settings and to close the Group Project Attributes dialog.
5. Save the modified project.



# Part X

## Documenting Source Code



## Introduction

You can document your source code with SNIFF+'s Documentation Editor. In this chapter, you will learn how to do so.

### This chapter covers the following topics

- Write and browse source code documentation
- Manage source code documentation
- Export source code documentation
- Modify the appearance of source code documentation

### Assumptions made in this chapter

- You know how to browse and edit source code in SNIFF+
- You know how to work with SNIFF+ file types

### Related SNIFF+ topics

- Specifying your SNIFF+ Preferences — [Reference Guide — Preferences — page 123](#)
- Using the Documentation Editor — [Reference Guide — Documentation Editor — page 89](#)
- Using the Documentation Synchronization dialog — [Reference Guide — Documentation Synchronizer — page 93](#)

## Documentation Editor modes

The Documentation Editor operates in one of two modes:

- **Editing mode (read/write)**—Mode of operation for documenting a software project. When the Documentation Editor is in this mode, you can generate documentation files out of your source code.
- **Browsing mode (read-only)**—Usual mode of operation for simply browsing documentation files. When the Documentation Editor is in browsing mode, no changes can be made to existing documentation, and obsolete documentation will not be displayed.

You can set the operating mode of the Documentation Editor in your Preferences.

## Writing source code documentation

This section covers the steps you must complete in order to document your source code. These steps are:

1. Switch the Documentation Editor to editing mode
2. Add the **Documentation file** type to project
3. Determine which symbols are to be documented

Once you have completed these first three steps for a project, you generally do not have to repeat them.

1. Generate a documentation file from your source code.
2. Edit the documentation frames in the generated documentation file

### Step 1: Switch to editing mode

To switch the Documentation Editor to editing mode, please complete the following steps:

1. Choose **Preferences...** from the **Tools** menu.  
The Preferences dialog appears.
2. Select the **Documentation Editor** node.  
The **Documentation Editor** view appears.
3. Under **Document Creation**, clear the **Use Read-Only Mode** check box.
4. Press **Ok** to apply and close your Preferences.

You can now generate and modify documentation.



## Step 2: Add the Documentation file type to project

The next step is to add the **Documentation** file type to your project. Documentation files created by SNIFF+ are associated with this file type. If the file type is not part of a project and you try to generate documentation files for the project, you will receive a warning message from SNIFF+.

To add the **Documentation** file type:

1. Open the Project Attributes dialog of the project for which you want to create documentation files.
2. Select the **File Types** node.
3. Press the **Show All** button.
4. Select the **Documentation** file type from the File Types list and press the **Add File Type** button.

The file type will now be part of the project.

5. Press the **Ok** button to add the file type to the project.
6. Save the project.

### Note

To add the **Documentation** file type to multiple projects at the same time, use the Group Project Attributes dialog. For details, please refer to [Using the Group Project Attributes dialog — page 131](#).

## Step 3: Determine which symbols to document

You can generate documentation frames for every symbol that is contained in your source files. However, you may not always want to do this.

You can select the symbol types to be documented in the Preferences dialog. Documentation frames will then be generated for these symbol types only. Note that your settings affect the documentation generation process only—they do not affect existing documentation.

To select the symbol types to be documented, please complete the following steps:

1. Choose **Preferences...** from the **Tools** menu.  
The Preferences dialog appears.
2. Select the **Documentation Editor** node.  
The **Documentation Editor** view appears.
3. Under **Documentable Symbol**, for each symbol type that you want to document, select the symbol type and then select the **Documentable** check box.
4. Apply and close your Preferences dialog by pressing the **Ok** button.

The next time you generate documentation, only documentation frames for the symbol types that you just selected will be generated.

## Step 4: Generate a documentation file

Once the Documentation Editor is in editing mode, you can generate a documentation file from your source code. To generate a documentation file, you need to:

1. Open the source file for which you want to generate a documentation file.

Note that the documentation file that you are generating contains documentable symbols taken from both the implementation file and its corresponding header file.

2. To generate documentation frames for all the (documentable) symbols in the file, position the text cursor outside the scope of any symbol (for example, at the beginning of the file).

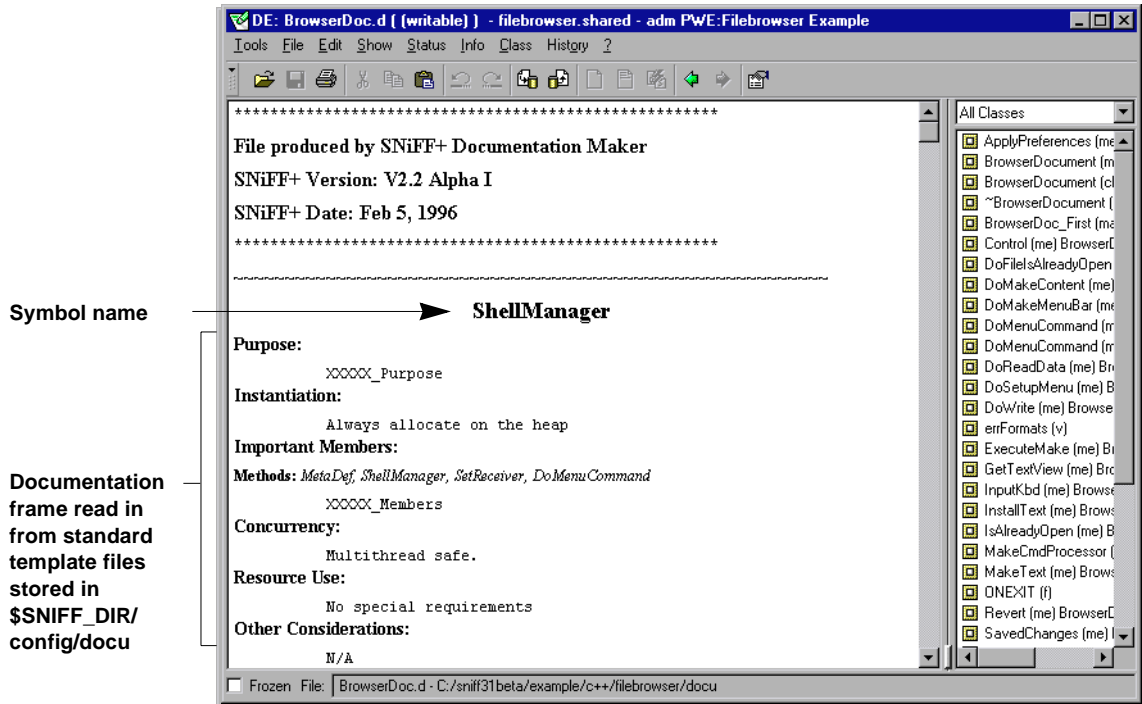
(Later on, you can generate documentation frames for any symbol types that are undocumentable at this time. To do so, make these symbol types documentable (in your Preferences) and then synchronize the documentation file with its source file. The synchronization process is described later in this chapter.)

3. Choose the **Show Documentation of File <file>** command from the **Info** menu.

You will then get a message asking you whether you want to generate a documentation file for the source file.

4. Press the **Yes** button to generate the documentation file.

The newly generated documentation file is loaded into a Documentation Editor.



The documentation file consists of a series of documentation frames, one for each documentable symbol in the corresponding source and header files. These documentation frames come from standard documentation template files that are stored in `$SNIFF_DIR/config/docu`.

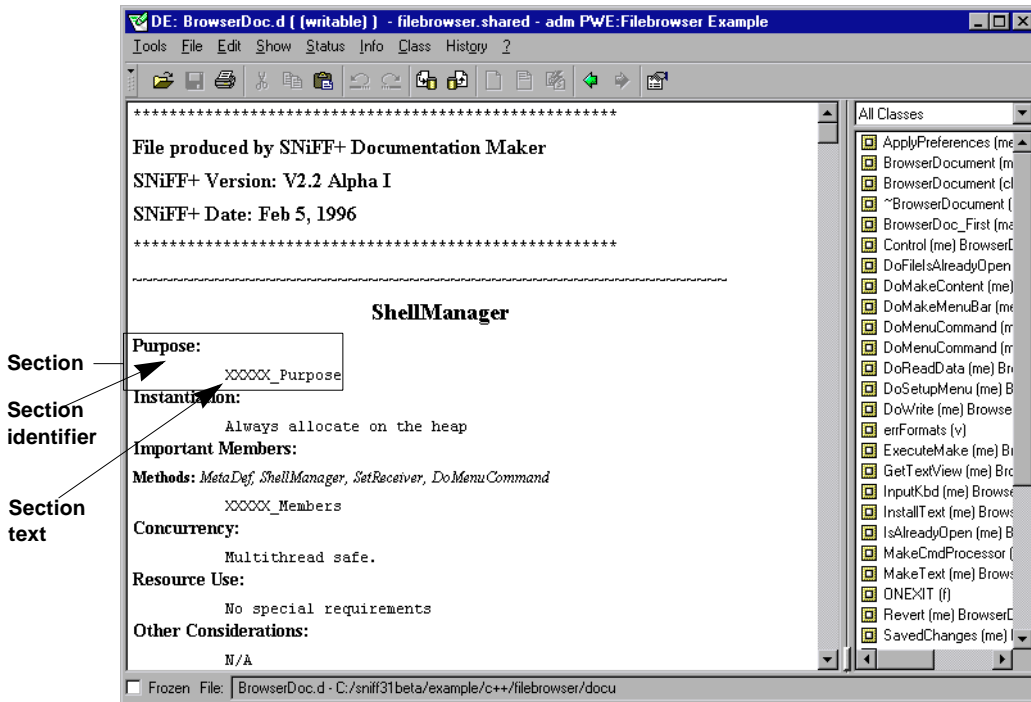
#### Note

You can also create documentation files in the Documentation Synchronizer.

## Step 5: Edit the documentation frames

The documentation frames that you've just generated contain information taken from documentation templates files. There is one documentation frame for each symbol that is listed in the Symbol List of the Documentation Editor.

A symbol's documentation starts with the *symbol name* and *symbol signature* (if it has one). The actual documentation of the symbol follows in the *documentation body*, which is split into a series of *sections*. Each section begins with a *section identifier*. Its *section text* follows on the next line.



You can change the contents of only the section text field. As a result, if you decide to change the layout of the documentation frames, please make sure that the section text fields are distinguishable from the other fields of a documentation frame.

You can apply the standard editing functions (Cut, Copy, Paste, etc.) to entire documentation frames, to a section of a documentation frame, or to the section text of a section.

For a description of the various commands that are available in the **Edit** menu of the Documentation Editor, please refer to [Reference Guide — Edit menu — page 16](#).

## Jumping between the source code and documentation

During the process of documenting a symbol, it may be necessary for you to jump to either its definition or implementation. You can do this by selecting the **Symbol(s) symbol** command in the **Show** menu of the Documentation Editor. The Choose Symbol dialog appears. You can then jump to either the symbol's definition or implementation from this dialog.

When you are in the source or header file, you can jump back to the corresponding documentation file by selecting the symbol and then choosing the **Show Documentation of symbol** command in the **Info** menu of the Source Editor.

## Changing the documentation status of a symbol

A symbol's documentation can be in one of three possible states: undocumented, partially documented, or (fully) documented. Initially, the documentation status of a symbol is undocumented.

You alone are responsible for determining what the documentation status of a symbol is. SNIFF+ doesn't automatically change the status when you have made changes to a symbol's documentation.

To change the documentation status of a symbol, choose one of the commands that are available in the **Status** menu of the Documentation Editor.

## Looking at the status of a symbols documentation

In addition to monitoring the documentation status of individual symbols in your source files, you may also want to look at the documentation status of entire source files and projects. You can do so in the Documentation Synchronization dialog. You can open this dialog by choosing the **Documentation Synchronizer...** command in the Documentation Editor's **Info** menu.

Your source files and projects can be in one of four possible documentation states — empty, undocumented, partially documented and fully documented. The following table summarizes these states:

Source Files	Projects
<b>Empty file:</b> This source file doesn't have a corresponding documentation file.	<b>Empty project:</b> None of the source files in the project has a corresponding documentation file.
<b>Undocumented file:</b> All symbols in the corresponding documentation file are undocumented.	<b>Undocumented project:</b> Project contains at least one source file that contains only empty documentation frames.
<b>Partially documented file:</b> At least one symbol in the corresponding documentation file is either partially documented or fully documented.	<b>Partially documented project:</b> Project contains at least one source file that is either partially documented or fully documented.
<b>Fully documented file:</b> All symbols in the corresponding documentation file are fully documented.	<b>Fully documented project:</b> All source files in the project are fully documented.

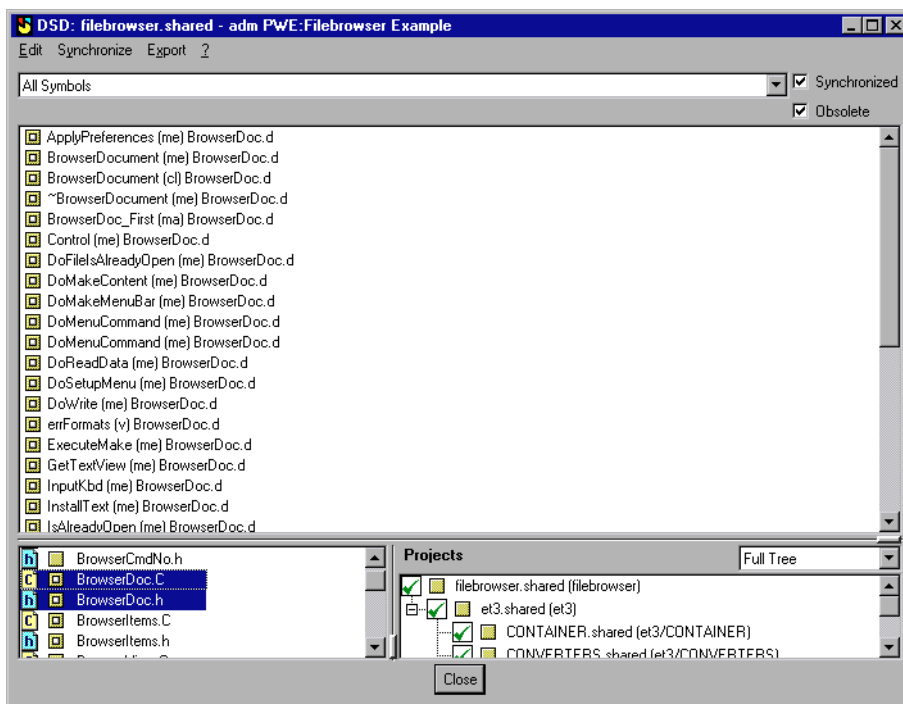
## Updating documentation

You should update your documentation files whenever you have:

- added symbols to source files
- renamed symbols in source files
- deleted symbols from source files
- moved symbols from one source file to another that is either in the same project or a different one

To update your documentation file, choose **Documentation Synchronizer...** in the Documentation Editor's **Info** menu. The Documentation Synchronization dialog appears.

(For a detailed description of the Documentation Synchronization dialog, please refer to [Reference Guide — Documentation Synchronizer — page 93](#). In this section, we will just discuss the update process.)



## Updating documentation — procedures

(Unless otherwise stated, all the commands referred to below are to be executed in the Documentation Synchronization dialog.)

### A symbol is added to the source code

When you add a symbol to one of your source files, you will have to generate a documentation frame for it in the source file's corresponding documentation file.

- Select the source file from the File List and then choose the **Synchronize Documentation of Selected Files** command from the **Synchronize** menu.

SNiFF+ generates a documentation frame for the symbol in the corresponding documentation file of the selected source file.

### A symbol is renamed in the source code

When you rename a symbol in a source file, you will have to generate a new documentation frame for the renamed symbol in the source file's corresponding documentation file.

The original symbol no longer exists in SNiFF+'s Symbol Table. As a result, it becomes obsolete when you synchronize the source file with its corresponding documentation file.

You can then paste the contents of the original (now obsolete) symbol's documentation frame into the newly generated documentation frame and then delete the obsolete symbol's documentation frame from the documentation file.

1. To generate a new documentation frame for the renamed symbol and make the original symbol obsolete, select the source file from the File List and then choose the **Synchronize Documentation of Selected Files** command from the **Synchronize** menu.

A documentation frame is generated for the renamed symbol in the source file's corresponding documentation file.

The original (now obsolete) symbol is displayed in italics in the Documentation Editor and in the Documentation Synchronization dialog.

2. To copy the contents of the original (now obsolete) symbol's documentation frame into the newly generated documentation frame, select the obsolete symbol from the Symbol List and then choose the **Matching Symbols for Obsolete Symbol** command from the **Synchronize** menu.

Alternatively, you can use the **Copy** and **Paste** commands in the **Edit** menu.

3. To delete the obsolete symbol's documentation frame from the documentation file, select the obsolete symbol from the Symbol List and then choose the **Delete** command from the **Edit** menu.

### A symbol is deleted from the source code

A symbol that is deleted from your source code no longer exists in SNiFF+'s Symbol Table. As a result, it becomes obsolete when you update the source file's corresponding documentation file.

1. To make the deleted symbol obsolete, select the source file from the File List and then choose the **Synchronize Documentation of Selected Files** command from the **Synchronize** menu.

The deleted (now obsolete) symbol is displayed in italics in the Documentation Editor and in the Documentation Synchronization dialog.

2. To delete the obsolete symbol's documentation frame from the documentation file, select the obsolete symbol from the Symbol List and then choose the **Delete** command from the **Edit** menu.

The obsolete symbol's documentation frame disappears from the documentation file.

## Symbols are moved to another file

You can move symbols and their documentation from one file to another. These files can be in the same project or in two different ones.

You can move documentation from one documentation file to another by using the **Copy** and **Paste** commands in the **Edit** menu. But before you can do this, you will have to generate a documentation frame for the symbol in the new source file's corresponding documentation file.

After you have moved the documentation, you can delete the symbol's documentation frame in the original documentation file.

1. To generate a documentation frame for the moved symbol in its new source file's corresponding documentation file, select the new source file from the File List and then choose the **Synchronize Documentation of Selected Files** command from the **Synchronize** menu.

A documentation frame is generated for the moved symbol in the new source file's corresponding documentation file.

2. To move documentation from the original documentation file to the new documentation file:
3. Select the original source file from the File List and then the moved symbol from the Symbol List.
4. Choose the **Copy** command from the **Edit** menu.
5. Select the new documentation file from the File List and then the moved symbol from the Symbol List.
6. Choose the **Paste** command from the **Edit** menu.

The documentation for the moved symbol now exists in both the original documentation file and in the new documentation file.

7. To delete the symbol's documentation frame in the original documentation file:
8. Select the original source file from the File List and then the moved symbol from the Symbol List.
9. Choose the **Delete** command from the **Edit** menu.

The symbol's documentation frame disappears from the original documentation file.



## Browsing documentation

Once you have finished documenting your source code, you can make the documentation readable but not editable by switching the mode of the Documentation Editor to **Read Only**. When the Documentation Editor is in this mode, it can only be used for browsing documentation.

### Switching the Documentation Editor to browsing mode

To switch the Documentation Editor to browsing mode, please complete the following steps:

1. Choose **Preferences...** from the **Tools** menu.  
The Preferences dialog appears.
2. Select the **Documentation Editor** node.  
The **Documentation Editor** view appears.
3. Under **Document Creation**, select the **Use Read-Only Mode** check box.
4. Press **Ok** to apply and close your Preferences.

You can now only browse your documentation.

### Browsing a symbol's documentation

You can browse a symbol's documentation in any tool that displays symbolic information.

To browse the documentation of a documented symbol, select the symbol and then choose the **Show Documentation of *symbol*** command from the **Info** menu. The symbol's documentation is then loaded into a Documentation Editor.

## Managing documentation together with source code

You can manage your documentation files in the same way as the other files of your projects. Documentation files, like all other project files, are associated with a file type. The file type describes the various attributes of files of a particular type, including where these files are located (relative to the project directory).

You should store your project documentation files in a subdirectory of the project directory and version control them in the same way as your other version controlled files. You can then keep your documentation up-to-date by checking them in and out along with their corresponding source files, regardless of version or configuration.

## Exporting documentation

You can export your documentation in two different formats: MIF (Maker Interchange Format) and HTML.

Please refer to [Reference Guide — Export menu — page 98](#) for instructions on how to export your documentation files.

## Changing the layout of source documentation

### Adapting documentation templates

The layout of a symbol's documentation frame is determined by a set of customizable documentation template files. There is one documentation template file for each type of symbol in your source code.

SNiFF+ comes with a set of standard documentation template files (stored in `$SNiFF_DIR/config/docu`). You can either use these files for generating the documentation frames for your symbols, or you can create your documentation template files. See also [Creating documentation templates files — page 290](#).

### Modifying the appearance of documentation in SNiFF+

You can modify the appearance of the documentation that you see in the Documentation Editor. For example, you can change the fonts or font sizes that are used in documentation frames. Note that these changes do not affect your documentation template files—they only affect the layout of a symbol's documentation in the Documentation Editor.

See also [Reference Guide — Documentation Editor view — page 140](#).

## Creating documentation templates files

### Introduction

When you document symbols in your source code, SNiFF+ uses documentation template files to generate documentation frames for each symbol. There is one documentation template file for each type of symbol.

To create documentation template files, complete these steps:

1. Customize an existing template file or create a new one.
2. Name it.
3. Determine where to store it in your file system.
4. Specify its location in SNiFF+.

### Step 1: Customize documentation template files

There are two ways in which you can customize documentation template files:

- by using HTML tags for layout
- by using macros to extract information from SNiFF+'s Symbol Table and comments from your source files

In this section, you will find out which HTML tags are recognized by SNiFF+ and which macros you can use for extracting symbolic information and comments.

## Using HTML tags in documentation template files

SNiFF+'s Documentation Editor recognizes the following HTML tags:  
(In the following list, only the starting tags are shown.)

```
<p>
<sub>, <sup>
<code>
<b>, <i>, <em>, <cite>, <strong>
<h1> ... <h6>
<center>
<plaintex>
<pre>
<ul>
<ol>
<dt>
<dd>
<body>
<html>
<!--
```

Note that you can use other HTML tags in your documentation template files as well. You may want to do so if you plan to browse your documentation template files with an HTML browser. SNiFF+ will ignore any HTML tags that are not listed above.

## Maintaining formatting (newline) in HTML

Use the `<pre>` tag to maintain formatting (newline) in HTML. For example:

```
<pre><SNiFFINSERT CODE="@FIRST_COMMENT@"></pre>
```

(The macro used in this example is described below.)

## Using macros in documentation template files

You can use the macros listed in the following table to incorporate information from SNiFF+'s Symbol Table into your documentation. Please note that macro expansion only works when the **Use Parser Comments for Syntax Highlighting** checkbox in the Project Attributes > Parser view is selected.

Each macro is associated with a particular program scope (see table). If you set the macro in an incorrect scope, SNiFF+ will not be able to expand the macro, and you will get an error message when trying to generate documentation frames.

For example, suppose you want to list the base classes of each class in your source code. You would use the `@BASE_CLASS_LIST@` macro to do so. Since this macro has a class scope, you would have to set this macro in the documentation template file used for docu-

menting classes. Then, whenever documentation frames are generated for classes in your source code, the `@BASE_CLASS_LIST@` macro is expanded in each documentation frame. See also [Step 2: Name your documentation template file — page 295](#).

Macros that have a symbol scope can be set in any documentation template file.

(To see how to set macros in documentation template files, please refer to the example template file on [page 298](#))

Macro	Scope	Description
<code>@BASE_CLASS_LIST@</code>	class	Comma-separated list of the base classes of the class.
<code>@BASE_CLASS_FULL_LIST@</code>	class	Numbered list of the base classes of the class.
<code>@CLASSES_IN_FILE@</code>	file	List of all classes in the documentation file's corresponding source file(s). Note that this macro can only be set in the <code>Header.dtmpl</code> template file (located in <code>\$SNIFF_DIR/config/docu</code> ).
<code>@COMMENT_BEFORE_DEF@</code>	symbol	Extract the comments that precede the definition of the symbol in the source file. Note that the characters <code>/</code> and <code>*</code> are not extracted from the comment text.
<code>@COMMENT_AFTER_DEF@</code>	symbol	Extract the first comment lines that follow the definition of the symbol in the source file. Code lines are allowed between the definition of the symbol and the first comment lines. Note that the characters <code>/</code> and <code>*</code> are not extracted from the comment text. Note: We recommend not using both the <code>@COMMENT_BEFORE_DEF@</code> and <code>@COMMENT_AFTER_DEF@</code> macros for the same symbol, since comments following the symbol's definition may belong to the <u>next symbol's</u> definition. To avoid this ambiguity, always place comment lines either before or after a symbol's definition and use the appropriate macro to extract them.

Macro	Scope	Description
<code>@COMMENT_BEFORE_DEF_ FROM(X) TO(Y)@</code>	symbol	Extract all comments that precede the definition of the symbol and are between the user-defined tags <code>X</code> and <code>Y</code> . These tags must be alphanumeric and part of the comment that is being extracted. Note that the characters <code>/</code> and <code>*</code> are not extracted from the comment text.
<code>@COMMENT_AFTER_DEF_ FROM(X) TO(Y)@</code>	symbol	Extract all comments that follow the definition of the symbol and are between the user-defined tags <code>X</code> and <code>Y</code> . These tags must be alphanumeric and part of the comment that is being extracted. Note that the characters <code>/</code> and <code>*</code> are not extracted from the comment text. Note: See the note in the description of the <code>@COMMENT_AFTER_DEF@</code> macro.
<code>@DER_CLASS_LIST@</code>	class	Comma-separated list of the derived classes of the class.
<code>@DER_CLASS_FULL_LIST@</code>	class	Numbered list of the derived classes of the class.
<code>@ENUM_LIST@</code>	enumeration	Comma-separated list of the enumeration items of the enumeration.
<code>@ENUM_FULL_LIST@</code>	enumeration	Numbered list of the enumeration items of the enumeration.
<code>@FIRST_COMMENT@</code>	file	Extract the first comment lines in the source file. These comment lines must be the first non-empty lines in the file. Note that the characters <code>/</code> and <code>*</code> are not extracted from the comment text. Note that this macro can only be set in the <code>Header.dtmpl</code> template file (located in <code>\$SNIFF_DIR/config/docu</code> ).
<code>@INSTVAR_LIST@</code>	class	Comma-separated list of the instance variables in the class.

Macro	Scope	Description
@INSTVAR_FULL_LIST@	class	Numbered list of the instance variables in the class. The signatures of the variables are included in the list.
@METHOD_LIST@	class	Comma-separated list of the methods defined in the class.
@METHOD_FULL_LIST@	class	Numbered list of the methods defined in the class. The signatures of the methods are included in the list.
@PARAM_LIST@	method or function	List of parameter types of a function or method.
@PARAM_LIST_WITH_NAME@	method or function	List of parameter types and parameter names of a function or method.
@RET_VAL@	method or function	Return value of the function or method.
@SNIFF_DATE@	file	SNiFF+ version date.
@SNIFF_VERSION@	file	SNiFF+ version number.
@SYMBOL_MEMBEROF@	class	Class name of the symbol.
@SYMBOL_NAME@	symbol	Name of the symbol.
@SYMBOL_TYPE@	symbol	Type of the symbol (e.g., instance variable, global variable).
@TYPE_LIST@	class	Comma-separated list of typedefs in the class.
@TYPE_FULL_LIST@	class	Numbered list of typedefs in the class. The signatures of the typedefs are included in the list.

## Step 2: Name your documentation template file

There are two points to keep in mind when naming documentation template files:

- Template files must have the same names as the symbol types for which they are designed.
- The names of symbol types are hard-coded in SNIFF+.

For example, suppose you have created and want to name a template file for subroutines in FORTRAN. In SNIFF+, the symbol type for subroutines is `subroutine`, and you would therefore name the template file `subroutine.dtmpl` (all template files have the extension `dtmpl`). The following table lists the names that you must use for your C/C++, FORTRAN, IDL, Java and Ada documentation template files.

C/C++	FORTRAN	IDL	Java	Ada	Default template file/ symbol type
macro	—	macro	—	—	Macro
enum	—	enum	—	enum	Enumeration
typedef	—	—	—	typedef	TypeDef
variable	variable	—	—	object	GlobalVariable
function	subroutine	—	—	subprog	GlobalFunction
class	common_block	interface	class	package	Class
struct	block	struct	interface	record	Struct
inst_var	—	attribute	field	pkg object	InstanceVariable
methodDef	method	method	method	pkg sub- prog def	Method
union	union	union	—	—	Union

The last column in the table lists the names of the default template files used for the listed symbol types. SNIFF+ uses the default template files if it can't find your template files. See also [How SNIFF+ searches for documentation template files — page 297](#).

## Step 3: Determine where to store template files

You have two options for storing your documentation template files:

- If you want to use a template file for all of your SNIFF+ projects, store it in any directory in your file system.
- If you want to make a template file project-specific, store it in the project directory of a SNIFF+ project and then add it to the project. The template file is then valid for this project only.

For details about storing and specifying the location of template files, please refer to [Step 4: Specify the location of template files — page 296](#).

## Step 4: Specify the location of template files

### For template files stored in a template directory

1. Create a template directory and use the following table to name its language-specific sub-directories:

C/C+	FORTRAN	IDL	Java	Ada
Ansi_C/C++ (Create a directory called <code>Ansi_C</code> and, in it, a subdirectory called <code>C++</code> ).	FORTRAN_77	IDL	Java	Ada9X

If your language is not listed in the above table, launch SNIFF+. Then,

2. Load a project that contains source files in the language you want to document.
3. Create a language-specific subdirectory and name it according to the name used in the **Language** drop down in the Symbol Browser, Hierarchy Browser and in the Cross Referencer.
4. Store your template file in the appropriate language-specific subdirectory.
5. Enter the template directory's path in the **Template Directory** field in the **Documentation Editor** view of the Preferences dialog.

### For template files stored in a project directory

1. Store the template file in the `<project>/docu/dtmdl` directory. `<project>` is the name of project directory in the shared source working environment where the project's source files are stored.
2. Make sure that the **Document Template** file type has been loaded into the project. If it hasn't, load it into the project.
3. Add the documentation template file to the project (by choosing the **Add/Remove Files to/from <project>...** command from the **Project** menu of the Project Editor).

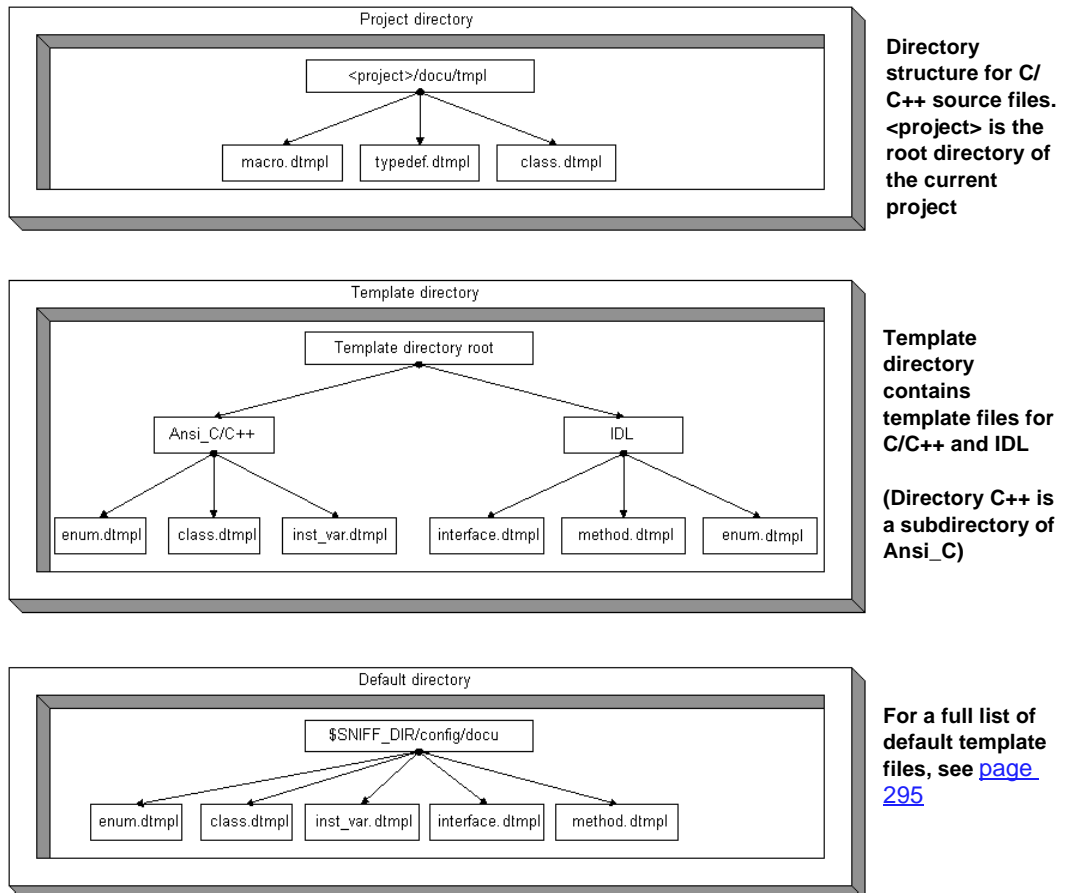


## How SNIFF+ searches for documentation template files

When you document a particular symbol in your source code, SNIFF+ searches for the appropriate documentation template file in three directories:

- If the **Document Template** file type is part of the current project, SNIFF+ first searches for the template file in the directory specified by the file type's **Directory** attribute (default value is `/docu/tmpl`).
- If the file isn't found, SNIFF+ continues searching for it in the template directory specified in the **Template Directory** field in the **Documentation Editor** view of the Preferences dialog.
- If the template file isn't found, SNIFF+ continues searching for it in the default directory (`$SNIFF_DIR/config/docu`). This directory contains default documentation template files. If SNIFF+ cannot find the appropriate template file for the symbol you are documenting, it will use the `Default.dtmpl` default template file.

To illustrate this procedure, let's use the following figure with three different scenarios:



## Scenario 1

You want to document a C++ source file that contains classes, enums and methods. SNIFF+ will use the following template files to create documentation frames:

- `class.dtpl` located in `<project>/docu/dtpl`
- `enum.dtpl` located in the `<template_directory>/Ansi_C/C++/dtpl`
- `Method.dtpl` located in `$SNIFF_DIR/config/docu`

## Scenario 2

You want to document an IDL file that contains interfaces and methods. SNIFF+ will use the following template files to create documentation frames:

- `interface.dtpl` located in `<template_directory>/IDL`
- `method.dtpl` located in `<template_directory>/IDL`

## Scenario 3

You want to document a FORTRAN file that contains common blocks and methods. SNIFF+ will use the following template files to create documentation frames:

- `Class.dtpl` located in `$SNIFF_DIR/config/docu`
- `Method.dtpl` located in `$SNIFF_DIR/config/docu`

## Sample documentation template file

	Description
<pre>&lt;SNIFFSYMBOL TYPE="Class" NAME=" " SSTATUS="new" DSTATUS="empty" ATTRIBS="true"&gt;</pre>	<p>All documentation template files must begin with this header information. This is used for internal management of documentation frames. Do not modify any of its attributes.</p>
<pre>&lt;SNIFFSECT SNAME="Symbol Name"&gt;</pre>	<p>The definition of a Section in the symbol's documentation frame is contained between the start tag (code to the left) and <code>&lt;/SNIFFSECT&gt;</code>. This particular Section is called <code>Symbol Name</code>. It tells SNIFF+ to insert the name of the symbol at the top of the documentation frame. Do not modify a Section's start tag.</p>

	Description
<pre> &lt;center&gt; &lt;h2&gt;  &lt;SNIFFINSERT CODE="@SYMBOL_NAME@"&gt;  &lt;/h2&gt; &lt;/center&gt; &lt;/SNIFFSECT&gt;  &lt;SNIFFSECT SNAME="Symbol Sign"&gt; &lt;center&gt; &lt;h4&gt; &lt;SNIFFINSERT CODE="@RET_VAL@"&gt; &lt;SNIFFINSERT CODE="@SYMBOL_MEMBEROF@"&gt;:: &lt;SNIFFINSERT CODE="@SYMBOL_NAME@"&gt;( &lt;SNIFFINSERT CODE="@PARAM_LIST@"&gt;) &lt;/h4&gt; &lt;/center&gt; &lt;/SNIFFSECT&gt;  &lt;SNIFFSECT SNAME="Parameters"&gt;  &lt;h4&gt; Parameters: &lt;em&gt;&lt;SNIFFINSERT CODE="@PARAM_LIST@"&gt;&lt;/em&gt; &lt;/h4&gt;  &lt;SNIFFSECTT&gt; &lt;pre&gt;XXXXX_Parameters &lt;/pre&gt; &lt;/SNIFFSECTT&gt; </pre>	<p>You can change or add HTML tags in any Sections as desired.</p> <p>SNIFF+ macro directive that tells SNIFF+ to expand the macro @SYMBOL_NAME@.</p> <p>The expanded macro is then inserted after the directive's end tag. Macros can be set anywhere after a template file's header information.</p> <p>This particular Section is called Symbol Sign. It tells SNIFF+ to insert the signature of the symbol right below its name.</p> <p>Do not modify any of the attributes in this Section's definition.</p> <p>HTML tags are used for layout.</p> <p>This Section is the first one in the documentation frame that contains editable text (see below).</p> <p>The Section Identifier for this Section is Parameters.</p> <p>Note that you can modify the name of this section (given by SNAME attribute). In general, you can modify the SNAME attribute for all sections except for Symbol Name and Symbol Sign.</p> <p>The definition of the Section Text field of a Section is contained between &lt;SNIFFSECTT&gt; and &lt;/SNIFFSECTT&gt;.</p> <p>Text contained in the Section Text field is editable in the Documentation Editor.</p>

Description	
<code>&lt;/SNIFFSECT&gt;</code> <code>&lt;/SNIFFSYMBOL&gt;</code>	All documentation template files must end with this end tag.

# Part XI

## Glossary and Index



# Glossary

---

**Absolute Project** is a project that is set up outside of a Working Environment. SNiFF+ therefore references these projects using an absolute path.

**Adaptor** is the specific implementation of a SNiFF+ interface. For example, the generic CMVC interface of SNiFF+ has adaptors for CMVC tools. The SNiFF+ debugger interface has adaptors for debuggers (gdb, dbx, etc.).

**Branches** occur in a version tree when you create new versions of a file from the middle instead of the end of the tree. Basically, SNiFF+ allows you to perform the same operations on branches that you can perform on the main trunk of a version tree.

**Browser** is a tool that is used for viewing (and not editing) data only. SNiFF+ offers several browsers like the Symbol Browser, the Class Browser and the Hierarchy Browser. The information displayed in browsers can be filtered in several ways.

**Build** is the process of creating the targets of a project. The build steps are usually described in makefiles which are executed by programs like Make. A build can involve translations of source files and the construction of binary files by compilers, linkers and other tools.

**Check-in** is the process of checking in a working file from a working environment, thereby creating a new version of the file in the Repository. A complete project can be also checked in. Typically, after a file has been checked in, locks made on the file are removed from the Repository. A file check-in can be associated with a change set. Note that SNiFF+ doesn't check in files itself. It delegates the operation to your underlying CMVC tool (by means of CMVC adaptors).

**Check-out** is the process of creating an editable working file in the working environment from a specific version of the file in the Repository. Depending on what actions are planned with the file, a lock of that file in the Repository is set. A check-out can be associated with a change set. SNiFF+ delegates the actual check out operation to your underlying CMVC tool (by means of CMVC adaptors).

**CMVC** is the abbreviation for configuration management and version control.

**Concurrent lock** is a lock that, unlike an exclusive lock, does not prevent others from locking the same version of a file. Versions that are concurrently locked must be merged back into the Repository.

**Configuration** is a coherent and consistent state of a system or project. A configuration has a name and refers to specific versions of files in the Repository. Typically, a configuration is a buildable state of a system.

**Configuration management** is the process of controlling and administrating the components of configurations. Configuration management includes the freezing (baselining) of configurations.

---

**Default Configuration** is the version of your software system that you work on. You can set your Default Configuration when you define your working environments. SNIFF+ uses your Default Configuration for the default value when you choose one of the various version control commands (e.g., checking out file versions, locking/unlocking file versions), and during the updating of Shared Source and Private Working Environments. Source files are made up-to-date with respect to the Default Configuration.

By default, the HEAD version of your software system is the Default Configuration for your Private Working Environment.

**Dependency** as used in Make is a relationship between two files that says that one file must be updated or rebuilt when the other one changes. In the Makefile, a dependency is a word listed after the colon ':' on the same line as a target. Source-level dependencies can be extracted from source code and are typically stored in dependency files. SNIFF+ generates dependency files that are included by Makefiles as part of its Make Support feature. Build-order dependencies must always be specified explicitly.

**Derived file** is a file that can be generated (derived) from another file. A typical example of a derived file is an object file that is generated from a source file after compilation.

**Documentation template** is a file that describes the structure and content of documentation frames. Each documented symbol type has its own documentation template. When documentation for a symbol is generated, SNIFF+ creates a documentation frame for the symbol. You can customize documentation templates.

**Documentation frames** are created when you tell SNIFF+ to document a symbol in your source code. Empty documentation frames represent the initial, undocumented state of a symbol's documentation. The structure and content of documentation frames are described by documentation templates.

**Editor** is a tool that is used for both viewing and changing data. SNIFF+ offers a number of editors (e.g., Source Editor and Project Editor). Tools that just show, but do not modify data, are called browsers.

**Exclusive lock** is set on a version in the Repository when a file is checked out for modification. Each version can have only one exclusive lock, thus preventing other developers from modifying the same version.

**File** is a component of a project. Each file is associated with a file type.

**File type** is associated with a file and determines several attributes of the file. Every file of a SNIFF+ project has a file type. SNIFF+ comes with a set of predefined file types, but there can be any number of file types in a project. Examples of file types are C++ implementation, C++ header, makefile, yacc source, shell script, etc. A file's file type determines how SNIFF+ treats the file and what operations may be performed on the file.

**Freezing a configuration** is the process of creating a "virtual snapshot" of the system (or, to be exact, of its source files) at special times during the software development process. You do this in SNIFF+ by associating the current state (configuration) of all project source files with a single symbolic name. The process of creating a single configuration and associating it with a symbolic name is called "freezing a configuration".



---

**HEAD** is the latest version on the trunk or branch of a file's version tree.

**History** reflects all the different versions of a file and is stored in the Repository file.

**Inheritance** is a directed relationship between two classes in which one class inherits the attributes of another classes. In single inheritance, a class inherits from only one class. Multiple inheritance means that a class inherits from several classes.

**INIT** is used by SNIFF+ as name to refer to the initial version of a file in the Repository. The **INIT** version is created when a file is checked into the Repository for the first time.

**Interfaces** are intermediaries between two components or tools or between the user and the machine. SNIFF+ uses interfaces to interact with the outside world and external tools. A SNIFF+ interface can have multiple specific implementations. called adaptors. For example, the generic CMVC interface of SNIFF+ has adaptors for CMVC tools. The SNIFF+ debugger interface has adaptors for debuggers like gdb and dbx.

**Lock** is a mechanism that controls access to versions of files in the Repository. SNIFF+ distinguishes between exclusive locks (only one developer can modify a specific version) and concurrent locks (multiple developers can simultaneously modify the same version).

**Locking** is the process of setting locks.

**Main branch** is the starting branch of a file's version tree. Unless otherwise specified by your default version control configuration, the main branch is the default branch for CMVC operations.

**Make** is the program that reads Makefiles and drives the building process. SNIFF+ integrates a wide range of different Make implementations.

**Makefile** is a text file read by Make programs that describes the building of targets. A Makefile contains source-level dependencies and build-order dependencies. As part of its Make Support feature, SNIFF+ generates Make Support Files that contain both kinds dependency information.

**Make macro** is a variable in a Makefile which can be assigned a string value. The value can be set in the Makefile itself, on the command line or by setting an environment variable with the same name. SNIFF+ uses Make macros to separate general, platform-specific, project-specific, and team-specific information. A coherent, generic and extendable set of Make macros is part of SNIFF+'s Make Support feature.

**Make Support File** is either generated out of the project's source code or supplied with the SNIFF+ package. Make Support Files contain the following information: generic Make rules, source-level dependencies, build-order dependencies, project-specific macros and platform-specific macros.

**Merge** is the process of combining the contents of two or more files into a single file. Typically, the files involved in a merge are versions of a single Repository file. A merge can be done automatically, but often requires manual intervention to resolve conflicts. SNIFF+'s Diff/Merge tool is used for merging files.

---

**Object file** is a derived file that is generated from source code after a build. SNIFF+ maintains a list of all object files for a project and generates a make support file containing this list.

**Shared Object Working Environment** is a working environment that, in contrast to a source working environment, stores only platform-specific files. Typically, a Shared Object Working Environment stores all object files of a project. A Shared Object Working Environment always accesses a corresponding Shared Source Working Environment. As a result, it must also have the same directory structure as the common (accessed) part of the corresponding Shared Source Working Environment.

**Obsolete file** is a file that is located in a project's directory but is not part of any SNIFF+ project. Obsolete files are generated by continuous development and changes to the project structure and should be deleted from time to time in order to keep a project's working environment clean. SNIFF+ Make Support feature offers mechanisms for finding and deleting obsolete files.

**Owner** is a developer that owns a file or a working environment.

**PDF** see **Project description file (PDF)**.

**Platform** is the combination of architecture, vendor and operating system. SNIFF+ executes on all supported platforms. Object working environments are platform-specific. The targets of SNIFF+ projects can be platform-specific. SNIFF+ executes the `sniff_arch` script to determine which platform its running on.

**Preferences** are the customizable attributes of SNIFF+. SNIFF+ supports user-level and site-level preferences. Most preferences can be edited with the Preferences dialog.

**Private Working Environment (PWE)** is a directory tree that contains the projects and working files of a single developer. A Private Working Environment is accessible and changeable by only one developer. All check-in and check-out operations work with files in the Private Working Environment. A Private Working Environment must have the same directory structure as the common (accessed) part of the corresponding Shared Source Working Environment.

**Projects** consists of files, attributes and subprojects. A project is the main organizational element in SNIFF+ and is described by a Project Description File (PDF). Project hierarchies can be built by adding one project to another project, thus creating a superproject-subproject relationship between the two projects.

**Project Description File (PDF)** is the file that describes a project's attributes, structure and contents. A PDF is a structured ASCII file that is created, saved and opened by SNIFF+. PDFs can also be generated with the `sniff_genproj` batch program.

**Project history** is the set of all configurations of a project.

**PWE** see **Private Working Environment (PWE)**.

**RCS** is a widely used revision control system that is licensed under the GNU public license. SNIFF+ integrates RCS as an underlying version control tool and also supplies it with the package.

---

**Repository** contains all Repository files of version-controlled projects. The Repository is typically directly accessed only by the managing CMVC tool and usually stores the different versions in an optimized delta format to save space.

**Repository file** is the file in the Repository that saves all the complete version tree of a file.

**Root directory** see **Working environment root directory**.

**SCCS** is the widely used source code control system that is supplied with most Unix implementations. SNiFF+ integrates SCCS as an underlying version control tool.

**Shared file** is a source file that is shared among the members of a development team. Shared files are located in a Shared Source Working Environment.

**Shared working environment (SWE)** is a directory tree that contains the files (source or object) shared in a team. Shared working environments are accessed (shared) among several developers in a team. There are two types of shared working environments: Shared Source and Shared Object.

**Shared Source Working Environment (SSWE)** is a shared working environment that contains source files only. Typically the platform-specific files are contained in a corresponding Shared Object Working Environment.

**SOWE** see **Shared Object Working Environment (SOWE)**.

**SSWE** see **Shared source working environment (SSWE)**.

**Symbol** is a named language construct in the source code.

**Symbol information** is extracted from the source files of a project. A project's symbolic information is stored in a Symbol Table, which is saved to disk and transparently managed by SNiFF+.

**Symbol Table** is the information base that contains information about the declaration, definition and use of named program elements such as classes, methods, variables and functions of a project. Each project has its own Symbol Table that is generated and maintained by the appropriate language parser. Symbol Tables are kept in memory and are persistently stored to disk.

**Symbolic link** is a symbolic reference to a file in the Unix file system. In contrast to hard links, symbolic links can span different file systems.

**Target** is the result of a build process. SNiFF+ allows multiple targets to be built in a single project.

**Team** is a group of software developers working together on a set of projects and sharing a set of common working environments.

**Update** is the process of checking out all new HEAD versions of projects in a working environment. Typically, updates are done automatically overnight and are followed by automatic builds of all Shared Object Working Environments and Private Working Environments.

**VCS** is the abbreviation for version control system.

---

**Version** is a particular revision and an element of the version tree of a file. A version is created by checking in a working file. The version of a file that you check out is your working file.

**Version control** is the process of managing and administering versions of files. The Project Editor in SNIFF+ is the main tool for version control.

**Version tree** is the hierarchical structure in which all versions of a file are organized. A version tree has one main trunk and can have several branches. The version tree is typically stored in a Repository file.

**Working file** is a file that has been checked out of the Repository in a working environment (usually in a Private Working Environment). A working file can be directly accessed. Each working file has a corresponding Repository file.

**Working Environment** is a directory tree that contains projects and working files. SNIFF+ distinguished between private and shared working environments. A shared working environment is accessed among several developers in a team and is overridden by their Private Working Environments. Shared working environments can be split into Shared Source and Shared Object Working Environments in order to separate platform-independent from platform-dependent files. Shared working environments can override other shared working environments, resulting in multiple levels of overriding working environments. The common part of overridden and overriding working environments must have the same directory structure.

**Working Environments Administrator** is the person who is responsible for the setup, administration and maintenance of working environments. An administrator is informed of the results of automatic updates and builds. Typically, shared working environments are administered by experienced developers with a thorough understanding of all projects that reside in a working environment.

**Working Environment root directory** is the root directory of a working environment. All root projects that are located in the working environment are subdirectories of the working environment root directory. If many projects need to be managed in a working environment, groups of projects can be located in subdirectories of the working environment root directory.

---

# Index

## A

- Absolute Projects 303
- Adaptor 303
- Adding new source files 129
- Architecture of SNIFF+ 10
- Archiver, and specifying for Make Support 101
- Attributes, modifying project 126

## B

- Bean 7
- Branches 138, 303
  - creating for RCS 143
  - creating for SCCS during check-out 141
  - definition of 138
- Browser 303
- Browsing documentation 289
- Build 303
- Building 73
  - help targets 191
  - see also* Make Support
  - specifying purify and quantify targets 100
  - targets 185
  - trouble shooting 186
  - using own Makefiles with SNIFF+ 105
- Building targets recursively 88

## C

- Change sets 138
  - definition of 138
  - showing differences 153
- Change sets, creating 143
- Check-in 303
- Checking in files 143
- Checking out a file 141
- Check-out 303
- ClearCase integration 267
  - changing config spec 271
  - changing View in SNIFF+ 272
  - ClearCase custom menu 271
  - introduction 267
  - notifying SNIFF+ of checkin 272

- overview 268
- setting Make command for use 274
- setting up SNIFF+ project for 268
- working with 271
- CMVC 303
- Codewright integration 251
  - features 251
  - menus 255
  - setting up 252
  - working with 254
- Comments, and extracting for source documentation 291
- Compilers, and specifying for Make Support 101
- Compiling, remote 207
- Concurrent lock 303
- Configuration 303
- Configuration management 303
- Configuration Manager
  - freezing (baselining) configurations 149
- Configurations 147
  - checking out 149
  - comparing 148
  - creating 149
  - definition of 138
  - deleting 150
  - freezing 149
  - freezing Default Configuration 150
  - freezing HEAD 149
  - looking at 147
  - renaming 150
- Configuring the Parser 178
  - examples 181
  - with a configuration file 178
- Creating configurations 149
- Creating workspace projects 162
- Cross Reference subsystems 219
  - and synchronizing (updating) Working Environments 229
  - corrupt X-Ref database 228
  - DB-driven cross referencing and Working Environments 226
  - extracting symbol information 221
  - how the X-Ref subsystems work 221
  - location of generated X-Ref information 225
  - maintenance of X-Ref databases 228
  - overview 220

- RAM-based vs DB-driven X-ref in Java 224
- RAM-based vs. DB-driven X-Ref in C/C++ 222
- selecting 229
- storage of generated files 225
- where locking information is maintained 228
- working environments and cross referencing 226
- X-Ref database access control 226
- Cross-platform development 193
  - introduction 193
  - limitations of 193
  - setting up 199
  - setting up projects for 204
  - Unix setup 199
  - Unix-Windows differences 196
  - Windows setup 201

## D

- Debugging 187
  - choosing an adaptor 187
  - remote 207
  - Source Editor in debugging mode 189
  - targets 187
  - useful debugging commands 188
- Default Configuration 304
- Default Configurations 140
  - definition of 140
  - specifying 154
- Default target 85
- Default working environment 45
  - specifying 52
- define directive, using 181
- Deleting a file version 144
- Dependency 304
- Derived file 304
- Diff/Merge tool
  - comparing file versions with 152
  - merging three-way differences 153
  - showing change set differences 153
- Disabling Make Support 106
- Documentation 279
  - adding file type to project 281
  - browsing in read-only mode 289
  - changing a symbol's documentation status 285
  - creating template files, *see* Documentation templates
  - exporting 289
  - extracting comments 291

- file type 281
- jumping between code and documentation 284
- looking at symbol's status 285
- managing together with source code 289
- updating, *see* Updating documentation
- writing, *see* Writing source documentation
- Documentation Editor
  - changing a symbol's documentation status 285
  - editing documentation 283
  - switching to editing mode 280
  - switching to read-only mode 289
- Documentation file type, adding to project 281
- Documentation frames 304
- Documentation frames, editing 283
- Documentation Synchronization Dialog 287
  - looking at documentation status 285
  - updating documentation 286
- Documentation template 304
- Documentation templates 290
  - creating 290
  - determining where to store 296
  - naming template files 295
  - sample template file 298
  - specifying location 296
  - using HTML tags in 291
  - using macros for customizing 291
  - using macros for extracting comments 291
- Documenting source code, *see* Documentation

## E

- Editor 304
- Editor integrations, Codewright 251
- Editor integrations, Emacs 233
- Editor integrations, MS Developer Studio 257
- Editor integrations, Vim 243
- Emacs integration 233
  - features 233
  - menus 239
  - setting up 235
  - working with 237
- Exclusive lock 304
- Exporting documentation 289

Exporting targets 85  
 Extracting comments for source  
   documentation 291

## F

File 304  
 File history, looking at 144  
 File type 304  
 File Types  
   adding 133  
   creating new 133  
 Freezing a configuration 149, 304

## G

General Makefile 80  
 Generated files  
   .sniffdir 23  
   Makefiles 23  
   PDF 23  
 Generating documentation files 282  
 Group Project Attributes dialog  
   using the dialog 131  
 GUI Builder 7

## H

HEAD 305  
 HEAD configuration of project, freezing 149  
 Help Targets 190  
   building 191  
   definition 190  
   useful help targets 190  
 Hierarchical project structures 24  
 History 305

## I

`ignore` directive, using 181  
 Include directives for preprocessing 177  
 Inheritance 305  
 INIT 305  
 Integration, VisaJ 7  
 Interfaces 305  
 is 307

## J

JAR 7

Java 7

## L

Language Makefiles 81  
   specifying 103  
 Linker, and specifying for Make Support 101  
 Linking targets, *see* Exporting targets  
 Lock 305  
 Locking 305  
 Locking a file 142  
 Locking information, displaying 146

## M

Main branch 305  
 Make 73, 305  
   *see also* Make Support  
 Make macro 305  
 Make Support 73  
   *also see* Debugging  
   *also see* Setting up Make Support  
   and recursive Make 88  
   and `symbolic_links` 76  
   and `VPATH` 75  
   basic concepts 13  
   building targets 185  
   disabling 106  
   exporting targets 85  
   General Makefile 80  
   help targets 191  
   Language Makefiles 81  
   Make Support Files 80  
   Platform Makefile 82  
   Project Makefiles 77  
   target types 85  
   technical overview 74  
   updating Make Support Files 80  
   using own Makefiles with SNIFF+ 105  
 Make Support File 305  
 Make Support Files 80  
   adding to project 130  
   updating 80  
 Makefile 305  
 Makefiles 77  
   General Makefile 80  
   Language Makefiles 81  
   Platform Makefile 82  
   Project Makefile 77

- SNiFF+ Makefiles structure 77
- using own with SNiFF+ 105
- Merge 305
- Merging 151
  - file versions 151
  - showing change set differences 153
  - three-way differences 153
- MFC 173
- Modifying projects 126
  - adding and removing files 129
  - adding Make Support Files 130
  - adding subprojects 127
  - also see* Group Project Attributes dialog
  - general procedures 126
  - removing subprojects 128
- MS Developer Studio integration 257
  - features 257
  - menus 261
  - setting up 258
  - working with 259
- O**
- Object file 306
- Obsolete file 306
- Owner 306
- P**
- Parser configuration file 178
  - example directives 178
  - modifiers 180
  - specifying the location of 178
- Parsing 173
  - browsing MFC code 173
  - browsing WinAPI code 173
- PDF 306
- Platform 306
- Platform Makefile 82
  - customizing 101
  - for your platform 102
- Preferences 306
- Preprocessing 173
  - configuring the Parser 178
  - full preprocessing 174
  - ignoring strings 181
  - overview 174
  - parser configuration file 178
  - selective `#ifdef` resolution 181
  - source code 174
- Preprocessing source code, *see* Preprocessing
- Preprocessing, and include directives 177
- Preprocessor macros 174
- Private Working Environment 31
  - initializing 56
  - manual setup 62
  - unattended updates of 167
  - updating in SNiFF+ 165
  - wizard setup 43
- Private Working Environment (PWE) 306
- Project attributes
  - modifying 126
- Project description file (PDF) 306
- Project Editor
  - adding and removing files 129
  - adding subprojects 127
  - displaying locking information 146
  - looking at file history 144
  - removing subprojects 127
- Project history 306
- Project setup overview 43
- Project Setup Wizard 43
- Project structures 24
- Projects 23, 306
  - adding and removing files 129
  - adding documentation file type 281
  - adding Make Support Files 130
  - adding subprojects 127
  - also see* Modifying projects and Make Support 73
  - and tracking dependencies 24
  - basic concepts 11
  - building targets 185
  - closing 124
  - default target 85
  - deleting 125
  - generated files 23
  - help targets 190
  - manual team setup 63
  - opening 122
  - organizing project structures 26
  - project directories 23
  - project structures 24
  - removing subprojects 128
  - saving 124



- setup overview 43
- setup procedures 44
- Setup Wizard 43
- types of 25
- using own Makefiles with SNiFF+ 105
- workspace projects for updating 162
- Purify and quantify Make support 100
- PWE 306
- PWE see Private Working Environment (PWE) 306

## R

- Rapid Reference Technology™ 219
- RCS, *see* Version Control
- Recursively building targets 88
- Remote compiling 207
  - invoking 216
  - overview 208
  - preparing 210
  - scenarios 209
  - setting up 212
- Remote debugging 207
  - invoking 216
  - overview 208
  - preparing 210
  - scenarios 209
  - setting up 212
- Replacing version control comments 144
- Repository 307
- Repository file 307
- Repository Working Environment 30
  - initializing 54
  - manual setup 60
  - wizard setup 43
- Resolving complex `#if` directives 182
  - `#ifdef` and `#if` directives 182
- Resolving preprocessor directives 181
  - `#ifdef` and `#if` directives 181
  - different class definitions 181
  - unbalanced braces 181
- Rollback, during unsuccessful builds 160
- Root directory 307
- Running an executable 187
- Running SNiFF+ without display 169

## S

- Samba 205
- SCCS, and creating new branch during check-out 141
- SCCS, and functionality differences in SNiFF+ 136
- script,update/synchronize Working Environments
- Setting up Make Support 88
  - also see* Make Support
  - and no `VPATH` support 89
  - exporting targets for linking 97
  - generating directories list for recursive Make 97
  - generating include directives 90
  - specifying external libraries for builds 96
  - specifying Make command 89
  - specifying platform-specific 101
  - specifying purify and quantify targets 100
  - specifying targets 96
- Shared file 307
- Shared Object Working Environment 31, 306
  - initializing 55
  - manual setup 61
  - unattended updates of 167
  - unsuccessful builds in 160
  - updating in SNiFF+ 164
  - wizard setup 43
- Shared Projects 25
  - manual team setup 63
  - suggestions after manual setup 68
- Shared Source Working Environment 30
  - manual setup 61
  - manually creating projects in 64
  - unattended updates of 167
  - updating in SNiFF+ 163
  - wizard setup 43
- Shared Source Working Environment (SSWE) 307
- Shared working environment (SWE) 307
- Showing and merging three-way differences 153
- Single-user development 39
- SNiFF+ architecture 10
- SNiFF+J 7

SNIFF\_BATCH 169  
 Source code documentation, *see* Documentation  
 Source Editor  
   in debugging mode 189  
 Source files, and adding to projects 129  
 SOWE 307  
 Specifying Default Configurations 154  
 SSWE 307  
 Subprojects 24  
   adding 127  
   removing 128  
 Symbol 307  
 Symbol information 307  
 Symbol Table 307  
 Symbolic information 16  
   basic concepts 16  
 Symbolic link 307  
 symbolic\_links , and sharing object files 76  
 Symbols 286  
   browsing documentation 289  
   changing documentation status 285  
   jumping between code and documentation 284  
   looking at documentation status 285  
   selecting symbol types to document 281  
   updating documentation 286

## T

Targets 307  
   building help targets 191  
   building projects 185  
   building purified and quantified 100  
   specifying for Make Support 96  
   types of 85  
 Team 307  
 Team development 30  
 Team Projects, *see* Shared Projects  
 Templates, for new projects 47  
   creating 47  
   editing 50  
   using with new projects 49  
 Three-way differences, showing and merging 153  
 Tool integration, basic concepts 18

## U

Unattended updates 167

  and project structure changes 167  
   and SNIFF\_BATCH 169  
   concept 159  
   cron script examples 168  
 undefine directive, using 182  
 Unlocking a file version 144  
 Update 307  
 updateWS.sh  
 Updating documentation 286  
   of a deleted symbol 287  
   of a moved symbol 288  
   of a new symbol 287  
   of a renamed symbol 287  
 Updating in SNIFF+ 163  
   Private Working Environment 165  
   Shared Object Working Environment 164  
   Shared Source Working Environment 163  
 Updating outside SNIFF+ 166  
   and SNIFF\_BATCH 169  
 Updating Working Environments 157  
   in SNIFF+, *see* Updating in SNIFF+  
   outside SNIFF+, *see* Updating outside SNIFF+  
   process model 161  
   reasons for updating 158  
   technical overview 158  
   unattended updates, *see* Unattended updates  
   unsuccessful builds 160  
   without Xserver host 169  
   Working Environments Administrator 159  
   Workspace projects 162  
 Using own Makefiles 105

## V

VCS 307  
 Version 308  
 Version Control 135  
   and branches 138  
   and Repository 136  
   and Working Environments 136  
   basic concepts 14  
   checking in files 143  
   checking out a file 141  
   configurations, *see* Configurations  
   creating change sets 143  
   creating new branch 143  
   creating new branch for SCCS 141  
   Default Configurations 140

- deleting a file version 144
  - differences between SCCS and RCS support 136
  - displaying locking information 146
  - file locking mechanisms 137
  - HEAD 137
  - INIT 137
  - initial check-in 54
  - locking a file 142
  - looking at file history 144
  - merging files, *see* Merging
  - of documentation 289
  - replacing comments 144
  - specifying Default Configurations 154
  - technical overview 136
  - unlocking a file version 144
  - version tree 137
  - Version control 308
  - Version tree 308
  - Vim integration 243
    - configuring 245
    - features 243
    - how it works 244
    - menus 247
    - setting up 244
    - working with 245
  - VisaJ 7
  - VPATH macro
    - and Make 75
    - and sharing source files 75
    - platforms supported on 89
  - VPATH macro 75
- ## W
- WinAPI 173
  - Working Environment 308
  - Working Environment Configuration
    - Directory 51
  - Working Environment root directory 308
  - Working Environments 29
    - and Default Configurations 32
    - and file permissions 58
    - and file sharing 33
    - and Make Support 13, 30, 73
    - and single users 39
    - and `symbolic_links` 76
    - and team support 30
    - and VPATH 75
    - basic concepts 12
    - cleaning up 190
    - examples of using 36
    - initializing 53
    - manual team setup 57
    - permissions for creating 59
    - see also* Updating Working Environments
    - specifying a Configuration Directory 51
    - specifying default 52
    - specifying Default Configurations 154
    - types of 30
    - when to use 29
    - workspace projects for updating 162
  - Working Environments Administrator 308
  - Working Environments tool
    - opening shared projects 123
  - Working file 308
  - Workspace projects 162
  - Writing source documentation 280
    - adding Documentation file type 281
    - Documentation Editor in editing mode 280
    - editing 283
    - generating 282
    - selecting symbol types to document 281



## **Colophon**

This manual was produced with FrameMaker.

We at TakeFive have tried to make the information contained in this manual as accurate as possible. We cannot, however, guarantee that it is error-free.

© 1992-1999 TakeFive Software GmbH.  
All rights reserved.



**sniff** \ˈsnɪf\ *vb* -ED/-ING/-S

[ME *sniffen*; prob. akin to ME *snivelen* to snivel]

*vt* (14c)

**3:** to recognize or detect by or as if by smelling  
<German shepherd dogs are parachuted in the  
Austrian Alps to *sniff* out survivors of avalanches  
— P.T.White>

*Webster's Unabridged Third New International Dictionary*

